2025 Semester 1
CITS4404: Artificial Intelligence and Adaptive Systems

Project Report

Word Count: 2991

Date - 9th May 2025

Group 19 - Trading Bots & Optimisation
Vinayak Gupta 24066272
Shu-Chi Liu 23831023
Zixiao Ma 24116864
Gerard Newman 22705329
Jordan Rigden 22496593
Lewei Xu 23709058

Youtube Video Link: https://youtu.be/3PJk-bYkG1I
MS Teams Video Link (has transcript, requires uwa login):
https://uniwa-my.sharepoint.com/:v:/g/personal/23709058_student_uwa_edu_au/EbHYw4H63vVMpHwpUAaKZpoBapY0ffbcMdsl8lXEchGk8Q?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D&e=WSyufr
Github Repository: https://github.com/LeweiXu/CITS4404-Project

Abstract

This project systematically explores and compares multiple trading strategies combined with various nature-inspired optimisation techniques, to evaluate their effectiveness in algorithmic trading. Six distinct trading algorithms—spanning moving average crossovers, MACD, Bollinger Bands, and other indicator-based approaches—were developed and benchmarked against six optimisation algorithms including Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Differential Evolution (DE), Grey Wolf Optimizer (GWO), and grid-based searches. Each combination was rigorously tested and optimised using historical BTC/USD data from 2014 to 2019, with out-of-sample performance evaluated on datasets from 2020 onwards. The analysis revealed significant variations in performance across different bot-optimiser combinations, highlighting notable trade-offs between exploration efficiency, hyperparameter sensitivity, and practical deployability. The findings provide valuable insights into which optimisation strategies enhance specific trading methodologies, reinforcing the importance of matching optimisation algorithms carefully to trading contexts. Overall, this comparative study demonstrates the necessity of balanced algorithmic selection, parameter tuning, and thorough validation when developing robust automated trading systems.

Introduction

This report outlines the development and implementation of various trading bots. These trading bots consist of a number of “pluggable” hyperparameters and contain an evaluation function that returns a “fitness” value based on these hyperparameters. The bot itself only generates signals based on a set of BTC/USDT OHLCV data and its internal hyperparameters (and hence we can calculate a fitness based on these signals). To optimise the hyperparameters, the bot is passed to an *optimisation algorithm* that utilizes the bot's internal fitness evaluation function and a set of training data to try and find the global optima in the hypothesis space. Due to the nature of the bots, we do not have a function that defines the hypothesis space, and in turn, the gradients (first and second derivatives) are not available. As a result, various gradient methods are not applicable. Instead, the bot can be described as a “black box” in which we input a set of hyperparameters and it returns a fitness value.

Assumptions

As the BTC/USDT trading in real life can be quite complicated, we will be using a simplified trading process for this project. In real life, the conversion rates between BTC/USDT varies between when you initialize the trade, to when the trade is actually finalized, and the transaction fee often varies depending on the platform. The exact conversion rate may also vary depending on the market orders for Bitcoin and the sell offers at the time. The historical BTC/USDT dataset contains 4 exchange rates: open, high, low & close. These exchange rates represent the first traded price of BTC, the highest traded price of BTC, the lowest traded price of BTC, and the last traded price of BTC for a specific period respectively.

To produce results that are somewhat “realistic” we will be using the opening or closing exchange rate to calculate the fitness of our bots. We will assume that all trades occur *instantaneously* and the conversion rate is exactly what was recorded in the historical datasets. We will not be using high/low exchange rates. Although these two exchange rates represent what our trades could have resulted in, the opening/closing exchange rates represent what our trades would result in on average.

Bot Design

A bot can be defined as an object (python class) containing 2 methods and 2 attributes: a generate_signals() method, a fitness() method, an array of hyperparameters, and an array of bounds for the hyperparameters.

generate_signals()

This object method generates buy/sell/hold signals of the same length of the data inputted. E.g. If a daily dataset were used for 2019, it would generate an array of length 365 containing either 1 (buy), -1 (sell) or 0 (hold). This is the heart of the bot, where a **trading strategy** is implemented, and it uses the array of hyperparameters stored within the bot object to tweak its behaviour. For example, in the simple strategy of moving average

crossover there are 2 hyperparameters stored within the bot: short_ma_window and long_ma_window. These 2 hyperparameters are stored as attributes in the bot object, and the optimisation algorithm will edit these hyperparameters to generate different signals.

fitness()

The fitness method will use the internal generate_signals() method to generate buy/sell/hold signals first and then calculate the fitness (aka the balance after trading over the period of the dataset). The optimisation algorithms will generally skip the generate_signals() method entirely and will only need to call the fitness() function to optimise the hyperparameters. The fitness() method starts off with $1000 USD and multiplies each transaction by 0.97 for the 3% transaction fee. As the fitness() function is the same for each bot, we can move this method to a higher-order object called *class bot* and have the individual bots extend this class, thereby inheriting the fitness() method. (See bot_base.py in the bots/ folder).

Hyperparameters array

An array of hyperparameters is stored internally within the bot, these hyperparameters are also the “pluggable” variables in the black box. The number of hyperparameters (or length of the array) is the number of dimensions of the hypothesis space. The more hyperparameters there are, the higher the dimensionality, and hence the larger the search space becomes. There are various hyperparameters, and each trading strategy will generally have a different set of hyperparameters.

Bounds array

The bounds array stores an array of lists or tuples that defines all the values the corresponding hyperparameter may take. The bounds array must be of the same length as the array of hyperparameters. Some hyperparameters are continuous, and some are discrete, to deal with this, continuous parameters are defined as a 2-tuple e.g. (0, 5.0), whilst discrete parameters are defined as a list of all possible values. An infinite hyperparameter can be defined as a tuple (-inf, inf). How this is handled is up to the optimizer implementation.

Language Representation

Given the above bot design, the hypothesis space can be defined as a length(bounds) dimensional space, and each element in the bounds array determines the nature of the dimension (continuous, discrete, finite, infinite). The model refers to the trading bot object itself: more specifically, the combination of its trading strategy (generate_signals()), the way its fitness is calculated (fitness()), and its hyperparameters. The model (bot object) is a black box that takes parameters as input and returns an evaluation of the hypothesis space by calling the fitness() function.

Our chosen language is limited to bots with a fixed number of parameters, two examples would be one with a pair of windows of variable length, this would be a bot with two

parameters, and another bot with windows of fixed length but each window shape is defined by parameters. Our language gives no way to combine these two types, which would be a bot with variable window length and variable window shape. Another example that would be out of our hypothesis space is bots that work on multiple time scales at the same time, say using hourly and daily data simultaneously.

Algorithms implemented

Particle Swarm Optimizer (PSO)

The pos and vel matrices store particle positions and velocities, respectively. Particle movement is updated in parallel using vectorized operations. The velocity update incorporates inertia, cognitive, and social terms. The inertia parameter preserves partial past velocity, using linear decay to transition from global exploration to local exploitation. The cognitive coefficient weights the pull toward a particle's personal best, while random vectors maintain exploration diversity. The social coefficient weights the global best position, with random vectors maintaining diversity, enabling swarm-wide solution sharing [1].

Input: $f(\mathbf{x})$ // Target fitness function to be optimized
$\mathbf{lb},\ \mathbf{ub}$ // Lower and upper bound vectors of search space
N // Swarm size (number of particles)
T // Maximum number of iterations
$w_{\text{init}},\ w_{\text{final}}$ // Initial and final inertia weight values
$c_1,\ c_2$ // Cognitive and social parameters
$v_{\max\%}$ // Maximum velocity ratio
early_stop // Early stopping threshold (iterations without improvement)

Initialize: $\mathbf{X}_i \sim U(\mathbf{lb}, \mathbf{ub})$ // Randomly initialize particle positions in search space
$\mathbf{V}_i \sim U(-\Delta, \Delta) \cdot 0.1,\ \Delta = \mathbf{ub} - \mathbf{lb}$ // Initialize particle velocities
$\mathbf{P}_i \leftarrow \mathbf{X}_i$ // Initialize personal best positions
$f_i \leftarrow f(\mathbf{X}_i)$ // Calculate fitness values for each particle
$\mathbf{G} \leftarrow \arg\min f_i$ // Initialize global best position

For $t = 1$ to T : // Main iteration loop
$w \leftarrow w_{\text{init}} - \frac{t}{T}(w_{\text{init}} - w_{\text{final}})$ // Linear decreasing inertia weight
For each $i = 1$ to N : // Loop through each particle
$r_1,\ r_2 \sim U(0,1)^d$ // Generate random coefficient vectors
$\mathbf{V}_i \leftarrow w\mathbf{V}_i + c_1 r_1 \odot (\mathbf{P}_i - \mathbf{X}i) + c_2 r_2 \odot (\mathbf{G} - \mathbf{X}i)$ // Update velocity
$\mathbf{V}i \leftarrow \text{clip}(\mathbf{V}i,\ -v_{\max},\ v_{\max}),\ v_{\max} = v_{\max\%} \cdot \Delta$ // Velocity clamping
$\mathbf{X}_i \leftarrow \text{clip}(\mathbf{X}_i + \mathbf{V}_i,\ \mathbf{lb},\ \mathbf{ub})$ // Update position with boundary handling
$f_i \leftarrow f(\mathbf{X}_i)$ // Evaluate fitness at new position
If $f_i < f(\mathbf{P}_i)$: // Update personal best if improved
$\mathbf{P}_i \leftarrow \mathbf{X}_i$ // Update personal best position
If $f_i < f(\mathbf{G})$: // Update global best if improved
$\mathbf{G} \leftarrow \mathbf{X}\_i$, reset no-improvement counter
Else: increment no-improvement counter
If no improvement count $\geq$ early_stop : break // Early stopping condition

Output: $\mathbf{G},\ f(\mathbf{G}),$ convergence history // Return best solution found

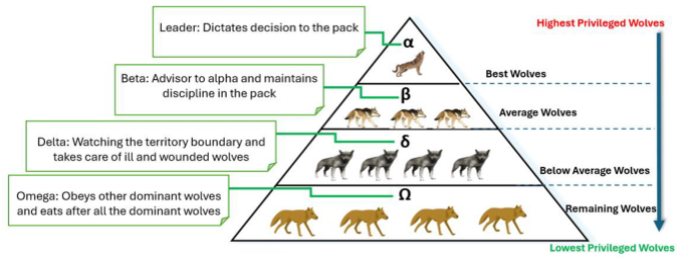
Bacterial Foraging Optimizer (BFO)

The general implementation of the optimiser can be followed by the pseudocode below:

```
Initialize colony with random bacteria positions within bounds
Set colony_best and best to very low fitness values

For each elimination-dispersal event:
    For each reproduction event:
        For each chemotactic step:
            Fitness = very low
            For each bacterium in the colony:
                Previous_fitness = fitness
                Set current bacterium and reset direction

                Evaluate fitness
                If fitness improved:
                    Update best
                If fit > Previous_fitness then run else tumble

                Move bacterium using chemotactic step (depends on run and direction and swarming)

            Store updated bacterium

        Update colony_best if best is the best so far

        Perform reproduction: keep top half and create mutated copies
    Perform elimination-dispersal: with some probability, replace each bacterium with a new random one
Evaluate final colony and return the best bacterium found
```

Each mechanism is modularised and accepts parameters (Eg. reproduction rate, dispersal probability), although fixed values are used by default to reduce dimensional complexity. Tuning these can impact performance - for instance, lower elimination-dispersal or higher reproduction may lead to premature convergence on suboptimal solutions [2], [3].

Grey Wolf Optimizer (GWO)



The Grey Wolf Optimizer (GWO) is a population-based metaheuristic inspired by the social hierarchy and cooperative hunting strategy of grey wolves. Wolves are ranked into alpha (leader), beta (advisor), delta (scout), and omega (follower), with optimization guided by the top three. During a hunt, real wolves track, encircle, and simultaneously approach prey from multiple sides, forcing it into a vulnerable position. The GWO pseudocode is shown to the right [4], [5].

```
Input: Training data (price series), bot structure with parameter bounds
Output: Optimized hyperparameters for trading strategy

1. Initialize pack of wolves (parameter sets) randomly within bounds
2. For each wolf:
    a. Set bot.hyperparams = wolf's 7 parameters
    b. Run generate_signals() on training data
    c. Compute fitness = strategy performance (e.g., total return)
3. Rank wolves by fitness and identify:
    α = best wolf
    β = second best
    δ = third best

4. For t = 1 to max_iterations:
    a = 2 * (1 - t / max_iterations)  // controls exploration vs exploitation

  For each wolf i:
    For each parameter d:
      Compute attraction forces A1, A2, A3 and coefficients C1, C2, C3
      Calculate:
        Dα = |C1 * α[d] - position_i[d]|
        Dβ = |C2 * β[d] - position_i[d]|
        Dδ = |C3 * δ[d] - position_i[d]|

        X1 = α[d] - A1 * Dα
        X2 = β[d] - A2 * Dβ
        X3 = δ[d] - A3 * Dδ

        New position = (X1 + X2 + X3) / 3

  Clamp new positions to valid bounds and snap discrete values

  Re-evaluate fitness for all wolves
  Update α, β, δ based on new fitness scores

5. After final iteration:
    Return α as best hyperparameter set
    Set bot.hyperparams = α
```

Ant Colony Optimizer (ACO)

ACO is a population-based metaheuristic inspired by ant foraging: artificial “ants” probabilistically construct solutions based on pheromone trails, which encode past solution quality. High-quality solutions deposit more pheromone, biasing future sampling toward promising regions [6], [7].

Pseudocode

```
Initialize τ[i][j] = 1 ∀ parameters i, values j
FOR t = 1 to T:
  // Evaporation
  τ[i][j] ← (1 - ρ) · τ[i][j]
  FOR ant = 1 to m:
    FOR each hyperparameter i:
      Sample value index j with P(j) ∝ τ[i][j]
      solution[ant][i] = grid[i][j]
    score[ant] = bot.fitness(solution[ant])
  Let b = argmax(score)
  FOR each parameter i:
    j* = index of solution[b][i]
    τ[i][j*] += score[b] / best_score_so_far
  Update global best if score[b] > best_score_so_far
RETURN global best solution
```

Hooke-Jeeves Direct Search Algorithm

A direct search method that requires 2n evaluation per step for an n-dimensional search space and searches by “sampling” in 2n directions [8]. Our implementation of Hooke-Jeeves involves selecting a random location in the search space as the starting point. If the parameter bounds is defined as a tuple (continuous variable), the optimizer discretizes the variable into 300 discrete values (i.e. converts a 2-tuple into a 300 length list) and then picks a random starting point from there. At each step, if the parameter is discrete, we simply take the next or previous value in the list, as step size is <= 1, there is no need to multiply by the step size. If the parameter is continuous, we multiply the direction by the step size as usual.

Bots implemented

Simple Bot

A simple trading bot mentioned in the project specifications mainly for demonstration purposes. The bot buys when the short-term simple moving average crosses above the long-term one, and sells when the reverse happens. The bot has 2 hyperparameters: the window size for the short term SMA and the long term SMA respectively.

Custom WMA Bot

Following a similar approach to the simple bot, Custom WMA tracks the crossover of two WMA filters each of length 50 where each value in the WMA windows is treated as a hyperparameter, resulting in a total of 100 hyperparameters.

Bollinger Bot

The Bollinger bot implements a classic volatility-based “bounce” strategy. For a chosen lookback window N and bandwidth multiplier K, the moving average and standard deviation of closing prices Pt over the prior N periods are computed as in the figure to the right. A buy signal (+1) is triggered when $P_t < L_t$, and a sell signal (−1) when $P_t > U_t$. These raw state signals are then compared to generate discrete trade entries and exits (+1 buy, −1 sell, 0 hold). The bot has two hyperparameters, the window length N and the Bandwidth multiplier K.

$$U_t = MA_t + K\sigma_t$$

$$L_t = MA_t - K\sigma_t$$

$$MA_t = \frac{1}{N}\sum_{i=0}^{N-1} P_{t-i}$$

$$\sigma_t = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(P_{t-i} - MA_t)^2}$$

GWO Bot

The trading bot uses the optimized parameters—SMA windows, thresholds, momentum period, etc.—to generate signals. A buy is triggered when the short SMA crosses above the medium, the medium is above the long SMA (both by a threshold), momentum is positive, and enough bars have passed since the last trade. A sell signal is triggered by the opposite conditions. This strategy, paired with GWO-tuned hyperparameters, ensures trades occur only during strong, confirmed uptrends or downtrends, maximizing precision and profitability.

MACD Bot (Moving Average Convergence Divergence)

This bot takes 3 parameters: fast_window, slow_window, signal_window and calculates signals based on an MACD line and a signal line. It buys when the signal line drops below the MACD line and sells when the reverse happens. The MACD line is calculated by subtracting the fast ema (calculated using the fast_window parameter) by the slow ema (calculated using the slow_window parameter). The signal line is calculated using an ema of the MACD line and the signal_window parameter.

WMA RSI Bot

The WMA RSI bot uses closing prices for calculations and combines WMA crossover and RSI to generate signals. A buy signal is generated when the Fast WMA > Slow WMA & RSI < buy threshold, a sell signal when the Fast WMA < Slow WMA & RSI > sell threshold, otherwise a neutral signal is generated. The RSI function calculates the relative momentum of an asset's price over a specified period, measuring its overbought or oversold condition on a scale from 0 to 100. It computes the average gains and losses, then calculates the Relative Strength (RS) and initial RSI value. Exponential smoothing is used to update these

averages, generating the full RSI time series. The bots pseudocode is shown in the figure below.

Input: prices: list of historical prices // closing prices over time
period: window size (e.g., 14) // RSI lookback period

Initialize: $\text{deltas} \leftarrow \text{diff}(\text{prices})$ // price changes between each time step
$\text{seed} \leftarrow \text{deltas}_{1\ldots\text{period}}$ // first (period) changes
$\text{gain} \leftarrow \frac{1}{\text{period}} \sum \max(\text{seed}_i, 0)$ // average of positive changes
$\text{loss} \leftarrow \frac{1}{\text{period}} \sum |\min(\text{seed}_i, 0)|$ // average of absolute negative changes
$RS \leftarrow \frac{\text{gain}}{\text{loss}}$ // relative strength
$\text{RSI}_{\text{period}} \leftarrow 100 - \frac{100}{1 + RS}$ // first RSI value

For $t = \text{period} + 1$ to T : // iterate over remaining prices
$\text{delta} \leftarrow \text{deltas}_t$ // current price change
$\text{gain} \leftarrow \begin{cases} \delta, & \text{if } \delta > 0 \\ 0, & \text{otherwise} \end{cases}$ // separate gain
$\text{loss} \leftarrow \begin{cases} -\delta, & \text{if } \delta < 0 \\ 0, & \text{otherwise} \end{cases}$ // separate loss
$\text{avg\_gain} \leftarrow \frac{\text{avg\_gain} \cdot (\text{period} - 1) + \text{gain}}{\text{period}}$ // update average gain
$\text{avg\_loss} \leftarrow \frac{\text{avg\_loss} \cdot (\text{period} - 1) + \text{loss}}{\text{period}}$ // update average loss
$RS \leftarrow \frac{\text{avg\_gain}}{\text{avg\_loss}}$ // recompute RS
$\text{RSI}_t \leftarrow 100 - \frac{100}{1 + RS}$ // compute RSI

Output: RSI series // full RSI values over time

Experiments & Evaluation

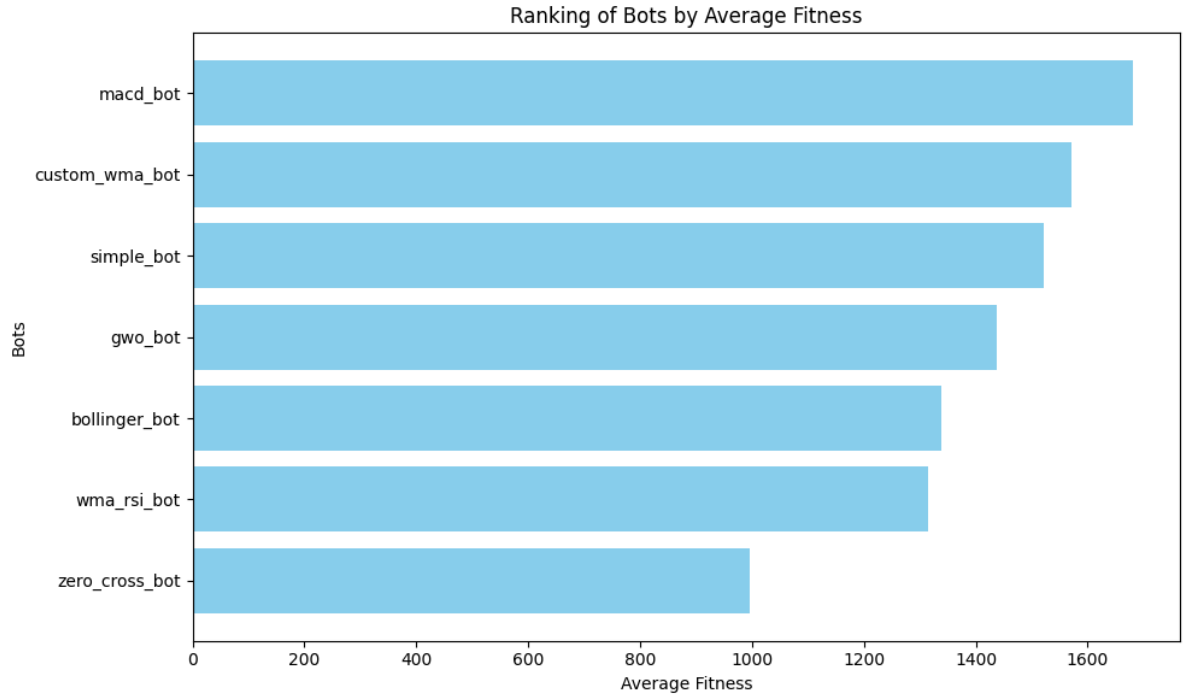
Initially, bots were created as bot-optimizer pairs, in which an optimizer was made specifically for a single bot implementation of n-dimensions. We realized for testing purposes this wouldnt work very well, so instead, we defined the “bounds” attribute for the bot and allowed optimizers to take the “bounds” attribute and optimize over any number of parameters. We had kept some aspects of the initial representation, where in each bot implementation file, we would choose a training dataset and an optimizer that we think would work best for that bot. Here are the results, seeded with np.random.seed(4404):

```
All trading bots start with $1000 cash and all transactions incur a 3% fee
Each bot begins trading on the first day of the year and the balance after 365 days of trading is the score
Leaderboard:
Rank  Bot                 2020 Daily ($)  2021 Daily ($)  2020 Hourly ($) 2021 Hourly ($) Average ($)
-------------------------------------------------------------------------------------------------------
1     gwo_bot             2961.92         552.95          770.54          730.63          1254.01
2     macd_bot            2216.05         893.91          272.28          296.42          919.66
3     wma_rsi_bot         1708.26         1000.00         285.12          203.58          799.24
4     bollinger_bot       2223.18         649.53          36.18           92.89           750.45
5     simple_bot          1860.60         937.24          49.73           64.97           728.14
6     zero_cross_bot      1000.00         1000.00         664.14          155.39          704.88
(trading-venv) lingwei@FlowX16:~/CITS4404/Project$
```

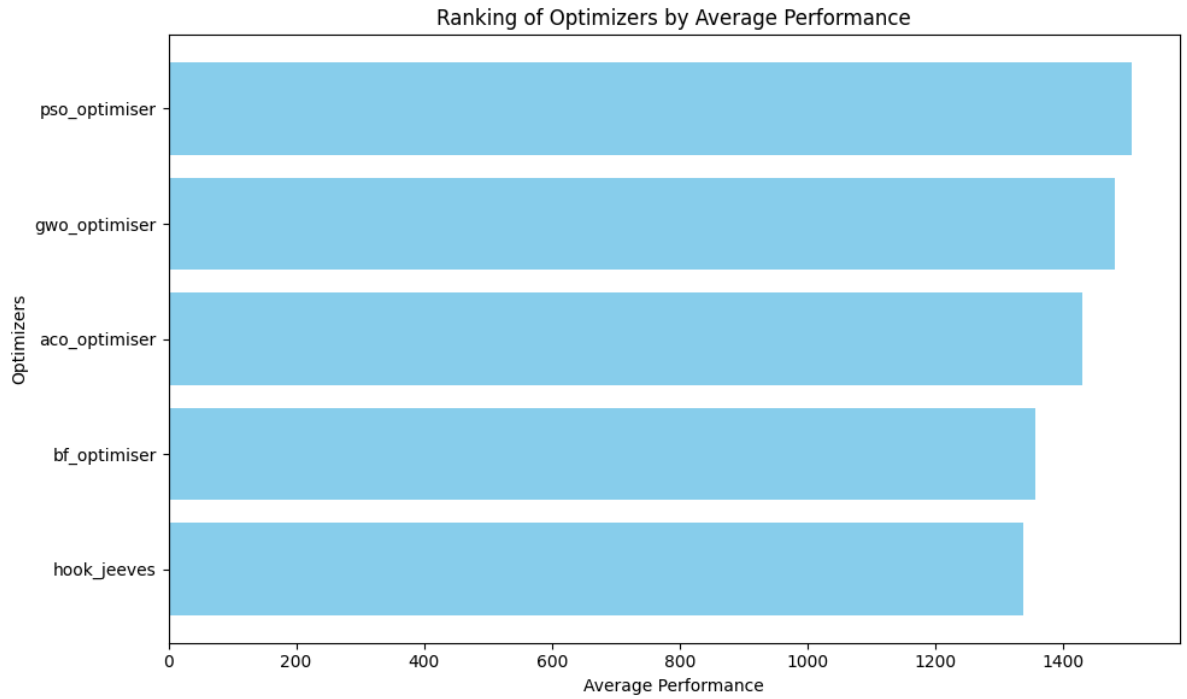
The training datasets consisted of daily data from before 2020. We can see that gwo and macd bots do very well, even in the hourly test that it wasn't specifically trained for.

Bot & Optimizer Performance

To enable a fairer comparison between bots and optimizers, we ran each bot against each optimizer (seeded with np.random.seed(4404)). The training dataset used was the entire daily dataset from 2014-2019, and the parameter bounds were as large as possible. These are the average results from testing the fitness from the 2020/2021 dataset.



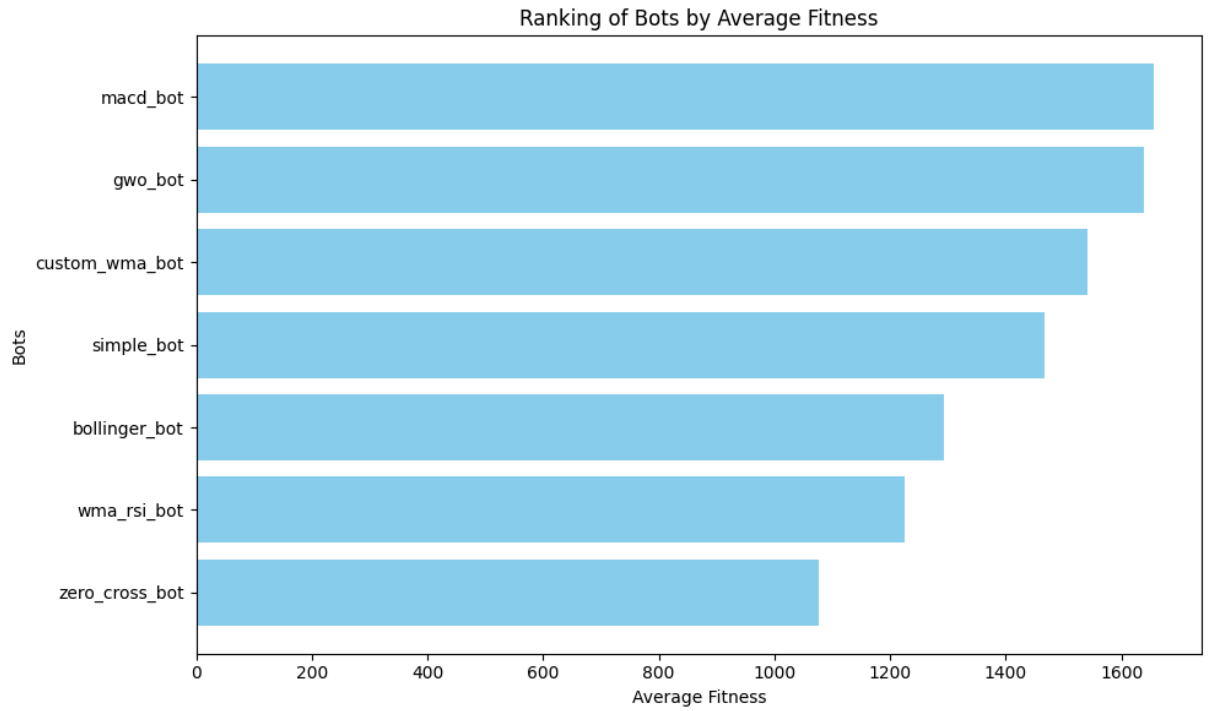
Here, we can see that macd_bot outperformed gwo_bot on average, whilst simple_bot (SMA crossover bot) does significantly better on average than the bollinger_bot and wma_rsi_bot. This may be due to simple_bot being a more stable/consistent bot, while bollinger_bot and wma_rsi_bot tend to have more variation/volatility in how they generate signals.



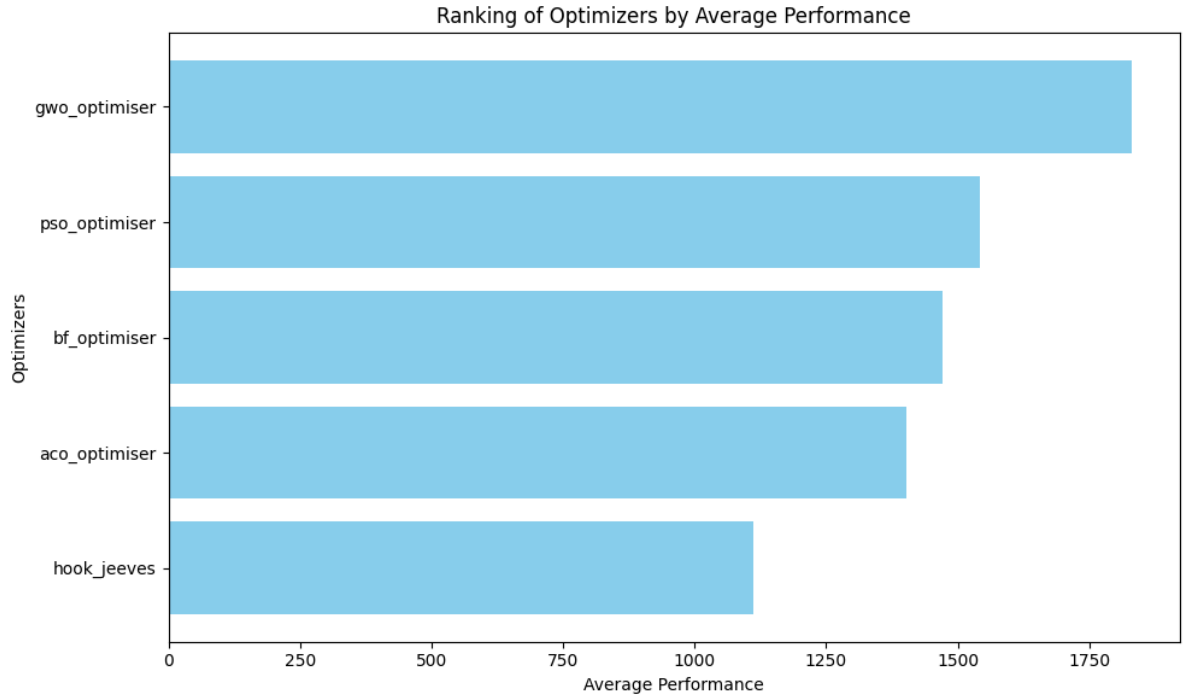
Here we can see that on average, all the optimizers are fairly similar in performance, with pso and gwo optimisers taking the lead. Hook-Jeeves (as expected) performs the worst. This may be attributed to the fact that it lacks an “exploration” phase and fails to find global optimas.

Realistic Bounds vs Large Search Spaces

There is a problem of “realistic” values for hyperparameters and values that simply wouldn’t make sense to use in a real life setting. For example, in the SMA crossover bot (simple_bot), window size that is too high (e.g. above 100) would cause the bot to have a slow reaction to market changes, reduced sensitivity and result in few trades overall. In the context of this project, where we are optimizing parameters in a controlled environment, optimizers will pick parameters that will maximise the fitness in a training dataset. In the video, we showed that ACO optimizer picked very large window sizes for the MACD bot due to the upward trend in BTC/USD prices. We are interested to see the effects of a “rocky” training dataset where there is no upward trend, and if we limit the parameter bounds to more “realistic” values.



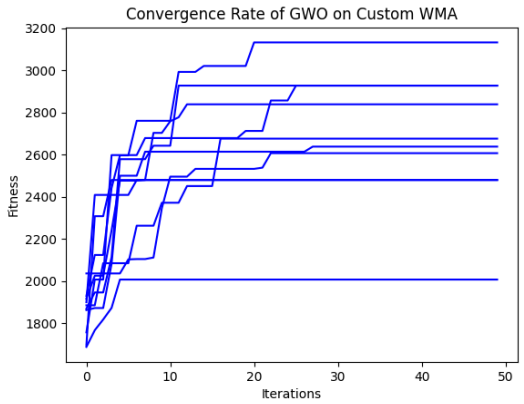
When we run the previous experiment of running every bot against every optimizer again, we find that the majority of bots seem to do better on average, although macd_bot did decrease in average fitness slightly.

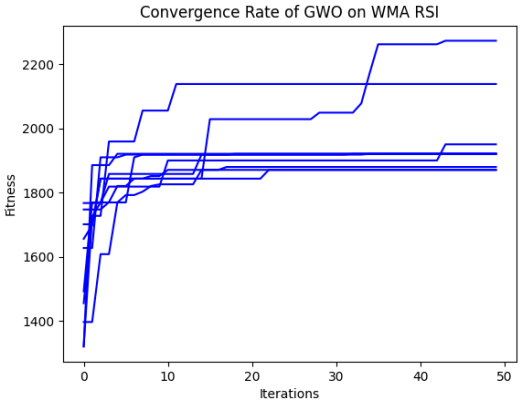


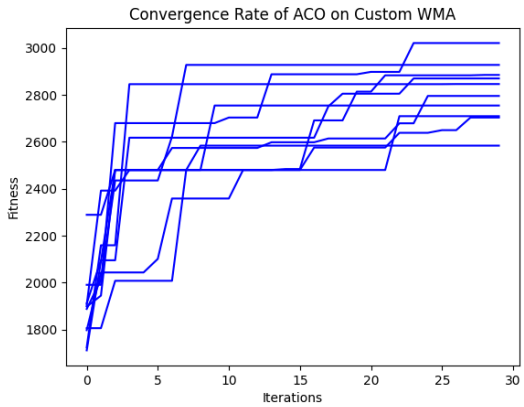
Similarly for the optimisers, an increase of approximately $100-200 average fitness is observed. Gwo_optimiser seems to have performed significantly better with the introduction of a more “rocky” dataset.

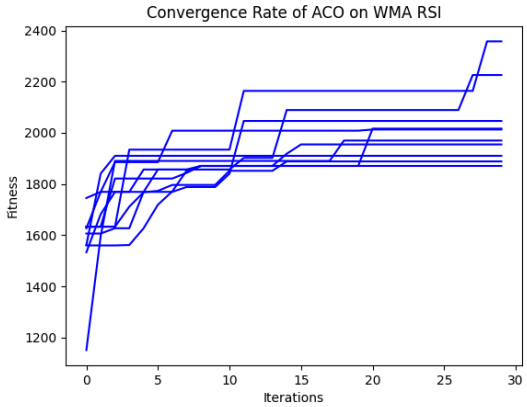
Convergence Experiments

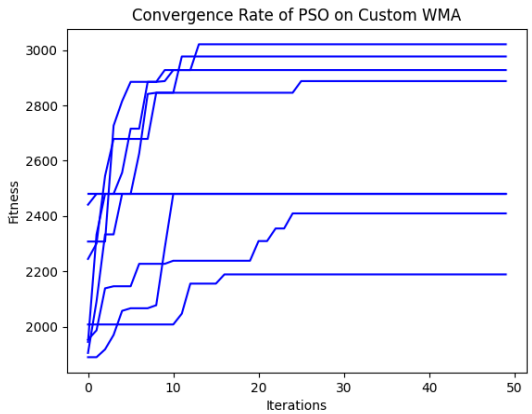
The convergence rate and range are two important figures of merit for optimisation algorithms. As each Algorithm involves some amount of randomness, the convergence point and rate can have significant variance between runs. The following presents two experiments that attempt to measure these properties. The first set of plots below track the highest found fitness by GWO, ACO and PSO against iteration number on the Custom WMA and WMA RSI bots. It can be seen that the convergence point is not consistent and rather falls in a range, we can also see that the algorithm has usually converged by around 50 iterations. From these plots it would seem that there is insufficient exploration for optimal search.

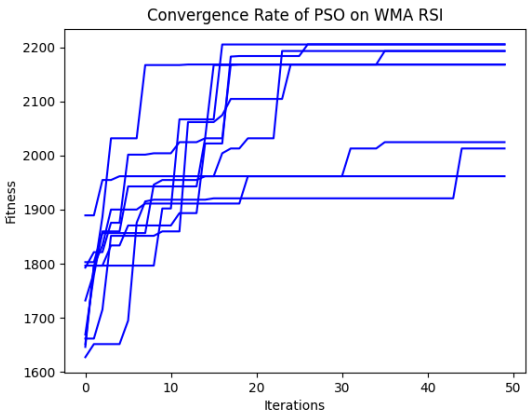




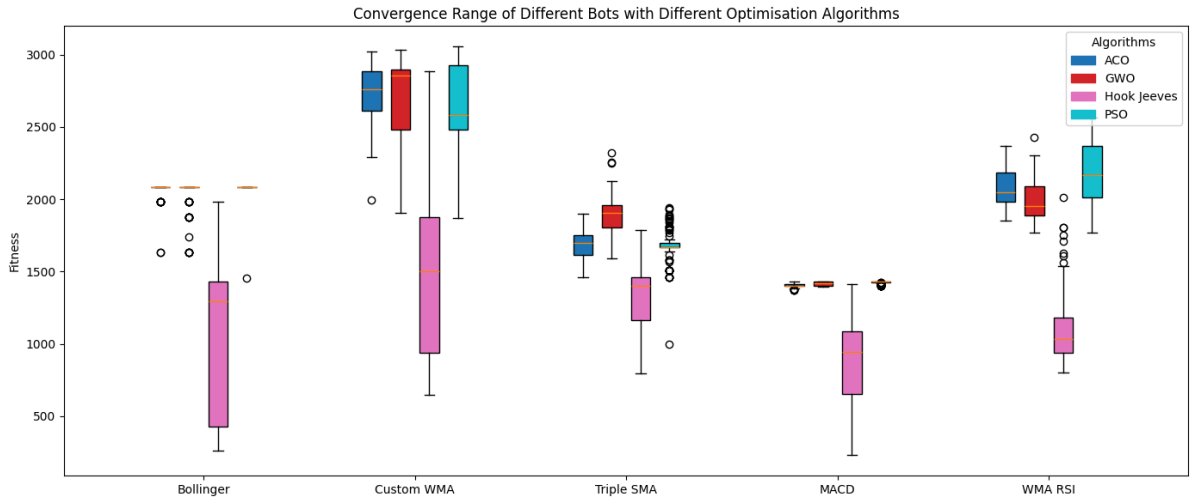








To get a better picture of the convergence range, 100 runs were completed for each bot against each algorithm, and the convergence ranges are plotted below in the figure below. We can see that Hook Jeeves consistently performs worse than the population based methods, this is easy to explain as each run can only reach one local minima, while the population methods can reach many, severely limiting the potential for Hook Jeeves. Amongst the population methods, there is little difference in the convergence point, although one thing to note is it appears that the more hyperparameters a bot has, the wider the range, this is likely explained by the increase in local minima.



Computational Complexity

These population-based methods support configurable parameters such as population size, iteration count, and optimiser-specific settings. Larger values improve exploration and increase the chance of finding the global optimum but also raise computational cost. Current settings prioritize efficiency, though further experiments: such as comparing the time complexity of Hooke-Jeeves and GWO across varying population sizes, could help balance search effectiveness with runtime constraints. Future work could focus on systematically identifying the point of diminishing returns in search performance relative to population size and iterations.

Conclusions

Through the development of trading bots and optimisers—including nature-inspired algorithms we gained practical insights into applying metaheuristic techniques to noisy, real-world datasets like Bitcoin prices. Each algorithm had different strengths, from GWO's strong balance between exploration and exploitation, to PSO's intuitive swarm behavior and fast convergence. Implementing and testing these strategies taught us how critical proper hyperparameter tuning is for profitability, and how simple strategies (like moving averages) can yield robust results when combined with intelligent search techniques.

While our bots performed well on historical data, real-world deployment would require handling transaction fees, slippage, live data ingestion, and market shocks—factors not fully captured in our backtests. Still, the framework we built is realistic and extendable. Future work could include adding reinforcement learning agents, multi-asset portfolios, live trading interfaces. We now better understand both the technical implementation and the limitations of AI-based trading systems in volatile environments.

References

[1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in Proceedings of the IEEE International Conference on Neural Networks (ICNN), Perth, Australia, 1995, pp. 1942–1948.

[2] K. M. Passino, "Biomimicry of bacterial foraging for distributed optimization and control," in IEEE Control Systems Magazine, vol. 22, no. 3, pp. 52-67, June 2002, doi: 10.1109/MCS.2002.1004010.

[3] E. S. Ali and S. M. Abd-Elazim, "Bacteria foraging optimization algorithm based load frequency controller for interconnected power system," *International Journal of Electrical Power & Energy Systems*, vol. 33, no. 3, pp. 633–638, Mar. 2011, doi: 10.1016/j.ijepes.2010.12.022.

[4] Image - Alqahtany, S.S., Shaikh, A. & Alqazzaz, A. Enhanced Grey Wolf Optimization (EGWO) and random forest based mechanism for intrusion detection in IoT networks. Sci Rep 15, 1916 (2025). https://doi.org/10.1038/s41598-024-81147-x

[5] Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey Wolf Optimizer. Advances in Engineering Software, 69, 46–61. https://doi.org/10.1016/j.advengsoft.2013.12.007

[6] Dorigo, M., & Di Caro, G. (1999). Ant Colony Optimization: A New Meta-Heuristic. IEEE Transactions on Evolutionary Computation.

[7] Stützle, T., & Hoos, H. H. (2000). MAX-MIN Ant System. Future Generation Computer Systems.

[8] R. Hooke and T. A. Jeeves, “`` Direct Search’’ Solution of Numerical and Statistical Problems,” *Journal of the ACM*, vol. 8, no. 2, pp. 212–229, Apr. 1961, doi: https://doi.org/10.1145/321062.321069.