An Algorithmic Exploration of Accumulation/Distribution Divergence

PyQuantLab Following 14 min read · Jun 10, 2025

In financial markets, price movements often appear to be the dominant surface phenomenon. Yet, beneath this visible activity, powerful currents of buying and selling pressure — accumulation and distribution — are constantly at play. Understanding these underlying forces is a recurring quest for many market participants. Does Smart Money quietly buy up assets during a downtrend (accumulation), setting the stage for a reversal? Or do they discreetly sell off their holdings during a rally (distribution), anticipating a downturn?

The Accumulation/Distribution (A/D) Line is a volume-based indicator designed to illuminate these hidden currents. It attempts to gauge the cumulative flow of money into or out of an asset. The fundamental hypothesis suggests that if the price closes near its high for the day, a significant portion of the day's trading volume was likely driven by buyers (accumulation). Conversely, if the price closes near its low, sellers likely dominated (distribution). By continuously summing these daily "money flows," the A/D line provides a cumulative measure of buying and selling pressure.

While the A/D line itself can indicate trends in money flow, its perceived power, many researchers hypothesize, lies in detecting divergences. A divergence occurs when price action and an indicator move in opposite directions. This contradiction in movement may signal a potential weakening of the current trend or an impending reversal, often attributed to a subtle shift in institutional behavior.

We will explore an algorithmic approach to an Accumulation/Distribution strategy focused on detecting such divergences. The aim is to investigate how these traditional concepts, often applied in discretionary trading, might be translated into quantifiable rules within a `backtrader` framework. Can programmatic methods reliably uncover these subtle shifts in market intent?

Looking to supercharge your algorithmic trading research? The Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python strategies, fully documented in comprehensive PDF manuals:

Ultimate Algorithmic Strategy Bundle

Divergence as a Potential Clue

The core of this strategy revolves around understanding and detecting divergences between price and the A/D Line:

1. **The Accumulation/Distribution Line (A/D Line):**

- CLV = ((Close — Low) — (High — Close)) / (High — Low)
- Calculation: The A/D Line is a running total of the Money Flow Volume for each period. Money Flow Volume is determined by the Close Location Value (CLV) multiplied by the bar's volume.
- If the Close is near the High, CLV is positive, suggesting buying pressure or accumulation.
- If the Close is near the Low, CLV is negative, suggesting selling pressure or distribution.
- If `High == Low` (zero spread), CLV is typically undefined or set to 0 to avoid division by zero.

- A/D Line = Previous A/D Line + (CLV \times Volume)
- Hypothesis: A rising A/D Line suggests ongoing accumulation (buying pressure), while a falling A/D Line suggests ongoing distribution (selling pressure).

2. **Divergence Signals:** This is where the A/D Line is often hypothesized to offer predictive insights. Divergences highlight discrepancies between price behavior and the underlying money flow.

- **Bullish Divergence:**
- Price Action: The price forms a lower low (a new trough below the previous one).
- A/D Line Action: The A/D Line, simultaneously, forms a higher low (a new trough above the previous one).
- Hypothesis: This contradictory movement suggests that despite falling prices, buying pressure (accumulation) is actually increasing or stabilizing. It may hint that the downtrend is weakening and a reversal to the upside might be imminent.
- **Bearish Divergence:**
- Price Action: The price forms a higher high (a new peak above the previous one).

- A/D Line Action: The A/D Line, simultaneously, forms a lower high (a new peak below the previous one).
- Hypothesis: This contradictory movement suggests that despite rising prices, selling pressure (distribution) is increasing or buying pressure is waning. It may hint that the uptrend is weakening and a reversal to the downside might be imminent.

3. **Swing Highs/Lows:** To effectively detect divergences, it is necessary to identify significant peaks and troughs (swing highs and lows) in both the price series and the A/D Line. A swing high is generally defined as a peak with lower highs on either side within a specific window, and a swing low is a trough with higher lows on either side.

4. **Trend Filter:** Integrating a simple trend filter (e.g., a Moving Average) with divergence signals may enhance their reliability. A bullish divergence might be considered more robust if the overall market trend is already bullish or appears to be transitioning to bullish.

The overarching strategy idea is to identify these hidden divergences between price action and money flow, potentially acting on them as early signals of trend exhaustion or reversal, possibly in alignment with the broader market trend.

Algorithmic Implementation: A `backtrader` Strategy

The following `backtrader` code provides a concrete implementation of these Accumulation/Distribution divergence concepts. Each snippet will be presented and analyzed to understand how the theoretical ideas are translated into executable code.

Step 1: Initial Setup and Data Loading

Any quantitative exploration begins with preparing the environment and fetching historical market data.

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams['figure.figsize'] = (10, 6)

# Download data and run backtest
data = yf.download('ETH-USD', '2021-01-01', '2024-01-01')
data.columns = data.columns.droplevel(1)
data_feed = bt.feeds.PandasData(dataname=data)
```

Analysis of this Snippet:

- Module Imports and Plotting Configuration: Essential Python libraries for numerical computation, data manipulation,

backtesting, and plotting are imported. `matplotlib.pyplot` is configured with `%matplotlib inline` for displaying plots directly within the output area.

- Data Acquisition: `yfinance.download('ETH-USD', ...)` fetches historical data for Ethereum. The subsequent `data.columns.droplevel(1)` call ensures column headers are in a single-level format (e.g., 'Close', 'Volume'), which `backtrader` expects.
- Data Feed Preparation: `bt.feeds.PandasData(dataname=data)` converts the prepared `pandas` DataFrame into a data feed for `backtrader`, enabling the backtesting engine to process it bar by bar.

Step 2: Defining the Accumulation/Distribution Strategy: `AccumulationDistributionStrategy` Initialization

This section outlines the `AccumulationDistributionStrategy` class, including its parameters, the calculation of the A/D line, and the initialization of internal state variables for tracking swing points.

```
class AccumulationDistributionStrategy(bt.Strategy):
    params = (
        ('swing_period', 7),      # Period to confirm swing highs/lo
        ('divergence_lookback', 30), # Bars to look back for diverg
        ('trend_period', 30),     # Trend filter period (for SMA)
        ('stop_loss_pct', 0.02), # 2% stop loss
```

```
    )

    def __init__(self):
        # Calculate Close Location Value (CLV)
        # CLV = ((Close - Low) - (High - Close)) / (High - Low)
        self.clv = ((self.data.close - self.data.low) - (self.data.
        # Calculate A/D Line: Cumulative sum of (CLV * Volume)
        # SumN is used to create a running sum, effectively the A/D
        self.ad_line = bt.indicators.SumN(self.clv * self.data.volu

        # Trend filter using a Simple Moving Average on close price
        self.trend_ma = bt.indicators.SMA(period=self.params.trend_

        # Lists to store detected swing points (bar index, value) f
        self.price_highs = []
        self.price_lows = []
        self.ad_highs = []
        self.ad_lows = []

        # backtrader's order tracking variables
        self.order = None
        self.stop_order = None
```

Analysis of the `__init__` Method:

- `params` : These parameters are adjustable controls for the strategy. `swing_period` defines the window used to identify swing highs and lows. `divergence_lookback` specifies how far back the strategy considers for price and A/D swings when searching for divergences. `trend_period` sets the lookback for the trend-confirming Simple Moving Average (SMA).
- A/D Line Calculation:

- `self.clv`: This calculates the Close Location Value (CLV) for each bar. CLV quantifies where the closing price is relative to the bar's total range, scaled between -1 and +1. A positive CLV suggests that buying pressure dominated the bar, while a negative value implies selling pressure.
- `self.ad_line`: This is the core Accumulation/Distribution Line. `bt.indicators.SumN` (configured with `period=1` to function as a running sum) accumulates the product of `clv` and `self.data.volume`. This cumulative sum attempts to represent the net money flow into or out of the asset over time.
- Trend Filter: `self.trend_ma` initializes an SMA on the closing price. This indicator will be used to filter trading signals, aiming to align trades with the broader market trend.
- Swing Point Storage: Lists (`price_highs`, `price_lows`, `ad_highs`, `ad_lows`) are initialized to store the identified swing points (both their bar index and value) for both the price series and the A/D line. These historical points are essential for the subsequent divergence detection logic.
- Order Tracking: Standard `backtrader` variables (`order`, `stop_order`) are set up to manage pending and active trade orders within the simulation.

Step 3: Identifying Swing Points and Detecting Divergences

This section contains helper methods crucial for the strategy: `is_swing_high` and `is_swing_low` identify significant peaks and troughs in a data series, which are then used by `detect_divergence` to find contradictory movements between price and the A/D line.

```
    def is_swing_high(self, data_series, period):
        """Check if current bar (or the bar at `period` offset from
        # Needs enough data to evaluate 'period' bars before and af
        if len(data_series) < period * 2 + 1:
            return False

        # The potential swing high is at the 'period' offset from t
        center_value = data_series[-period - 1] # -1 because next()

        # Check if this center_value is the highest within the 2*pe
        for i in range(period * 2 + 1):
            if i == period: # Skip the center bar itself
                continue
            # If any other bar in the window is higher or equal, it
            if data_series[-(period * 2 + 1 - i)] >= center_value:
                return False
        return True

    def is_swing_low(self, data_series, period):
        """Check if current bar (or the bar at `period` offset from
        # Needs enough data
        if len(data_series) < period * 2 + 1:
            return False

        # The potential swing low is at the 'period' offset from th
        center_value = data_series[-period - 1]

        # Check if this center_value is the lowest within the 2*per
```

```
            for i in range(period * 2 + 1):
                if i == period: # Skip the center bar itself
                    continue
                # If any other bar in the window is lower or equal, it'
                if data_series[-(period * 2 + 1 - i)] <= center_value:
                    return False
            return True

        def detect_divergence(self):
            """Detect bullish/bearish divergences between price and A/D
            # Need at least two recent swing highs/lows for both price
            if (len(self.price_highs) < 2 or len(self.price_lows) < 2 o
                len(self.ad_highs) < 2 or len(self.ad_lows) < 2):
                return None

            # Get the two most recent swing points for price and A/D li
            # Each swing point is a tuple: (bar_index, value)
            recent_price_highs = self.price_highs[-2:]
            recent_price_lows = self.price_lows[-2:]
            recent_ad_highs = self.ad_highs[-2:]
            recent_ad_lows = self.ad_lows[-2:]

            # Check for Bearish divergence: Price makes Higher High, A/
            # Compare the value of the most recent high with the previo
            if (len(recent_price_highs) >= 2 and len(recent_ad_highs) >
                if (recent_price_highs[-1][1] > recent_price_highs[-2][
                    recent_ad_highs[-1][1] < recent_ad_highs[-2][1]):
                    return 'bearish'

            # Check for Bullish divergence: Price makes Lower Low, A/D
            # Compare the value of the most recent low with the previou
            if (len(recent_price_lows) >= 2 and len(recent_ad_lows) >=
                if (recent_price_lows[-1][1] < recent_price_lows[-2][1]
                    recent_ad_lows[-1][1] > recent_ad_lows[-2][1]):
                    return 'bullish'

            return None # No divergence detected
```

Analysis of Swing Points and Divergence Detection:

- `is_swing_high(data_series, period)` / `is_swing_low(data_series, period)` : These functions are designed to algorithmically identify local peaks and troughs within a time series. A swing high is detected if a central bar's value is the highest within a window of `period` bars both before and after it. The logic for `is_swing_low` is analogous. The accuracy of these functions relies heavily on the chosen `period` and the precise indexing relative to the `data_series` provided.
- `detect_divergence()` : This function is central to the strategy. It retrieves the two most recent identified swing points for both the price and the A/D line.
- Bearish Divergence Logic: It looks for the simultaneous occurrence of price making a *higher high* (a new peak above the previous one) while the A/D line forms a *lower high* (a new peak below its previous one). This contradictory movement suggests underlying bearish sentiment.
- Bullish Divergence Logic: Conversely, it seeks a scenario where the price makes a *lower low* (a new trough below the previous one) while the A/D line simultaneously forms a *higher low* (a new trough above its previous one). This hints at underlying buying pressure despite falling prices.
- The effectiveness of this divergence detection directly depends on how accurately the `is_swing_high` and `is_swing_low` functions identify true, significant swing points.

Step 4: Implementing Risk Management

This section contains the `notify_order` method, a standard `backtrader` component responsible for managing order status updates and implementing a stop-loss mechanism.

```
    def notify_order(self, order):
        if order.status in [order.Completed]:
            if order.isbuy() and self.position.size > 0:
                stop_price = order.executed.price * (1 - self.param
                self.stop_order = self.sell(exectype=bt.Order.Stop,
            elif order.issell() and self.position.size < 0:
                stop_price = order.executed.price * (1 + self.param
                self.stop_order = self.buy(exectype=bt.Order.Stop,

        if order.status in [order.Completed, order.Canceled, order.
            self.order = None
            if order == self.stop_order:
                self.stop_order = None
```

Analysis of `notify_order()`:

- Order Status Management: This method is a callback function invoked by `backtrader` whenever the status of a trade order changes. It is vital for tracking pending orders and ensuring the strategy's order management logic remains coherent.
- Fixed Percentage Stop-Loss: Upon the successful completion of a trade (when a buy or sell order is `Completed`), a stop-loss order (`bt.Order.Stop`) is immediately placed. For a long

position, it's a sell stop order set `stop_loss_pct` below the entry price. For a short position, it's a buy stop order set `stop_loss_pct` above the entry price. This is a fundamental risk management technique aimed at limiting potential losses on any single trade to a predefined percentage.

- Order Cleanup: Once a trade order or a stop-loss order reaches a final state (either `Completed`, `Canceled`, or `Rejected`), the corresponding `self.order` or `self.stop_order` variable is reset to `None`. This signals that the strategy is ready to place new primary orders without confusion.

Step 5: The Trading Logic: Orchestrating Divergence Signals

This is the main loop that executes for each new bar of data, updating swing points, detecting divergences, and making trading decisions based on these signals, often filtered by the broader trend.

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

        # Skip if not enough data for swing detection
        # (swing_period * 2 + 1 bars are needed for the swing logic
        if len(self.data) < self.params.swing_period * 2 + 1:
            return

        # Create historical arrays for price and A/D line within th
```

```
        # The slice is reversed for how is_swing_high/low functions
        price_array = [self.data.close[-i] for i in range(min(len(s
        ad_array = [self.ad_line[-i] for i in range(min(len(self.da

        # Detect and store new swing highs for both price and A/D l
        # A swing high is identified 'swing_period' bars ago relati
        if self.is_swing_high(price_array, self.params.swing_period
            # Calculate the actual bar index and values for the det
            swing_bar_idx = len(self.data) - self.params.swing_peri
            swing_price_val = self.data.close[-self.params.swing_pe
            swing_ad_val = self.ad_line[-self.params.swing_period]

            self.price_highs.append((swing_bar_idx, swing_price_val
            self.ad_highs.append((swing_bar_idx, swing_ad_val))

            # Keep only the most recent 5 swing highs for efficienc
            if len(self.price_highs) > 5:
                self.price_highs = self.price_highs[-5:]
                self.ad_highs = self.ad_highs[-5:]

        # Detect and store new swing lows for both price and A/D li
        if self.is_swing_low(price_array, self.params.swing_period)
            # Calculate the actual bar index and values for the det
            swing_bar_idx = len(self.data) - self.params.swing_peri
            swing_price_val = self.data.close[-self.params.swing_pe
            swing_ad_val = self.ad_line[-self.params.swing_period]

            self.price_lows.append((swing_bar_idx, swing_price_val)
            self.ad_lows.append((swing_bar_idx, swing_ad_val))

            # Keep only the most recent 5 swing lows
            if len(self.price_lows) > 5:
                self.price_lows = self.price_lows[-5:]
                self.ad_lows = self.ad_lows[-5:]

        # Attempt to detect a divergence using the stored swing poi
        divergence = self.detect_divergence()

        if divergence is None:
            return # No divergence detected, skip trading

        # --- Trading signals based on detected divergences and tre

        if divergence == 'bullish':
```

```
                # Hypothesis: Bullish divergence + confirming uptrend
                # If price is above its trend MA, it aligns with underl
                if self.data.close[0] > self.trend_ma[0]:
                    if self.position.size < 0: # If currently short, cl
                        if self.stop_order is not None: self.cancel(sel
                        self.order = self.close()
                    elif not self.position: # If no position, open long
                        self.order = self.buy()

            elif divergence == 'bearish':
                # Hypothesis: Bearish divergence + confirming downtrend
                # If price is below its trend MA, it aligns with underl
                if self.data.close[0] < self.trend_ma[0]:
                    if self.position.size > 0: # If currently long, clo
                        if self.stop_order is not None: self.cancel(sel
                        self.order = self.close()
                    elif not self.position: # If no position, open shor
                        self.order = self.sell()
```

Analysis of `next()` (The Trade Orchestrator):

- Initialization Check: `if len(self.data) < self.params.swing_period * 2 + 1: return` ensures that enough historical data has accumulated for the `is_swing_high` and `is_swing_low` logic to function correctly.
- Historical Data Preparation: `price_array` and `ad_array` are created as slices of recent price and A/D line data. These arrays serve as the necessary input for the swing point detection functions.
- Swing Point Detection & Storage: The `is_swing_high()` and `is_swing_low()` functions are called for both price and the

A/D line. If new swing points are identified, their bar index and value are appended to the respective historical lists (`price_highs`, `price_lows`, etc.). These lists are kept to a limited size (5 recent swings) to maintain efficiency and focus on recent market structure.

- Divergence Detection: `divergence = self.detect_divergence()` attempts to identify a bullish or bearish divergence based on the most recently stored swing points.
- Trading Decisions: If a `divergence` is detected (either 'bullish' or 'bearish'), the strategy then filters this signal with the overall `trend_ma`.
- For a `bullish` divergence, the strategy seeks `uptrend` confirmation (`self.data.close[0] > self.trend_ma[0]`) before considering a long entry or closing an existing short position. This aims to align the short-term divergence signal with the broader market direction.
- For a `bearish` divergence, the strategy seeks `downtrend` confirmation (`self.data.close[0] < self.trend_ma[0]`) before considering a short entry or closing an existing long position.
- The use of `self.order` to block new main orders and `self.cancel(self.stop_order)` to manage associated stop-loss orders are standard `backtrader` practices for robust trade management.

Step 6: Executing the Accumulation/Distribution Experiment: The Backtest Execution

This section sets up `backtrader`'s core engine, adds the strategy and data, configures the broker, and executes the simulation.

```
cerebro = bt.Cerebro()
cerebro.addstrategy(AccumulationDistributionStrategy)
cerebro.adddata(data_feed)
cerebro.addsizer(bt.sizers.PercentSizer, percents=95)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)

print(f'Start: ${cerebro.broker.getvalue():,.2f}')
results = cerebro.run()
print(f'End: ${cerebro.broker.getvalue():,.2f}')
print(f'Return: {((cerebro.broker.getvalue() / 100000) - 1) * 100:.

# Fix matplotlib plotting issues
plt.rcParams['figure.max_open_warning'] = 0
plt.rcParams['agg.path.chunksize'] = 10000

try:
    cerebro.plot(iplot=False, style='candlestick', volume=False)
    plt.show()
except Exception as e:
    print(f"Plotting error: {e}")
    print("Strategy completed successfully - plotting skipped")
```

Analysis of the Backtest Execution:

- `cerebro = bt.Cerebro()` : This line initializes the central `backtrader` engine responsible for running the simulation.
- Strategy and Data Loading: `cerebro.addstrategy(...)` registers the defined Accumulation/Distribution strategy, and `cerebro.adddata(...)` feeds it the historical data for the simulation.
- Broker Configuration: `cerebro.addsizer(...)` , `cerebro.broker.setcash(...)` , and `cerebro.broker.setcommission(...)` configure the initial capital, the percentage of capital to use per trade, and simulate trading commissions, all contributing to a more realistic backtest environment.

- `cerebro.run()` : This command initiates the entire backtest simulation, allowing the strategy to execute its logic sequentially through the historical data bars. The `results` variable stores the outcome of the backtest.
- Basic Performance Output: The `print` statements provide a straightforward summary of the simulation, displaying the starting and ending portfolio values, along with the overall percentage return achieved by the strategy over the backtest period.
- Plotting Configuration: `plt.rcParams` lines configure `matplotlib` for plotting, potentially preventing warnings with large datasets. The `cerebro.plot()` call generates a visual representation of the backtest. It is configured to use a `candlestick` style. The `volume=False` setting for the plot is an interesting detail: while the A/D line is fundamentally a volume-derived indicator, the plot itself will not show the raw volume bars. This might limit some visual analysis of the raw volume behavior that the A/D line is derived from.

Conclusion

This `backtrader` strategy represents an intriguing attempt to automate the detection of Accumulation/Distribution divergences—a common technique in discretionary technical analysis. It highlights the potential for indicators that go beyond

simple price action to provide insights into underlying money flow and potential shifts in trend.

- Subjectivity in Swing Detection: The `is_swing_high` and `is_swing_low` functions are crucial to this strategy. However, the definition of a "significant" swing high or low can often be subjective in traditional analysis. The choice of `swing_period` is therefore critical and might heavily influence the frequency and perceived accuracy of detected divergences.
- Divergence Reliability: While divergences are powerful conceptually, their predictive power in practice can be inconsistent, and false signals are a known challenge. The effectiveness of this specific algorithmic definition of divergence in real-world scenarios warrants rigorous testing and validation.
- Lag vs. Lead: Divergences are often touted as leading indicators, suggesting future price movements. However, the time required for swing detection and divergence confirmation might introduce some inherent lag, potentially diminishing their “leading” quality.
- Trend Confirmation: The strategy uses a simple SMA as a trend filter. Exploring more advanced trend filters or market regime detection methods could potentially improve the strategy’s ability to filter out less reliable divergences.

This strategy serves as a compelling starting point for further research into momentum and money flow dynamics. The journey of translating these often-subjective concepts into precise, testable algorithms is a continuous and fascinating challenge in quantitative trading.

Python | Algorithmic Trading | Quantitative Finance | Crypto | Backtesting

Written by PyQuantLab

655 followers · 6 following

Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

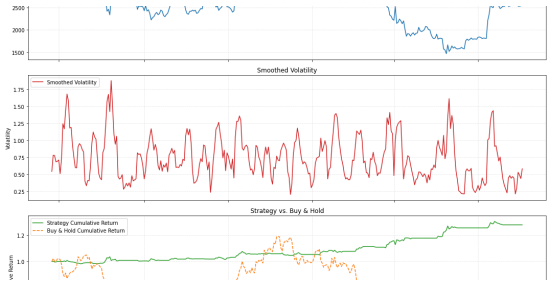
Following

No responses yet

Steven Feng CAI

What are your thoughts?

More from PyQuantLab



PyQuantLab

Volatility Clustering Trading Strategy with Python

Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python...

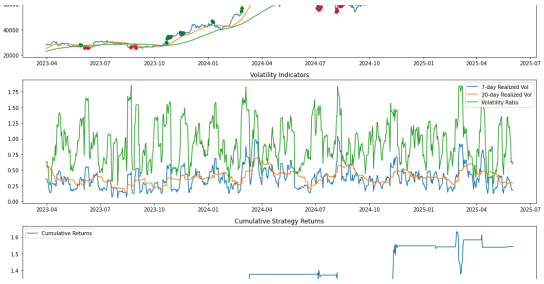
Jun 3 32 3

PyQuantLab

An Algorithmic Exploration of Volume Spread Analysis...

📢 Note: You can read all articles for free on our website: pyquantlab.com

Jun 9 54 1



PyQuantLab

Trend-Volatility Confluence Trading Strategy

The Ultimate Algorithmic Strategy Bundle has you covered with over 80...

Jun 3 60

PyQuantLab

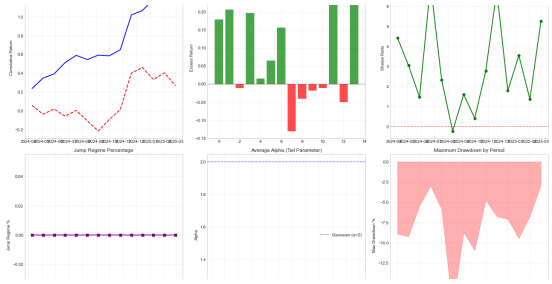
Building an Adaptive Trading Strategy with Backtrader: A...

Note: You can read all articles for free on our website: pyquantlab.com

Jun 4 64

See all from PyQuantLab

Recommended from Medium



PyQuantLab

Capturing Market Momentum with Levy Flights: A Python...

Financial markets are complex systems, often exhibiting behaviors...

Jun 5 102

MarketMuse

“I Let an AI Bot Trade for Me for 7 Days—It Made $8,000...

Subtitle: While you’re analyzing candlestick patterns, AI bots are fron...

Jun 3 75 3

In InsiderFinance Wire by Nayab Bhutta

Don’t Trade Without This! Why I Switched from...

The Indicator Addiction (That Almost Ruined My Trading)

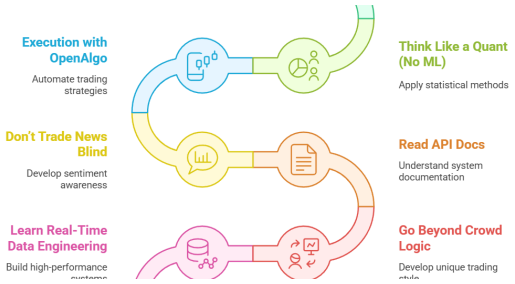
Jun 12 152 3

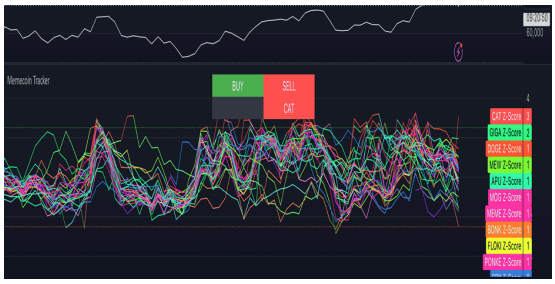
In DataDrivenInvestor by Mr. Q

Why You Should Ignore Most Backtested Trading...

To be more precise, while the title is limited in length, what I truly mean ar...

Jun 18 133 2





Rajandran R (Creator - OpenAlgo)

Algorithmic Trading Roadmap 2025: From Curio...

The dream of algorithmic trading is simple: let code take over the...

Jun 22 23 3

Yong kang Chia

An Engineer's Guide to Building and Validating...

From data collection to statistical validation—a rigorous framework for...

Jun 15 9

See more recommendations