Profitable Bitcoin Trading Strategies

Trend, Mean Reversion & Hybrid Models

Ali Azary

Profitable Bitcoin Trading Strategies: Trend, Mean Reversion & Hybrid Models

🔐 **Take the Next Step: Get the Complete Python Code Package!**

You've just unlocked a treasure trove of powerful Bitcoin trading strategies. But reading about them is only the beginning.

To **transform this knowledge into real-world action**, we've created a **fully coded implementation** of every strategy in this guide—ready for backtesting, customization, and deployment.

📦 **The "Profitable Bitcoin Strategies" Python Code Package Includes:**

- ✅ **All strategies from the book** — fully implemented in modular, well-commented Python classes
- 📊 **Backtest-ready** — easily integrated with Backtrader or your own frameworks
- 🔁 **Rolling backtests** — analyze strategy consistency across different market conditions
- ⚙️ **Customizable Parameters** — every strategy has configurable params and clearly structured next() logic
- 🧠 **Advanced & Hybrid Models** — includes Random Forest and Rough Path strategies for cutting-edge insights
- 🚀 **Save countless hours** — go from theory to live testing without writing the base code yourself

Whether you're a trader, researcher, or developer, this premium package is your shortcut to building, testing, and deploying Bitcoin strategies that consistently outperform buy-and-hold benchmarks.

🎁 **Bonus**: All code is optimized for BTC-USD and tested across multiple timeframes.

👉 Ready to unlock the full code base and accelerate your trading journey?

🎯 **Get the Profitable Bitcoin Strategies Code Package now and start building your edge.**

💻 **Visit Profitable Bitcoin Strategies to purchase.**

Table of Contents

1. Introduction to Algorithmic Trading

What is Algorithmic Trading?

Algorithmic trading, often shortened to algo-trading, involves using computer programs to execute trades automatically based on predefined sets of rules or algorithms. These algorithms can consider various factors such as timing, price, quantity, and other market conditions. The primary benefits include speed, accuracy, and the ability to backtest strategies against historical data to assess their potential profitability and risks.

The Role of Backtesting

Backtesting is the process of testing a trading strategy using historical data to determine its hypothetical performance. It's a crucial step in the development of any algorithmic trading strategy, allowing traders to evaluate the viability and profitability of a strategy before risking real capital. A robust backtest can reveal how a strategy would have performed under various market conditions, helping to identify potential weaknesses and areas for optimization.

A particularly robust form of backtesting is **rolling backtesting**. Instead of testing a strategy once over a single, long historical period, rolling backtesting divides the entire historical dataset into multiple, consecutive, or overlapping "windows" (e.g., 3-month, 6-month, or 1-year periods). The strategy is then run independently on each window. This approach provides a more realistic assessment of a strategy's performance across various market cycles and conditions, helping to identify consistency and adaptability. It also helps to mitigate the risk of "overfitting" a strategy to a single historical period.

The following Python function, `run_rolling_backtest`, demonstrates how such a rolling backtest can be implemented using the `backtrader` library and `yfinance` for data:

```
def run_rolling_backtest(
    ticker,
    start,
    end,
    window_months,
    strategy_params=None
):
    strategy_params = strategy_params or {}
    all_results = []
    start_dt = pd.to_datetime(start)
    end_dt = pd.to_datetime(end)
    current_start = start_dt

    while True:
        current_end = current_start + rd.relativedelta(months=window_months)
        if current_end > end_dt:
            break
```

```
        print(f"\nROLLING BACKTEST: {current_start.date()} to
{current_end.date()}")

        # Fetch data using yfinance
        data = yf.download(ticker, start=current_start, end=current_end,
auto_adjust=False, progress=False)

        if data.empty or len(data) < 90:
            print("Not enough data for this period.")
            current_start += rd.relativedelta(months=window_months)
            continue

        if isinstance(data.columns, pd.MultiIndex):
            data = data.droplevel(1, 1)

        # Calculate Buy & Hold return for the period
        start_price = data['Close'].iloc[0]
        end_price = data['Close'].iloc[-1]
        benchmark_ret = (end_price - start_price) / start_price * 100

        feed = bt.feeds.PandasData(dataname=data)
        cerebro = bt.Cerebro()

        cerebro.addstrategy(strategy_class, **strategy_params)
        cerebro.adddata(feed)
        cerebro.broker.setcash(100000)
        cerebro.broker.setcommission(commission=0.001)
        cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

        start_val = cerebro.broker.getvalue()
        cerebro.run()
        final_val = cerebro.broker.getvalue()
        strategy_ret = (final_val - start_val) / start_val * 100

        all_results.append({
            'start': current_start.date(),
            'end': current_end.date(),
            'return_pct': strategy_ret,
            'benchmark_pct': benchmark_ret, # Add benchmark return
            'final_value': final_val,
        })

        print(f"Strategy Return: {strategy_ret:.2f}% | Buy & Hold Return:
{benchmark_ret:.2f}%")
        current_start += rd.relativedelta(months=window_months)

    return pd.DataFrame(all_results)
```

Explanation of `run_rolling_backtest` function:

- **Parameters**:
  - `ticker`: The financial instrument symbol (e.g., 'BTC-USD').
  - `start`, `end`: The overall historical period for the rolling backtest.
  - `window_months`: The duration of each individual rolling window in months.
  - `strategy_params`: A dictionary of parameters to pass to the `backtrader` strategy.
- **Iteration**: The function iterates through the overall `start` to `end` period, creating `window_months`-long segments.
- **Data Fetching**: For each window, it uses `yf.download` to fetch historical data for the specified `ticker`. It also handles multi-level columns by dropping the second level.
- **Benchmark Calculation**: It calculates the simple Buy & Hold return for each individual window to serve as a benchmark for comparison against the strategy's performance within that same window.
- **Backtrader Setup**: A `backtrader.Cerebro` instance is created for each window. The specified strategy (`strategy_class`) is added, along with the fetched data, initial cash, commission, and position sizer.
- **Execution**: `cerebro.run()` executes the backtest for the current window.
- **Results Storage**: The `strategy_ret` (strategy's return) and `benchmark_ret` (Buy & Hold return) for each window are stored in `all_results`, along with the start and end dates of the window and the final portfolio value.
- **Output**: The function prints the strategy and Buy & Hold returns for each window and finally returns a `pandas.DataFrame` containing all the aggregated results.

Key Performance Indicators in Backtesting

When evaluating a strategy through backtesting, several **Key Performance Indicators (KPIs)** are commonly used:

- **Total Return**: The overall percentage gain or loss over the backtesting period.
- **Annualized Return**: The average return earned by an investment over a year, often used to compare strategies with different timeframes.
- **Sharpe Ratio**: Measures risk-adjusted return. A higher Sharpe Ratio indicates better performance for the amount of risk taken. It calculates the excess return per unit of standard deviation of returns.

  $$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}$$

  Where R_p is the portfolio return, R_f is the risk-free rate, and σ_p is the standard deviation of the portfolio's excess return.
- **Max Drawdown**: The largest peak-to-trough decline in the portfolio's value during the backtesting period. It indicates the maximum loss an investor would have faced.

- **Win Rate**: The percentage of winning trades out of the total number of trades.
- **Profit Factor**: The ratio of gross winning trades to gross losing trades. A profit factor greater than 1 indicates a profitable strategy.

$$\text{Profit Factor} = \frac{\text{Gross Profit}}{\text{Gross Loss}}$$

- **Total Trades**: The total number of buy and sell operations executed during the backtest.
- **Average Win/Loss**: The average profit from winning trades and the average loss from losing trades.

2. Trend-Following Strategies

Trend-following strategies aim to profit by analyzing the momentum of an asset's price to determine its prevailing direction. The core idea is that once a trend is established, it is more likely to continue than to reverse. These strategies typically involve entering a trade after a trend has been identified and exiting when the trend shows signs of reversal or weakening.

Donchian Breakout Strategy

- **Logic and Idea**: This strategy is a classic trend-following approach based on **Donchian Channels**. It aims to capture trends by entering a trade when the price breaks out of the highest high or lowest low over a specified period. The strategy incorporates additional filters like a Moving Average (MA) for higher-timeframe trend confirmation. **Trailing stops** based on Average True Range (ATR) are used for risk management.

- **Main Parts of the Strategy Class Code (`DonchianBreakoutStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('donchian_period', 20),     # Donchian channel period
    ('adx_period', 14),          # ADX period (not fully utilized
in this specific next method snippet, but defined)
    ('adx_threshold', 25),       # ADX threshold for trend
strength (not fully utilized)
    ('roc_period', 14),          # Rate of Change period (not
fully utilized)
    ('roc_threshold', 2.0),      # ROC threshold (2% minimum
momentum) (not fully utilized)
    ('ma_period', 50),           # Moving average for higher
timeframe trend
    ('atr_period', 14),          # ATR period for trailing stops
    ('atr_multiplier', 2.0),     # ATR multiplier for trailing
stops
    ('printlog', True),
)
```

        - `donchian_period`: The number of bars used to calculate the highest high and lowest low that define the Donchian Channel.
        - `ma_period`: The period for the Simple Moving Average (SMA), used as a filter to confirm the broader trend direction.
        - `atr_period`: The period for calculating the Average True Range (ATR), which is a measure of volatility used to set the trailing stop distance.
        - `atr_multiplier`: A multiplier applied to the ATR value to determine the actual distance of the trailing stop.

- `printlog`: A boolean flag to control whether logging messages are printed.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Skip if order is pending
    if self.order:
        return

    # Handle trailing stops for existing positions
    if self.position:
        if not self.trail_order:
            if self.position.size > 0:
                self.log(f"Placing ATR trailing stop for long
position")
                self.trail_order = self.sell(
                    exectype=bt.Order.StopTrail,
                    trailamount=self.atr[0] *
self.params.atr_multiplier)
            elif self.position.size < 0:
                self.log(f"Placing ATR trailing stop for short
position")
                self.trail_order = self.buy(
                    exectype=bt.Order.StopTrail,
                    trailamount=self.atr[0] *
self.params.atr_multiplier)
        return

    # Ensure sufficient data
    if len(self) < max(self.params.donchian_period,
self.params.ma_period):
        return

    # Donchian Channel breakout signals (SIMPLIFIED)
    long_breakout = self.datahigh[0] > self.donchian_high[-1]
    short_breakout = self.datalow[0] < self.donchian_low[-1]

    # Just basic trend direction - no other filters!
    trend_up = self.dataclose[0] > self.ma[0]
    trend_down = self.dataclose[0] < self.ma[0]

    # SIMPLE entry conditions
    long_signal = long_breakout and trend_up
    short_signal = short_breakout and trend_down

    if long_signal:
        self.log(f"LONG breakout signal at
```

```
{self.dataclose[0]:.2f}")
            self.log(f"High: {self.datahigh[0]:.2f} > Donchian:
{self.donchian_high[-1]:.2f}")

            self.cancel_trail()
            if self.position and self.position.size < 0:
                self.order = self.buy()  # Close short and go long
            elif not self.position:
                self.order = self.buy()

        elif short_signal:
            self.log(f"SHORT breakout signal at
{self.dataclose[0]:.2f}")
            self.log(f"Low: {self.datalow[0]:.2f} < Donchian:
{self.donchian_low[-1]:.2f}")

            self.cancel_trail()
            if self.position and self.position.size > 0:
                self.order = self.sell()  # Close long and go short
            elif not self.position:
                self.order = self.sell()
```

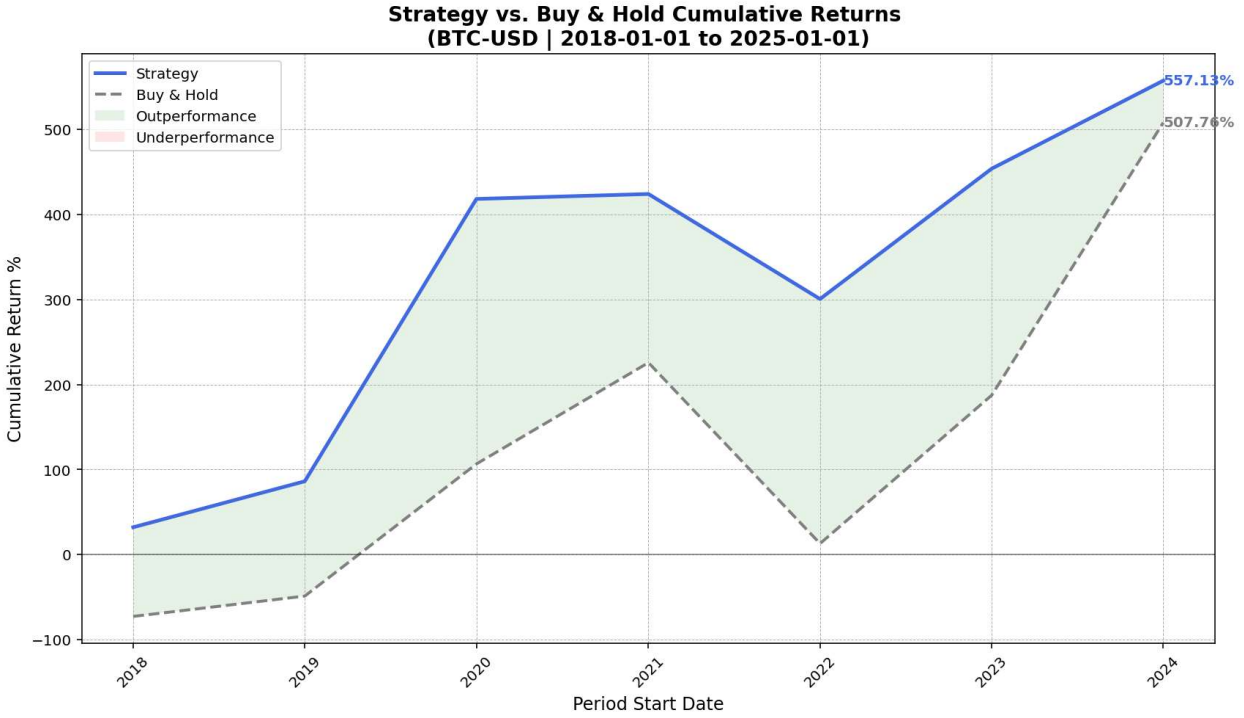
- **Order Check and Trailing Stop Management**: The method first checks for any pending orders (`self.order`). If the strategy is currently in a position and a trailing stop order (`self.trail_order`) hasn't been placed yet, it creates one using `bt.Order.StopTrail`, with the `trailamount` based on the current ATR and `atr_multiplier`.
- **Data Sufficiency**: Ensures there is enough historical data for all indicators to be calculated accurately.
- **Breakout Signals**:
  - `long_breakout`: True if the current bar's high (`self.datahigh[0]`) is greater than the highest high recorded over the `donchian_period` from the *previous* bar (`self.donchian_high[-1]`).
  - `short_breakout`: True if the current bar's low (`self.datalow[0]`) is less than the lowest low recorded over the `donchian_period` from the *previous* bar (`self.donchian_low[-1]`).
- **Trend Filter**: `trend_up` is true if the current closing price (`self.dataclose[0]`) is above the Simple Moving Average (`self.ma[0]`), indicating an uptrend. `trend_down` is true if below, indicating a downtrend.
- **Entry Conditions**:
  - `long_signal`: A buy signal is generated if both `long_breakout` and `trend_up` are true.

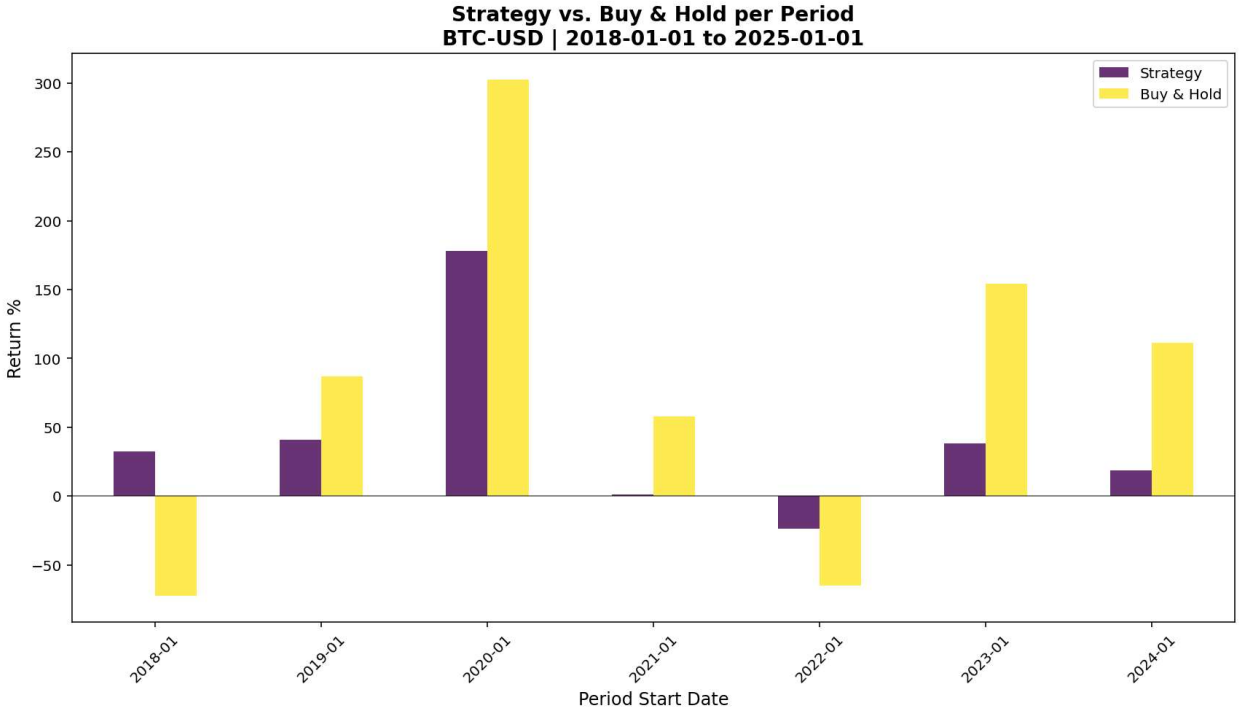
- `short_signal`: A sell signal is generated if both `short_breakout` and `trend_down` are true.

- **Trade Execution**: If a `long_signal` or `short_signal` is active:
  - Any existing trailing stop is immediately canceled (`self.cancel_trail()`) to prevent interference with the new entry.
  - If the strategy is currently in an *opposing* position (e.g., short when a long signal appears), it first closes that position.
  - Finally, the new `buy()` or `sell()` order is placed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	40.82
Median Return %	32.18
Std Dev %	59.95
Win Rate %	85.71
Sharpe Ratio	0.68
Min Return %	-23.56
Max Return %	178.18


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
557.13%
507.76%
Cumulative Return %
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
Period Start Date

Heikin Ashi Trend Strategy

- **Logic and Idea**: This strategy leverages **Heikin-Ashi candles**, which smooth out price data and make trends easier to identify compared to traditional candlesticks. The core idea is to enter a trade when a strong trend is confirmed by a sequence of specific Heikin-Ashi candles (e.g., consecutive green candles with no lower wick for an uptrend). A **trailing stop-loss** is used for risk management, allowing profits to run while limiting downside.

- **Main Parts of the Strategy Class Code (`HeikinAshiTrendStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

      ```
      params = (
          ('consecutive_candles', 3), # Number of strong HA candles
      needed for entry
          ('trail_percent', 0.02),   # Trailing stop
      )
      ```

        - `consecutive_candles`: The minimum number of consecutive Heikin-Ashi candles that must show a strong trend confirmation (e.g., strong green for long, strong red for short) before an entry signal is generated.
        - `trail_percent`: The percentage used for the trailing stop-loss order, set once a position is opened.
    - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

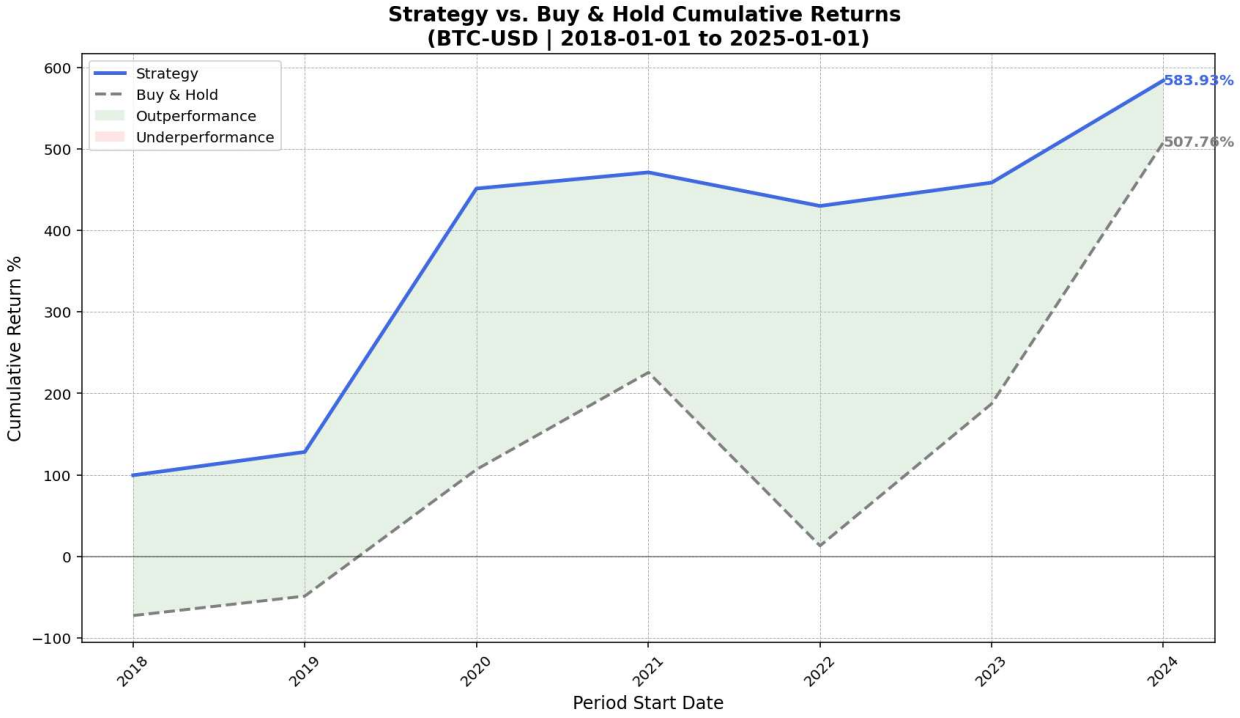
      ```
      def next(self):
          # Make sure we have enough bars to check the sequence
          if len(self) < self.p.consecutive_candles:
              return

          if self.order:
              return

          # --- Entry Logic ---
          if not self.position:
              # --- Check for a BUY Signal ---
              is_buy_signal = True
              # Loop backwards from the current candle (i=0) to check
      the sequence
              for i in range(self.p.consecutive_candles):
                  # CORRECTED: Perform the check on the historical data
      inside the loop
                  is_green_past = self.ha.ha_close[-i] >
      self.ha.ha_open[-i]
                  has_no_lower_wick_past = self.ha.ha_open[-i] ==
      ```

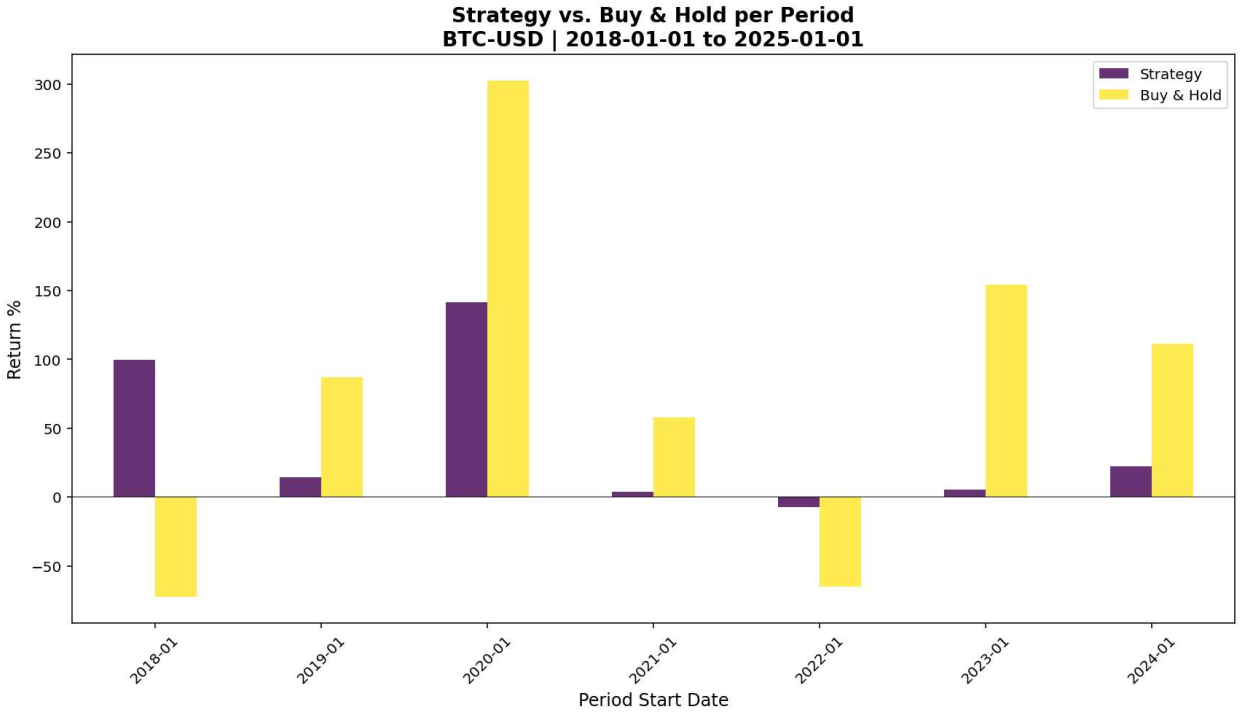
```
self.ha.ha_low[-i]

            if not (is_green_past and has_no_lower_wick_past):
                is_buy_signal = False
                break  # If one candle fails, the sequence is
broken

        if is_buy_signal:
            self.order = self.buy()
            return  # Exit to avoid checking for a sell signal on
the same bar

        # --- Check for a SELL Signal ---
        is_sell_signal = True
        # Loop backwards to check the sequence
        for i in range(self.p.consecutive_candles):
            # CORRECTED: Perform the check on the historical data
inside the loop
            is_red_past = self.ha.ha_close[-i] <
self.ha.ha_open[-i]
            has_no_upper_wick_past = self.ha.ha_open[-i] ==
self.ha.ha_high[-i]

            if not (is_red_past and has_no_upper_wick_past):
                is_sell_signal = False
                break  # If one candle fails, the sequence is
broken

        if is_sell_signal:
            self.order = self.sell()
```

- **Data Sufficiency and Order Check**: The method first checks if there are enough historical bars available to evaluate the `consecutive_candles` sequence and ensures no orders are currently pending.
- **Buy Signal Logic (`is_buy_signal`)**: If the strategy is not currently in a position, it attempts to detect a buy signal. It loops backward through the last `consecutive_candles`. For each candle, it checks if it's a "strong green" Heikin-Ashi candle (where `ha_close` is greater than `ha_open` and `ha_open` is equal to `ha_low`, indicating a strong bullish candle with no lower wick). If *all* candles in the sequence meet these criteria, `is_buy_signal` remains `True`, and a `buy()` order is placed.
- **Sell Signal Logic (`is_sell_signal`)**: Similarly, if not in a position, it checks for a sequence of "strong red" Heikin-Ashi candles (where `ha_close` is less than `ha_open` and `ha_open` is equal to `ha_high`, indicating a strong bearish candle with no upper wick). If all candles in the sequence confirm this pattern, a `sell()` order is placed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	39.97
Median Return %	14.34
Std Dev %	52.91
Win Rate %	85.71
Sharpe Ratio	0.76
Min Return %	-7.22
Max Return %	141.65





Ichimoku Cloud Strategy

- **Logic and Idea**: The **Ichimoku Kinko Hyo** (Ichimoku Cloud) is a comprehensive indicator that provides insights into trend direction, momentum, support, and resistance levels. This strategy focuses on confirmed breakouts from the "Kumo" (cloud), validated by the Tenkan-sen/Kijun-sen cross (momentum) and the Chikou Span (lagging span, for trend confirmation). The idea is to enter only when multiple components of the Ichimoku system align to confirm a strong trend. A **trailing stop** is used for risk management.

- **Main Parts of the Strategy Class Code (`IchimokuCloudStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # Default Ichimoku parameters
    ('tenkan', 7),
    ('kijun', 14),
    ('senkou', 30),
    ('senkou_lead', 14),  # How far forward to plot the cloud
    ('chikou', 14),       # How far back to plot the lagging span
    # Strategy parameters
    ('trail_percent', 0.02), # Trailing stop loss of 4%
)
```

        - `tenkan`: Period for the Tenkan-sen (conversion line).
        - `kijun`: Period for the Kijun-sen (base line).

- `senkou`: Period for the Senkou Span B (leading span B).
- `senkou_lead`: How many periods to project the Kumo cloud forward.
- `chikou`: How many periods to shift the Chikou Span (lagging span) backward.
- `trail_percent`: The percentage for the trailing stop-loss order.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

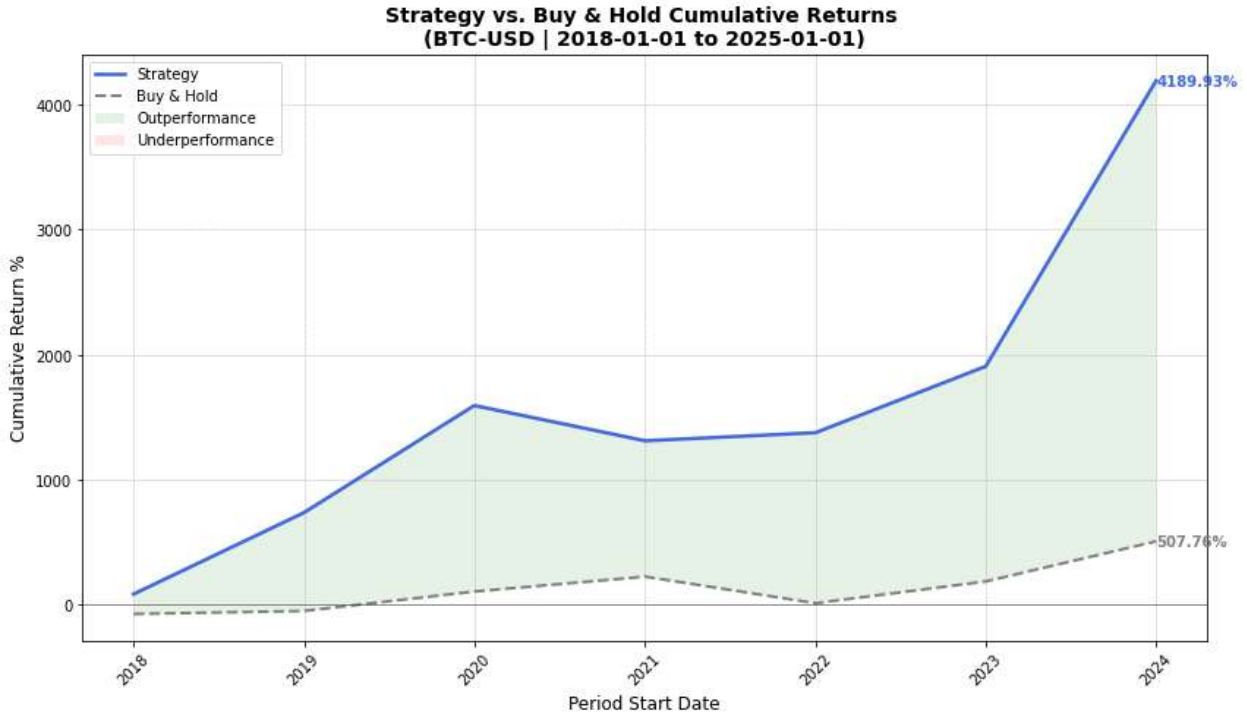
```
def next(self):
    # Check for pending orders
    if self.order:
        return

    # Check if we are in a position
    if not self.position:
        # --- Bullish Entry Conditions ---
        # 1. Price is above both lines of the Kumo cloud
        is_above_cloud = (self.data.close[0] >
self.ichimoku.senkou_span_a[0] and
                          self.data.close[0] >
self.ichimoku.senkou_span_b[0])

        # 2. Tenkan-sen is above Kijun-sen
        is_tk_cross_bullish = self.ichimoku.tenkan_sen[0] >
self.ichimoku.kijun_sen[0]

        # 3. Chikou Span is above the price from 26 periods ago
        is_chikou_bullish = self.ichimoku.chikou_span[0] >
self.data.high[-self.p.chikou]

        if is_above_cloud and is_tk_cross_bullish and
is_chikou_bullish:
            self.order = self.buy()

        # --- Bearish Entry Conditions ---
        # 1. Price is below both lines of the Kumo cloud
        is_below_cloud = (self.data.close[0] <
self.ichimoku.senkou_span_a[0] and
                          self.data.close[0] <
self.ichimoku.senkou_span_b[0])

        # 2. Tenkan-sen is below Kijun-sen
        is_tk_cross_bearish = self.ichimoku.tenkan_sen[0] <
self.ichimoku.kijun_sen[0]

        # 3. Chikou Span is below the price from 26 periods ago
        is_chikou_bearish = self.ichimoku.chikou_span[0] <
```

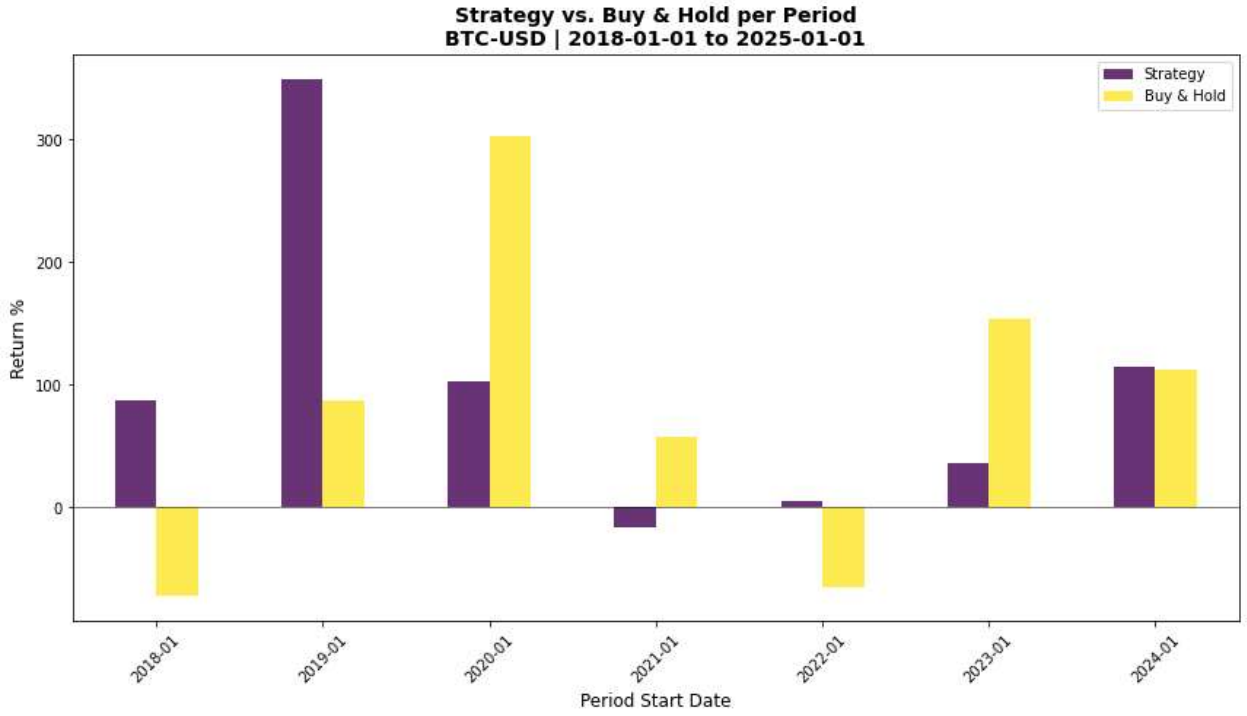
```
self.data.low[-self.p.chikou]

        if is_below_cloud and is_tk_cross_bearish and
is_chikou_bearish:
            self.order = self.sell()
```

- **Order Check**: Ensures no orders are pending (`self.order`).
- **Bullish Entry Conditions**: If not currently in a position, the strategy checks for a strong bullish signal from the Ichimoku Cloud by verifying three conditions:
  1. `is_above_cloud`: The current closing price is above both the Senkou Span A and Senkou Span B lines (meaning price is above the "cloud" formation).
  2. `is_tk_cross_bullish`: The Tenkan-sen (fast line) is above the Kijun-sen (slow line), indicating bullish momentum.
  3. `is_chikou_bullish`: The Chikou Span (lagging span) is above the price of `chikou` periods ago, confirming the trend. If all three conditions are met, a `buy()` order is placed.
- **Bearish Entry Conditions**: Similarly, it checks for a strong bearish signal:
  1. `is_below_cloud`: The current closing price is below both Senkou Span A and Senkou Span B.
  2. `is_tk_cross_bearish`: The Tenkan-sen is below the Kijun-sen.
  3. `is_chikou_bearish`: The Chikou Span is below the price of `chikou` periods ago. If all three conditions are met, a `sell()` order is placed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	96.50
Median Return %	86.36
Std Dev %	112.88
Win Rate %	85.71
Sharpe Ratio	0.85
Min Return %	-16.65
Max Return %	349.15





Keltner Channel Breakout Strategy

- **Logic and Idea**: This strategy aims to capture trends by trading breakouts from **Keltner Channels**. Keltner Channels are volatility-based envelopes around a moving average, using Average True Range (ATR) to define their width. The idea is that a strong price move that breaks out of these channels indicates the start of a

new trend. Trades are exited when the price crosses back over the channel's centerline (EMA).

- **Main Parts of the Strategy Class Code (`KeltnerBreakoutStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('ema_period', 30),
    ('atr_period', 7),
    ('atr_multiplier', 1.0),
    ('printlog', True),
)
```

    - `ema_period`: The period for the Exponential Moving Average (EMA) used as the centerline of the Keltner Channel.
    - `atr_period`: The period for calculating the Average True Range (ATR), which measures volatility and is used to set the width of the channel bands.
    - `atr_multiplier`: A multiplier applied to the ATR value to determine the distance of the upper and lower bands from the centerline.
    - `printlog`: A boolean flag to enable or disable detailed logging during the backtest.
  - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Skip if we don't have enough data
    if len(self.data) < max(self.params.ema_period,
self.params.atr_period):
        return

    # Check if we have a pending order
    if self.order:
        return

    # Get previous day's values for signal generation
    if len(self.data) < 2:
        return

    prev_close = self.dataclose[-1]
    prev_upper = self.keltner.top[-1]
    prev_lower = self.keltner.bot[-1]
    current_ema = self.keltner.mid[0]
    current_close = self.dataclose[0]

    if not self.position:  # Not in market
        # Long entry: Previous close > Previous upper band
```

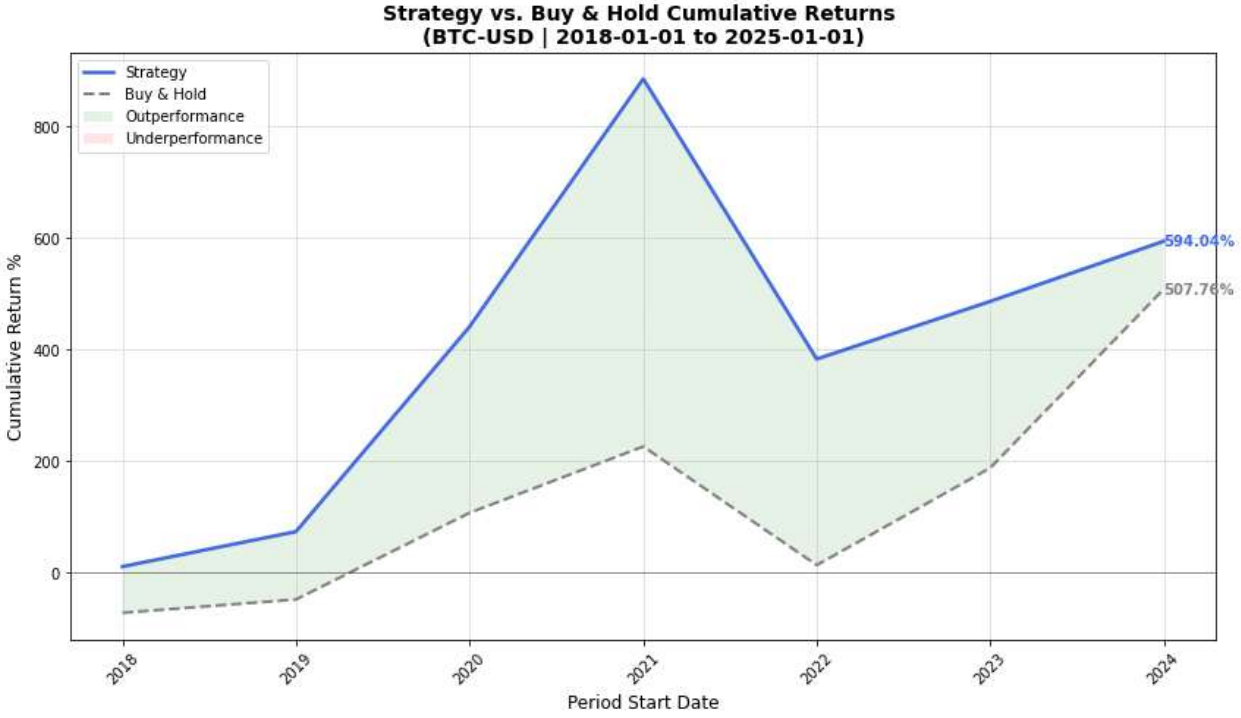
```
            if prev_close > prev_upper:
                self.log(f'BUY CREATE: Price {self.dataopen[0]:.2f}
(Prev Close: {prev_close:.2f} > Upper: {prev_upper:.2f})')
                self.order = self.buy()

            # Short entry: Previous close < Previous lower band
            elif prev_close < prev_lower:
                self.log(f'SELL CREATE: Price {self.dataopen[0]:.2f}
(Prev Close: {prev_close:.2f} < Lower: {prev_lower:.2f})')
                self.order = self.sell()

        else:  # In market
            # Exit conditions based on current close vs EMA
            if self.position.size > 0:  # Long position
                if current_close < current_ema:
                    self.log(f'CLOSE LONG: Price {current_close:.2f}
< EMA {current_ema:.2f}')
                    self.order = self.close()

            elif self.position.size < 0:  # Short position
                if current_close > current_ema:
                    self.log(f'CLOSE SHORT: Price {current_close:.2f}
> EMA {current_ema:.2f}')
                    self.order = self.close()
```

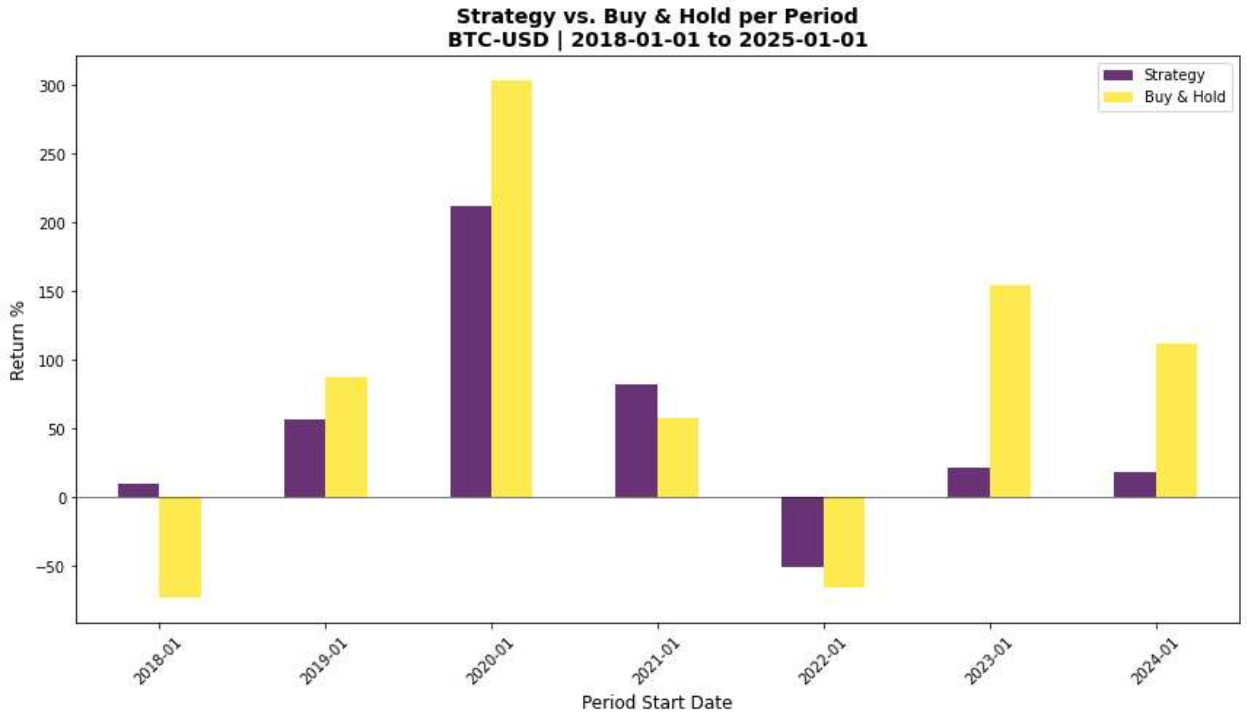
- **Data Sufficiency and Order Check**: The method first checks if there's enough historical data for the indicators and if any orders are pending. It also specifically checks for at least 2 bars to access `[-1]` (previous bar's data).
- **Value Retrieval**: It retrieves the previous day's closing price (`prev_close`), upper Keltner band (`prev_upper`), and lower Keltner band (`prev_lower`). It also gets the current bar's EMA midline (`current_ema`) and closing price (`current_close`).
- **Entry Logic (if no position)**:
  - **Long Entry**: If the `prev_close` was greater than the `prev_upper` band, signifying a strong bullish breakout on the previous bar, a `buy()` order is placed.
  - **Short Entry**: If the `prev_close` was less than the `prev_lower` band, indicating a strong bearish breakout, a `sell()` order is placed.
- **Exit Logic (if in position)**:
  - **Long Exit**: If currently in a long position, and the `current_close` falls below the `current_ema` (Keltner midline), the position is closed. This acts as a mean-reversion exit if the trend reverses back towards the average.

- **Short Exit**: If in a short position, and the `current_close` rises above the `current_ema`, the position is closed for the same reason.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	50.09
Median Return %	21.43
Std Dev %	76.54
Win Rate %	85.71
Sharpe Ratio	0.65
Min Return %	-51.02
Max Return %	212.23





MA Ribbon Pullback Strategy

- **Logic and Idea**: This strategy is a sophisticated trend-following approach that uses a "**ribbon**" of multiple Exponential Moving Averages (EMAs) to identify strong trends. The core idea is to enter a trade when the price "**pulls back**" to the fast end of an expanding MA ribbon in the direction of the trend. The **expansion of the ribbon** (faster EMAs fanning out above slower EMAs) and the **positive slope of the slowest EMA** confirm the trend strength. An EMA crossover of separate, faster and slower EMAs is used as an exit signal.

- **Main Parts of the Strategy Class Code (`MaRibbonPullbackStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # Define the periods for the ribbon EMAs
    ('ema_periods', (5, 8, 11, 14, 17, 20)),
    ('slope_period', 7), # Period to calculate slope of the
slowest EMA
    ('exit_ema_cross_short', 7), # Faster EMA for exit crossover
    ('exit_ema_cross_long', 30), # Slower EMA for exit crossover
    ('order_percentage', 0.95),
    ('ticker', 'BTC-USD'),
    ('min_slope_threshold', 0.01) # Minimum upward slope for
slowest EMA to consider trend 'up'
                                  # Adjust based on asset
volatility and timeframe
)
```

- `ema_periods`: A tuple of integers defining the periods for the Exponential Moving Averages that constitute the "ribbon." These are typically increasing periods to show the fanning out of the trend.
- `slope_period`: The period over which the slope of the slowest EMA is calculated. A positive slope indicates an uptrend.
- `exit_ema_cross_short`, `exit_ema_cross_long`: Periods for two distinct EMAs used for an exit crossover signal.
- `order_percentage`: The percentage of available capital to allocate to a trade.
- `min_slope_threshold`: The minimum positive value the slowest EMA's slope must exceed to confirm a strong upward trend. This filters out weak or sideways trends.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

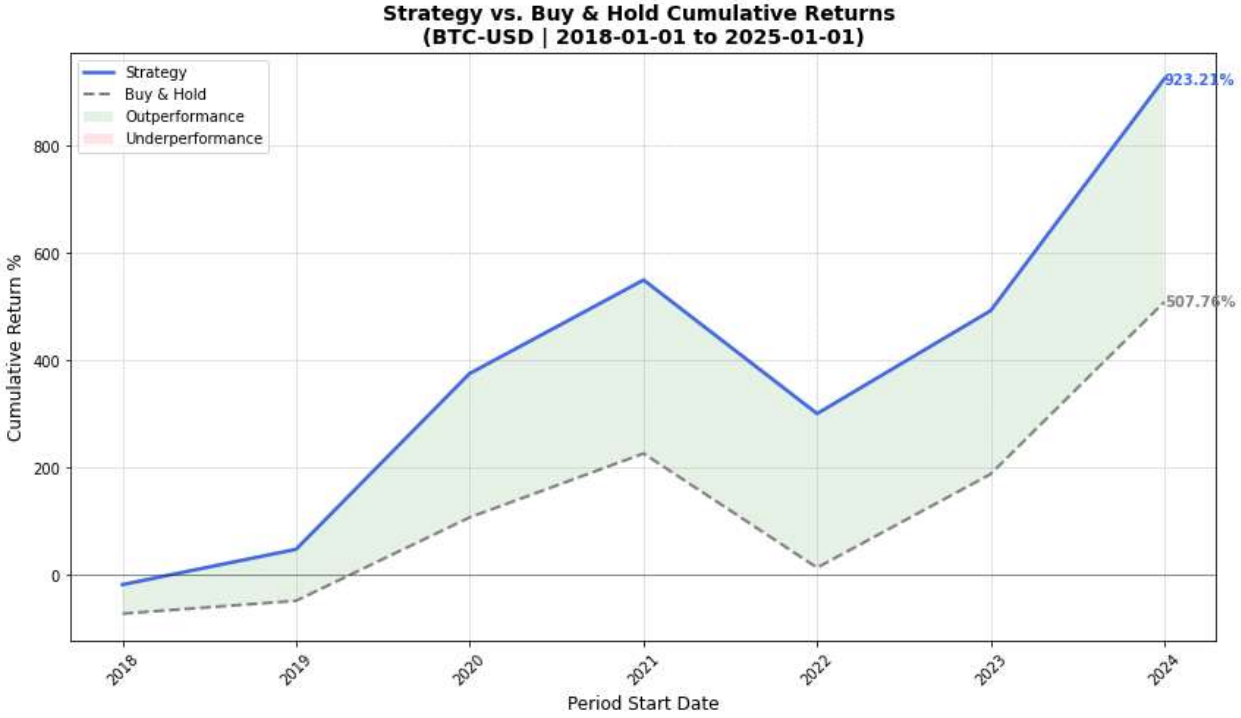
```
def next(self):
     # Check if indicators have enough data
    if len(self.data_close) < max(self.params.ema_periods) +
self.params.slope_period:
         return

    # Define expansion state (simplified)
    # 1. Fastest EMA is above slowest EMA
    # 2. Slowest EMA slope is positive and above threshold
    is_expanding_up = (self.ema_fastest[0] > self.ema_slowest[0]
and
                       self.slowest_ema_slope[0] >
self.params.min_slope_threshold)

    # Check for pullback touch (using low price)
    # Price low touches or goes slightly below the fastest EMA
    pullback_touch = self.data_low[0] <= self.ema_fastest[0]

    # --- Entry Logic ---
    if not self.position:
        if is_expanding_up and pullback_touch:
            self.log(f'BUY SIGNAL (Pullback):
Close={self.data_close[0]:.2f}, Low={self.data_low[0]:.2f},
FastEMA={self.ema_fastest[0]:.2f},
Slope={self.slowest_ema_slope[0]:.3f}, Current
ATR={self.atr[0]:.2f}') # Added ATR
            cash = self.broker.get_cash()
            size = (cash * self.params.order_percentage) /
self.data_close[0]
            self.log(f'Calculating Buy Size: Cash={cash:.2f},
Close={self.data_close[0]:.2f},
Percentage={self.params.order_percentage}, Size={size:.6f}')
```

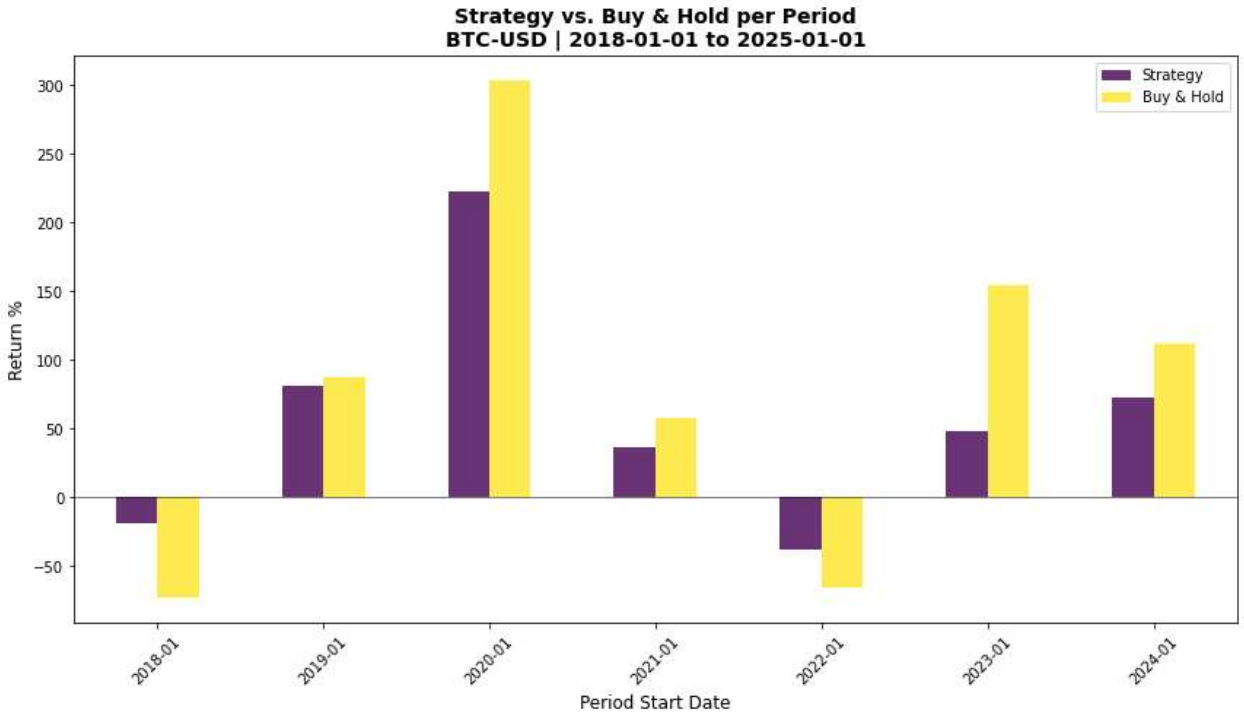
```
                self.order = self.buy(size=size)
            # (Optional: Add short entry logic for downward expansion
and pullback to resistance)

        # --- Exit Logic ---
        else: # We are in a long position
            # Exit if the faster exit EMA crosses below the slower
exit EMA
            if self.exit_crossover < 0:
                self.log(f'SELL SIGNAL (Exit - EMA Cross):
Close={self.data_close[0]:.2f}')
                self.order = self.close()
```

- **Data Sufficiency**: Ensures that enough historical data points are available for all moving averages and slope calculations to be valid.
- **Trend Expansion Check (`is_expanding_up`)**: This condition confirms a strong upward trend. It checks two criteria:
    1. The fastest EMA (`self.ema_fastest[0]`) is currently above the slowest EMA (`self.ema_slowest[0]`), indicating that the ribbon is fanning out upwards.
    2. The slope of the slowest EMA (`self.slowest_ema_slope[0]`) is positive and exceeds the `min_slope_threshold`, confirming a sustained upward momentum.
- **Pullback Detection (`pullback_touch`)**: This condition identifies a temporary dip in price during an uptrend. It checks if the current bar's low price (`self.data_low[0]`) is less than or equal to the fastest EMA (`self.ema_fastest[0]`). This signifies that price has pulled back to a short-term support level (the fastest EMA).
- **Entry Logic**: If the strategy is not currently in a position (`not self.position`), and both `is_expanding_up` and `pullback_touch` are true, a `buy()` order is placed. The trade size is calculated based on a percentage of the available cash.
- **Exit Logic**: If the strategy is currently in a long position (`else:` block), it looks for an exit signal. If the `exit_crossover` indicator shows a bearish cross (the faster exit EMA crosses below the slower exit EMA, indicated by a value less than 0), the current long position is closed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	5
Losing Periods	2
Mean Return %	57.67
Median Return %	47.95
Std Dev %	78.74
Win Rate %	71.43
Sharpe Ratio	0.73
Min Return %	-38.36
Max Return %	222.23





PSAR Trend Filter Strategy

- **Logic and Idea**: This strategy combines a long-term **Simple Moving Average (SMA)** as a trend filter with the **Parabolic SAR (Stop and Reverse)** indicator for entry signals. The SMA determines the overall market regime (long-only or short-only). Within that regime, the PSAR provides precise entry signals when it flips direction, indicating a potential trend continuation. A **trailing stop-loss** is used for risk management.

- **Main Parts of the Strategy Class Code (`PsarTrendFilterStrategy`)**:

  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # Trend filter parameters
    ('ma_period', 30),
    # Parabolic SAR parameters (standard defaults)
    ('psar_af', 0.01),
    ('psar_afmax', 0.1),
    # Exit management
    ('trail_percent', 0.02),  # Trailing stop of 2%
)
```

    - `ma_period`: The period for the Simple Moving Average (SMA), which acts as a long-term trend filter. Trades are only allowed in the direction of this SMA.

- `psar_af`: The initial acceleration factor for the Parabolic SAR (PSAR) indicator.
- `psar_afmax`: The maximum acceleration factor for PSAR. The acceleration factor increases as the trend progresses, making the PSAR line track closer to the price.
- `trail_percent`: The percentage used for the trailing stop-loss order placed after an entry.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

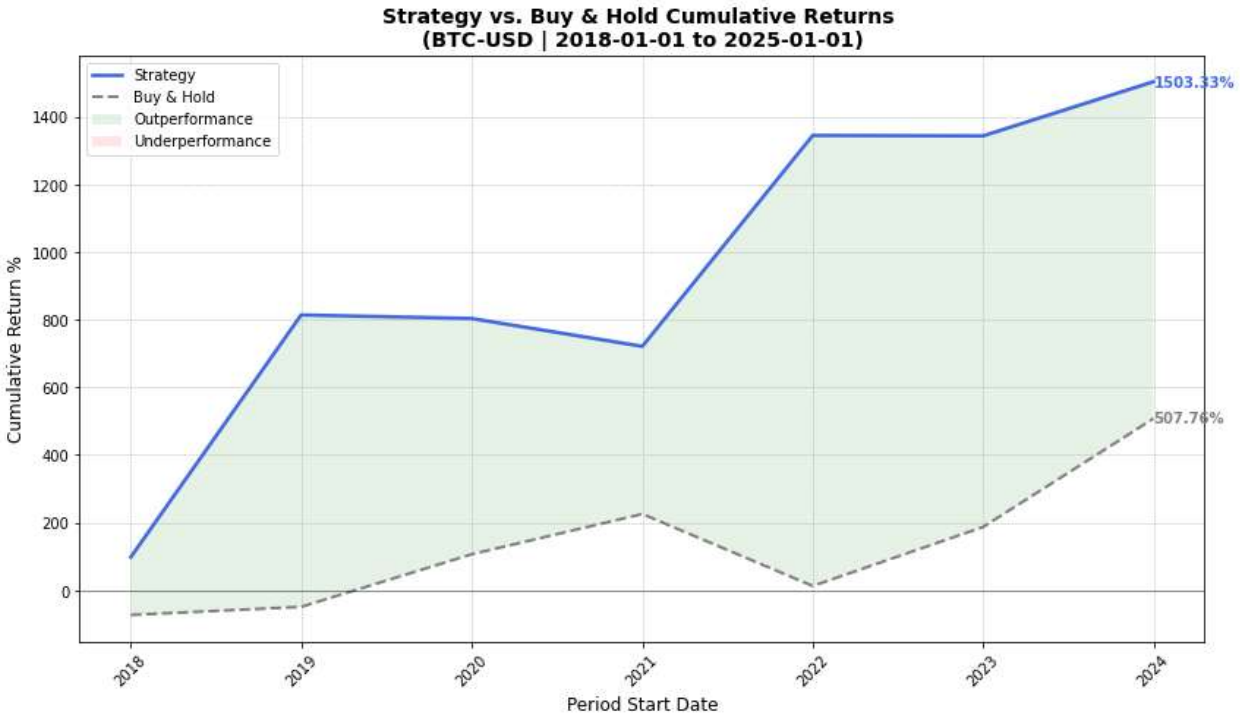
```
def next(self):
    # Check for pending orders
    if self.order:
        return

    # --- Entry Logic ---
    if not self.position: # Only consider new entries if not
currently in a position.
        # --- Long-Only Regime ---
        # If current closing price is above the SMA, it's
considered an uptrend.
        if self.data.close[0] > self.sma[0]:
            # A buy signal occurs when the price crosses ABOVE
the PSAR, indicated by psar_cross > 0.
            if self.psar_cross[0] > 0.0:
                self.order = self.buy() # Place a buy order

        # --- Short-Only Regime ---
        # If current closing price is below the SMA, it's
considered a downtrend.
        elif self.data.close[0] < self.sma[0]:
            # A sell signal occurs when the price crosses BELOW
the PSAR, indicated by psar_cross < 0.
            if self.psar_cross[0] < 0.0:
                self.order = self.sell() # Place a sell order
```

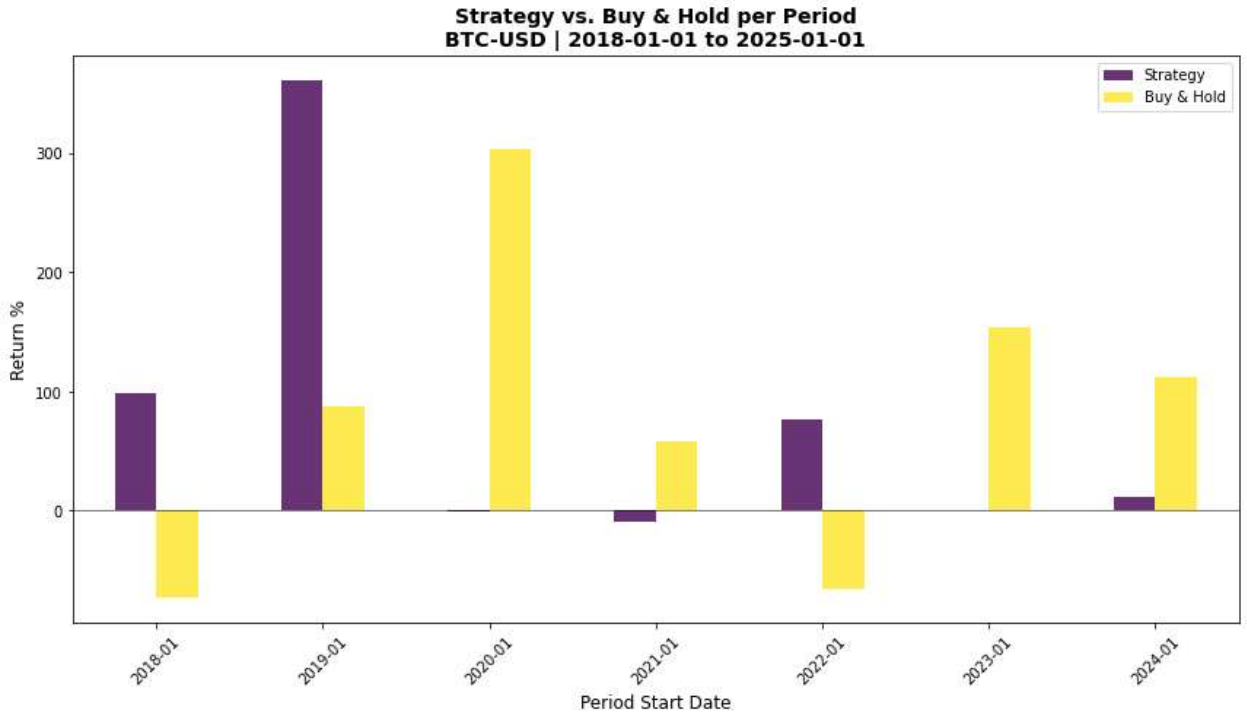
- **Order Check**: The method first checks for any pending orders (`self.order`) and returns if one exists.
- **Entry Logic (No Position)**: If the strategy is not currently in an open position (`not self.position`):
  - **Long-Only Regime**: If the current closing price (`self.data.close[0]`) is above the Simple Moving Average (`self.sma[0]`), the strategy is in a bullish regime. A `buy()` order is placed if the `self.psar_cross[0]` value is greater than 0, indicating that the price has just crossed *above* the PSAR line (a bullish flip of the PSAR).

- **Short-Only Regime**: If the current closing price is below the SMA, the strategy is in a bearish regime. A `sell()` order is placed if `self.psar_cross[0]` is less than 0, indicating that the price has just crossed *below* the PSAR line (a bearish flip of the PSAR).

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	4
Losing Periods	3
Mean Return %	76.52
Median Return %	11.09
Std Dev %	122.28
Win Rate %	57.14
Sharpe Ratio	0.63
Min Return %	-9.11
Max Return %	360.55





Regime Filtered Trend Strategy

- **Logic and Idea**: This sophisticated strategy adapts its trading style by classifying the current market into "**trending**" or "**ranging**" regimes. It uses a combination of indicators (ADX, Bollinger Band Width, Volatility, MA separation) to determine the regime with a degree of confidence. Trend-following signals (MA crossovers) are only acted upon when the market is confirmed to be in a trending regime. Position sizing and stop-loss multipliers are also adjusted based on the detected regime.

- **Main Parts of the Strategy Class Code (`RegimeFilteredTrendStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('ma_fast', 20),            # Fast moving average period
    ('ma_slow', 50),            # Slow moving average period
    ('adx_period', 14),         # ADX period
    ('adx_trending_threshold', 25), # ADX threshold for trending
regime
    ('bb_period', 20),          # Bollinger Bands period
    ('bb_width_threshold', 0.03), # BB width threshold for
trending (3% of mid-band)
    ('volatility_lookback', 20), # Volatility measurement period
for regime classification
    ('vol_trending_threshold', 0.025), # Normalized volatility
threshold for trending (2.5% of price)
    ('atr_period', 14),         # ATR period for stop-loss
calculation
```

```
    ('trail_atr_mult', 2.0),    # Trailing stop multiplier (for
trending regime)
    ('range_atr_mult', 1.5),    # Trailing stop multiplier (for
ranging regime)
    ('max_position_pct', 0.95), # Maximum percentage of cash to
risk per trade
    ('min_position_pct', 0.20), # Minimum percentage of cash to
risk per trade
    ('regime_confirmation', 3), # Number of consecutive bars
required to confirm a regime change
)
```

- `ma_fast`, `ma_slow`: Periods for the fast and slow moving averages used for trend-following signals.
- `adx_period`, `adx_trending_threshold`: Parameters for the ADX indicator, used to identify strong trends.
- `bb_period`, `bb_width_threshold`: Parameters for Bollinger Bands, where a wide band width can indicate trending.
- `volatility_lookback`, `vol_trending_threshold`: Parameters for calculating and classifying volatility (ATR/Close) to identify trending environments.
- `atr_period`, `trail_atr_mult`, `range_atr_mult`: Parameters for Average True Range (ATR) and its multipliers, which adapt stop-loss distances based on the detected market regime.
- `max_position_pct`, `min_position_pct`: Define the upper and lower bounds for dynamically adjusted position sizing.
- `regime_confirmation`: The number of consecutive bars that must confirm a new regime before it is considered established.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Skip if order is pending
    if self.order:
        return

    # Update volatility tracking
    current_vol = self.calculate_volatility()
    if current_vol > 0:
        self.volatility_history.append(current_vol)
        if len(self.volatility_history) >
self.params.volatility_lookback:
            self.volatility_history = self.volatility_history[-
self.params.volatility_lookback:]

    # Update regime classification
```

```
        self.update_regime_state()

        # Handle existing positions with adaptive stops
        if self.position:
            if not self.trail_order:
                stop_multiplier = self.get_adaptive_stop_multiplier()

                if self.position.size > 0:
                    self.trail_order = self.sell(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * stop_multiplier)
                elif self.position.size < 0:
                    self.trail_order = self.buy(
                        exectype=bt.Order.StopTrail,
                        trailamount=self.atr[0] * stop_multiplier)
            return

        # Ensure sufficient data
        required_bars = max(self.params.ma_slow,
    self.params.adx_period, self.params.bb_period)
        if len(self) < required_bars:
            return

        # Check if we should engage in trend following
        if not self.should_trade_trend_following():
            return  # Stay out during non-trending regimes

        # Moving average crossover signals (only in trending regimes)
        ma_bullish_cross = (self.ma_fast[0] > self.ma_slow[0] and
                            self.ma_fast[-1] <= self.ma_slow[-1])
        ma_bearish_cross = (self.ma_fast[0] < self.ma_slow[0] and
                            self.ma_fast[-1] >= self.ma_slow[-1])

        # Position sizing based on regime
        position_size_pct = self.calculate_regime_position_size()
        current_price = self.dataclose[0]

        # LONG ENTRY: MA bullish cross in trending regime
        if ma_bullish_cross and not self.position:
            self.cancel_trail()

            # Calculate position size
            cash = self.broker.getcash()
            target_value = cash * position_size_pct
            shares = target_value / current_price

            self.order = self.buy(size=shares)
```

```
        # SHORT ENTRY: MA bearish cross in trending regime
        elif ma_bearish_cross and not self.position:
            self.cancel_trail()

            # Calculate position size
            cash = self.broker.getcash()
            target_value = cash * position_size_pct
            shares = target_value / current_price

            self.order = self.sell(size=shares)

        # Alternative entry: Strong directional bias in confirmed
trending regime
        elif (not self.position and self.current_regime == "trending"
and
              self.regime_confidence > 0.8):

            # Strong trend continuation signals
            ma_spread = (self.ma_fast[0] - self.ma_slow[0]) /
self.ma_slow[0]

            if ma_spread > 0.03:  # Strong uptrend (3% MA spread)
                cash = self.broker.getcash()
                target_value = cash * (position_size_pct * 0.7)  #
Smaller position
                shares = target_value / current_price
                self.order = self.buy(size=shares)

            elif ma_spread < -0.03:  # Strong downtrend
                cash = self.broker.getcash()
                target_value = cash * (position_size_pct * 0.7)  #
Smaller position
                shares = target_value / current_price
                self.order = self.sell(size=shares)
```

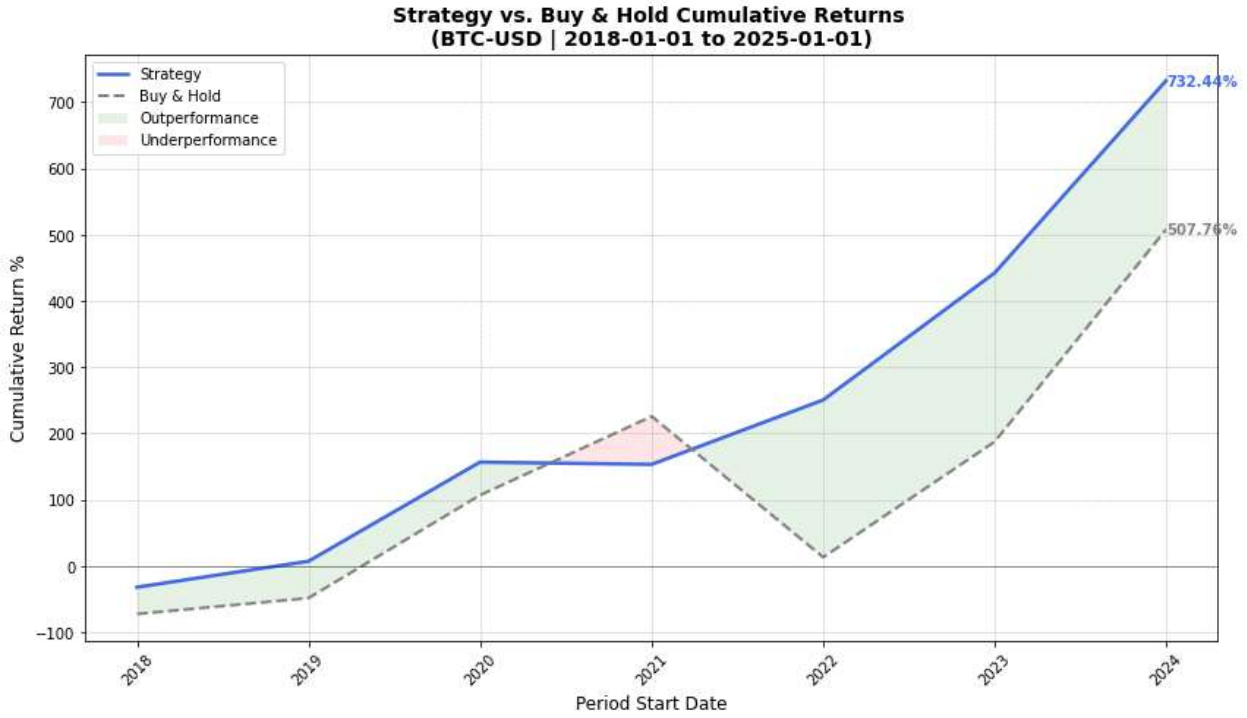
- **Order Check and Volatility/Regime Update**: The method first checks for pending orders. It then updates the internal `volatility_history` and calls `update_regime_state()` to classify the current market regime based on multiple indicators.
- **Adaptive Stop Management**: If the strategy is in an open position, it checks if a trailing stop (`self.trail_order`) is active. If not, it places one using an `adaptive_stop_multiplier` (which varies based on the current regime and volatility).
- **Data Sufficiency and Trend Following Check**: Ensures enough historical data is available. It then calls `should_trade_trend_following()` to determine if the current market

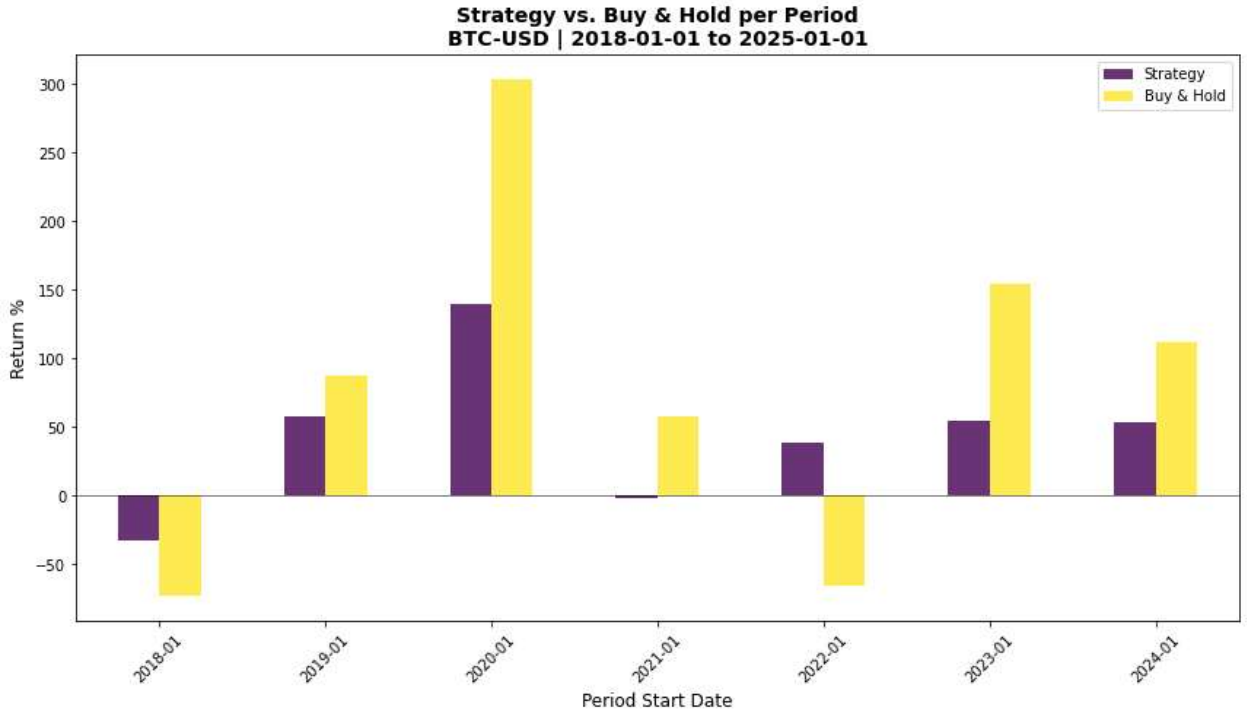
regime is conducive to trend-following (i.e., not ranging or highly uncertain). If not, it returns.

- **Entry Logic (No Position)**: If no position is open and trend-following is allowed:
  - It checks for `ma_bullish_cross` (fast MA crosses above slow MA) or `ma_bearish_cross` (fast MA crosses below slow MA).
  - If a valid crossover occurs, it calculates a dynamic `position_size_pct` using `calculate_regime_position_size()` and places a `buy()` or `sell()` order.
  - An "Alternative Entry" is included for highly confirmed trending regimes (`regime_confidence > 0.8`), which allows for a slightly smaller position size if there's a strong and sustained MA spread, even without a fresh crossover.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	5
Losing Periods	2
Mean Return %	44.39
Median Return %	53.53
Std Dev %	49.97
Win Rate %	71.43
Sharpe Ratio	0.89
Min Return %	-32.19
Max Return %	139.91





SuperTrend Confirmation Strategy

- **Logic and Idea**: This strategy uses the **SuperTrend indicator**, a popular tool for identifying trends and generating signals. The key idea here is to add a "**confirmation**" step: instead of entering immediately when the SuperTrend flips, the strategy waits for the *next* candle to close in the direction of the new trend. This

aims to filter out false signals and improve signal quality. A **trailing stop-loss** is used for risk management.

- **Main Parts of the Strategy Class Code (`SuperTrendConfirmationStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('st_period', 7),
    ('st_multiplier', 2.0),
    ('trail_percent', 0.02),
)
```

    - `st_period`: The period used for calculating the Average True Range (ATR) component of the SuperTrend indicator.
    - `st_multiplier`: The multiplier applied to the ATR to determine the distance of the SuperTrend line from the median price.
    - `trail_percent`: The percentage used for the trailing stop-loss order placed after an entry is completed.
  - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    if self.order: # If there's an existing order pending, do
nothing
        return

    # Determine if current and previous bars are in an uptrend
based on SuperTrend
    is_uptrend = self.data.close[0] > self.st.supertrend[0]
    was_uptrend = self.data.close[-1] > self.st.supertrend[-1]

    # --- Confirmation Logic ---
    # If we were waiting for a buy confirmation from the previous
bar's flip
    if self.waiting_for_buy_confirmation:
        self.waiting_for_buy_confirmation = False # Reset the
flag
        if is_uptrend and not self.position: # If still in
uptrend and no position, confirm and buy
            self.order = self.buy()
            return # Exit to avoid placing multiple orders

    # If we were waiting for a sell confirmation from the
previous bar's flip
    if self.waiting_for_sell_confirmation:
        self.waiting_for_sell_confirmation = False # Reset the
flag
```

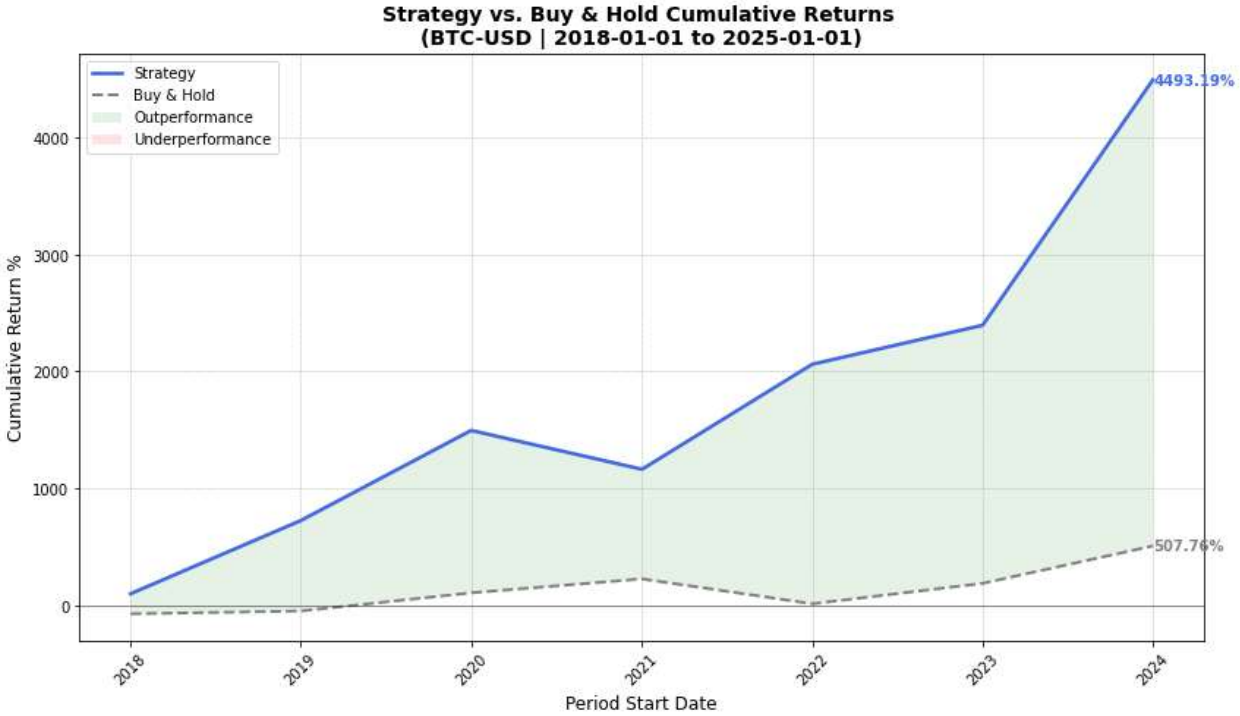
```
            if not is_uptrend and not self.position: # If still in
downtrend and no position, confirm and sell
                self.order = self.sell()
                return # Exit to avoid placing multiple orders

        # --- Flip Detection Logic ---
        # Detects if SuperTrend has just flipped from downtrend to
uptrend
        if is_uptrend and not was_uptrend:
            # Only set the flag if not already waiting for a sell
confirmation
            # (prevents conflicting signals from setting flags
simultaneously)
            if not self.waiting_for_sell_confirmation:
                self.waiting_for_buy_confirmation = True

        # Detects if SuperTrend has just flipped from uptrend to
downtrend
        if not is_uptrend and was_uptrend:
            # Only set the flag if not already waiting for a buy
confirmation
            if not self.waiting_for_buy_confirmation:
                self.waiting_for_sell_confirmation = True
```

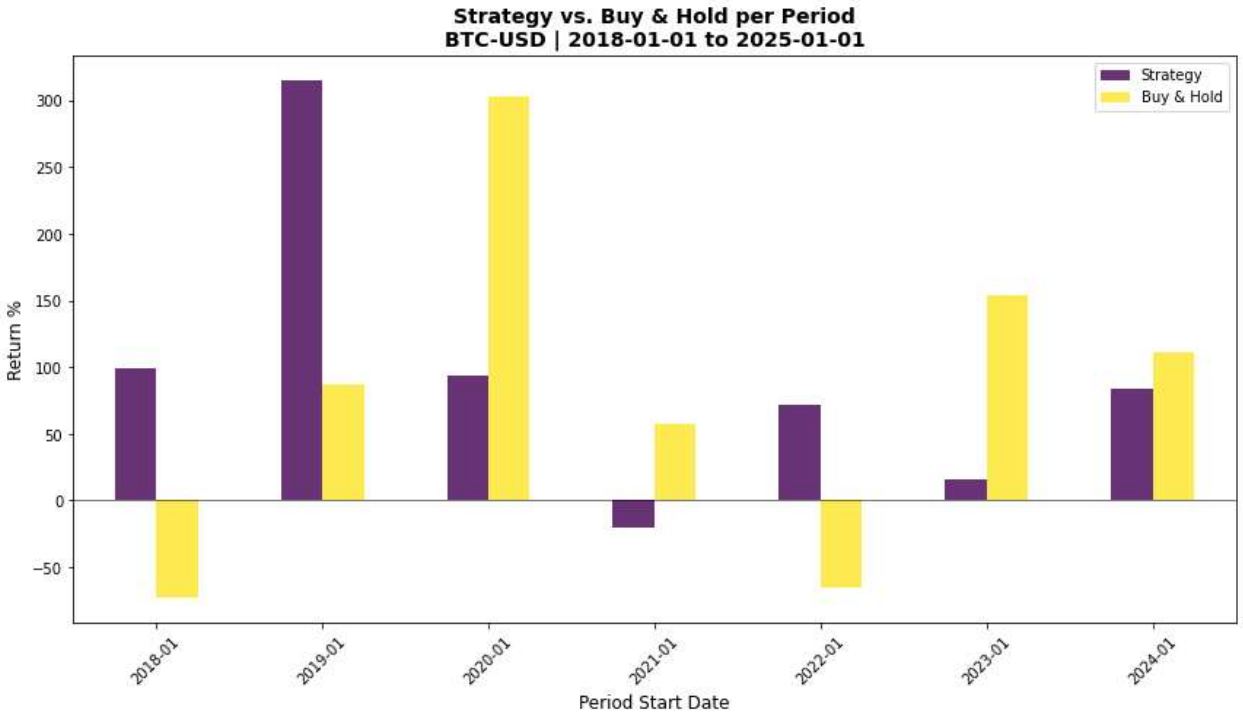
- **Order Check**: The method first checks if an order is pending (`self.order`) and returns if it is, preventing new entries.
- **Trend Status**: `is_uptrend` and `was_uptrend` booleans check if the current and previous closing prices, respectively, were above their corresponding SuperTrend values. This helps detect SuperTrend flips.
- **Confirmation Logic**:
  - If `self.waiting_for_buy_confirmation` was true from the previous bar (meaning a bullish flip occurred then), the flag is reset. If the `is_uptrend` remains true on the *current* bar and there's no open position, a `buy()` order is placed.
  - A similar logic applies for `self.waiting_for_sell_confirmation`.
- **Flip Detection Logic**:
  - If the current state is `is_uptrend` but the previous state `was_uptrend` was false, it means the SuperTrend just flipped from bearish to bullish. In this case, `self.waiting_for_buy_confirmation` is set to `True` (unless a sell confirmation was already pending).
  - Conversely, if `not is_uptrend` and `was_uptrend` was true, the SuperTrend just flipped from bullish to bearish, and `self.waiting_for_sell_confirmation` is set to `True`. The

strategy then waits for the *next* bar to confirm this new trend direction.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	93.83
Median Return %	84.12
Std Dev %	99.07
Win Rate %	85.71
Sharpe Ratio	0.95
Min Return %	-20.83
Max Return %	314.67





VIDYA Strategy

- **Logic and Idea**: This strategy uses the **Volatility Index Dynamic Average (VIDYA)**, an adaptive moving average that adjusts its smoothing period based on market volatility (often measured by CMO - Chande Momentum Oscillator). The idea is that the VIDYA responds faster during volatile, trending periods and slower during calm, ranging periods. Trades are initiated when price breaks significantly from the VIDYA, confirmed by ADX (trend strength) and overall momentum. An ATR-based trailing stop is used.

- **Main Parts of the Strategy Class Code (`VIDYAStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('cmo_period', 14), # Period for Chande Momentum Oscillator
(CMO), used for VIDYA's adaptability
    ('period_min', 10), # Minimum period for VIDYA's EMA
component
    ('period_max', 60), # Maximum period for VIDYA's EMA
component
    ('atr_period', 14), # Period for Average True Range (ATR),
used for trailing stops
    ('atr_multiplier', 1.5), # Multiplier for ATR to set trailing
stop distance
    ('cooldown_bars', 3), # Number of bars to wait after an exit
```

```
before re-entering
    ('threshold_pct', 0.01),  # Percentage deviation from VIDYA
for breakout entry (e.g., 1%)
    ('adx_period', 14), # Period for Average Directional Index
(ADX)
    ('adx_threshold', 20), # ADX threshold for trend strength
confirmation
    ('momentum_period', 30), # Period for Momentum indicator
    ('momentum_threshold', 0.01),  # Minimum percentage momentum
required for entry
)
```

- `cmo_period`: Period for the Chande Momentum Oscillator (CMO), which drives the adaptiveness of VIDYA.
- `period_min`, `period_max`: The minimum and maximum periods between which VIDYA's internal EMA period can dynamically adjust.
- `atr_period`, `atr_multiplier`: Parameters for the Average True Range (ATR), used to calculate the trailing stop distance.
- `cooldown_bars`: A cooling-off period (in bars) after an exit, during which new entries are prevented.
- `threshold_pct`: A percentage deviation from the VIDYA line that the price must exceed to trigger a breakout entry.
- `adx_period`, `adx_threshold`: Parameters for the ADX indicator, used to confirm sufficient trend strength for trading.
- `momentum_period`, `momentum_threshold`: Parameters for a general Momentum indicator, ensuring that entries are backed by sufficient price momentum.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    min_periods = max(self.params.cmo_period,
self.params.period_max, self.params.atr_period,
                      self.params.adx_period,
self.params.momentum_period)
    if len(self.data) < min_periods + 1: # Ensure all indicators
have enough data
        return

    # Cancel pending orders
    if self.order:
        return

    # Store previous VIDYA before update
    self.prev_vidya_value = self.vidya_value
```

```
        # Calculate adaptive period using lagged CMO
        lagged_norm_abs_cmo = min(1.0, abs(self.cmo[-1]) / 100.0) #
CMO from previous bar
        adaptive_period = self.params.period_max -
lagged_norm_abs_cmo * (self.params.period_max -
self.params.period_min)
        alpha = 2.0 / (adaptive_period + 1) # Calculate EMA smoothing
factor (alpha)

        # Initialize/update VIDYA
        if self.vidya_value is None: # For the very first calculation
            self.vidya_value = self.data.close[0]
            return
        self.vidya_value = alpha * self.data.close[0] + (1 - alpha) *
self.vidya_value # VIDYA calculation

        # Entry signals with filtering
        lagged_close = self.data.close[-1] # Closing price of the
previous bar
        lagged_vidya = self.prev_vidya_value # VIDYA value from the
previous bar

        # Cooldown check: prevent re-entry too quickly after an exit
        if (len(self.data) - self.last_exit_bar) <
self.params.cooldown_bars:
            return

        # TREND STRENGTH FILTER - ADX must be above threshold
        if self.adx[0] < self.params.adx_threshold:
            return # Skip if trend is not strong enough

        # MOMENTUM VALIDATOR - Recent momentum must be strong enough
        momentum_pct = (self.momentum[0] / self.data.close[-
self.params.momentum_period]) * 100 # Calculate momentum as
percentage
        if abs(momentum_pct) < self.params.momentum_threshold:
            return # Skip if momentum is too weak

        # Entry with threshold confirmation + filters
        if not self.position: # If no open position, look for entry
signals
            threshold = lagged_vidya * self.params.threshold_pct #
Calculate the price threshold for breakout

            # Long: price above VIDYA + positive momentum + strong
trend
            if (lagged_close > (lagged_vidya + threshold) and #
Previous close breaks above VIDYA + threshold
                momentum_pct > self.params.momentum_threshold): #
```

```
Confirmed by positive momentum
            self.order = self.buy() # Place buy order
            # Set trailing stop immediately after entry
            self.sell(exectype=bt.Order.StopTrail,
trailamount=self.params.atr_multiplier * self.atr[0])

        # Short: price below VIDYA + negative momentum + strong
trend
        elif (lagged_close < (lagged_vidya - threshold) and #
Previous close breaks below VIDYA - threshold
              momentum_pct < -self.params.momentum_threshold): #
Confirmed by negative momentum
            self.order = self.sell() # Place sell order
            # Set trailing stop immediately after entry
            self.buy(exectype=bt.Order.StopTrail,
trailamount=self.params.atr_multiplier * self.atr[0])

    # Exit on signal reversal (if currently in a position)
    elif self.position.size > 0 and lagged_close < lagged_vidya:
# If long, exit if price falls below VIDYA
        self.close()
        self.last_exit_bar = len(self.data) # Record exit bar for
cooldown
    elif self.position.size < 0 and lagged_close > lagged_vidya:
# If short, exit if price rises above VIDYA
        self.close()
        self.last_exit_bar = len(self.data) # Record exit bar for
cooldown
```

- **Data Sufficiency and Order/Cooldown Check**: Ensures enough bars for all indicators. It then checks for pending orders and if the strategy is within a `cooldown_bars` period after a previous exit.
- **VIDYA Calculation**:
  - It calculates `lagged_norm_abs_cmo` (normalized absolute CMO from the *previous* bar) which dictates how adaptive the VIDYA will be. A higher absolute CMO means higher volatility, leading to a shorter `adaptive_period`.
  - `alpha` is then calculated from this `adaptive_period` to be used in the Exponential Moving Average formula.
  - `self.vidya_value` is updated using the EMA formula, where the `alpha` dynamically changes based on market volatility.
- **Filter Conditions (No Position)**: Before entering a trade, several filters are applied:
  - **ADX Trend Strength**: `if self.adx[0] < self.params.adx_threshold: return` ensures that trades are

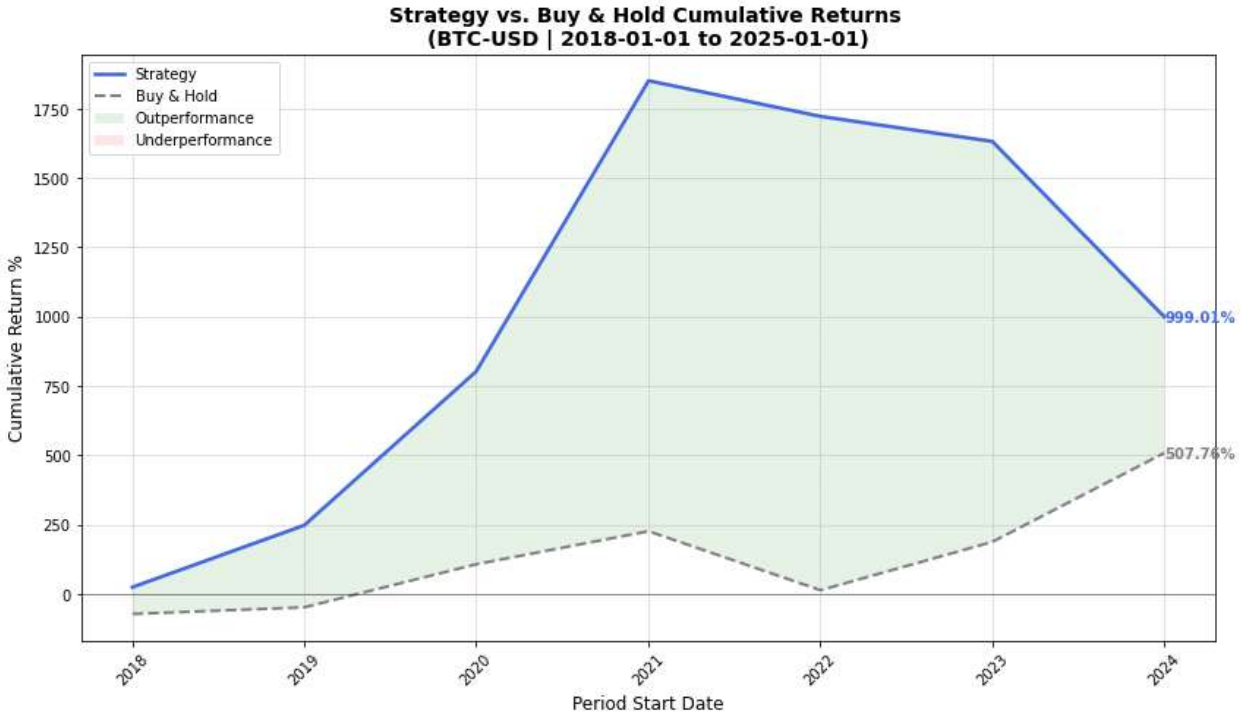
only considered when the ADX indicates a sufficiently strong trend.

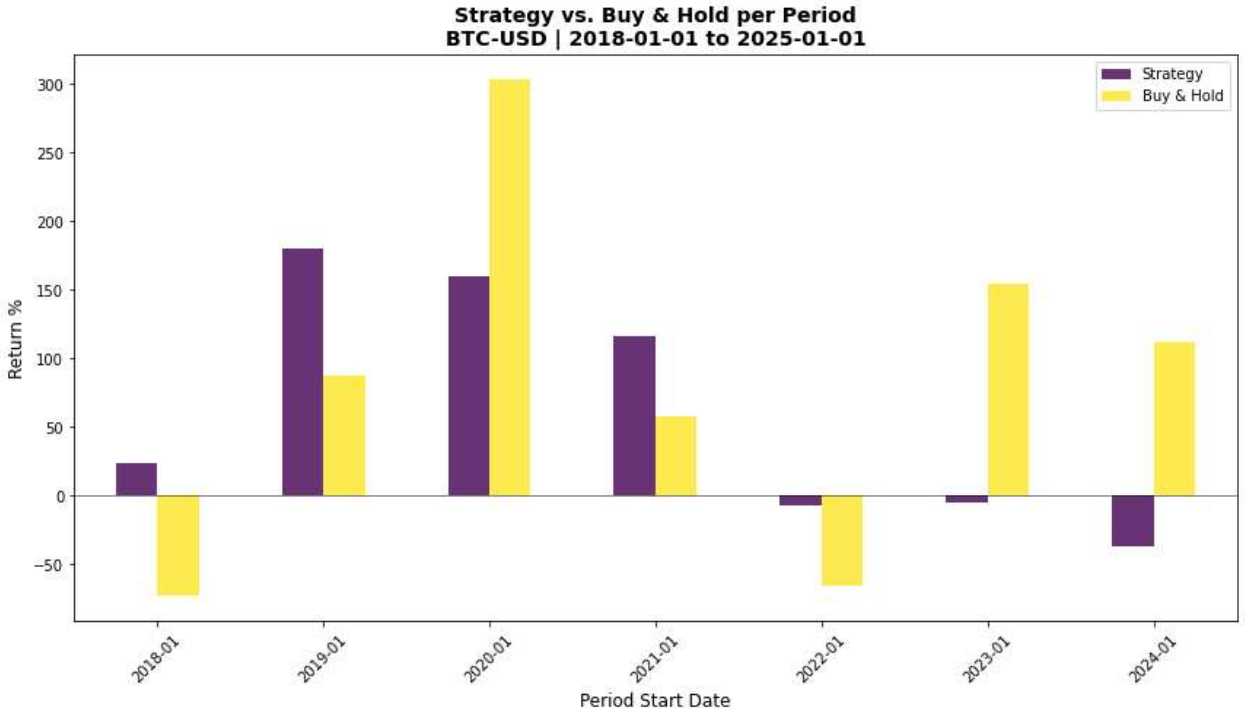
- **Momentum Validator**: `if abs(momentum_pct) < self.params.momentum_threshold: return` checks if the current price momentum (calculated as a percentage change) meets a minimum threshold, ensuring the move has conviction.

- **Entry Logic (No Position)**: If all filters pass and there's no open position:
  - A `threshold` is calculated as a percentage of the `lagged_vidya`.
  - **Long Entry**: A `buy()` order is placed if the `lagged_close` (previous bar's closing price) breaks *above* `lagged_vidya + threshold` and `momentum_pct` is positive and exceeds `momentum_threshold`. A trailing stop is immediately placed.
  - **Short Entry**: A `sell()` order is placed if `lagged_close` breaks *below* `lagged_vidya - threshold` and `momentum_pct` is negative and below `-momentum_threshold`. A trailing stop is immediately placed.
- **Exit Logic (In Position)**:
  - If in a long position, the position is closed if `lagged_close` falls *below* `lagged_vidya`, signaling a reversal against the trend.
  - If in a short position, the position is closed if `lagged_close` rises *above* `lagged_vidya`. `self.last_exit_bar` is updated to initiate the cooldown period.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	4
Losing Periods	3
Mean Return %	61.71
Median Return %	23.95
Std Dev %	81.77
Win Rate %	57.14
Sharpe Ratio	0.75
Min Return %	-36.52
Max Return %	180.12





Vortex Trend Capture Strategy

- **Logic and Idea**: This strategy employs the **Vortex Indicator (VI)** for identifying and confirming trend direction. The Vortex Indicator consists of two lines, VI+ and VI-, which measure positive and negative price movement. A crossover of these lines signals a potential trend. This strategy filters these signals with a long-term **Moving**

Average (MA) for macro trend alignment and an **Average True Range (ATR)** based volatility filter to ensure trades are taken in stable market conditions. Risk is managed with an **ATR-based trailing stop**.

- **Main Parts of the Strategy Class Code (`VortexTrendCaptureStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # Vortex Indicator
    ('vortex_period', 30),
    # Macro Trend Filter
    ('long_term_ma_period', 30),
    # Volatility Filter
    ('atr_period', 7),
    ('atr_threshold', 0.05), # Max ATR as % of price to allow
trades (e.g., 5%)
    # Risk Management
    ('atr_stop_multiplier', 3.0),
)
```

    - `vortex_period`: The period for calculating the Vortex Indicator's VI+ and VI- lines.
    - `long_term_ma_period`: The period for the Simple Moving Average (SMA), used to determine the prevailing macro trend direction.
    - `atr_period`: The period for calculating the Average True Range (ATR), used for both volatility filtering and trailing stop calculations.
    - `atr_threshold`: The maximum acceptable ATR value (as a percentage of price) for trades to be considered. This filters out overly volatile or choppy market conditions.
    - `atr_stop_multiplier`: A multiplier applied to the ATR value to determine the distance of the trailing stop.
  - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    if self.order: return # Exit if there's an existing order
pending.

    if not self.position: # Logic when not currently in a trade.
        # --- Filter Conditions ---
        # 1. Is market volatility stable? (ATR as percentage of
close price is below threshold)
        is_stable = (self.atr[0] / self.data.close[0]) <
self.p.atr_threshold

        # 2. Is price aligned with the macro trend?
```

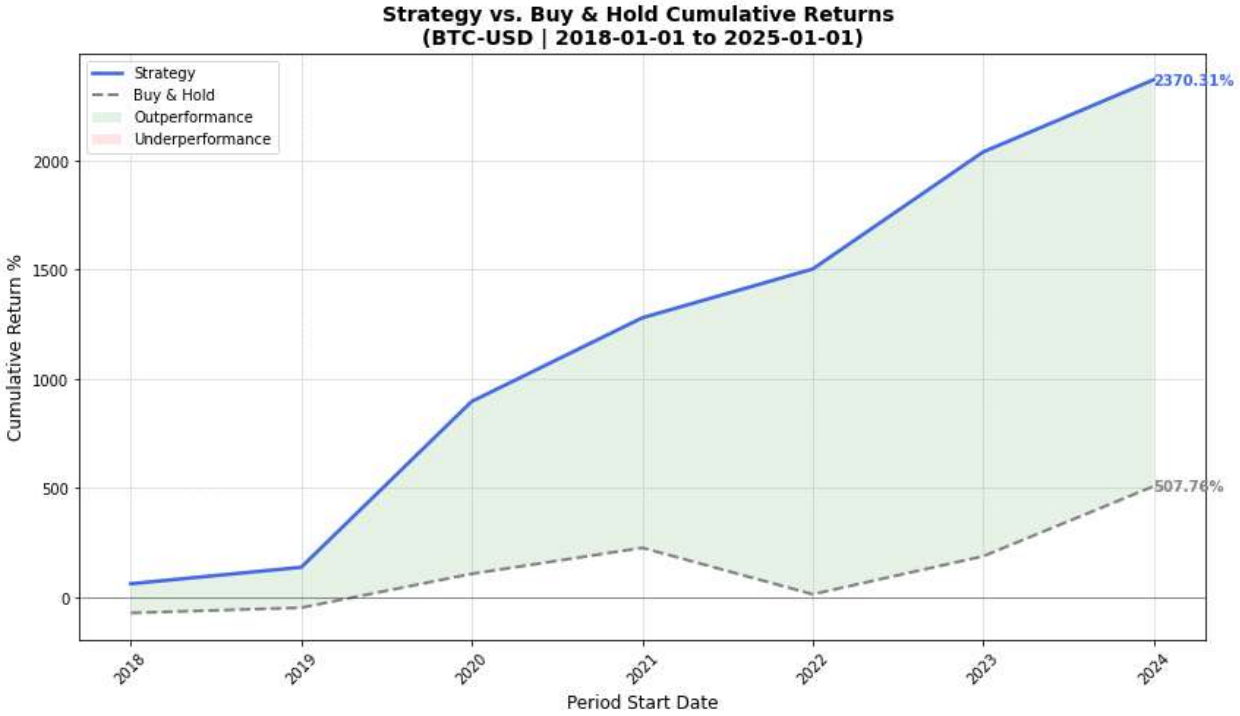
```
        is_macro_uptrend = self.data.close[0] >
self.long_term_ma[0]
        is_macro_downtrend = self.data.close[0] <
self.long_term_ma[0]

        # 3. Has a Vortex crossover signal occurred?
        is_buy_signal = self.vortex_cross[0] > 0 # VI+ crosses
above VI-
        is_sell_signal = self.vortex_cross[0] < 0 # VI- crosses
above VI+

        # --- Entry Logic ---
        # Buy if market is stable, macro trend is up, and Vortex
gives a buy signal.
        if is_stable and is_macro_uptrend and is_buy_signal:
            self.order = self.buy()
        # Sell if market is stable, macro trend is down, and
Vortex gives a sell signal.
        elif is_stable and is_macro_downtrend and is_sell_signal:
            self.order = self.sell()

    elif self.position: # Logic when currently in a trade, for
trailing stop management.
        # --- Manual ATR Trailing Stop Logic ---
        if self.position.size > 0: # If long position
            # Keep track of the highest price since entry.
            self.highest_price_since_entry =
max(self.highest_price_since_entry, self.data.high[0])
            # Calculate a new potential stop price based on the
highest price and ATR multiplier.
            new_stop = self.highest_price_since_entry -
(self.atr[0] * self.p.atr_stop_multiplier)
            # Update the stop price, ensuring it only moves up
(for long positions) to lock in profit.
            self.stop_price = max(self.stop_price, new_stop)
            # If the current closing price falls below the
trailing stop, close the position.
            if self.data.close[0] < self.stop_price: self.order =
self.close()
        elif self.position.size < 0: # If short position
            # Keep track of the lowest price since entry.
            self.lowest_price_since_entry =
min(self.lowest_price_since_entry, self.data.low[0])
            # Calculate a new potential stop price based on the
lowest price and ATR multiplier.
            new_stop = self.lowest_price_since_entry +
(self.atr[0] * self.p.atr_stop_multiplier)
            # Update the stop price, ensuring it only moves down
(for short positions).
```

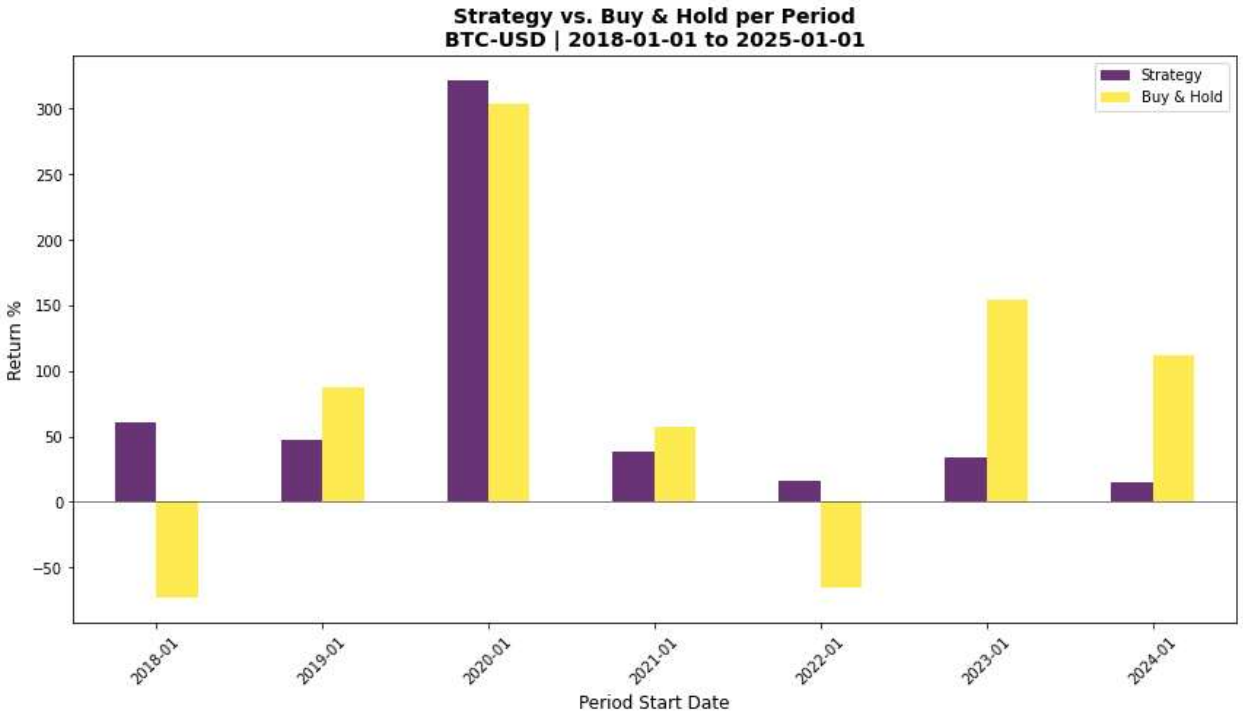
```
            self.stop_price = min(self.stop_price, new_stop)
            # If the current closing price rises above the
trailing stop, close the position.
            if self.data.close[0] > self.stop_price: self.order =
self.close()
```

- **Order Check**: The method starts by checking if `self.order` is pending. If true, it returns to avoid placing new orders.
- **Filter Conditions (No Position)**: If the strategy is not in a trade:
  - **Volatility Stability (`is_stable`)**: Checks if the current ATR, normalized by the closing price, is below the `atr_threshold`. This filters out excessively choppy or highly volatile periods.
  - **Macro Trend Alignment (`is_macro_uptrend`/`is_macro_downtrend`)**: Determines if the current closing price is above (uptrend) or below (downtrend) the `long_term_ma`. This ensures trades align with the broader market direction.
  - **Vortex Crossover Signal (`is_buy_signal`/`is_sell_signal`)**: Checks if `self.vortex_cross[0]` is positive (VI+ crosses above VI-, bullish) or negative (VI- crosses above VI+, bearish).
- **Entry Logic (No Position)**: A `buy()` order is placed if all three conditions are met for a long trade (stable, macro uptrend, and bullish Vortex signal). A `sell()` order is placed if all conditions align for a short trade.
- **Trailing Stop Logic (In Position)**: If the strategy is in a position, it manually manages a trailing stop. For a long position, it tracks the `highest_price_since_entry` and updates `self.stop_price` to trail that high. If the price falls below this `stop_price`, the position is closed. A similar logic applies to short positions, trailing the `lowest_price_since_entry`.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	7
Losing Periods	0
Mean Return %	76.06
Median Return %	38.54
Std Dev %	101.02
Win Rate %	100.00
Sharpe Ratio	0.75
Min Return %	15.46
Max Return %	320.77





ZLEMA Crossover Strategy

- **Logic and Idea**: This strategy utilizes the **Zero Lag Exponential Moving Average (ZLEMA)**, a variation of the EMA designed to reduce lag, making it more responsive to price changes. The strategy generates trading signals based on the crossover of a fast ZLEMA and a slow ZLEMA, similar to traditional moving average crossover systems but with potentially faster signal generation due to reduced lag. A fixed percentage stop-loss is used for risk management.

- **Main Parts of the Strategy Class Code (`ZLEMAStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('fast_period', 7),  # Period for the faster Zero Lag EMA
    ('slow_period', 30), # Period for the slower Zero Lag EMA
    ('stop_loss_pct', 0.01), # Fixed percentage stop-loss (e.g., 1%)
)
```

        - `fast_period`: The period for the faster ZLEMA, which responds quickly to price changes.
        - `slow_period`: The period for the slower ZLEMA, providing a smoother, longer-term average.
        - `stop_loss_pct`: A fixed percentage from the entry price used to set a static stop-loss order.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

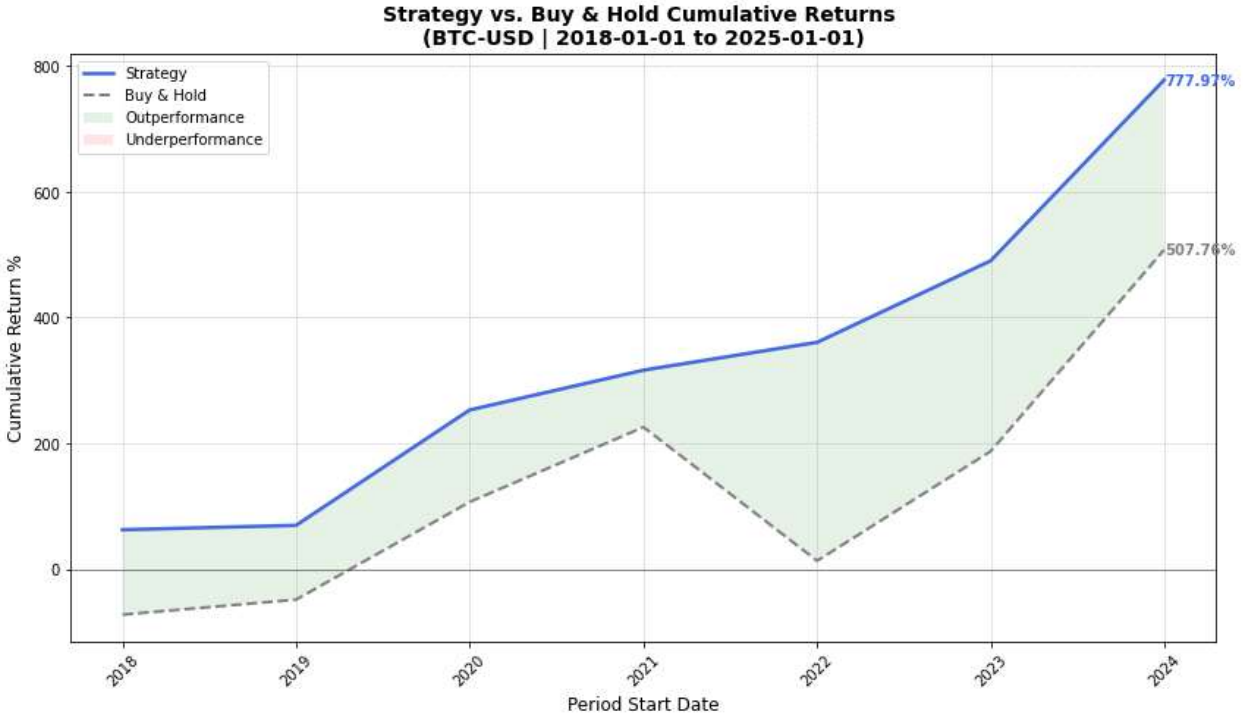
```
def next(self):
    if self.order is not None: # If an order is already pending,
return
        return

    # Zero Lag EMA crossover signals
    if self.crossover > 0:  # Fast ZLEMA crosses above slow ZLEMA
(bullish crossover)
        print(f"BUY SIGNAL at bar {len(self)}") # Logging the
signal
        if self.position.size < 0:  # If currently in a short
position, close it first
            if self.stop_order is not None: # Cancel any pending
stop order for the short position
                self.cancel(self.stop_order)
            self.order = self.close() # Close the short position
        elif not self.position:  # If no position, go long
            self.order = self.buy() # Place a buy order

    elif self.crossover < 0:  # Fast ZLEMA crosses below slow
ZLEMA (bearish crossover)
        print(f"SELL SIGNAL at bar {len(self)}") # Logging the
signal
        if self.position.size > 0:  # If currently in a long
position, close it first
            if self.stop_order is not None: # Cancel any pending
stop order for the long position
                self.cancel(self.stop_order)
            self.order = self.close() # Close the long position
        elif not self.position:  # If no position, go short
            self.order = self.sell() # Place a sell order
```

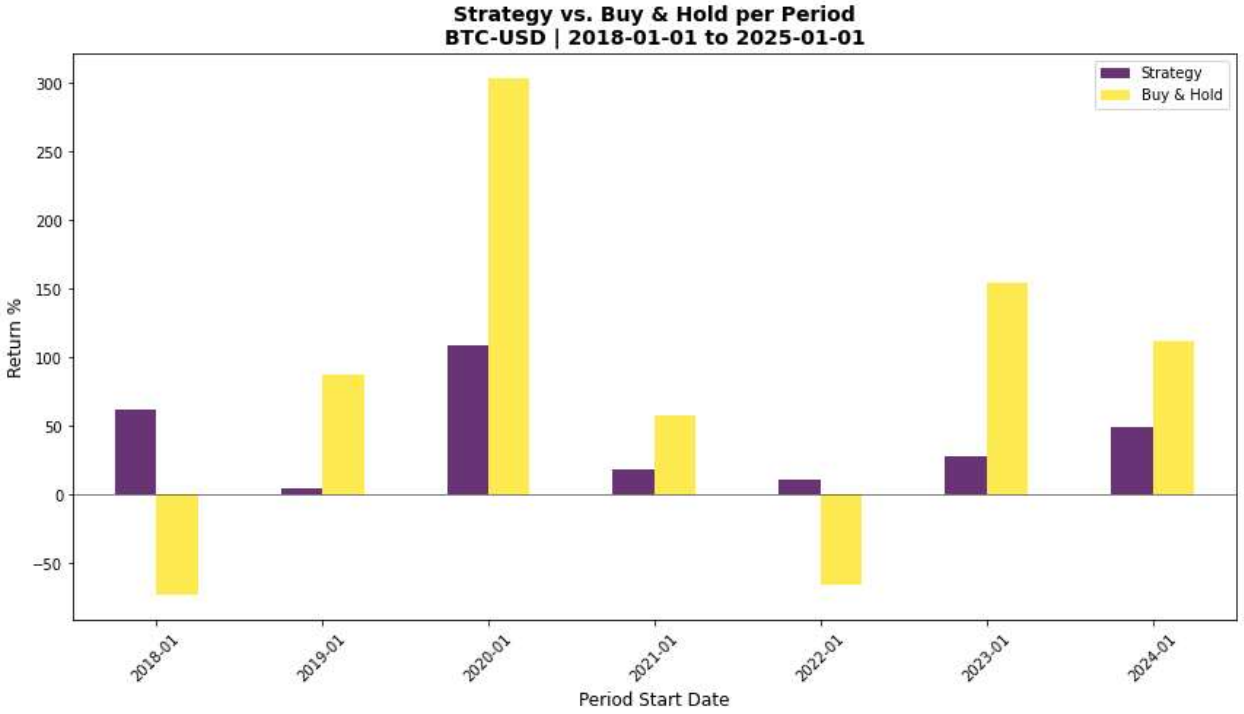
  - **Order Check**: The method first checks if `self.order` is not `None` (an order is pending) and returns if so.
  - **Crossover Signals**: It checks the `self.crossover` indicator, which signals when the `fast_zlema` crosses the `slow_zlema`.
    - **Long Signal (`self.crossover > 0`)**: If the fast ZLEMA crosses above the slow ZLEMA, indicating bullish momentum:
      - If the strategy is currently in a short position (`self.position.size < 0`), it first cancels any associated stop order and then closes the short position.
      - If no position is open (`not self.position`), a `buy()` order is placed.

- **Short Signal (`self.crossover < 0`)**: If the fast ZLEMA crosses below the slow ZLEMA, indicating bearish momentum:
  - If the strategy is currently in a long position (`self.position.size > 0`), it first cancels any associated stop order and then closes the long position.
  - If no position is open, a `sell()` order is placed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	7
Losing Periods	0
Mean Return %	40.07
Median Return %	28.16
Std Dev %	33.81
Win Rate %	100.00
Sharpe Ratio	1.19
Min Return %	4.32
Max Return %	108.26


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
777.97%
507.76%
Cumulative Return %
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
Period Start Date

3. Mean-Reversion Strategies

Mean-reversion strategies are based on the premise that asset prices tend to revert to their average or mean over time. These strategies typically involve buying when the price deviates significantly below its mean (expecting it to rise back) and selling when it deviates significantly above its mean (expecting it to fall back). They are often employed in sideways or ranging markets.

MA Bounce Strategy

- **Logic and Idea**: This strategy is a **mean-reversion** approach that seeks to capitalize on price "**bounces**" off a key moving average (MA) within an established trend. The idea is that in an uptrend, prices often pull back to a support level (the key MA) before continuing their upward movement. The strategy identifies these pullbacks and enters a long position when the price closes back above the MA, confirming the bounce. A longer MA acts as a trend filter, and a fixed percentage stop-loss is used.

- **Main Parts of the Strategy Class Code (`MaBounceStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('key_ma_period', 7),      # MA for bounce (e.g., 50 SMA)
    ('filter_ma_period', 30), # Longer MA for trend filter (e.g.,
200 SMA)
    ('ma_type', 'SMA'),         # Type of MA ('SMA' or 'EMA')
    ('order_percentage', 0.95),
    ('stop_loss_pct', 0.02)    # Example: 2% stop loss below
entry price
)
```

        - `key_ma_period`: The period for the "key" Moving Average, which is expected to act as a dynamic support/resistance level for price bounces.
        - `filter_ma_period`: The period for a longer Moving Average, used as a macro trend filter. Trades are generally only taken in the direction of this longer MA.
        - `ma_type`: A string ('SMA' or 'EMA') to specify whether Simple Moving Averages or Exponential Moving Averages should be used.
        - `order_percentage`: The percentage of available cash to use for each trade.
        - `stop_loss_pct`: The fixed percentage below the entry price to set a static stop-loss order.
    - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

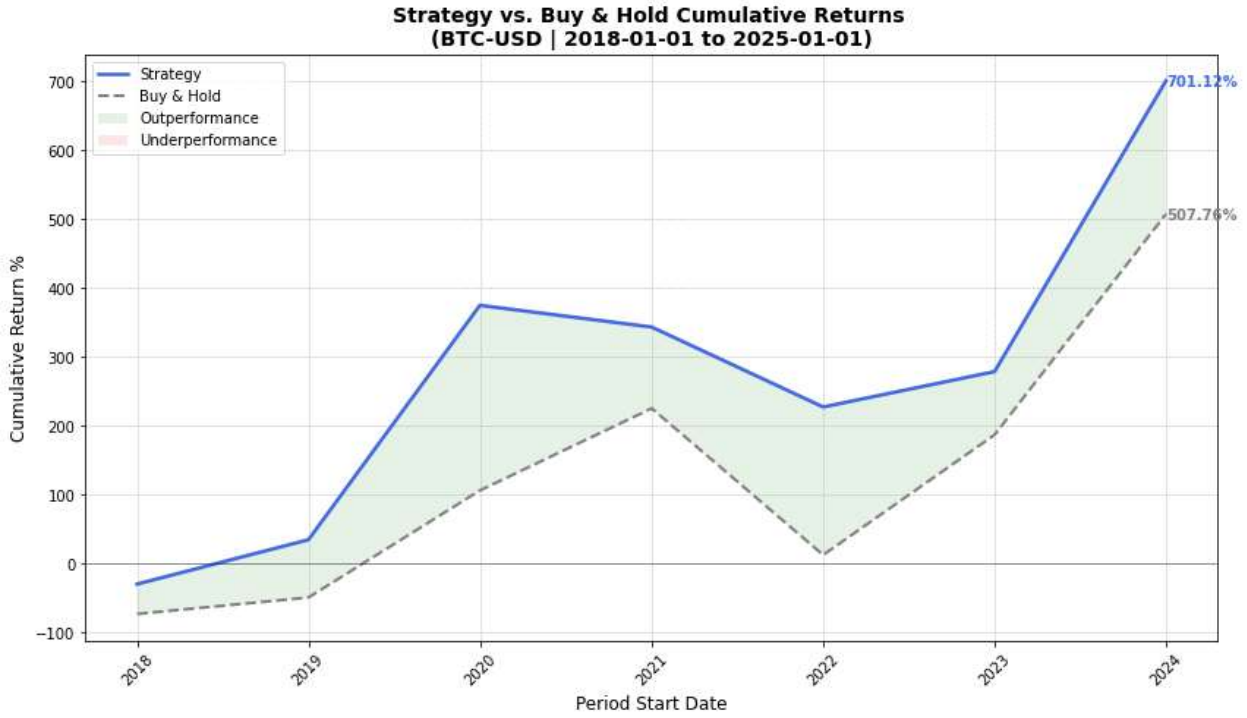
```
def next(self):
    # Check if indicators have enough data
    if len(self.data_close) < self.params.filter_ma_period:
        return

    # Check for open orders
    if self.order:
        return

    # --- Check Stop Loss ---
    if self.position and self.stop_price is not None:
        if self.data_close[0] < self.stop_price: # If current
close falls below calculated stop price
            self.order = self.close() # Close position
            return # Exit check for this bar

    # --- Entry Logic ---
    if not self.position: # Only look for entry if no position is
open
        # 1. Confirm Uptrend State (Price > Filter MA, Key MA >
Filter MA - optional but good)
        uptrend_confirmed = (self.data_close[0] >
self.filter_ma[0] and # Current price above filter MA
                             self.key_ma[0] > self.filter_ma[0])
# Key MA above filter MA (stronger trend)

        if uptrend_confirmed: # Only consider bounces in a
confirmed uptrend
            # 2. Check for Pullback: Low price touched or went
below the key MA in the previous bar
            touched_ma_prev_bar = self.data_low[-1] <=
self.key_ma[-1]

            # 3. Check for Rejection/Entry Trigger: Price closes
back ABOVE the key MA on the current bar
            closed_above_ma_curr_bar = self.data_close[0] >
self.key_ma[0]

            if touched_ma_prev_bar and closed_above_ma_curr_bar:
# If pullback and bounce confirmed
                cash = self.broker.get_cash()
                size = (cash * self.params.order_percentage) /
self.data_close[0] # Calculate position size
                self.order = self.buy(size=size) # Place a buy
order
```

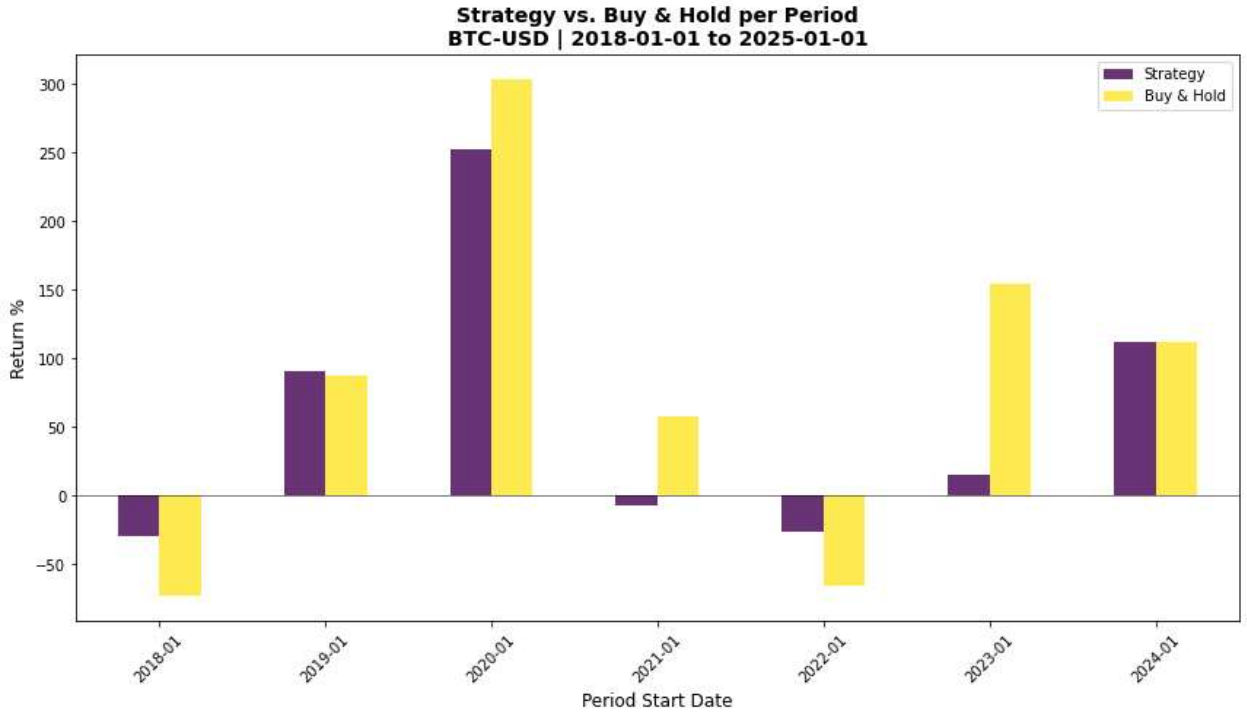
- **Data Sufficiency and Order Check**: The method ensures that enough historical data is available for indicator calculations and that no orders are pending.

- **Stop Loss Check (If in Position)**: If an open position exists and `self.stop_price` has been set, it checks if the current `data_close[0]` has fallen below this stop price. If so, the position is closed.
- **Entry Logic (No Position)**: If no position is open:
  - **Uptrend Confirmation (`uptrend_confirmed`)**: It first verifies a prevailing uptrend. This is typically when the current closing price is above the `filter_ma`, and optionally, when the `key_ma` is also above the `filter_ma` (for stronger confirmation).
  - **Pullback Detection (`touched_ma_prev_bar`)**: If an uptrend is confirmed, it checks if the low price of the *previous* bar (`self.data_low[-1]`) was less than or equal to the `key_ma` of the previous bar (`self.key_ma[-1]`). This signifies that price pulled back to touch or penetrate the key moving average.
  - **Bounce Confirmation (`closed_above_ma_curr_bar`)**: It then checks if the current bar's closing price (`self.data_close[0]`) has closed *above* the current `key_ma`. This confirms the "bounce" or rejection of the key MA as support.
  - If both the `touched_ma_prev_bar` and `closed_above_ma_curr_bar` conditions are met, a `buy()` order is placed with a calculated size.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	4
Losing Periods	3
Mean Return %	58.25
Median Return %	15.68
Std Dev %	94.23
Win Rate %	57.14
Sharpe Ratio	0.62
Min Return %	-29.28
Max Return %	251.81





Ornstein-Uhlenbeck (OU) Mean Reversion Strategy

- **Logic and Idea**: This advanced strategy applies the **Ornstein-Uhlenbeck (OU) process**, a mathematical model often used to describe mean-reverting processes, to price data. The core idea is to estimate the mean (equilibrium price) and the speed of reversion from historical price movements. Trades are initiated when the

price deviates significantly from this estimated mean (measured by a Z-score), expecting it to revert. The strategy also incorporates a Simple Moving Average (SMA) as a **trend filter** to ensure mean reversion trades are aligned with the broader trend.

- **Main Parts of the Strategy Class Code (`OUMeanReversionStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('lookback', 30),           # Rolling window for OU parameter
estimation
    ('sma_period', 30),         # SMA period for trend filter
    ('entry_threshold', 1.),   # Z-score threshold for entry
    ('exit_threshold', 0.1),    # Z-score threshold for exit
    ('printlog', False),        # Print trade logs
)
```

    - `lookback`: The number of historical bars used in the rolling window to estimate the Ornstein-Uhlenbeck process parameters. This window determines how frequently the model's mean and volatility are updated.
    - `sma_period`: The period for a Simple Moving Average (SMA) used as a trend filter. This ensures that mean-reversion trades are taken only when they align with the broader, underlying trend (e.g., buying when oversold in an uptrend).
    - `entry_threshold`: A Z-score threshold for opening a position. The price must deviate this many standard deviations from the estimated mean to trigger an entry.
    - `exit_threshold`: A Z-score threshold for closing a position. Once the price reverts and its Z-score moves back within this threshold, the trade is exited.
    - `printlog`: A boolean flag to enable or disable detailed logging of strategy actions.
  - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Need enough data for parameter estimation
    if len(self.dataclose) < self.params.lookback:
        return

    # Get recent log prices for parameter estimation
    recent_log_prices = np.array([np.log(self.dataclose[-i]) for
i in range(self.params.lookback-1, -1, -1)])

    # Estimate OU parameters
    mu, theta, sigma, eq_std =
```

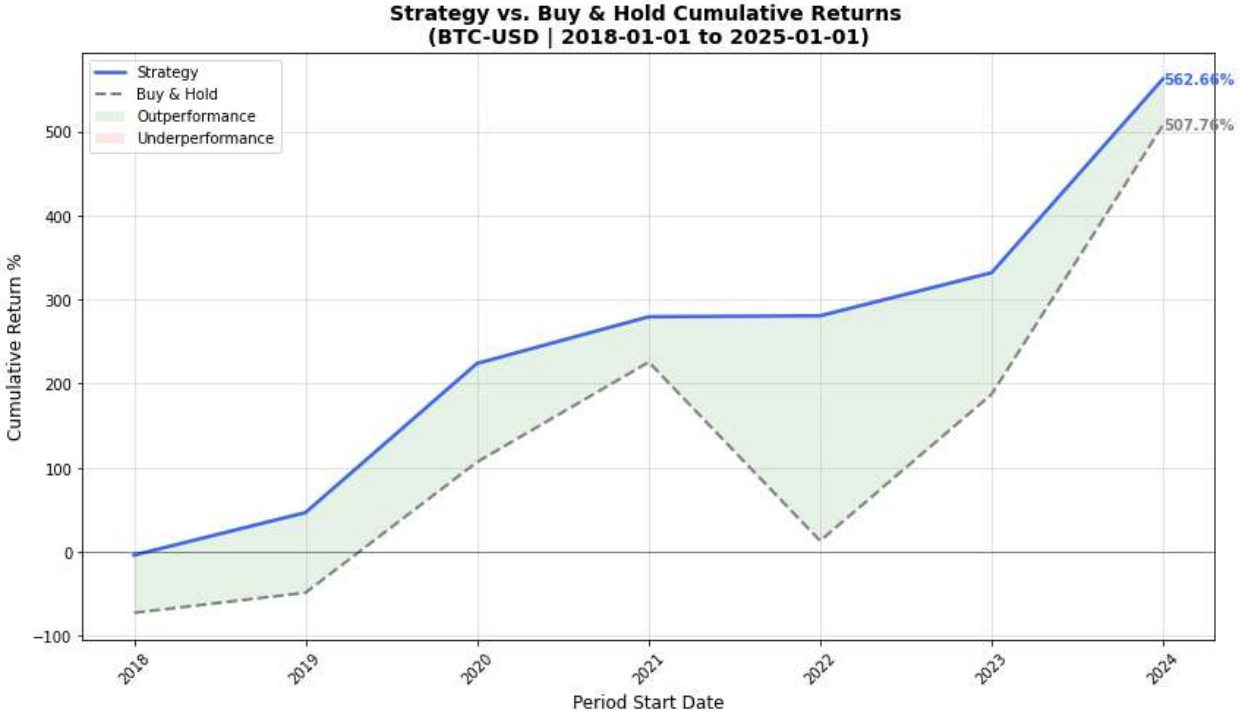
```
self.estimate_ou_parameters(recent_log_prices)

    # Ensure valid parameters are obtained
    if mu is None or eq_std is None or eq_std <= 0:
        return

    # Calculate current deviation and z-score
    current_log_price = np.log(self.dataclose[0])
    deviation = current_log_price - mu
    z_score = deviation / eq_std

    # Store for analysis (optional, for visualization/debugging)
    self.ou_params.append({'mu': mu, 'theta': theta, 'sigma':
sigma, 'eq_std': eq_std})
    self.z_scores.append(z_score)

    self.log(f'Close: {self.dataclose[0]:.4f}, Log Price:
{current_log_price:.4f}, '
             f'μ: {mu:.4f}, Z-Score: {z_score:.2f}')

    # Skip if we have a pending order
    if self.order:
        return

    # Trading logic
    if not self.position:  # No position - look for entry
        if z_score < -self.params.entry_threshold and
self.dataclose[0] > self.sma[0]:
            # Price below mean (oversold) AND in an uptrend - go
long (expect reversion up)
            self.log(f'LONG SIGNAL: Z-Score {z_score:.2f}')
            self.order = self.buy()
            self.position_type = 'long' # Track position type

        elif z_score > self.params.entry_threshold and
self.dataclose[0] < self.sma[0]:
            # Price above mean (overbought) AND in a downtrend -
go short (expect reversion down)
            self.log(f'SHORT SIGNAL: Z-Score {z_score:.2f}')
            self.order = self.sell()
            self.position_type = 'short' # Track position type

    else:  # We have a position - look for exit
        if self.position_type == 'long' and z_score > -
self.params.exit_threshold:
            # Exit long position if price reverts closer to mean
(Z-score rises)
            self.log(f'EXIT LONG: Z-Score {z_score:.2f}')
            self.order = self.sell()
```

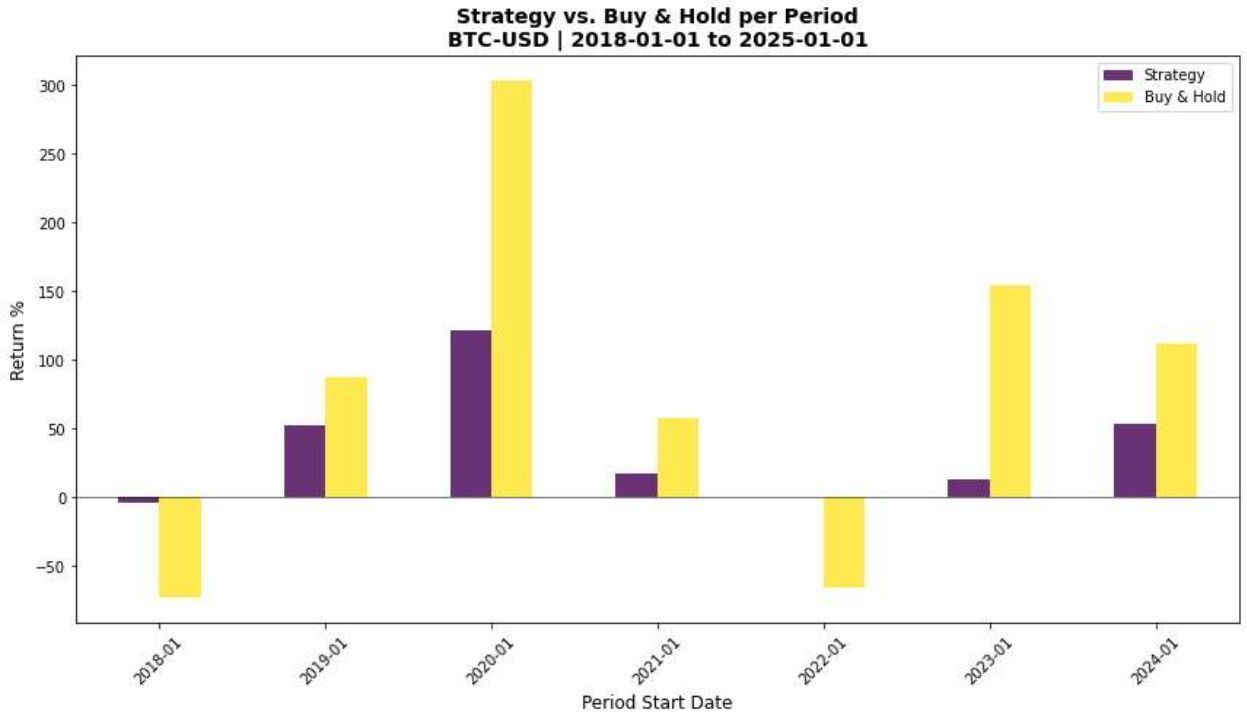
```
                self.position_type = None

        elif self.position_type == 'short' and z_score <
self.params.exit_threshold:
            # Exit short position if price reverts closer to mean
(Z-score falls)
            self.log(f'EXIT SHORT: Z-Score {z_score:.2f}')
            self.order = self.buy()
            self.position_type = None
```

- **Data Sufficiency and OU Parameter Estimation**: The method first ensures there's enough data for the `lookback` period to estimate the OU parameters. It then extracts `recent_log_prices` and calls `self.estimate_ou_parameters()` to get the estimated `mu` (mean), `theta` (reversion speed), `sigma` (volatility), and `eq_std` (equilibrium standard deviation). It returns if any parameters are invalid.
- **Z-score Calculation**: The `z_score` is calculated by normalizing the `deviation` of the current log price from the estimated `mu` by the `eq_std`. This `z_score` indicates how far the price is from its estimated mean in terms of standard deviations.
- **Order Check**: It checks for any pending orders (`self.order`) to prevent placing new ones.
- **Entry Logic (No Position)**: If the strategy is not in a position:
  - **Long Entry**: A `buy()` order is placed if the `z_score` is below `-entry_threshold` (indicating the price is significantly "oversold" relative to its mean) AND the `dataclose[0]` is above the `sma[0]` (confirming an overall uptrend, ensuring the mean-reversion is *with* the trend).
  - **Short Entry**: A `sell()` order is placed if the `z_score` is above `entry_threshold` (indicating the price is significantly "overbought") AND the `dataclose[0]` is below the `sma[0]` (confirming an overall downtrend).
- **Exit Logic (In Position)**: If the strategy is in an open position:
  - **Exit Long**: If in a long position and the `z_score` rises above `-exit_threshold` (meaning the price has reverted back closer to its estimated mean), the position is closed.
  - **Exit Short**: If in a short position and the `z_score` falls below `exit_threshold`, the position is closed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	36.29
Median Return %	17.17
Std Dev %	40.58
Win Rate %	85.71
Sharpe Ratio	0.89
Min Return %	-3.98
Max Return %	120.93





Pivot Point Strategy

- **Logic and Idea**: This strategy is a **mean-reversion and breakout system** based on traditional **Pivot Points (PP)** and their associated support (S1, S2, S3) and resistance (R1, R2, R3) levels. Pivot points are calculated from the previous period's high, low, and close. The strategy identifies two types of interactions: "**bounces**" (price reversing at a level) and "**breakouts**" (price breaking through a level). It uses volume and RSI for confirmation and a fixed percentage stop-loss for risk management. It can use daily, weekly, and monthly pivots.

- **Main Parts of the Strategy Class Code (`PivotPointStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('use_daily', True),         # Use daily pivot points
    ('use_weekly', True),        # Use weekly pivot points
    ('use_monthly', False),      # Use monthly pivot points
    ('bounce_threshold', 0.01), # 0.2% threshold for level
interaction
    ('breakout_threshold', 0.03), # 0.5% threshold for breakouts
    ('volume_multiplier', 1.2),  # Volume confirmation multiplier
    ('volume_period', 7),       # Volume average period
    ('rsi_period', 14),          # RSI for momentum confirmation
    ('stop_loss_pct', 0.05),    # 1.5% stop loss
)
```

- `use_daily`, `use_weekly`, `use_monthly`: Boolean flags to enable or disable the calculation and use of pivot points based on daily, weekly, or monthly periods.
- `bounce_threshold`: A percentage (e.g., 0.01 for 1%) defining how close the price must be to a pivot level to consider it a "bounce" interaction.
- `breakout_threshold`: A percentage defining how far the price must move beyond a pivot level to consider it a "breakout."
- `volume_multiplier`, `volume_period`: Parameters for a Simple Moving Average of volume, used to confirm trade signals (volume must exceed average by a multiplier).
- `rsi_period`: Period for the Relative Strength Index (RSI), used to confirm momentum and avoid overbought/oversold extremes.
- `stop_loss_pct`: The fixed percentage below/above the entry price for a stop-loss order.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    if self.order is not None: # If an order is currently
pending, return
        return

    # Update OHLC data and calculate pivots for current period
    self.update_ohlc_data()

    # Get current price action
    current_price = self.close[0]
    current_high = self.high[0]
    current_low = self.low[0]

    # Check level interactions (bounce or breakout)
    interaction, level_price, timeframe, level_name =
self.check_level_interaction(
        current_price, current_high, current_low
    )

    if interaction is None: # If no significant interaction with
any pivot level, return
        return

    # Trading logic based on pivot level interactions
    if interaction == 'touch' or interaction == 'near':
        # Bounce strategy - expect reversal at key levels
        if level_name.startswith('S'):  # Support level - expect
bounce up (long signal)
```

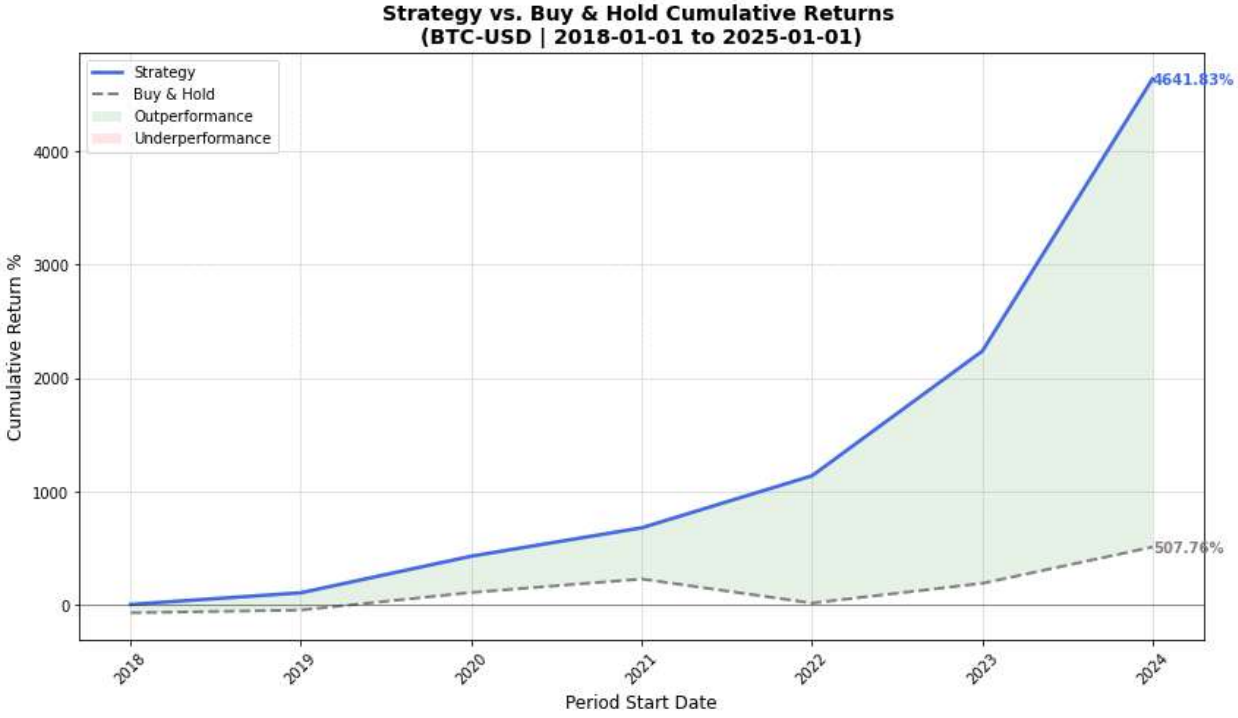
```
            if (self.momentum_confirmation('long') and # Check
RSI for bullish bias
                self.volume_confirmation()): # Check for
increased volume

                if self.position.size < 0:  # If currently short,
close short position
                    if self.stop_order is not None:
self.cancel(self.stop_order)
                    self.order = self.close()
                elif not self.position:  # If no position, go
long
                    self.order = self.buy()

        elif level_name.startswith('R'):  # Resistance level -
expect bounce down (short signal)
            if (self.momentum_confirmation('short') and # Check
RSI for bearish bias
                self.volume_confirmation()): # Check for
increased volume

                if self.position.size > 0:  # If currently long,
close long position
                    if self.stop_order is not None:
self.cancel(self.stop_order)
                    self.order = self.close()
                elif not self.position:  # If no position, go
short
                    self.order = self.sell()

    elif interaction == 'resistance_break': # Breakout of
Resistance (bullish signal)
        # Resistance breakout - go long
        if (self.momentum_confirmation('long') and
            self.volume_confirmation()):

            if self.position.size < 0:  # Close short
                if self.stop_order is not None:
self.cancel(self.stop_order)
                self.order = self.close()
            elif not self.position:  # Go long
                self.order = self.buy()

    elif interaction == 'support_break': # Breakout of Support
(bearish signal)
        # Support breakdown - go short
        if (self.momentum_confirmation('short') and
            self.volume_confirmation()):
```

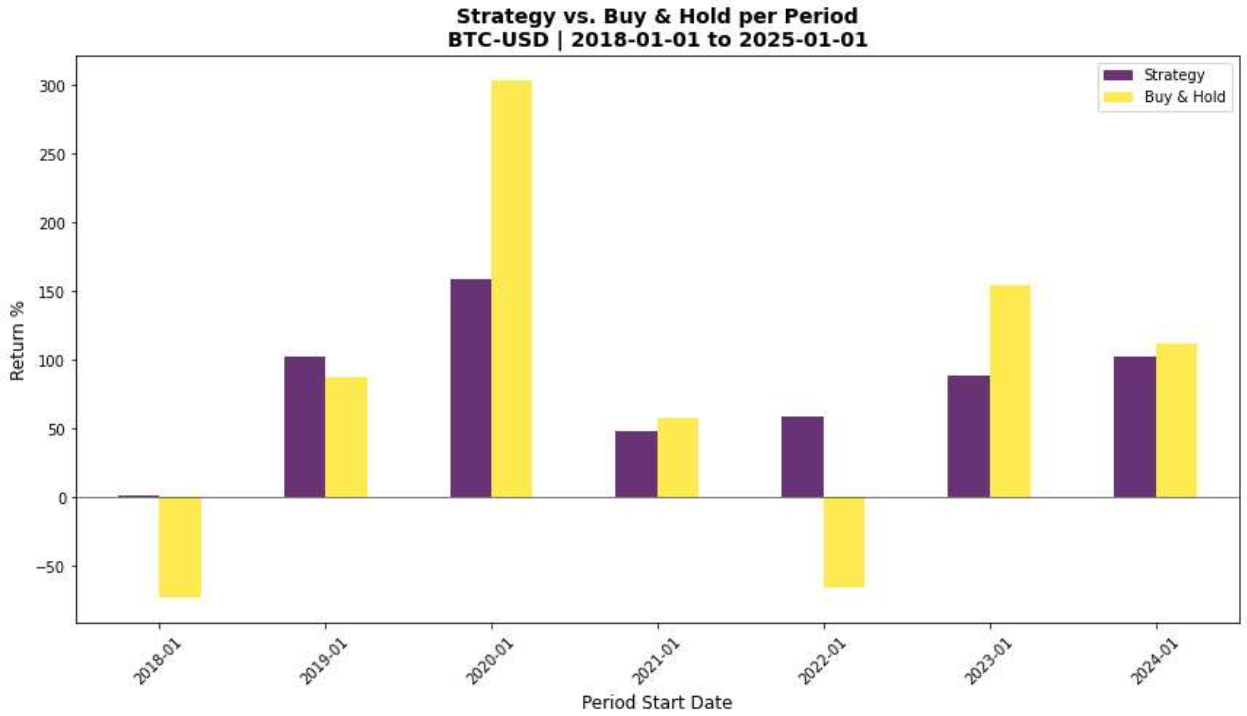
```
            if self.position.size > 0:  # Close long
                if self.stop_order is not None:
self.cancel(self.stop_order)
                self.order = self.close()
            elif not self.position:  # Go short
                self.order = self.sell()
```

- **Order Check and OHLC Update**: The method first checks for pending orders. It then calls `self.update_ohlc_data()` which is crucial for dynamically calculating daily, weekly, and/or monthly pivot points based on the OHLC data of the *previous* period.
- **Level Interaction Detection**: `self.check_level_interaction()` is called to determine if the current price is `touching`, `near`, or `breaking` out of any calculated pivot level, based on the defined thresholds. If no significant interaction is found, the method returns.
- **Trading Logic (Based on Interaction Type)**:
  - **Bounce Strategy (`'touch'` or `'near'`)**: If the price is interacting closely with a pivot level, it implies a potential reversal.
    - If interacting with a **Support (S) level**: It expects a bounce upwards. If `momentum_confirmation('long')` (RSI not overbought/oversold for long) and `volume_confirmation()` are true, it will close any existing short position or enter a new long position.
    - If interacting with a **Resistance (R) level**: It expects a bounce downwards. With appropriate confirmations, it will close existing long positions or enter a new short position.
  - **Breakout Strategy (`'resistance_break'` or `'support_break'`)**: If the price breaks significantly through a pivot level, it implies trend continuation.
    - **Resistance Breakout**: If a `resistance_break` occurs with bullish confirmations, it will close shorts or enter a long.
    - **Support Breakout**: If a `support_break` occurs with bearish confirmations, it will close longs or enter a short. In all cases, before placing a new entry, any existing `self.stop_order` is canceled if an opposing position is being closed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	7
Losing Periods	0
Mean Return %	79.93
Median Return %	88.93
Std Dev %	46.10
Win Rate %	100.00
Sharpe Ratio	1.73
Min Return %	1.15
Max Return %	158.07





Quantile Channel Strategy

- **Logic and Idea**: This is an advanced **mean-reversion strategy** that uses **Quantile Regression** to dynamically estimate price channels (upper, lower, and median/trend line) based on historical price distribution. Unlike standard linear regression which models the mean, quantile regression models specific quantiles (e.g., 20th, 50th, 80th percentiles). The strategy looks for price breakouts from these channels, expecting a reversion back to the mean. It also includes dynamic stop-loss management and a confidence measure for the channel.

- **Main Parts of the Strategy Class Code (`QuantileChannelStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('lookback_period', 60),      # Lookback for channel
estimation
    ('upper_quantile', 0.8),      # Upper channel quantile (80th
percentile)
    ('lower_quantile', 0.2),      # Lower channel quantile (20th
percentile)
    ('trend_quantile', 0.5),      # Trend line quantile (median)
    ('breakout_threshold', 1.02), # Breakout confirmation (2%
above/below)
    ('stop_loss_pct', 0.08),      # 8% stop loss
    ('rebalance_period', 1),      # Daily rebalancing (how often
to re-estimate channels and trade)
    ('min_channel_width', 0.02),  # Minimum 2% channel width (to
```

```
avoid very narrow, noisy channels)
    ('volume_confirm', False),    # Volume confirmation (if
available) - not used in snippet, but a parameter
)
```

- `lookback_period`: The number of historical bars used to fit the quantile regression models and estimate the channels.
- `upper_quantile`, `lower_quantile`, `trend_quantile`: The quantile levels (e.g., 0.8 for 80th percentile) used to define the upper band, lower band, and the central trend line of the channel.
- `breakout_threshold`: A multiplier (e.g., 1.02 for 2%) indicating how far the price must break beyond the channel bands to be considered a valid breakout.
- `stop_loss_pct`: A fixed percentage stop-loss that acts as an initial or ultimate safety net.
- `rebalance_period`: How often (in bars) the strategy re-estimates the channels and reviews its trading decisions.
- `min_channel_width`: A minimum percentage width for the channel to prevent trading on extremely narrow, potentially unstable channels.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Collect price and time data
    current_price = self.data.close[0]
    current_time = len(self.prices) # Simple sequential index for
time

    self.prices.append(current_price)
    self.time_indices.append(current_time)

    # Keep only recent history for lookback period
    if len(self.prices) > self.params.lookback_period * 2: # Keep
buffer larger than lookback
        self.prices = self.prices[-self.params.lookback_period *
2:]
        self.time_indices = self.time_indices[-
self.params.lookback_period * 2:]

    # Estimate channels
    upper_channel, lower_channel, trend_line, confidence =
self.estimate_channels()

    if upper_channel is None: # Not enough data yet for
estimation
        return
```

```
        # Store channel estimates for plotting/analysis (optional)
        self.upper_channel.append(upper_channel)
        self.lower_channel.append(lower_channel)
        self.trend_line.append(trend_line)
        self.channel_confidence = confidence # Store confidence

        # Calculate channel width (for analysis/debug)
        width = (upper_channel - lower_channel) / trend_line
        self.channel_width.append(width)

        # Rebalancing logic: only run full logic every
'rebalance_period' bars
        self.rebalance_counter += 1
        if self.rebalance_counter < self.params.rebalance_period:
            # Still check stop loss even on non-rebalance bars
            if self.position.size > 0 and current_price <=
self.stop_price:
                self.close()
            elif self.position.size < 0 and current_price >=
self.stop_price:
                self.close()
            return # Skip full trading logic until next rebalance
period

        # Reset rebalance counter for the new period
        self.rebalance_counter = 0

        # Detect breakout
        breakout = self.detect_breakout(current_price, upper_channel,
lower_channel)

        # Current position status
        current_pos = 0
        if self.position.size > 0: current_pos = 1
        elif self.position.size < 0: current_pos = -1

        # Trading logic with channel confirmation
        if breakout != 0 and confidence > 0.3:  # Require minimum
channel confidence for trading
            # Close existing opposing position if direction changed
            if current_pos != 0 and current_pos != breakout:
                self.close()
                current_pos = 0 # Position is now flat

            # Open new position on breakout if currently flat
            if current_pos == 0:
                if breakout == 1:  # Upper breakout - go long
                    self.buy()
```

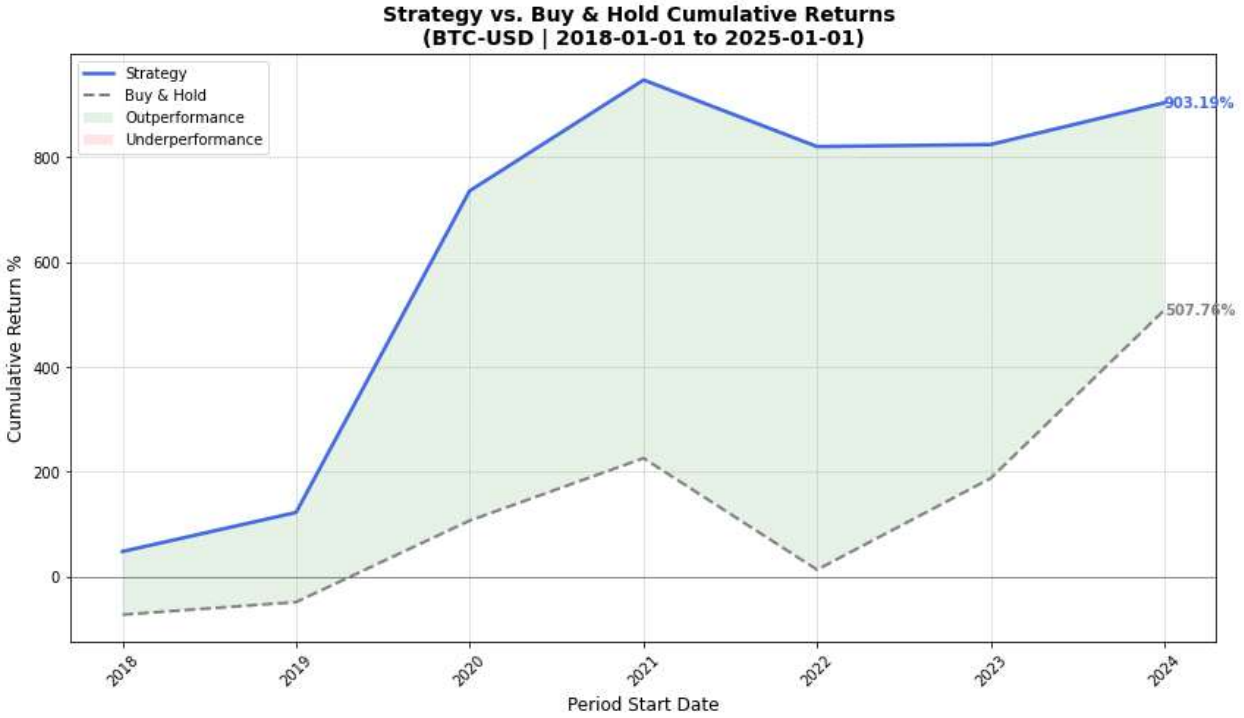
```
                self.stop_price = lower_channel  # Use lower
channel as stop (dynamic)
                self.trade_count += 1
                self.breakout_direction = 1

            elif breakout == -1:  # Lower breakout - go short
                self.sell()
                self.stop_price = upper_channel  # Use upper
channel as stop (dynamic)
                self.trade_count += 1
                self.breakout_direction = -1

    # Exit on return to channel (mean reversion)
    elif self.position.size != 0: # If in a position, check for
mean reversion exit
        in_channel = lower_channel <= current_price <=
upper_channel
        # If price is back inside the channel and close to the
trend line
        if in_channel and abs(current_price - trend_line) /
trend_line < 0.02: # 2% deviation from midline
            self.close()

    # Update trailing stops (dynamic adjustment based on channel
movement)
    if self.position.size > 0:  # Long position
        new_stop = max(self.stop_price, lower_channel) # Stop can
only move up, or stay at lower channel
        if new_stop > self.stop_price:
            self.stop_price = new_stop

    elif self.position.size < 0:  # Short position
        new_stop = min(self.stop_price, upper_channel) # Stop can
only move down, or stay at upper channel
        if new_stop < self.stop_price:
            self.stop_price = new_stop
```

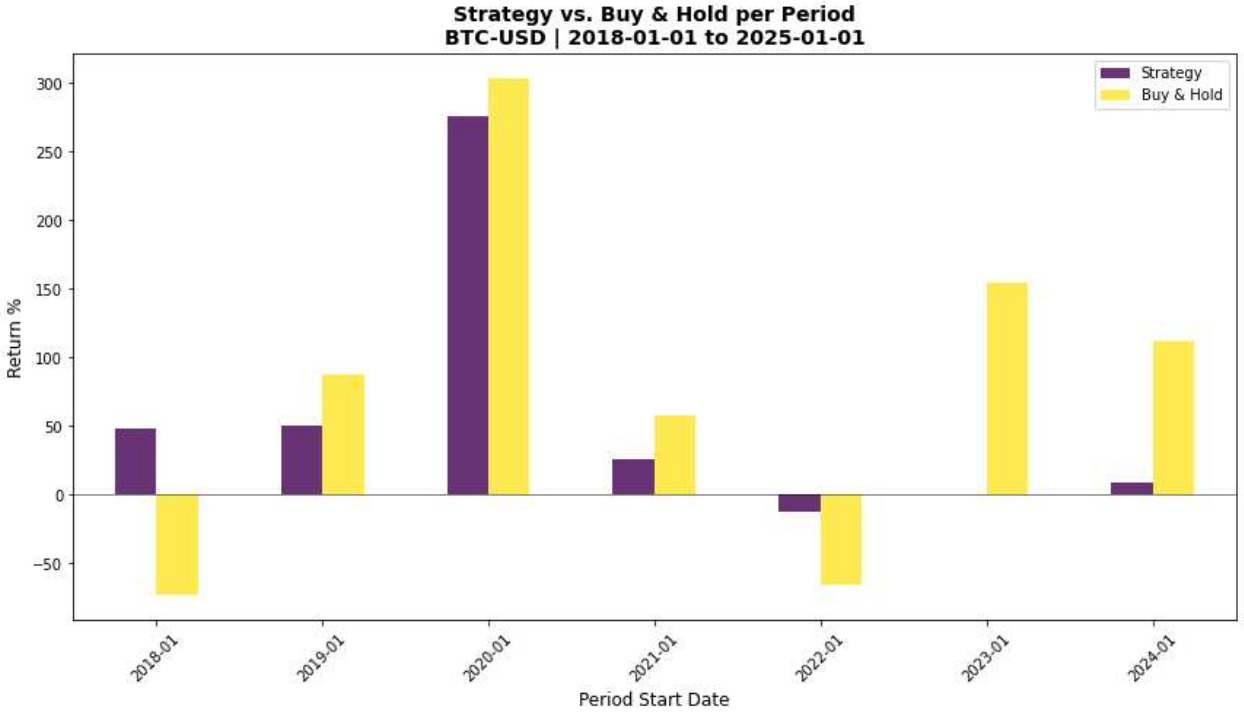
- **Data Collection and Channel Estimation**: The method appends the current price and a time index to internal buffers. It then calls `self.estimate_channels()` to re-calculate the `upper_channel`, `lower_channel`, and `trend_line` using quantile regression, along with a `channel_confidence` score.
- **Rebalancing Logic**: The `rebalance_counter` ensures that the full trading logic (including channel re-estimation and entry signal evaluation) only runs every `rebalance_period` bars. However, stop-loss checks are performed on every bar.

- **Breakout Detection (`breakout`)**: It calls `self.detect_breakout()` to determine if the current price has broken out of the established quantile channels, given the `breakout_threshold`.
- **Trading Logic**:
  - **Entry**: If a `breakout` is detected and the `channel_confidence` is above a minimum threshold (indicating a reliable channel), the strategy acts. If an opposing position exists, it's closed first. Then, if no position is open, a `buy()` order is placed for an upper breakout, or a `sell()` order for a lower breakout. The `stop_price` is dynamically set to the opposite channel boundary.
  - **Mean Reversion Exit**: If the strategy is in an open position, it also looks for mean-reversion exits. If the price returns `in_channel` and is sufficiently close to the `trend_line`, the position is closed. This prevents holding trades that fail to sustain the breakout.
  - **Dynamic Stop Update**: The `stop_price` is continuously updated to trail the price, but it is "anchored" by the dynamic channel boundaries (`lower_channel` for long, `upper_channel` for short), ensuring it adapts to the evolving channel structure.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	56.61
Median Return %	25.32
Std Dev %	92.10
Win Rate %	85.71
Sharpe Ratio	0.61
Min Return %	-12.14
Max Return %	275.93





VWAP Anchored Breakout Strategy

- **Logic and Idea**: This strategy focuses on **price breakouts from prior session highs/lows**, but critically anchors these breakouts with **Volume Weighted Average Price (VWAP)** and confirms them with **ADX** (for trend strength), **volume**, and **ATR expansion**. The idea is to identify strong, validated breakouts where price, volume,

and volatility all align to signal a new trend, with VWAP providing a key reference point. Both session and weekly VWAP are used. **Trailing stops** based on ATR are used for risk management.

- **Main Parts of the Strategy Class Code (`VWAPAnchoredBreakoutStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # VWAP Parameters
    ('vwap_session_length', 7),    # Session length for VWAP
calculation (e.g., 7 bars/days)
    ('vwap_weekly_length', 30),    # Weekly VWAP length (e.g.,
approx. 30 bars/days for a week)

    # Breakout Parameters
    ('breakout_lookback', 7),      # Lookback period for prior
high/low (e.g., highest/lowest of last 7 bars)
    ('adx_threshold', 20),          # ADX > 20 for trend
confirmation
    ('adx_period', 14),             # ADX calculation period

    # Volume and ATR Confirmation
    ('volume_multiplier', 1.2),     # Volume > 1.2x average
(e.g., 1.2 for 10% above average)
    ('volume_period', 7),          # Volume average period
    ('atr_period', 7),             # ATR period
    ('atr_expansion_threshold', 1.5), # ATR expansion threshold
(e.g., 1.5 for 50% higher than average ATR)
    ('atr_expansion_period', 7),   # Period to compare ATR
expansion

    # Trailing Stop Parameters
    ('trailing_stop_atr_multiplier', 2.0), # Trailing stop
distance (e.g., 2 * ATR from highest/lowest point)
    ('initial_stop_atr_multiplier', 3.),  # Initial stop loss
(e.g., 3 * ATR from entry)

    # Risk Management
    ('position_size_pct', 0.95),    # Position size percentage
)
```

    - `vwap_session_length`, `vwap_weekly_length`: Periods for calculating session-based and longer-term (weekly) Volume Weighted Average Prices. VWAP acts as a dynamic average price, weighted by volume.
    - `breakout_lookback`: The number of bars to look back for the highest high or lowest low to define the breakout levels.

- `adx_threshold`, `adx_period`: Parameters for the Average Directional Index (ADX), used to confirm the strength of the trend following a breakout.
- `volume_multiplier`, `volume_period`: Parameters for a Simple Moving Average of volume. Current volume must exceed this average by the `volume_multiplier` for breakout confirmation.
- `atr_period`, `atr_expansion_threshold`, `atr_expansion_period`: Parameters for Average True Range (ATR). `atr_expansion_threshold` checks if current ATR is significantly higher than its recent average, indicating increasing volatility confirming the breakout.
- `trailing_stop_atr_multiplier`, `initial_stop_atr_multiplier`: Multipliers for ATR to set the trailing stop distance and the initial stop loss distance respectively.
- `position_size_pct`: The percentage of available capital to allocate to a trade.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    if self.order: # If an order is pending, return.
        return

    # Skip if not enough data for all indicators to be calculated
    required_data = max(
        self.params.breakout_lookback,
        self.params.vwap_session_length,
        self.params.adx_period,
        self.params.atr_period
    )

    if len(self.dataclose) < required_data:
        return

    current_price = self.dataclose[0]

    # --- Handle existing positions ---
    if self.position:
        self.update_trailing_stop() # Always update trailing stop

        # Check exit conditions for the current position
        should_exit, exit_reason = self.check_exit_conditions()
        if should_exit:
            self.order = self.close() # Close the position
            # Reset tracking variables after closing a position
            self.entry_price = None
            self.stop_price = None
```

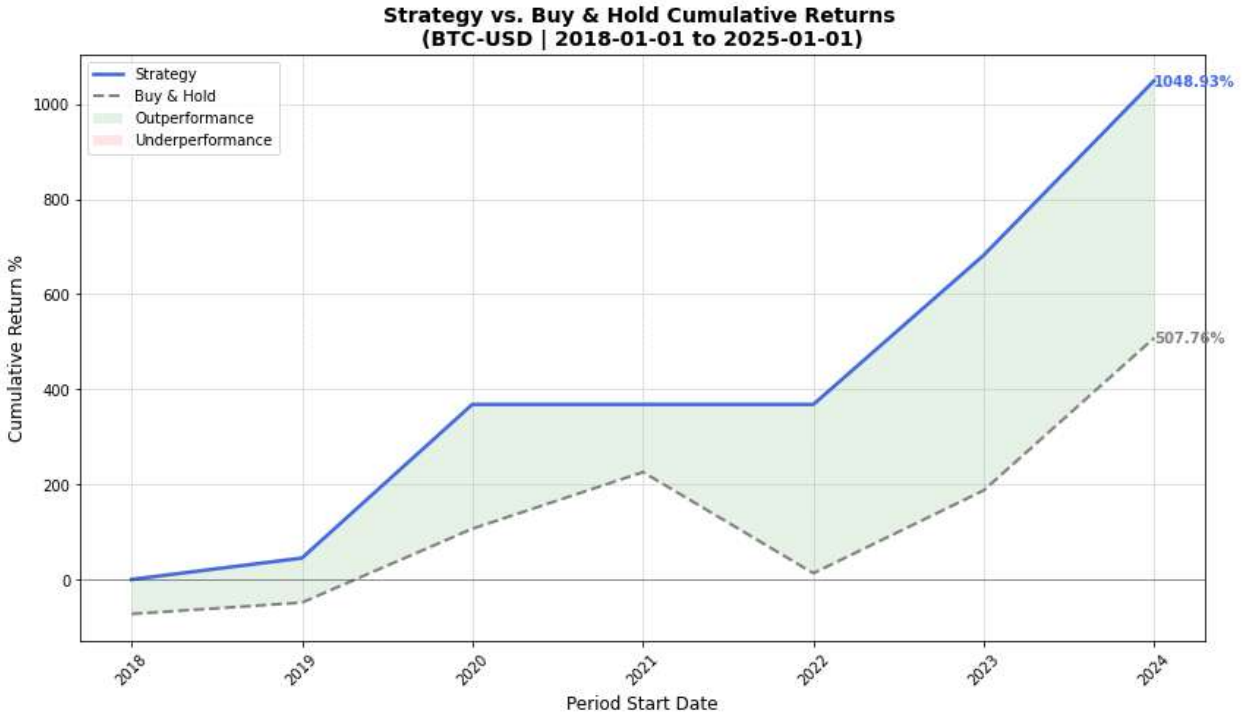
```
                self.trail_price = None
                self.position_type = None
                self.breakout_confirmed = False
                return # Exit this bar's logic

        # --- Look for breakout setup if not in position ---
        if not self.position:
            breakout_valid, direction =
self.check_breakout_conditions() # Check all complex entry
conditions

            if breakout_valid and direction: # If a valid breakout is
detected
                atr_value = self.atr[0] # Get current ATR

                if direction == "LONG": # If it's a bullish breakout
                    # Calculate initial and trailing stop prices
based on current price and ATR
                    self.stop_price = current_price -
(self.params.initial_stop_atr_multiplier * atr_value)
                    self.trail_price = current_price -
(self.params.trailing_stop_atr_multiplier * atr_value)

                    self.order = self.buy() # Place buy order
                    self.entry_price = current_price # Record entry
details
                    self.position_type = 1 # Mark as long position
                    self.breakout_confirmed = True # Confirm breakout
for tracking

                elif direction == "SHORT": # If it's a bearish
breakout
                    # Calculate initial and trailing stop prices
based on current price and ATR
                    self.stop_price = current_price +
(self.params.initial_stop_atr_multiplier * atr_value)
                    self.trail_price = current_price +
(self.params.trailing_stop_atr_multiplier * atr_value)

                    self.order = self.sell() # Place sell order
                    self.entry_price = current_price # Record entry
details
                    self.position_type = -1 # Mark as short position
                    self.breakout_confirmed = True # Confirm breakout
for tracking
```

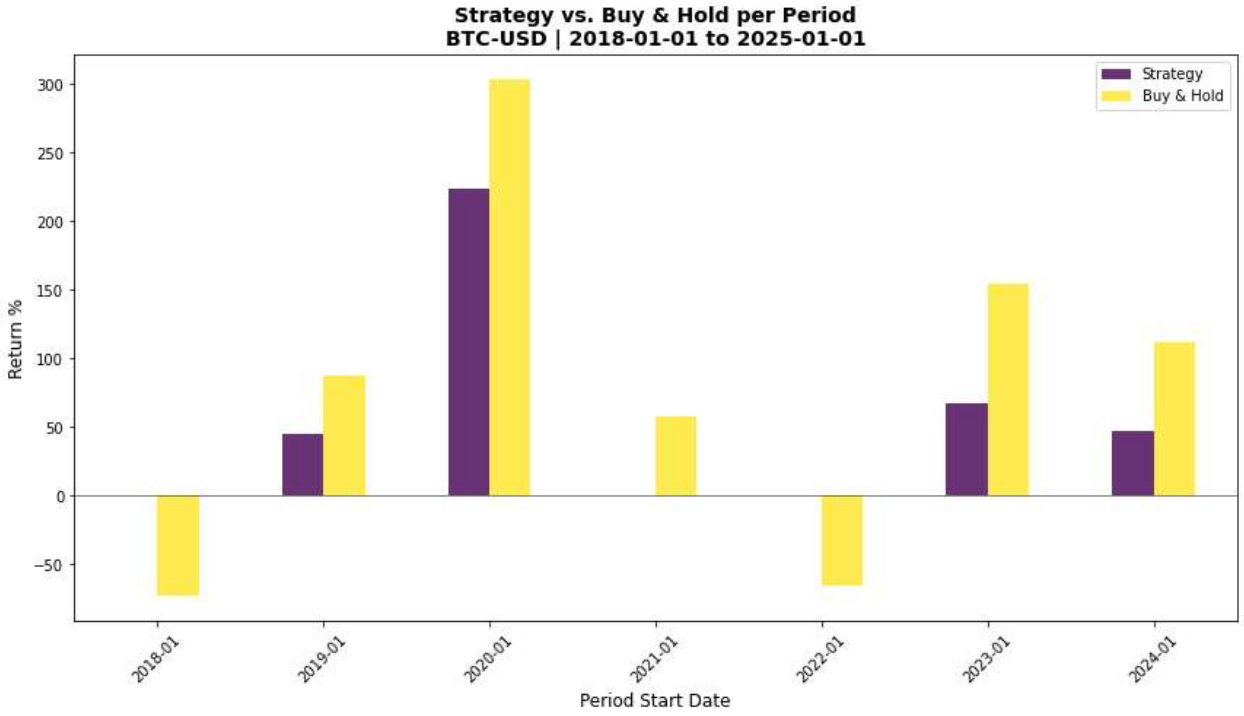
- **Order and Data Check**: The method starts by checking for any pending orders and ensuring enough historical data is available for all required indicators (`required_data`).

- **Position Management (If in Position)**: If the strategy has an open position, it first calls `self.update_trailing_stop()` to adjust the trailing stop. Then, `self.check_exit_conditions()` is called. If any exit condition (initial stop, trailing stop, or VWAP mean reversion) is met, the position is closed, and all associated tracking variables are reset.
- **Entry Logic (No Position)**: If no position is open:
  - `breakout_valid, direction = self.check_breakout_conditions()`: This crucial call evaluates all the complex entry filters: prior high/low breakout, VWAP alignment, ADX trend strength, volume confirmation, and ATR expansion. It returns `True` and the direction ("LONG" or "SHORT") if all conditions are met.
  - If a `breakout_valid` signal is detected, the strategy calculates initial and trailing stop prices based on the current ATR.
  - A `buy()` order is placed for a "LONG" breakout, and a `sell()` order for a "SHORT" breakout. Entry details (`entry_price`, `position_type`) are recorded.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	4
Losing Periods	3
Mean Return %	54.55
Median Return %	44.95
Std Dev %	73.28
Win Rate %	57.14
Sharpe Ratio	0.74
Min Return %	0.00
Max Return %	222.95


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
1048.93%
507.76%
Cumulative Return %
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
Period Start Date

4. Volatility Compression / Breakout Strategies

Designed to detect periods of low volatility (consolidation) and enter trades during sudden breakouts, often using indicators like Bollinger Bands, ROC, or statistical bands to anticipate explosive moves.

Bollinger Band Squeeze Strategy

- **Logic and Idea**: This strategy identifies periods of **low volatility** (a "squeeze") using Bollinger Bands. When the bands contract, it suggests that the market is consolidating before a significant price move. The strategy then waits for a "**breakout**," where the price closes outside the bands, indicating the start of a new trend. A **trailing stop** is used to manage risk and lock in profits as the trend progresses.

- **Main Parts of the Strategy Class Code (`BollingerBandSqueezeStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

      ```
      params = (
          ('bband_period', 7),
          ('bband_devfactor', 1.0),
          ('squeeze_period', 14),
          ('trail_percent', 0.02),  # Trailing stop loss percentage
      (e.g., 2%)
      )
      ```

        - `bband_period`: The number of bars used for calculating the Bollinger Bands.
        - `bband_devfactor`: The standard deviation multiplier for setting the upper and lower Bollinger Bands. A smaller factor makes the bands tighter.
        - `squeeze_period`: The lookback period used to identify the lowest Bollinger Bandwidth, indicating a volatility squeeze.
        - `trail_percent`: The percentage used to calculate the trailing stop-loss from the high/low of the trade.

    - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

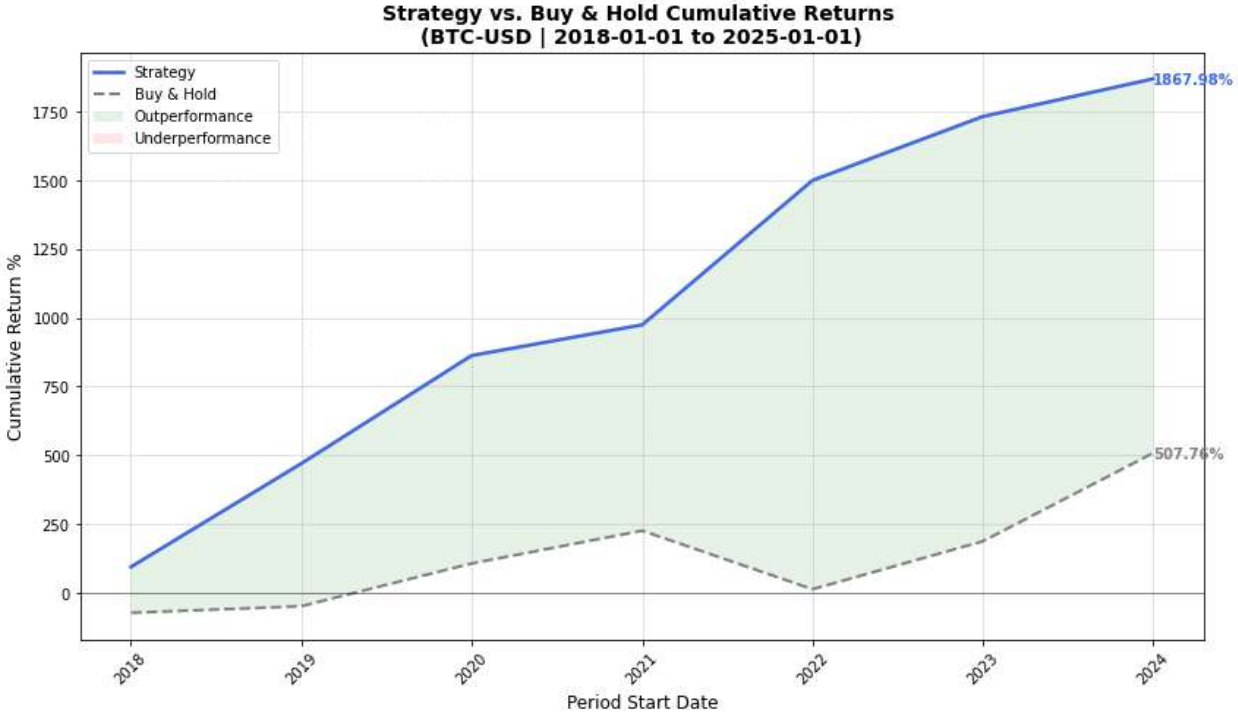
      ```
      def next(self):
          # Check for pending orders and sufficient data
          if self.order or len(self) < self.p.squeeze_period:
              return

          # Check if a squeeze is happening by comparing the current
      bandwidth to its historic low
          is_squeeze = self.lowest_bb_width[-1] == ((self.bband.top[-1]
      ```

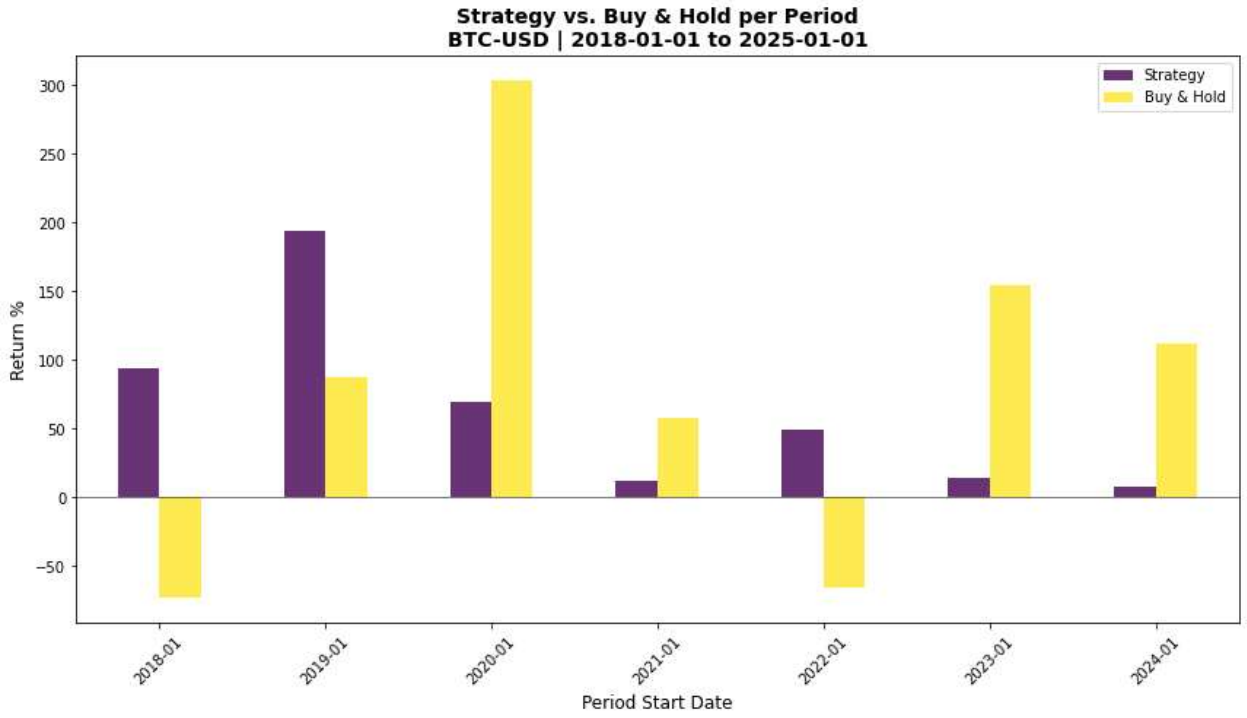
```
- self.bband.bot[-1]) / self.bband.mid[-1])

    # Only enter if not already in a position
    if not self.position:
        if is_squeeze:
            # Breakout to the upside
            if self.dataclose[0] > self.bband.top[0]:
                self.order = self.buy()
            # Breakout to the downside
            elif self.dataclose[0] < self.bband.bot[0]:
                self.order = self.sell()
```

- **Order and Data Check**: The method first checks if there's any pending order (`self.order`) to avoid placing multiple orders, and ensures enough historical data is available (`len(self) < self.p.squeeze_period`) for indicator calculations.
- **Squeeze Detection**: `is_squeeze` is a boolean that evaluates if the current Bollinger Bandwidth is equal to its lowest value over the defined `squeeze_period`. This condition signals that price volatility has contracted significantly.
- **Entry Conditions**: If a squeeze is detected (`is_squeeze` is `True`) and there is no open position (`not self.position`):
  - **Long Entry**: If the current closing price (`self.dataclose[0]`) breaks *above* the upper Bollinger Band (`self.bband.top[0]`), a `buy()` order is placed, signaling an upside breakout.
  - **Short Entry**: If the current closing price (`self.dataclose[0]`) breaks *below* the lower Bollinger Band (`self.bband.bot[0]`), a `sell()` order is placed, signaling a downside breakout.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	7
Losing Periods	0
Mean Return %	62.70
Median Return %	48.87
Std Dev %	61.25
Win Rate %	100.00
Sharpe Ratio	1.02
Min Return %	7.48
Max Return %	193.32





Momentum Ignition Strategy

- **Logic and Idea**: This strategy aims to capture significant price movements that occur after periods of low volatility (consolidation) followed by a "**momentum ignition**" event. It uses **standard deviation** of price to identify consolidation, **Rate of Change (ROC)** for a statistical momentum breakout, and a **Simple Moving Average (SMA)** for overall trend filtering. Risk is managed with an **ATR-based trailing stop**.

- **Main Parts of the Strategy Class Code (`MomentumIgnitionStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # Volatility Filter
    ('consolidation_period', 30),
    ('consolidation_threshold', 0.05), # Max StdDev as % of price
    # Momentum Breakout
    ('roc_period', 7),
    ('roc_ma_period', 30),
    ('roc_breakout_std', 1.0), # ROC must exceed N StdDevs of its
MA
    # Trend Filter
    ('trend_period', 30),
    # Risk Management
    ('atr_period', 7),
```

```
        ('atr_stop_multiplier', 3.0),
    )
```

- `consolidation_period`: The period over which the standard deviation of price is calculated to identify consolidation phases.
- `consolidation_threshold`: The maximum allowable standard deviation (as a percentage of the current price) for the market to be considered in a "consolidating" state. Lower values mean tighter consolidation.
- `roc_period`: The period for calculating the Rate of Change (ROC) indicator, which measures momentum.
- `roc_ma_period`: The period for calculating both a Simple Moving Average (SMA) and a Standard Deviation of the ROC indicator itself. These are used to identify statistically significant breakouts in momentum.
- `roc_breakout_std`: The number of standard deviations the current ROC must exceed from its `roc_ma` to signal a "momentum ignition" breakout.
- `trend_period`: The period for the Simple Moving Average used as a macro (long-term) trend filter.
- `atr_period`: The period for calculating the Average True Range (ATR), used to set the initial and trailing stop-loss distances.
- `atr_stop_multiplier`: A multiplier applied to the ATR value to determine the stop-loss distance.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
    def next(self):
        if self.order: return # Exit if a previous order is still
    pending execution.

        if not self.position: # Logic when not currently in a trade.
            # --- Filter Conditions ---
            # 1. Is the market consolidating (low price volatility)?
            is_consolidating = (self.price_stddev[0] /
    self.data.close[0]) < self.p.consolidation_threshold

            # 2. Is the macro trend aligned?
            is_macro_uptrend = self.data.close[0] > self.trend_sma[0]
            is_macro_downtrend = self.data.close[0] <
    self.trend_sma[0]

            if is_consolidating: # Only proceed if the market is
    consolidating.
                # 3. Has momentum "ignited" with a statistical
```

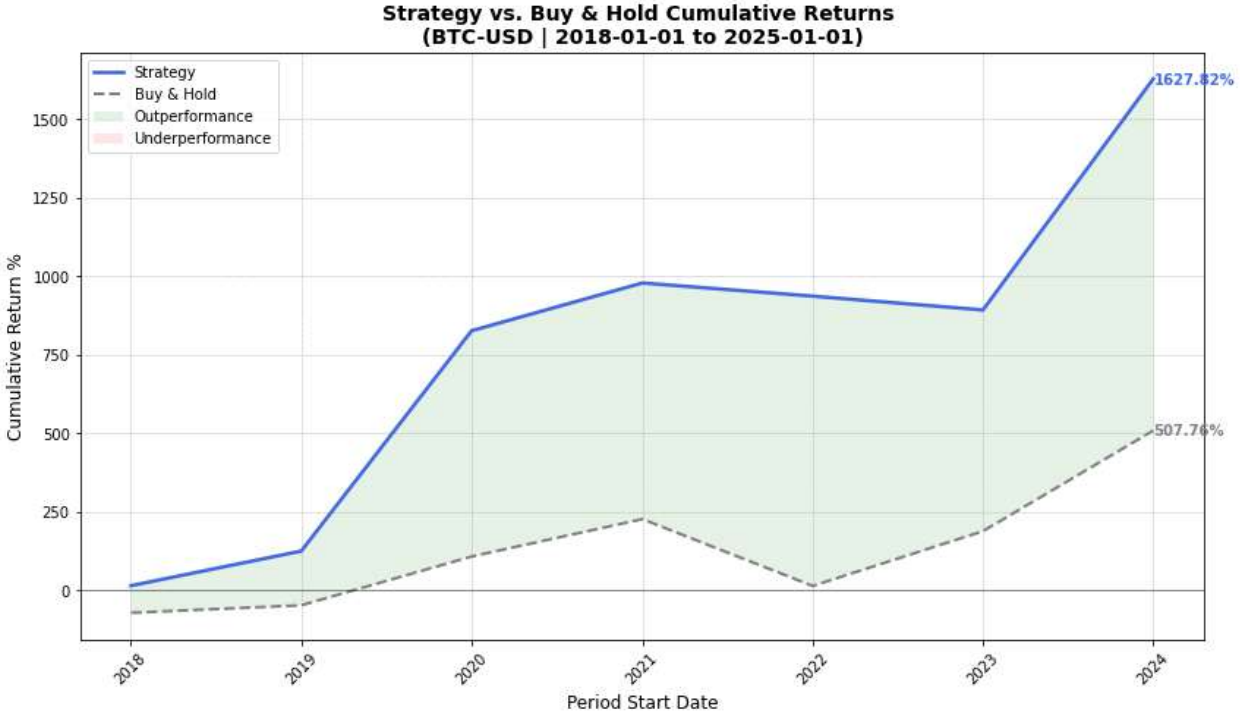
```
breakout?
            # Calculate upper and lower bands for ROC based on
its MA and StdDev.
            roc_upper_band = self.roc_ma[0] + (self.roc_stddev[0]
* self.p.roc_breakout_std)
            roc_lower_band = self.roc_ma[0] - (self.roc_stddev[0]
* self.p.roc_breakout_std)

            # Check if current ROC breaks these bands.
            is_mom_breakout_up = self.roc[0] > roc_upper_band
            is_mom_breakout_down = self.roc[0] < roc_lower_band

            # --- Entry Logic ---
            # Buy if macro trend is up AND momentum breaks out
upwards.
            if is_macro_uptrend and is_mom_breakout_up:
                self.order = self.buy()
            # Sell if macro trend is down AND momentum breaks out
downwards.
            elif is_macro_downtrend and is_mom_breakout_down:
                self.order = self.sell()

    elif self.position: # Logic when currently in a trade.
        # --- Manual ATR Trailing Stop Logic ---
        if self.position.size > 0: # If long position
            # Update highest price reached since entry.
            self.highest_price_since_entry =
max(self.highest_price_since_entry, self.data.high[0])
            # Calculate new potential stop price.
            new_stop = self.highest_price_since_entry -
(self.atr[0] * self.p.atr_stop_multiplier)
            # Update stop price, ensuring it only moves in the
direction of profit (up for long).
            self.stop_price = max(self.stop_price, new_stop)
            # Close position if price falls below the trailing
stop.
            if self.data.close[0] < self.stop_price: self.order =
self.close()
        elif self.position.size < 0: # If short position
            # Update lowest price reached since entry.
            self.lowest_price_since_entry =
min(self.lowest_price_since_entry, self.data.low[0])
            # Calculate new potential stop price.
            new_stop = self.lowest_price_since_entry +
(self.atr[0] * self.p.atr_stop_multiplier)
            # Update stop price, ensuring it only moves in the
direction of profit (down for short).
            self.stop_price = min(self.stop_price, new_stop)
            # Close position if price rises above the trailing
```

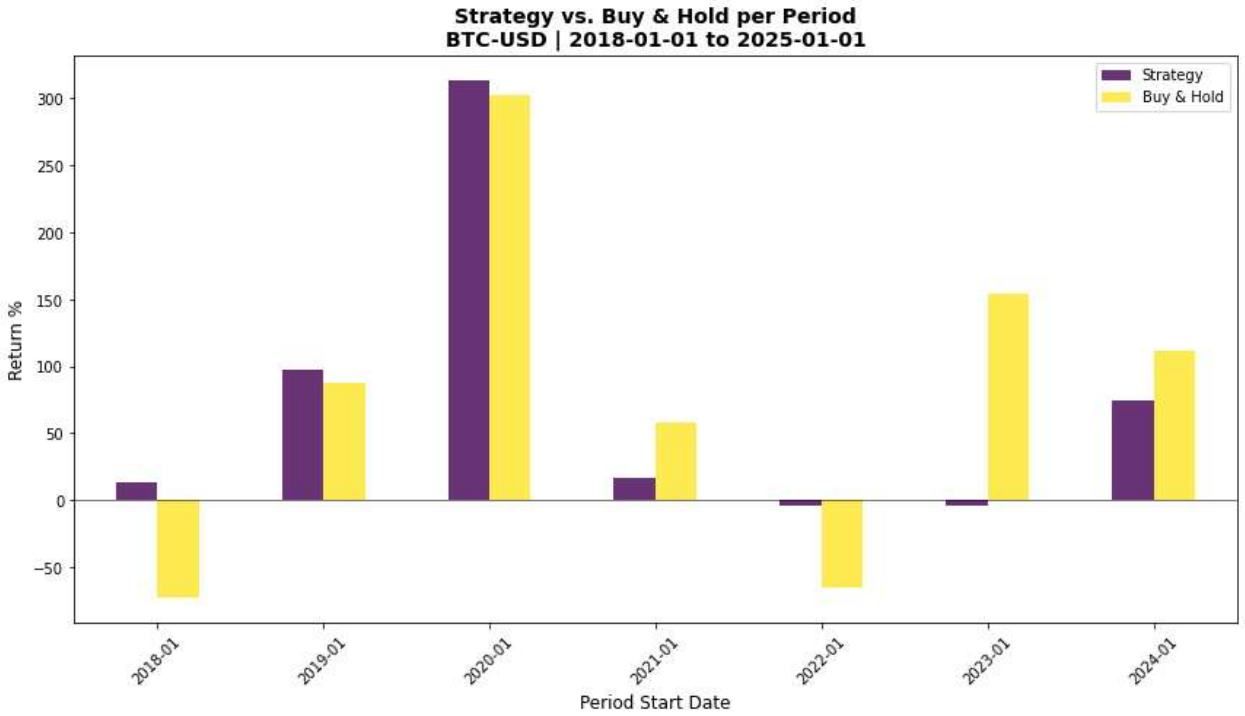
```
stop.
            if self.data.close[0] > self.stop_price: self.order =
self.close()
```

- **Order Check**: The method starts by checking if there's a pending order. If so, it returns to avoid placing duplicate orders.
- **Filter Conditions (No Position)**: If the strategy is not currently in a trade:
  - **Consolidation Check (`is_consolidating`)**: It checks if the current `price_stddev` (standard deviation of the closing price) normalized by the `data.close[0]` is below the `consolidation_threshold`. This identifies periods of low volatility.
  - **Macro Trend Alignment**: It checks if the current `data.close[0]` is above (`is_macro_uptrend`) or below (`is_macro_downtrend`) the `trend_sma`. This ensures trades are only taken in the direction of the broader trend.
  - **Momentum Breakout Detection**: If the market is indeed consolidating, it calculates upper and lower bands for the `roc` indicator based on its `roc_ma` and `roc_stddev`. `is_mom_breakout_up` (or `is_mom_breakout_down`) becomes true if the current `roc` breaks above (or below) these statistically significant bands, indicating a sudden surge in momentum.
- **Entry Logic (No Position)**: A `buy()` order is placed if the market is consolidating, in an uptrend, and experiences an upward momentum breakout. A `sell()` order is placed if consolidating, in a downtrend, and experiences a downward momentum breakout.
- **Trailing Stop Logic (In Position)**: If the strategy is in a long position, it continuously updates `self.highest_price_since_entry` and moves the `self.stop_price` upwards, trailing the price at a distance determined by `atr` and `atr_stop_multiplier`. If the price falls below this trailing stop, the position is closed. A similar logic applies for short positions, updating `self.lowest_price_since_entry` and moving the stop downwards.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	5
Losing Periods	2
Mean Return %	72.35
Median Return %	16.43
Std Dev %	104.79
Win Rate %	71.43
Sharpe Ratio	0.69
Min Return %	-4.25
Max Return %	313.18





Simple Volatility Momentum Strategy

- **Logic and Idea**: This strategy operates on the principle that when **volatility accelerates**, the price tends to move strongly in its current direction. It identifies "**volatility momentum**" by comparing the current volatility (standard deviation of returns) to its value a few periods ago. If volatility is increasing and the price is above/below a Simple Moving Average (SMA) for trend confirmation, a trade is initiated in the direction of the price trend. **ATR-based stop-losses** are used for risk management, and positions are also exited if volatility momentum ceases.

- **Main Parts of the Strategy Class Code (`SimpleVolatilityMomentumStrategy`)**:

  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('vol_window', 30),          # Volatility calculation period
(for std dev of returns)
    ('vol_momentum_window', 7),  # Vol momentum lookback
(difference between current vol and N bars ago)
    ('price_sma_window', 30),    # Price trend SMA
    ('atr_window', 14),          # ATR stop loss period
    ('atr_multiplier', 5.0),     # ATR stop multiplier
)
```

    - `vol_window`: The period used for calculating price volatility (standard deviation of returns).

- `vol_momentum_window`: The lookback period used to calculate "volatility momentum" by comparing the current volatility to the volatility `N` bars ago.
- `price_sma_window`: The period for the Simple Moving Average (SMA) of the closing price, used to determine the general price trend.
- `atr_window`: The period for the Average True Range (ATR), used to calculate the initial and trailing stop-loss distances.
- `atr_multiplier`: A multiplier applied to the ATR value to set the stop-loss distance.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Need enough data for indicators to warm up
    if len(self) < 30: # Or max(self.params.vol_window, etc) for
robustness
        return

    # Current values of indicators
    vol_momentum = self.vol_momentum[0]
    current_price = self.data.close[0]
    sma_price = self.price_sma[0]

    # --- Check Stop Loss (highest priority) ---
    if self.position: # If currently in a position
        if self.position.size > 0 and current_price <=
self.stop_price: # Long position hit stop
            self.close() # Close the position
            self.log(f'STOP LOSS - Long closed at
{current_price:.2f}')
            return # Exit this bar's logic
        elif self.position.size < 0 and current_price >=
self.stop_price: # Short position hit stop
            self.close() # Close the position
            self.log(f'STOP LOSS - Short closed at
{current_price:.2f}')
            return # Exit this bar's logic

    # --- Exit if volatility stops accelerating (if already in
trade) ---
    if self.position and vol_momentum <= 0: # If in a position
and vol momentum is non-positive
        self.close() # Close the position
        self.log(f'VOL MOMENTUM EXIT - Vol momentum:
{vol_momentum:.6f}')
        return # Exit this bar's logic
```

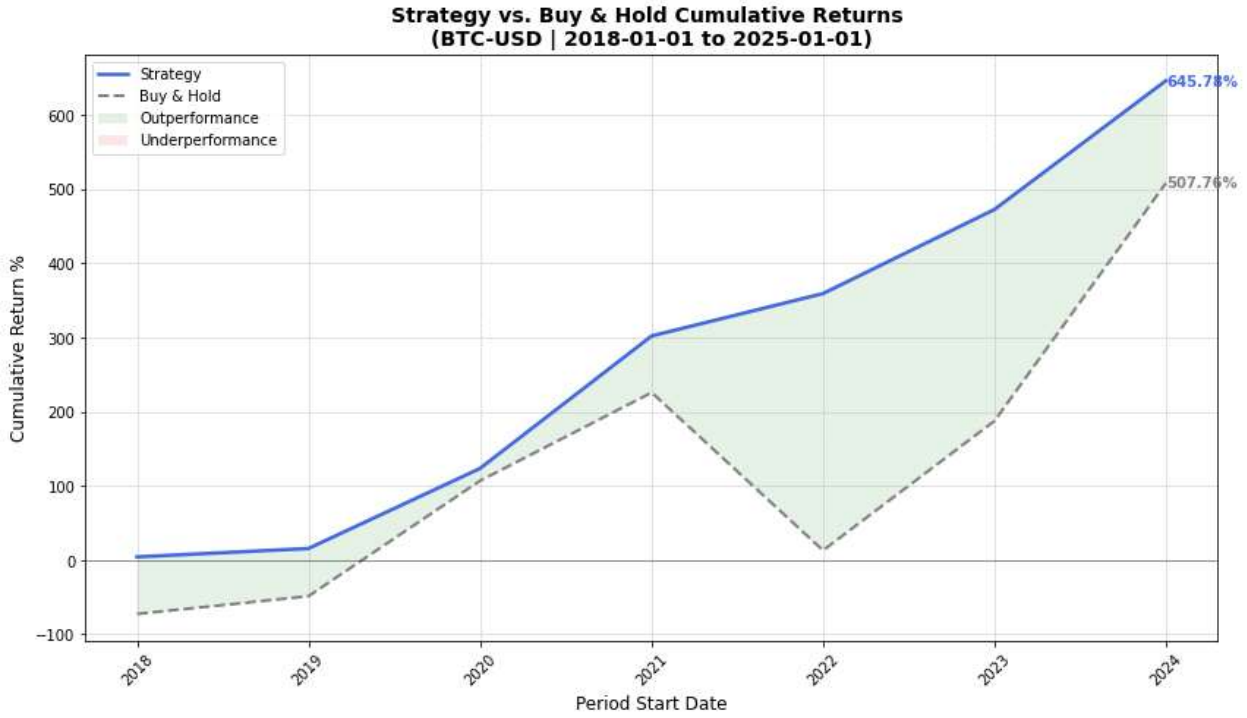
```
    # --- Entry signals: Vol accelerating + price direction (if
no position) ---
    if not self.position and vol_momentum > 0: # If no position
and volatility is accelerating

        # Long: Vol accelerating + price above SMA
        if current_price > sma_price: # Price is above SMA
(uptrend)
            self.buy() # Place a buy order
            # Calculate and set initial stop price
            self.stop_price = current_price - (self.atr[0] *
self.params.atr_multiplier)
            self.trade_count += 1 # Increment trade counter
            self.log(f'LONG - Price: {current_price:.2f}, Vol
Mom: {vol_momentum:.6f}, Stop: {self.stop_price:.2f}')

        # Short: Vol accelerating + price below SMA
        elif current_price < sma_price: # Price is below SMA
(downtrend)
            self.sell() # Place a sell order
            # Calculate and set initial stop price
            self.stop_price = current_price + (self.atr[0] *
self.params.atr_multiplier)
            self.trade_count += 1 # Increment trade counter
            self.log(f'SHORT - Price: {current_price:.2f}, Vol
Mom: {vol_momentum:.6f}, Stop: {self.stop_price:.2f}')

    # --- Update trailing stops (if in position) ---
    if self.position: # If already in a position
        if self.position.size > 0:  # Long position
            new_stop = current_price - (self.atr[0] *
self.params.atr_multiplier) # Calculate new potential stop
            if new_stop > self.stop_price: # If new stop is
higher, update it (trailing)
                self.stop_price = new_stop

        elif self.position.size < 0:  # Short position
            new_stop = current_price + (self.atr[0] *
self.params.atr_multiplier) # Calculate new potential stop
            if new_stop < self.stop_price: # If new stop is
lower, update it (trailing)
                self.stop_price = new_stop
```

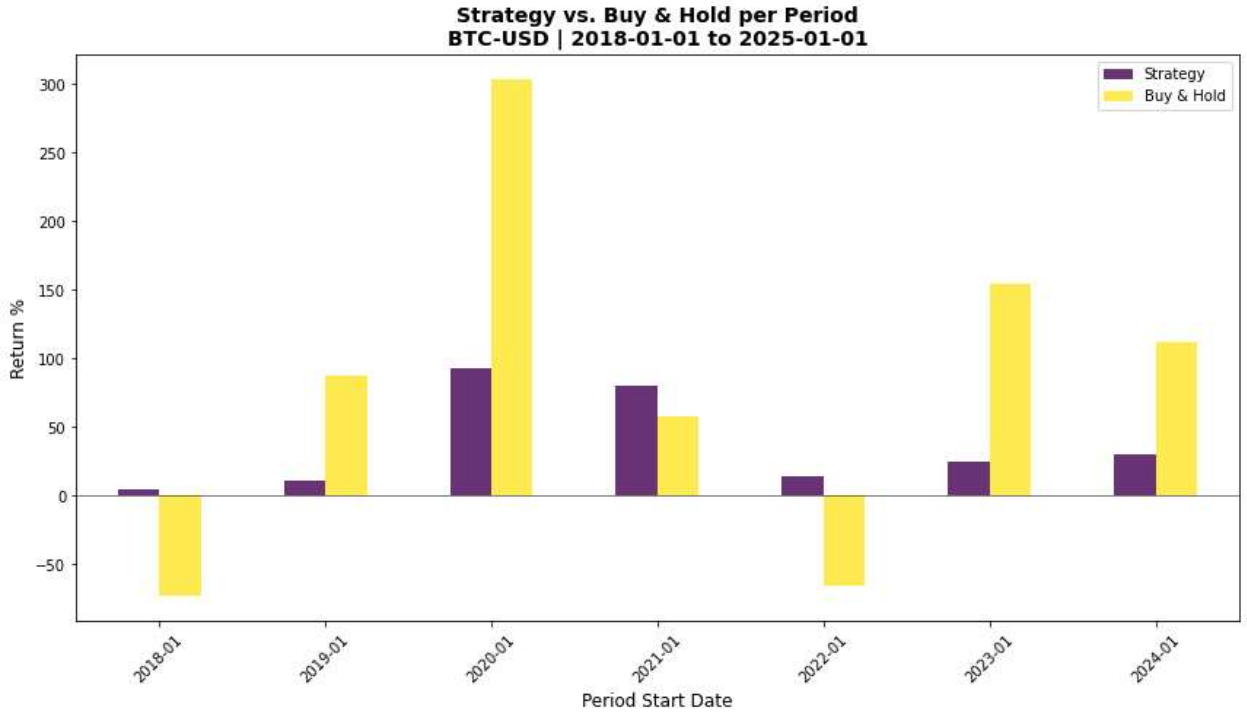
- **Data Sufficiency and Initial Checks**: The method first checks if enough historical data is available for indicator calculations.
- **Stop Loss Check (Highest Priority)**: If an open position exists, it first checks if the current price has hit the `self.stop_price`. If so, the position is closed, and the method returns.

- **Volatility Momentum Exit (If in Position)**: If the strategy is in a position, it also checks if `vol_momentum[0]` is no longer positive (i.e., volatility has stopped accelerating or is decelerating). If this condition is met, the position is closed to avoid holding a trade when momentum dissipates.
- **Entry Signals (No Position)**: If no position is open and `vol_momentum[0]` is greater than 0 (volatility is accelerating):
  - **Long Entry**: A `buy()` order is placed if the `current_price` is above the `sma_price` (indicating an uptrend). An initial `self.stop_price` is set using the ATR.
  - **Short Entry**: A `sell()` order is placed if the `current_price` is below the `sma_price` (indicating a downtrend). An initial `self.stop_price` is set using the ATR.
- **Trailing Stop Update (If in Position)**: If the strategy is in an open position, the `self.stop_price` is continuously updated to trail the current price, moving upwards for long positions and downwards for short positions, ensuring profits are protected as the trade moves favorably.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	7
Losing Periods	0
Mean Return %	36.79
Median Return %	24.68
Std Dev %	32.70
Win Rate %	100.00
Sharpe Ratio	1.13
Min Return %	4.25
Max Return %	93.21





Statistically Validated Regression Channel Breakout Strategy

- **Logic and Idea**: This strategy trades breakouts from a **Linear Regression Channel**, but with a crucial "**statistical validation**" step. It doesn't just enter on any channel breakout; it requires the breakout candle to have an unusually large bar range (High

- Low) and high volume, both exceeding a statistically significant threshold (e.g., average + N standard deviations). This aims to confirm that the breakout is strong and genuine, reducing false signals. A fixed percentage stop-loss is used, and positions are also exited if the price crosses back over the channel's midline (mean reversion exit).

- **Main Parts of the Strategy Class Code (`StatValidatedRegChannelBreakout`)**:
    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('channel_period', 30),    # Lookback for regression channel
(for linear regression and standard deviation)
    ('channel_mult', 1.),      # Std Dev multiplier for channel
width (how wide bands are from midline)
    ('valid_period', 7),       # Lookback for range/volume
validation statistics (avg and std dev)
    ('valid_mult', 1.2),       # Std Dev multiplier for
range/volume validation (how many std dev above avg for
validation)
    ('stop_loss_perc', 0.05),  # Stop loss percentage (e.g., 0.03
= 3%)
    ('printlog', False),
)
```

        - `channel_period`: The lookback period for calculating the Linear Regression Channel (midline, upper, and lower bands).
        - `channel_mult`: A multiplier applied to the standard deviation of price to determine the width of the channel bands around the midline.
        - `valid_period`: The lookback period for calculating the average and standard deviation of the bar range (High - Low) and volume. These statistics are used for the "statistical validation" of a breakout.
        - `valid_mult`: A multiplier applied to the standard deviation of range and volume. The current bar's range and volume must exceed their respective averages plus this multiple of their standard deviation to be considered "validated."
        - `stop_loss_perc`: A fixed percentage from the entry price used to set a static stop-loss order.
    - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Check if an order is pending or if we cannot calculate
indicators yet (e.g., not enough data points)
    if self.order or not
math.isfinite(self.regchannel.lines.upper[0]) or \
       not math.isfinite(self.avg_range[0]) or not
```

```
math.isfinite(self.std_range[0]) or \
       not math.isfinite(self.avg_volume[0]) or not
math.isfinite(self.std_volume[0]):
        return

    current_range = self.bar_range[0] # Current bar's High - Low
    current_volume = self.datavolume[0] # Current bar's volume

    # Define validation thresholds: average + (valid_mult * std
dev)
    range_threshold = self.avg_range[0] + self.p.valid_mult *
self.std_range[0]
    volume_threshold = self.avg_volume[0] + self.p.valid_mult *
self.std_volume[0]

    # Check if standard deviations are too low (near zero) to
avoid division by zero or nonsensical validation
    min_std_dev_threshold = 1e-9
    if self.std_range[0] < min_std_dev_threshold or
self.std_volume[0] < min_std_dev_threshold:
        is_validated = False # Cannot validate if std dev is
essentially zero
        self.log(f"WARN: Range or Volume StdDev too low
({self.std_range[0]:.2f}, {self.std_volume[0]:.2f}). Skipping
validation.", dt=self.datas[0].datetime.date(0))
    else:
        # A breakout is validated if both current range AND
current volume exceed their respective thresholds.
        is_validated = (current_range > range_threshold and
current_volume > volume_threshold)

    # --- Entry Logic ---
    if not self.position: # If no position is open
        # Check for Long Breakout: current close is above the
upper channel band
        if self.dataclose[0] > self.regchannel.lines.upper[0]:
            self.log(f'Potential LONG Breakout:
Close={self.dataclose[0]:.2f} >
Upper={self.regchannel.lines.upper[0]:.2f}')
            if is_validated: # If breakout is statistically
validated
                self.log(f'--> VALIDATED:
Range={current_range:.2f} > Thr={range_threshold:.2f},
Vol={current_volume:.0f} > Thr={volume_threshold:.0f}')
                self.log(f'>>> Placing BUY Order')
                if self.stop_order: self.cancel(self.stop_order)
# Cancel any existing stop
                self.order = self.buy() # Place buy order
            else:
```

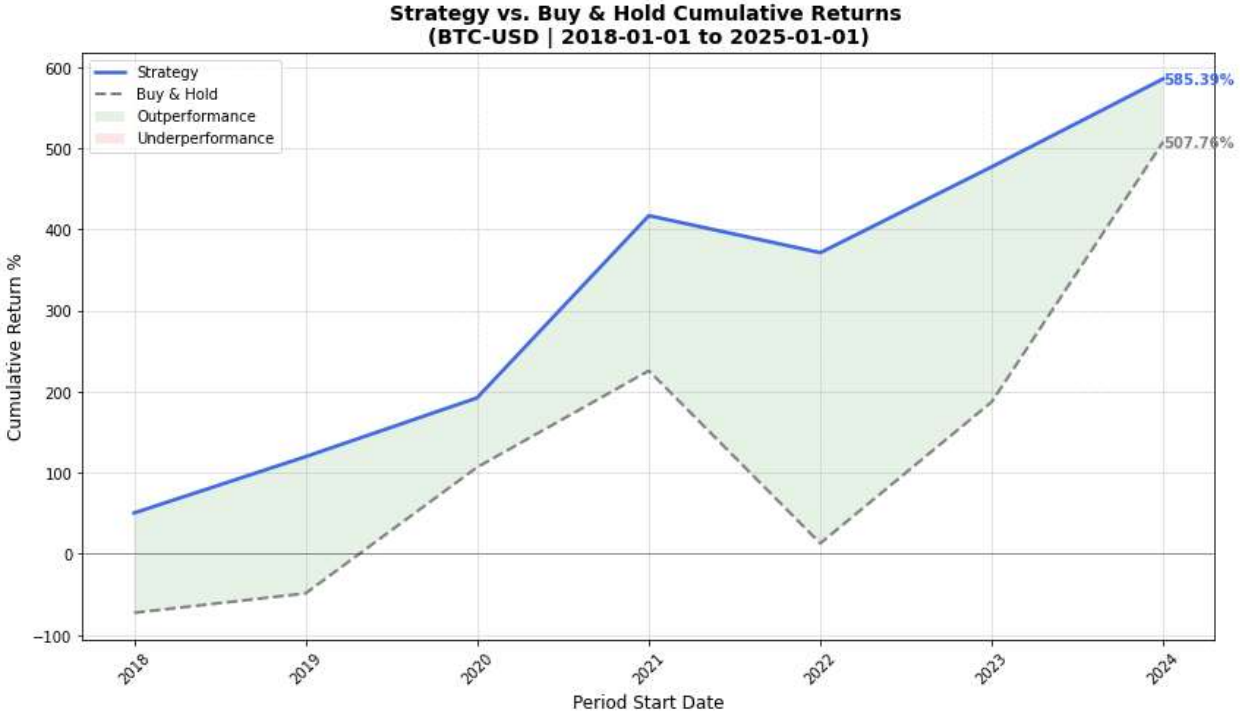
```
                self.log(f'--> NOT Validated:
Range={current_range:.2f} <= Thr={range_threshold:.2f} or
Vol={current_volume:.0f} <= Thr={volume_threshold:.0f}')

        # Check for Short Breakout: current close is below the
lower channel band
        elif self.dataclose[0] < self.regchannel.lines.lower[0]:
            self.log(f'Potential SHORT Breakout:
Close={self.dataclose[0]:.2f} <
Lower={self.regchannel.lines.lower[0]:.2f}')
            if is_validated: # If breakout is statistically
validated
                self.log(f'--> VALIDATED:
Range={current_range:.2f} > Thr={range_threshold:.2f},
Vol={current_volume:.0f} > Thr={volume_threshold:.0f}')
                self.log(f'>>> Placing SELL Order')
                if self.stop_order: self.cancel(self.stop_order)
# Cancel any existing stop
                self.order = self.sell() # Place sell order
            else:
                 self.log(f'--> NOT Validated:
Range={current_range:.2f} <= Thr={range_threshold:.2f} or
Vol={current_volume:.0f} <= Thr={volume_threshold:.0f}')

    # --- Exit Logic (if in position) ---
    else: # If a position is open
        # Exit long position if price crosses back below the
midline (mean reversion)
        if self.position.size > 0 and self.dataclose[0] <
self.regchannel.lines.midline[0]:
            self.log(f'Midline CLOSE LONG SIGNAL: Close
{self.dataclose[0]:.2f} < Midline
{self.regchannel.lines.midline[0]:.2f}')
            if self.stop_order: # Cancel existing stop loss
first
                self.log(f'Cancelling Stop Order Ref:
{self.stop_order.ref} before closing.')
                self.cancel(self.stop_order)
                self.stop_order = None
            self.order = self.close() # Place the close order
        # Exit short position if price crosses back above the
midline (mean reversion)
        elif self.position.size < 0 and self.dataclose[0] >
self.regchannel.lines.midline[0]:
            self.log(f'Midline CLOSE SHORT SIGNAL: Close
{self.dataclose[0]:.2f} > Midline
{self.regchannel.lines.midline[0]:.2f}')
            if self.stop_order: # Cancel existing stop loss
first
                self.log(f'Cancelling Stop Order Ref:
```

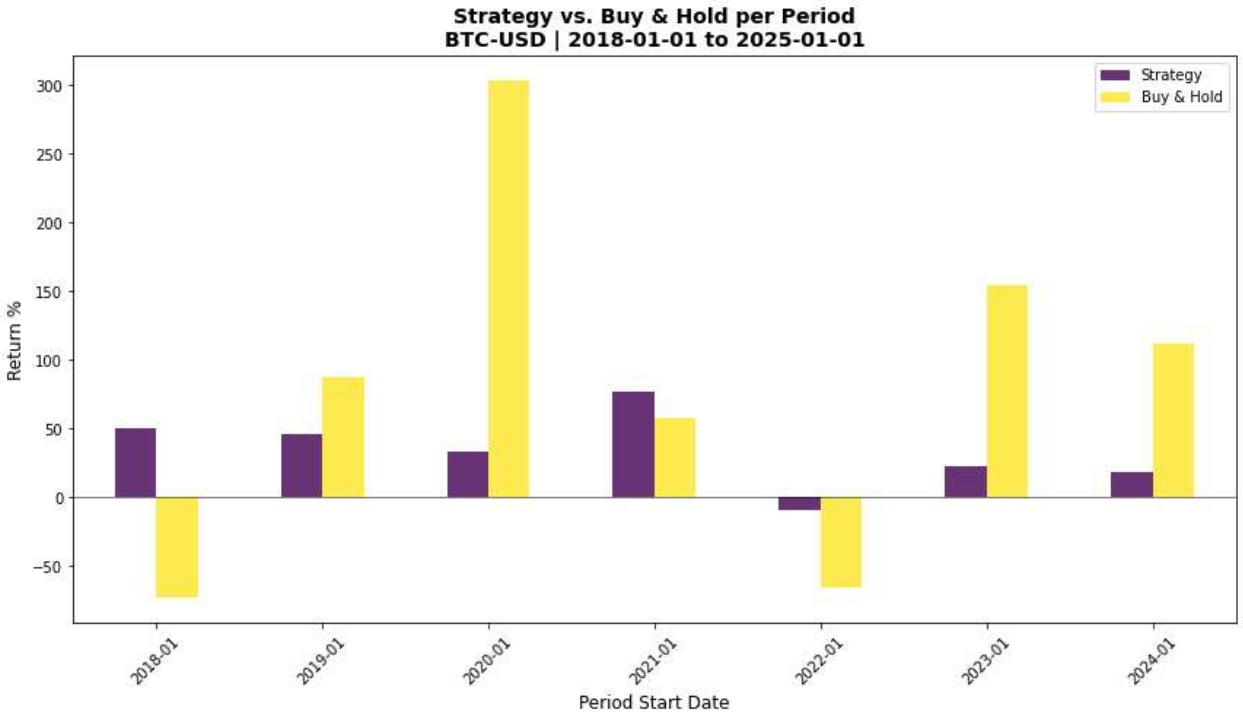
```
{self.stop_order.ref} before closing.')
                    self.cancel(self.stop_order)
                    self.stop_order = None
                self.order = self.close() # Place the close order
```

- **Initialization Checks**: The method begins by ensuring no orders are pending and that all necessary indicator values (from `regchannel`, `avg_range`, `std_range`, `avg_volume`, `std_volume`) are finite and available. It also logs a warning if standard deviations are too low, which can indicate insufficient data for validation.
- **Validation Thresholds**: `range_threshold` and `volume_threshold` are calculated. These represent a dynamic bar range and volume level that, if exceeded, indicate a statistically significant move. They are derived from the average and standard deviation of historical range and volume, multiplied by `valid_mult`.
- **Statistical Validation (`is_validated`)**: This boolean is `True` only if the `current_range` of the bar *and* its `current_volume` both exceed their respective calculated thresholds. This is the core filtering mechanism to confirm genuine breakouts.
- **Entry Logic (No Position)**: If no position is open:
  - **Long Breakout**: If the current `dataclose[0]` breaks above the `regchannel.lines.upper[0]` AND `is_validated` is true, a `buy()` order is placed.
  - **Short Breakout**: If the current `dataclose[0]` breaks below the `regchannel.lines.lower[0]` AND `is_validated` is true, a `sell()` order is placed. Any existing `self.stop_order` is canceled before placing the new entry.
- **Exit Logic (In Position)**: If the strategy is in an open position:
  - **Mean Reversion Exit**: If a long position is open and `dataclose[0]` falls below `regchannel.lines.midline[0]`, or if a short position is open and `dataclose[0]` rises above `regchannel.lines.midline[0]`, the position is closed. This acts as a profit-taking or loss-cutting mechanism if the initial breakout fails to sustain and the price reverts to the channel's mean. Any active stop loss is canceled before this manual close.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	34.11
Median Return %	32.95
Std Dev %	25.23
Win Rate %	85.71
Sharpe Ratio	1.35
Min Return %	-8.84
Max Return %	76.88




Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
300
250
200
150
100
50
0
−50
2018-01
2019-01
2020-01
2021-01
2022-01
2023-01
2024-01
Period Start Date

5. Advanced and Hybrid Strategies

Combine elements from different categories (e.g., trend-following with mean-reversion) or incorporate more complex techniques like machine learning, regime switching, or Kalman filtering to adapt dynamically to market conditions.

Kalman Filter Trend Strategy

- **Logic and Idea**: This advanced strategy uses a **Kalman Filter**, a powerful algorithm for estimating the state of a dynamic system from noisy measurements. In trading, it can be used to estimate the "true" price and its velocity (trend) by filtering out market noise. The strategy generates signals based on the estimated velocity: a positive velocity indicates an uptrend, and a negative velocity indicates a downtrend. A **trailing stop** is used for risk management.

- **Main Parts of the Strategy Class Code (`KalmanFilterTrendWithTrail`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

      ```
      params = (
          ('process_noise', 1e-3), # Controls the uncertainty in the
      model's prediction of the system's state.
          ('measurement_noise', 1e-1), # Controls the uncertainty in
      the actual price measurement.
          ('trail_percent', 0.02), # Percentage for the trailing stop-
      loss.
      )
      ```

        - `process_noise`: A parameter influencing the Kalman Filter's `Q` matrix, which represents the uncertainty in the system's process model (how the underlying price and velocity are assumed to evolve). A higher value makes the filter more responsive to changes but potentially more noisy.
        - `measurement_noise`: A parameter influencing the Kalman Filter's `R` matrix, representing the uncertainty in the actual price measurement. A higher value makes the filter smoother but less responsive to current price.
        - `trail_percent`: The percentage used to calculate the trailing stop-loss from the high/low of the trade, applied once an entry is completed.
    - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

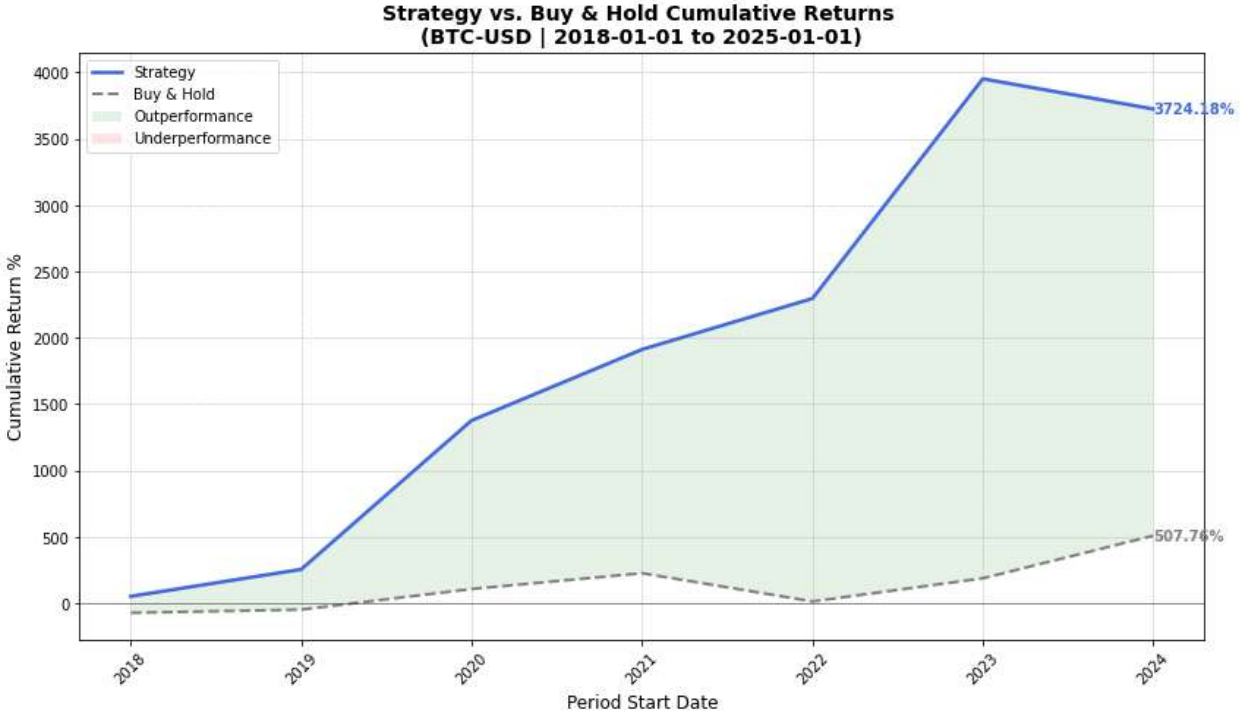
      ```
      def next(self):
          if self.order: # If there's an existing order pending, do
      nothing
              return
      ```

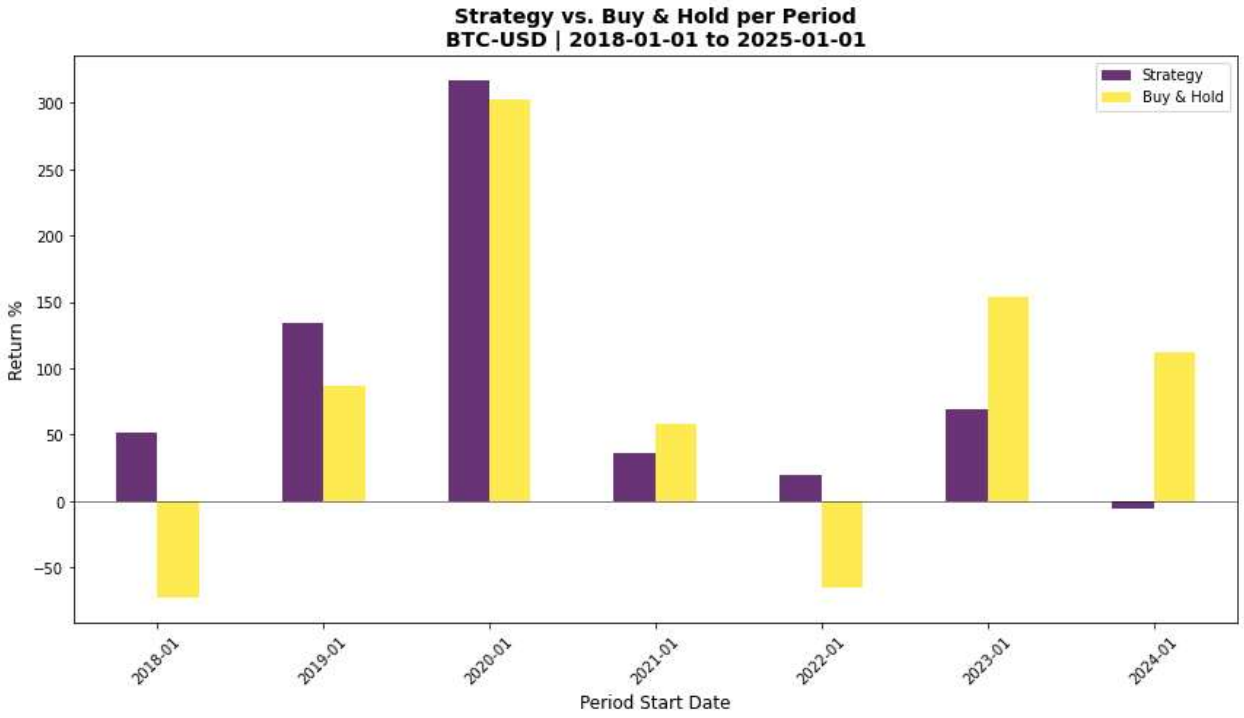
```
    if len(self.kf_velocity) == 0: # Ensure Kalman Filter has
calculated values
        return

    estimated_velocity = self.kf_velocity[0] # Get the current
estimated velocity
    current_position_size = self.position.size

    if current_position_size == 0: # If not currently in a
position
        if self.stop_order: # If there's a lingering stop order
(e.g., from a previous manual close), cancel it
            self.cancel(self.stop_order)
            self.stop_order = None

        if estimated_velocity > 0: # If velocity is positive
(uptrend)
            self.order = self.buy() # Place a buy order
        elif estimated_velocity < 0: # If velocity is negative
(downtrend)
            self.order = self.sell() # Place a sell order
```

- **Order and Data Check**: The method first checks if `self.order` is pending and ensures that the Kalman Filter's `kf_velocity` line has been populated with data.
- **Velocity Retrieval**: It retrieves the `estimated_velocity` from the `kf_velocity` line of the custom `KalmanFilterIndicator`. This velocity is the core signal from the Kalman Filter, indicating the estimated trend direction.
- **Entry Logic (No Position)**: If the strategy is not currently in a position (`current_position_size == 0`):
  - Any existing `self.stop_order` is canceled (this handles scenarios where a stop order might remain after a manual closing of a position).
  - If `estimated_velocity` is greater than 0, indicating an upward trend, a `buy()` order is placed.
  - If `estimated_velocity` is less than 0, indicating a downward trend, a `sell()` order is placed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	88.69
Median Return %	51.53
Std Dev %	101.58
Win Rate %	85.71
Sharpe Ratio	0.87
Min Return %	-5.60
Max Return %	316.62





OBV Momentum Strategy

- **Logic and Idea**: This strategy combines **On-Balance Volume (OBV)** with other filters to identify momentum-driven trades. OBV is a cumulative indicator that relates volume to price changes: rising OBV suggests buying pressure, falling OBV suggests selling pressure. The strategy looks for OBV crossovers with its own Moving Average (MA) as the primary signal, but filters these signals with **RSI** (to avoid overbought/oversold conditions) and a **volume average** (to ensure significant volume accompanies the move). A **trailing stop** is used for risk management.

- **Main Parts of the Strategy Class Code (`OBVmomentumStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('obv_ma_period', 30),
    ('trail_percent', 0.02),
    ('rsi_period', 14),
    ('volume_ma_period', 7),
)
```

        - `obv_ma_period`: The period for the Simple Moving Average (SMA) applied to the OBV line. The crossover of OBV with its MA is a primary signal.
        - `trail_percent`: The percentage used for the trailing stop-loss order once a position is entered.

- `rsi_period`: The period for the Relative Strength Index (RSI), used as an oscillator to filter out overbought or oversold conditions.
- `volume_ma_period`: The period for a Simple Moving Average (SMA) of the trading volume, used to confirm that momentum signals are accompanied by significant trading activity.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    if self.order: # Check if there is already a pending order
        return

    if not self.position: # Logic when not currently in a trade
        # Long signal: OBV crosses up + RSI not overbought + volume above average
        if (self.obv_cross[0] > 0.0 and # OBV crosses above its MA (bullish cross)
            self.rsi[0] < 70 and # RSI is not in overbought territory (below 70)
            self.data.volume[0] > self.volume_ma[0]): # Current volume is above its moving average
            self.order = self.buy() # Place a buy order

        # Short signal: OBV crosses down + RSI not oversold + volume above average
        elif (self.obv_cross[0] < 0.0 and # OBV crosses below its MA (bearish cross)
              self.rsi[0] > 30 and # RSI is not in oversold territory (above 30)
              self.data.volume[0] > self.volume_ma[0]): # Current volume is above its moving average
            self.order = self.sell() # Place a sell order
```

- **Order Check**: The method begins by checking if `self.order` is not `None`, meaning an order is currently pending. If so, it returns to prevent placing duplicate orders.
- **Entry Logic (No Position)**: If the strategy is not holding any open position (`not self.position`):
  - **Long Signal**: A `buy()` order is placed if three conditions are met:
    1. `self.obv_cross[0] > 0.0`: The OBV line has just crossed *above* its moving average, indicating increasing buying pressure.
    2. `self.rsi[0] < 70`: The Relative Strength Index (RSI) is below 70, meaning the asset is not currently

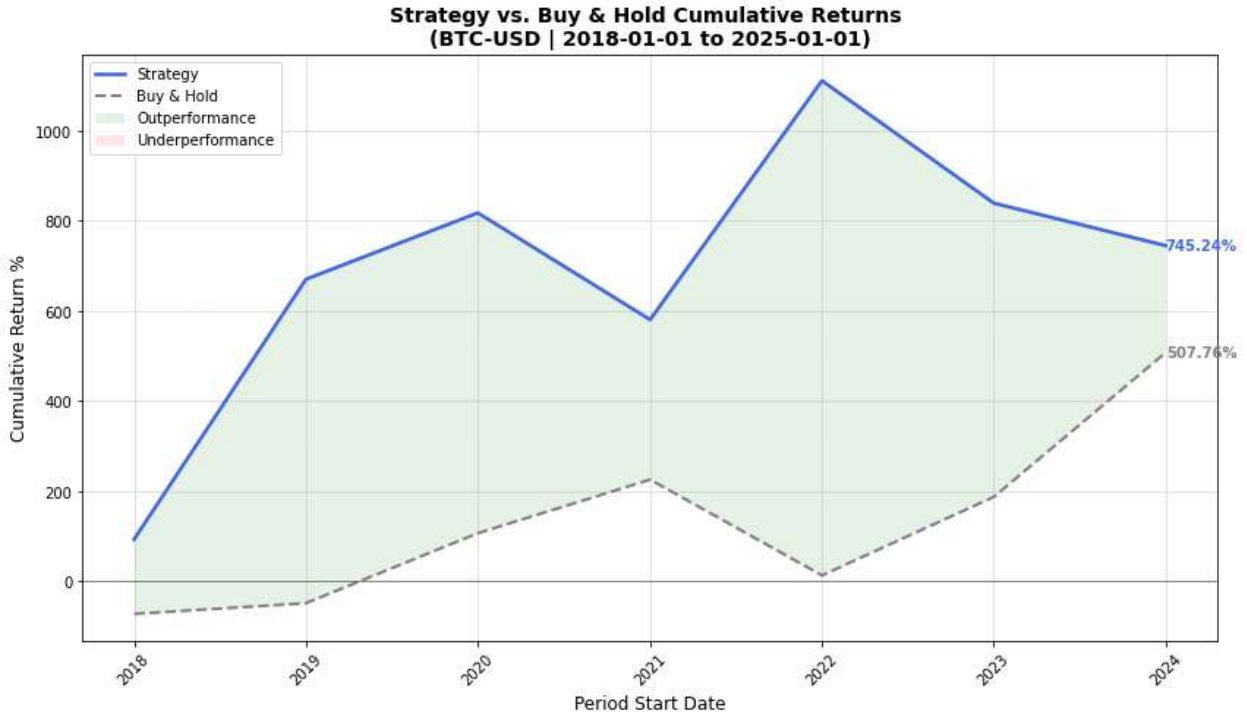
overbought, which helps to avoid entries at potential reversals.

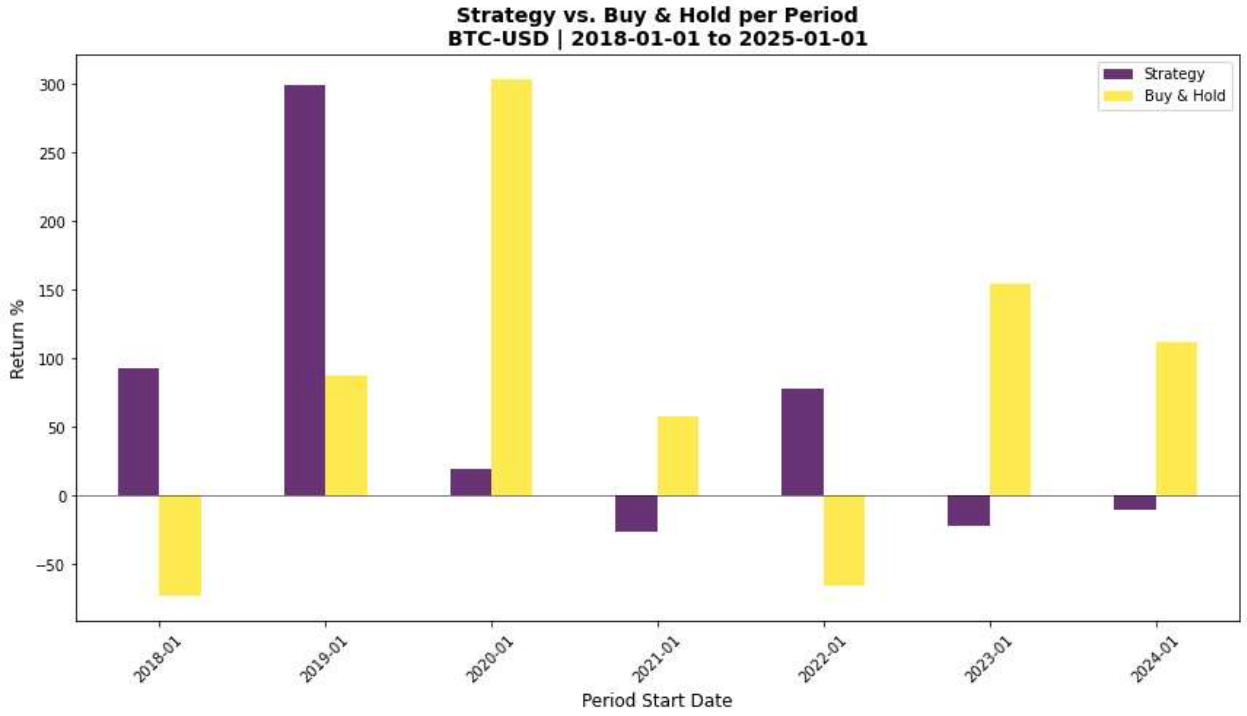
3. `self.data.volume[0] > self.volume_ma[0]`: The current trading volume is greater than its moving average, confirming that the price movement is supported by significant market activity.

- **Short Signal**: A `sell()` order is placed if three analogous conditions are met for a bearish signal:
  1. `self.obv_cross[0] < 0.0`: The OBV line has just crossed *below* its moving average, indicating increasing selling pressure.
  2. `self.rsi[0] > 30`: The RSI is above 30, meaning the asset is not currently oversold, avoiding entries at potential oversold bounces.
  3. `self.data.volume[0] > self.volume_ma[0]`: The current volume is above its moving average, confirming the bearish move.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	4
Losing Periods	3
Mean Return %	61.56
Median Return %	19.13
Std Dev %	106.45
Win Rate %	57.14
Sharpe Ratio	0.58
Min Return %	-25.84
Max Return %	299.13





RandomForest-Enhanced MA Ribbon Strategy

- **Logic and Idea**: This is a sophisticated hybrid strategy that enhances the MA Ribbon Pullback Strategy by incorporating a **Machine Learning model (Random Forest Classifier)** to filter trade signals. The Random Forest is trained on various technical features derived from the MA ribbon and other indicators to predict future price

movements. A trade signal from the MA Ribbon is only acted upon if the Random Forest model's "confidence" (probability of a positive outcome) exceeds a certain threshold. This aims to improve signal quality and reduce false positives. The model is retrained periodically to adapt to changing market conditions.

- **Main Parts of the Strategy Class Code (`RandomForestEnhancedMaRibbonStrategy`)**:
  - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    # EXACT original parameters inherited from
MaRibbonPullbackStrategy
    ('ema_periods', (5, 8, 11, 14, 17, 20)),
    ('slope_period', 7),
    ('exit_ema_cross_short', 7),
    ('exit_ema_cross_long', 30),
    ('order_percentage', 0.95),
    ('ticker', 'BTC-USD'),
    ('min_slope_threshold', 0.001),
    # RandomForest parameters only
    ('rf_threshold', 0.65),      # RF confidence threshold for 3-
month data
    ('retrain_frequency', 25),   # Retrain every 25 bars for 3-
month data
)
```

    - `ema_periods`, `slope_period`, `exit_ema_cross_short`, `exit_ema_cross_long`, `order_percentage`, `min_slope_threshold`: These are parameters inherited from the base `MaRibbonPullbackStrategy`, defining the MA ribbon and exit conditions.
    - `rf_threshold`: The minimum confidence (predicted probability of a positive outcome) from the Random Forest model required for a trade signal to be accepted.
    - `retrain_frequency`: How often (in number of bars) the Random Forest model is retrained on new data. This allows the model to adapt to changing market conditions.
  - **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    # Collect features for RF training (minimal overhead)
    features = self.calculate_features() # Extract features from
indicators
    if features is not None and len(self.data) > 35: # Ensure
enough data and valid features
        target = self.calculate_target_label() # Determine the
```

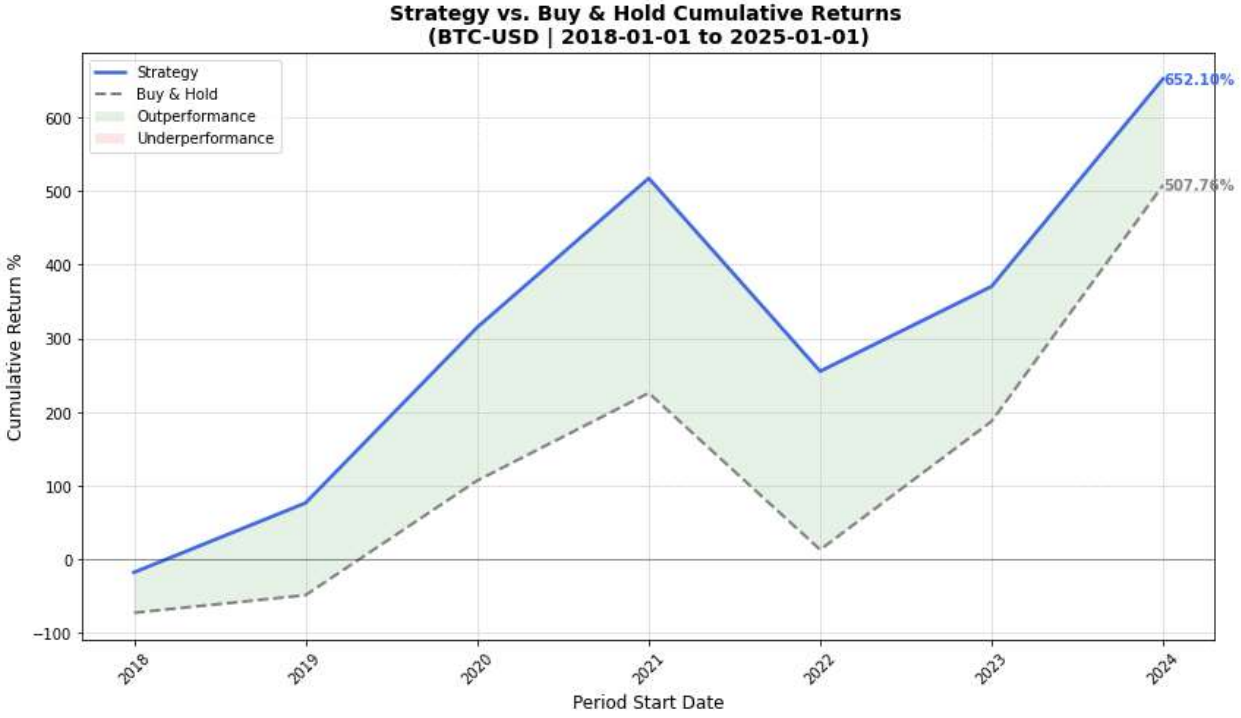
```
target label for training
            self.feature_buffer.append(features) # Add features to
buffer
            self.label_buffer.append(target) # Add label to buffer

            # Keep buffer size manageable for memory/performance
            if len(self.feature_buffer) > 80:
                self.feature_buffer = self.feature_buffer[-60:]
                self.label_buffer = self.label_buffer[-60:]

        # Retrain RF frequently for dynamic markets
        if len(self.data) - self.last_retrain >=
self.params.retrain_frequency:
            if self.train_random_forest(): # Attempt to train the
model
                self.last_retrain = len(self.data) # Update last
retrain bar if successful

        # EXACT original data check for MA Ribbon indicators
        if len(self.data_close) < max(self.params.ema_periods) +
self.params.slope_period:
            return

        # EXACT original expansion state of MA Ribbon
        is_expanding_up = (self.ema_fastest[0] > self.ema_slowest[0]
and
                           self.slowest_ema_slope[0] >
self.params.min_slope_threshold)

        # EXACT original pullback touch detection
        pullback_touch = self.data_low[0] <= self.ema_fastest[0]

        # --- Entry Logic (with RF enhancement) ---
        if not self.position: # If no position is open
            if is_expanding_up and pullback_touch: # If MA Ribbon
signal is present
                self.total_signals += 1 # Count all potential signals
                # RF ENHANCEMENT: Get confidence from Random Forest
                rf_confidence = self.get_rf_confidence(features) #
Get prediction probability

                # Enter trade ONLY if RF model is ready OR confidence
meets threshold
                if not self.rf_ready or rf_confidence >
self.params.rf_threshold:
                    self.log(f'BUY SIGNAL (Pullback):
Close={self.data_close[0]:.2f}, Low={self.data_low[0]:.2f},
FastEMA={self.ema_fastest[0]:.2f},
Slope={self.slowest_ema_slope[0]:.3f}, RF: {rf_confidence:.3f}')
```

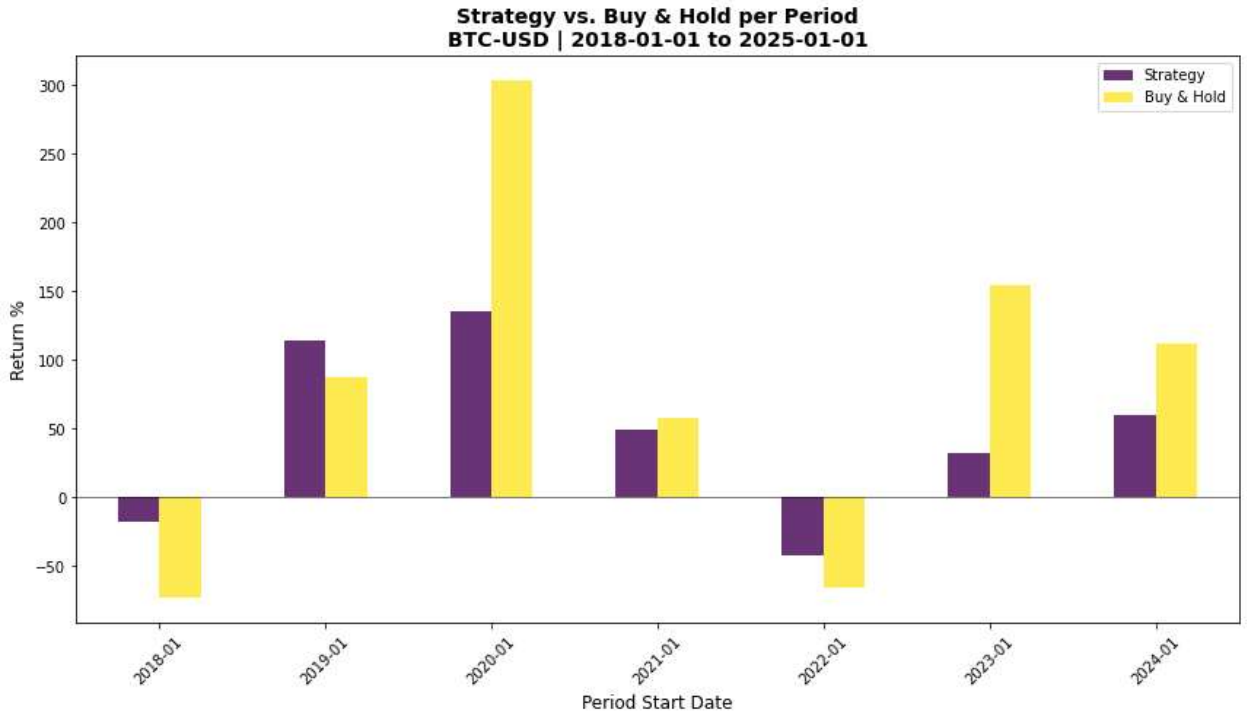
```
                cash = self.broker.get_cash()
                size = (cash * self.params.order_percentage) /
self.data_close[0]
                self.log(f'Calculating Buy Size: Cash={cash:.2f},
Close={self.data_close[0]:.2f},
Percentage={self.params.order_percentage}, Size={size:.6f}')
                self.order = self.buy(size=size)
            else:
                self.rf_filtered_signals += 1 # Count signals
filtered by RF
                self.log(f'PULLBACK signal filtered - RF
confidence {rf_confidence:.3f} < {self.params.rf_threshold}')

    # --- Exit Logic (EXACT original logic) ---
    else: # We are in a long position
        if self.exit_crossover < 0: # If bearish EMA crossover
occurs
            self.log(f'SELL SIGNAL (Exit - EMA Cross):
Close={self.data_close[0]:.2f}')
            self.order = self.close() # Close the position
```

- **Data Collection and RF Training**: On each bar, features are calculated and appended to `feature_buffer` and `label_buffer`. These buffers store historical data for the Random Forest. The buffers are kept to a manageable size. The `train_random_forest()` method is called periodically based on `retrain_frequency` to update the model.
- **MA Ribbon Signal**: The method checks for the core MA Ribbon conditions: `is_expanding_up` (ribbon fanning out) and `pullback_touch` (price pulling back to the fastest EMA).
- **Random Forest Enhancement (Entry Logic)**: If a potential MA Ribbon signal is present, the strategy:
  - Increments `self.total_signals`.
  - Calls `self.get_rf_confidence(features)` to get the predicted probability (confidence) from the Random Forest model for a positive outcome.
  - A `buy()` order is placed *only if* the `rf_ready` flag is true (model is trained) OR the `rf_confidence` is greater than the `rf_threshold`. This is the filtering step. If the signal is not confirmed by the RF, `self.rf_filtered_signals` is incremented.
- **Exit Logic**: The exit condition remains the same as the base MA Ribbon strategy: closing the position if the faster exit EMA crosses below the slower exit EMA.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	5
Losing Periods	2
Mean Return %	48.57
Median Return %	48.90
Std Dev %	55.48
Win Rate %	71.43
Sharpe Ratio	0.88
Min Return %	-36.02
Max Return %	126.35





Rough Path Momentum Strategy

- **Logic and Idea**: This is a highly advanced strategy that applies concepts from "**rough path theory**," a branch of mathematics used to analyze and integrate paths (like price series) that are not necessarily smooth. The core idea is to extract "**path signatures**" – a sequence of iterated integrals that capture the geometric and dynamic properties of a price path, including its momentum, volatility, and correlation structure, in a more robust way than traditional indicators. The strategy generates signals based on these momentum signatures and uses a **trailing stop** for risk management. It also checks for "signature invariance" (stability) to confirm genuine trends.

- **Main Parts of the Strategy Class Code (`RoughPathMomentumStrategy`)**:

    - **`params`**: This tuple defines the configurable parameters for the strategy.

```
params = (
    ('signature_window', 30),   # Window for path signature
calculation (number of past bars to consider)
    ('signature_depth', 3),     # Signature truncation level
(level of iterated integrals to compute, typically 2 or 3)
    ('momentum_threshold', 0.1), # Momentum signature threshold
(absolute value for entry)
    ('trailing_stop_pct', 0.05), # Percentage for the trailing
stop-loss
)
```

- `signature_window`: The number of recent price return increments to use when calculating the path signature. This defines the "path" being analyzed.
- `signature_depth`: The truncation level for the path signature calculation. Higher depths capture more complex geometric information about the path, but are computationally intensive. A depth of 2 or 3 is common.
- `momentum_threshold`: An absolute threshold for the `momentum_signature`. The `momentum_signature` must exceed this value (either positively or negatively) to trigger an entry.
- `trailing_stop_pct`: The percentage used for the trailing stop-loss, calculated from the highest/lowest price seen since entry.

- **`next(self)`**: This method contains the main trading logic, executed on each new bar of data.

```
def next(self):
    self.update_trailing_stop() # Always update the trailing stop
if in a position

    if self.order is not None: # If a trade order is pending,
return.
        return

    # Store path increments (returns)
    if not np.isnan(self.returns[0]): # Ensure the current return
is not NaN
        self.path_increments.append(self.returns[0])

    # Keep only recent window for signature calculation
    if len(self.path_increments) > self.params.signature_window *
2: # Keep buffer slightly larger
        self.path_increments = self.path_increments[-
self.params.signature_window * 2:]

    # Skip if not enough data for signature calculation
    if len(self.path_increments) < self.params.signature_window:
        return

    # Calculate momentum signature for the most recent path
    recent_path = self.path_increments[-
self.params.signature_window:]
    momentum_sig = self.extract_momentum_signature(recent_path)
    self.momentum_signature = momentum_sig # Store for
analysis/debug

    # Check signature invariance (stability) - to confirm genuine
trends
```

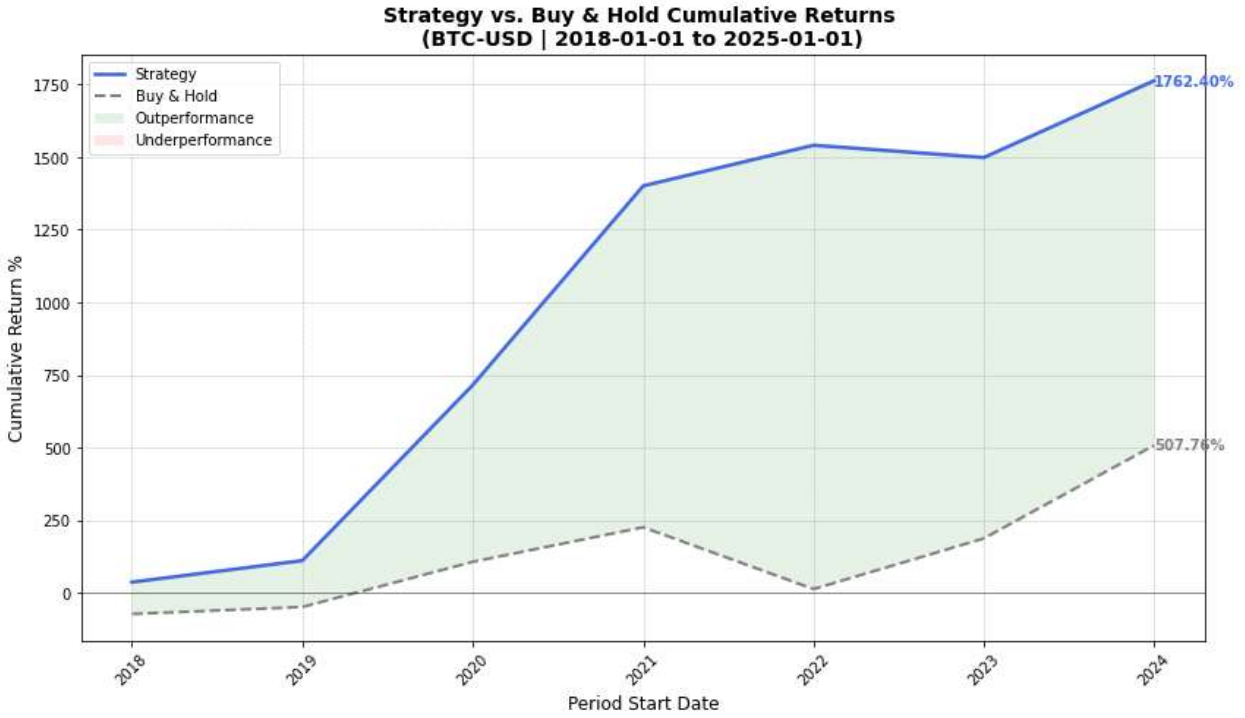
```
        is_stable = self.is_signature_invariant(recent_path)

        # Trading signals based on momentum signatures and stability
        if abs(momentum_sig) > self.params.momentum_threshold and
is_stable: # If momentum is strong AND path is stable

            # Strong positive momentum signature (bullish)
            if momentum_sig > self.params.momentum_threshold:
                if self.position.size < 0:  # If currently short,
close it
                    if self.stop_order is not None:
self.cancel(self.stop_order)
                    self.order = self.close()
                elif not self.position:  # If no position, go long
                    self.order = self.buy()

            # Strong negative momentum signature (bearish)
            elif momentum_sig < -self.params.momentum_threshold:
                if self.position.size > 0:  # If currently long,
close it
                    if self.stop_order is not None:
self.cancel(self.stop_order)
                    self.order = self.close()
                elif not self.position:  # If no position, go short
                    self.order = self.sell()
```

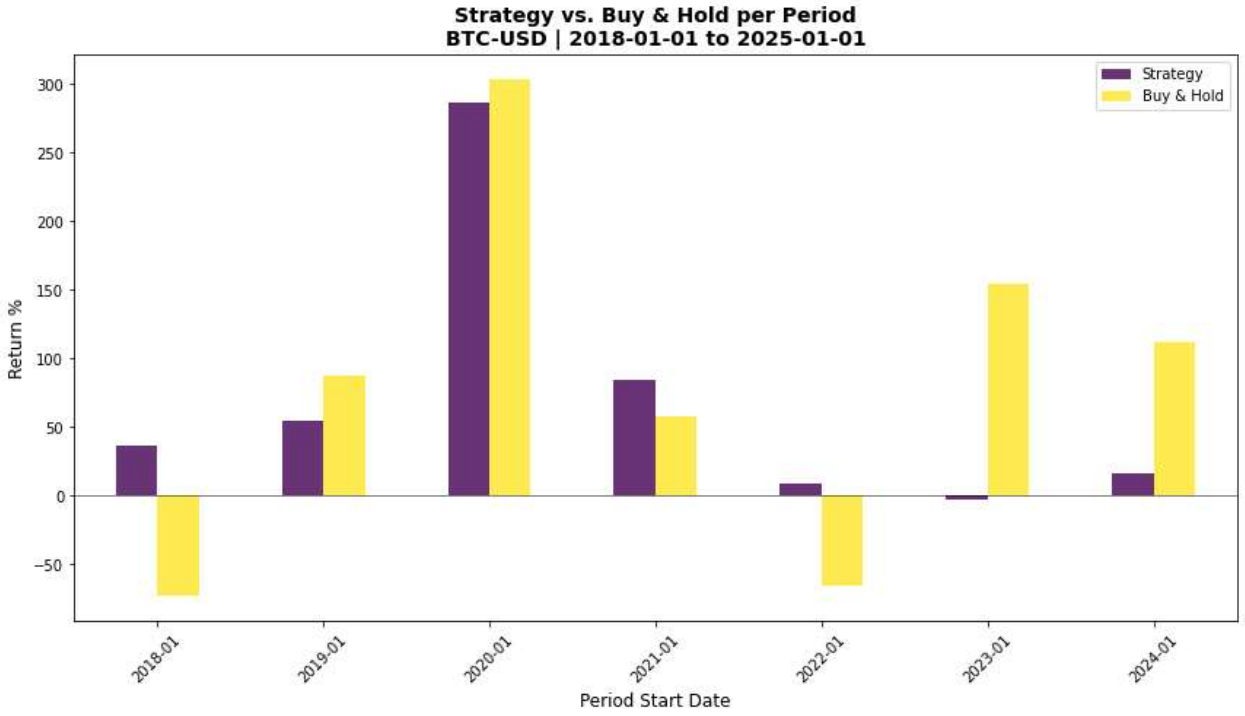
- **Trailing Stop Update**: The method calls `self.update_trailing_stop()` on every bar to ensure the trailing stop is adjusted if the price moves favorably.
- **Path Data Collection**: The current bar's percentage return is appended to `self.path_increments`. The buffer is trimmed to keep only the most recent data required for signature calculations.
- **Data Sufficiency Check**: It returns if there aren't enough increments in the `path_increments` list to form a `signature_window`-sized path.
- **Signature Calculation**: `recent_path` is extracted, and `self.extract_momentum_signature(recent_path)` is called to compute the aggregated momentum signature from the rough path integrals.
- **Signature Invariance Check (`is_stable`)**: `self.is_signature_invariant(recent_path)` is called to assess the stability of the path's underlying process. This acts as a filter to ensure that the detected momentum is likely to persist.
- **Trading Signals**: If the absolute value of the `momentum_sig` exceeds `momentum_threshold` AND the path is deemed `is_stable`:

- **Long Entry**: If `momentum_sig` is positive, and the strategy is either short (which is then closed) or flat, a `buy()` order is placed.
- **Short Entry**: If `momentum_sig` is negative, and the strategy is either long (which is then closed) or flat, a `sell()` order is placed. Any active stop order is canceled before a new entry is placed or an opposing position is closed.

Key Performance Indicators
BTC-USD | 2018-01-01 to 2025-01-01

Metric	Value
Total Periods	7
Winning Periods	6
Losing Periods	1
Mean Return %	69.24
Median Return %	36.83
Std Dev %	92.67
Win Rate %	85.71
Sharpe Ratio	0.75
Min Return %	-2.58
Max Return %	286.21


Strategy vs. Buy & Hold Cumulative Returns
(BTC-USD | 2018-01-01 to 2025-01-01)
Strategy
Buy & Hold
Outperformance
Underperformance
1762.40%
507.76%
Cumulative Return %
Period Start Date


Strategy vs. Buy & Hold per Period
BTC-USD | 2018-01-01 to 2025-01-01
Strategy
Buy & Hold
Return %
Period Start Date