Masters Dissertation

Trading Strategy Refinement: Exploring Genetic Algorithm Normalization in a Multi-Threshold Environment within Directional change Paradigm

Sheehab Hossain Pranto

Registration: 2201113

Supervisor: Themistoklis Melissourgos

University of Essex

MSc Financial Technology (CS)

School of Computer Science & Electronic Engineering

Centre for Computational Finance and Economic Agents

August 29th, 2023

Declaration

This report is submitted as part requirement for the degree of MSc Financial Technology at the University of Essex. This paper is written by the author expect where specified.

The report may be freely copied and distributed provided the source is acknowledged.

Essex, August 29$^{\text{th}}$, 2023	Sheehab
Place, Date	Signature

Abstract

In academic work, the Directional Changes (DC) paradigm is defined as an event-based alternative to the traditional time-series approach with fixed intervals. The DC-based approach records price movements when specific events occur rather than in fixed time intervals. Significant price changes within this paradigm are identified using a threshold.

This paper builds upon the established Directional Changes (DC) paradigm, which employs multiple thresholds for strategy evaluation. While the foundational DC paradigm provides a robust framework for trading strategies, this research introduces a novel normalizated GA model designed to enhance fairness in decision-making. Refining the weighting mechanism within the DC paradigm with a genetic algorithm that aims to optimize trading strategies and offer a more equitable approach to financial forecasting. A methodological approach is employed to optimise the weights of the thresholds, specifically, a genetic algorithm.

The findings underscore the potential of this enhanced weighting model to improve upon traditional trading strategies within the DC framework such as buy-and-hold, MACD, and RSI. Furthermore, this strategy demonstrates superior performance compared to previously known single-threshold and multi-threshold strategies under standard efficiency metrics.

Keywords: Multi Thresholds, Weighting Decision, Trade Signal, Directional Changes

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Motivation

The rapid surge in the volume and velocity of financial data has ushered in both opportunities and challenges in the realm of computational finance. Initially perceived as a treasure trove for traders, this deluge of data soon manifested as a double-edged sword, inducing frenzied trading strategies that rendered market data chaotic and its predictability diminished. Traditional time-series analysis, which relies on fixed intervals for data sampling, often overlooks pivotal market events, leading to missed profitable trading opportunities. Such conventional methods, although foundational in financial forecasting, are susceptible to the pitfalls of unexpected market fluctuations outside their set intervals.

In response to these challenges, the Directional Changes (DC) paradigm emerged as a promising alternative. Pioneered by Guillaume et al.[2], DC transitions from the constraints of 'physical time scale' to an event-driven approach, focusing on significant market events rather than fixed time intervals. This innovative method captures the essence of market dynamics by observing data from an event-based perspective, ensuring no significant price movement goes unnoticed.

In recent years, the field of financial forecasting has experienced notable progress, particularly in relation to the concepts of return and risk. One significant contribution has been Markowitz's revolutionary modern portfolio theory[3], which has played a crucial role in directing research towards the creation of investment portfolios that generate profits for investors and effectively manage risk. Building on these foundations, a specific implementation of the DC-based trading paradigm has demonstrated its prowess in generating profitable and risk-averse trading strategies, notably outshining traditional technical analysis-based strategies. This success was achieved by harnessing the power of Genetic Algorithms (GA) to optimize the recommendations of multiple DC-based strategies [4].

However, the quest for refining financial forecasting methodologies remains relentless. This paper delves deeper into the DC paradigm and the results from [4], where the authors use multiple thresholds and a weighing mechanism on the strategies to capture a border spectrum of market events. This paper presents a new GA model on top of their current model in hopes of enhancing performance.

This paper is structured into five chapters. The subsequent chapter reviews prior research on DC, Evolutionary algorithms, and Directional change indicators, such as scaling laws, emphasizing their role in trading strategies. Chapter 3 details our workflow, including data preparation, strategy implementation, and the approach to handling multiple thresholds. Chapter 4 contrasts our results with those from [4]. The paper concludes with insights into the efficacy of merging multiple thresholds with genetic algorithms to devise trading strategies.

In essence, these DC indicators serve as the backbone for robust financial analysis, enabling traders to make informed decisions based on historical and current market data.

2 Background & Related Work

The literature on Directional Change (DC) and its applications in trading strategies is vast and multifaceted. This section provides a background on relevant topics and an overview of the seminal works and recent advancements in the field, focusing on the discoveries made in evolutionary algorithms trading strategy optimization and the findings about DC and their usage on trading strategies via evolutionary algorithms.

2.1 Directional Change (DC)

The Directional Change (DC) framework introduces an event-based approach to market price analysis, diverging from traditional time-series methods that sample at fixed intervals. Instead, the DC framework captures significant price shifts by recording data when a change surpasses a trader-determined threshold, denoted as $\theta > 0$.

This methodology segments price data into 'uptrend' and 'downtrend' intervals, each marked by a DC event, typically followed by an overshoot (OS) event. Such granularity offers traders a focused view of pivotal price movements, sidestepping minor fluctuations. However, a challenge arises in the retrospective confirmation of trend changes, which only materializes after prices deviate by the set threshold.

In the Directional Change (DC) framework, the threshold θ is user-defined, tailored to the specific asset under consideration. Figure 1 illustrates the sequence of DC and OS events with a threshold set at $\theta = 5\%$. Each point, such as point A, represents a time step (T_A) and its corresponding price (P_A).

Consider a financial asset priced at \$100 at time-step θ, which drops to \$96 by time-step T_{EXT_i}. As this price variation is less than the threshold θ, the interval from 0 to T_{EXT_i} isn't labelled as a DC event. However, between T_{EXT_i} and T_{DCC_i}, the price undergoes a notable 5% shift, marking this period as a DC event.

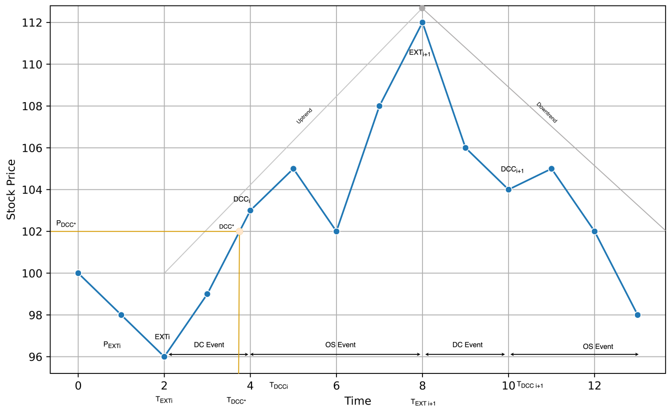

Figure 1: A TIME SERIES TO DC DIAGRAM WITH TWO DC CONFIRMATION POINTS AND OVERSHOOT EVENTS.

Within this context, two pivotal points emerge the extreme point (EXT_i) and the directional change confirmation point (DCC_i). For simplicity, we'll use discrete time steps (0, 1, 2, ...) representing specific moments when the asset's price is recorded, such as a stock's daily closing price. EXT_i denotes the starting boundary of the DC interval, while DCC_i signifies the earliest instance of a DC event. It is followed by an overshoot event from T_{DCC_i} to $T_{EXT_{i+1}}$.

To identify a subsequent DC event, the price must shift by the threshold θ in the reverse direction of the prior DC event, as depicted at point DCC_{i+1} in Fig. 1. Sometimes, the price fluctuation during a DC event can surpass the minimum change set by θ. To address this, we introduce the theoretical confirmation point, DCC^*, evident in Fig. 1 where a 4.8$ change (given $\theta = 5\%$) between points EXT_i and DCC^* confirms a DC event. T_{DCC^*} suggests that the time-step $T_{DCC^*} = T_{DCC_i}$ is the DC event's endpoint.

The algorithm 1 is designed to analyze the intrinsic time series of a market, specifically for a threshold value. Using the DC approach, the market's movements are categorized into alternating uptrends and downtrends.

A "downturn event" is the opposite. It signifies a decline in the market. A downturn event occurs when the difference between the current price $p(t)$ and the last high

Algorithm 1 DIRECTIONAL CHANGE EVENT GENERATION PSEUDOCODE (SOURCE: [1])

Require: Initialise variables (event is Upturn event, $p_h = p_l = p(t_0)$, $\Delta xdc(\text{Fixed}) \geq 0$, $t_{dc_0} = t_{dc_1} = t_{os_0} = t_{os_1} = t_0$)

if event is Upturn Event **then**
    if $p(t) \leq p_h \times (1 - \Delta xdc)$ **then**
        event $\leftarrow$ DownturnEvent
        $p_l \leftarrow p(t)$
        $t_{dc_1} \leftarrow t$ ▷ End time for a Downturn Event
        $t_{os_0} \leftarrow t + 1$ ▷ Start time for a Downward Overshoot Event
    else
        if $p_h < p(t)$ **then**
            $p_h \leftarrow p(t)$
            $t_{dc_0} \leftarrow t$ ▷ Start time for Downturn Event
            $t_{os_1} \leftarrow t - 1$ ▷ End time for an Upward Overshoot Event
        end if
    end if
else
    if $p(t) \leq p_l \times (1 + \Delta xdc)$ **then**
        event $\leftarrow$ UpturnEvent
        $p_h \leftarrow p(t)$
        $t_{dc_1} \leftarrow t$ ▷ End time for an Upturn Event
        $t_{os_0} \leftarrow t + 1$ ▷ Start time for an Upward Overshoot Event
    else
        if $p_l > p(t)$ **then**
            $p_l \leftarrow p(t)$
            $t_{dc_0} \leftarrow t$ ▷ Start time for Upturn Event
            $t_{os_1} \leftarrow t - 1$ ▷ End time for a Downward Overshoot Event
        end if
    end if
end if

price p_h falls below the threshold. This can be expressed as:

$$p_t > p_h * (1 - \Delta xdc)$$

Conversely, an "upturn event" is a specific type of market movement. It is identified when the difference between the current market price, denoted as $p(t)$, and the last recorded low price, p_l, exceeds a certain threshold. Mathematically, this is represented as:

$$p_t > p_l * (1 + \Delta xdc)$$

The DC framework's advent has illuminated previously hidden market regularities, offering traders a fresh perspective and unveiling innovative research avenues. Scholars have explored this domain using diverse techniques, from classical machine learning to deep neural networks.

2.2 Directional Change (DC) Indicators

In the realm of financial analysis, models often rely on specific parameters to interpret and predict market behaviour. These parameters, commonly referred to as indicators, play a pivotal role in shaping the strategies employed by traders and analysts. Here, we delve into the intricacies of the DC model's indicators, many of which have been previously introduced in scholarly works.

Indicators have long been a cornerstone in the literature of technical analysis. Their primary function is to unearth concealed patterns within financial datasets. By revealing these patterns, indicators empower decision-making tools like the Directional Change (DC) approach to optimize trading strategies, thereby enhancing profitability.

Below, we provide a comprehensive breakdown of the indicators that have been instrumental in formulating our strategies:

- **Number of DC events (NDC)**: This indicator tallies the cumulative count of DC events over a specified duration. It offers insights into the frequency of significant market movements, which can be indicative of market volatility or stability.

- **Number of Overshoot Events (NOS)**: NOS quantifies the total occurrences of Overshoot (OS) events within the analyzed dataset. An overshoot event typically signifies a price movement beyond what's expected or predicted, and tracking its frequency can be crucial for risk assessment.

- **Theoretical Confirmation Point (DCC*)**: DCC* denotes the earliest moment when a price alteration matches the value of θ. During an uptrend, the DCC* can be mathematically represented as:

$$P_{DCC^*} = P_{EXT_i} \times (1 + \theta) \tag{1}$$

  This equation underscores the significance of θ in determining the confirmation point during price ascensions.

- **Overshoot Values at Current Points (OSV_{CUR})**: OSV_{CUR} is an indicator designed to gauge the magnitude of an OS event. The magnitude essentially captures the extent to which the price has deviated from expectations. The formula to compute OSV_{CUR} is:

$$OSV_{CUR} = \frac{P_{CUR} - P_{DCC^*}}{\theta \times P_{DCC^*}} \tag{2}$$

  In this equation, P_{CUR} stands for the asset's prevailing price. By evaluating the overshoot values at current points, traders can get a sense of the market's overreactions, which can be pivotal for strategy adjustments.

2.3 Directional Change(DC) Scaling Laws

Scaling laws elucidate the functional relationship between interrelated physical quantities over significant intervals. Within the DC context, these laws are not mere mathematical constructs but vital tools that knit together the dynamics of price movements, their durations, and frequencies. The pioneering insights into this domain provided a panoramic view of the foreign exchange markets, where 13 currency pairs unfurled the mysteries of 17 scaling laws [5]. The subsequent unveiling of 12 additional scaling laws deepened our understanding, transforming the conventional time-series data into a more intuitive event-driven framework [6].

A cornerstone revelation from this body of work was the discernment that the average lifespan of an OS event eclipses that of a DC event by approximately a factor of two [6]. The research odyssey continued with the introduction of four and then an additional scaling law, broadening the horizons of DC from the confines of foreign exchange to the vast expanse of equity products [7], [8]. These groundbreaking insights are not just academic marvels; they are actively shaping the contours of trading strategies, signalling a new dawn in financial forecasting [9].

Furthermore, the DC paradigm has been enriched and made more accessible with the advent of indicators. These indicators serve as a compass for novices, enabling them to navigate the DC landscape and wield them with the finesse of tools in technical analysis. It's worth noting that the seminal work by Tsang et al. [8] blazed the trail by introducing four pivotal indicators. This was complemented by Tao's comprehensive lexicon, which cataloged a plethora of DC-centric indicators, further democratizing the knowledge and application of the DC paradigm [10].

2.4 Optimizations algorithms (Evolutionary Approach)

Evolutionary algorithms (EA) have gained prominence as a potent tool for addressing intricate financial optimization challenges. They emulate natural selection processes to pinpoint optimal solutions within expansive search spaces. A comprehensive study by Hu et al. [11] delved into 51 journal articles but couldn't conclusively determine the superior performance of any particular EA in diverse financial research domains. Concurrently, trading strategy optimization leveraging Genetic Algorithms (GA) has garnered attention [12], [13]. Notably, Salman et al. [9] ventured to optimize various strategies within the DC paradigm. It's noteworthy that fruitful outcomes have been realized not just via GA but also through genetic programming within the DC framework [14] and even outside it [15]. This paper proposes harnessing GA for optimizing trading recommendations based on diverse θ values derived from DC, aiming to discern if multiple threshold values can enhance trading decision quality by furnishing a richer data profile.

2.5 Trading Strategies

DC-based trading strategies, particularly those employing classification tasks, have demonstrated superiority over traditional technical analysis techniques [16]. Recent research has underscored the efficacy of the DC trend reversion projection algorithm, which outperformed a majority of both DC and non-DC benchmarks, such as the exponential moving average, in terms of both return and risk [17]. These studies underscore the adaptability and potential of the DC framework for refining trading strategies. A cursory review of the literature reveals a limited application of DC in crafting trading strategies. This research aims to bridge this gap by devising strategies based on multiple thresholds, enriched with GA optimization, to offer traders a more holistic decision-making tool.

3 Methodology

In the formulation of the experiment, a selection of 17 top-traded stocks from the NYSE was made. These include ALL, ASGN, CI, COP, EME, EVR, GILD, GPK, ISRG, MKL, MOH, PEG, PXD, QCOM, UBSI, VFC, XEL. To replicate the trading strategies delineated in the original paper, the timeframe spanning from 27 November 2009 to 27 November 2019 was employed. This period was chosen explicitly as pre-2020. During the pandemic, stock prices might not accurately represent standard market conditions. The trading indicators were derived from the daily adjusted closing price of each stock. The data acquisition was facilitated through a widely recognized stock data API library, yfinance [18].

3.1 Data Processing and Preparation

For the experimental procedures, Python code was utilized to cleanse, organize, and assess the data. Initially, the stock data was retrieved and archived in a CSV format using the yfinance library, retaining only the adjusted closing prices for each stock. Subsequently, the data was partitioned into three segments: the initial eight years served as the training set, while the concluding two years were designated for evaluating test outcomes. Such a division ensures that the weights derived from the training phase are not merely tailored to the training data but are also adept at handling previously unseen test data, thereby mitigating the risk of overfitting. Prior to the optimization process, the Genetic algorithm necessitates the configuration of several hyperparameters to achieve optimal results. To facilitate this, the training dataset was further bifurcated, allocating 80 percent for training and the residual 20 percent for validation.

For this experiment, there are few constraints when implementing the strategies. The constraints are there to reflect real-world scenarios. The first one is there can be only one buy or sell position at a time. If there is an open position, that position needs to be closed before a new position can be opened. The trades can only

Table 1: THRESHOLDS CHOSEN FOR THE STRATEGIES (%)

	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_9	θ_{10}
W	0.098	0.22	0.48	0.72	0.98	1.22	1.55	1.70	2	2.55

be long positions; no short selling is allowed. Each trade adds 0.025% transaction cost.

The experiment uses ten threshold values from table 1 to achieve multi-threshold results. In the previous study [9], only a single threshold of 2.5% was selected. For the purpose of this experiment, multiple thresholds are assigned to a weight, and each indicator generated will generate different results depending on the weight assigned to it. An optimization operation is performed using a genetic algorithm to optimise these results. The first and second strategy uses all ten thresholds, but the third strategy only uses the first five thresholds as it depends on multiple DC events occurring in a small threshold.

For each threshold, there will be a new column of indicators for each stock price data, and each strategy will have its own indicators list for each sock price. This is going to be generated and stored as a binary and an Excel file for visualization. To generate the binary file a built-in Python library called *Pickle* is used. Using the binary file, the indicator lists can easily be loaded into memory for further optimization.

3.2 Formulation of Individual Strategies

Advancing further necessitates addressing the trading strategies. Drawing from prior research, the DC scaling law indicator was identified. Given that the original study employed the DC scaling law in crafting its strategy, this paper also engaged with that approach. The subsequent strategy incorporates one of the DC indicators, leveraging the theoretical directional change confirmation point to gauge the magnitude of an OS event and subsequently trade based on this insight. The third strategy is anchored in the frequencies of overshoot events during an upturn. Given its reliance on recurrent upturn events, only smaller thresholds are selected for this strategy. The table 2 describes a brief overview of the logic followed by each strategy.

Table 2: STRATEGY DESCRIPTION

Strategy	Description
St1	Buying: Twice the duration of DC from P_{DCC} in DT Selling: Twice the duration of DC from P_{DCC} in UT
St2	Buying: $\lvert OSV_{CUR}\rvert \geq \lvert OSV_{Best}\rvert$ in DT Selling: $\lvert OSV_{CUR}\rvert \geq \lvert OSV_{Best}\rvert$ in UT
St3	Buying: 3rd consecutive OS in UT Selling: P_{DCC} in DT

3.2.1 Strategy 1 definition

The trading strategy presented herein is fundamentally anchored on the scaling laws delineated in the paper by [6]. This paper posits a theoretical proposition wherein each Directional Change (DC) event is approximately equivalent to twice the duration of an overshoot (OS) event. However, it is imperative to note, as highlighted in the aforementioned study, that an overshoot event does not invariably succeed every DC event. Such instances were meticulously accounted for during the strategy's implementation; any DC event not succeeded by an overshoot event was systematically excluded.

Mathematically, this relationship can be articulated as:

$$Duration_{DC} \approx 2 * Duration_{OS} \tag{3}$$

To operationalize the indicators requisite for this strategy, the price data for each stock is subjected to profiling via our proprietary DC event generation algorithm, as detailed in 1. For each predetermined threshold value, distinct indicators are synthesized for the respective stock price data.

Given that this strategy incorporates ten disparate threshold values, ten unique sets of DC events will be generated corresponding to each threshold. Subsequent to this profiling, the data is then processed through our strategy one algorithm, as elucidated in 2.

From the meticulously profiled data, specific events can be discerned for each threshold. These events serve as pivotal markers, furnishing the essential data for decision-making. Each DC event is characterized by an Extreme point and a

Directional Change confirmation point. The duration of a DC event can be computed as:

$$Duration_{DC} = T_{DCC_i} - T_{EXT_i} \tag{4}$$

Wherein T_{DCC_i} represents the confirmation point, and T_{EXT_i} denotes the preceding extreme price point. Upon ascertaining the duration of the DC event, the subsequent step involves the identification of the Over-Shoot event. It is salient to acknowledge that not every DC event is succeeded by an OS event. To ascertain this, a specific condition is instituted:

$$T_{EXT_{i+1}} - T_{DCC_i} > 0 \tag{5}$$

In the event this condition is unmet, the pseudocode disregards the event, advancing to the subsequent one. Alternatively, in instances resembling a DC event in downturn, the strategy entails awaiting twice the duration of that specific DC event prior to purchasing a position. Subsequently, the position is liquidated for a DC event that is in an upturn and is twice the duration of the previously discerned DC event. Given the ten thresholds, ten distinct profiled datasets are synthesized for each threshold. Employing the pseudocode for each threshold culminates in the generation of Buy, Sell, and Hold indicators for each. This process initially devised for a singular stock, is replicated across the 17 stocks under consideration.

Algorithm 2 STRATEGY ST1 DEFINITION

Require: Profiled data, DC duration, PDCC at DT, PDCC at UT

```
for each trend in Profiled data do
    if Overshoot event occurred in the trend then
        Determine the duration of the DC event.
        Confirm the DC at P_DCC at DT.
        Wait for a time period equivalent to 2× DC duration.
        Buy the stock.
        Wait for the P_DCC at UT.
        Wait for a time period equivalent to 2× DC duration.
        Sell the stock
    else
        Continue to the next trend
    end if
end for
```

Ensure: Buy and sell decisions based on the strategy.

3.2.2 Strategy 2 definition

In the development of this trading strategy, the initial phase of data profiling fundamentally mirrors the procedures delineated in strategy one. For each predetermined threshold, a distinct column will be dedicated to the trade signal. The efficacy of this strategy is contingent upon the magnitude of the Overshoot (OS) event and the extent to which the price deviates from anticipated values. The indicator, denoted as OSV_{CUR}, is meticulously crafted to quantify the magnitude of an OS event.

Leveraging equation 1, the theoretical price corresponding to the Directional Change (DC) event is synthesized. This value subsequently serves as a foundation to derive OSV_{CUR} using equation 2. These mathematical formulations culminate in the generation of a series of OSV_{CUR} values for each profiled dataset. The strategy then embarks on an evaluation to ascertain if OSV_{CUR} is greater than or equal to OSV_{Best}. Should this condition be satisfied, the strategy further probes whether the current DC event is indicative of a downturn. If this secondary condition aligns, a buy signal is generated. Conversely, in instances where the DC signal suggests an upturn, a sell signal is initiated.

In the quest to synthesize OSV_{Best}, an innovative approach was adopted, wherein the median value from each OSV_{CUR} quartile is harnessed. This methodology ensures that the context remains pertinent, facilitating a balanced comparison irrespective of market volatilities. Adhering to these stipulated conditions results in the generation of a dataset bearing a striking resemblance to the previous one. Within this dataset, each stock is represented by ten columns, each corresponding to a specific threshold. Analogously, trade signals for each stock are synthesized utilizing the pseudocode referenced in 3.

3.2.3 Strategy 3 definition

The third strategy, as delineated in this section, introduces a nuanced approach to trading, diverging from the methodologies employed in Strategies 1 and 2. This strategy hinges on the meticulous tracking of consecutive Overshoot (OS) events during an Upturn (UT) trend. The core premise of this strategy is rooted in the observation that a series of OS events in an UT, devoid of any OS in a Downturn (DT) prior to the third OS, can serve as a potent indicator to initiate a buy position.

Algorithm 3 STRATEGY ST2 DEFINITION

Require: DC profiled data, OSV_{CUR} values

Obtain distribution of all OSV_{CUR} values for DC profiled data.
Divide these values into quartiles.
Select one median OSV_{CUR} for each quartile.
Set this value as OSV_{Best}.
for each Trend in Profiled Data **do**
    if $|OSV_{CUR}| \geq |OSV_{Best}|$ **then**
        if Trend direction is DT **then**
            Buy the stock.
        else
            Wait for the opposite trend direction.
            Sell the stock.
        end if
    end if
end for

Ensure: Buy and sell decisions based on the strategy.

The efficacy of this strategy is intrinsically linked to the frequency of the DC events. Notably, a small threshold invariably leads to an increase in DC events. Conversely, an overly large threshold results in a marked reduction in the frequency of DC events, thereby diminishing the likelihood of fulfilling the conditions stipulated for Strategy 3. Consequently, in the context of Strategy 3, only the initial five thresholds are harnessed, as opposed to the complete set of ten thresholds.

To operationalize this strategy, an OS counter for UT, denoted as $OS_{count_{ut}}$, is initialized to zero. As the trading progresses, the strategy monitors the trend direction. Upon detecting an UT trend direction, the $OS_{count_{ut}}$ is incremented. If this counter reaches a value of three and no OS event was observed in the preceding DT trend, a buy position is initiated. Conversely, in the event of a DT trend direction, the $OS_{count_{ut}}$ is reset to zero. Otherwise, If a position was previously opened, it is promptly closed.

This strategy, as detailed in the pseudocode 4, offers a dynamic approach to trading, capitalizing on the patterns of consecutive OS events in UT trends. It is predicated on the hypothesis that a series of OS events in an UT, unaccompanied by any preceding OS in a DT, can be indicative of a favorable buying opportunity.

Algorithm 4 STRATEGY ST3 DEFINITION

Initialize OS_count_UT to 0
while Trading **do**
    if Trend direction is UT **then**
        $OS_{count_{ut}} = OS_{count_{ut}} + 1$
        if $OS_{count_{ut}} == 3$ and no OS in DT prior to 3rd OS **then**
            Buy a position.
        end if
    else if Trend direction is DT **then**
        $OS_{count_{ut}} = 0$
        if Position is open **then**
            Close the position.
        end if
    end if
end while
Ensure: Buy and sell decisions based on the strategy.

3.3 Decision Weighing Mechanism

The strategies under consideration are designed to generate trade signals for stock prices, leveraging the DC indicators as a foundation. Specifically, for strategies 1 and 2, ten distinct trade signals are anticipated for each stock. In contrast, strategy 3 yields five trade signals, corresponding to each threshold under evaluation. Each of these trade signals, specific to a stock, is subsequently mapped to a weight. The overarching decision-making process involves aggregating the weights corresponding to each threshold's decision. The decision with the highest cumulative weight is then selected as the final decision.

Mathematically, this process can be articulated as follows:

1. **Notation**:
    - d_j denotes a specific decision, which could be 'buy', 'sell', or 'hold'.
    - v_i represents the decision derived from the i^{th} threshold.
    - w_i signifies the weight associated with the i^{th} threshold's decision.

2. **Weight Aggregation for Decisions**: The cumulative weight corresponding to decision d_j is given by:

$$S_{d_j} = \sum_{i=1}^{n} w_i \cdot \delta(v_i, d_j)$$

Here, n stands for the total number of thresholds, while $\delta(v_i, d_j)$ is the Kronecker delta function, which assumes a value of 1 if $v_i = d_j$ and 0 otherwise.

Consequently, the decision corresponding to a specific profiled dataset is determined by $Max(N_{d_j})$.

3.4 Employing the Genetic Algorithm

In the context of this study, where the objective is to make a nuanced, weighted decision across multiple thresholds based on the DC indicator for each strategy, there is a compelling need for an optimization algorithm. Among the plethora of available optimizers, evolutionary methods stand out due to their practicality, bolstered by a substantial body of existing literature that attests to their efficacy. Specifically, for the purposes of this experiment, the Genetic Algorithm (GA) has been chosen as the optimizer of choice. The implementation leverages the PyGad library [19], a Python-based framework that not only facilitates the deployment of genetic algorithms but also offers compatibility with other renowned libraries. Moreover, the active maintenance of this library ensures that potential issues encountered during its utilization are promptly addressed.

3.4.1 Population Initialization

The genesis of the algorithm involves the generation of an initial population comprising 100 randomly created chromosomes. Each chromosome is instantiated with ten floating-point values, ranging between 0 and 1. Subsequent to their creation, these values undergo normalization to ensure their cumulative sum equals 1. It's worth noting that each chromosome symbolizes a weight corresponding to each threshold. In the specific case of Strategy 3, the model employs chromosomes of a reduced length, five as opposed to ten, whilst still adhering to the normalization constraint.

θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	θ_7	θ_8	θ_9	θ_{10}
0.12	0.08	0.11	0.09	0.10	0.07	0.13	0.06	0.12	0.12

Table 3: EXAMPLE CHROMOSOME WITH WEIGHT VALUES

3.4.2 Crossover and Mutation Mechanisms

PyGad offers a diverse array of crossover functions, encompassing single-point crossover, double-point crossover, and uniform crossover, to name a few. The process of parent selection is equally versatile, supporting widely recognised methods such as Tournament, Steady State, Stochastic Universal, Rank, and Random selection. For the purposes of this study, double point cross-over and random mutation and a Tournament selection mechanism with a size parameter of 2 were selected, was deemed most appropriate. It's noteworthy that post-crossover, one parent is retained for the ensuing generation. A crossover probability of 0.95 was selected, and upon the culmination of the crossover function, the entire population is subjected to normalization.

Mutation, on the other hand, is introduced with a probability of $1-0.95$, infusing the subsequent generation with an element of randomness. This mirrors sporadic natural phenomena where offspring inherit unanticipated traits. Following the mutation process, the new generation undergoes normalization.

3.4.3 Weight Normalization

A normalisation procedure is instituted to circumvent the potential pitfall of disproportionate weight allocation to a specific threshold. This helps the summation of the weights to stay within the range of 1. If not placed, the total weight can be very large, and one threshold can far outweigh the other thresholds, rendering the purpose of multi-thresholds undone. This process is orchestrated using the subsequent formula:

$$\frac{g_i}{\sum_{i=1}^{n} g_i} \tag{6}$$

The fraction represents the ratio of the value of a specific gene, denoted as g_i, to the summation of the values of all genes in the set. The numerator, g_i, is the value of the i^{th} gene. The denominator is the total sum of gene values, where the summation runs from the first gene (i=1) to the n^{th} gene, with n being the total number of genes in the set. This equation essentially gives the relative proportion of a particular gene's value in comparison to the total gene values.

3.4.4 Evaluating Chromosome's fitness

In the context of this study, the assessment of the fitness of the weights is intrinsically linked to the Sharpe ratio derived from each weight set. The function, when integrated into the Genetic Algorithm, leverages the Sharpe ratio as a metric. To compute the Sharpe ratio, each stock is subjected to a designated strategy, which subsequently dictates trading actions based on the generated signals. This process culminates in the determination of the Rate of Return (RoR) using the following mathematical representation:

$$RoR = \sum_{i=1}^{n} (SellingPrice - (BuyingPrice + TransactionCost))/BuyingPrice \tag{7}$$

Where n denotes the total number of trades.

Subsequent to this, the returns from the trades are harnessed to ascertain the associated risk, as articulated by:

$$Risk = \sqrt{var(Returns)} \tag{8}$$

Upon the calculation of both the Rate of Return and Risk for each stock, these values are then employed to derive the Sharpe ratio. It's noteworthy that the computation of the Sharpe ratio incorporates a risk-free interest rate of 2.5%, aligned with the yield of US treasury bonds. The formula for the Sharpe ratio is given by:

$$SR = \frac{RoR - R_f}{Risk} \tag{9}$$

Where R_f represents the risk-free interest rate.

For each stock under consideration, a singular RoR, risk, and Sharpe ratio are computed. The overarching fitness function value is then determined by taking the average Sharpe ratio across all 17 stocks.

3.4.5 Optimization of Hyperparameters

The efficacy of the Genetic Algorithm is contingent upon meticulous hyperparameter optimization to ensure optimal outcomes. In the context of this study, a systematic grid search was conducted, encompassing a range of parameters. Specifically, the initial population was varied across the set 20, 50, 70, 100, 150, 200, 300, the number of generations was examined over the range 15, 18, 25, 30, 35, 45, and the crossover probability p was tested for values within the set 0.75, 0.85, 0.95, 0.99. During this grid search, strategies were trained on the designated training data. In instances where the derived Sharpe ratio was negative, the fitness value was set to negative infinity. Conversely, if the Sharpe ratio was positive, the fitness function proceeded to compute the Sharpe ratio on the validation dataset, subsequently returning that value.

Upon the culmination of the grid search, the ensuing values, as tabulated below, were deemed most appropriate for the experiment:

Table 4: OPTIMIZED PARAMETERS FOR THE GENETIC ALGORITHM

Parameter	Selected Value
Population size	100
Generations	18
Crossover probability	0.95
Mutation probability	0.05
Tournament size	2

4 Experiment Results

4.0.1 Comparative Analysis of Trading Strategies

The primary objective of this research was to ascertain whether the application of a stochastic search technique, specifically a genetic algorithm with normalized population, could enhance trading performance when optimizing recommendations from multiple thresholds, as compared to previous work done with non-normalized GA optimised multi-threshold strategies. Notably, this methodology normalized the population in the genetic algorithm to be within a range from 0-1. Both normalized and non-normalized GA is used to optimize the three strategies described in our methodology. Given the implementation of 10 thresholds for St1 and St2, and 5 thresholds for St3, each threshold yielded unique recommendations, resulting in varied outcomes. For clarity in subsequent sections, the GA optimized results will be denoted as *GA* and normalized ones will be denoted as *GA Norm* and the single threshold results will be denoted as $\theta 1$ through $\theta 10$.

To rigorously assess the efficacy of the multi-threshold strategies (St1, St2, and St3), they were juxtaposed against their single-threshold counterparts. The latter were based on 10 and 5 distinct thresholds for St1/St2 and St3, respectively. The strategies optimized using the GA are henceforth referred to as GA-optimized strategies, labeled as GA1, GA2, and GA3 and the normalized GA-optimized strategies are labeled as GA1 Norm, GA2 Norm and GA3 Norm.

Benchmarking Against Established Financial Metrics

For a comprehensive evaluation, we benchmarked our strategies against two well-established technical analysis tools: the relative strength index (RSI) and the moving average convergence divergence (MACD). Historically, these tools, along with the Buy-and-Hold strategy, have been the touchstones for comparative studies in this domain. In our analysis, the default period lengths for MACD were set at 26 and 12, while for RSI it was set at 14.

The Buy and Hold (BandH) strategy, a passive investment approach, was also employed as a benchmark. This strategy entails purchasing a financial product and retaining it over an extended duration, irrespective of market volatilities. In our model, the trading action was simulated by purchasing the financial product at the onset of the test set and liquidating it at its conclusion.

Results and Discussion

Table 5 provides a comprehensive overview of the performance metrics - Sharpe Ratio (SR), Rate of Return (RoR), and Standard Deviation (Risk) - for each strategy across various thresholds and GA-optimized results. It is evident that while some thresholds underperformed across all metrics, others, such as $\theta 1$ and $\theta 2$ for St1, $\theta 1$, and $\theta 6$ for St2, and $\theta 2$ for St3, showcased commendable SR values relative to their peers. The RoR for all thresholds was moderate, with St2 under $\theta 3$ and $\theta 2$ under St3 registering a marginally higher profit. The risk performance across individual thresholds oscillated within a narrow range. That being said $\theta 9$ and $\theta 10$ for St1, $\theta 3$, $\theta 4$ and $\theta 8$ for St2, and $\theta 5$ for St3 produced negative Sharpe ratio and performed poorly compared to it's peers.

Table 5: COMPARATIVE PERFORMANCE RESULTS: GA NORMALIZED VALUES, GA VALUES AND 10 INDIVIDUAL DC-THRESHOLDS. BEST METRIC VALUES HIGHLIGHTED IN BOLDFACE

	Sharpe Ratio			Rate of Return			Risk		
	St1	St2	St3	St1	St2	St3	St1	St2	St3
GA Norm.	**3.18**	**2.97**	**7.59**	**0.12**	**0.14**	**0.21**	**0.04**	**0.04**	**0.02**
GA	2.08	2.26	**7.59**	**0.12**	0.13	**0.21**	0.06	**0.04**	**0.02**
θ_1	2.08	1.34	7.11	0.09	**0.14**	0.17	0.05	0.05	**0.02**
θ_2	2.07	0.72	**7.59**	0.09	0.11	**0.21**	0.05	0.05	**0.02**
θ_3	1.16	-0.42	6.83	0.02	-0.001	0.20	0.05	**0.04**	**0.02**
θ_4	1.92	-1.10	5.44	0.08	-0.001	0.15	0.05	**0.04**	**0.02**
θ_5	0.74	0.37	0.00	0.03	0.05	0.09	0.05	**0.04**	**0.02**
θ_6	0.33	1.49	-	-0.01	0.13	-	0.05	0.05	-
θ_7	0.38	0.94	-	-0.01	0.13	-	0.06	0.06	-
θ_8	0.55	-0.22	-	-0.02	0.03	-	0.06	**0.04**	-
θ_9	-0.02	0.97	-	-0.03	0.07	-	0.06	**0.04**	-
θ_{10}	-0.38	0.11	-	-0.04	0.04	-	0.07	0.05	-

A salient observation from Table 5 is the marked improvement in SR and RoR metrics when employing the GA normalized and GA optimization. For instance, the normalized GA-optimized strategies consistently outperformed or matched their single-threshold counterparts across all ten thresholds in terms of RoR. In terms of risk, the normalized GA-optimized strategies mirrored the performance of their single-threshold counterparts, with minor variations.

For the Sharpe Ratio, the GA Normalized values display the highest figures across all stages, with values of 3.18, 2.97, and 7.59 for St1, St2, and St3 respectively. When compared to the GA values, St1 and St2 show a decrease of approximately 34% and 24%, respectively, while St3 remains consistent. Among the individual DC-thresholds, represented by θ_1 to θ_{10}, there's a notable variance. The Sharpe Ratio for St1 ranges from a decrease of about 112% in θ_{10} to no change in θ_1 and θ_2, compared to the GA Normalized value.

Regarding the Rate of Return (RoR), the GA Normalized values again lead with figures of 0.12, 0.14, and 0.21 for St1, St2, and St3 respectively. The GA values for St1 remain consistent with the normalized values, but St2 sees a marginal 7% decrease. Among the DC-thresholds, θ_3 registers the most significant drop in St2, with a decrease of about 100%, while θ_1 matches the highest normalized value.

On the risk front, the GA Normalized values consistently present the lowest risk across all stages, with values of 0.04, 0.04, and 0.02 for St1, St2, and St3 respectively. The GA values for St1 indicate a 50% increase in risk, but St2 and St3 match the normalized values. The individual DC-thresholds maintain a relatively stable risk profile, especially for St2, with most hovering around the 0.04 or 0.05 mark. St3 consistently showcases the lowest risk at 0.02 for those thresholds that provide values.

In the table 6 for the GA1 strategy, the normalized version (GA1 Norm) and the non-normalized version (GA1) both exhibit an identical RoR of 0.12. However, the risk for the GA1 Norm is lower at 0.04 compared to 0.06 for GA1, indicating a 50% increase in risk when not normalized. This increased risk results in a decrease in the Sharpe Ratio from 3.18 for GA1 Norm to 2.08 for GA1, marking a decline of approximately 34%.

The GA2 strategy, when normalized (GA2 Norm), presents a RoR of 0.14, which slightly decreases to 0.13 in its non-normalized version (GA2). Both versions maintain an identical risk level of 0.04. The Sharpe Ratio, however, experiences a drop from

Table 6: COMPARATIVE PERFORMANCE RESULTS: GA NORMALIZED VALUES, GA VALUES WITH BENCH MARKED TRADING STRATEGIES. BEST METRIC VALUES HIGHLIGHTED IN BOLDFACE

Strategy	RoR (Average)	Risk (Average)	Sharpe Ratio (Average)
GA1 Norm	0.12	0.04	3.18
GA1	0.12	0.06	2.08
GA2 Norm	0.14	0.04	2.97
GA2	0.13	0.04	2.26
GA3 Norm	**0.21**	**0.02**	**7.59**
GA3	**0.21**	**0.02**	**7.59**
MACD	0.10968	0.05733	0.93920
RSI	0.08938	0.03	2.35408
Buy and Hold	0.19	-	-

2.97 in the normalized version to 2.26 in the non-normalized one, translating to a decrease of around 24%.

The GA3 strategy remains consistent between its normalized (GA3 Norm) and non-normalized (GA3) versions, with both showcasing a RoR of 0.21, a risk of 0.02, and an impressive Sharpe Ratio of 7.59.

Comparatively, the MACD strategy has a RoR of 0.10968, a risk of 0.05733, and a Sharpe Ratio of 0.93920. The RSI strategy, on the other hand, offers a RoR of 0.08938, a risk of 0.03, and a Sharpe Ratio of 2.35408. Notably, the Buy and Hold strategy provides a RoR of 0.19 but doesn't specify values for risk or the Sharpe Ratio.

In summary, while the GA strategies, especially when normalized, tend to outperform the MACD and RSI in terms of the Sharpe Ratio, the Buy and Hold strategy offers a competitive RoR without specified risk metrics.

Overall, the GA Normalized values predominantly outshine both the GA and individual DC thresholds in terms of Sharpe Ratio and RoR, while also consistently offering the lowest risk.

In conclusion, the GA's multi-threshold optimization significantly bolstered trading performance, particularly in the SR and RoR metrics. While the robust performance in SR and RoR can be partially attributed to the prevailing bull market during the test period, the DC paradigm's intrinsic merits also played a pivotal role. This assertion is further corroborated when comparing the GA-optimized strategies against established benchmarks like BandH, RSI, and MACD, where similar high-performance metrics were observed.

5 Conclusion

From the results of our research, it can be concluded that the normalized GA optimization on the three trading strategies with multi-threshold offers a steady increase to already predominant GA optimization. The superiority of the GA normalized optimization over its predecessor, the GA optimized method, is clearly demonstrated. This enhancement in performance is attributed to two primary factors: firstly, the enriched strategy space provides traders with a more diverse set of options, and secondly, the stochastic search via GA in the multi-threshold model adeptly identifies strategies that surpass the performance of single-threshold ones.

This experiment, comprised of testing 17 stocks under varying DC thresholds for different strategies, further confirmed these assertions. The results were unequivocal: the multi-threshold DC paradigm, when optimized using a normalized GA, not only generates profitable trading strategies but also consistently outshines individual thresholds in terms of Sharpe Ratio (SR) and Rate of Return (RoR). Moreover, when pitted against established benchmarks like MACD and RSI, the Normalized GA-optimized strategy emerged as the superior contender, showcasing its statistical dominance.

For future expansion on this work, the chromosomes can be expanded to encapsulate multiple thresholds and strategies. It should unlock even greater performance capabilities. Also drawn from the results, the risk can be further optimized for both normalised and regular GA-optimized methods.

Bibliography

[1] M. Aloud, E. Tsang, R. Olsen, and A. Dupuis, "A directional change events approach for studying financial time series," *Economics*, vol. 36, p. (online), 2012.

[2] D. Guillaume, M. Dacorogna, R. Dave, U. Müller, R. Olsen, and O. Pictet, "From the bird's eye to the microscope: a survey of new stylized facts of the intra-daily foreign exchange markets," *Finance Stoch*, vol. 1, pp. 95–129, 1997.

[3] H. Markowitz, "Portfolio selection*," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.

[4] O. Salman, T. Melissourgos, and M. Kampouridis, "Optimization of trading strategies using a genetic algorithm under the directional changes paradigm with multiple thresholds," in *Proceedings of the Institute of Electrical and Electronics Engineers (IEEE)*, Institute of Electrical and Electronics Engineers (IEEE), 2023. Print.

[5] J. B. Glattfelder, A. Dupuis, and R. Olsen, "An extensive set of scaling laws and the fx coastline." Unpublished manuscript, 2008.

[6] J. B. Glattfelder, A. Dupuis, and R. B. Olsen, "Patterns in high-frequency fx data: discovery of 12 empirical scaling laws," *Quantitative Finance*, vol. 11, no. 4, pp. 599–614, 2011.

[7] E. P. Tsang, R. Tao, A. Serguieva, and S. Ma, "Profiling high-frequency equity price movements in directional changes," *Quantitative finance*, vol. 17, no. 2, pp. 217–225, 2017.

[8] E. Tsang and J. Chen, "Regime change detection using directional change indicators in the foreign exchange market to chart brexit," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 2, no. 3, pp. 185–193, 2018.

[9] O. Salman, M. Kampouridis, and D. Jarchi, "Trading strategies optimization by genetic algorithm under the directional changes paradigm," in *2022 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE, 2022.

[10] R. Tao, *Using directional change for information extraction in financial market data.* PhD thesis, University of Essex, 2018.

[11] Y. Hu, K. Liu, X. Zhang, L. Su, E. Ngai, and M. Liu, "Application of evolutionary computation for rule discovery in stock algorithmic trading: A literature review," *Applied Soft Computing*, vol. 36, pp. 534–551, 2015.

[12] C.-F. Huang, C.-J. Hsu, C.-C. Chen, B.-R. Chang, and C.-A. Li, "An intelligent model for pairs trading using genetic algorithms," *Computational Intelligence and Neuroscience*, vol. 2015, pp. 16–16, 2015.

[13] R. Kumar, P. Kumar, and Y. Kumar, "Multi-step time series analysis and forecasting strategy using arima and evolutionary algorithms," *International Journal of Information Technology*, vol. 14, no. 1, pp. 359–373, 2022.

[14] X. Long, M. Kampouridis, and P. Kanellopoulos, "Genetic programming for combining directional changes indicators in international stock markets," in *Parallel Problem Solving from Nature–PPSN XVII: 17th International Conference, PPSN 2022, Dortmund, Germany, September 10-14, 2022, Proceedings, Part II*, pp. 33–47, Springer, 2022.

[15] E. Christodoulaki and M. Kampouridis, "Using strongly typed genetic programming to combine technical and sentiment analysis for algorithmic trading," in *2022 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, IEEE, 2022.

[16] A. Adegboye, M. Kampouridis, and F. Otero, "Improving trend reversal estimation in forex markets under a directional changes paradigm with classification algorithms," *International Journal of Intelligent Systems*, 2021.

[17] A. Adegboye, M. Kampouridis, and F. Otero, "Algorithmic trading with directional changes," *Artificial Intelligence Review*, pp. 1–26, 2022.

[18] R. Aroussi, "yfinance," 2023. Python library.

[19] A. F. Gad, "Pygad," 2023. Python library.

6 Appendix

```python
"""
This script contains functions for calculating Directional Change (DC) and
related indicators.
"""
from typing import List, Tuple
from collections import namedtuple
import numpy as np
import pandas as pd

DCEvent = namedtuple("DCEvent", ["index", "price", "event"])
ThresholdSummary = namedtuple("ThresholdSummary", ["dc", "p_ext"])


# flake8: noqa: C901
def calculate_dc(
    prices: List[float], threshold: float
) -> Tuple[
    List[Tuple[int, float]],
    List[Tuple[int, float]],
    List[Tuple[float, int, str]],
]:
    """
    Calculate Directional Change (DC) based on given price data and threshold.

    Parameters:
    - prices: A list of price data.
    - threshold: A threshold value to determine upturns and downturns.

    Returns:
```

```python
    - upturn, downturn, extreme price points
    """

    last_low_index = 0
    last_high_index = 0
    last_low_price = prices[last_low_index]
    last_high_price = prices[last_high_index]

    upturn_dc = []
    downturn_dc = []
    p_ext = []

    current_index = 1

    # First while loop: Determine
    # the initial event (either an upturn or downturn)
    # This loop will break once the first event is identified.
    while current_index < len(prices):
        if prices[current_index] <= last_high_price * (1 - threshold):
            downturn_dc.append((current_index, prices[current_index]))
            p_ext.append((last_high_price, last_high_index, "DR"))
            event = "DR"
            break
        elif prices[current_index] >= last_low_price * (1 + threshold):
            upturn_dc.append((current_index, prices[current_index]))
            p_ext.append((last_low_price, last_low_index, "UR"))
            event = "UR"
            break
        elif prices[current_index] > last_high_price:
            last_high_index = current_index
            last_high_price = prices[current_index]
        elif prices[current_index] < last_low_price:
            last_low_index = current_index
            last_low_price = prices[current_index]
        current_index += 1
```

```python

    # Second while loop: Determine subsequent events based on the initial event
    # This loop will continue until all prices are processed.
    while current_index < len(prices):
        if event == "DR":
            if prices[current_index] < last_low_price:
                last_low_index = current_index
                last_low_price = prices[current_index]
            elif prices[current_index] >= last_low_price * (1 + threshold):
                upturn_dc.append((current_index, prices[current_index]))
                p_ext.append((last_low_price, last_low_index, "UR"))
                last_high_index = current_index
                last_high_price = prices[current_index]
                event = "UR"
        elif event == "UR":
            if prices[current_index] > last_high_price:
                last_high_index = current_index
                last_high_price = prices[current_index]
            elif prices[current_index] <= last_high_price * (1 - threshold):
                downturn_dc.append((current_index, prices[current_index]))
                p_ext.append((last_high_price, last_high_index, "DR"))
                event = "DR"
                last_low_index = current_index
                last_low_price = prices[current_index]
        current_index += 1

    return upturn_dc, downturn_dc, p_ext


def calculate_dc_indicators(
    prices: List[float],
    thresholds: List[float],
    chunk_size: int = 4,
) -> List[pd.DataFrame]:
```

```
    """
    Calculate DC indicators based on given parameters.

    Args:
    - prices: List of price values.
    - thresholds: List of Threshold value for calculation.
    - chunk_size: Size of chunks for splitting overshoot data.

    Returns:
    - List of ThresholdSummary objects containing DC data, p_ext, all_overshoot, all_overshoot_with_osv_best data.
    """

    # upturn: List[Tuple[int, float]],
    # downturn: List[Tuple[int, float]],
    # p_ext: List[Tuple[float, int, str]],
    # threshold: float,

    summaries = []
    for threshold in thresholds:
        upturn, downturn, p_ext = calculate_dc(prices, threshold)
        upturn = [DCEvent(x[0], x[1], "UR") for x in upturn]
        downturn = [DCEvent(x[0], x[1], "DR") for x in downturn]
        p_ext = [DCEvent(x[1], x[0], x[2]) for x in p_ext]
        all_overshoot = compute_all_overshoot(
            prices, upturn, downturn, p_ext, threshold
        )
        chunks = split_into_chunks(all_overshoot, chunk_size)
        medians = [np.median([x[2] for x in chunk]) for chunk in chunks]
        all_overshoot_with_osv_best = [
            (x + (medians[i],)) for i, chunk in enumerate(chunks) for x in chunk
        ]
        indexes = [x[0] for x in all_overshoot_with_osv_best]
        osv_prices = [x[1] for x in all_overshoot_with_osv_best]
        osv_cur = [x[2] for x in all_overshoot_with_osv_best]
        event = [x[3] for x in all_overshoot_with_osv_best]
```

```
        osv_best = [x[4] for x in all_overshoot_with_osv_best]

        summaries.append(
            pd.DataFrame(
                data={
                    "price": osv_prices,
                    "osv_cur": osv_cur,
                    "osv_best": osv_best,
                    "event": event,
                },
                index=indexes,
            )
        )
    return summaries


def compute_all_overshoot(
    prices: List[float],
    upturn: List[Tuple[int, float]],
    downturn: List[Tuple[int, float]],
    p_ext: List[Tuple[float, int, str]],
    threshold: float,
) -> List[Tuple[int, float, float, str]]:
    """
    Compute all overshoot values based on given parameters.

    Args:
    - prices: List of price values.
    - upturn: List of upturn events.
    - downturn: List of downturn events.
    - p_ext: List of price extension events.
    - threshold: Threshold value for calculation.

    Returns:
    - List of tuples containing overshoot data.
    """
    all_overshoot = []
    dc_data, p_ext_data = merge_dc_events(upturn, downturn,
        p_ext)

    dc_indexes = dc_data.index
```

```
    p_ext_indexes = p_ext_data.index
    i = 0
    while i < len(dc_indexes) - 1:
        if p_ext_indexes[i + 1] - dc_indexes[i] > 0:
            p_dcc = p_ext_data.iloc[i]["price"] * (1 +
                threshold)
            for j in range(dc_indexes[i], p_ext_indexes[i + 1])
                :
                osv_cur = (prices[j + 1] / p_dcc) / (threshold
                    * p_dcc)
                all_overshoot.append(
                    (
                        j + 1,
                        prices[j + 1],
                        osv_cur,
                        dc_data.iloc[i]["event"],
                    )
                )
        i += 1

    return all_overshoot


def split_into_chunks(
    data: List[Tuple[int, float, float, str]], chunk_size: int
) -> List[List[Tuple[int, float, float, str]]]:
    """
    Split data into chunks of specified size.

    Args:
    - data: List of data to be split.
    - chunk_size: Size of each chunk.

    Returns:
    - List of chunks.
    """
    return [data[i : i + chunk_size] for i in range(0, len(data
        ), chunk_size)]


def compute_threshold_dc_summaries(
```

```
    prices: List[float], thresholds: List[float]
) -> List[ThresholdSummary]:
    """
    Compute summaries for each threshold based on price data.

    Args:
    - prices: List of price values.
    - thresholds: List of threshold values.

    Returns:
    - List of ThresholdSummary objects containing DC data and
      p_ext data.
    """
    summaries = []
    for threshold in thresholds:
        upturn, downturn, p_ext = calculate_dc(prices,
            threshold)
        upturn = [DCEvent(x[0], x[1], "UR") for x in upturn]
        downturn = [DCEvent(x[0], x[1], "DR") for x in downturn
            ]
        p_ext = [DCEvent(x[1], x[0], x[2]) for x in p_ext]
        dc_data, p_ext_data = merge_dc_events(upturn, downturn,
             p_ext)
        summaries.append(ThresholdSummary(dc_data, p_ext_data))
    return summaries


def merge_dc_events(
    upturn: List[DCEvent],
    downturn: List[DCEvent],
    p_ext: List[DCEvent],
) -> Tuple[pd.DataFrame, pd.DataFrame]:
    """
    Merge upturn and downturn events into a single DataFrame.

    Args:
    - upturn: List of upturn DC events.
    - downturn: List of downturn DC events.
    - p_ext: List of p_ext events.

    Returns:
```

```python
    - Tuple of DataFrames containing merged DC data and p_ext
      data.
    """
    dc_indexes, dc_prices, dc_event = [], [], []
    if upturn[0].index < downturn[0].index:
        events = list(zip(upturn, downturn))
    else:
        events = list(zip(downturn, upturn))
    for a, b in events:
        dc_indexes.extend([a.index, b.index])
        dc_prices.extend([a.price, b.price])
        dc_event.extend([a.event, b.event])
    dc_data = pd.DataFrame(
        data={"price": np.array(dc_prices), "event": dc_event},
        index=dc_indexes,
    )
    p_ext_data = pd.DataFrame(
        data={
            "price": [x.price for x in p_ext],
            "event": [x.event for x in p_ext],
        },
        index=[x.index for x in p_ext],
    )
    return dc_data, p_ext_data
```

```python
"""
This script runs the strategy1 based on thresholds, weights,
  and prices.
"""
import datetime
import os
import pickle
from typing import List, Tuple, Dict, Union
from collections import namedtuple
import numpy as np
import pandas as pd
from helper.dc import compute_threshold_dc_summaries

DCEvent = namedtuple("DCEvent", ["index", "price", "event"])
ThresholdSummary = namedtuple("ThresholdSummary", ["dc", "p_ext
  "])
```

```python

BUY_COST_MULTIPLIER = 1.00025


def get_thresholds_decision(
    threshold_dc_summary: ThresholdSummary, prices: List[float]
) -> List[str]:
    """
    Get decisions based on threshold summaries and prices.

    Parameters:
    - threshold_dc_summary (ThresholdSummary): Summary of the
        threshold.
    - prices (List[float]): List of prices.

    Returns:
    - List[str]: Decisions for each price.
    """
    decisions = ["h"] * len(prices)

    dc = threshold_dc_summary.dc
    p_ext = threshold_dc_summary.p_ext

    i = 0
    while i < dc.shape[0] - 1:
        if (
            dc.iloc[i]["event"] == "DR"
            and p_ext.iloc[i + 1].name - dc.iloc[i].name > 0
        ):
            j = dc.iloc[i].name + ((dc.iloc[i].name - p_ext.
                iloc[i].name) * 2)
            if j < len(prices):
                decisions[j] = "b"
        elif (
            dc.iloc[i]["event"] == "UR"
            and p_ext.iloc[i + 1].name - dc.iloc[i].name > 0
        ):
            j = dc.iloc[i].name + ((dc.iloc[i].name - p_ext.
                iloc[i].name) * 2)
            if j < len(prices):
                decisions[j] = "s"
```

```
        i += 1

    return decisions


def calculate_decision(row: pd.Series, weights: List[float]) -> str:
    """
    Calculate decision based on row data and weights.

    Parameters:
    - row (pd.Series): Row data.
    - weights (List[float]): Weights for decision.

    Returns:
    - str: Decision.
    """

    decisions_options = [("s", 0), ("h", 0), ("b", 0)]
    for i in range(1, len(weights) + 1):
        if row[i] == "b":
            decisions_options[2] = (
                "b",
                decisions_options[2][1] + weights[i - 1],
            )
        elif row[i] == "s":
            decisions_options[0] = (
                "s",
                decisions_options[0][1] + weights[i - 1],
            )
        else:
            decisions_options[1] = (
                "h",
                decisions_options[1][1] + weights[i - 1],
            )

    return max(decisions_options, key=lambda x: x[1])[0]


def set_decisions(
    df: pd.DataFrame, theta_thresholds: List[float]
```

```python
) -> Dict[str, pd.DataFrame]:
    """
    Set decisions based on dataframe and thresholds.

    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - theta_thresholds (List[float]): List of thresholds.

    Returns:
    - Dict[str, pd.DataFrame]: Decision by thresholds.
    """
    stock_decision_by_thresholds = {}

    for col in df.columns[1:]:
        stock_decision_by_thresholds[col] = pd.DataFrame({"prices": df[col]})
        threshold_dc_summaries = compute_threshold_dc_summaries(
            df[col], theta_thresholds
        )

        for i in range(len(theta_thresholds)):
            decisions = get_thresholds_decision(
                threshold_dc_summaries[i], df[col]
            )
            stock_decision_by_thresholds[col][f"threshold_{i}"] = decisions
    return stock_decision_by_thresholds


def get_stock_returns(
    df: pd.DataFrame, weights: List[float], stock_data: pd.DataFrame
) -> Dict[str, List[Union[float, None]]]:
    """
    Get stock returns based on dataframe, weights, and stock data.

    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - weights (List[float]): Weights for decision.
```

```
    - stock_data (pd.DataFrame): Stock data.

    Returns:
    - Dict[str, List[Union[float, None]]]: Stock returns.
    """
    stock_returns = {}

    for col in df.columns[1:]:
        stock_df = stock_data[col]
        returns = [None] * stock_df.shape[0]
        buy_price = 0

        last_decision = "h"

        for i in range(stock_df.shape[0]):
            row = stock_df.loc[i]
            new_decision = calculate_decision(row, weights)
            if new_decision == "b" and (
                last_decision == "s" or last_decision == "h"
            ):
                last_decision = new_decision
                buy_price = row["prices"]
            elif new_decision == "s" and last_decision == "b":
                last_decision = new_decision
                returns[i] = (
                    row["prices"] - (buy_price *
                       BUY_COST_MULTIPLIER)
                ) / (buy_price)
                buy_price = 0

        if buy_price != 0:
            returns[-1] = (
                row["prices"] - (buy_price *
                   BUY_COST_MULTIPLIER)
            ) / (buy_price)

        stock_returns[col] = returns
    return stock_returns


def calculate_metrics(
```

```
    returns: List[Union[float, None]], risk_free_rate: float = 0.01
) -> Tuple[float, float, float]:
    """
    Calculate RoR, Risk, and Sharpe Ratio from a return array.

    Parameters:
    - returns (List[Union[float, None]]): Array of returns.
    - risk_free_rate (float): Risk-free rate. Default is 0.01 (1%).

    Returns:
    - Tuple[float, float, float]: RoR, Risk, Sharpe Ratio.
    """
    try:
        returns = np.array(returns)
        returns_only = returns[returns != np.array(None)]
        RoR = sum(returns_only)

        volatility = np.std(returns_only)
        sharpe_ratio = (RoR - risk_free_rate) / volatility

        return RoR, volatility, sharpe_ratio
    except Exception:
        return 0, 0, 0


def load_strategy_1(
    df: pd.DataFrame,
    thresholds: list,
    pkl_filename="data/strategy1_data.pkl",
    excel_filename="output/strategy1_output.xlsx",
    export_excel: bool = False,
) -> dict:
    """
    Load strategy 1 data. If the data file exists, it reads from the file.
    Otherwise, it sets decisions based on the provided dataframe and thresholds,
    and optionally exports the results to an Excel file.

```

```
    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
    - thresholds (list): List of thresholds for making decisions.
    - export_excel (bool, optional): Whether to export the results to an Excel file. Defaults to False.

    Returns:
    - dict: Dictionary containing decisions by thresholds.
    """

    stock_decision_by_thresholds = {}
    # Check if the file exists
    if os.path.exists(pkl_filename):
        # If the file exists, load it
        with open(pkl_filename, "rb") as file:
            stock_decision_by_thresholds = pickle.load(file)
    else:
        stock_decision_by_thresholds = set_decisions(df, thresholds)

        if export_excel:
            # Create a new Excel writer object
            # pylint: disable=abstract-class-instantiated
            with pd.ExcelWriter(excel_filename, engine="openpyxl") as writer:
                for (
                    sheet_name,
                    stock_data,
                ) in stock_decision_by_thresholds.items():
                    stock_data.to_excel(
                        writer, sheet_name=sheet_name, index=False
                    )

        # If the file doesn't exist, save the dictionary to the file
        with open(pkl_filename, "wb") as file:
            pickle.dump(stock_decision_by_thresholds, file)

    return stock_decision_by_thresholds
```

```python


def strategy1_fitness_function(
    df: pd.DataFrame, weights: list, stock_data: pd.DataFrame
) -> float:
    """
    Calculate the fitness of strategy 1 based on Sharpe Ratios for given weights.

    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
    - weights (list): List of weights for making decisions.
    - stock_data (pd.DataFrame): Dataframe containing stock data.

    Returns:
    - float: Mean of the Sharpe Ratios, representing the fitness of the strategy.
    """

    sharpe_ratios = [0] * (len(df.columns) - 1)
    RoRs = [0] * (len(df.columns) - 1)
    volatility_list = [0] * (len(df.columns) - 1)

    stock_returns = get_stock_returns(df, weights, stock_data)

    for idx, col in enumerate(df.columns[1:]):
        RoR, volatility, sharpe_ratio = calculate_metrics(
            stock_returns[col], 0.025
        )

        sharpe_ratios[idx] = sharpe_ratio
        RoRs[idx] = RoR
        volatility_list[idx] = volatility

    return RoRs, volatility_list, sharpe_ratios
```

```python
"""
This script runs the strategy1 based on thresholds, weights, and prices.
"""
```

```python
import datetime
import os
import pickle
import math
import numpy as np
import pandas as pd
from typing import List, Tuple, Dict, Union
from helper.dc import calculate_dc_indicators


BUY_COST_MULTIPLIER = 1.00025


def get_thresholds_decision(
    threshold_overshoot_summary: pd.DataFrame, prices: List[float]
) -> List[str]:
    """
    Get decisions based on threshold summaries and prices.

    Parameters:
    - threshold_dc_summary (ThresholdSummary2): Summary of the threshold.
    - prices (List[float]): List of prices.

    Returns:
    - List[str]: Decisions for each price.
    """
    decisions = ["h"] * len(prices)

    i = 0

    while i < len(threshold_overshoot_summary.index):
        if threshold_overshoot_summary.iloc[i]["event"] == "DR" and abs(
            threshold_overshoot_summary.iloc[i]["osv_cur"]
        ) >= abs(threshold_overshoot_summary.iloc[i]["osv_best"]):
            decisions[i] = "b"
        elif threshold_overshoot_summary.iloc[i]["event"] == "UR" and abs(
```

```python
            threshold_overshoot_summary.iloc[i]["osv_cur"]
        ) >= abs(threshold_overshoot_summary.iloc[i]["osv_best"
            ]):
            decisions[i] = "s"
        i += 1

    return decisions


def calculate_decision(row: pd.Series, weights: List[float]) ->
    str:
    """
    Calculate decision based on row data and weights.

    Parameters:
    - row (pd.Series): Row data.
    - weights (List[float]): Weights for decision.

    Returns:
    - str: Decision.
    """
    decisions_options = [("s", 0), ("h", 0), ("b", 0)]
    for i in range(1, len(weights) + 1):
        if row[i] == "b":
            decisions_options[2] = (
                "b",
                decisions_options[2][1] + weights[i - 1],
            )
        elif row[i] == "s":
            decisions_options[0] = (
                "s",
                decisions_options[0][1] + weights[i - 1],
            )
        else:
            decisions_options[1] = (
                "h",
                decisions_options[1][1] + weights[i - 1],
            )

    return max(decisions_options, key=lambda x: x[1])[0]

```

```

def set_decisions(
    df: pd.DataFrame, theta_thresholds: List[float]
) -> Dict[str, pd.DataFrame]:
    """
    Set decisions based on dataframe and thresholds.

    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - theta_thresholds (List[float]): List of thresholds.

    Returns:
    - Dict[str, pd.DataFrame]: Decision by thresholds.
    """
    stock_decision_by_thresholds = {}

    for col in df.columns[1:]:
        stock_decision_by_thresholds[col] = pd.DataFrame({"
           prices": df[col]})

        threshold_dc_summaries = calculate_dc_indicators(
            df[col], theta_thresholds
        )

        for i in range(len(theta_thresholds)):
            decisions = get_thresholds_decision(
                threshold_dc_summaries[i], df[col]
            )
            stock_decision_by_thresholds[col][f"threshold_{i}"]
                = decisions
    return stock_decision_by_thresholds


def get_stock_returns(
    df: pd.DataFrame,
    weights: List[float],
    stock_data: pd.DataFrame,
) -> Dict[str, List[Union[float, None]]]:
    """
    Get stock returns based on dataframe, weights, and stock
        data.
```

```
    
    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - weights (List[float]): Weights for decision.
    - stock_data (pd.DataFrame): Stock data.

    Returns:
    - Dict[str, List[Union[float, None]]]: Stock returns.
    """
    stock_returns = {}
    for col in df.columns[1:]:
        stock_df = stock_data[col]
        returns = [None] * stock_df.shape[0]
        buy_price = 0

        last_decision = "h"

        for i in range(stock_df.shape[0]):
            row = stock_df.loc[i]
            new_decision = calculate_decision(row, weights)
            if new_decision == "b" and (
                last_decision == "s" or last_decision == "h"
            ):
                last_decision = new_decision
                buy_price = row["prices"]
            elif new_decision == "s" and last_decision == "b":
                last_decision = new_decision
                returns[i] = (
                    (row["prices"]) - (buy_price *
                       BUY_COST_MULTIPLIER)
                ) / (buy_price)
                buy_price = 0

        if buy_price != 0:
            returns[-1] = (
                row["prices"] - (buy_price *
                   BUY_COST_MULTIPLIER)
            ) / (buy_price)

        stock_returns[col] = returns
    return stock_returns
```

```python


def calculate_metrics(
    returns: List[Union[float, None]], risk_free_rate: float = 0.01
) -> Tuple[float, float, float]:
    """
    Calculate RoR, Risk, and Sharpe Ratio from a return array.

    Parameters:
    - returns (List[Union[float, None]]): Array of returns.
    - risk_free_rate (float): Risk-free rate. Default is 0.01 (1%).

    Returns:
    - Tuple[float, float, float]: RoR, Risk, Sharpe Ratio.
    """
    try:
        returns = np.array(returns)
        returns_only = returns[returns != np.array(None)]
        RoR = sum(returns_only)

        volatility = np.std(returns_only)
        sharpe_ratio = (RoR - risk_free_rate) / volatility

        return RoR, volatility, sharpe_ratio
    except Exception:
        return 0, 0, 0


def load_strategy_2(
    df: pd.DataFrame,
    thresholds: list,
    pkl_filename="data/strategy2_data.pkl",
    excel_filename="output/strategy2_output.xlsx",
    export_excel: bool = False,
) -> dict:
    """
    Load strategy 1 data. If the data file exists, it reads from the file.
    Otherwise, it sets decisions based on the provided
```

```
    dataframe and thresholds,
    and optionally exports the results to an Excel file.

    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
    - thresholds (list): List of thresholds for making decisions.
    - export_excel (bool, optional): Whether to export the results to an Excel file. Defaults to False.

    Returns:
    - dict: Dictionary containing decisions by thresholds.
    """

    stock_decision_by_thresholds = {}

    # Check if the file exists
    if os.path.exists(pkl_filename):
        # If the file exists, load it
        with open(pkl_filename, "rb") as file:
            stock_decision_by_thresholds = pickle.load(file)
    else:
        stock_decision_by_thresholds = set_decisions(df, thresholds)

        if export_excel:
            # Create a new Excel writer object
            # pylint: disable=abstract-class-instantiated
            with pd.ExcelWriter(excel_filename, engine="openpyxl") as writer:
                for (
                    sheet_name,
                    stock_data,
                ) in stock_decision_by_thresholds.items():
                    stock_data.to_excel(
                        writer, sheet_name=sheet_name, index=False
                    )

        # If the file doesn't exist, save the dictionary to the file
```

```
        with open(pkl_filename, "wb") as file:
            pickle.dump(stock_decision_by_thresholds, file)

    return stock_decision_by_thresholds


def strategy2_fitness_function(
    df: pd.DataFrame, weights: list, stock_data: pd.DataFrame
) -> float:
    """
    Calculate the fitness of strategy 1 based on Sharpe Ratios
        for given weights.

    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
    - weights (list): List of weights for making decisions.
    - stock_data (pd.DataFrame): Dataframe containing stock
        data.

    Returns:
    - float: Mean of the Sharpe Ratios, representing the
        fitness of the strategy.
    """

    sharpe_ratios = [0] * (len(df.columns) - 1)
    RoRs = [0] * (len(df.columns) - 1)
    volatility_list = [0] * (len(df.columns) - 1)

    stock_returns = get_stock_returns(df, weights, stock_data)

    for idx, col in enumerate(df.columns[1:]):
        RoR, volatility, sharpe_ratio = calculate_metrics(
            stock_returns[col], 0.025
        )

        sharpe_ratios[idx] = sharpe_ratio
        RoRs[idx] = RoR
        volatility_list[idx] = volatility

    return np.mean(RoRs), np.mean(volatility_list), np.mean(
        sharpe_ratios)
```

```
"""
This script runs the strategy1 based on thresholds, weights,
    and prices.
"""
import datetime
import os
import pickle
import math
import numpy as np
import pandas as pd
from typing import List, Tuple, Dict, Union
from helper.dc import (
    calculate_dc_indicators,
    compute_threshold_dc_summaries,
    ThresholdSummary,
)

BUY_COST_MULTIPLIER = 1.00025


def get_thresholds_decision(
    threshold_overshoot_summary: pd.DataFrame,
    threshold_dc_summary: ThresholdSummary,
    prices: List[float],
) -> List[str]:
    """
    Get decisions based on threshold summaries and prices.

    Parameters:
    - threshold_dc_summary (ThresholdSummary2): Summary of the
        threshold.
    - prices (List[float]): List of prices.

    Returns:
    - List[str]: Decisions for each price.
    """
    decisions = ["h"] * len(prices)
    dc = threshold_dc_summary.dc
    p_ext = threshold_dc_summary.p_ext
    i = 0
```

```
    buy_counter = 0
    bought_index = -1

    while i < dc.shape[0] - 1:
        if dc.iloc[i]["event"] == "DR" and bought_index != -1:
            decisions[dc.iloc[i].name] = "s"
            buy_counter -= 1
            bought_index = -1
        elif (
            dc.iloc[i]["event"] == "UR"
            and p_ext.iloc[i + 1].name - dc.iloc[i].name > 0
        ):
            buy_counter += 1
            if buy_counter == 3:
                decisions[dc.iloc[i].name] = "b"
                bought_index = dc.iloc[i].name + 1
        elif (
            dc.iloc[i]["event"] == "DR"
            and p_ext.iloc[i + 1].name - dc.iloc[i].name > 0
        ):
            buy_counter = 0
        i += 1

    return decisions


def calculate_decision(row: pd.Series, weights: List[float]) ->
    str:
    """
    Calculate decision based on row data and weights.

    Parameters:
    - row (pd.Series): Row data.
    - weights (List[float]): Weights for decision.

    Returns:
    - str: Decision.
    """
    decisions_options = [("s", 0), ("h", 0), ("b", 0)]
    for i in range(1, len(weights) + 1):
        if row[i] == "b":
```

```
                decisions_options[2] = (
                    "b",
                    decisions_options[2][1] + weights[i - 1],
                )
            elif row[i] == "s":
                decisions_options[0] = (
                    "s",
                    decisions_options[0][1] + weights[i - 1],
                )
            else:
                decisions_options[1] = (
                    "h",
                    decisions_options[1][1] + weights[i - 1],
                )

    return max(decisions_options, key=lambda x: x[1])[0]


def set_decisions(
    df: pd.DataFrame, theta_thresholds: List[float]
) -> Dict[str, pd.DataFrame]:
    """
    Set decisions based on dataframe and thresholds.

    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - theta_thresholds (List[float]): List of thresholds.

    Returns:
    - Dict[str, pd.DataFrame]: Decision by thresholds.
    """
    stock_decision_by_thresholds = {}

    for col in df.columns[1:]:
        stock_decision_by_thresholds[col] = pd.DataFrame({"
            prices": df[col]})

        threshold_overshoot_summaries = calculate_dc_indicators
            (
                df[col], theta_thresholds
        )
```

```

        threshold_dc_summaries = compute_threshold_dc_summaries
            (
            df[col], theta_thresholds
        )

        for i in range(len(theta_thresholds)):
            decisions = get_thresholds_decision(
                threshold_overshoot_summaries[i],
                threshold_dc_summaries[i],
                df[col],
            )
            stock_decision_by_thresholds[col][f"threshold_{i}"]
                = decisions
    return stock_decision_by_thresholds


def get_stock_returns(
    df: pd.DataFrame, weights: List[float], stock_data: pd.
        DataFrame
) -> Dict[str, List[Union[float, None]]]:
    """
    Get stock returns based on dataframe, weights, and stock
        data.

    Parameters:
    - df (pd.DataFrame): Dataframe with data.
    - weights (List[float]): Weights for decision.
    - stock_data (pd.DataFrame): Stock data.

    Returns:
    - Dict[str, List[Union[float, None]]]: Stock returns.
    """
    stock_returns = {}
    for col in df.columns[1:]:
        stock_df = stock_data[col]
        returns = [None] * stock_df.shape[0]
        buy_price = 0

        last_decision = "h"

```

```
        for i in range(stock_df.shape[0]):
            row = stock_df.loc[i]
            new_decision = calculate_decision(row, weights)
            if new_decision == "b" and (
                last_decision == "s" or last_decision == "h"
            ):
                last_decision = new_decision
                buy_price = row["prices"]
            elif new_decision == "s" and last_decision == "b":
                last_decision = new_decision
                returns[i] = (
                    (row["prices"]) - (buy_price *
                       BUY_COST_MULTIPLIER)
                ) / (buy_price)
                buy_price = 0
        if buy_price != 0:
            returns[-1] = (
                row["prices"] - (buy_price *
                   BUY_COST_MULTIPLIER)
            ) / (buy_price)

        stock_returns[col] = returns
    return stock_returns


def calculate_metrics(
    returns: List[Union[float, None]], risk_free_rate: float =
       0.01
) -> Tuple[float, float, float]:
    """
    Calculate RoR, Risk, and Sharpe Ratio from a return array.

    Parameters:
    - returns (List[Union[float, None]]): Array of returns.
    - risk_free_rate (float): Risk-free rate. Default is 0.01
       (1%).

    Returns:
    - Tuple[float, float, float]: RoR, Risk, Sharpe Ratio.
    """
    try:
```

```
        returns = np.array(returns)
        returns_only = returns[returns != np.array(None)]
        RoR = sum(returns_only)

        volatility = np.std(returns_only)
        if volatility == 0:
            sharpe_ratio = 0
        else:
            sharpe_ratio = (RoR - risk_free_rate) / volatility

        return RoR, volatility, sharpe_ratio
    except Exception:
        return 0, 0, 0


def load_strategy_3(
    df: pd.DataFrame,
    thresholds: list,
    pkl_filename="data/strategy3_data.pkl",
    excel_filename="output/strategy3_output.xlsx",
    export_excel: bool = False,
) -> dict:
    """
    Load strategy 1 data. If the data file exists, it reads
        from the file.
    Otherwise, it sets decisions based on the provided
        dataframe and thresholds,
    and optionally exports the results to an Excel file.

    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
    - thresholds (list): List of thresholds for making
        decisions.
    - export_excel (bool, optional): Whether to export the
        results to an Excel file. Defaults to False.

    Returns:
    - dict: Dictionary containing decisions by thresholds.
    """

    stock_decision_by_thresholds = {}
```

```python

    # Check if the file exists
    if os.path.exists(pkl_filename):
        # If the file exists, load it
        with open(pkl_filename, "rb") as file:
            stock_decision_by_thresholds = pickle.load(file)
    else:
        stock_decision_by_thresholds = set_decisions(df,
            thresholds)

        if export_excel:
            # Create a new Excel writer object
            # pylint: disable=abstract-class-instantiated
            with pd.ExcelWriter(excel_filename, engine="
                openpyxl") as writer:
                for (
                    sheet_name,
                    stock_data,
                ) in stock_decision_by_thresholds.items():
                    stock_data.to_excel(
                        writer, sheet_name=sheet_name, index=
                            False
                    )

        # If the file doesn't exist, save the dictionary to the
            file
        with open(pkl_filename, "wb") as file:
            pickle.dump(stock_decision_by_thresholds, file)

    return stock_decision_by_thresholds


def strategy3_fitness_function(
    df: pd.DataFrame, weights: list, stock_data: pd.DataFrame
) -> float:
    """
    Calculate the fitness of strategy 1 based on Sharpe Ratios
        for given weights.

    Parameters:
    - df (pd.DataFrame): Dataframe containing the data.
```

```
    - weights (list): List of weights for making decisions.
    - stock_data (pd.DataFrame): Dataframe containing stock
       data.

    Returns:
    - float: Mean of the Sharpe Ratios, representing the
       fitness of the strategy.
    """

    sharpe_ratios = [0] * (len(df.columns) - 1)
    RoRs = [0] * (len(df.columns) - 1)
    volatility_list = [0] * (len(df.columns) - 1)

    stock_returns = get_stock_returns(df, weights, stock_data)

    for idx, col in enumerate(df.columns[1:]):
        RoR, volatility, sharpe_ratio = calculate_metrics(
            stock_returns[col], 0.025
        )

        sharpe_ratios[idx] = sharpe_ratio
        RoRs[idx] = RoR
        volatility_list[idx] = volatility

    return np.mean(RoRs), np.mean(volatility_list), np.mean(
       sharpe_ratios)
```

```
import pygad
import numpy as np
import pandas as pd
import itertools
import logging
import datetime

from strategy1 import load_strategy_1,
   strategy1_fitness_function


# Configure logging settings
logging.basicConfig(
    level=logging.DEBUG,
```

```
    format="%(asctime)s - %(levelname)s - %(message)s",
    filename="app_train_1.log",
    filemode="w",
)  # 'w' will overwrite the log file each time the script runs. Use 'a' to append.

# Create a logger object
logger = logging.getLogger()


def split_func(df):
    # Define the split ratios
    train_ratio = 0.8

    # Calculate the split indices
    total_rows = len(df)
    train_split_idx = int(total_rows * train_ratio)

    # Split the data
    train_df = df.iloc[:train_split_idx].reset_index(drop=True)
    test_df = df.iloc[train_split_idx:].reset_index(drop=True)

    return train_df, test_df


def normalize_population(population):
    """
    Normalize a population of chromosomes such that the sum of genes in each chromosome is 1.

    Parameters:
    - population (numpy.ndarray): Population of chromosomes.

    Returns:
    - numpy.ndarray: Normalized population of chromosomes.
    """
    return population / population.sum(axis=1, keepdims=True)


def on_crossover(ga_instance, offspring_crossover):
    return normalize_population(offspring_crossover)
```

```python


def on_mutation(ga_instance, offspring_mutation):
    return normalize_population(offspring_mutation)


def initialize_population(num_genes, sol_per_pop):
    """
    Initialize a population of chromosomes with genes such that the sum of genes in each chromosome is 1.

    Parameters:
    - num_genes (int): Number of genes in each chromosome.
    - sol_per_pop (int): Number of solutions (chromosomes) in the population.

    Returns:
    - numpy.ndarray: Initialized population of chromosomes.
    """
    return np.random.rand(sol_per_pop, num_genes)


def on_generation(ga_instance):
    current_timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    with open("output/strategy1_train_run_1.txt", "a") as f:
        f.write(f"Generation completed at {current_timestamp}.\n")

    ga_instance.logger.info(
        "Generation = {generation}".format(
            generation=ga_instance.generations_completed
        )
    )
    ga_instance.logger.info(
        "Fitness    = {fitness}".format(
            fitness=ga_instance.best_solution(
                pop_fitness=ga_instance.last_generation_fitness
            )[1]
        )
```

```python
    )


def run_ga(params, loader_function):
    """
    Run the Genetic Algorithm (GA) using pygad library.

    Returns:
    - tuple: Best solution chromosome, its fitness value, and
       its index.
    """
    # Parameters
    num_genes = params["num_genes"]
    num_solutions = params["num_solutions"]
    num_generations = params["num_generations"]
    crossover_probability = params["crossover_probability"]
    mutation_probability = 1 - params["crossover_probability"]
    tournament_size = 2

    fitness_func = loader_function()

    # Create an instance of the GA class
    ga_instance = pygad.GA(
        num_generations=num_generations,
        num_parents_mating=2,
        fitness_func=fitness_func,
        sol_per_pop=num_solutions,
        num_genes=num_genes,
        gene_type=np.float32,
        init_range_low=0,
        init_range_high=1,
        crossover_type="two_points",
        parent_selection_type="tournament",
        K_tournament=tournament_size,
        crossover_probability=crossover_probability,
        on_crossover=on_crossover,
        on_mutation=on_mutation,
        mutation_probability=mutation_probability,
        mutation_type="random",
        keep_parents=1,
        initial_population=initialize_population(num_genes,
```

```
            num_solutions),
        logger=logger,
        on_generation=on_generation,
        random_mutation_max_val=1,
        random_mutation_min_val=0,
        parallel_processing=50,
    )

    ga_instance.run()

    return ga_instance.best_solution()


def loader_function_strategy_1() -> callable:
    """
    Load strategy 1 data and return a fitness function for
        evaluating solutions.

    Returns:
    - callable: A fitness function that evaluates the fitness
        of a solution based on strategy 1.
    """

    # Read the data from CSV
    df = pd.read_csv("data/stock_data.csv")

    train_df, test_df = split_func(df)

    # Define thresholds
    thresholds = (
        np.array([0.098, 0.22, 0.48, 0.72, 0.98, 1.22, 1.55,
            1.70, 2, 2.55])
        / 100
    )
    # Load strategy 1 decisions
    stock_decision_by_thresholds_train = load_strategy_1(
        df=test_df,
        thresholds=thresholds,
        pkl_filename="data/strategy1_train_data_1.pkl",
    )

```

```python
    def fitness_func(
        ga_instance: pygad.GA, solution: list, solution_idx: int
    ) -> float:
        """
        Fitness function for evaluating a given solution.

        Parameters:
        - solution (list): The solution to evaluate.
        - solution_idx (int): Index of the solution.

        Returns:
        - float: Fitness value of the solution.
        """
        print("Running fitness function for solution " + str(solution_idx))
        print("Weights are " + str(solution))
        print(f"Weight sum: {sum(solution)}")
        # Use the solution to generate trading signals and calculate returns for the training set
        RoR, volatility, sharpe_ratio = strategy1_fitness_function(
            test_df, solution, stock_decision_by_thresholds_train
        )

        print(
            "Train fitness function for solution "
            + str(sharpe_ratio)
            + "\t"
            + str(volatility)
            + "\t"
            + str(RoR)
        )

        print("-" * 50)

        return sharpe_ratio

    return fitness_func
```

```
if __name__ == "__main__":
    param_grid = {
        "num_genes": [10],
        "num_solutions": [100],
        "num_generations": [18],
        "crossover_probability": [0.95],
    }
    for i in range(50):
        # Get the current timestamp
        current_timestamp = datetime.datetime.now().strftime(
            "%Y-%m-%d %H:%M:%S"
        )

        with open("output/strategy1_train_run_1.txt", "a") as f:
            f.write(
                f"--------Starting a new GA instance at {current_timestamp}.------------\n"
            )
            f.write(
                f"Starting running GA for strategy 1 with parameters. Run {i + 1}\n"
            )
        all_params = [
            dict(zip(param_grid.keys(), values))
            for values in itertools.product(*param_grid.values())
        ]

        solution, solution_fitness, _ = run_ga(
            all_params[0], loader_function_strategy_1
        )
        with open("output/strategy1_train_run_1.txt", "a") as f:
            f.write(
                str(i)
                + "\t"
                + str(solution)
                + "\t"
                + str(solution_fitness)
```

```
                + "\n"
            )
            f.write(
                f"----------Finished␣running␣GA␣for␣strategy␣1.
                    ␣Run␣{i␣+␣1}-------------\n"
            )
```