Member-only story

Dynamic Trend Capture: Rolling Backtest of Polynomial Channel Breakout Strategy with Adaptive Trailing Stops

PyQuantLab Following 14 min read · 5 days ago

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

This article introduces a sophisticated quantitative trading strategy, `PolynomialChannelBreakoutStrategy`, which leverages polynomial regression to dynamically identify price channels. Instead of fixed-width bands, these channels adapt to the non-linear trends of the market. The strategy aims to capitalize on

channel breakouts and employs an Average True Range (ATR)-based trailing stop for robust risk management.

🚀 Supercharge Your Algorithmic Trading Today

Unlock the full potential of your trading with powerful tools:

✅ Ultimate Algo Trading Bundle
The all-in-one arsenal: 6 eBooks + 80+ Python strategies + powerful Backtester App. Master everything from technical analysis to machine learning — across crypto, stocks, and forex.
👉 Get the Ultimate Bundle →

✅ Algo Strategy Code Bundle
Deploy instantly with 80+ ready-to-run strategies across 5 core categories. Includes step-by-step guides for rapid implementation and customization.
👉 Get the Strategy Bundle →

✅ Algo Trading Value Pack
Just starting out? Get 3 beginner-friendly eBooks + 30+ strategies designed for fast wins and hands-on learning.
👉 Start with the Value Pack →

Choose your level. Automate your edge. Start winning with code.

1. The Polynomial Channel Breakout Strategy Concept

Traditional trading channels often rely on simple linear regression or fixed-width bands (like Bollinger Bands). However, real-world price movements are rarely linear. This strategy addresses this by fitting a polynomial curve to historical price data, allowing the channel to better capture the actual shape of the trend.

Key Components:

- Polynomial Channel Indicator: This custom `backtrader` indicator is the heart of the strategy. It:
- Fits a polynomial regression line to the closing prices over a defined `lookback` period. The `degree` parameter controls the flexibility of this curve (e.g., degree 1 for linear, degree 2 for quadratic, etc.).
- Calculates the residuals (the differences between actual prices and the regression line).
- Determines the upper and lower channels by adding/subtracting a `channel_width` multiple of the standard deviation of these residuals from the regression line. This creates a dynamic, volatility-adaptive channel around the polynomial trend.
- Breakout Entry Logic:

- Long Entry: Occurs when the current price breaks above the upper polynomial channel. This signals a strong upward momentum beyond the established trend.
- Short Entry: Occurs when the current price breaks below the lower polynomial channel. This signals a strong downward momentum.
- Adaptive Trailing Stop-Loss: This is the primary exit mechanism, designed to protect profits and limit losses.
- An ATR-based trailing stop is calculated as a multiple (`trail_atr_mult`) of the Average True Range (`atr_period`).
- For long positions, the stop price moves up as the market moves favorably, but never moves down.
- For short positions, the stop price moves down as the market moves favorably, but never moves up.
- If the price hits this trailing stop level, the position is automatically closed.
- Optional Regression Line Exit: The strategy includes an optional (disabled by default) exit rule where:
- A long position exits if the price breaks below the regression line.
- A short position exits if the price breaks above the regression line.

This acts as a trend-reversal signal, closing trades when the price crosses the central polynomial trend.

2. The `PolynomialChannelIndicator`

This custom indicator is fundamental to the strategy. It leverages `scikit-learn` for polynomial regression.

```
import backtrader as bt
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
import warnings

# Suppress sklearn warnings if they occur during fitting
warnings.filterwarnings('ignore')

class PolynomialChannelIndicator(bt.Indicator):
    lines = ('upper_channel', 'lower_channel', 'regression_line')

    params = (
        ('degree', 3),           # Polynomial degree
        ('channel_width', 2.0),  # Channel width in standard deviat
        ('lookback', 50),        # Lookback period for regression
    )

    plotinfo = dict(
        plot=True,
        subplot=False, # Plot on the main price chart
        plotlinelabels=True
    )

    plotlines = dict(
        upper_channel=dict(color='red', ls='--', alpha=0.7),
        lower_channel=dict(color='green', ls='--', alpha=0.7),
```

```
        regression_line=dict(color='blue', ls='-', alpha=0.8)
    )

    def __init__(self):
        self.addminperiod(self.params.lookback) # Ensure enough dat

    def next(self):
        """Calculate polynomial channels for current bar."""
        if len(self.data) < self.params.lookback:
            return

        # Get price data for the lookback period, reversed to get n
        prices = np.array([self.data.close[-i] for i in range(self.

        try:
            # Create x values (time points from 0 to lookback-1)
            x = np.arange(len(prices)).reshape(-1, 1)

            # Create a pipeline for polynomial features and linear
            poly_reg = make_pipeline(PolynomialFeatures(self.params
            poly_reg.fit(x, prices) # Fit the model to prices

            # Predict values using the fitted model
            y_pred = poly_reg.predict(x)

            # Calculate residuals (difference between actual and pr
            residuals = prices - y_pred
            std_residuals = np.std(residuals) # Standard deviation

            # The current regression value is the last predicted po
            current_regression = y_pred[-1]

            # Set channel boundaries for the current bar
            self.lines.upper_channel[0] = current_regression + (sel
            self.lines.lower_channel[0] = current_regression - (sel
            self.lines.regression_line[0] = current_regression

        except Exception as e:
            # Fallback in case of calculation error (e.g., singular
            # Use previous values if available, otherwise NaN
            if len(self) > 1:
                self.lines.upper_channel[0] = self.lines.upper_chan
                self.lines.lower_channel[0] = self.lines.lower_chan
                self.lines.regression_line[0] = self.lines.regressi
```

```
            else:
                self.lines.upper_channel[0] = float('nan')
                self.lines.lower_channel[0] = float('nan')
                self.lines.regression_line[0] = float('nan')
```

Explanation of `PolynomialChannelIndicator`:

- `lines` and `plotlines`: Define the indicator's outputs (`upper_channel`, `lower_channel`, `regression_line`) and their plotting properties (colors, line styles).
- `params`: Configurable parameters for the polynomial `degree`, `channel_width` (in standard deviations), and `lookback` period for regression.
- `__init__(self)`: Sets the minimum period required for the indicator to start calculating, ensuring enough data.
- `next(self)`: This method runs for each new bar.
- It extracts the `close` prices for the `lookback` period.
- It uses `make_pipeline` from `sklearn` to create a `PolynomialFeatures` transformer followed by `LinearRegression`. This pipeline fits a polynomial curve to the price data.
- It calculates the `residuals` (the difference between actual prices and the fitted curve) and their standard deviation. This

standard deviation represents the typical deviation from the polynomial trend.

- The `current_regression` value is the last point on the fitted polynomial curve.
- Finally, it sets the `upper_channel` and `lower_channel` based on the `current_regression` plus/minus `channel_width` multiples of `std_residuals`. The `regression_line` itself is also exposed.
- Includes error handling to prevent issues if regression fails (e.g., due to insufficient data or degenerate cases).

3. The `PolynomialChannelBreakoutStrategy` Implementation

```
import backtrader as bt
import backtrader.indicators as btind # Used for ATR
import numpy as np
# Import PolynomialChannelIndicator here or ensure it's defined abo

class PolynomialChannelBreakoutStrategy(bt.Strategy):
    params = (
        ('degree', 3),              # Polynomial degree for the chan
        ('channel_width', 2.0),     # Channel width in standard devi
        ('lookback', 30),           # Lookback period for polynomial
        ('trail_atr_mult', 3.0),    # ATR multiple for trailing stop
        ('atr_period', 14),         # ATR period for trailing stop
        ('use_regression_exit', False), # Option to exit on regress
        ('printlog', True),         # Enable/disable logging
    )

    def __init__(self):
```

```
        self.dataclose = self.datas[0].close

        # Instantiate our custom Polynomial Channel Indicator
        self.poly_channel = PolynomialChannelIndicator(
            self.datas[0], # Pass the data feed to the indicator
            degree=self.params.degree,
            channel_width=self.params.channel_width,
            lookback=self.params.lookback
        )

        # ATR for trailing stops
        self.atr = btind.ATR(period=self.params.atr_period)

        # Trailing stop variables
        self.trail_stop = None       # The current price level of t
        self.entry_price = None      # Price at which the current p
        self.position_type = 0       # 0: no position, 1: long, -1:
        self.order = None            # To track active entry/exit o

        # Counters for logging strategy activity
        self.signal_count = 0
        self.long_signals = 0
        self.short_signals = 0
        self.exit_signals = 0        # Exits from regression line
        self.trail_exits = 0         # Exits from trailing stop hit

    def log(self, txt, dt=None):
        """Logging function for strategy actions."""
        if self.params.printlog:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()}: {txt}')

    def notify_order(self, order):
        """Handles order notifications and sets initial trailing st
        if order.status in [order.Submitted, order.Accepted]:
            return # Order is pending, nothing to do yet

        if order.status in [order.Completed]:
            if order.isbuy(): # A buy order has completed (either e
                self.log(f'BUY EXECUTED: Price: {order.executed.pri
                         f'Cost: {order.executed.value:.2f}, Comm:
                # If we are now in a long position, set initial tra
                if self.position.size > 0:
                    self.position_type = 1
```

```
                    self.entry_price = order.executed.price
                    # Calculate initial trailing stop
                    self.trail_stop = self.entry_price - (self.atr[
                    self.log(f'INITIAL LONG STOP set at: {self.trai

            elif order.issell(): # A sell order has completed (eith
                self.log(f'SELL EXECUTED: Price: {order.executed.pr
                         f'Cost: {order.executed.value:.2f}, Comm:
                # If we are now in a short position, set initial tr
                if self.position.size < 0: # This means it was an o
                    self.position_type = -1
                    self.entry_price = order.executed.price
                    # Calculate initial trailing stop
                    self.trail_stop = self.entry_price + (self.atr[
                    self.log(f'INITIAL SHORT STOP set at: {self.tra
                else: # This means it was a closing order for a lon
                    self.position_type = 0 # Position closed
                    self.trail_stop = None # Reset trailing stop tr
                    self.entry_price = None

        elif order.status in [order.Canceled, order.Margin, order.R
            self.log(f'Order Failed: Status {order.getstatusname()}
            # Reset order reference if it failed

        self.order = None # Clear general order reference after pro

    def notify_trade(self, trade):
        """Handle trade notifications (when a position is fully clo
        if not trade.isclosed:
            return # Only interested in closed trades

        self.log(f'TRADE CLOSED: Gross P&L: {trade.pnl:.2f}, Net P&
        # Reset position type and trailing stop after trade closure
        self.position_type = 0
        self.trail_stop = None
        self.entry_price = None

    def update_trailing_stop(self, current_price):
        """Dynamically updates the ATR-based trailing stop."""
        if self.trail_stop is None or self.position_type == 0:
            return # No active position or stop

        # Ensure ATR has a valid value
        if np.isnan(self.atr[0]):
```

```
            return

        stop_distance = self.atr[0] * self.params.trail_atr_mult

        if self.position_type == 1: # Long position
            new_stop = current_price - stop_distance
            # Trail stop up with price, never down (only raise the
            if new_stop > self.trail_stop:
                old_stop = self.trail_stop
                self.trail_stop = new_stop
                # Log only if the stop moved significantly
                if abs(new_stop - old_stop) / old_stop > 0.005: # >
                    self.log(f'LONG STOP UPDATED: {old_stop:.2f} ->

        elif self.position_type == -1: # Short position
            new_stop = current_price + stop_distance
            # Trail stop down with price, never up (only lower the
            if new_stop < self.trail_stop:
                old_stop = self.trail_stop
                self.trail_stop = new_stop
                # Log only if the stop moved significantly
                if abs(new_stop - old_stop) / old_stop > 0.005: # >
                    self.log(f'SHORT STOP UPDATED: {old_stop:.2f} -

    def next(self):
        """Main strategy logic executed on each bar."""

        # 1. Skip if indicators not ready or pending order
        # Ensure enough data for PolynomialChannelIndicator and ATR
        min_indicator_period = max(self.params.lookback, self.param
        if len(self) < min_indicator_period + 1: # +1 for current b
            return

        if self.order: # Prevent new orders if one is already pendi
            return

        # Check for NaN values from indicators
        if (np.isnan(self.poly_channel.upper_channel[0]) or
            np.isnan(self.poly_channel.lower_channel[0]) or
            np.isnan(self.poly_channel.regression_line[0]) or
            np.isnan(self.atr[0])):
            self.log("Indicators not ready (NaN values). Waiting fo
            return
```

```
        current_price = self.dataclose[0]
        # Get previous prices and indicator values for crossover ch
        prev_price = self.dataclose[-1]
        upper_channel = self.poly_channel.upper_channel[0]
        lower_channel = self.poly_channel.lower_channel[0]
        regression_line = self.poly_channel.regression_line[0]

        prev_upper = self.poly_channel.upper_channel[-1]
        prev_lower = self.poly_channel.lower_channel[-1]
        prev_regression = self.poly_channel.regression_line[-1]

        # 2. Handle existing positions (Trailing Stop & Optional Re
        if self.position:
            # Update trailing stop (this just updates the price, do
            self.update_trailing_stop(current_price)

            # Check if trailing stop has been hit (current price cr
            # For backtrader, this logic is usually handled by the
            # but here we manage it manually for clear logging and
            if self.position_type == 1: # Long position
                if current_price <= self.trail_stop:
                    self.trail_exits += 1
                    self.log(f'LONG TRAILING STOP HIT: Price {curre
                    self.order = self.close() # Close the position
                    return # Exit after placing close order

            elif self.position_type == -1: # Short position
                if current_price >= self.trail_stop:
                    self.trail_exits += 1
                    self.log(f'SHORT TRAILING STOP HIT: Price {curr
                    self.order = self.close() # Close the position
                    return # Exit after placing close order

            # Optional: Exit if price crosses the regression line
            if self.params.use_regression_exit:
                # Exit long if price breaks below regression line
                if (self.position_type == 1 and
                    current_price < regression_line and
                    prev_price >= prev_regression): # Crossover che

                    self.exit_signals += 1
                    self.log(f'LONG REGRESSION EXIT: Price {current
                    self.order = self.close()
                    return
```

```
                # Exit short if price breaks above regression line
                elif (self.position_type == -1 and
                      current_price > regression_line and
                      prev_price <= prev_regression): # Crossover c

                    self.exit_signals += 1
                    self.log(f'SHORT REGRESSION EXIT: Price {curren
                    self.order = self.close()
                    return

        # 3. Entry Logic - only if currently no position
        else:
            # Long signal: Current price breaks above upper channel
            if (current_price > upper_channel and
                prev_price <= prev_upper):

                self.signal_count += 1
                self.long_signals += 1
                self.log(f'LONG ENTRY SIGNAL #{self.signal_count}:
                self.order = self.buy() # Place buy order

            # Short signal: Current price breaks below lower channe
            elif (current_price < lower_channel and
                  prev_price >= prev_lower):

                self.signal_count += 1
                self.short_signals += 1
                self.log(f'SHORT ENTRY SIGNAL #{self.signal_count}:
                self.order = self.sell() # Place sell (short) order

    def stop(self):
        """Called at the very end of the backtest to provide a summ
        self.log(f'\n=== STRATEGY SUMMARY ===')
        self.log(f'Total Entry Signals Generated: {self.signal_coun
        self.log(f'Total Long Entry Signals: {self.long_signals}')
        self.log(f'Total Short Entry Signals: {self.short_signals}'
        self.log(f'Total Trailing Stop Exits: {self.trail_exits}')
        if self.params.use_regression_exit:
            self.log(f'Total Regression Line Exits: {self.exit_sign
        self.log(f'Final Portfolio Value: ${self.broker.getvalue():
```

Explanation of `PolynomialChannelBreakoutStrategy`:

- `params`: Extensive parameters for the `PolynomialChannelIndicator` (polynomial `degree`, `channel_width`, `lookback`), `ATR` trailing stop (`trail_atr_mult`, `atr_period`), and optional `use_regression_exit`.
- `__init__(self)`:
- Initializes `self.dataclose` and instantiates `self.poly_channel` (our custom indicator) and `self.atr` (from `backtrader.indicators`).
- Initializes `self.trail_stop`, `self.entry_price`, and `self.position_type` for managing the adaptive trailing stop and current position state.
- `self.order` tracks pending orders.
- Includes counters (`signal_count`, `long_signals`, `short_signals`, `exit_signals`, `trail_exits`) for detailed logging and strategy summary.
- `log(self, txt, dt=None)`: A simple logging function.
- `notify_order(self, order)`: Handles order notifications.
- When a `buy` order completes and results in a long position, it sets `self.position_type = 1`, records `self.entry_price`, and calculates the initial `self.trail_stop` based on ATR.

- When a `sell` order completes and results in a short position, it sets `self.position_type = -1`, records `self.entry_price`, and calculates the initial `self.trail_stop`.
- If a `sell` order completes and closes a long position (i.e., `self.position_type` was 1 and now it's 0), it resets `self.position_type` and `self.trail_stop`.
- It also handles `Canceled`, `Margin`, or `Rejected` orders by logging the failure and clearing the `self.order` reference.
- `notify_trade(self, trade)`: Logs the profit/loss of a fully closed trade and resets `self.position_type` and `self.trail_stop` to reflect no active position.
- `update_trailing_stop(self, current_price)`: This method is called in `next()` to continuously adjust the trailing stop.
- It calculates a `new_stop` price based on the `current_price` and `ATR` multiplier.
- For long positions, it only updates `self.trail_stop` if `new_stop` is higher (trailing upwards).
- For short positions, it only updates `self.trail_stop` if `new_stop` is lower (trailing downwards).
- It logs significant stop updates.
- `next(self)`: This is the core logic, executed on each new bar.

- Indicator Readiness Check: Ensures that the `PolynomialChannelIndicator` and `ATR` have enough data and are not producing `NaN` values before proceeding.
- Order Pending Check: Prevents new orders if one is already being processed.
- Handle Existing Positions:
- Calls `self.update_trailing_stop()` to adjust the stop price for the current bar.
- Checks if `current_price` has hit the `self.trail_stop`. If so, it increments `self.trail_exits`, logs the event, and `self.close()` s the position.
- If `self.params.use_regression_exit` is true, it also checks for price crossing the `regression_line` as an exit signal, logging and closing the position if it occurs.
- Entry Logic (only if no position):
- Long Entry: Checks if `current_price` has crossed `above` the `upper_channel` (from below or at the previous bar). If so, it logs a long signal, increments counters, and places a `self.buy()` order.
- Short Entry: Checks if `current_price` has crossed `below` the `lower_channel` (from above or at the previous bar). If so, it logs a short signal, increments counters, and places a `self.sell()` order.

- `stop(self)` : Called at the end of the backtest to print a summary of the strategy's activity and final portfolio value.

4. Backtesting and Analysis

The provided script includes a dedicated `run_backtest()` function for single, direct backtests and also leverages your robust rolling backtesting framework for comprehensive performance evaluation.

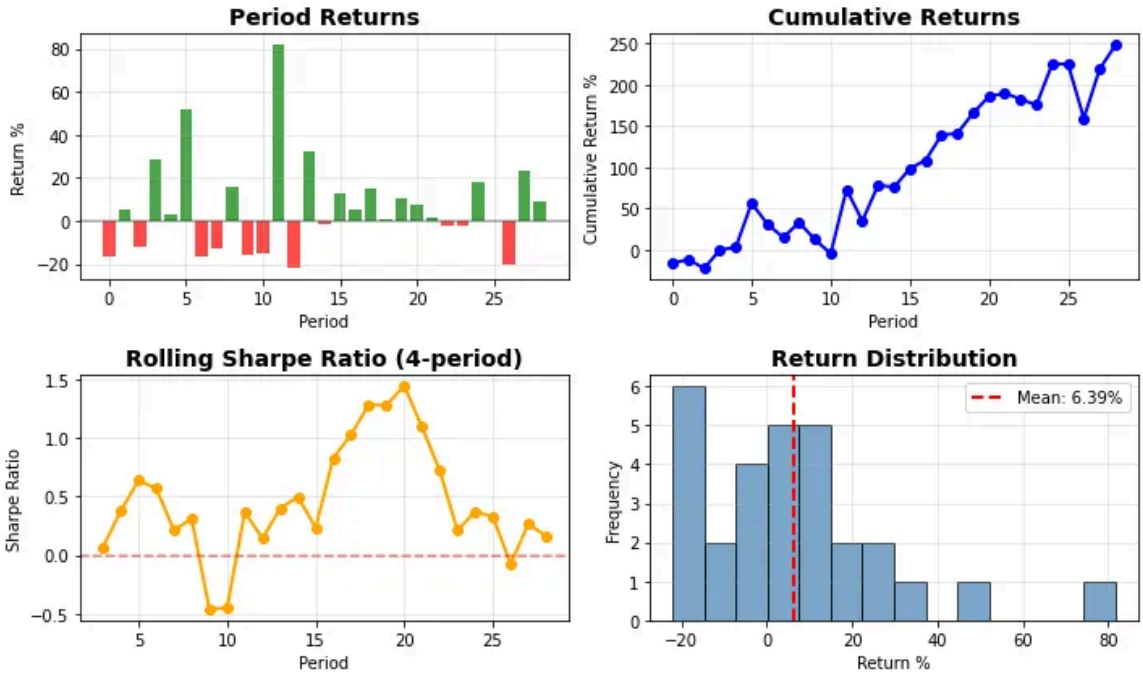
```
# ... (imports from the general rolling backtest script) ...
import dateutil.relativedelta as rd # Already present
import seaborn as sns # Already present
from datetime import datetime # For current date

# Define the strategy for the rolling backtest
strategy = PolynomialChannelBreakoutStrategy

def run_rolling_backtest(
    ticker="BTC-USD",
    start="2018-01-01",
    end="2025-06-24", # Current date in Luxembourg
    window_months=3,
    strategy_params=None
):
    strategy_params = strategy_params or {}
    all_results = []
    start_dt = pd.to_datetime(start)
    end_dt = pd.to_datetime(end)
    current_start = start_dt

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

```
    if current_start >= current_end:
        break

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

# Data download using yfinance, respecting user's preferenc
# Using the saved preference: yfinance download with auto_a
data = yf.download(ticker, start=current_start, end=current

# Apply droplevel if data is a MultiIndex, as per user's pr
if isinstance(data.columns, pd.MultiIndex):
    data = data.droplevel(1, axis=1)

# Check for sufficient data after droplevel for strategy wa
# Requires enough bars for PolynomialChannelIndicator's loo
lookback = strategy_params.get('lookback', PolynomialChanne
atr_period = strategy_params.get('atr_period', PolynomialCh
min_bars_needed = max(lookback, atr_period) + 1 # +1 for cu

if data.empty or len(data) < min_bars_needed:
    print(f"Not enough data for period {current_start.date(
    if current_end == end_dt:
        break
    current_start = current_end
    continue

feed = bt.feeds.PandasData(dataname=data)
cerebro = bt.Cerebro()
cerebro.addstrategy(strategy, **strategy_params)
cerebro.adddata(feed)
cerebro.broker.setcash(100000)
cerebro.broker.setcommission(commission=0.001)
cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

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

all_results.append({
    'start': current_start.date(),
    'end': current_end.date(),
    'return_pct': ret,
    'final_value': final_val,
```

```
        })

        print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")

        if current_end == end_dt:
            break
        current_start = current_end

    return pd.DataFrame(all_results)


def report_stats(df):
    returns = df['return_pct']
    stats = {
        'Mean Return %': np.mean(returns),
        'Median Return %': np.median(returns),
        'Std Dev %': np.std(returns),
        'Min Return %': np.min(returns),
        'Max Return %': np.max(returns),
        'Sharpe Ratio': np.mean(returns) / np.std(returns) if np.st
    }
    print("\n=== ROLLING BACKTEST STATISTICS ===")
    for k, v in stats.items():
        print(f"{k}: {v:.2f}")
    return stats

def plot_four_charts(df, rolling_sharpe_window=4):
    """
    Generates four analytical plots for rolling backtest results.
    """
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12,

    periods = list(range(len(df)))
    returns = df['return_pct']

    # 1. Period Returns (Top Left)
    colors = ['green' if r >= 0 else 'red' for r in returns]
    ax1.bar(periods, returns, color=colors, alpha=0.7)
    ax1.set_title('Period Returns', fontsize=14, fontweight='bold')
    ax1.set_xlabel('Period')
    ax1.set_ylabel('Return %')
    ax1.axhline(y=0, color='black', linestyle='-', alpha=0.3)
    ax1.grid(True, alpha=0.3)
```

```
    # 2. Cumulative Returns (Top Right)
    cumulative_returns = (1 + returns / 100).cumprod() * 100 - 100
    ax2.plot(periods, cumulative_returns, marker='o', linewidth=2,
    ax2.set_title('Cumulative Returns', fontsize=14, fontweight='bo
    ax2.set_xlabel('Period')
    ax2.set_ylabel('Cumulative Return %')
    ax2.grid(True, alpha=0.3)

    # 3. Rolling Sharpe Ratio (Bottom Left)
    rolling_sharpe = returns.rolling(window=rolling_sharpe_window).
        lambda x: x.mean() / x.std() if x.std() > 0 else np.nan, ra
    )
    valid_mask = ~rolling_sharpe.isna()
    valid_periods = [i for i, valid in enumerate(valid_mask) if val
    valid_sharpe = rolling_sharpe[valid_mask]

    ax3.plot(valid_periods, valid_sharpe, marker='o', linewidth=2,
    ax3.axhline(y=0, color='red', linestyle='--', alpha=0.5)
    ax3.set_title(f'Rolling Sharpe Ratio ({rolling_sharpe_window}-p
    ax3.set_xlabel('Period')
    ax3.set_ylabel('Sharpe Ratio')
    ax3.grid(True, alpha=0.3)

    # 4. Return Distribution (Bottom Right)
    bins = min(15, max(5, len(returns)//2))
    ax4.hist(returns, bins=bins, alpha=0.7, color='steelblue', edge
    mean_return = returns.mean()
    ax4.axvline(mean_return, color='red', linestyle='--', linewidth
                label=f'Mean: {mean_return:.2f}%')
    ax4.set_title('Return Distribution', fontsize=14, fontweight='b
    ax4.set_xlabel('Return %')
    ax4.set_ylabel('Frequency')
    ax4.legend()
    ax4.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()

if __name__ == '__main__':
    # Run a single backtest example for illustration
    run_backtest()

    # Then run the rolling backtest for a more robust evaluation
    current_date = datetime.now().date()
```

```
df = run_rolling_backtest(
    ticker="BTC-USD", # Default ticker for article's example
    start="2018-01-01",
    end=current_date, # Use the current date
    window_months=3,
    # strategy_params={ # Example of how to override default pa
    #     'degree': 4,
    #     'channel_width': 2.5,
    #     'lookback': 40,
    #     'trail_atr_mult': 4.0,
    #     'atr_period': 20,
    #     'use_regression_exit': True, # Enable optional exit
    #     'printlog': False,
    # }
)

print("\n=== ROLLING BACKTEST RESULTS ===")
print(df)

stats = report_stats(df)
plot_four_charts(df)
```



5. Conclusion

The `PolynomialChannelBreakoutStrategy` offers a sophisticated approach to trend trading by dynamically adapting to the non-linear nature of market price movements through polynomial regression. Its ability to identify strong breakouts from these adaptive channels, combined with a robust ATR-based trailing stop, provides a comprehensive framework for managing both entry and exit points. The optional regression line exit adds another layer of responsiveness to trend shifts. The rigorous use of both single and rolling backtests is crucial for evaluating such a complex strategy, offering deeper insights into its performance consistency and resilience across various market conditions. Further research could focus on optimizing the polynomial `degree` and `channel_width` parameters, or integrating additional filters to enhance profitability and reduce whipsaws in less trending environments.

Python | Algorithmic Trading | Quantitative Finance | Trading Strategy | Bitcoin

Written by PyQuantLab

655 followers · 6 following

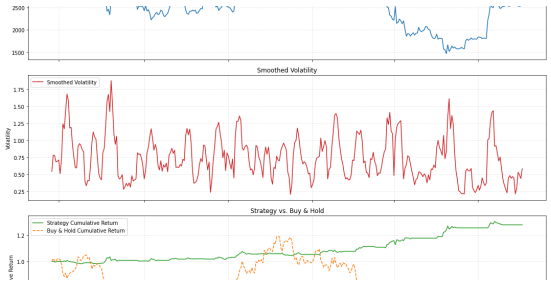
Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

No responses yet

Steven Feng CAI

What are your thoughts?

More from PyQuantLab



PyQuantLab

PyQuantLab

Volatility Clustering Trading Strategy with Python

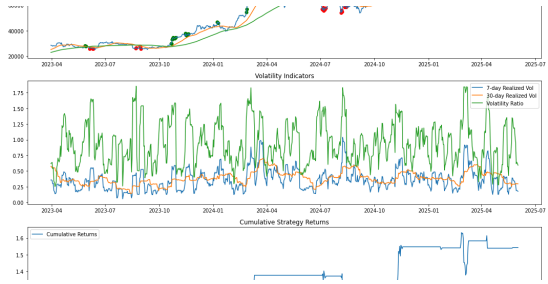
Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python...

Jun 3 32 3

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

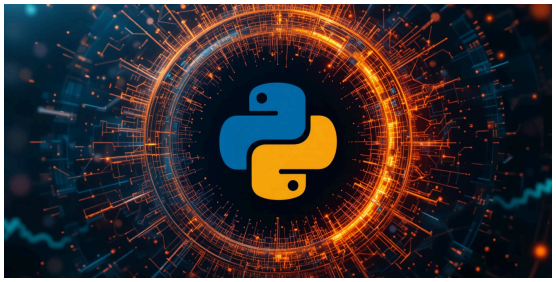
Building an Adaptive Trading Strategy with Backtrader: A...

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

Jun 4 64

See all from PyQuantLab

Recommended from Medium



Unicorn Day

The Quest for the Perfect Trading Score: Turning...

Navigating the financial markets... it feels like being hit by a tsunami of da...

3d ago 43

Swapnilphutane

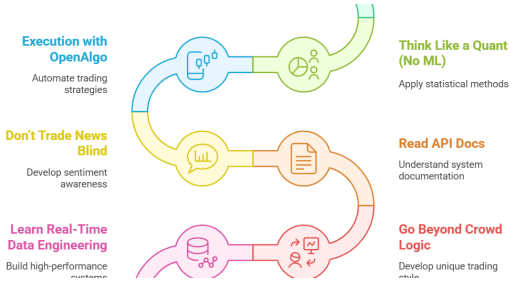
How I Built a Multi-Market Trading Strategy That Pass...

When I first got into trading, I had no plans of building a full-blown system....

6d ago 16 1

MarketMuse

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



Rajandran R (Creator - OpenAlgo)

Algorithmic Trading Roadmap 2025: From Curio...

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

Jun 3 75 3

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

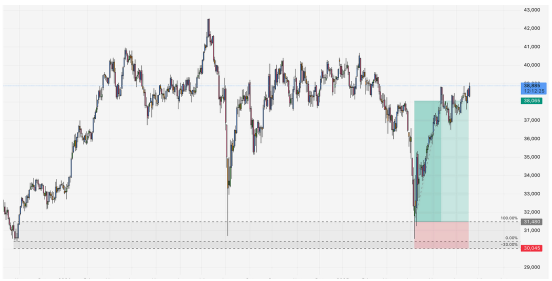
Jun 22 23 3

Remedy

How I Turned My Trading Strategy into a Professional...

By Chinedu Uzochukwu

6d ago 1 1



Candence

Exposing Bernd Skorupinski Strategy: How I Profited ove...

I executed a single Nikkei Futures trade that banked $16,400 with a...

Jun 19 2

See more recommendations