Member-only story

Capturing Market Momentum with Levy Flights: A Python Implementation

PyQuantLab Following 14 min read · Jun 5, 2025

Financial markets are complex systems, often exhibiting behaviors that traditional models based on normal distributions struggle to capture. One such characteristic is the presence of “heavy tails” in return distributions and sudden, large movements or “jumps.” Levy flights, and the associated Levy stable distributions, provide a mathematical framework to model these phenomena. This article explores a Python-based trading strategy that attempts to detect and leverage market momentum by analyzing Levy flight characteristics, identifying jump and

diffusion regimes, and applying this understanding to generate trading signals.

https://www.pyquantlab.com/bundles/Ultimate%20Algo%20Trading%20Bundle.html

The **Ultimate Algo Trading Value Pack** is the definitive all-in-one toolkit for quantitative traders and developers. Inside, you'll find six comprehensive PDF eBooks covering everything from technical analysis to advanced machine learning, five fully coded Python strategy packs totaling over 80 unique trading systems, and the Backtester App — complete with its own Strategy Code Manual PDF — for rapid development and backtesting.

Understanding Levy Flights in Finance 🦎

In contrast to Brownian motion (which underlies models assuming normal distributions and constant volatility), Levy flights are a type of random walk where the step lengths are not constant but are drawn from a probability distribution with heavy tails. This means that while most price changes (steps) might be small, there's a non-negligible probability of very large price changes occurring.

Key Characteristics of Levy Stable Distributions:

Levy stable distributions are uniquely characterized by their characteristic function:

$$\phi(t) = \mathbb{E}\left[e^{itX}\right] = \exp\Big(i\,\delta\,t \;-\; \gamma\,|t|^{\alpha}\,\Big[1 + i\,\beta\,\operatorname{sign}(t)\,\omega(\alpha,t)\Big]\Big)\,.$$

where:

- α is the stability parameter (tail index)
- β is the skewness parameter
- γ is the scale parameter
- δ is the location parameter

and ω(α,t) is defined as:

$$\omega(\alpha,t) = \begin{cases} \tan\left(\frac{\pi\alpha}{2}\right) & \text{if } \alpha \neq 1 \\ \frac{2}{\pi}\log|t| & \text{if } \alpha = 1 \end{cases}$$

- Stability Parameter (α): This parameter, typically $1 < \alpha \leq 2$, governs the “heaviness” of the tails of the distribution.
- If $\alpha = 2$, the distribution is Gaussian (normal).
- If $\alpha < 2$, the distribution has heavier tails than a Gaussian, indicating a higher probability of extreme events (jumps). The smaller the α, the heavier the tails.

- Skewness Parameter (β): This parameter, $-1 \leq \beta \leq 1$, determines the skewness or asymmetry of the distribution.
- $\beta = 0$ implies a symmetric distribution.
- $\beta > 0$ implies a skew towards positive returns.
- $\beta < 0$ implies a skew towards negative returns.
- Scale Parameter (γ): Analogous to the standard deviation in a normal distribution, it measures the width or dispersion of the distribution.
- Location Parameter (δ): Represents the mean or center of the distribution.

Why are they relevant? Financial asset returns often display:

1. Leptokurtosis (Fat Tails): More extreme returns (both positive and negative) than predicted by a normal distribution.
2. Jumps: Sudden, large price movements that are hard to explain with continuous diffusion models.

By estimating the parameters of a Levy stable distribution from historical returns, we can gain insights into the underlying market dynamics, such as the propensity for jumps ($\alpha < 2$) and any inherent biases ($\beta \neq 0$). This strategy uses these insights to differentiate between jump regimes (dominated by large, sudden

moves) and diffusion regimes (more random, smaller price changes), and then calculates momentum differently for each.

Python Implementation: A Walkthrough

The provided Python script implements a sophisticated Levy flight momentum strategy, structured into several classes:

1. `LevyFlightConfig` : Manages all parameters for the analysis, strategy, and backtest.
2. `LevyFlightAnalyzer` : Handles the core analysis – estimating Levy parameters, detecting jumps, identifying market regimes, and calculating Levy-based momentum.
3. `LevyMomentumStrategy` : Uses the analyzer's output to generate trading signals and execute trades with risk management.
4. `LevyWalkForwardBacktest` : Implements a walk-forward methodology to test the strategy over time.

Let’s delve into each.

1. Configuration: `LevyFlightConfig`

This class centralizes all the tunable parameters of the strategy. This is excellent practice for managing complexity and facilitating experimentation.

```
class LevyFlightConfig:
    """Configuration class for Levy Flight Momentum Detection"""
    def __init__(self):
        # Levy Flight Parameters
        self.levy_window = 50  # Window for Levy parameter estimati
        self.alpha_bounds = (1.1, 2.0)  # Stability parameter bound
        self.beta_bounds = (-1.0, 1.0)  # Skewness parameter bounds
        self.jump_threshold = 2.0  # Standard deviations for jump d
        self.diffusion_threshold = 0.5  # Threshold for diffusive v

        # Momentum Detection Parameters
        self.momentum_lookback = 20  # Lookback for momentum calcul
        self.jump_momentum_weight = 2.0  # Weight for jump-driven m
        self.diffusion_momentum_weight = 1.0  # Weight for diffusio
        self.regime_persistence = 0.8  # Regime switching persisten

        # Signal Processing
        self.jump_signal_decay = 0.9  # Exponential decay for jump
        self.trend_ema_span = 12  # EMA span for trend extraction
        self.volatility_window = 20  # Window for volatility estima
        self.signal_threshold = 0.02  # Trading signal threshold

        # Risk Management
        self.position_sizing = 1.0
        self.stop_loss = 0.04  # 4% stop loss
        self.take_profit = 0.08  # 8% take profit
        self.max_position_hold = 10  # Max days to hold position

        # Backtest Parameters
        self.train_period = 90
        self.test_period = 30
        self.transaction_cost = 0.0015
        self.initial_capital = 10000

    def display_parameters(self):
        # ... (displays parameters neatly) ...
```

Key Parameters Explained:

- Levy Flight Settings:
- `levy_window`: The number of past return observations used to estimate the Levy distribution parameters.
- `alpha_bounds`, `beta_bounds`: Constraints for the stability and skewness parameters during estimation. α is typically between 1 (Cauchy-like, very heavy tails) and 2 (Gaussian).
- `jump_threshold`: How many standard deviations (or scaled by `gamma`) a return must be to be considered a potential jump.
- `diffusion_threshold`: A threshold on jump intensity to switch between "jump" and "diffusion" regimes.
- Momentum Detection:
- `momentum_lookback`: Window for calculating short-term momentum.
- `jump_momentum_weight`, `diffusion_momentum_weight`: Different weights applied to momentum calculated from jumps versus normal diffusion, allowing the strategy to react differently based on the nature of price moves.
- Signal Processing & Risk Management: These include parameters for smoothing signals, setting trade entry thresholds, and defining exit rules (stop-loss, take-profit, max holding period).

2. Core Analysis: `LevyFlightAnalyzer` 🔬

This class is the heart of the Levy flight detection mechanism.

2.1. Estimating Levy Parameters (`estimate_levy_parameters`)

This method attempts to fit a Levy stable distribution to a window of returns.

```
    def estimate_levy_parameters(self, returns):
        """Estimate Levy stable distribution parameters using MLE""
        try:
            # ... (input validation and data cleaning) ...

            # Initial parameter guesses based on moments (kurtosis,
            alpha_init = np.clip(2.0 - np.abs(stats.kurtosis(return
            # ... (beta_init, gamma_init, delta_init) ...

            def levy_loglike(params):
                alpha, beta, gamma, delta = params
                # ... (bounds check) ...
                try:
                    # Approximate log-likelihood
                    centered_returns = (returns_clean - delta) / ga
                    if alpha == 2: # Gaussian case
                        loglike = -0.5 * np.sum(centered_returns**2
                    else: # Approximation for general Levy case
                        loglike = -np.sum(np.abs(centered_returns)*
                    return -loglike
                except: return 1e10

            result = minimize(
                levy_loglike,
                [alpha_init, beta_init, gamma_init, delta_init],
                bounds=[(1.1, 1.99), (-0.99, 0.99), (1e-6, None), (
                method='L-BFGS-B'
            )
            # ... (return results or initial guesses on failure) ..
        except Exception as e:
```

```
            # Fallback to simpler moment-based estimates if optimiz
            # ... (fallback calculations) ...
```

- Challenges: Estimating all four parameters of a Levy stable distribution via Maximum Likelihood Estimation (MLE) is notoriously difficult because the probability density function (PDF) generally doesn't have a closed-form expression (except for special cases like Gaussian, Cauchy, and Levy-Smirnov).
- Implementation Approach:
- The code uses an approximation for the log-likelihood function. For $\alpha = 2$, it correctly uses the Gaussian log-likelihood. For other values, it uses a simplified form involving `abs(centered_returns)**alpha` . This is a practical heuristic but not the exact Levy log-likelihood (which often involves numerical integration of the characteristic function).
- It provides initial parameter guesses based on sample moments (kurtosis for α, skewness for β, standard deviation for γ, mean for δ). This helps the optimizer.
- `scipy.optimize.minimize` with the `L-BFGS-B` method is used to find parameters that maximize the (approximate) log-likelihood.
- Robust error handling and fallbacks to moment-based estimates are included if the optimization fails, which is a

good design choice.

2.2. Detecting Jumps (`detect_jumps`)

This method identifies returns that are likely “jumps” rather than part of normal diffusive price movements, given the estimated Levy parameters.

```
    def detect_jumps(self, returns, alpha, gamma):
        """Detect jump events in the time series"""
        # ... (initialization) ...
        std_returns = returns / (gamma + 1e-10) # Standardize by Le
        threshold = self.config.jump_threshold

        for i in range(len(returns)):
            if np.abs(std_returns[i]) > threshold:
                # Check if it's a true jump or just high volatility
                local_vol = np.std(returns[max(0, i-5):i+6])
                if np.abs(returns[i]) > threshold * local_vol: # Co
                    jumps[i] = 1
                    jump_magnitudes[i] = returns[i]
        return jumps, jump_magnitudes
```

- Returns are first standardized using the estimated scale parameter `gamma` .
- A return is flagged as a potential jump if its standardized value exceeds `config.jump_threshold` .

- An additional check compares the raw return to `threshold * local_vol` (standard deviation in a small rolling window) to further distinguish jumps from periods of generally high (but perhaps not Levy-jump-like) volatility.

2.3. Identifying Market Regime (`identify_regime`)

This method classifies the recent market behavior as either "jump-dominated" or "diffusion-dominated."

```
    def identify_regime(self, returns, jumps):
        # ... (input validation) ...
        jump_intensity = np.sum(jumps[-10:]) / 10 # Proportion of j

        if jump_intensity > self.config.diffusion_threshold:
            new_regime = 'jump'
        else:
            new_regime = 'diffusion'

        # Apply persistence to avoid rapid switching
        if hasattr(self, 'regime_state') and self.regime_state != n
            if np.random.random() > (1 - self.config.regime_persist
                new_regime = self.regime_state # Keep old regime wi

        self.regime_state = new_regime
        return new_regime
```

- It calculates `jump_intensity` as the proportion of detected jumps in a recent window (last 10 periods).

- If this intensity exceeds `config.diffusion_threshold`, the regime is classified as 'jump'.
- A persistence mechanism is included: the regime only switches from its previous state if a random draw overcomes the `config.regime_persistence` factor. This prevents overly frequent regime changes.

2.4. Calculating Levy Momentum (`calculate_levy_momentum`)

This is where distinct momentum measures for jumps and diffusion are computed.

```
    def calculate_levy_momentum(self, returns, jumps, jump_magnitud
        # ... (input validation and slicing) ...

        # Jump-driven momentum: exponentially weighted average of r
        jump_momentum = 0.0
        if np.sum(recent_jumps) > 0:
            jump_returns = recent_jump_mags[recent_jumps == 1]
            if len(jump_returns) > 0:
                weights = np.array([self.config.jump_signal_decay**
                weights = weights[::-1] # More recent jumps get hig
                jump_momentum = np.average(jump_returns, weights=we

        # Diffusion momentum: simple mean of recent non-jump return
        diffusion_returns = recent_returns[recent_jumps == 0]
        diffusion_momentum = 0.0
        if len(diffusion_returns) > 0:
            diffusion_momentum = np.mean(diffusion_returns)
```

```
        return jump_momentum, diffusion_momentum
```

- Jump Momentum: Calculated as an exponentially weighted average of the magnitudes of detected jumps within the `momentum_lookback` window. Recent jumps are given higher weight due to `jump_signal_decay`.
- Diffusion Momentum: Calculated as the simple average of returns that were *not* classified as jumps within the lookback window.

3. Strategy Logic: `LevyMomentumStrategy` 🧭

This class brings together the analysis from `LevyFlightAnalyzer` to make trading decisions.

3.1. Generating Signals (`generate_signals`)

This method performs a rolling analysis to generate a raw signal series.

```
    def generate_signals(self, prices):
        # ... (initialization, pct_change) ...
        signals = np.zeros(len(prices))
        # ... (levy_data dictionary for storing analysis outputs) ...
```

```
        for i in range(self.config.levy_window, len(returns)): # Ro
            window_returns = returns.iloc[i-self.config.levy_window

            alpha, beta, gamma, delta = self.analyzer.estimate_levy
            jumps, jump_magnitudes = self.analyzer.detect_jumps(win
            regime = self.analyzer.identify_regime(window_returns,
            jump_momentum, diffusion_momentum = self.analyzer.calcu
                window_returns, jumps, jump_magnitudes, regime
            )

            # Combine signals based on regime
            if regime == 'jump':
                signal = (jump_momentum * self.config.jump_momentum
                          diffusion_momentum * self.config.diffusio
            else: # diffusion regime
                signal = (diffusion_momentum * self.config.diffusio
                          jump_momentum * self.config.jump_momentum

            # Apply trend filter (boost signal if aligned with EMA
            trend_prices = prices.iloc[max(0, i-self.config.trend_e
            if len(trend_prices) > 5:
                trend_direction = 1 if trend_prices.iloc[-1] > tren
                signal *= (1 + 0.3 * trend_direction)

            # Normalize signal by volatility using tanh (to bound b
            volatility = np.std(window_returns[-self.config.volatil
            signal = np.tanh(signal / (volatility + 1e-6))

            signals[i] = signal # Store signal for the current poin
                                # Note: returns array is 1 shorter
                                # This signal corresponds to prices

        # Adjust signals index to align with prices
        # The loop goes up to len(returns)-1. `returns` starts from
        # So, the last signal signals[len(returns)-1] is for return
        # The signals array should be shifted to align with the pri
        # Current code creates pd.Series(signals, index=prices.inde
        # but the loop for calculation only fills up to len(returns
        # A more robust way to align:
        final_signals = pd.Series(0.0, index=prices.index)
        if len(returns) > 0: # Ensure returns is not empty
             final_signals.iloc[self.config.levy_window+1 : len(ret

        # Store analysis data
```

```
            if len(levy_data['alpha']) > 0:
                analysis_index_start = self.config.levy_window + 1 # +1
                analysis_index_end = analysis_index_start + len(levy_da
                self.levy_analysis = pd.DataFrame(levy_data, index=pric

            return final_signals # Return pd.Series aligned with price
```

- A rolling window approach is used: for each step `i`, the last `levy_window` returns are analyzed.
- The combined signal gives different importance to jump and diffusion momentum based on the identified `regime`. In a 'jump' regime, jump momentum is more influential. In 'diffusion', its influence is halved.
- A trend filter is applied: the signal is boosted if it aligns with the direction of a short-term trend (defined by price relative to its EMA).
- The final signal is normalized by recent volatility and passed through `np.tanh` to bound it between -1 and 1. This makes the signal strength adaptive to volatility.
- The indexing for `signals` and `levy_analysis` needs care to ensure correct alignment with the original `prices` index. The corrected `final_signals` part aims to address this.

3.2. Executing Trades (`execute_trades`)

This method simulates trade execution based on the generated signals, incorporating risk management rules.

```
    def execute_trades(self, prices, signals, verbose=False):
        # ... (initialization of capital, positions list, trades li

        for i, (date, price) in enumerate(prices.items()):
            # ... (skip first day, calculate daily_return holder, u

            # Risk management checks (Stop Loss, Take Profit, Max H
            if self.position != 0 and self.entry_price > 0:
                pnl_pct = (price - self.entry_price) / self.entry_p
                if self.config.stop_loss and pnl_pct < -self.config
                    # ... (execute stop loss, update capital with t
                elif self.config.take_profit and pnl_pct > self.con
                    # ... (execute take profit) ...
                elif self.position_days >= self.config.max_position
                    # ... (execute max hold exit) ...

            # Trading signals
            if i < len(signals):
                signal_value = signals.iloc[i] # Use signal for cur

                if signal_value > self.config.signal_threshold and
                    # ... (handle existing short, enter long, recor
                elif signal_value < -self.config.signal_threshold a
                    # ... (handle existing long, enter short, recor

            # Calculate daily P&L based on holding position
            # ... (calculate P&L if self.position is 1 or -1) ...
            daily_returns.append(daily_return_for_day) # Store actu
            # Capital update should be based on actual P&L, not jus
            # The current script's capital update: capital *= (1 +

        return trades, daily_returns_series, capital # daily_return
```

- It iterates through prices, applying risk management rules (stop-loss, take-profit, maximum holding period) *before* checking for new trade signals.
- New trades (long or short) are entered if the signal exceeds `config.signal_threshold` and no conflicting position is open.
- Transaction costs are applied when trades are opened/closed.
- Daily portfolio returns are calculated based on the active position.

4. Backtesting Framework: `LevyWalkForwardBacktest` 📈

This class orchestrates the walk-forward backtesting process.

```
class LevyWalkForwardBacktest:
    # ... (__init__, get_data) ...

    def run_backtest(self, start_date=None, end_date=None):
        # ... (date handling) ...
        while current_date < end_date - timedelta(days=self.config.
            # Define train and test periods
            train_start = current_date
            train_end = current_date + timedelta(days=self.config.t
            test_start = train_end
            test_end = train_end + timedelta(days=self.config.test_
            # ... (break if test_end > end_date) ...

            train_data = self.get_data(train_start, train_end) # Pr
            test_data = self.get_data(test_start, test_end)   # Pri
            # ... (data validation) ...

            result = self.run_strategy_period(train_data, test_data
            # ... (store result, increment current_date) ...
```

```
        return self.compile_results()

    def run_strategy_period(self, train_data, test_data, iteration)
        try:
            strategy = LevyMomentumStrategy(self.config)

            # Generate signals: Use part of train_data for history/
            # This ensures the signal generation has enough histori
            combined_data = pd.concat([train_data.tail(self.config.
            signals_on_combined = strategy.generate_signals(combine
            # Extract signals relevant only to the test_data period
            test_signals = signals_on_combined.tail(len(test_data))

            trades, daily_returns, final_capital = strategy.execute
            # ... (calculate metrics for this period and return) ..
        # ... (error handling) ...

    def compile_results(self):
        # ... (aggregates results from all walk-forward periods) ..

    def plot_results(self):
        # ... (plots various performance charts based on compiled r
```

- Walk-Forward Logic: The `run_backtest` method iterates, creating rolling training and testing periods.
- Data Handling for Signals: In `run_strategy_period`, `combined_data = pd.concat([train_data.tail(self.config.levy_window), test_data])` is crucial. It takes the last `levy_window` portion of the training data and appends the test data. Signals are then generated on this `combined_data`. This allows the `generate_signals` method (which itself uses a rolling window of `levy_window`) to have sufficient historical context when

generating signals for the very beginning of the `test_data`. The `test_signals` are then sliced out for actual trading simulation on the `test_data`. This is a sound approach to avoid lookahead bias while ensuring signal stability.

- Results Compilation and Plotting: The class aggregates results from each test period and provides methods for summarizing and visualizing performance.

Supporting Functions and Execution 🛠️

The script also includes:

- `test_levy_configurations()`: A function to quickly test different sets of Levy parameters on a short period. This is useful for sensitivity analysis.
- `analyze_levy_characteristics()`: A standalone function to download data for a symbol, estimate its overall Levy parameters, analyze jump frequency, and plot diagnostic charts (price, returns with jumps, return distribution vs. normal, Q-Q plot). This helps in understanding the baseline characteristics of an asset before applying the strategy.

Main Execution Block (`if __name__ == "__main__":`)

1. Initializes `LevyFlightConfig`.

2. Allows for customization of parameters (commented out in the provided code, but available for users).
3. Optionally runs `analyze_levy_characteristics` for the target symbol (BTC-USD) and can even dynamically adjust some configuration parameters based on the analysis (e.g., if heavy tails are detected).
4. Initializes and runs the `LevyWalkForwardBacktest`.
5. Prints a detailed summary of performance metrics and generates plots.
6. Includes robust error handling and troubleshooting tips.

How It All Works Together: Strategy Flow

1. Configuration: Define all parameters (Levy estimation, momentum, risk, backtest).
2. Data Acquisition: Fetch historical price data using `yfinance`.
3. Walk-Forward Loop:

- For each period:
- Training Data Slice: Used implicitly by `generate_signals` to establish historical context.
- Testing Data Slice: The period on which trading is simulated.
- Signal Generation (`LevyMomentumStrategy.generate_signals` on combined train-tail + test data):
- Calculate rolling returns.
- For each point in the (combined) returns window:
- Estimate Levy parameters () from the preceding `levy_window` of returns using `LevyFlightAnalyzer.estimate_levy_parameters`.
- Detect jumps using `LevyFlightAnalyzer.detect_jumps`.
- Identify market regime (jump/diffusion) using `LevyFlightAnalyzer.identify_regime`.
- Calculate jump momentum and diffusion momentum using `LevyFlightAnalyzer.calculate_levy_momentum`.

- Combine these momentums based on the current regime and configured weights.
- Apply a trend filter (EMA-based).
- Normalize the signal using volatility and `tanh`.
- Trade Execution (`LevyMomentumStrategy.execute_trades` on test data with its signals):
- Apply stop-loss, take-profit, and max holding period rules.
- Enter long/short positions if the signal strength exceeds `signal_threshold`.
- Calculate daily portfolio returns and track capital.

4. Results Aggregation & Reporting: Compile metrics (total return, Sharpe, drawdown, win rate, etc.) across all walk-forward periods and plot the performance.

Conclusion & Potential Enhancements 💡

This Python script presents a sophisticated and well-structured approach to developing a trading strategy based on Levy flight theory. It correctly identifies that financial markets often deviate from Gaussian assumptions and attempts to model and exploit the characteristics of jumps and heavy tails.

Key Strengths:

- Advanced Concept: Tackles non-Gaussian properties of financial returns.
- Regime Adaptability: Differentiates between jump and diffusion regimes and adapts momentum calculation.
- Comprehensive Framework: Includes configuration, analysis, strategy, backtesting, risk management, and visualization.
- Robustness: Includes error handling, parameter fallbacks, and a walk-forward testing approach.
- Parameterization: Highly configurable, allowing for extensive experimentation.

Potential Areas for Exploration/Refinement:

- Levy Parameter Estimation: The log-likelihood approximation is a heuristic. For more precision, one might explore methods based on numerical integration of the characteristic function or specialized libraries for Levy distribution fitting, though these are computationally more intensive.
- Jump Detection: The current jump detection is a multi-step heuristic. Alternative jump detection models (e.g., based on bipower variation) could be explored.
- Signal Combination and Normalization: The way jump/diffusion momentums are combined and signals are

trend-filtered and normalized offers many avenues for tuning and alternative formulations.

- Computational Cost: The rolling estimation of Levy parameters can be computationally intensive. For very frequent re-estimation or very long backtests, performance profiling might be needed.
- Parameter Optimization: The large number of parameters suggests that a systematic parameter optimization process (e.g., grid search, Bayesian optimization, genetic algorithms) over a validation set could be beneficial, though computationally expensive.

Overall, this script provides an excellent foundation for exploring advanced quantitative trading strategies that go beyond traditional mean-variance frameworks by incorporating the unique statistical properties often observed in financial markets.

Algorithmic Trading | Quantitative Finance | Python | Trading Strategy | Momentum Strategy

Written by PyQuantLab

655 followers · 6 following

Following

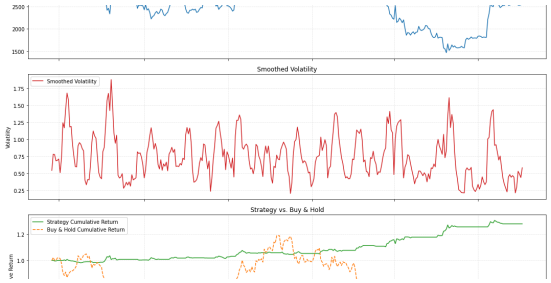
Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

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...

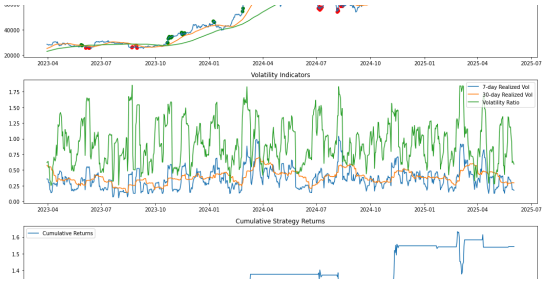
Jun 3 32 3

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

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...

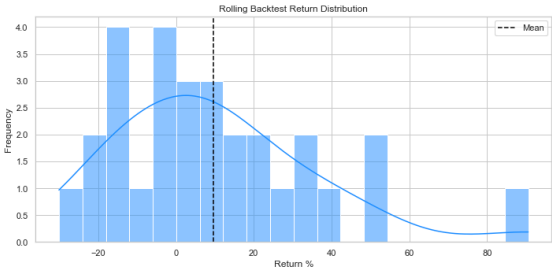
Jun 3 75 3

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



PyQuantLab

Evaluating Adaptive Kalman Filter Strategy Consistency...

In DataDrivenInvestor by Mr. Q

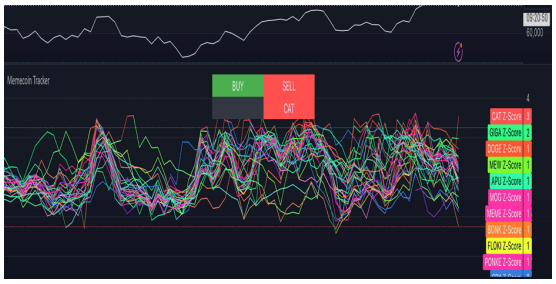
Why You Should Ignore Most Backtested Trading...

A common method for testing a trading strategy is to run a single...

Jun 19 8

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

Jun 18 133 2



Yong kang Chia

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

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

Jun 15 9

Trading Dude

Battle-Tested Backtesters: Comparing VectorBT, Ziplin...

When it comes to developing and validating trading strategies in Pytho...

Jun 13 13

See more recommendations