THE UNIVERSITY OF WESTERN AUSTRALIA

Genetic Algorithms for Bitcoin Trading

CITS4404 - Assignment 2

Team Members:
Walsh, Maguire (22718133)
Alexander, Kane (22710428)
Fordham, Blake (22708346)
Maisey, Stuart (21835493)
Radich, Josh (22744833)

Word Count: 2969
June 5, 2023

Abstract

Various gene manipulation methods, including mutation, crossover, weight optimization, and a combination of these were implemented to produce both buy and sell expressions that dictated the trading bots Bitcoin trading actions. Although the leading genomes, especially those generated using only crossover, exhibited encouraging returns—yielding an optimal return of $191.71 AUD—a noticeable discrepancy between the training and testing sets suggested potential overfitting. This limitation could be attributed to the chosen data sets, indicating the need to improve model generalizability. A significant finding was the prevalence of specific indicators in high fitness expressions, suggesting their potential relevance for Bitcoin trading. Future work could focus on further refining these genetic manipulation strategies to enhance the trading bot's efficiency and generalizability.

Contents

1 Introduction

Our genetic algorithm was implemented by building a genome of buy/sell expressions with an optimization process to manipulate the internal sub-expressions for higher fitness performance.

The implementation of our genome contained a fitness score and a set of genes, each of which contained a set of buy and sell conjugates that represented a combination of various Technical Analysis indicators and their parameters. The details of these technical analysis indicators and their parameters will be discussed in more detail in the Methods section.

As mentioned, our algorithm also contained an optimization process, which tests each genome by using it in the trading process and observes its final money total to determine its fitness. The genes are then mutated and the fitness comparison process is started again until the specified number of generations is exhausted. The best-performing genome can then be retrieved and used to determine the final total and evaluate it with other strategies.

Furthermore, an analysis of the frequency distribution of technical analysis indicators across the top-performing genomes is provided. This analysis shows that some indicators are more commonly used than others, suggesting that reducing the search space by removing less frequently used indicators could be beneficial for time or complexity reasons.

2 Expression Framework

2.1 Literal Expressions

A literal is a comparison between two values or the negation of a comparison. Values can be technical indicators with their parameters, candle values, or constants (c). Literals abide by the grammar rules below:

$$\text{lit} \rightarrow \neg\text{lit}$$

$$\text{lit} \rightarrow \text{value} > \text{c} \times \text{value}$$

$$\text{value} \rightarrow \text{indicator (t, } P_i\text{)}$$

$$\text{value} \rightarrow \text{candle\_value(t)}$$

$$\text{value} \rightarrow \text{c}$$

Following these grammar rules in their entirety would allow some illogical literals to exist. These literals would increase the complexity and search space of the genetic algorithm, without providing meaningful generalizable rules. An example of such would be a literal where both values are constant:

$$\text{constant1} > \text{constant2} \times \text{value}$$

This literal would signal if and only if the value, for example, open price, was over a constant value. Although this literal does abide by the grammar rules, it would only likely work on the training set and wouldn't be generalizable to the future. Removing the possibility of illogical literals such as these also reduces the possibility of the genetic algorithm overfitting the training set. The approach undertaken to remove these literals was to introduce manual value pairs, in which a literal can only consist of certain predefined inputs.

2.2 Literal Expressions (Indicators)

Value Pair 1	Parameters	Initialization Range	Value Pair 2	Parameters	Initialization Range
Trend Indicators					
WMA	Window	[10..40]	**Candle (o)**		
EMA	Window	[10..40]	**Candle (o)**		
SMA	Window	[10..40]	**Candle (o)**		
ADX	Window	[10..30]	**Constant**		25
DPO	Window	[14..28]	**Constant**		0
Aroon Up	Window	[14..28]	**Aroon Down**	Window	[14..28]
Vortex Positive	Window	[10,30]	**Vortex Negative**	Window	[10,30]
MACD	Window F, S & Sign	[12, 26, 9] [24, 52, 18]	**MACD Signal**	Window F, S & Sign	[12, 26, 9] [24, 52, 18]
STC	Window F & S, Cycle, Smooth 1 & 2	[20..40] [40..60] [5..15] [2..5] [2..5]	**Constant**		50
KST	Roc 1,2,3,4	[10..20] [15..25] [20..30] [30..40]	**KST Signal**	Roc 1,2,3,4	[10, 20] [15, 25] [20, 30] [30, 40]
Mass Index	Window F & S	[7..15] [20..40]	**Constant**		30
PSAR	Step, Max Step	[0.01,0.03] [0.01,0.03]	**Candle (o)**		
MACD	Window F, S & Sign	[12, 26, 9] [24, 52, 18]	**MACD Signal**	Window F, S & Sign	[12, 26, 9] [24, 52, 18]
Volitility Indicators					
ATR	Window	[10..20]	**Candle (h)**		
Momentum Indicators					
RSI	Window	[14..28]	**Constant**		30 70
Stochastic Oscillator K	Window & Window Smooth	[10..20] [3..5]	**Constant**		20
Stochastic Oscillator D	Window & Window Smooth		**Constant**		80
KAMA	Window, Pow 1 & 2	[8..14] [2..4] [25..35]	**Candle (c)**		
PPO Line	Window F, S & Sign	[24..30] [10..14] [7..11]	**PPO Signal**	Window F, S & Sign	[24..30] [10..14] [7..11]
Volume Indicators					
MFI	Window	[10..20]	**Constant**		100

Table 1: Literal Expression Pairs

Table 1 defines the possible value pairs for each literal expression. If a value is defined as an indicator it can take a list of parameters each with a random initialization between set limits. Each literal expression contains the parameter c which is randomly initialized between [0.94,1.06]. The following are examples of literal expressions generated randomly selected from the pairs table:

$$\text{Literal Example 1: RSI}(23) > 0.98 * 30$$
$$\text{Literal Example 2: STC}(23,45,8,3,3) > 1.05 * 50$$

The expression is formatted in such a way that the indicator label is coupled with the parameters used to initialise it. The label is then used as a column identifier to directly look up the location at time 't' within a modified OHLCV data array.

2.3 DNF Expression Tree

The main component for all genetic algorithm strategies used in this report is the disjunctive-normal-form (DNF) expression tree. The DNF expression tree is used to hold the expressions and sub-expressions in classes appropriate for their classification. The top level of the DNF tree holds a list of `dnf` buy and sell expressions. The generator follows the grammar rules below:

$$\text{dnf} \rightarrow \text{conj} \vee \text{dnf}$$
$$\text{dnf} \rightarrow \text{conj}$$
$$\text{conj} \rightarrow \text{lit} \wedge \text{conj}$$
$$\text{conj} \rightarrow \text{lit}$$

The DNF expressions are built using conjunctions (CONJ) and literals (LIT). Conjunctions consist of one or more literals combined with the logical AND operator. DNF expressions consist of one or more conjunctions combined with the logical OR operator.

The tree structure and classes of the diagram can be seen below.

```
classes: DNF_Expression_Tree, DNF, CONJ, LIT, VALUE
    Common Functions:
        generate()
        mutate()
    Specific Functions:
        DNF_Expression_Tree: count_literals(), update_weights()
        DNF, CONJ, LIT: to_node(), replace()

class Variables:
    DNF_Expression_Tree:
        buy_expression: DNF
        sell_expression: DNF
    DNF:
        conjugates: List<CONJ>
    CONJ:
        literals: List<LIT>
    LIT:
        value: VALUE
        negation: Boolean
    VALUE:
        value1: Indicator | CandleValue | Number
        constant: Number
        value2: Indicator | CandleValue | Number
```

2.4 Expression Generator

When a complete buy-and-sell DNF expression is initialized, it begins at the top with the generation of DNF statements. In accordance with the grammar rules, a DNF expression can take one of two forms: it can either be a CONJ expression, or a CONJ expression logically OR'd with another DNF expression.

Upon descending the tree, the DNF expression unfolds into its constituent CONJ expressions. These CONJ expressions can either exist as a single LIT expression or as a LIT expression logically AND'd with another CONJ expression.

Further down the tree, the LIT expressions are generated. A LIT can either be a negated LIT or an expression in the form of $value > c \times value$.

To populate the LIT expression, we must initialize the values. A 'value' can be generated in one of three ways: it can be an indicator function with parameters, a candle value function of time 't', or it can simply be a constant 'c'.

The randomness incorporated in the generation process at each level ensures a variety of DNF expressions in the tree, which can include varying numbers of CONJ and LIT expressions, random negations of LIT expressions, and a wide range of LIT expressions that follow the value pairs as seen in Section 2.2.

Generating an expression makes use of the method:

1. `generate()`: Randomly generates a complete buy and sell expression.

The pseudocode for the expression generator can be found in Appendix 7.2.

2.5 Expression Mutation

Expression mutation occurs when any sub-expression inside an existing expression is regenerated into a completely new sub-expression.

To determine which sub-expression statement should be regenerated, a depth variable determines how far down an expression tree the regeneration occurs:

- At depth 1, the DNF_Expression_Tree class mutate method randomly chooses an existing DNF to regenerate, with the possibility of generating either a `conj` $\vee$ `dnf` or `conj` statement.
- At depth 2, the DNF class mutate method randomly chooses an existing CONJ to regenerate, with the possibility of generating either a `lit` $\wedge$ `conj` or `lit` statement.
- At depth 3, the CONJ class mutate method regenerates an existing LIT statement, with the possibility of generating either a $\neg$`lit` or `lit` statement.
- At depth 4, the LIT class mutate method regenerates an existing indicator of the same type, with new parameters, such that `indicator(t,`P_i`)` $\rightarrow$ `indicator(t,`P_{new}`)`.

We can visualize this by simulating mutating Expression1 in the example below:

Expression 1:

$$(\text{ATR}(19) > 1.00 \times h)$$
$$\vee$$
$$\neg(\text{Mass}(12, 35) > 1.00 \times 30)$$

At depth 1, mutating the Expression1 `dnf` sub-expression $(\text{ATR}(19) > 1.00 \times h)$ results in Expression2. Regenerating the `dnf` sub-expression as `dnf` $\rightarrow$ `conj` $\vee$ `dnf`.

Expression 2:

$$(\text{MACD}(12, 26, 9) > 1 \times \text{MACDSig}(12, 26, 9))$$
$$\vee$$
$$(\text{StochD}(20, 4) > 1 \times 80)$$
$$\vee$$
$$\neg(\text{Mass}(12, 35) > 1 \times 30)$$

At depth 2, mutating the Expression1 `conj` expression $(\text{ATR}(19) > 1.00 \times h)$ results in Expression3. Regenerating the `conj` expression as `conj` $\rightarrow$ `lit` $\wedge$ `conj`.

Expression 3:

$$(\text{MFI}(15) > 1.00 \times 100) \wedge \neg(\text{SMA}(32) > 1.00 \times o))$$
$$\vee$$
$$\neg(\text{Mass}(12, 35) > 1.00 \times 30)$$

At depth 3, mutating the Expression1 `lit` expression $(\text{ATR}(19) > 1.00 \times h)$ results in Expression 4. Regenerating the `lit` expression as `lit` $\rightarrow$ `value` $>$ `c` $\times$ `value`.

Expression 4:

$$(\text{PSAR}(0.02, 0.26) > 1.00 \times o)$$
$$\vee$$
$$\neg(\text{Mass}(12, 35) > 1.00 \times 30)$$

At depth 4, mutating the Expression1 `value` expression $\text{ATR}(19)$ produces $\text{ATR}(26)$, resulting in Expression 5. In accordance with Section 2.1, `value` is regenerated as the same indicator.

Expression 5:

$$(\text{ATR}(26) > 1.00 \times h)$$
$$\vee$$
$$\neg(\text{Mass}(12, 35) > 1.00 \times 30)$$

Performing a mutation makes use of the DNF tree method:

1. `mutate()`: regenerates a sub-expression given the depth level.

The pseudocode for the the expression mutation can be found in Appendix 7.3. It details how the mutation() method works for each class in the DNF tree.

2.6 Expression Crossover

Expression crossover exchanges sub-expressions between two full expressions. As expressions may be of different lengths, for two expressions to exchange any particular sub-expression, the index location for each sub-expression is recorded in the `path` list. Thus, while two sub-expressions may not occur in the same location in each expression, the sub-expressions can still be exchanged.

The `depth` variable determines how far down the tree the replacement occurs:

- At a depth of 1, an instance of `DNF` is swapped between two expressions. A `DNF` may be one or more `CONJ` expressions combined with a logical `OR`.
- At a depth of 2, an instance of `CONJ` is swapped between two expressions. A `CONJ` may be one or more `LIT` expressions combined with a logical `AND`.
- At a depth of 3, an instance of `LIT` is swapped between two expressions.

If we consider the conjugate expression:

$$Expression1: \quad (\texttt{CONJ\_1} \wedge \texttt{CONJ\_2}) \vee \texttt{CONJ\_3}$$
$$Expression2: \quad \texttt{CONJ\_4} \vee \texttt{CONJ\_5}$$

At a depth of 1, DNF($\texttt{CONJ\_1} \wedge \texttt{CONJ\_2}$) or DNF(`CONJ_3`) from Expression1 may be swapped with DNF(`CONJ_4`) or DNF(`CONJ_5`) from Expression2.

At a depth of 2, any `CONJ` instance from Expression1 may be swapped with any `CONJ` instance from Expression2.

At a depth of 3, any `LIT` contained within any `CONJ` statement may be swapped between the two expressions.

Performing sub-expression crossover makes use of the DNF tree methods:

1. `get_node()`: fetches the sub-expressions (node) to be exchanged and records how to get back to their position (path).
2. `replace()`: exchanges the sub-expressions by following the path back to the node to be changed.

The pseudocode for the expression crossover can be found in Appendix 7.4. It details how the get_node() and replace() methods works for each class in the DNF Tree. The calling function for the expression crossover is also listed.

2.7 Expression Weight Optimization

Weight Optimization refers to adjusting the constants c in each `lit` statement within an expression, such that it maximizes fitness for a particular time period.

Each `lit` statement becomes $value > c_i \times value \rightarrow value > c_{\text{new}} \times value$. In practice:

Before (Fitness: \$139.60):

Buy Conditions: $(\text{SMA}(26) > 1.00 \times o) \wedge (\text{PSAR}(0.01, 0.27) > 1.00 \times o)$

Sell Conditions: $(\text{MACD}(12,26,9) > 1.00 \times \text{MACDSig}(12,26,9))$

After (Fitness: \$153.44):

Buy Conditions: $(\text{SMA}(26) > 0.99 \times o) \wedge (\text{PSAR}(0.01, 0.27) > 1.06 \times o)$

Sell Conditions: $(\text{MACD}(12,26,9) > 0.90 \times \text{MACDSig}(12,26,9))$

The Weight Optimization algorithm takes in a single buy-and-sell expression pair and generates a population of possible weights for the expression. The algorithm calculates the fitness of the weights (the dollar amount at the last time interval) and then performs a tournament, bifurcating the population into elite and non-elite groups. A tournament is conducted among a subset of the genes, adding the winners and the elite back into the main population pool. The top-performing gene after n generations is returned, which is used to update each constant c in the buy-and-sell expression.

Performing weight optimization makes use of the DNF Tree methods:

1. `count_literals()`: A helper function that counts the amount of literals in an expression.
2. `update_weights(weights)`: Updates all the weights in order they are generated in the tree with the incoming weight array.

The pseudocode for the weight optimization code can be found in Appendix 7.5

3 Training Methodology

3.1 Fitness Selection

In order to assess the fitness of a a set of expression statements, the three expression manipulation methods described in Sections 2.5, 2.6, 2.7, were utilized as functions within a larger algorithm that accepts a large amount of expressions (labelled as genes) and manipulates them in order to increase their fitness for a given time period 't'.

Two strategies were derived in order to select for the highest fitness genes. The first strategy involves a tournament selection process, in which the random genes in the population are paired against each other, with the loser being evicted from the the gene pool. The second strategy involves a replicate and cull approach, in which elitism is employed to replicate the top 10% of the population while culling the bottom 10%.

In both processes, the population of genes is iterated through and one of the three expression manipulation methods is applied before the tournament selection or replicate and cull process is conducted. The results section will further discuss the the outcomes of employing the various expression manipulation strategies when applied to a wide set of genes.

The pseudocode for the Fitness Selection algorithm can be seen in Appendix 7.6.

3.2 Training and Testing

A compromise between the two selection methods was decided on, combining aspects of both. To compare our expression manipulation methods, a geneset of 5000 expressions (genes) was generated. From this geneset, all genes that did not trade, or did not produce a profit for the first 200 days were culled from the population. The remaining geneset consisted of 2351 genes.

The geneset was then manipulated in four distinct ways for 50 generations, that is, for each expression in the geneset, the gene was manipulated 50 times and if the manipulation produced a higher fitness the parent gene was replaced by the child.

The four genesets were generated: 50 generations of expression mutation, 50 generations of expression crossover, 50 generations of weight optimization, and then 50 generations that involved a combination of all three manipulation methods which involved iterating between mutation and crossover, and then optimization every 10 generations. In addition to this, the original geneset of 2351 unmanipulated genes was preserved as a comparison dataset.

The training set consisted of the first 200 rows of daily BTC data starting from 23/05/2021 with the following 520 rows being reserved as testing data. The relatively small amount of training data enabled faster training and more testing data to fully evaluate the generalizability of expressions.

4 Results

4.1 Expression Manipulation Results

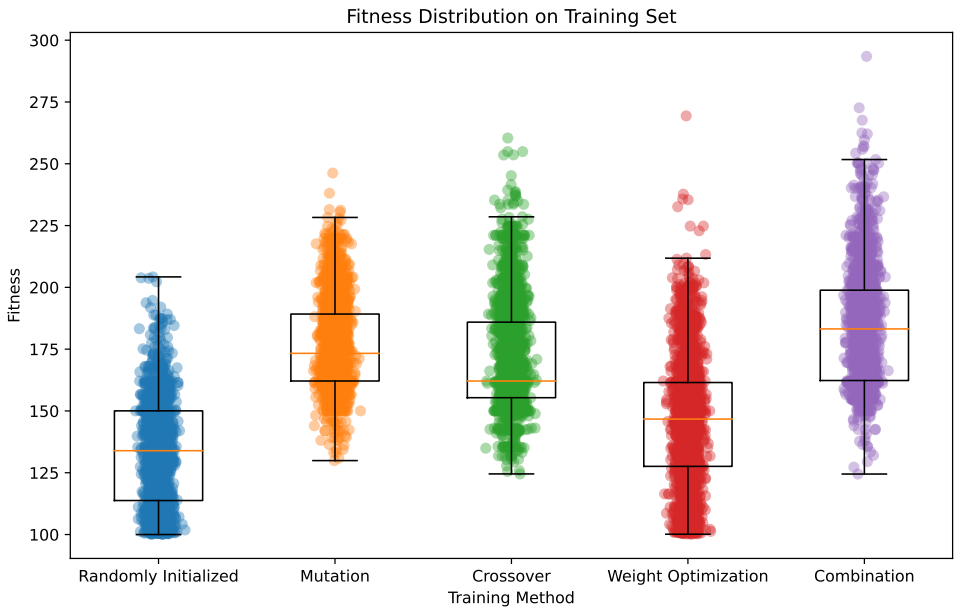

Figure 1: Fitness Distribution on Training Set

The training set (0 - 200 days) in Figure 1 demonstrates the improved averaged fitness for all expression manipulation methods.

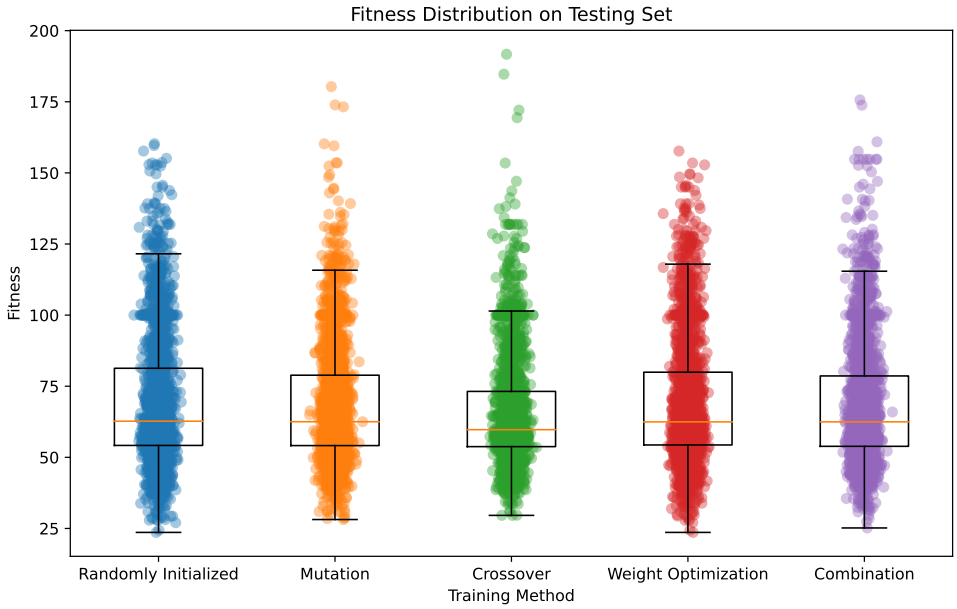

Figure 2: Fitness Distribution on Testing Set

The testing set (200 - 720 days) in Figure 2 demonstrates the prevalence of overfitting for the expression manipulation methods, but also distinct instances of high fitness expressions.

	Randomly Initialized	Mutation	Crossover	Weight Optimization	Combination
Profitable Genes (Test Set)	244	177	117	227	187
Average Test Fitness	69.41	68.34	66.38	69.06	68.41
Average Train Fitness	132.78	176.56	170.81	145.66	184.27

Table 2: Training and Testing Results

4.2 Leading Genomes

	Randomly Initialized	Mutation	Crossover	Weight Optimization	Combination
Best Test Fitness	160.27	180.32	191.71	157.67	175.67
Buy Condition	Stochastic K(16,5) >0.99 * 20	EMA(37) >1.04 * o	SMA(28) >1.02 * o	¬(RSI(27) >0.94 * 70) ∧ (MASS(13,31) >1.03 * 30)	PPO Line(28,12) > 1.01 * PPO Signal (28,12,11)
Sell Conditions	(MACD(24,52,18) >0.95 * MACD Signal(24,52,18)) ∨ (¬(MACD(12,26,9) >0.98 * MACD Signal (12,26,9)))	¬(WMA(29) >1.02 * o)	(PPO Line(24,14) > 0.99 * PPO Signal(24,14,7)) ∨ ¬(ADX(15) >1.02 * 25)	¬(STC (32,46,9,3,5) >1.03 * 50)	¬(MACD(12,26,9) > 0.97 * MACD Signal(12,26,9))

Table 3: Leading Genomes

Table 3 shows the leading genomes for each of the trained or randomly initialized genesets, with their fitness and BUY/SELL conditions.

4.3 Indicator Results

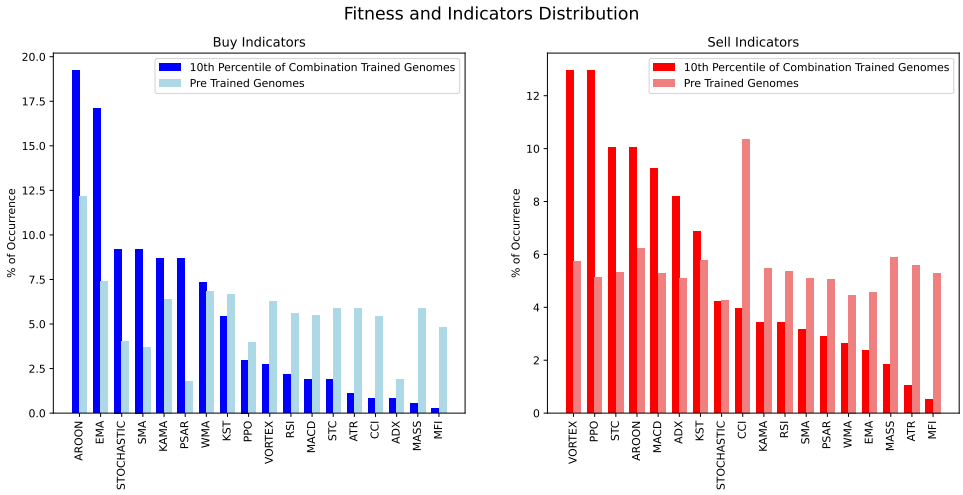

Figure 3: Fitness and Indicators Distribution

Figure 3 shows the relative distribution of indicators in leading expressions after undergoing Combination training. This Figure compares these leading indicators with their distribution before being trained but after the initial cull.

5 Discussion

When comparing Figures 1 and 2, the following trends can be observed: the performance on the training set is vastly better than on the testing set across all expression manipulation methods. The average fitnesses of the expression manipulations on the training set are significantly higher than the average fitness of the randomly initialised algorithm. Whereas the average fitness on the testing set for the expression manipulation methods barely

varied from the average fitness of the randomly initialised algorithm. This indicates that the genetic algorithms are overfitting to the training data and do not generalise well to external data.

One factor that could be contributing to the poorer performance on the testing set is that the training set has more opportunities for profitable trades. The training set is comprised of the first 200 days out of the total 720 days provided. This period of time exhibits significant growth in Bitcoin's value, whereas the last 520 days used for the testing set include significant drops and small rises in value. Thus it is not clear as to whether the algorithms are actually overfitting or not. This could be clarified by using a larger dataset for testing to see if the performance drop is still present.

As highlighted in Section 4.3, after undergoing Combination training some indicators were shown to be prevalent in high fitness expressions. These prominent indicators differed between the BUY and SELL DNF's with a different distribution before training. This implies that some indicators are more well-suited for the given training data, and could be extrapolated as being more relevant when making future trades. The prominent SELL indicators were Vortex and PPO while the BUY indicators were AROON and EMA.

Interestingly, although these indicators are shown to be correlated with higher fitness, the leading genome for Combination did not incorporate any of these indicators. This shows that a large search space of indicators can be fruitful. As shown in Figure 3, however, some indicators very rarely appear in any of the top 10% of genomes. As a result, if the search space needed to be reduced for time or complexity reasons, decisions to remove indicators could be based on this frequency distribution.

The limitations of our solution include the numerous unexplored parameters that have not yet received attention. Several parameters, such as the maximum length of DNF expressions (determining the number of statements that should be connected via OR or AND operators), the rate of weight mutation, and the selection of indicators, are among these considerations. Furthermore, the geneset used to test and train the data could be larger in size but was limited by compute performance and time.

6 Conclusion

This project set out to design an efficient Bitcoin trading bot using a genetic algorithm to optimize Technical Analysis Indicators. A number of different genetic manipulation methods were evaluated; mutation, crossover, and weight optimization which showed promising returns from leading genomes. The discrepancy in performance between the training and testing sets, however, suggests a degree of overfitting. This is a limitation of our current model and indicates that improvements could be made in terms of generalization to new data. This could be in part due to our choice of training and test data sets given more time we would have liked to explore this more.

A key finding from this project was the prevalence of certain indicators within higher fitness expressions, suggesting that these indicators may have particular relevance in the context of Bitcoin trading. Notably, the Vortex and PPO indicators were dominant in the SELL expressions, while the AROON and EMA indicators were dominant in the BUY expressions. These results could inform future research and development of trading bots.

Future work could further optimize the genetic algorithm parameters, explore the use of additional or different technical indicators, and implement strategies to reduce overfitting, thereby enhancing the bot's generalizability and overall performance.

7 Appendix

7.1 Code Repository

All code referenced in this project can be sourced at: https://github.com/kanealex/GA-BTC-Trading

7.2 Expression Generator Pseudocode

```
class DNF_Expression_Tree:
    function init():
        self.buy_expression, self.sell_expression = generate()
```

```
    function generate():
        buy_expression = new DNF.generate()
        sell_expression = new DNF.generate()
        return buy_expression, sell_expression

class DNF:
    function init():
        self.conjugates: List<CONJ> = generate()

    function generate():
        if random_condition():
            conjugates.extend([CONJ.generate() ∨ new DNF.generate()])
        else:
             conjugates.append(new CONJ.generate())
        return conjugates

class CONJ:
    function init():
        self.literals: List<LIT> = generate()

    function generate():
        if random_condition():
            literals.extend([LIT.generate() ∧ new CONJ.generate()])
        else:
            literals.append(new LIT.generate())
        return literals

class LIT:
    function init():
        self.value: VALUE = generate()

    function generate():
        if random_condition():
            return not (new LIT.generate())
        else:
            self.value = VALUE.generate()
        return value

class VALUE:
    self.value1, self.constant, self.value2
    function generate():
        value_types = INDICATOR_LIST(random_condition())
        values = []
        for value in value_types:
            if value == 'indicator':
                value.append(indicator(t, Pi))
            elif value == 'candle_value':
                value.append(candle_value(t))
            elif value == 'constant':
                value.append(constant())
        self.value1, self.constant, self.value2 = value
```

7.3 Expression Mutation Pseudocode

The following pseudocode displays the methods added to the original classes listed in the "DNF Expression Generator". The class variables are also listed for clarity.

```
class DNF:
    self.conjugates: List<CONJ>
    function mutate():
        depth = random_integer(1, 4)
        if random_condition():
            which_conj = random_integer(0, length_of(self.conjugates) - 1)
            decrement depth
            if depth equals 0:
                self.conjugates[which_conj] = new Conjugate()
            else:
                call mutate method of self.conjugates[which_conj] passing depth

class CONJ:
    self.literals: List<LIT>
    function mutate(depth):
        which_lit = random_integer(0, length_of(self.literals) - 1)
        decrement depth
        if depth equals 0:
            self.literals[which_lit] = new Literal()
        else:
            call mutate method of self.literals[which_lit] passing depth

class LIT:
    self.value: VALUE
    self.negated: Boolean
    function mutate(depth):
        decrement depth
        if depth equals 0:
            self.negated = random_choice([True, False])
            self.values = new VALUE()
        else:
            call mutate method of self.value

class VALUE:
    self.value1, self.constant, self.value2
    function mutate():
        value_types = INDICATOR_LIST(self.indicator_name)
        self.value1, self.constant, self.value2
        = values.append(VALUE.generate(value_types))
```

7.4 Expression Crossover Pseudocode

The calling function to perform a crossover between two expressions:

```
function crossover(gene1, gene2):
        depth = random_integer(1, 3)
        node1, path1 = gene1.get_node(depth)
        node2, path2 =  gene2.get_node(depth)
        gene1.replace(node2, path1)
        gene2.replace(node1, path2)
        return gene1, gene2
```

The DNF Expression Tree is modified to include the following methods:

```
class DNF:
    function get_node(depth):
        path = []
        depth = depth - 1
            node_index = random_integer(0, length of self.conjugates - 1)
            append node_index to path
            if depth equals 0:
                node = self.conjugates[node_index]
                return node, path
            else:
                node, path = self.conjugates[node_index].get_node(depth, path)
                return node, path

    function replace(node, path):
            node_index = path.pop(0)
            if length of path equals 0:
                self.conjugates[node_index] = node
            else:
                self.conjugates[node_index].replace(node, path)

class CONJ:
    function get_node(depth, path):
        depth = depth - 1
        node_index = random_integer(0, length of self.literals - 1)
        append node_index to path
        if depth equals 0:
            node = self.literals[node_index]
            return node, path
        else:
            node, path = self.literals[node_index].get_node(depth, path)

    function replace(node, path):
        node_index = path.pop(0)
        if length of path equals 0:
            self.literals[node_index] = node
        else:
            self.literals[node_index].replace(node, path)

class LIT:
    function get_node(depth, path):
        depth = depth - 1
        append 0 to path
        if depth equals 0:
            node = self.value
            return node, path

    function replace(node, path):
        if length of path equals 1:
            self.value = node
```

7.5 Weight Optimization Pseudocode

```
function Optimization(generations, buy_sell_expression)
    Initialize algorithm parameters
```

```
    best_weights = Train(generations, buy_sell_expression)
    Return best_weights

function Train(generations, buy_sell_expression)
    population = InitializePopulation(buy_sell_expression)
    Initialize mutation_rate, best_weights
    for generation in generations:
        population = TournamentSelection(population, mutation_rate)
        population = EvaluatePopulation(population, buy_sell_expression)
        Sort the population by fitness
        Update best_weights if a better solution is found
        Update mutation_rate if needed
    Return best_weights

function InitializePopulation(buy_sell_expression):
    Count number of weights in buy_sell_expression
    Initialize population with random weights for each individual in population
    Return population

function EvaluatePopulation(population, buy_sell_expression):
    for each individual in population
        individual.fitness = Fitness(individual.weights, buy_sell_expression)
    Return population

function Fitness(weights, buy_sell_expression):
    Calculate trading performance of buy_sell_expression with weights
    Return fitness

function TournamentSelection(population, mutation_rate)
    Select elite individuals
    Initialize parents list
    For each non-elite individual
        Perform a tournament and select the winner
        Mutate the winners genes by mutation_rate
        Add the mutated winner to the parents list
    Combine elite and parents to form a new population
    Return new population
```

7.6 Fitness Selection Pseudocode

```
function FitnessSelection(geneList, iterations, pattern, type)
    for i in range from 0 to iterations:
        Perform TournamentSelection() or ReproduceAndCull() depending on type
        which = pattern[i % length of pattern]
        for each gene in geneList:
            currentGene = gene
            Remove currentGene from geneList
            if which == 'ExpressionMutation':
                mutatedGene = ExpressionMutation(currentGene)
                append Gene with higher fitness (mutatedGene, currentGene)
                    to geneList
            elif which == 'ExpressionCrossover':
                append currentGene to crossOverList
                if length of crossOverList > 1
                    crossGene1, crossGene2, oldGene1, oldGene2
                        = ExpressionCrossover(crossOverList)
```

```
                    append Gene with higher fitness (crossGene1, oldGene1)
                        to geneList
                    append Gene with higher fitness (crossGene2, oldGene2)
                        to geneList
                    empty crossOverList
            elif which == 'WeightOptimization':
                optimizedGene = Optimization(currentGene)
                append Gene with higher fitness (optimizedGene, currentGene)
                    to geneList
    return geneList

function TournamentSelection(geneList)
    for gene in geneList
        randomly select firstGene and secondGene from geneList
        append Gene with higher fitness (firstGene, secondGene) to geneList
    return geneList

function ReproduceAndCull(geneList)
    sort geneList by fitness in descending order
    topGenes = top 10% of geneList
    geneList = top 90% of geneList
    extend geneList with topGenes
    return geneList
```