Member-only story

An Algorithmic Exploration of Quantile Regression Channels

PyQuantLab Following 14 min read · Jun 15, 2025

Financial markets often exhibit complex behaviors that defy simple statistical descriptions. While traditional price channels, like Bollinger Bands or moving average envelopes, typically rely on mean-based calculations or fixed standard deviations, what if a more nuanced understanding of price distribution could yield more adaptive boundaries?

Quantile Regression, unlike ordinary least squares (OLS) regression that models the conditional mean, focuses on modeling conditional quantiles of a response variable. In the context of price channels, this means we can directly estimate

lines representing, for example, the 20th, 50th (median), and 80th percentiles of price over time. This approach allows for channels that are more flexible and responsive to the underlying distribution of prices, potentially adapting better to non-normal price movements and identifying dynamic areas of support and resistance.

This article explores an algorithmic strategy that employs Quantile Regression to define dynamic price channels. The aim is to investigate whether these data-adaptive channels can provide robust signals for breakouts (indicating new trends) or mean-reversion (indicating price returning to “fair value”), offering a more sophisticated interpretation of price behavior than fixed-width or mean-based channels.

📦 Everything you need to go from zero to confident algo trader — in one complete package.

👉 Start building & backtesting proven strategies today:

🛒 Get the Ultimate Bundle

💡 Just Starting Out? Grab the Algo Trading Value Pack

> ⚡ 3 concise eBooks
> 📘 30+ Python-coded strategies
> 🧠 Simple, practical, and actionable — ready to use in Backtrader and real trading platforms

🎯 Ideal if you're looking to learn fast and see results in just a few days.

🛒 Start with the Value Pack

e Core Idea: Flexible Channels from Price Distribution

The strategy's theoretical foundation lies in its ability to dynamically embrace the shape of price distribution:

1. Quantile Regression (QR):

- Concept: Instead of fitting a single line through the *average* of data points, QR fits lines through specific *quantiles* (percentiles). For example, a 0.20-quantile line would represent the level below which 20% of the data falls, and a 0.80-quantile line would represent the level below which 80% of the data falls.
- Advantage for Channels: This allows for channels that can be asymmetric and adapt to varying price volatility and skewness. It posits that price tends to oscillate between these data-driven boundaries.

2. Dynamic Quantile Channels:

- The strategy constructs a channel using three quantile lines: a `lower_quantile` (e.g., 20th percentile), an `upper_quantile` (e.g., 80th percentile), and a `trend_quantile` (e.g., 50th percentile/median). These lines are estimated dynamically over a `lookback_period`.
- Hypothesis: Price breaking out of these channels (moving above the upper or below the lower band) could signal a strong new trend or momentum. Conversely, price returning towards the median or within the channel might signal mean-reversion or a cessation of momentum.

3. Channel Confidence: A `channel_confidence` metric is introduced as an attempt to quantify the reliability of the estimated channels, based on the ratio of expected price dispersion to the actual channel width. This could serve as a filter for trading only when the channel is deemed "trustworthy."

4. Trading Hypothesis:

- Breakout Entry: Enter a long position on an upper channel breakout or a short position on a lower channel breakout, particularly if the `confidence` in the channel is sufficient.
- Mean-Reversion Exit: Close positions if price returns significantly within the channel, particularly near the median line, hypothesizing a return to “fair value.”
- Fixed Stop Loss: A fixed percentage stop-loss provides a basic risk management safeguard.

The overarching aim is to explore whether defining price channels through Quantile Regression, adapting to price distribution rather than just its mean, can unlock novel and effective trading signals.

Algorithmic Implementation: A `backtrader` Strategy

The following `backtrader` code provides a concrete implementation for exploring this Quantile Channel Strategy.

The code uses `scipy.optimize.minimize` for the quantile regression fitting, which is a computationally intensive but mathematically robust approach.

Step 1: The Quantile Regression Engine

This custom `QuantileRegression` class is the mathematical backbone of the strategy, implementing the core algorithm for fitting quantile lines.

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize # For optimization in Quantile
from scipy.stats import norm # For standard normal distribution (no

%matplotlib inline
plt.rcParams['figure.figsize'] = (12, 8) # Set default plot size

class QuantileRegression:
    """Quantile Regression implementation for channel estimation"""

    def __init__(self, tau=0.5):
        self.tau = tau # Quantile level (e.g., 0.5 for median, 0.8

    def quantile_loss(self, y_true, y_pred):
        """Quantile loss function (pinball loss) for a given tau."""
        residual = y_true - y_pred
        # This loss function penalizes under- and over-prediction d
        return np.mean(np.maximum(self.tau * residual, (self.tau -

    def fit(self, X, y):
        """Fits the quantile regression model using numerical optim
        # Reshape X if it's a 1D array to handle single feature cor
```

```
            n_features = X.shape[1] if len(X.shape) > 1 else 1

            # Initialize coefficients (intercept + slope(s)) to zeros
            initial_params = np.zeros(n_features + 1)

            def objective(params):
                """The objective function to minimize: the quantile los
                # Predict y based on current parameters
                if len(X.shape) == 1: # Handle 1D input (time_normalize
                    y_pred = params[0] + params[1] * X # params[0] is i
                else: # Handle multi-dimensional input (not applicable
                    y_pred = params[0] + np.dot(X, params[1:])
                return self.quantile_loss(y, y_pred)

            # Use scipy.optimize.minimize to find the parameters that m
            try:
                result = minimize(objective, initial_params, method='L-
                self.coef_ = result.x # Store the optimized coefficient
                return self
            except Exception:
                # Fallback for numerical stability issues during optimi
                # This simply returns a horizontal line at the specifie
                self.coef_ = np.array([np.quantile(y, self.tau), 0])
                return self

        def predict(self, X):
            """Predicts y values using the fitted model."""
            if not hasattr(self, 'coef_'):
                raise ValueError("Model must be fitted before predictio

            # Use the stored coefficients to make predictions
            if len(X.shape) == 1:
                return self.coef_[0] + self.coef_[1] * X # Intercept +
            else:
                return self.coef_[0] + np.dot(X, self.coef_[1:])
```

Analysis of the Quantile Regression Engine:

- `QuantileRegression` Class: This custom class implements the core mathematical concept. It takes a `tau` parameter (the quantile level, e.g., 0.2, 0.5, 0.8).
- `quantile_loss()` : This is the heart of quantile regression. It's also known as the "pinball loss" function. It penalizes under- and over-predictions differently based on the `tau` value, allowing the regression line to fit a specific quantile of the data rather than just the mean.
- `fit(X, y)` : This method performs the actual regression. It uses `scipy.optimize.minimize` to find the set of coefficients (`params`) that minimize the `quantile_loss` for the given input data `X` (typically time or an independent variable) and `y` (the price data). This numerical optimization is what makes the quantile lines adapt to the data's distribution.
- `predict(X)` : Once the model is fitted, this method uses the optimized coefficients to predict the corresponding quantile value for new `X` inputs (e.g., to predict the channel boundary at the current time).

Step 2: Channel Construction: Strategy Initialization and Estimation

This section covers the `QuantileChannelStrategy` 's setup, initializing the various quantile regression models, and the `estimate_channels` method, which dynamically calculates the upper, lower, and median lines of the channel.

```
class QuantileChannelStrategy(bt.Strategy):
    params = (
        ('lookback_period', 60),      # Lookback for channel estima
        ('upper_quantile', 0.8),      # Upper channel quantile (80t
        ('lower_quantile', 0.2),      # Lower channel quantile (20t
        ('trend_quantile', 0.5),      # Trend line quantile (median
        ('breakout_threshold', 1.02), # Breakout confirmation (2% a
        ('stop_loss_pct', 0.08),      # 8% fixed stop loss
        ('rebalance_period', 1),      # Rebalance every N days (1 =
        ('min_channel_width', 0.02),  # Minimum 2% channel width (t
        ('volume_confirm', False),    # Volume confirmation (curren
    )

    def __init__(self):
        # Data storage for channel estimation
        self.prices = []      # Stores historical closing prices
        self.time_indices = [] # Stores numerical time indices (0,

        # Channel estimates storage for plotting/analysis (not dire
        self.upper_channel = []
        self.lower_channel = []
        self.trend_line = []
        self.channel_width = []
        self.channel_confidence = 0 # Placeholder for calculated co

        # Quantile regression models for each channel line
        self.upper_qr = QuantileRegression(tau=self.params.upper_qu
        self.lower_qr = QuantileRegression(tau=self.params.lower_qu
        self.trend_qr = QuantileRegression(tau=self.params.trend_qu

        # Trading variables
        self.rebalance_counter = 0 # Counts bars until next rebalan
        self.stop_price = 0        # The current stop loss price (f
        self.trade_count = 0       # Number of trades
        self.breakout_direction = 0 # 1=upper breakout, -1=lower br

        # Breakout tracking (for analysis, not strategy logic)
        self.upper_breakouts = 0
        self.lower_breakouts = 0
        self.false_breakouts = 0

    def estimate_channels(self):
```

```
        """
        Estimates the upper, lower, and median (trend) lines of the
        using Quantile Regression on a lookback window of price dat
        """
        # Ensure enough data for the lookback period
        if len(self.prices) < self.params.lookback_period:
            return None, None, None, 0 # Return None if not enough

        # Get recent price and time data for regression
        recent_prices = np.array(self.prices[-self.params.lookback_
        recent_times = np.array(self.time_indices[-self.params.look

        # Normalize time indices for numerical stability in regress
        time_normalized = (recent_times - recent_times[0]) / (recen

        try:
            # Fit each quantile regression model
            self.upper_qr.fit(time_normalized, recent_prices)
            self.lower_qr.fit(time_normalized, recent_prices)
            self.trend_qr.fit(time_normalized, recent_prices)

            # Predict the channel levels at the "current" point (no
            current_time_norm = 1.0

            upper_level = self.upper_qr.predict(np.array([current_t
            lower_level = self.lower_qr.predict(np.array([current_t
            trend_level = self.trend_qr.predict(np.array([current_t

            # Calculate channel width and enforce minimum width
            channel_width = (upper_level - lower_level) / trend_lev
            if channel_width < self.params.min_channel_width:
                # If too narrow, expand symmetrically to meet min_c
                mid_price = (upper_level + lower_level) / 2
                half_width = mid_price * self.params.min_channel_wi
                upper_level = mid_price + half_width
                lower_level = mid_price - half_width
                channel_width = self.params.min_channel_width # Upd

            # Calculate channel confidence (relative to price stand
            price_std = np.std(recent_prices)
            expected_width = 2 * price_std / np.mean(recent_prices)
            confidence = min(1.0, expected_width / (channel_width +

            return upper_level, lower_level, trend_level, confidenc
```

```
        except Exception as e:
            # Fallback to simple quantile calculation if regression
            # This means it won't be a sloping line, just a horizon
            upper_level = np.quantile(recent_prices, self.params.up
            lower_level = np.quantile(recent_prices, self.params.lo
            trend_level = np.quantile(recent_prices, self.params.tr
            confidence = 0.5 # Default confidence on fallback

            return upper_level, lower_level, trend_level, confidenc
```

Analysis of Channel Construction:

- `params`: These parameters control the channel's behavior: `lookback_period` (how many past bars to consider), `upper_quantile`, `lower_quantile`, `trend_quantile` (defining the channel boundaries), `breakout_threshold`, `min_channel_width` (to prevent overly flat channels), `stop_loss_pct`, and `rebalance_period`.
- `__init__(self)`:
- Data Storage: `self.prices` and `self.time_indices` are lists to store the historical price and time data needed by the `QuantileRegression` models.
- QR Model Initialization: Three instances of `QuantileRegression` are created: `upper_qr`, `lower_qr`, and `trend_qr`, each configured for a specific quantile (`tau`).

- Tracking Variables: Variables are initialized to store channel estimates (`upper_channel` , `lower_channel` , `trend_line`), confidence (`channel_confidence`), and basic trade tracking (`rebalance_counter` , `stop_price` , `trade_count`).
- `estimate_channels()` : This is the core function for channel estimation.
- Data Preparation: It gathers recent price and time data, normalizing the `time_indices` for numerical stability in regression.
- Quantile Regression Fitting: It calls the `fit()` method on each of the `upper_qr` , `lower_qr` , and `trend_qr` models, using the normalized time as the independent variable (`X`) and prices as the dependent variable (`y`).
- Prediction: It then uses the `predict()` method to get the current values of these quantile lines.
- Channel Width & Minimum Width: It calculates the channel width and enforces a `min_channel_width` to avoid overly narrow or flat channels that might give false signals.
- Channel Confidence: A `confidence` metric is calculated, hypothesizing that a channel that is tighter relative to the price's standard deviation (more "predictive" of the current price range) is more confident. This attempts to quantify the reliability of the channel.

- Error Handling: A `try-except` block handles potential numerical issues during optimization, falling back to a simpler, horizontal quantile calculation if regression fails.

Step 3: Trading Logic and Backtesting Execution

This section contains the `next` method, which orchestrates the strategy's bar-by-bar decision-making, along with the full `backtrader` execution block for running the backtest and analyzing results.

```
    def detect_breakout(self, current_price, upper_channel, lower_c
        """Detect channel breakout with confirmation."""
        breakout = 0

        # Upper breakout: Price significantly above upper channel
        if current_price > upper_channel * self.params.breakout_thr
            breakout = 1 # Signal for long
            self.upper_breakouts += 1 # Track for analysis

        # Lower breakout: Price significantly below lower channel
        elif current_price < lower_channel / self.params.breakout_t
            breakout = -1 # Signal for short
            self.lower_breakouts += 1 # Track for analysis

        return breakout

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

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

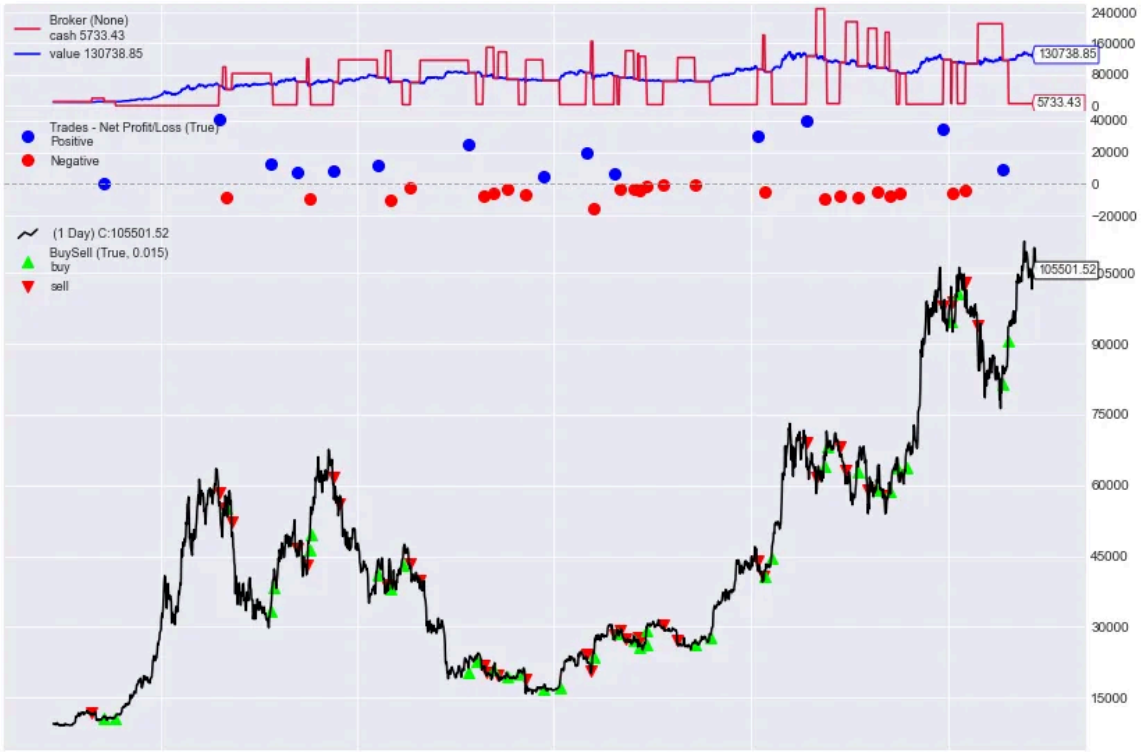
```
        # Keep only recent history for lookback period (plus a buff
        if len(self.prices) > self.params.lookback_period * 2:
            self.prices = self.prices[-self.params.lookback_period
            self.time_indices = self.time_indices[-self.params.look

        # Estimate channels for the current bar
        upper_channel, lower_channel, trend_line, confidence = self

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

        # Store channel estimates for potential external plotting o
        self.upper_channel.append(upper_channel)
        self.lower_channel.append(lower_channel)
        self.trend_line.append(trend_line)
        self.channel_confidence = confidence # Store current confid

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

        # Rebalancing logic (fixed frequency and stop-loss check)
        self.rebalance_counter += 1
        if self.rebalance_counter < self.params.rebalance_period:
            # Check stop loss only during non-rebalance bars
            if self.position.size > 0 and current_price <= self.sto
                self.close() # Close long position
                self.log(f'STOP LOSS - Long closed at {current_pric
            elif self.position.size < 0 and current_price >= self.s
                self.close() # Close short position
                self.log(f'STOP LOSS - Short closed at {current_pri
            return # Skip trading logic if not a rebalance bar

        # Reset rebalance counter for next period
        self.rebalance_counter = 0

        # Detect breakout from the channel
        breakout = self.detect_breakout(current_price, upper_channe

        # Determine current position state (1=long, -1=short, 0=fla
        current_pos = 0
        if self.position.size > 0:
            current_pos = 1
        elif self.position.size < 0:
```

```
            current_pos = -1

        # --- Trading logic with channel confirmation ---
        # Only trade if a breakout occurred AND channel confidence
        if breakout != 0 and confidence > 0.3: # Require minimum co
            # Close existing position if breakout direction is oppo
            if current_pos != 0 and current_pos != breakout:
                self.close()
                current_pos = 0 # Now flat

            # Open new position if currently flat and a breakout oc
            if current_pos == 0:
                if breakout == 1: # Upper breakout - go long
                    self.buy()
                    self.stop_price = lower_channel # Set initial s
                    self.trade_count += 1
                    self.breakout_direction = 1 # Store breakout di
                    self.log(f'UPPER BREAKOUT LONG - Price: {curren

                elif breakout == -1: # Lower breakout - go short
                    self.sell()
                    self.stop_price = upper_channel # Set initial s
                    self.trade_count += 1
                    self.breakout_direction = -1 # Store breakout d
                    self.log(f'LOWER BREAKOUT SHORT - Price: {curre

        # --- Exit on return to channel (mean reversion) ---
        # If currently in a position and price moves back inside th
        elif self.position.size != 0:
            in_channel = lower_channel <= current_price <= upper_ch
            # Exit if price returned to channel AND is close to the
            if in_channel and abs(current_price - trend_line) / tre
                self.close()
                self.log(f'RETURN TO CHANNEL - Position closed at {

        # --- Update trailing stop for existing positions ---
        # This acts as a manual trailing stop, moving the stop to t
        if self.position.size > 0: # Long position
            new_stop = max(self.stop_price, lower_channel) # Stop n
            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 n
```

```
            if new_stop < self.stop_price:
                self.stop_price = new_stop

    def log(self, txt, dt=None):
        """Standard logging function for strategy messages."""
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()}: {txt}')

    def notify_order(self, order):
        """Notifies about order status changes."""
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'LONG EXECUTED - Price: {order.executed.p
            elif order.issell():
                if self.position.size == 0: # If closing a long or
                    self.log(f'POSITION CLOSED - Price: {order.exec
                else: # If opening a short
                    self.log(f'SHORT EXECUTED - Price: {order.execu

    def notify_trade(self, trade):
        """Notifies about trade closures and tracks false breakouts
        if trade.isclosed:
            self.log(f'TRADE CLOSED - PnL: {trade.pnl:.2f}')
            # Hypothesis: A trade with very small profit/loss is a
            if abs(trade.pnl) < abs(trade.size * trade.price * 0.01
                self.false_breakouts += 1

    def stop(self):
        """Called at the very end of the backtest. Prints final str
        print(f'\n=== QUANTILE REGRESSION CHANNEL BREAKOUT RESULTS
        print(f'Total Trades: {self.trade_count}')
        print(f'Upper Breakouts: {self.upper_breakouts}')
        print(f'Lower Breakouts: {self.lower_breakouts}')
        print(f'False Breakouts: {self.false_breakouts}')

        # Print channel characteristics if available
        if len(self.channel_width) > 0:
            avg_width = np.mean(self.channel_width)
            width_std = np.std(self.channel_width)
            print(f'Average Channel Width: {avg_width:.3f}')
            print(f'Channel Width Std: {width_std:.3f}')
            print(f'Final Confidence: {self.channel_confidence:.3f}

        if len(self.upper_channel) > 0:
```

```
            print(f'Final Upper Channel: {self.upper_channel[-1]:.2
            print(f'Final Lower Channel: {self.lower_channel[-1]:.2
            print(f'Final Trend Line: {self.trend_line[-1]:.2f}')

        # Calculate and print a "Breakout Success Rate"
        success_rate = 1 - (self.false_breakouts / max(1, self.trad
        print(f'Breakout Success Rate: {success_rate:.2%}')

# Download BTC-USD data for the backtest
print("Downloading BTC-USD data...")
ticker = "BTC-USD"
data = yf.download(ticker, period="5y", interval="1d")

# Clean multi-level columns if yfinance returns them
data.columns = data.columns.droplevel(1)

# Create backtrader data feed from the pandas DataFrame
bt_data = bt.feeds.PandasData(dataname=data)

# Initialize Cerebro engine
cerebro = bt.Cerebro()

# Add the Quantile Regression Channel strategy
cerebro.addstrategy(QuantileChannelStrategy)

# Add the data feed to Cerebro
cerebro.adddata(bt_data)

# Set initial capital for the backtest
cerebro.broker.setcash(10000.0)

# Set commission for realism
cerebro.broker.setcommission(commission=0.001) # 0.1% commission

# Add a position sizer to allocate 95% of capital per trade
cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

# Add analyzers to compute performance metrics after the backtest
cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe')
cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')

print(f'Starting Portfolio Value: {cerebro.broker.getvalue():.2f}')
```

```
# Run the backtest simulation
results = cerebro.run()
strat = results[0] # Get the strategy instance from the results

print(f'Final Portfolio Value: {cerebro.broker.getvalue():.2f}')

# Print performance metrics from analyzers with error handling
try:
    sharpe = strat.analyzers.sharpe.get_analysis()
    if sharpe and 'sharperatio' in sharpe and sharpe['sharperatio']
        print(f'Sharpe Ratio: {sharpe["sharperatio"]:.2f}')
    else:
        print('Sharpe Ratio: N/A (insufficient trades or data)')
except Exception:
    print('Sharpe Ratio: N/A (calculation error)')

try:
    drawdown = strat.analyzers.drawdown.get_analysis()
    if drawdown and 'max' in drawdown and 'drawdown' in drawdown['m
        print(f'Max Drawdown: {drawdown["max"]["drawdown"]:.2f}%')
    else:
        print('Max Drawdown: N/A')
except Exception:
    print('Max Drawdown: N/A (calculation error)')

try:
    returns = strat.analyzers.returns.get_analysis()
    if returns and 'rtot' in returns:
        print(f'Total Return: {returns["rtot"]:.2%}')
    else:
        print('Total Return: N/A')
except Exception:
    print('Total Return: N/A (calculation error)')

# Plot results of the backtest
print("\nPlotting Quantile Channel Strategy results...")
cerebro.plot(iplot=False, style='line')
plt.show()
```



Conclusion

The exploration of the Quantile Channel Strategy delves into a fascinating frontier of technical analysis, moving beyond traditional mean-based price channels towards a more sophisticated, data-adaptive approach. By harnessing the power of Quantile Regression, this strategy endeavors to define dynamic price boundaries that inherently adapt to the underlying distribution of prices, offering a potentially more nuanced understanding of market behavior.

The core ambition lies in its ability to estimate channels that are not merely symmetric deviations from an average, but rather responsive to where prices truly concentrate across various

quantiles. This provides a compelling framework for identifying potential breakouts and mean-reversion points based on the market's evolving internal structure. The introduction of concepts like "channel confidence" further attempts to filter signals, adding a layer of self-awareness to the strategy.

However, this innovative approach comes with inherent complexities. The computational intensity of fitting quantile regressions for each bar necessitates significant processing power and optimization for practical application. Furthermore, the selection and tuning of parameters, as well as the interpretation of signals from these statistically derived channels, require rigorous empirical validation to guard against overfitting. The strategy remains highly experimental, serving primarily as a potent research tool rather than a readily deployable system.

Ultimately, the Quantile Channel Strategy exemplifies the continuous quest in quantitative finance: to push the boundaries of market understanding by integrating advanced mathematical concepts. It highlights the intriguing possibilities of building strategies that are not just reactive to price, but are deeply informed by the probabilistic landscape of market data, continuously refining our understanding of where value truly lies.

Python Trading Crypto Regression Finance

Written by PyQuantLab

655 followers · 6 following

Following

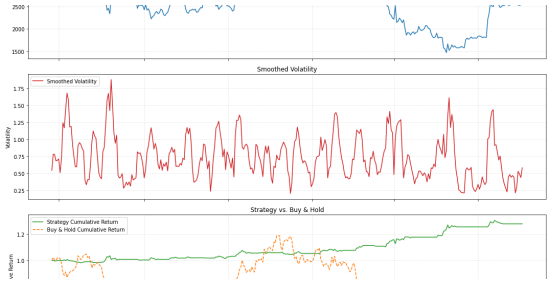
Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

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...

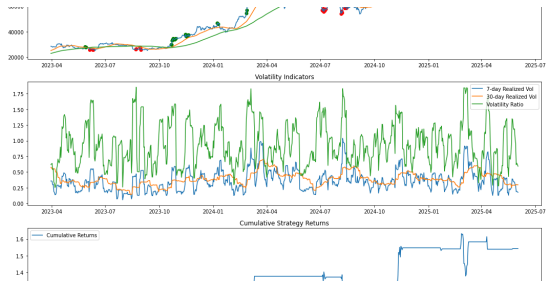
Jun 3 32 3

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

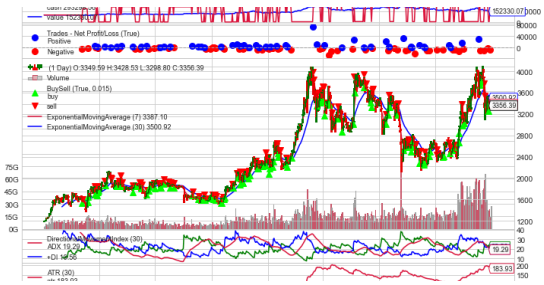


PyQuantLab

Trend-Volatility Confluence Trading Strategy

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

Jun 3 60



PyQuantLab

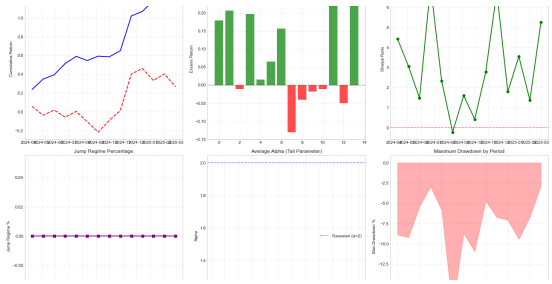
Building an Adaptive Trading Strategy with Backtrader: A...

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

Jun 4 64

See all from PyQuantLab

Recommended from Medium



PyQuantLab

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

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

Jun 5 102

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

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...

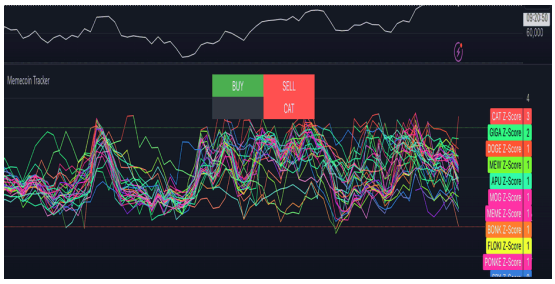
Jun 3 75 3

FMZQuant

AI-Driven Multi-Factor Quantitative Trading Strategy

Overview

Jun 9 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

See more recommendations