COLLECTED ARTICLES ON TRADING STRATEGIES AND BACKTESTING

Insights, Technigues, and Code for the Modern Trader

Ali AZARY

PyQuantLab

Contents

Preface

Welcome to **Collected Articles on Trading Strategies and Backtesting**.

This book brings together a curated selection of my most practical articles on trading strategy design, performance analysis, and the art of robust backtesting. Each chapter is built from real questions, challenges, and experiments encountered in the world of algorithmic and systematic trading. From developing your first Bitcoin trading model with Python to exploring advanced backtesting with Backtrader, regime detection, and combining classic indicators with machine learning, every article is crafted to help you bridge the gap between theory and real-world application.

You'll find strategies tested on real data, code examples ready for implementation, and practical discussions about performance, risk, and market structure. Whether you're a self-taught trader, a professional quant, or just passionate about the intersection of programming and finance, I hope this collection sparks new ideas and gives you actionable tools for your trading journey.

For those looking to go further, my other books and guides cover technical analysis, advanced quantitative trading, machine learning, and more—designed to help you master every aspect of the modern trading stack:

https://www.pyquantlab.com/#books

- **Advanced Quantitative Trading**: Master powerful Python-based strategies and backtesting techniques for real-world edge.
- **Backtrader Essentials**: Your fast track to building, testing, and optimizing strategies with the Backtrader library.
- **Practical Financial Machine Learning**: A step-by-step guide for applying cutting-edge ML to finance and trading.
- **The Complete Technical Analysis Guide**: Proven, ready-to-use technical trading systems you can start using right away.
- **Desktop App Development with PyQt5**: Build professional financial and trading apps in Python.
- **Moving Average Convergence**: Discover ten crossover and ribbon strategies for consistent results.

Thank you for reading, and happy coding!

Ali Azary

A Bitcoin Trading Strategy with State-Space Models with Python and Backtrader

In the quest for profitable algorithmic trading strategies, identifying and acting on market trends is a cornerstone. While simple moving averages can offer a glimpse, more sophisticated statistical models can provide deeper insights into the underlying dynamics of price movements. This article explores a Python-based Bitcoin trading strategy that leverages state-space models to decompose price data, estimate trends, and generate trading signals, all within the flexible Backtrader framework.

We'll walk through the components of this strategy, from data acquisition and model fitting to signal generation and backtesting, complete with code explanations to help you understand and potentially adapt this approach.

The Core Idea: State-Space Models for Trend Estimation

At the heart of our strategy lies the concept of a **State-Space Model (SSM)**. SSMs provide a powerful way to represent a time series (like Bitcoin prices) through a set of unobserved or latent "state" variables that evolve over time. The observed data is then considered a function of these hidden states.

Specifically, we use the `UnobservedComponents` model from the `statsmodels` library in Python. This allows us to decompose the time series into components like:

- **Trend:** The underlying direction of the price. We'll model this as a "local linear trend" (`'lltrend'`), which means the model estimates both the current level of the trend and its slope (rate of change). Importantly, these components are stochastic, allowing them to change over time to adapt to new market conditions.
- **Seasonality (Optional):** Patterns that repeat over a fixed period (e.g., weekly). While not the focus of this specific version, it can be incorporated.
- **Irregular Component (Noise):** The random fluctuations not captured by other components.

By filtering the observed data through this model, we can obtain estimates of the unobserved trend level and, crucially for our strategy, the **slope of the trend**. A positive slope suggests an uptrend, while a negative slope indicates a downtrend.

Before feeding prices into the model, we apply a **logarithmic transformation** (`np.log(prices)`). This is a common practice in financial modeling as it helps to stabilize the variance of the series and makes exponential growth appear linear, often improving model fit and interpretation.

The Trading Strategy: `StateSpaceTrendVolatility`

Let's dive into the `StateSpaceTrendVolatility` class, the engine of our trading logic, built using Backtrader.

Python

```
import backtrader as bt
import numpy as np
import statsmodels.api as sm
import pandas as pd
from datetime import datetime
import warnings

class StateSpaceTrendVolatility(bt.Strategy):
    params = (
        ('model_update_period', 60),    # Default: How often to re-fit the
model (bars)
        ('lookback_period', 365),       # Default: Data window for model
fitting (bars)
        ('trend_slope_threshold_buy', 0.0005), # Default: Min positive slope
for buy
        ('trend_slope_threshold_sell', -0.0005),# Default: Max negative slope
for sell
    )

    def __init__(self):
        self.btc_close = self.datas[0].close
        self.order = None
        self.model_fit_day = -self.p.model_update_period # Ensure model fits
early
        self.trend_level = None
        self.trend_slope = None
        self.log_price_at_fit = None # For debugging model fit

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()} {txt}')

    # notify_order and notify_trade methods for logging (standard Backtrader)
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            # self.bar_executed = len(self) # Optional: track bar of
```

```
execution
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}')

    # ... (fit_state_space_model and next methods discussed below) ...
```

Strategy Parameters:

- `model_update_period`: Defines how often (in trading bars) the state-space model is re-estimated using fresh data.
- `lookback_period`: Specifies the window of past data (in bars) used for each model fitting.
- `trend_slope_threshold_buy`: The minimum positive slope value that triggers a buy signal.
- `trend_slope_threshold_sell`: The maximum (most negative) slope value that triggers a sell/short signal.

These parameters are crucial for tuning the strategy's responsiveness and sensitivity.

Fitting the State-Space Model (`fit_state_space_model`)

This method is where the statistical heavy lifting occurs. It's called periodically as defined by `model_update_period`.

Python

```
    def fit_state_space_model(self):
        current_bar_index = len(self)
        if current_bar_index < self.p.lookback_period:
            self.log(f"Not enough data to fit model yet. Have
{current_bar_index}, need {self.p.lookback_period}.")
            return False

        # Collect historical close prices for the lookback period
        dates = [bt.num2date(self.datas[0].datetime[-i]) for i in
range(self.p.lookback_period - 1, -1, -1)]
        closes = [self.datas[0].close[-i] for i in
range(self.p.lookback_period - 1, -1, -1)]

        ts_data = pd.Series(closes, index=pd.to_datetime(dates)).dropna()
        self.log(f"Fitting SSM: Data from {ts_data.index[0].date()} to
{ts_data.index[-1].date()}, {len(ts_data)} points.")
        if ts_data.empty or (ts_data <= 0).any():
            self.log("ERROR: Data for model is empty or contains non-positive
values.")
            return False

        try:
            log_ts_data = np.log(ts_data)
            self.log_price_at_fit = log_ts_data.iloc[-1]

            if log_ts_data.isnull().any() or np.isinf(log_ts_data).any():
                self.log("ERROR: log_ts_data contains NaN or Inf values!")
                return False

            # Define the Unobserved Components model: Local Linear Trend
            model = sm.tsa.UnobservedComponents(
                log_ts_data,
                level='lltrend',
                # irregular=True # Optional: add if residuals are white noise
            )

            with warnings.catch_warnings(): # Suppress common fitting
warnings
                warnings.simplefilter("ignore")
                result = model.fit(method='lbfgs', disp=False, maxiter=500)

            self.log(f"DEBUG: Model converged:
{result.mle_retvals['converged']}")
            self.log(f"DEBUG: Model log-likelihood: {result.llf:.4f}")

            filtered_state = result.filter_results.filtered_state

            if filtered_state.shape[1] >= 2: # Expecting at least level and
slope
                self.trend_level = filtered_state[-1, 0]  # Last log-level
                self.trend_slope = filtered_state[-1, 1]  # Last log-slope
                self.log(f"Model Fit: Actual Last Log-Price:
{self.log_price_at_fit:.4f}, Estimated Log-Trend Level:
{self.trend_level:.4f}, Slope: {self.trend_slope:.4f}")

                # Sanity check for model fit
                if np.isclose(self.trend_level, 0.0, atol=1e-3) and not
np.isclose(self.log_price_at_fit, 0.0, atol=1e-3):
                    self.log("WARNING: Estimated log-trend level is
suspiciously close to zero. Check model fit quality.")

                if np.isnan(self.trend_level) or np.isnan(self.trend_slope):
                    self.log("ERROR: Trend level or slope is NaN after model 
fit.")
                    return False
                return True
            else:
                self.log(f"ERROR: Filtered state unexpected shape: 
{filtered_state.shape}")
                return False
        except Exception as e:
            self.log(f"ERROR fitting state-space model: {e}")
            import traceback
            self.log(traceback.format_exc())
            return False
```

Key steps in `fit_state_space_model`:

1. **Data Collection:** Gathers `lookback_period` worth of closing prices from the Backtrader data feed.
2. **Preprocessing:** Converts to a Pandas Series, drops NaNs, and checks for non-positive values before log transformation.
3. **Log Transformation:** Applies `np.log()` to the price data.
4. **Model Definition:** Initializes `sm.tsa.UnobservedComponents` with `level='lltrend'`.
5. **Model Fitting:** Uses `model.fit()` (with L-BFGS optimizer by default here) to estimate the model parameters and unobserved states. Warnings common during optimization are suppressed for cleaner output.
6. **State Extraction:** Retrieves the `filtered_state` (estimates based on data up to the current point). For a 'lltrend' model, the last values of the first state component (level) and the second state component (slope) are extracted.
7. **Debugging & Sanity Checks:** Includes logs for convergence, log-likelihood, and a crucial check comparing the estimated log-trend level to the actual last log price used in fitting. This helps flag obviously poor model fits.

Generating Signals (next method)

The `next()` method is called for each new bar of data. It decides whether to refit the model and then checks for trading signals.

Python

```
    def next(self):
        # Periodically re-fit the model
        if len(self) >= self.model_fit_day + self.p.model_update_period and \
           len(self) >= self.p.lookback_period:
            if self.fit_state_space_model():
                self.model_fit_day = len(self) # Update last fit day
            else:
                self.log("Model fitting failed. Holding off on trading.")
                # Consider closing open positions if model becomes unreliable
                return

        if self.order: # An order is pending, do nothing
            return

        if self.trend_level is None or self.trend_slope is None:
            # Model not ready or failed, do nothing
            return

        current_position_size = self.getposition().size

        # Buy Signal Logic
        if self.trend_slope > self.p.trend_slope_threshold_buy:
            if current_position_size == 0: # Not in market
                self.log(f'BUY CREATE @ {self.btc_close[0]:.2f}, Slope:
{self.trend_slope:.5f}, Est.LogLvl: {self.trend_level:.4f}')
                self.order = self.buy()
            elif current_position_size < 0: # Currently short
                self.log(f'COVER SHORT & CONSIDER BUY @
{self.btc_close[0]:.2f}, Slope: {self.trend_slope:.5f}')
                self.order = self.close() # Close short position
                                          # Buy will be re-evaluated on next
bar if signal persists

        # Sell Signal Logic
        elif self.trend_slope < self.p.trend_slope_threshold_sell:
            if current_position_size == 0: # Not in market
                self.log(f'SELL CREATE (SHORT) @ {self.btc_close[0]:.2f},
Slope: {self.trend_slope:.5f}, Est.LogLvl: {self.trend_level:.4f}')
                self.order = self.sell()
            elif current_position_size > 0: # Currently long
                self.log(f'LIQUIDATE LONG & CONSIDER SHORT @
{self.btc_close[0]:.2f}, Slope: {self.trend_slope:.5f}')
                self.order = self.close() # Close long position
                                          # Short will be re-evaluated on
next bar
        # else: # Optional: Neutral zone - close positions if slope flattens
            # if current_position_size != 0:
            #     self.order = self.close()
```

Signal logic:

- If the `trend_slope` exceeds `trend_slope_threshold_buy`, a buy order is placed (if not already long). If short, the short position is closed.
- If the `trend_slope` falls below `trend_slope_threshold_sell`, a sell (short) order is placed (if not already short). If long, the long position is closed.

Setting Up the Backtest

The `if __name__ == '__main__':` block orchestrates the backtest:

Python

```
from curl_cffi import requests # For robust yfinance downloads
session = requests.Session(impersonate="chrome") # Mimic browser
from sys import exit # For clean exit on data errors
import matplotlib.pyplot as plt
# %matplotlib qt5 # For interactive plots in IPython/Spyder, run this in your
console

if __name__ == '__main__':
    cerebro = bt.Cerebro()

    # --- Data Feed Section ---
    try:
        data_df = yf.download('BTC-USD', period='3y', interval='1d',
session=session)
    except Exception as e_yf_sess: # Fallback if session download fails
        print(f"Failed to download data using yfinance with custom session:
{e_yf_sess}")
        print("Attempting yfinance download without custom session...")
        try:
            data_df = yf.download('BTC-USD', period='3y', interval='1d')
        except Exception as e_yf:
            print(f"Failed to download data using standard yfinance: {e_yf}")
            exit() # Exit if data can't be fetched

    if data_df.empty:
        print("Could not download BTC-USD data. Exiting.")
        exit()

    print("DEBUG: Columns after yf.download:", data_df.columns)
    if isinstance(data_df.columns, pd.MultiIndex):
        print("DEBUG: DataFrame has MultiIndex columns. Assuming ('Field',
'') structure and getting level 0.")
        # This handles a common yfinance MultiIndex like ('Open', ''),
('Close', '')
        if all(col_val == '' for col_val in
data_df.columns.get_level_values(1)):
            data_df.columns = data_df.columns.get_level_values(0)
        else: # Fallback for other MultiIndex structures - adjust as needed
            print(f"DEBUG: MultiIndex structure is not ('Field', '').
Attempting droplevel(0) or check structure manually. Current:
{data_df.columns}")
            # If structure is ('Ticker', 'Field'), droplevel(0) might work
            # For safety, you might need to inspect and flatten based on
```

```
specific structure
            # data_df.columns = data_df.columns.droplevel(0) # Example

    rename_map = { # Standardize to lowercase for Backtrader
        'Open': 'open', 'High': 'high', 'Low': 'low', 'Close': 'close',
        'Adj Close': 'adjclose', 'Volume': 'volume'
    }
    data_df.rename(columns=rename_map, inplace=True)
    for col in ['open', 'high', 'low', 'close', 'volume']: # Check essential
columns
        if col not in data_df.columns:
            raise ValueError(f"Missing required column '{col}'. Columns are:
{data_df.columns}")
    data_df['openinterest'] = 0 # Required by Backtrader
    data_df.index = pd.to_datetime(data_df.index)

    data = bt.feeds.PandasData(dataname=data_df)
    cerebro.adddata(data)

    # Add Strategy with tuned parameters (example values)
    cerebro.addstrategy(StateSpaceTrendVolatility,
                        model_update_period=90,  # Refit every ~3 months
                        lookback_period=365,     # Use 1 year of data for
fitting
                        trend_slope_threshold_buy=0.001, # Stricter buy
threshold
                        trend_slope_threshold_sell=-0.001)# Stricter sell
threshold

    # Broker, Sizer, Analyzers
    cerebro.broker.setcash(100000.0)
    cerebro.broker.setcommission(commission=0.001) # 0.1%
    cerebro.addsizer(bt.sizers.PercentSizer, percents=90) # Use 90% of cash

    cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe_ratio',
timeframe=bt.TimeFrame.Days, annualize=True, riskfreerate=0.0)
    cerebro.addanalyzer(bt.analyzers.AnnualReturn, _name='annual_return')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trade_analyzer')

    print('Starting Portfolio Value: %.2f' % cerebro.broker.getvalue())
    results = cerebro.run() # Optimization can be run with
cerebro.optstrategy
    print('Final Portfolio Value: %.2f' % cerebro.broker.getvalue())

    # Print Analysis (with safer dictionary access)
    strat = results[0] # Assuming single strategy run
    print(f"\n--- Strategy Analysis ---")
    # ... (Detailed analyzer printout as in your provided script) ...
    # (Example for Sharpe)
    sharpe_analysis = strat.analyzers.sharpe_ratio.get_analysis()
    print(f"Sharpe Ratio: {sharpe_analysis.get('sharperatio', 'N/A'):.2f}")


    # Plotting
    plt.rcParams['figure.figsize'] = [12, 7] # Adjust figure size
    # For Spyder/IPython with Qt5 backend, ensure '%matplotlib qt5' was run
in console
    cerebro.plot(iplot=False, style='candlestick')
    plt.tight_layout()
    # plt.show() # May be needed depending on environment
```

Data Handling Highlights:

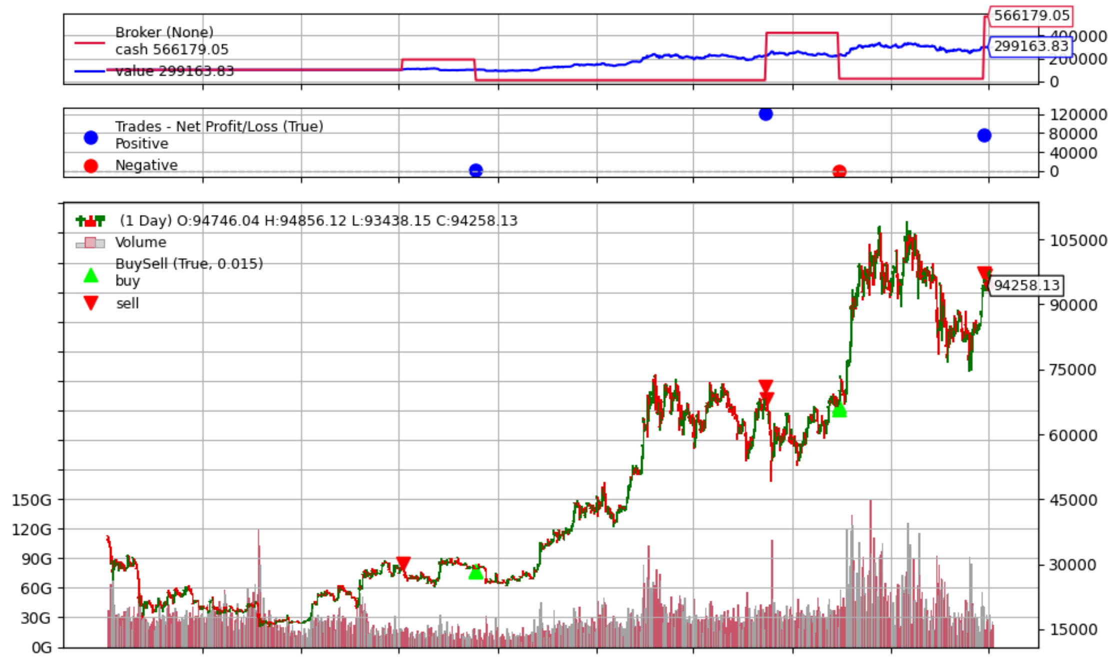
- **`curl_cffi` with `Session(impersonate="chrome")`:** This makes `yfinance` downloads more robust by mimicking a web browser, which can help bypass potential blocks from Yahoo Finance.
- **Robust Column Handling:** The code now includes more careful checks and attempts to flatten potential MultiIndex columns returned by `yfinance`, a common source of issues. It then renames columns to lowercase as Backtrader expects and adds the mandatory `openinterest` column.
- **Strategy Parameter Overrides:** Notice how parameters like `model_update_period=90`, `lookback_period=365`, etc., are passed when adding the strategy. This allows for easy tuning without modifying the class defaults.

Interpreting Results & Plotting:

The script includes standard Backtrader analyzers to evaluate performance: Sharpe Ratio, Annual Return, Drawdown, and Trade Analyzer. cerebro.plot(iplot=False) will use your Matplotlib backend (like Qt5 if you've set %matplotlib qt5 in an interactive console) to display the results, including price, trades, and portfolio value.

Stochastic Volatility Aspect

The original prompt for this type of strategy often mentions "stochastic volatility." In this implementation, while we don't have a dedicated GARCH component within the UnobservedComponents model for the observation error's variance, the level='lltrend' specification itself means the level and the trend (slope) are stochastic. Their variances are estimated by the model. This means the trend itself can become more or less variable, indirectly reflecting changes in market volatility through its own dynamics. A full GARCH-in-SSM is a more advanced topic, often requiring custom state-space model definitions.



Pasted image 20250506162131.png

Pasted image 20250506162131.png

```
--- Strategy Analysis ---
Sharpe Ratio: 0.99
Annual Returns:
  2022: 0.00%
  2023: 42.05%
  2024: 108.99%
  2025: 0.77%
Max Drawdown: 26.29%
Max Money Drawdown: 87883.43
Total Trades: 5
Winning Trades: 3
Losing Trades: 1
Win Rate: 60.00%
Average Winning Trade: 66385.05
Average Losing Trade: -1016.25
Profit Factor: 195.97
```

Key Considerations and Path Forward

1. **Model Health is Paramount:** The success of this strategy hinges on the state-space model accurately estimating the trend level and slope. The debugging logs (`Actual Last Log-Price` vs. `Estimated Log-Trend Level`) are vital. If the model fit is poor, the signals will be unreliable.
2. **Parameter Tuning:**

  - `lookback_period`: Shorter periods make the model more reactive but potentially noisier. Longer periods offer more stability but slower adaptation.
  - `model_update_period`: Balances adaptiveness with computational cost.
  - `trend_slope_threshold_buy/sell`: These are highly sensitive. Small changes can drastically alter the number of trades and performance. Systematic optimization (e.g., using `cerebro.optstrategy`) is recommended, followed by out-of-sample validation to avoid overfitting.
3. **Risk Management:** The current strategy lacks explicit stop-losses. Adding risk management rules is crucial for any real-world application.
4. **Computational Cost:** Fitting state-space models can be computationally intensive, especially with long lookback periods or frequent updates.
5. **Overfitting:** Be extremely cautious when interpreting backtest results, especially after optimization. What works on historical data may not work in the future. Walk-forward optimization and testing on truly unseen data are essential.

Conclusion

This state-space modeling approach offers a sophisticated way to define and trade market trends. By decomposing prices into unobserved components, we aim to get a clearer picture of the underlying market direction. Backtrader provides the environment to rigorously test such ideas. However, success depends on a well-fitting statistical model, careful parameter tuning, robust data handling, and sound risk management principles. This framework serves as a solid starting point for further exploration and refinement in your algorithmic trading journey.

A Guide to Live Trading with Backtrader on Alpaca

`alpaca-backtrader-api` is a dedicated Python library designed to integrate Alpaca's trading API directly within the `backtrader` framework. It simplifies the process for developers and quants, allowing them to leverage Alpaca's REST and streaming interfaces for both fetching market data and executing trades, all while using `backtrader`'s familiar structure for strategy definition, indicator calculation, and analysis.

I will explore how to set up data feeds, manage authentication, switch between paper and live trading, and handle specific considerations like multiple data streams.

Prerequisites

Before integrating Alpaca with Backtrader using this library, ensure you have:

1. **Python:** A working Python installation. The underlying `alpaca-trade-api` often requires Python 3.7 or higher, even though the README mentions 3.5+. It's safest to use Python 3.7+.
2. **`backtrader`:** The core backtesting library installed (`pip install backtrader`).

3. **Alpaca Account:** An account registered at alpaca.markets. You can start with a paper trading account for risk-free testing.
4. **Alpaca API Keys:** You'll need an **API Key ID** and a **Secret Key** obtained from your Alpaca account dashboard. Keep these secure!

Installation

Installing the library is straightforward using pip:

Bash

```
pip install alpaca-backtrader-api
```

This command downloads and installs the library and its necessary dependencies, including the base `alpaca-trade-api`.

Core Concept: The `AlpacaStore` Bridge

The central component provided by this library is `alpaca_backtrader_api.AlpacaStore`. This class acts as the main interface to the Alpaca API within `backtrader`.

- **Connection:** It handles the connection to Alpaca using your API keys.
- **Broker Access:** It provides a factory method (`.getbroker()`) to retrieve an `AlpacaBroker` instance configured for your account.
- **Data Access:** It provides a factory method (`.getdata()`) to retrieve an `AlpacaData` instance for fetching market data feeds.

You typically instantiate `AlpacaStore` once and use it to configure `backtrader`'s Cerebro engine.

Authentication and Configuration

Connecting to Alpaca requires authenticating with your API keys.

1. **Obtain Keys:** Log in to your Alpaca account dashboard and navigate to the API Keys section to generate or view your Key ID and Secret Key.

2. **Configure `AlpacaStore`:** Pass the keys directly to the `AlpacaStore` constructor:

   Python

   ```
   import alpaca_backtrader_api as AlpacaApi
   import os # Recommended for secure key handling

   # --- Method 1: Using Environment Variables (Recommended) ---
   ALPACA_API_KEY = os.environ.get('ALPACA_API_KEY')
   ALPACA_SECRET_KEY = os.environ.get('ALPACA_SECRET_KEY')

   # --- Method 2: Hardcoding (NOT Recommended for production/shared code)
   ```

```
---
# ALPACA_API_KEY = "YOUR_API_KEY_ID"
# ALPACA_SECRET_KEY = "YOUR_SECRET_KEY"

if not ALPACA_API_KEY or not ALPACA_SECRET_KEY:
    raise ValueError("Alpaca API Key/Secret Key not found.")

# Configure Paper/Live Mode (see next section)
ALPACA_PAPER = True # Start with True for paper trading

store = AlpacaApi.AlpacaStore(
    key_id=ALPACA_API_KEY,
    secret_key=ALPACA_SECRET_KEY,
    paper=ALPACA_PAPER
)
```

Security Best Practice: Avoid hardcoding your API keys directly in your script. Use environment variables, a configuration file, or a secrets management system to keep them secure, especially if you share your code or use version control (like Git).

Paper Trading vs. Live Trading

The library makes it easy to switch between Alpaca's paper trading environment (simulated trading with live data) and the live market. This is controlled by the `paper` argument in the `AlpacaStore` constructor:

- `paper=True`: Connects to the Alpaca paper trading environment. **Always start with this.**
- `paper=False`: Connects to the live trading environment (requires a funded live account). This is the default if the parameter is omitted.

Python

```
# For Paper Trading
store = AlpacaApi.AlpacaStore(key_id=..., secret_key=..., paper=True)

# For Live Trading (Use with extreme caution!)
store = AlpacaApi.AlpacaStore(key_id=..., secret_key=..., paper=False)
```

Getting Market Data (`AlpacaData`)

You can fetch both historical and live market data using the data factory obtained from the store.

Python

```
# Get the data factory from the store
DataFactory = store.getdata
# Alternatively: from alpaca_backtrader_api import AlpacaData
# --- Historical Data for Backtesting ---
data_hist = DataFactory(
    dataname='SPY', # Ticker symbol
    timeframe=bt.TimeFrame.Days,
    fromdate=datetime(2020, 1, 1),
    todate=datetime(2021, 12, 31),
    historical=True # Key parameter for historical data download
    # compression=1 # Optional: specify compression if needed
)

# --- Live Data for Paper/Live Trading ---
data_live = DataFactory(
    dataname='AAPL', # Ticker symbol
    timeframe=bt.TimeFrame.Minutes,
    compression=1,
    historical=False, # Key parameter for live data stream
    # qcheck=0.5 # Optional: data check frequency
    # use_basic_auth=True # Optional: might be needed
)
```

Key Data Parameters:

- `dataname`: The ticker symbol (e.g., ‘AAPL’, ‘SPY’, ‘BTCUSD’ for crypto - check Alpaca documentation for specific symbol formats).
- `timeframe`: Use `backtrader.TimeFrame` enums (e.g., `Days`, `Minutes`, `Seconds`).
- `compression`: The number associated with the timeframe (e.g., `5` for 5-minute bars).
- `historical`: Set to `True` to download historical bars for backtesting. Set to `False` to connect to the live data stream (requires appropriate Alpaca data subscription - Polygon data may be needed for non-IEX real-time US stock data).
- `fromdate`, `todate`: Used only when `historical=True`.

Note on Real-Time Data: Accessing real-time data (especially for US stocks beyond IEX) often requires specific data subscriptions via Alpaca (e.g., Polygon). Ensure your Alpaca account has the necessary permissions for the data you are requesting when `historical=False`.

Executing Trades (`AlpacaBroker`)

To execute trades in paper or live mode, you need to get the `AlpacaBroker` instance from the store and add it to Cerebro.

Python

```
# Get the broker instance from the store
broker = store.getbroker()
# Alternatively: from alpaca_backtrader_api import AlpacaBroker
# Add the broker to Cerebro
cerebro.setbroker(broker)
```

Once the Alpaca broker is set, you use standard `backtrader` methods within your strategy to place orders:

Python

```
# Inside your Strategy's next() method:
self.buy(size=10)
self.sell(size=5)
self.close() # Closes the current position for the data feed
# Order methods like order_target_size, order_target_percent also work.
```

The `AlpacaBroker` handles translating these commands into the appropriate Alpaca API calls.

Putting It Together: Code Examples

Let's implement a simple SMA Crossover strategy with the live trading example.

Backtesting with Alpaca Historical Data

This setup uses historical data from Alpaca but runs the backtest using `backtrader`'s internal simulated broker.

Python

```
import alpaca_backtrader_api as AlpacaApi
import backtrader as bt
from datetime import datetime
import os
import math # For sizing example

# --- Strategy Class (SmaCrossStrategy from previous example) ---
class SmaCrossStrategy(bt.Strategy):
    params = (('fast_ma', 10), ('slow_ma', 50), ('order_percentage', 0.95),
('ticker', 'SPY'))
    def __init__(self):
        self.sma_fast = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.fast_ma)
        self.sma_slow = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.slow_ma)
        self.crossover = bt.indicators.CrossOver(self.sma_fast,
self.sma_slow)
        self.order = None
        self.holding = 0
    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0) # Use date for daily
backtest
```

```
        print(f'{dt.isoformat()} - {txt}')
    def notify_order(self, order): # Simplified for backtest example
        if order.status in [order.Submitted, order.Accepted]: return
        if order.status in [order.Completed]:
            if order.isbuy(): self.log(f'BUY EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}')
            elif order.issell(): self.log(f'SELL EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}')
            self.holding += order.executed.size
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
self.log('ORDER Canceled/Margin/Rejected')
        self.order = None
    def notify_trade(self, trade):
         if not trade.isclosed: return
         self.log(f'TRADE PROFIT - GROSS: {trade.pnl:.2f}, NET:
{trade.pnlcomm:.2f}')
    def _get_size(self, price): # Simplified for backtest example
        cash = self.broker.get_cash()
        if cash <= 0: return 0
        size = (cash * self.params.order_percentage) / price
        return math.floor(size)
    def next(self):
        if self.order: return
        if self.holding == 0:
            if self.crossover > 0:
                size = self._get_size(self.data.close[0])
                if size > 0:
                    self.log(f'BUY SIGNAL - Close: {self.data.close[0]:.2f},
Creating Order Size: {size}')
                    self.order = self.buy(size=size)
        elif self.holding > 0:
            if self.crossover < 0:
                self.log(f'SELL SIGNAL (Exit) - Close:
{self.data.close[0]:.2f}')
                self.order = self.close()
# --- End Strategy Class ---

# --- Load Credentials ---
ALPACA_API_KEY = os.environ.get('ALPACA_API_KEY')
ALPACA_SECRET_KEY = os.environ.get('ALPACA_SECRET_KEY')
if not ALPACA_API_KEY or not ALPACA_SECRET_KEY: raise ValueError("Keys
required even for historical data.")
# --- End Credentials ---

if __name__ == '__main__':
    cerebro = bt.Cerebro()

    store = AlpacaApi.AlpacaStore(
        key_id=ALPACA_API_KEY,
        secret_key=ALPACA_SECRET_KEY,
        paper=True # Doesn't matter much for historical, but set True/False
    )

    DataFactory = store.getdata
    data0 = DataFactory(
        dataname='SPY', # Example Stock
        timeframe=bt.TimeFrame.Days,
        fromdate=datetime(2020, 1, 1),
        todate=datetime(2022, 12, 31),
        historical=True # <<< Get historical data
    )
    cerebro.adddata(data0)

    # Add strategy
    cerebro.addstrategy(SmaCrossStrategy)

    # Use Backtrader's simulated broker for backtesting
    cerebro.broker.setcash(10000.0)
    cerebro.broker.setcommission(commission=0.001) # Example commission

    print('Starting Backtest Portfolio Value: %.2f' %
cerebro.broker.getvalue())
    cerebro.run()
    print('Final Backtest Portfolio Value: %.2f' % cerebro.broker.getvalue())
    cerebro.plot()
```

Paper/Live Trading with Alpaca Broker & Data

This setup connects to Alpaca for live data and uses the Alpaca broker for paper or live order execution.

Python

```
import alpaca_backtrader_api as AlpacaApi
import backtrader as bt
from datetime import datetime
import os
import math


# --- Strategy Class (Use the more detailed SmaCrossStrategy from the start
of this example) ---
class SmaCrossStrategy(bt.Strategy):
    params = (('fast_ma', 10), ('slow_ma', 50), ('order_percentage', 0.95),
('ticker', 'AAPL'))
    def __init__(self):
        self.sma_fast = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.fast_ma)
        self.sma_slow = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.params.slow_ma)
        self.crossover = bt.indicators.CrossOver(self.sma_fast,
self.sma_slow)
        self.order = None
        self.holding = 0
    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.datetime(0) # Use datetime for live
        print(f'{dt.isoformat()} - {txt}')
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
self.log(f'ORDER {order.ordtypename()} ACCEPTED/SUBMITTED - Ref:
{order.ref}'); self.order = order; return
        if order.status in [order.Completed]:
            if order.isbuy(): self.log(f'BUY EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}, Cost:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            elif order.issell(): self.log(f'SELL EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}, Cost:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            self.holding += order.executed.size
            self.log(f'Current Holding: {self.holding}')
            self.order = None
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
self.log(f'ORDER Canceled/Margin/Rejected - Ref: {order.ref}, Status:
{order.getstatusname()}'); self.order = None
    def notify_trade(self, trade):
         if not trade.isclosed: return
         self.log(f'TRADE PROFIT - GROSS: {trade.pnl:.2f}, NET:
{trade.pnlcomm:.2f}')
    def _get_size(self, price):
        cash = self.broker.get_cash()
        if cash <= 0: return 0
        size = (cash * self.params.order_percentage) / price
        return math.floor(size) # Round down for shares
    def next(self):
        # Add check for live data if needed: if not
hasattr(self.data.datetime, 'datetime'): return
        if self.order: return
        if self.holding == 0:
            if self.crossover > 0:
                size = self._get_size(self.data.close[0])
                if size > 0:
                    self.log(f'BUY SIGNAL - Close: {self.data.close[0]:.2f},
Creating Order Size: {size}')
                    self.order = self.buy(size=size)
        elif self.holding > 0:
            if self.crossover < 0:
                self.log(f'SELL SIGNAL (Exit) - Close:
{self.data.close[0]:.2f}')
```

```
              self.order = self.close()
# --- End Strategy Class ---

# --- Load Credentials ---
ALPACA_API_KEY = os.environ.get('ALPACA_API_KEY')
ALPACA_SECRET_KEY = os.environ.get('ALPACA_SECRET_KEY')
ALPACA_PAPER = True # <<< Set to True for Paper Trading, False for Live
if not ALPACA_API_KEY or not ALPACA_SECRET_KEY: raise ValueError("Keys
required.")
# --- End Credentials ---

if __name__ == '__main__':
    cerebro = bt.Cerebro()

    store = AlpacaApi.AlpacaStore(
        key_id=ALPACA_API_KEY,
        secret_key=ALPACA_SECRET_KEY,
        paper=ALPACA_PAPER
    )

    DataFactory = store.getdata
    data0 = DataFactory(
        dataname='AAPL',
        timeframe=bt.TimeFrame.Minutes,
        compression=1,
        historical=False # <<< Get live data stream
    )
    cerebro.adddata(data0)

    # Set the Alpaca Broker for Paper or Live execution
    broker = store.getbroker()
    cerebro.setbroker(broker)

    cerebro.addstrategy(SmaCrossStrategy)

    print(f"Starting Alpaca Strategy (Paper Trading: {ALPACA_PAPER})...")
    start_value = broker.get_value() # Get initial portfolio value from
broker
    print('Starting Portfolio Value: %.2f' % start_value)
    try:
        cerebro.run() # Runs indefinitely on live data
    except KeyboardInterrupt:
        print("Manually Interrupted.")
    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        print("Strategy Stopped.")
        end_value = broker.get_value()
        print('Ending Portfolio Value: %.2f' % end_value)
        # cerebro.plot() # Plotting might not work as expected with
indefinite live run
```

Conclusion

The `alpaca-backtrader-api` library provides a vital and relatively straightforward way for Python developers to integrate the powerful `backtrader` strategy development and testing framework with Alpaca's modern, API-driven brokerage services. By understanding how to configure the `AlpacaStore`, differentiate between historical and live data feeds, and set the appropriate broker for paper or live trading, users can streamline their workflow from backtesting to execution. As always, thorough testing in the paper trading environment is essential before deploying any strategy with real capital.

Disclaimer: This guide is for educational purposes only and does not constitute financial advice. Algorithmic trading involves significant risk. Ensure you understand the Alpaca API, the `alpaca-backtrader-api` library, and the inherent risks of trading before deploying any capital. Verify information with the official documentation.

A Simple Long Momentum Portfolio Strategy 15% to 2400% Annual Returns!

Momentum strategies have garnered much attention in the cryptocurrency space, where strong trends can produce significant gains. However, these strategies can be vulnerable to sudden downturns. One simple way to mitigate risk is by implementing a stop loss. I am using Binance USDC pairs and download the price data from yahoo finance using yfinance Python library. I then rank them using a risk-adjusted return (30-day return divided by the corresponding volatility) to select the top 10 trending assets. I hold these for one week and, if the portfolio's return for that week falls below –5%, I liquidate the portfolio so the loss is capped at –5%.

For performance analysis, I am calculating the cumulative weekly returns for each year from 2020 to 2024. Each year's returns are aligned by week number (Week 1, Week 2, ...) and overlaid on a single chart to facilitate comparison across years.

It is worth noting that the strategy can be used for any asset class like stocks, currencies, or bonds for that matter. I usually use cryptocurrencies because I am mainly trading cryptocurrencies myself.

Below is the complete Python code so you can try it yourself:

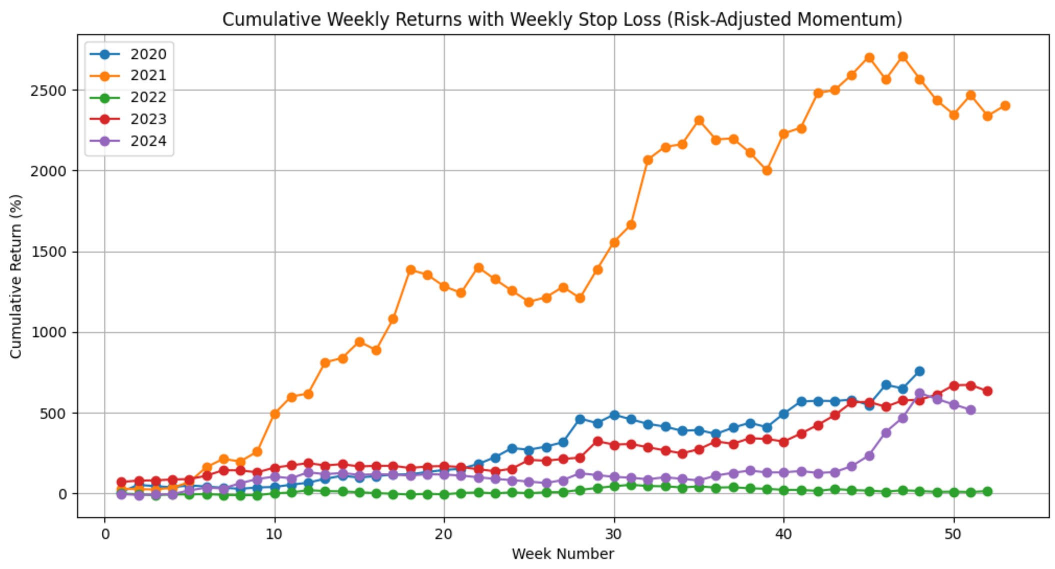
```
import requests
import pandas as pd
import yfinance as yf
```

```
from binance_pairs import BinanceSymbols
import matplotlib.pyplot as plt

# Get Binance USDC pairs and convert them to ticker symbols in the format
'SYMBOL-USD'
pairs = BinanceSymbols().get_usdc_pairs()
tickers = [pair[:-4] + '-USD' for pair in pairs]

# Download data for the entire period (2020-01-01 to 2024-12-31)
data = yf.download(tickers, start='2020-01-01', end='2024-12-31')
# Extract the 'Close' prices and sort by date
data = data.xs('Close', axis=1, level=0).sort_index()

# Drop tickers with any missing data to ensure consistency across years
data = data.dropna(axis=1)

# Define the weekly stop loss threshold (if weekly return is below -5%, cap
it at -5%)
stop_loss_threshold = -0.05

# List of years for analysis
years = [2020, 2021, 2022, 2023, 2024]
yearly_results = {}

# For each year, iterate over the data in 7-day steps to compute weekly
returns
for year in years:
    start_date = pd.to_datetime(f"{year}-01-01")
    end_date = pd.to_datetime(f"{year}-12-31")
    results = []

    # Loop over the data, starting at index 30 to have a 30-day lookback
period
    for i in range(30, len(data) - 7, 7):
        current_date = data.index[i]
        # Process only weeks within the current year
        if current_date < start_date or current_date > end_date:
            continue

        # Calculate the "monthly" return using a 30-day lookback
        monthly_returns = (data.iloc[i] / data.iloc[i - 30]) - 1

        # Calculate volatility over the past 30 days for each asset
        volatilities = {}
        for ticker in tickers:
            try:
                daily_returns = data[ticker].iloc[i-
30:i].pct_change().dropna()
```

```
                vol = daily_returns.std()
                volatilities[ticker] = vol
            except Exception:
                continue
        vol_series = pd.Series(volatilities)
        if vol_series.empty:
            continue

        # Compute the risk-adjusted metric: monthly return divided by
volatility
        risk_adjusted = monthly_returns[vol_series.index] / (vol_series + 1e-
8)

        # Select the top 10 tickers based on the risk-adjusted metric
        top10_tickers = risk_adjusted.nlargest(10).index

        # Calculate the next week's (7-day) returns for these assets
        weekly_returns = (data.iloc[i + 7][top10_tickers] /
data.iloc[i][top10_tickers]) - 1
        avg_weekly_return = weekly_returns.mean()

        # Apply the weekly stop loss: cap the weekly return at -5% if
necessary
        avg_weekly_return_sl = avg_weekly_return if avg_weekly_return >=
stop_loss_threshold else stop_loss_threshold

        results.append((current_date, avg_weekly_return_sl))

    # Create a DataFrame for the year and compute cumulative returns
    df_year = pd.DataFrame(results, columns=['Week Start', 'Avg 1 Week
Return'])
    df_year['Cumulative Return'] = (df_year['Avg 1 Week Return'] +
1).cumprod() - 1
    df_year = df_year.sort_values('Week Start').reset_index(drop=True)
    # Assign sequential week numbers for plotting on the x-axis
    df_year['Week'] = df_year.index + 1
    yearly_results[year] = df_year

# Plot the cumulative weekly returns for each year on the same chart using
week number as the x-axis
plt.figure(figsize=(12, 6))
for year, df_year in yearly_results.items():
    plt.plot(df_year['Week'], df_year['Cumulative Return'] * 100, marker='o',
label=str(year))

plt.xlabel('Week Number')
plt.ylabel('Cumulative Return (%)')
plt.title('Cumulative Weekly Returns with Weekly Stop Loss (Risk-Adjusted
Momentum)')
plt.legend()
plt.grid(True)
plt.show()
```

In this implementation:

- **Data Acquisition & Lookback:**
  For each year, data is downloaded starting 30 days before the year's start, ensuring that our 30-day lookback for momentum and volatility calculations is fully covered.
- **Weekly Computation:**
  The strategy calculates the average weekly return of the top 10 risk-adjusted assets and applies a stop loss by capping any weekly loss at –5%.
- **Cumulative Returns & Alignment:**
  Cumulative returns are computed by compounding weekly returns. Each week within the year is assigned a sequential week number so that all years can be overlaid on the same x-axis (Week 1, Week 2, ...).
- **Visualization:**
  The final plot overlays the cumulative weekly returns (expressed in percent) for each year from 2020 to 2024. This allows for a direct visual comparison of how the strategy performed week by week across different market environments.



Conclusion

Overlaying cumulative weekly returns for multiple years provides a clear picture of seasonal or cyclical performance trends in a momentum strategy with a weekly stop loss.

This visualization can help traders understand how the strategy behaves during various market conditions, making it easier to adjust parameters or risk management techniques as needed. We only considered long positions, however if we used a margin or futures account and used both long and short positions for top gaining and losing ones we could have definitely made even more phenomenal results. I might try it and share the results later in another article. By combining a risk-adjusted momentum selection with a weekly stop loss, this simple strategy aims to capture trends while containing drawdowns—a valuable approach in the volatile world of cryptocurrencies.

Algorithmic Bitcoin Trading Strategy using Machine Learning Classification

This tutorial provides a comprehensive guide to developing an algorithmic trading strategy for Bitcoin using machine learning classification techniques. We'll cover everything from fetching real-time Bitcoin data and engineering predictive features to building and evaluating classification models, and finally, backtesting the strategy. This guide is designed to be self-contained, with all necessary Python code and explanations.

1. Introduction: Classification for Trading Signals

Cryptocurrency markets, known for their volatility and 24/7 trading, present unique challenges and opportunities for algorithmic trading. Machine learning, particularly classification, can be employed to predict market movements and generate trading signals (e.g., buy, sell, or hold).

The core idea is to transform the problem of predicting price movements into a classification task. For instance, we can classify the next period's expected price movement into categories like "price will rise" (buy signal) or "price will fall" (sell signal). One powerful aspect of machine learning is **feature engineering**, where we create new, informative features from raw data (like price and volume) to improve model performance. Technical indicators are a common source for such features.

This tutorial will focus on:

- Building a trading strategy based on classifying buy/sell signals.
- Engineering features using common technical indicators.
- Developing a framework to backtest the trading strategy's performance.
- Choosing appropriate evaluation metrics for a trading strategy.

2. Problem Definition: Predicting Buy/Sell Signals

We aim to predict whether the current trading signal for Bitcoin is to **buy (1)** or **sell (0)**. This signal will be determined by comparing short-term and long-term price trends. For example, if a short-term moving average of the price is above a long-term moving average, it might indicate an uptrend (buy signal), and vice-versa.

- **Data:** We'll use historical Bitcoin price data. We will fetch up-to-date data using `yfinance`.
- **Features:** We will create various trend and momentum technical indicators from the price data to serve as input features for our classification model.
- **Target Variable:** A binary signal (1 for buy, 0 for sell) derived from the relationship between short-term and long-term moving averages.

3. Getting Started: Setting Up the Environment

3.1. Python Packages

We'll need several Python libraries:

- **`yfinance`**: For fetching financial data (Bitcoin prices).
- **`pandas`**: For data manipulation and analysis.
- **`numpy`**: For numerical operations.
- **`matplotlib.pyplot`** and **`seaborn`**: For data visualization.
- **`scikit-learn`**: For machine learning tasks, including:
  - `model_selection` (for `train_test_split`, `KFold`, `cross_val_score`, `GridSearchCV`)
  - Various classifiers (e.g., `LogisticRegression`, `DecisionTreeClassifier`, `RandomForestClassifier`)
  - `metrics` (for `accuracy_score`, `confusion_matrix`, `classification_report`)

```
import yfinance as yf
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, KFold, cross_val_score,
GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier, AdaBoostClassifier, ExtraTreesClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import warnings
```

```
warnings.filterwarnings(action='ignore')

# Set a consistent style for plots
plt.style.use('seaborn-v0_8-whitegrid')
pd.set_option('display.width', 100)
```

3.2. Loading the Data

We will fetch Bitcoin (BTC-USD) data using `yfinance`. The original context uses minute-by-minute data; for simplicity and common practice with `yfinance` for daily strategies, we'll fetch daily data. The principles remain the same.

```
ticker = 'BTC-USD'
start_date = '2018-01-01'
end_date = pd.to_datetime('today').strftime('%Y-%m-%d')

try:
    raw_data = yf.download(ticker, start=start_date, end=end_date,
auto_adjust=False, progress=False)
    if raw_data.empty:
        raise ValueError("No data downloaded. Check ticker or date range.")

    dataset = raw_data[['Open', 'High', 'Low', 'Close', 'Volume']].copy()
    dataset.rename(columns={'Volume': 'Volume_(BTC)'}, inplace=True)
    print("Successfully downloaded Bitcoin data.")
except Exception as e:
    print(f"Error downloading data: {e}")
    print("Using a dummy dataset for demonstration purposes.")
    dates = pd.date_range(start='2020-01-01', periods=1000, freq='D')
    data_dummy = {
        'Open': np.random.rand(1000) * 10000 + 30000,
        'High': np.random.rand(1000) * 10000 + 35000,
        'Low': np.random.rand(1000) * 10000 + 25000,
        'Close': np.random.rand(1000) * 10000 + 30000,
        'Volume_(BTC)': np.random.rand(1000) * 100 + 10
    }
    dataset = pd.DataFrame(data_dummy, index=dates)

print("\nDataset shape:", dataset.shape)
dataset.dropna(axis=0, how='all', inplace=True) # Drop rows if all values are
NaN (can happen with yfinance for some dates)
print("Dataset shape after dropping all-NaN rows:", dataset.shape)


# 4. Exploratory Data Analysis (EDA)
print("\nDataset Info:")
dataset.info()
```

4. Exploratory Data Analysis (EDA)

A quick look at the data structure.

```
print("\nDataset Info:")
dataset.info()

print("\nSummary Statistics:")
print(dataset.describe())
```

Visualizing the closing price helps understand its trend and volatility.

```
plt.figure(figsize=(14, 7))
dataset['Close'].plot(grid=True)
plt.title(f'{ticker} Closing Price ({start_date} to {end_date})')
plt.ylabel('Price (USD)')
plt.savefig('bitcoin_closing_price.png')
print("\nSaved Bitcoin closing price plot to bitcoin_closing_price.png")
# plt.show()
plt.close()
```

Bitcoin's price chart typically shows significant volatility and distinct trend periods.

5. Data Preparation

5.1. Data Cleaning

Financial data can have missing values, especially for less liquid assets or specific exchanges. For daily `yfinance` data, NaNs are less common for major assets like BTC-USD but should still be checked. The PDF uses `ffill()` (forward fill) to handle NaNs.

```
print("\nMissing values before cleaning (after initial load):")
print(dataset.isnull().sum())
dataset.fillna(method='ffill', inplace=True)
dataset.fillna(method='bfill', inplace=True)
print("\nMissing values after initial ffill/bfill:")
print(dataset.isnull().sum())
dataset.dropna(inplace=True) # Drop any remaining rows with NaNs, if any
print("Dataset shape after full NaN drop:", dataset.shape)

if dataset.empty:
    print("Dataset is empty after initial cleaning. Exiting.")
    exit()
```

The `Timestamp` column in the original PDF's dataset (minute data) was not useful for modeling and was dropped. For our daily data, the DatetimeIndex is useful and kept.

5.2. Preparing the Target Variable (`signal`)

The trading signal (our target variable) is generated by comparing a short-term moving average (MAVG) with a long-term MAVG.

- If short-term MAVG > long-term MAVG: Buy signal (1)
- Otherwise: Sell signal (0)

We'll use a 10-period rolling mean for the short-term MAVG and a 60-period rolling mean for the long-term MAVG, applied to the 'Close' price.

```
short_window = 10
long_window = 60
dataset['short_mavg'] = dataset['Close'].rolling(window=short_window,
min_periods=1).mean()
dataset['long_mavg'] = dataset['Close'].rolling(window=long_window,
min_periods=1).mean()
dataset['signal'] = 0.0
valid_signal_idx_start = max(short_window, long_window) -1
if len(dataset) > valid_signal_idx_start :
    dataset.loc[dataset.index[valid_signal_idx_start:], 'signal'] = np.where(
        dataset['short_mavg'][valid_signal_idx_start:] >
dataset['long_mavg'][valid_signal_idx_start:], 1.0, 0.0
    )
```

5.3. Feature Engineering: Technical Indicators

Raw price/volume data might not be sufficient for a model to learn complex patterns. Technical indicators can extract underlying trend, momentum, volatility, and other characteristics from the market data. We will create several common indicators to use as features.

Technical Indicators to Implement:

1. **Exponential Moving Average (EMA):** Similar to SMA but gives more weight to recent prices.
   $\text{EMA}_{\text{today}} = (\text{Value}_{\text{today}} \times \text{Multiplier}) + \text{EMA}_{\text{yesterday}} \times (1 - \text{Multiplier})$
   where $\text{Multiplier} = \frac{2}{\text{Period}+1}$

2. **Rate of Change (ROC):** Measures the percentage change in price between the current price and the price n periods ago. $\text{ROC} = \left(\frac{\text{Close}_{\text{today}} - \text{Close}_{\text{n periods ago}}}{\text{Close}_{\text{n periods ago}}}\right) \times 100$

3. **Momentum (MOM):** Measures the absolute change in price over n periods. $\text{MOM} = \text{Close}_{\text{today}} - \text{Close}_{\text{n periods ago}}$

4. **Relative Strength Index (RSI):** A momentum oscillator that measures the speed and change of price movements. RSI oscillates between 0 and 100.
   - Typically, RSI > 70 indicates overbought conditions, and RSI < 30 indicates oversold conditions.
   - Calculation involves average gains and average losses over a period. $\mathrm{RS} = \frac{\text{Average Gain}}{\text{Average Loss}}$ $\mathrm{RSI} = 100 - \frac{100}{1+\mathrm{RS}}$
5. **Stochastic Oscillator (%K and %D):** Compares a particular closing price of an asset to a range of its prices over a certain period of time.
   - %K Line: $\%K = \left(\frac{\text{Current Close} - \text{Lowest Low over period}}{\text{Highest High over period} - \text{Lowest Low over period}}\right) \times 100$
   - %D Line: Typically a 3-period SMA of %K (slow stochastic).
6. **Moving Average (MA):** Simple moving average (already used for signal, but can be features too).

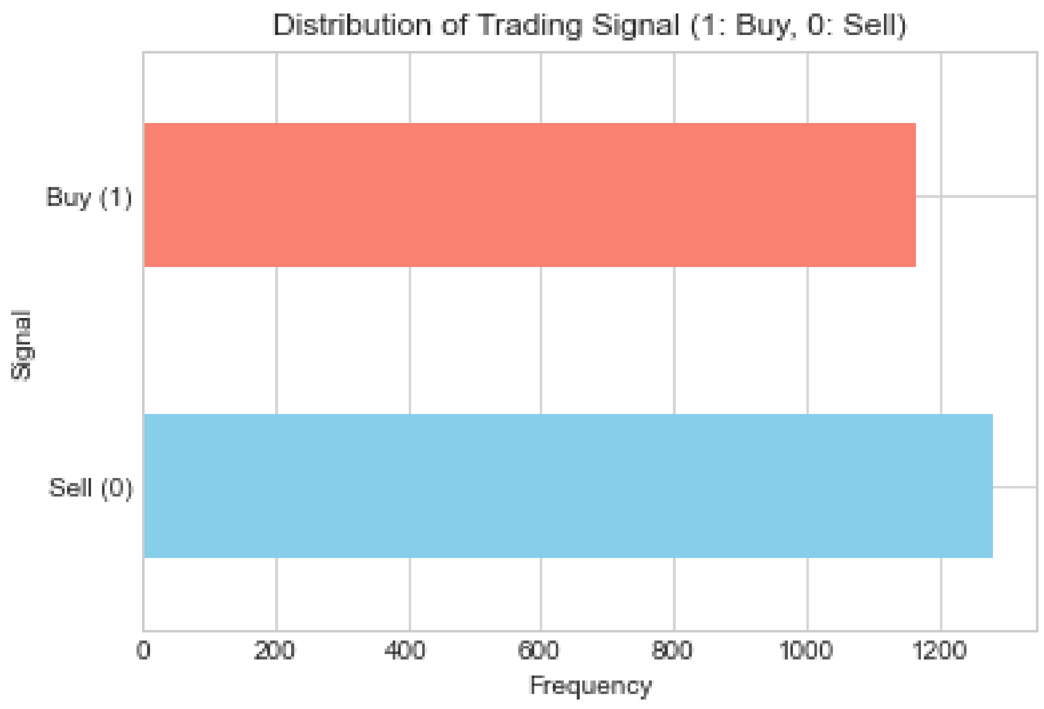
```
for n_ema in [10, 30, 200]:
    dataset[f'EMA{n_ema}'] = EMA(dataset['Close'], n_ema)
for n_roc in [10, 30]:
    dataset[f'ROC{n_roc}'] = ROC(dataset['Close'], n_roc)
for n_mom in [10, 30]:
    dataset[f'MOM{n_mom}'] = MOM(dataset['Close'], n_mom)
for n_rsi in [10, 30, 200]:
    dataset[f'RSI{n_rsi}'] = RSI(dataset['Close'], n_rsi)
stoch_periods = [10, 30, 200]
d_smooth_period = 3
for n_stoch in stoch_periods:
    dataset[f'%K_{n_stoch}'] = STOK(dataset['Close'], dataset['Low'],
dataset['High'], n_stoch)
    dataset[f'%D_{n_stoch}_{d_smooth_period}'] =
STOD(dataset[f'%K_{n_stoch}'], d_smooth_period)
for n_ma in [21, 63, 252]:
    dataset[f'MA{n_ma}'] = MA(dataset['Close'], n_ma)

initial_rows = len(dataset)
dataset.replace([np.inf, -np.inf], np.nan, inplace=True) # Replace infs
created by indicators like RSI if loss is 0
dataset.dropna(inplace=True)
print(f"\nDropped {initial_rows - len(dataset)} rows due to NaNs/infs from
feature engineering.")

if dataset.empty:
    print("Dataset is empty after feature engineering and NaN drop. Cannot
proceed.")
    exit()
```

5.4. Data Visualization (Post Feature Engineering)

Let's check the distribution of our target variable `signal` after all data preparation.

```
plt.figure(figsize=(6, 4))
dataset['signal'].value_counts().plot(kind='barh', color=['skyblue',
'salmon'])
plt.title('Distribution of Trading Signal (1: Buy, 0: Sell)')
plt.xlabel('Frequency')
plt.ylabel('Signal')
plt.yticks(ticks=[0,1], labels=['Sell (0)', 'Buy (1)']) # Adjust based on
value_counts order
# plt.show()
plt.savefig('bitcoin_signal_distribution.png')
print("\nSaved trading signal distribution plot to
bitcoin_signal_distribution.png")
plt.close()
```

The distribution might be relatively balanced or slightly skewed depending on the market period and MAVG parameters. The PDF's example shows it as relatively balanced.



Pasted image 20250521115229.png

Pasted image 20250521115229.png

6. Evaluate Algorithms and Models

6.1. Prepare Data for Modeling

Separate features (X) and target (y). Drop columns used for target creation if they are not intended as features.

```
if 'signal' not in dataset.columns:
    print("Error: 'signal' column is missing from the dataset before
splitting.")
    exit()

features_to_drop_for_X = ['signal', 'short_mavg', 'long_mavg']
X = dataset.drop(columns=features_to_drop_for_X, errors='ignore')
y = dataset['signal']

X = X.apply(pd.to_numeric, errors='coerce').dropna(axis=1,
how='all').fillna(0)

if X.empty or len(X) != len(y) or X.shape[1] == 0:
    print("Feature set X is empty, mismatched with y, or has no columns after
final processing. Cannot proceed.")
    exit()
```

6.2. Train-Test Split

The PDF uses the last 100,000 observations for faster calculation. For daily data, this is a very long period. Let's use a standard chronological split for time series, e.g., 80% for training, 20% for testing.

```
split_index = int(len(X) * 0.8)
if split_index < 1 or split_index >= len(X) -1 :
    print(f"Cannot perform train-test split with current data size: {len(X)}.
Need more data after NaN drops.")
    exit()

X_train = X.iloc[:split_index]
X_test = X.iloc[split_index:]
y_train = y.iloc[:split_index]
y_test = y.iloc[split_index:]

if X_train.empty or X_test.empty or y_train.empty or y_test.empty:
    print("Training or testing set is empty. Cannot proceed with model
evaluation.")
    exit()
```

6.3. Test Options and Evaluation Metric

Given the signal distribution, **accuracy** can be a reasonable starting metric if the classes are somewhat balanced. We also need to look at precision, recall, and F1-score for buy/sell signals.

```
scoring_metric = 'accuracy'
num_folds = 5
kfold = KFold(n_splits=num_folds, shuffle=True, random_state=42)
```

6.4. Compare Models and Algorithms

Spot-check various classification algorithms.

```
models_btc = []
models_btc.append(('LR', LogisticRegression(solver='liblinear', max_iter=200,
random_state=42)))
models_btc.append(('LDA', LinearDiscriminantAnalysis()))
models_btc.append(('CART', DecisionTreeClassifier(random_state=42)))
models_btc.append(('RF', RandomForestClassifier(random_state=42, n_jobs=-1)))
models_btc.append(('GBM', GradientBoostingClassifier(random_state=42)))

results_btc = []
names_btc = []
print(f"\nSpot-checking models using {scoring_metric}:")
for name, model in models_btc:
    try:
        cv_results = cross_val_score(model, X_train, y_train, cv=kfold,
scoring=scoring_metric, n_jobs=-1)
        results_btc.append(cv_results)
        names_btc.append(name)
        print(f"{name}: {cv_results.mean():.4f} ({cv_results.std():.4f})")
    except Exception as e:
        print(f"Could not evaluate {name}: {e}")
```

The PDF identifies Random Forest as performing well among ensemble models. Let's assume it's a good candidate.

7. Model Tuning and Grid Search (Random Forest)

We'll tune hyperparameters for Random Forest using `GridSearchCV`.

```
best_model_btc = None
chosen_model_name_for_tuning = 'RF'
model_to_tune_proto = None
for name, model_proto_iter in models_btc:
    if name == chosen_model_name_for_tuning:
        model_to_tune_proto = model_proto_iter
        break
```

```
if model_to_tune_proto is not None:
    param_grid = {
        'n_estimators': [50, 100], 'max_depth': [5, 10, None], 'criterion':
['gini', 'entropy']
    } if isinstance(model_to_tune_proto, RandomForestClassifier) else {
        'n_estimators': [50, 100], 'learning_rate': [0.05, 0.1], 'max_depth':
[3,5]
    }
    grid = GridSearchCV(estimator=model_to_tune_proto, param_grid=param_grid,
scoring=scoring_metric, cv=kfold, n_jobs=-1)
    try:
        grid_result = grid.fit(X_train, y_train)
        print(f"\nBest {scoring_metric} for {chosen_model_name_for_tuning}:
{grid_result.best_score_:.4f} using {grid_result.best_params_}")
        best_model_btc = grid_result.best_estimator_
    except Exception as e:
        print(f"GridSearchCV failed for {chosen_model_name_for_tuning}: {e}")
        best_model_btc = model_to_tune_proto
        print(f"Using default (untuned) {chosen_model_name_for_tuning}
parameters due to GridSearchCV error.")
        best_model_btc.fit(X_train, y_train)
else:
    print(f"\nModel '{chosen_model_name_for_tuning}' not found or CV failed.
Using a default RF.")
    best_model_btc = RandomForestClassifier(random_state=42,
n_estimators=100, n_jobs=-1)
    if not X_train.empty and not y_train.empty:
         best_model_btc.fit(X_train, y_train)
    else:
        print("Cannot fit default model as training data is empty.")
        best_model_btc = None
```

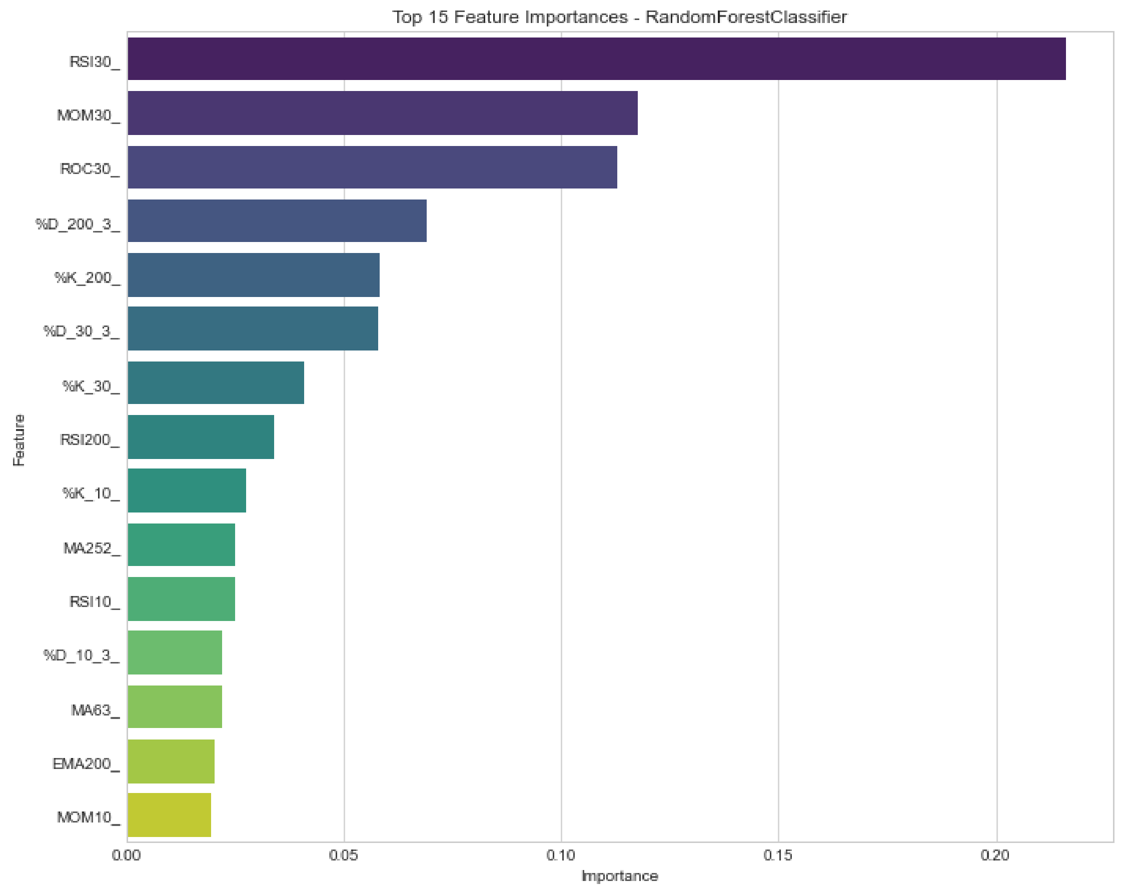
8. Finalize the Model and Evaluate

8.1. Results on the Test Dataset

Evaluate the tuned (or best chosen) model on the unseen test set.

```
if best_model_btc and not X_test.empty and not y_test.empty:
    y_pred_test = best_model_btc.predict(X_test)
    print(f"\nPerformance of Final Model
({best_model_btc.__class__.__name__}) on Test Set:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred_test):.4f}")
    cm_test = confusion_matrix(y_test, y_pred_test)
    print("\nConfusion Matrix (Test Set):\n", cm_test)

    print("\nClassification Report (Test Set):")
    print(f"Unique values in y_test: {np.unique(y_test,
return_counts=True)}")
    print(f"Unique values in y_pred_test: {np.unique(y_pred_test,
return_counts=True)}")
    print(classification_report(y_test, y_pred_test, target_names=['Sell
(0)', 'Buy (1)'], labels=[0, 1], zero_division=0))

    if hasattr(best_model_btc, 'feature_importances_'):
        importances = best_model_btc.feature_importances_
        feature_names_original = X_train.columns

        str_feature_names = []
        for name in feature_names_original:
            if isinstance(name, tuple):
                str_feature_names.append('_'.join(map(str, name)))
            else:
                str_feature_names.append(str(name))

        feature_importance_df = pd.DataFrame({'feature': str_feature_names,
'importance': importances})
        feature_importance_df =
feature_importance_df.sort_values(by='importance', ascending=False)
        print("\nTop 15 Feature Importances (with stringified feature
names):")
        print(feature_importance_df.head(15))
        plt.figure(figsize=(10, 8))
        sns.barplot(x='importance', y='feature',
data=feature_importance_df.head(15), palette='viridis')
        plt.title(f'Top 15 Feature Importances -
{best_model_btc.__class__.__name__}')
        plt.xlabel('Importance')
        plt.ylabel('Feature')
        plt.tight_layout()
        # plt.savefig('bitcoin_feature_importance.png')
        print("\nSaved feature importance plot to
bitcoin_feature_importance.png")
        # plt.close()
else:
    print("\nNo model was finalized for evaluation or test set is empty.")
```

The model's accuracy and other metrics on the test set give an indication of its real-world performance. For tree-based models like Random Forest or GBM, we can examine feature importances.



Pasted image 20250521115521.png

Pasted image 20250521115521.png

This helps understand which technical indicators were most influential in the model's predictions. Momentum indicators like RSI and MOM often show high importance.

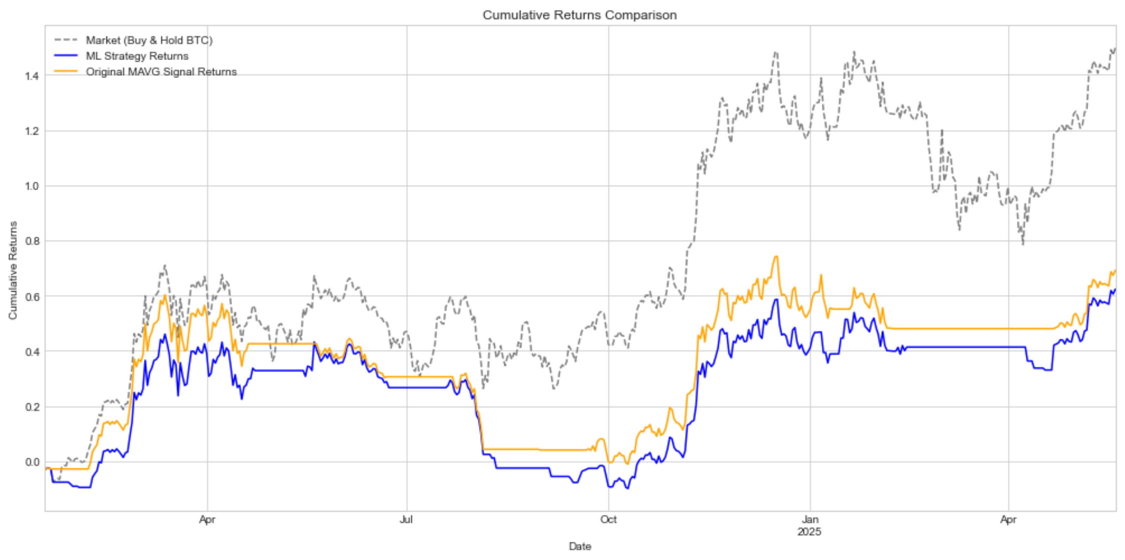
9. Backtesting the Trading Strategy (Simplified)

Backtesting simulates how the strategy would have performed on historical data. We'll create a simple backtest:

- Calculate daily market returns.
- Calculate strategy returns by multiplying market returns by the *predicted signal from the previous day* (since we trade on the next bar after a signal). A 1 means hold (or buy if not holding), a `0` means be out of the market (or sell if holding). This is a long-only interpretation for simplicity.

```
if best_model_btc and not X_test.empty and 'y_pred_test' in locals() and not
y_test.empty:
    backtest_df = pd.DataFrame(index=X_test.index)
    if 'Close' in dataset.columns and 'signal' in dataset.columns and
X_test.index.isin(dataset.index).all():
        backtest_df['Market_Returns'] = dataset.loc[X_test.index,
'Close'].pct_change()
        backtest_df['Predicted_Signal'] = y_pred_test
        backtest_df['Strategy_Returns'] = backtest_df['Market_Returns'] *
backtest_df['Predicted_Signal'].shift(1)
        backtest_df['Actual_MAVG_Signal_Returns'] =
backtest_df['Market_Returns'] * dataset.loc[X_test.index, 'signal'].shift(1)
        backtest_df.dropna(inplace=True)

        if not backtest_df.empty:
            backtest_df['Cumulative_Market_Returns'] = (1 +
backtest_df['Market_Returns']).cumprod() - 1
            backtest_df['Cumulative_Strategy_Returns'] = (1 +
backtest_df['Strategy_Returns']).cumprod() - 1
            backtest_df['Cumulative_Actual_MAVG_Signal_Returns'] = (1 +
backtest_df['Actual_MAVG_Signal_Returns']).cumprod() - 1
            print("\nBacktesting Results (Last 5 days):\n",
backtest_df.tail())
            plt.figure(figsize=(14, 7))
            backtest_df['Cumulative_Market_Returns'].plot(label='Market (Buy
& Hold BTC)', color='gray', linestyle='--')
            backtest_df['Cumulative_Strategy_Returns'].plot(label='ML
Strategy Returns', color='blue')

backtest_df['Cumulative_Actual_MAVG_Signal_Returns'].plot(label='Original
MAVG Signal Returns', color='orange')
            plt.title('Cumulative Returns Comparison')
            plt.ylabel('Cumulative Returns')
            plt.legend()
            plt.tight_layout()
            # plt.savefig('bitcoin_backtest_returns.png')
            print("\nSaved backtesting returns plot to
bitcoin_backtest_returns.png")
            # plt.close()
        else:
            print("\nBacktest DataFrame is empty after processing; cannot
plot returns.")
    else:
        print("\nCould not perform backtesting: 'Close' or 'signal' column
missing or index mismatch.")
else:
    print("\nSkipping backtesting as no model was finalized or
test/prediction data is unavailable.")

print("\n--- Tutorial: Algorithmic Bitcoin Trading Strategy Finished ---")
```

The plot comparing cumulative returns helps assess if the machine learning strategy added value over a simple buy-and-hold or the original MAVG crossover rule.



Pasted image 20250521115650.png

Pasted image 20250521115650.png

10. Conclusion and Next Steps

This tutorial demonstrated a complete workflow for building a Bitcoin trading strategy using machine learning classification. We covered:

- Defining the problem as a classification task.
- Fetching real market data using `yfinance`.
- Extensive feature engineering using technical indicators.
- Training, tuning, and evaluating various classification models.
- Assessing feature importance.
- Performing a simplified backtest.

The results of such a strategy can vary greatly depending on the chosen period, features, model, and market conditions. Key takeaways include the importance of robust feature engineering and careful model evaluation.

Further improvements and considerations could include:

- More sophisticated feature engineering (e.g., volatility measures, order book data if available).
- Different ways to define the target variable (e.g., predicting price change magnitude, multi-class signals like buy/sell/hold).
- Advanced backtesting with considerations for transaction costs, slippage, and risk management.
- Time series cross-validation techniques.

- Exploring more complex models like LSTMs or other deep learning architectures, though they require more data and computational resources.

This framework provides a solid foundation for developing and testing algorithmic trading strategies based on machine learning.

Breakout Strategy with OBV and ATR Confirmation

Breakout trading is a popular strategy aiming to capitalize on significant price movements when an asset's price breaks through a defined support or resistance level. However, not all breakouts lead to sustained trends; many turn out to be "false breakouts" or "fakes." To improve the reliability of breakout signals, traders often incorporate secondary indicators related to volume and volatility.

This article explores a specific breakout strategy applied to Ethereum (ETH-USD) daily data from January 1, 2020, to April 30, 2025. The strategy identifies breakouts using recent price highs/lows and confirms them with two key indicators:

1. **On-Balance Volume (OBV):** A momentum indicator relating price and volume, suggesting accumulation (buying pressure) or distribution (selling pressure).
2. **Average True Range (ATR):** A measure of market volatility.

The goal is to generate buy (UpBreak) or sell/short (DownBreak) signals only when the price breakout is confirmed by corresponding OBV momentum *and* occurs during a period of elevated volatility (ATR significantly above its recent average). We'll walk through the Python implementation using `yfinance`, `pandas`, `talib`, and `matplotlib`, and analyze the results.

The Strategy Logic

The core idea is to filter potential price breakouts for higher probability trades:

- **Price Breakout:** The closing price must move above the highest close or below the lowest close observed over a defined `lookback` period (e.g., 20 days).
- **OBV Confirmation:**
  - For an *upward* breakout, the OBV should also simultaneously break above its own high over the same `lookback` period, indicating volume supports the upward price move.
  - For a *downward* breakout, the OBV should break below its `lookback` low, suggesting volume confirms the selling pressure.
- **Volatility Filter (ATR):** The breakout should occur when market volatility is higher than usual. We require the current ATR (14-day period) to be significantly higher (e.g., 1.2 times) than its average over the `lookback` period. This helps filter out breakouts happening during low-volatility "noise."

Implementation in Python

1. Data Acquisition and Preparation

First, we download the historical daily price data for ETH-USD using the `yfinance` library and handle potential multi-level columns.

Python

```python
import numpy as np
import pandas as pd
import yfinance as yf
import talib
import matplotlib.pyplot as plt

# --- Configuration ---
ticker = 'ETH-USD'
start_date = '2020-01-01'
end_date = '2025-04-30' # Data up to Apr 30, 2025 used in the results
atr_period = 14
lookback   = 20      # Lookback window for highs/lows
atr_thresh = 1.2     # ATR multiplier threshold
horizon    = 30      # Forward return horizon (days)

# 1. Download and Prep Data
print(f"Downloading data for {ticker}...")
data = yf.download(ticker, start=start_date, end=end_date, progress=False)

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

print(f"Data downloaded. Shape: {data.shape}")
# Basic cleaning (ensure essential columns exist, drop NaNs)
required_cols = ['High', 'Low', 'Close', 'Volume']
if not all(col in data.columns for col in required_cols):
     raise ValueError("Missing required columns")
data.dropna(subset=required_cols, inplace=True)
```

The execution yielded:

Downloading data for ETH-USD...

Data downloaded. Shape: (1946, 5)

2. Indicator Calculation

We use the popular `TA-Lib` library to compute OBV and ATR.

Python

```
# 2. Compute Indicators
data['OBV'] = talib.OBV(data['Close'], data['Volume'])
data['ATR'] = talib.ATR(data['High'], data['Low'], data['Close'],
timeperiod=atr_period)

# Drop initial rows where indicators couldn't be calculated
data.dropna(inplace=True)
```

3. Defining Breakout Conditions

We calculate the rolling maximum/minimum price and OBV over the `lookback` window. We use `.shift(1)` to compare the *current* bar's values against the highs/lows established by the *preceding* `lookback` period. We also calculate the rolling mean of ATR.

Python

```
# 3. Compute Rolling Highs/Lows/Means for Breakout Conditions
data['PriceHigh'] = data['Close'].shift(1).rolling(lookback).max()
data['OBVHigh']   = data['OBV'].shift(1).rolling(lookback).max()
data['PriceLow']  = data['Close'].shift(1).rolling(lookback).min()
data['OBVLow']    = data['OBV'].shift(1).rolling(lookback).min()
data['ATRmean']   = data['ATR'].shift(1).rolling(lookback).mean()

# Drop rows with NaNs from rolling calculations
data.dropna(inplace=True)
```

4. Generating Signals

We combine the price, OBV, and ATR conditions using boolean logic to create the `UpBreak` and `DownBreak` signals (1 if conditions met, 0 otherwise).

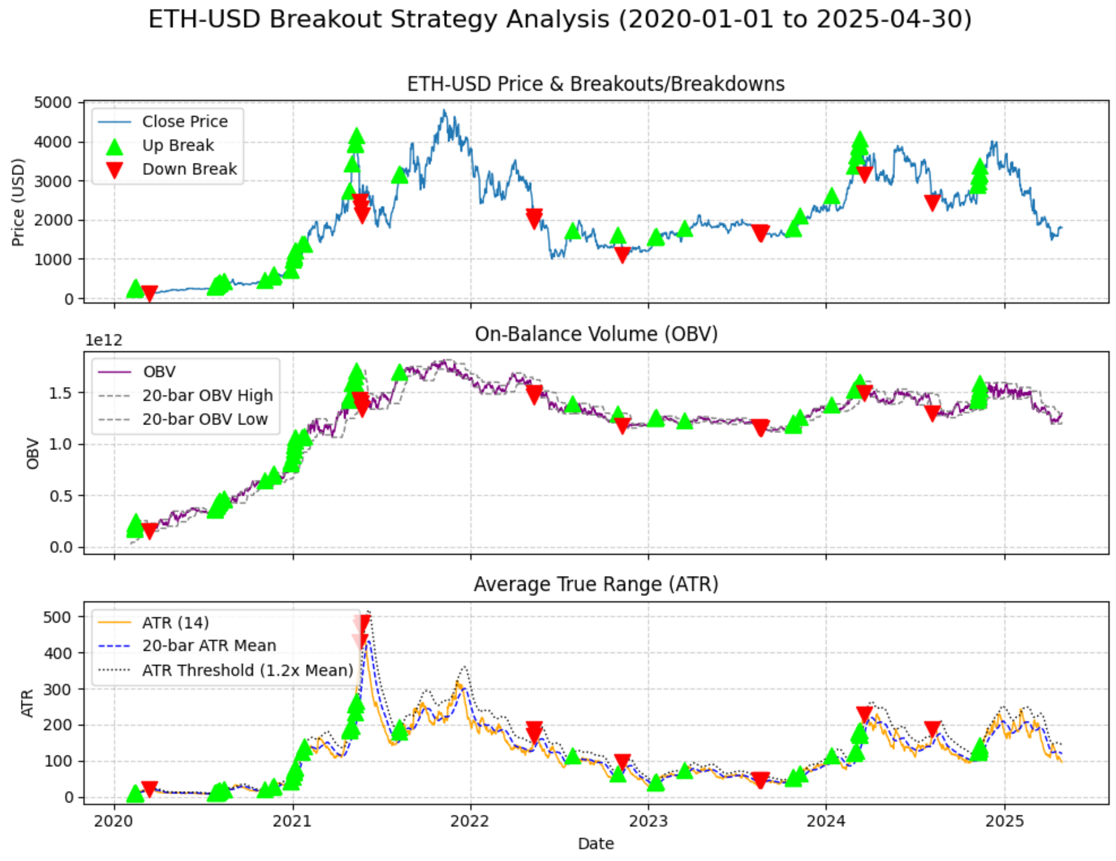
Python

```
# 4. Generate Breakout Signals
# Up Breakout: Close > Price High, OBV > OBV High, ATR > ATR Mean * Threshold
data['UpBreak']   = (
    (data['Close'] > data['PriceHigh']) &
    (data['OBV']   > data['OBVHigh'])   &
    (data['ATR']   > data['ATRmean'] * atr_thresh)
).astype(int)

# Down Breakout: Close < Price Low, OBV < OBV Low, ATR > ATR Mean * Threshold
data['DownBreak'] = (
    (data['Close'] < data['PriceLow'])  &
    (data['OBV']   < data['OBVLow'])    &
    (data['ATR']   > data['ATRmean'] * atr_thresh)
).astype(int)
```

5. Visualization

Plotting the price, OBV, and ATR along with the generated signals is crucial for visually inspecting the strategy's behavior. The code generates three stacked plots:

- Top: ETH-USD Close price with green upward arrows marking `UpBreak` signals and red downward arrows marking `DownBreak` signals.
- Middle: OBV with its rolling high/low bands and corresponding signal markers.
- Bottom: ATR with its rolling mean, the calculated ATR threshold (mean * 1.2), and corresponding signal markers.



###

Performance Evaluation

To assess the predictive power of these signals, we calculate the forward return over a fixed `horizon` (30 days in this analysis). This tells us the percentage change in price 30 days *after* a signal occurred.

Python

```
# 6. Compute Forward Returns for Performance Evaluation
data['FwdRet'] = data['Close'].shift(-horizon) / data['Close'] - 1

# 7. Evaluate Signal Performance (Win Rate & Expectancy)
up_signals   = data[data['UpBreak'] == 1].dropna(subset=['FwdRet'])
```

```
down_signals = data[data['DownBreak'] == 1].dropna(subset=['FwdRet'])
# Calculate return for short trades (profit if price goes down)
down_signals = down_signals.assign(ShortRet = -down_signals['FwdRet'])

# ... [Calculations for wins, win rates, average win/loss, expectancy] ...
```

Results: Win Rate & Expectancy

The analysis over the specified period yielded the following performance metrics based on a 30-day forward return:

```
Calculating signal performance...
Up-break signals:     56  →  Winners:    32  (57.1%)
Up-break expectancy per trade (30-day horizon): 6.38%
Down-break signals:   12  →  Winners:     9  (75.0%)
Down-break expectancy per trade (30-day horizon): 2.83%
```

- **Up Breaks (Longs):** There were 56 signals. 57.1% of these were “winners” (price was higher 30 days later). The average expected return (expectancy) for taking any given Up Break signal, considering both winners and losers, was +6.38% over the next 30 days.
- **Down Breaks (Shorts):** There were fewer signals (12). A higher percentage, 75.0%, were “winners” (price was lower 30 days later). The expectancy per short trade was +2.83% (profit from price decrease) over the next 30 days.

These results suggest that, historically over this period and using this specific fixed horizon, both signal types had a positive expectancy, although the win rate and average outcome differed.

Overall Strategy Returns (Simplified)

To get a sense of the overall potential, we combined all trades and calculated aggregate returns. **Important Caveat:** These calculations are highly simplified. They sum the fixed 30-day forward returns (`total_simple`) or calculate the compounded product (`total_compound`) assuming each trade is independent and held for exactly 30 days. This *does not* represent a realistic portfolio simulation (which would need to handle overlapping trades, position sizing, compounding within trades, transaction costs, etc.).

Python

```
# 8. Calculate Overall Strategy Returns (Simplified)
up_trades   = up_signals.assign(Return=up_signals['FwdRet'], Type='Long')
down_trades = down_signals.assign(Return=down_signals['ShortRet'],
Type='Short')
all_trades = pd.concat([up_trades, down_trades]).sort_index()

# ... [Calculations for total_simple and total_compound] ...
```

Results: Simplified Overall Performance

```
Calculating overall strategy return (simplified)...

--- Simplified Strategy Performance (30-day horizon) ---
Number of evaluated trades: 68
Simple sum total return:    391.46%
Compound total return:      179.57%
Average return per trade:    5.76%

--- Performance by Trade Type ---
        count      mean       sum       std
Type
Long       56  0.063841  3.575118  0.306096
Short      12  0.028293  0.339521  0.171162

Analysis complete.
```

- A total of 68 signals (56 long, 12 short) were evaluated.
- The simple sum of the 30-day returns for all trades was +391.46%.
- The compounded return, calculated as `(1 + R1) * (1 + R2) * ... - 1`, was +179.57%. The significant difference between simple and compound returns highlights the impact of large winners/losers and the simplified nature of this metric.
- The average return across all 68 trades was +5.76% over the 30-day horizon.
- The breakdown confirms the long signals had a higher average return (6.38%) than the short signals (2.83%) during this period, though shorts had a higher win rate.

Conclusion

This analysis demonstrated how to implement and test a breakout trading strategy for ETH-USD using Python, confirming price breaks with OBV and filtering by ATR. The historical results from Jan 2020 to Apr 2025, based on a 30-day fixed forward return, showed positive expectancy for both long and short signals generated by this specific set of rules. Long signals were more frequent and had higher average returns, while short signals were less common but had a higher win rate.

It is crucial to remember that:

- **Past performance is not indicative of future results.** Market conditions change.
- The return calculations shown are highly simplified and **do not represent actual portfolio returns.** A proper backtest would require a more sophisticated engine handling entries, exits (e.g., stop-losses, take-profits, opposite signals), position sizing, and costs.
- The parameters (`lookback`, `atr_thresh`, `horizon`) were chosen arbitrarily for this example. Optimization might yield different results.

This framework provides a starting point for exploring quantitative breakout strategies. Further research could involve testing different parameters, applying the strategy to other assets or timeframes, incorporating risk management rules, and building a more robust event-driven backtesting system.

Building and Backtesting a Multi-Indicator Trading Strategy with Backtrader

Technical indicators are fundamental tools in a trader's toolkit, helping to quantify price momentum, trend strength, and potential reversals. Two of the most widely used indicators are the Exponential Moving Average (EMA) and the Average Directional Index (ADX). In this article, we'll explore their mathematical foundations, trading applications, and how to implement them using the popular **TA-Lib** Python library. ## Exponential Moving Average (EMA)

Definition & Formula

An EMA is a type of moving average that places greater weight on more recent prices, making it more responsive to new information than a Simple Moving Average (SMA). The EMA at time *t* is calculated as:

[*t = P_t + (1 -)* {t-1}]

where:

- (P_t) is the price (usually the closing price) at time *t*.
- (=) is the smoothing factor for an *n*-period EMA.
- (_{t-1}) is the previous period's EMA.

Trading Applications

- **Trend identification**: Traders often use two EMAs (e.g., EMA(9) and EMA(21)) and look for crossovers. A "bullish" signal occurs when the shorter EMA crosses above the longer; a "bearish" when it crosses below.
- **Dynamic support/resistance**: Price tends to "bounce" off key EMAs.
- **Momentum confirmation**: A rising EMA confirms upward momentum.

Average Directional Index (ADX)

Definition & Components

The ADX measures trend strength rather than direction. It's derived from two other indicators, the Positive Directional Indicator (+DI) and the Negative Directional Indicator (–DI). The steps are:

1. Compute True Range (TR) and smoothed directional movements:
   - (+DM_t = (H_t - H_{t-1}, 0))
   - (-DM_t = (L_{t-1} - L_t, 0))
2. Smooth these values (often via an EMA or Wilder's smoothing).
3. Calculate the Directional Indicators:
   - (_t = 100 (/))
   - (_t = 100 (/))
4. The ADX is then the EMA (or Wilder's) of the Directional Movement Index (DX):

   [_t = 100] [_t = (, ;)]

Trading Applications

- **Trend strength**: ADX above 25–30 generally indicates a strong trend; below 20 suggests a weak or sideways market.
- **Filter for trend-following systems**: Only take signals when ADX is above a threshold.
- **Directional confirmation**: +DI > –DI suggests an uptrend; –DI > +DI suggests a downtrend.

Average True Range (ATR)

Definition & Formula

The Average True Range (ATR) measures market volatility by decomposing the True Range (TR) over a given period and then smoothing it. The True Range for period *t* is:

[*t = (H_t - L_t,; |H_t - C*{t-1}|,; |L_t - C_{t-1}|)]

where:

- (H_t) and (L_t) are the high and low of the current period.

- (C_{t-1}) is the close of the previous period.

The ATR is then an *n*-period moving average of the TR. In Wilder's smoothing (the default):

[_t =]

Trading Applications

- **Volatility Filter**: Require ATR to exceed a minimum threshold to avoid low-volatility "noise" and to cap entries when ATR is extremely high (blow-off tops).
- **Stop-Loss Placement**: Place stops at a multiple of ATR (e.g., 2×ATR below entry for longs) to adapt dynamically to changing volatility.
- **Position Sizing**: Use ATR to scale position size so that risk (in ATR units) is consistent across different volatility regimes.
- **Trailing Stops**: Trail the stop by an ATR multiple, letting it widen in volatile periods and tighten when volatility contracts.

Implementing with TA-Lib

Here's a compact example showing **EMA(9/21) cross**, **ADX(14) filter**, and **ATR(14) for stop-loss**, all via TA-Lib:

Python

```python
import talib
import pandas as pd

# Load your OHLCV data as a DataFrame 'df'
# df = pd.read_csv('your_data.csv', parse_dates=True, index_col='Date')

# Compute indicators
df['EMA9']   = talib.EMA(df['Close'], timeperiod=9)
df['EMA21']  = talib.EMA(df['Close'], timeperiod=21)
df['ADX14']  = talib.ADX(df['High'], df['Low'], df['Close'], timeperiod=14)
df['ATR14']  = talib.ATR(df['High'], df['Low'], df['Close'], timeperiod=14)

# Signal logic (example for a long entry)
df['Signal'] = (
    (df['EMA9'].shift(1) < df['EMA21'].shift(1)) &  # previously below
    (df['EMA9']    > df['EMA21'])        &  # now above
    (df['ADX14']  > 25)                  &  # trend strong
    (df['ATR14']/df['Close'] > 0.005)       # minimum volatility
)

# Compute stop level for each signal bar
df['StopPrice'] = df['Close'] - 2.0 * df['ATR14']
```

```
print(df[['Close','EMA9','EMA21','ADX14','ATR14','Signal','StopPrice']].tail(
10))
```

Building and Backtesting A Strategy

Algorithmic trading relies on well-defined strategies that can be rigorously tested against historical data. Python, with libraries like `backtrader`, `yfinance`, and `matplotlib`, provides a powerful ecosystem for developing, backtesting, and analyzing such strategies.

Now we will go through a custom trading strategy named `SimpleDualMA`. This strategy demonstrates how multiple common technical indicators—Dual Exponential Moving Averages (EMAs), Average Directional Index (ADX), Volume Simple Moving Average (SMA), and Average True Range (ATR) for a trailing stop-loss—can be combined to make trading decisions. We'll explore its implementation using `backtrader` and then set up a backtest using historical Bitcoin (BTC-USD) data.

The Strategy: `SimpleDualMA`

The `SimpleDualMA` strategy is designed to be a trend-following system. It aims to enter trades when a trend is established and confirmed by multiple factors, and it uses a dynamic trailing stop-loss to manage risk and lock in profits.

Core Components & Indicators:

1. **Dual Exponential Moving Averages (EMAs):**
   - **Purpose:** EMAs are used to identify trend direction. A faster EMA (e.g., 9-period) reacting more quickly to price changes, and a slower EMA (e.g., 21-period) providing a more stable trendline.
   - **Signal:** A "golden cross" (fast EMA crosses above slow EMA) signals a potential buy. A "death cross" (fast EMA crosses below slow EMA) signals a potential sell.
   - **Parameters:** `fast=9`, `slow=21`
2. **Average Directional Index (ADX):**
   - **Purpose:** ADX measures the strength of a trend, irrespective of its direction. It helps filter out trades during choppy or non-trending market conditions.
   - **Signal:** A trade is considered only if the ADX value is above a certain threshold (e.g., 25), indicating a sufficiently strong trend.
   - **Parameters:** `adx_period=14`, `adx_limit=25`
3. **Volume Filter (Volume SMA):**
   - **Purpose:** Volume can confirm the strength behind a price move. A breakout or trend continuation on high volume is generally considered more significant.

- **Signal:** A trade is considered only if the current bar's volume is greater than its Simple Moving Average (SMA) multiplied by a certain factor, suggesting increased market participation.
- **Parameters:** `vol_sma=7` (period for volume SMA), `vol_mul=1.2` (multiplier for current volume vs. SMA)

4. **Average True Range (ATR) Trailing Stop-Loss:**
    - **Purpose:** ATR measures market volatility. Using it for a stop-loss allows the stop to be wider during volatile periods and tighter during calm periods. A trailing stop adjusts as the trade moves in a favorable direction, protecting profits.
    - **Signal (Exit):**
        - For a **long position**, the stop is initially set at `entry_price - (stop_atr_multiplier * ATR)`. If the price rises, the stop-loss also rises, maintaining the same ATR distance from the current high (or a modified high). If the price falls below this trailing stop, the position is closed.
        - For a **short position**, the stop is initially set at `entry_price + (stop_atr_multiplier * ATR)`. If the price falls, the stop-loss also falls. If the price rises above this trailing stop, the position is closed.
    - **Parameters:** `atr_period=30`, `stop_atr=10.0` (multiplier for ATR value to set stop distance)

Strategy Implementation in Python with `backtrader`

Let's look at the Python code.

Python

```
import backtrader as bt
import yfinance as yf
import matplotlib.pyplot as plt
# %matplotlib qt5 # Use this magic in Jupyter for interactive Qt plots

class SimpleDualMA(bt.Strategy):
    params = dict(
        fast=9, slow=21,           # EMA periods
        adx_period=14, adx_limit=25, # ADX parameters
        vol_sma=7, vol_mul=1.2,     # Volume filter parameters
        atr_period=30, stop_atr=10.0, # ATR Trailing Stop parameters
    )

    def __init__(self):
        d = self.datas[0] # Primary data feed

        # Indicators
        self.ema_fast = bt.ind.EMA(d.close, period=self.p.fast)
        self.ema_slow = bt.ind.EMA(d.close, period=self.p.slow)
        self.adx      = bt.ind.ADX(d, period=self.p.adx_period)
        self.vol_sma  = bt.ind.SMA(d.volume, period=self.p.vol_sma)
        self.atr      = bt.ind.ATR(d, period=self.p.atr_period)

        # To keep track of pending orders, entry price, and trailing stop
        self.order = None
        self.price_entry = None # Price at which the position was entered
        self.trailing_stop_price = None # Current trailing stop price

        # Crossover signal for EMAs
        self.crossover = bt.indicators.CrossOver(self.ema_fast,
self.ema_slow)

    def log(self, txt):
        ''' Logging function for this strategy'''
        dt = self.data.datetime.date(0).isoformat()
        print(f"{dt} {txt}")

    def notify_order(self, order):
        ''' Handles order notifications and sets initial trailing stop '''
        if order.status in [order.Submitted, order.Accepted]:
            # Buy/Sell order submitted/accepted to/by broker - Nothing to do
            return

        # Check if an order has been completed
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
                self.price_entry = order.executed.price
                # Calculate initial trailing stop for a long position
                self.trailing_stop_price = self.price_entry - self.p.stop_atr
* self.atr[0]
                self.log(f'INITIAL TRAILING STOP (LONG) SET AT:
{self.trailing_stop_price:.2f}')
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
                self.price_entry = order.executed.price
                 # Calculate initial trailing stop for a short position
                self.trailing_stop_price = self.price_entry + self.p.stop_atr
* self.atr[0]
                self.log(f'INITIAL TRAILING STOP (SHORT) SET AT:
{self.trailing_stop_price:.2f}')

            self.bar_executed = len(self) # Bar number when order was
```

```
executed

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order Canceled/Margin/Rejected: Status
{order.getstatusname()}')
            if self.order and order.ref == self.order.ref: # Check if it was
the main entry order
                self.price_entry = None # Reset on rejected entry
                self.trailing_stop_price = None

        # Clear the pending order flag
        self.order = None

    def next(self):
        ''' Core logic executed on each new bar of data '''
        price = self.data.close[0] # Current closing price

        # 1) If an order is pending, do nothing
        if self.order:
            return

        # 2) Manage existing position with trailing stop
        if self.position.size > 0:  # Currently LONG
            # Re-initialize trailing_stop_price if it's somehow None (e.g.,
after deserialization, though less common in simple scripts)
            if self.trailing_stop_price is None and self.price_entry is not
None:
                self.trailing_stop_price = self.price_entry - self.p.stop_atr
* self.atr[0]
                self.log(f'RE-INITIALIZED TRAILING STOP (LONG) AT:
{self.trailing_stop_price:.2f}')

            if self.trailing_stop_price is not None:
                # Calculate new potential stop price based on current price
and ATR
                new_potential_stop = price - self.p.stop_atr * self.atr[0]
                # Update trailing stop price only if the new potential stop
is higher (moves in favor of the trade)
                if new_potential_stop > self.trailing_stop_price:
                    self.trailing_stop_price = new_potential_stop
                    # self.log(f"LONG TRAILING STOP UPDATED TO:
{self.trailing_stop_price:.2f}") # Optional: log every update

                # Check if price hits the trailing stop
                if price < self.trailing_stop_price:
                    self.log(f"LONG TRAILING STOP HIT @ {price:.2f}
(stop={self.trailing_stop_price:.2f})")
                    self.order = self.close() # self.close() will create and
```

```
assign an order to self.order
                    return

        elif self.position.size < 0:  # Currently SHORT
            if self.trailing_stop_price is None and self.price_entry is not
None:
                self.trailing_stop_price = self.price_entry + self.p.stop_atr
* self.atr[0]
                self.log(f'RE-INITIALIZED TRAILING STOP (SHORT) AT:
{self.trailing_stop_price:.2f}')

            if self.trailing_stop_price is not None:
                # Calculate new potential stop price
                new_potential_stop = price + self.p.stop_atr * self.atr[0]
                # Update trailing stop price only if the new potential stop
is lower
                if new_potential_stop < self.trailing_stop_price:
                    self.trailing_stop_price = new_potential_stop
                    # self.log(f"SHORT TRAILING STOP UPDATED TO:
{self.trailing_stop_price:.2f}") # Optional

                # Check if price hits the trailing stop
                if price > self.trailing_stop_price:
                    self.log(f"SHORT TRAILING STOP HIT @ {price:.2f}
(stop={self.trailing_stop_price:.2f})")
                    self.order = self.close()
                    return

        # 3) No open position OR stop loss triggered and closed -> check for
new entry
        if not self.position: # Check if position is effectively zero (no
open trades)
            self.trailing_stop_price = None # Reset trailing stop when out of
market
            self.price_entry = None         # Reset entry price

            # Entry condition filters
            vol_ok = self.data.volume[0] > self.vol_sma[0] * self.p.vol_mul
            adx_ok = self.adx.adx[0] >= self.p.adx_limit # Access .adx line
for ADX value

            # 3a) LONG entry: fast EMA crosses above slow EMA, with ADX and
Volume confirmation
            if self.crossover[0] > 0 and adx_ok and vol_ok: # crossover > 0
for fast crossing above slow
                self.log(f"BUY CREATE  @ {price:.2f}
(ADX={self.adx.adx[0]:.1f}, Vol={self.data.volume[0]:.0f})")
                self.order = self.buy()
                return # Important to return after placing an order to avoid
conflicting logic in the same bar

            # 3b) SHORT entry: fast EMA crosses below slow EMA, with ADX and
Volume confirmation
            if self.crossover[0] < 0 and adx_ok and vol_ok: # crossover < 0
for fast crossing below slow
                self.log(f"SELL CREATE @ {price:.2f}
(ADX={self.adx.adx[0]:.1f}, Vol={self.data.volume[0]:.0f})")
                self.order = self.sell()
                return
```

Explanation of the `SimpleDualMA` Class:

- **`params`:** A dictionary holding all tunable parameters for the strategy. This makes it easy to adjust them later for optimization.
- **`__init__(self)`:**
    - Initializes all the indicators (EMAs, ADX, Volume SMA, ATR) using the data feed (`self.datas[0]`).
    - Sets up `self.crossover` which is a `backtrader` built-in indicator that conveniently gives `1` for a fast-above-slow crossover and `-1` for a fast-below-slow crossover.
    - Initializes state variables like `self.order` (to track pending orders), `self.price_entry`, and `self.trailing_stop_price`.
- **`log(self, txt)`:** A utility function for printing messages with timestamps during the backtest.
- **`notify_order(self, order)`:** This method is called by `backtrader` whenever there's an update to an order's status.
    - It logs when orders are executed.
    - Crucially, upon a successful buy or sell order completion, it records the `order.executed.price` as `self.price_entry` and calculates the **initial `self.trailing_stop_price`** based on the entry price and the current ATR value multiplied by `self.p.stop_atr`.
- **`next(self)`:** This is the heart of the strategy, called for each new bar of data.
    1. **Pending Order Check:** If an order is already active (`self.order` is not `None`), it does nothing and waits for the order to be processed.
    2. **Position Management (Trailing Stop):**
        - If **long** (`self.position.size > 0`): It calculates a new potential stop-loss (`price - self.p.stop_atr * self.atr[0]`). If this new stop is higher than the current `self.trailing_stop_price`, the trailing stop is updated (it only moves up). If the current price drops below the `self.trailing_stop_price`, a `self.close()` order is issued.

    - If **short** (`self.position.size < 0`): Similar logic, but the stop moves down (`price + self.p.stop_atr * self.atr[0]`). If the new stop is lower, it's updated. If the price rises above the `self.trailing_stop_price`, the position is closed.
3. **New Entry Check:** If there's no open position:
    - It resets `trailing_stop_price` and `price_entry`.
    - It checks the **volume condition** (vol_ok): current volume must be `self.p.vol_mul` times greater than the `self.p.vol_sma` period SMA of volume.
    - It checks the **ADX condition** (adx_ok): the ADX value (`self.adx.adx[0]`) must be greater than or equal to `self.p.adx_limit`.
    - **Long Entry:** If `self.crossover[0] > 0` (fast EMA crossed above slow), and adx_ok, and vol_ok, a buy order (`self.buy()`) is placed.
    - **Short Entry:** If `self.crossover[0] < 0` (fast EMA crossed below slow), and adx_ok, and vol_ok, a sell order (`self.sell()`) is placed.
    - A `return` statement is used after placing an order to ensure no other logic fires on the same bar.

Backtesting Setup

The `if __name__ == "__main__":` block sets up and runs the backtest:

Python

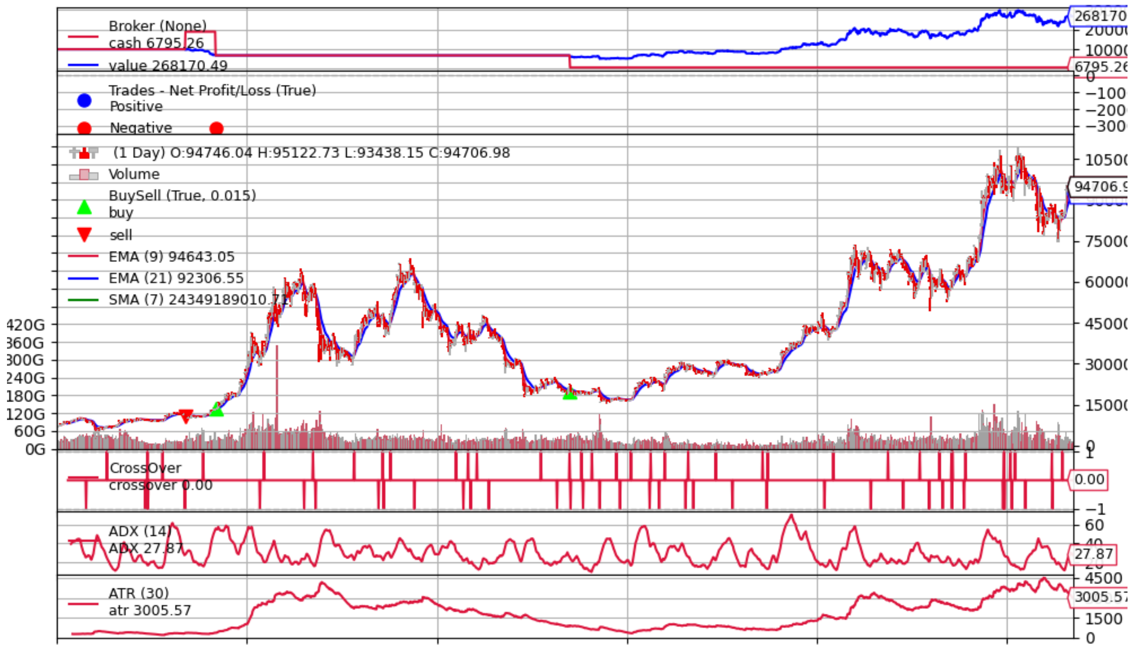
```
if __name__ == "__main__":
    cerebro = bt.Cerebro() # Create the backtesting engine
    cerebro.addstrategy(SimpleDualMA) # Add our strategy

    # Fetch daily BTC-USD data from yfinance
    # Using yf.download for more robust data fetching, or
yf.Ticker().history()
    print("Fetching BTC-USD data...")
    df = yf.Ticker("BTC-USD").history(start="2020-01-01", interval="1d")
    # Ensure column names match backtrader's expectations
    df.rename(columns={
        "Open": "open", "High": "high", "Low": "low",
        "Close": "close", "Volume": "volume"
    }, inplace=True)
    df["openinterest"] = 0 # Add a zero openinterest column as backtrader
requires it

    data = bt.feeds.PandasData(dataname=df) # Create a backtrader data feed
    cerebro.adddata(data) # Add data to Cerebro

    # Set initial cash, commission, and sizer
    cerebro.broker.setcash(100_000.00) # Initial capital
    cerebro.broker.setcommission(commission=0.001) # 0.1% commission per
trade
    cerebro.addsizer(bt.sizers.PercentSizer, percents=90) # Use 90% of
portfolio equity for each trade

    print("Starting Portfolio Value:
{:.2f}".format(cerebro.broker.getvalue()))
    results = cerebro.run() # Run the backtest
    print("Final Portfolio Value:
{:.2f}".format(cerebro.broker.getvalue()))

    # Plotting the results
    # For non-Jupyter environments, ensure a GUI backend for matplotlib is
available.
    # Example:
    # import matplotlib
    # matplotlib.use('TkAgg') # Or 'Qt5Agg', etc.
    try:
        # Set iplot=False for static plots, style for candlestick appearance
        cerebro.plot(iplot=False, style='candlestick', barup='green',
bardown='red')
    except Exception as e:
        print(f"Plotting error: {e}. If not in Jupyter, try
cerebro.plot(iplot=False). Ensure a GUI backend for matplotlib is set up
(e.g., TkAgg, Qt5Agg).")
        # Fallback to a simpler plot if the styled one fails
        # cerebro.plot(iplot=False)
```

Backtesting Setup Explanation:

1. **Cerebro Engine:** `cerebro = bt.Cerebro()` creates the main backtesting controller.
2. **Add Strategy:** `cerebro.addstrategy(SimpleDualMA)` adds our defined strategy.
3. **Data Fetching:**
    - `yf.Ticker("BTC-USD").history(start="2020-01-01", interval="1d")` fetches daily Bitcoin data from Yahoo Finance starting from January 1, 2020.
    - Column names are renamed to lowercase (open, `high`, `low`, `close`, `volume`) as expected by backtrader's `PandasData` feed.
    - An `openinterest` column is added and set to 0, as it's required by `backtrader`.
4. **Data Feed:** `data = bt.feeds.PandasData(dataname=df)` converts the pandas DataFrame into a format `backtrader` can use.
5. **Broker Simulation:**
    - `cerebro.broker.setcash(100_000.00)` sets the initial portfolio value.

- `cerebro.broker.setcommission(commission=0.001)` sets a 0.1% commission fee for each trade, simulating real-world costs.
- `cerebro.addsizer(bt.sizers.PercentSizer, percents=90)` tells backtrader to use 90% of the current portfolio equity for each trade. This is a form of position sizing.

6. **Run Backtest:** `results = cerebro.run()` executes the strategy over the historical data.
7. **Plotting:** `cerebro.plot(...)` generates a chart showing the price, trades, and potentially indicators.
   - `iplot=False` is generally recommended for scripts run outside of Jupyter notebooks to produce static plots.
   - `style='candlestick'` provides a familiar chart type.
   - A `try-except` block is included to handle potential plotting issues, especially in environments without a properly configured Matplotlib GUI backend.



###

Further Considerations and Potential Improvements

- **Parameter Optimization:** The chosen parameters (`fast`, `slow`, `adx_limit`, `vol_mul`, `stop_atr`, etc.) are just examples. These could be optimized using `backtrader`'s optimization capabilities or other techniques to find values that yield better historical performance (though be wary of overfitting).
- **Different Assets and Timeframes:** Test the strategy on other financial instruments and timeframes (e.g., hourly, 4-hour) to see how it performs under different market dynamics.

- **Risk Management:** While an ATR trailing stop is used, one could incorporate more sophisticated risk management, like limiting maximum drawdown or adjusting position size based on volatility or conviction.
- **Slippage and Fill Rates:** The current backtest assumes orders are filled at the close price of the bar where the signal occurs. Real-world trading involves slippage. `backtrader` can simulate this to some extent.
- **Walk-Forward Analysis:** To build more robust strategies, employ walk-forward optimization and testing, which helps to avoid overfitting to a specific historical period.
- **Statistical Analysis:** Use tools like `pyfolio` (via `backtrader`'s analyzers) to get a deeper statistical breakdown of the strategy's performance (Sharpe ratio, Sortino ratio, drawdown analysis, etc.).

Conclusion

This `SimpleDualMA` strategy provides a practical example of how multiple technical indicators can be combined to create a comprehensive trading system. By using `backtrader`, we can effectively implement, backtest, and visualize the performance of such strategies. The detailed logging, ATR-based trailing stop, and inclusion of trend strength and volume confirmation make it a more robust approach than a simple crossover system. Remember that historical performance is not indicative of future results, and any strategy should be thoroughly tested and understood before being deployed with real capital.

Can EMA, Accumulation Distribution Oscillator, and ADX Deliver Reliable Signals

How do you reliably catch the trend and avoid getting shredded by the chop? Trend following is a classic trading approach, but notoriously difficult in volatile markets like crypto where false signals abound.

Many traders rely on a single indicator, like a moving average, but quickly find it insufficient. Price might cross above an average, only to immediately reverse. This is where combining *complementary* indicators comes in. Can a multi-layered approach, using trend direction, momentum confirmation, and a chop filter, significantly improve our odds of identifying worthwhile trend signals in Bitcoin?

Let's explore a strategy combining three powerful technical tools:

1. **Exponential Moving Average (EMA):** To define the primary trend direction.
2. **Accumulation/Distribution (A/D) Oscillator:** To confirm momentum aligns with the trend.

3. **Average Directional Index (ADX):** To filter out signals during low-trend (choppy) conditions.1

The Strategy Logic

1. **Identify the Trend:** We use a 50-period EMA (Exponential Moving Average). Why EMA? It gives more weight to recent prices, making it slightly more responsive than a Simple Moving Average (SMA).2
    - **Uptrend:** Price is consistently trading *above* the 50-EMA.
    - **Downtrend:** Price is consistently trading *below* the 50-EMA.
2. **Confirm Momentum with A/D Oscillator:** Simply being above the EMA isn't enough. We need confirmation that buying or selling pressure supports the move. We use the **Chaikin Oscillator** (often referred to as an A/D Oscillator because it's derived from the Accumulation/Distribution Line).3 This indicator measures the momentum of the A/D line by comparing short-term (3-period) and long-term (10-period) EMAs of the A/D line itself.4
    - **Long Confirmation:** In an uptrend (Price > 50-EMA), we look for the Chaikin Oscillator to cross *above* its zero line.5 This suggests accumulation momentum is building, supporting the price trend.
    - **Short Confirmation:** In a downtrend (Price < 50-EMA), we look for the Chaikin Oscillator to cross *below* its zero line, indicating distribution momentum aligns with the falling price.6
    - *(Optional Sophistication):* For extra confirmation, we can require the oscillator's *slope* to be positive for longs (momentum increasing) and negative for shorts (momentum decreasing).7
3. **Filter the Chop with ADX:** Here's the crucial step to combat false signals. Choppy, range-bound markets are where basic trend-following systems often fail.8
The **Average Directional Index (ADX)** specifically measures trend *strength* (not direction).9 A low ADX value indicates a weak or non-existent trend.10
    - **The Filter:** We only consider entry signals (both long and short) if the ADX value is *above* a certain threshold, typically 20 or 25. If ADX is below this level, the market is deemed too choppy, and signals based on the EMA and A/D Oscillator are ignored. This helps us stay sidelined when clear directional momentum is lacking.

Putting it into Code (Python)

Using libraries like `yfinance` (to fetch data), `pandas_ta` (for indicators), and `matplotlib` (for plotting), we can implement this.11

First, fetch the data (e.g., for Bitcoin 'BTC-USD') and calculate the indicators:

Python

```
import yfinance as yf
import pandas_ta as ta
import pandas as pd

# Parameters
ticker = 'BTC-USD'
start_date = '2023-01-01' # Example start date
end_date = pd.to_datetime('today').strftime('%Y-%m-%d')
ema_length = 50
adosc_fast = 3
adosc_slow = 10
adx_length = 14
adx_threshold = 25 # Filter threshold
use_sophistication_layer = True # Optional slope filter
enable_adx_filter = True # Enable/disable ADX filter

# Fetch Data
data = yf.download(ticker, start=start_date, end=end_date, progress=False)
# (Include error handling and column checks as in the full script)

# Calculate Indicators
data[f'EMA_{ema_length}'] = ta.ema(data['Close'], length=ema_length)
data['AD_Osc'] = ta.adosc(data['High'], data['Low'], data['Close'],
data['Volume'], fast=adosc_fast, slow=adosc_slow)
data['AD_Osc_Slope'] = data['AD_Osc'].diff(1)
adx_df = ta.adx(data['High'], data['Low'], data['Close'], length=adx_length)
adx_col_name = f'ADX_{adx_length}' # e.g., 'ADX_14'
if adx_col_name in adx_df.columns:
    data['ADX'] = adx_df[adx_col_name]
else:
    # Handle cases where the column name might differ slightly or calculation
fails
    print(f"Warning: ADX column '{adx_col_name}' not found. Check ta.adx
output.")
    data['ADX'] = pd.NA # Assign NA if calculation fails

# Drop initial NaN rows
data.dropna(inplace=True)
```

Next, define the signal logic, incorporating all conditions:

Python

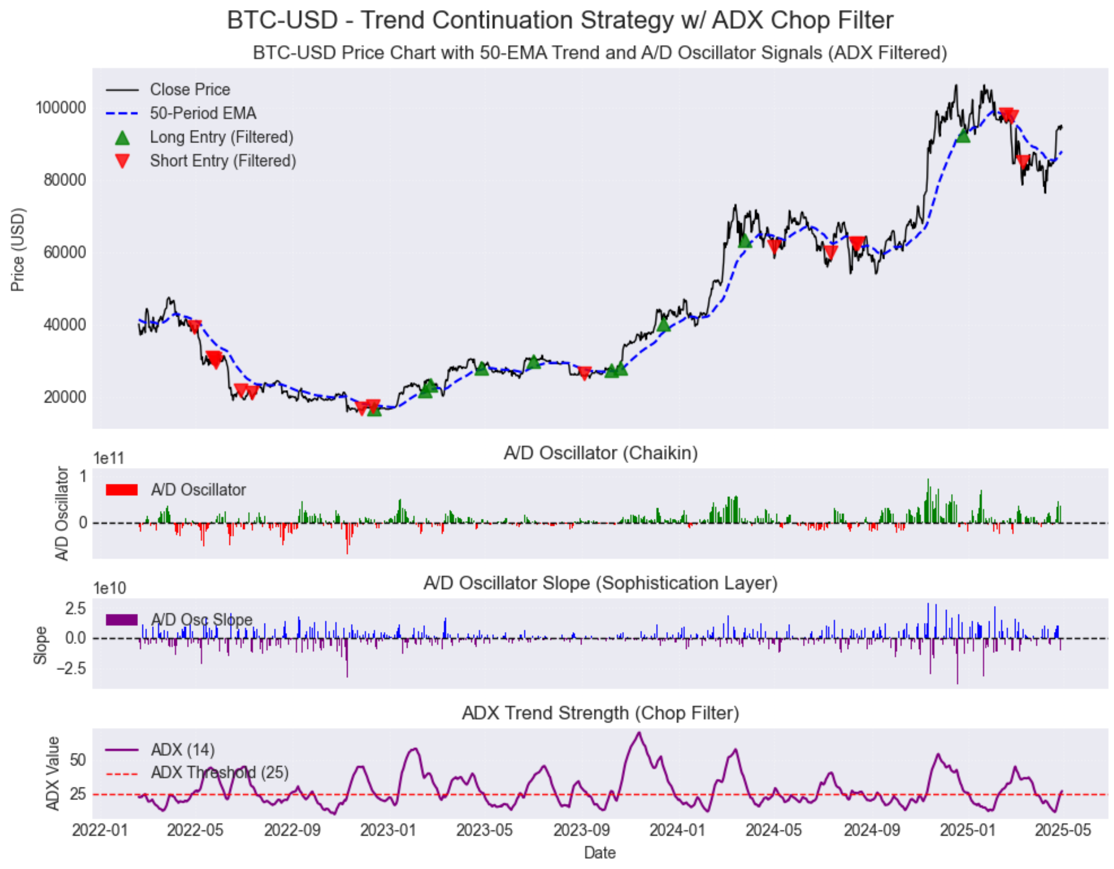
```
# Define Trend Condition based on ADX filter
is_trending = (data['ADX'].notna() & (data['ADX'] > adx_threshold)) if
enable_adx_filter else True

# --- Long Signal Logic ---
long_trend = data['Close'] > data[f'EMA_{ema_length}']
```

```
long_momentum_cross = (data['AD_Osc'] > 0) & (data['AD_Osc'].shift(1) <= 0)
long_slope_condition = (data['AD_Osc_Slope'] > 0) if use_sophistication_layer
else True

# Combine all long conditions
data['Long_Signal'] = long_trend & long_momentum_cross & long_slope_condition
& is_trending

# --- Short Signal Logic ---
short_trend = data['Close'] < data[f'EMA_{ema_length}']
short_momentum_cross = (data['AD_Osc'] < 0) & (data['AD_Osc'].shift(1) >= 0)
short_slope_condition = (data['AD_Osc_Slope'] < 0) if
use_sophistication_layer else True

# Combine all short conditions
data['Short_Signal'] = short_trend & short_momentum_cross &
short_slope_condition & is_trending
```

Visualizing the Strategy

A multi-panel chart helps understand how the indicators interact:

- **Panel 1:** Price, 50-EMA, and entry signal markers (e.g., green up arrows for long, red down arrows for short).
- **Panel 2:** Chaikin Oscillator with zero line. Observe crosses aligning with signals.
- **Panel 3 (Optional):** Oscillator slope, showing momentum changes.
- **Panel 4:** ADX indicator with the threshold line (e.g., at 25). Notice how signals only appear when ADX is above the threshold.



Pasted image 20250501132010.png

Pasted image 20250501132010.png

The Verdict: Does It Deliver?

Combining these indicators offers several potential advantages over simpler strategies:

1. **Trend Confirmation:** The EMA sets the stage, ensuring we're looking in the right direction.
2. **Momentum Validation:** The A/D (Chaikin) Oscillator confirms underlying buying/selling pressure, reducing entries based solely on price crossing an average.12
3. **Chop Reduction:** The ADX filter is key. By demanding a minimum level of trend strength, it aims to keep traders out during directionless, whipsaw-prone periods, potentially saving capital and reducing frustration. Search results confirm ADX is a reliable gauge for this.

However, no system is foolproof.

- **Lag:** All these indicators are lagging; they are based on past price action. They won't catch the exact top or bottom.

- **Parameter Sensitivity:** The chosen lengths (50 for EMA, 14 for ADX, etc.) and the ADX threshold (25) are common but not universally optimal. They might need tuning for different assets or timeframes.
- **Whipsaws Still Possible:** Strong, brief counter-trend moves or sudden volatility spikes can still trigger false signals even with filters.
- **Backtesting is Crucial:** This strategy, like any other, *must* be rigorously backtested on historical data for the specific asset and timeframe you intend to trade. Analyze performance metrics (win rate, profit factor, drawdown) before risking real capital.
- **Risk Management is Non-Negotiable:** Always use stop-losses and appropriate position sizing. No indicator combination can replace sound risk management.

Conclusion

Can this combination of EMA, A/D Oscillator, and ADX guarantee profitable Bitcoin trend signals? No. Can it potentially provide *more reliable* signals than simpler methods by filtering out noise and demanding confirmation? Yes, that's the objective.

By layering trend direction, momentum confirmation, and a trend-strength filter, this strategy attempts a more robust approach to navigating Bitcoin's challenging trends. It provides a logical framework, but its real-world effectiveness hinges on thorough testing, careful parameter tuning, and disciplined execution with robust risk management. It's a tool, not a magic bullet, but potentially a valuable one in a trader's arsenal.

Can Kalman Filters Improve Your Trading Signals

Kalman filters offer an advanced technique for signal processing, often used to extract underlying states, like trend or velocity, from noisy data. Applying this to financial markets allows us to estimate price movements potentially more adaptively than standard indicators.

This article details a `backtrader` strategy using a Kalman filter (via a custom `KalmanFilterIndicator`) to estimate price velocity. The strategy enters trades based on the *sign* of this estimated velocity and relies exclusively on a trailing stop-loss for exits.

Strategy Logic Overview:

1. **Filtering:** A `KalmanFilterIndicator` estimates the underlying price and its velocity based on closing prices.
2. **Entry Signal:** Enter long if the estimated velocity turns positive (`> 0`). Enter short if the velocity turns negative (`< 0`). Entries only happen when flat.
3. **Exit Signal:** A percentage-based trailing stop-loss (`trail_percent`) manages exits. Once a position is open, the Kalman velocity sign is ignored for exiting.

The Supporting Indicator: `KalmanFilterIndicator`

(The code for `KalmanFilterIndicator` as you provided it is assumed here. It calculates `kf_price` and `kf_velocity` and is set to plot on the main chart panel using `plotinfo = dict(subplot=False)`).

The Strategy Class: `KalmanFilterTrendWithTrail`

This class orchestrates the trading logic using the indicator's output.

1. Parameters (`params`)

These allow configuration of the filter and the trailing stop.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    params = (
        # Parameters passed to the Kalman Filter Indicator
        ('process_noise', 1e-5),    # Filter parameter: Assumed noise in the
price model (Q)
        ('measurement_noise', 1e-1),# Filter parameter: Assumed noise in the
price data (R)

        # Strategy-specific parameter
        ('trail_percent', 0.05),    # Trailing stop loss percentage (e.g.,
0.05 = 5%)
        ('printlog', True),         # Enable logging output
    )
```

- `process_noise` & `measurement_noise`: Control the Kalman filter's behavior. Finding good values requires testing and optimization specific to the asset and timeframe.
- `trail_percent`: Determines the percentage drawdown from the peak price (for longs) or trough price (for shorts) that triggers the stop-loss.

2. Initialization (`__init__`)

Sets up the strategy by creating the indicator instance.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    def __init__(self):
        # Instantiate the Kalman Filter Indicator, passing relevant
parameters
        self.kf = KalmanFilterIndicator(
            process_noise=self.p.process_noise,
            measurement_noise=self.p.measurement_noise
        )

        # Create convenient references to the indicator's output lines
        self.kf_price = self.kf.lines.kf_price
        self.kf_velocity = self.kf.lines.kf_velocity

        # Initialize order trackers
        self.order = None # Tracks pending entry orders
        self.stop_order = None # Tracks pending stop orders

        if self.params.printlog:
            # Log the parameters being used
            print(f"Strategy Parameters: Process 
Noise={self.params.process_noise}, "
                  f"Measurement Noise={self.params.measurement_noise}, "
                  f"Trail Percent={self.params.trail_percent * 100:.2f}%")
```

3. Entry Logic (next)

The next method contains the core logic executed on each bar. For entries, it checks the position status and the Kalman velocity sign.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    def next(self):
        # If an entry order is pending, do nothing
        if self.order:
            return

        # Get the estimated velocity from the indicator
        # Need to check length because indicator might need warmup
        if len(self.kf_velocity) == 0:
             return # Indicator not ready yet

        estimated_velocity = self.kf_velocity[0]
        current_position_size = self.position.size
        current_close = self.data.close[0] # For logging

        # --- Trading Logic ---
        # Only evaluate entries if FLAT
        if current_position_size == 0:
            if self.stop_order: # Safety check - cancel any stray stop orders 
if flat
                self.log("Warning: Position flat but stop order exists. 
Cancelling.", doprint=True)
                self.cancel(self.stop_order)
                self.stop_order = None

            # --- Entry Signal Check ---
            if estimated_velocity > 0:
                 # Positive velocity -> Go Long
                 self.log(f'BUY CREATE (KF Vel > 0), 
Close={current_close:.2f}, KF Vel={estimated_velocity:.4f}', doprint=True)
                 self.order = self.buy() # Place buy order and track it
            elif estimated_velocity < 0:
                 # Negative velocity -> Go Short
                 self.log(f'SELL CREATE (KF Vel < 0 - Short Entry),
Close={current_close:.2f}, KF Vel={estimated_velocity:.4f}', doprint=True)
                 self.order = self.sell() # Place sell order and track it
        else:
             # If already in a position, do nothing here.
             # The trailing stop placed via notify_order handles the exit.
             pass
```

This logic is straightforward: if flat, buy on positive velocity, sell on negative velocity. If already in a position, it relies entirely on the trailing stop.

4. Exit Logic (`notify_order`)

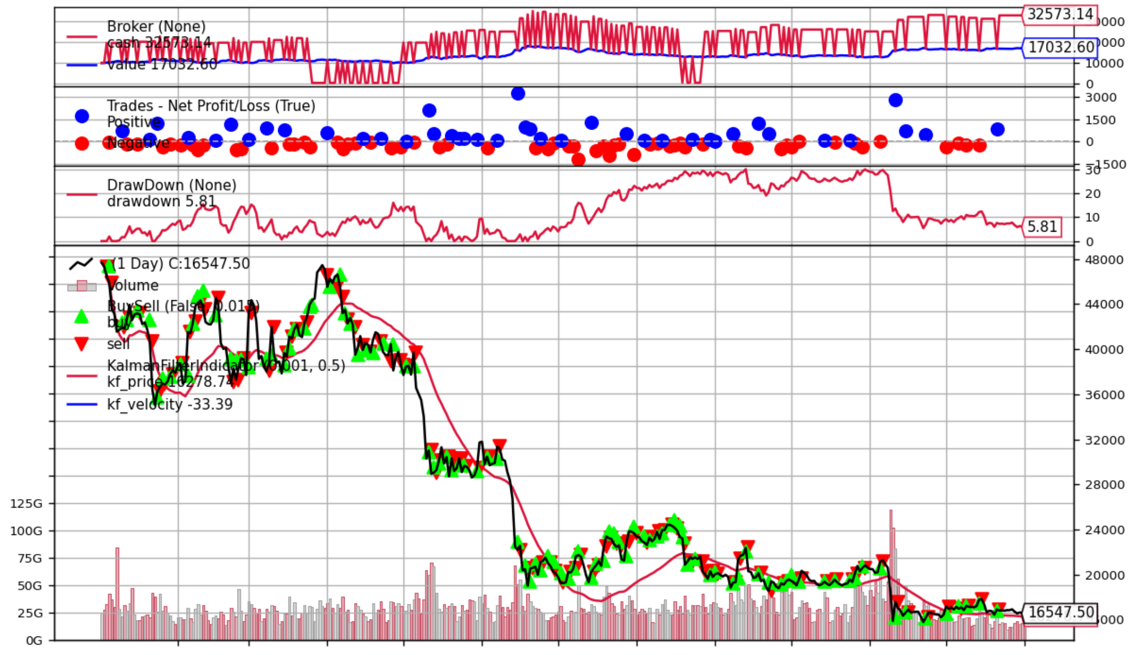
Exits are handled by placing a `StopTrail` order immediately after an entry order is successfully filled. This logic resides within the `notify_order` method.

```
# --- Inside KalmanFilterTrendWithTrail class ---
    def notify_order(self, order):
        # (Initial checks for Submitted/Accepted status omitted for brevity)
        ...
        if order.status == order.Completed:
            # Check if it's the ENTRY order we were waiting for
            if self.order and order.ref == self.order.ref:
                entry_type = "BUY" if order.isbuy() else "SELL"
                exit_func = self.sell if order.isbuy() else self.buy #
Determine exit order type

                # Log entry execution (code omitted for brevity)
                ...

                # Place the TRAILING STOP order if trail_percent is valid
                if self.p.trail_percent and self.p.trail_percent > 0.0:
                    self.stop_order = exit_func(exectype=bt.Order.StopTrail,

trailpercent=self.p.trail_percent)
                    self.log(f'Trailing Stop Placed for {entry_type} order
ref {self.stop_order.ref} at {self.p.trail_percent * 100:.2f}% trail',
doprint=True)
                else:
                     self.log(f'No Trailing Stop Placed
(trail_percent={self.p.trail_percent})', doprint=True)

                self.order = None # Reset entry order tracker

            # Check if it's the STOP order that completed
            elif self.stop_order and order.ref == self.stop_order.ref:
                # Log stop execution (code omitted for brevity)
                ...
                self.stop_order = None # Reset stop order tracker
                self.order = None # Reset entry tracker too
        # Handle Failed orders (code omitted for brevity)
        ...
```

This ensures that as soon as an entry trade is confirmed, the trailing stop is activated to manage the exit.

Running the Backtest

The `__main__` block in your provided code sets up `cerebro`, fetches data (BTC-USD, 2021-2023), configures the broker/sizer/analyzers, and runs the strategy with specific parameters (`process_noise=0.001`, `measurement_noise=0.5`, `trail_percent=0.02`). It then prints performance metrics and attempts to plot the results, including the Kalman Filter price overlayed on the main chart.



Pasted image 20250424185531.png

Pasted image 20250424185531.png

Tuning and Considerations

- **Parameter Sensitivity:** This strategy's performance is highly dependent on the `process_noise`, `measurement_noise`, and `trail_percent` parameters. The values used (`0.001`, `0.5`, `0.02`) are specific examples and likely require optimization for different market conditions or assets.
- **Whipsaws:** Using only the sign of the velocity can lead to frequent entries and exits (whipsaws) in non-trending or choppy markets, potentially hurting performance even with a trailing stop.

- **Model Limitations:** The constant velocity model is a simplification. Real market dynamics are more complex.
- **Optimization:** Thorough backtesting and optimization across various parameter combinations are essential to evaluate the strategy's potential robustness.

Conclusion

This `backtrader` strategy demonstrates using a Kalman filter's velocity estimate for trend direction signals, combined with a trailing stop for risk management. While conceptually interesting, its practical effectiveness hinges critically on careful parameter tuning and understanding its limitations, particularly the sensitivity to noise when using only the velocity sign for entries.

Can the Hurst Exponent Reliably Identify Trends

1. Regime Detection via the Hurst Exponent

1.1. Concept

The **Hurst exponent** (H) quantifies the long-term memory or persistence of a time series.

- (H=0.5) indicates a pure random walk (no autocorrelation).
- (H>0.5) signals persistence (trending behavior).
- (H<0.5) signals anti-persistence (mean-reverting behavior).

1.2. R/S Analysis

One classic method to estimate (H) is **Rescaled-Range Analysis**. Given a time series ({X_t}), for a window of length (n):

1. Compute the mean: [{X}*n ;=;* {t=1}^n X_t.]
2. Form the mean-adjusted cumulative deviate series: [Y_k ;=; _{t=1}^k (X_t - {X}_n), k = 1,2,,n]
3. Compute the range: [R(n) ;=; *{1 k n} Y_k ;-;* {1 k n} Y_k]
4. Compute the standard deviation: [S(n) ;=;]
5. Form the rescaled range: []
6. Repeat for multiple window sizes (n), then fit a power-law: [E!;; n^H]

   Taking logs and performing a linear regression:[(R(n)/S(n)) ;=; H,(n) ;+;]

2. Trend Signal: SMA Crossover

2.1. Simple Moving Averages

Define two SMAs on price (P_t):

[*{,t}* ={i=0}^{N_f-1}P_{t-i}, *{,t}* ={i=0}^{N_s-1}P_{t-i}]

with (N_f < N_s) (e.g. 20 vs 50, or 30 vs 50).

2.2. Crossover Signal

The **crossover indicator** is

[_t =

]

3. Filtering Trends using Hurst Exponent

Can we improve trend-following strategies by only trading when the market exhibits strong trending characteristics? The Hurst exponent (H) offers a potential way to quantify this. It measures the long-term memory or persistence of a time series:

- **H > 0.5:** Suggests persistence (trending behavior). Trends are likely to continue.
- **H < 0.5:** Suggests anti-persistence (mean-reverting behavior). Trends are likely to reverse.
- **H ≈ 0.5:** Suggests a random walk (no memory).

This article explores a `backtrader` strategy that calculates the Hurst exponent and uses it as a regime filter. It only activates a simple trend-following system (an SMA Crossover) when H indicates a trending regime (H > threshold). Exits are managed using a trailing stop-loss.

Strategy Logic Overview:

1. **Hurst Calculation:** A custom `HurstExponentIndicator` calculates H over a rolling window (using the `hurst` library).
2. **Regime Filter:** The strategy checks if the current H is above a defined `hurst_threshold` (e.g., 0.7 in the example run).
3. **Entry Signal:** If the market is deemed "trending" (H > threshold) *and* the strategy is flat, it checks for an SMA crossover signal to enter long (fast SMA > slow SMA) or short (fast SMA < slow SMA).
4. **Exit Signal:** Once in a position, a percentage-based trailing stop-loss (`trail_percent`) handles the exit exclusively.

The Supporting Indicator: `HurstExponentIndicator`

(The code for `HurstExponentIndicator`, corrected for the `AttributeError`, is assumed here. It calculates `hurst` over a window and plots it in a separate panel).

The Strategy Class: `HurstFilteredTrendStrategy`

Let's examine the strategy class implementing this logic.

1. Parameters (`params`)

These allow configuration of the Hurst filter, the trend system, and the exit mechanism.

Python

```
# --- Inside HurstFilteredTrendStrategy class ---
    params = (
        # Hurst parameters
        ('hurst_window', 100),       # Window for Hurst calculation
        ('hurst_threshold', 0.55),   # Hurst value above which trend signals
are enabled

        # Trend parameters (SMA Crossover example)
        ('sma_fast', 20),            # Fast SMA period
        ('sma_slow', 50),            # Slow SMA period

        # Exit parameter
        ('trail_percent', 0.05),     # Trailing stop percentage

        # Logging
        ('printlog', True),
    )
```

- `hurst_window` & `hurst_threshold`: Define the filter. Longer windows give smoother but laggier H values. The threshold determines how strong the persistence must be. (Note: Your example run used `hurst_window=180`, `hurst_threshold=0.7`).
- `sma_fast` & `sma_slow`: Define the periods for the simple moving average crossover signal. (Note: Your example run used `sma_fast=30`, `sma_slow=50`).
- `trail_percent`: The trailing stop percentage. (Note: Your example run used `trail_percent=0.05`).
- **Tuning is crucial for all these parameters.**

2. Initialization (`__init__`)

Sets up the required indicators.

Python

```
# --- Inside HurstFilteredTrendStrategy class ---
    def __init__(self):
        # Instantiate the Hurst Exponent Indicator
        self.hurst = HurstExponentIndicator(
            self.data.close, # Pass the close price series
            window=self.p.hurst_window
        )

        # Instantiate the trend indicators (SMA Crossover)
        sma_fast = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.p.sma_fast)
        sma_slow = bt.indicators.SimpleMovingAverage(self.data.close,
period=self.p.sma_slow)
        self.sma_cross = bt.indicators.CrossOver(sma_fast, sma_slow) # 1 for
cross up, -1 for cross down

        # Order trackers
        self.order = None
        self.stop_order = None

        # (Parameter logging code omitted for brevity)
        ...
```

3. Core Logic (next)

This method executes on each bar, checks the regime, and looks for entry signals if applicable.

Python

```
# --- Inside HurstFilteredTrendStrategy class ---
    def next(self):
        # Check if indicators/orders are ready
        if self.order or len(self.hurst) == 0 or len(self.sma_cross) == 0:
            return

        current_hurst = self.hurst.hurst[0]
        current_cross = self.sma_cross[0]
        current_close = self.data.close[0]
        current_position_size = self.position.size

        # Check if Hurst calculation failed
        if np.isnan(current_hurst):
            return # Skip bar if Hurst is invalid

        # --- Regime Filter ---
        # Check if Hurst value is above the threshold
        is_trending_regime = current_hurst > self.p.hurst_threshold

        # --- Trading Logic ---
        if current_position_size == 0: # If FLAT
            # (Safety check for stray stop orders omitted)
            ...
            # --- Entry Check ---
            # Only consider entries if the regime filter allows it
            if is_trending_regime:
                if current_cross > 0: # Buy signal: Fast SMA crosses above
Slow SMA
                    self.log(f'BUY CREATE (Hurst Trend Regime & SMA Cross >
0), H={current_hurst:.3f}, Close={current_close:.2f}', doprint=True)
                    self.order = self.buy()
                elif current_cross < 0: # Sell signal: Fast SMA crosses below
Slow SMA
                    self.log(f'SELL CREATE (Hurst Trend Regime & SMA Cross <
0), H={current_hurst:.3f}, Close={current_close:.2f}', doprint=True)
                    self.order = self.sell()
            # If not in a trending regime, do nothing.

        else: # If IN A POSITION
            # Exits are handled by the trailing stop, so no action needed
here.
            pass
```

The key is the `if is_trending_regime:` check, which gates the SMA crossover signal.

4. Exit Logic (`notify_order`)

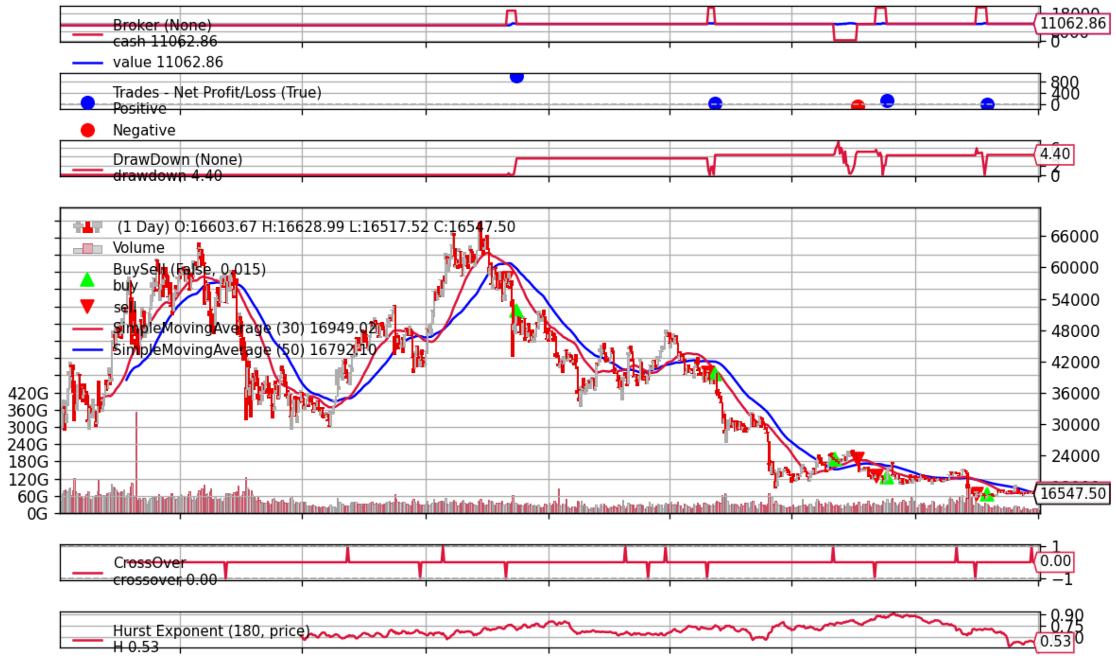
Exits are managed by placing a `StopTrail` order after an entry fills.

Python

```
# --- Inside HurstFilteredTrendStrategy class ---
    def notify_order(self, order):
        # (Initial checks for Submitted/Accepted omitted)
        ...
        if order.status == order.Completed:
            # If it was our entry order completing...
            if self.order and order.ref == self.order.ref:
                entry_type = "BUY" if order.isbuy() else "SELL"
                exit_func = self.sell if order.isbuy() else self.buy #
Function to place opposite order

                # (Log entry execution omitted)
                ...

                # Place the TRAILING STOP order
                if self.p.trail_percent and self.p.trail_percent > 0.0:
                    self.stop_order = exit_func(exectype=bt.Order.StopTrail,

trailpercent=self.p.trail_percent)
                    # (Log stop placement omitted)
                    ...
                self.order = None # Reset entry order tracker

            # If it was our stop order completing...
            elif self.stop_order and order.ref == self.stop_order.ref:
                 # (Log stop execution omitted)
                 ...
                 self.stop_order = None # Reset stop order tracker
        # (Handle Failed orders omitted)
        ...
```

Performance Example & Considerations

Running this strategy (with parameters `hurst_window=180`, `hurst_threshold=0.7`, `sma_fast=30`, `sma_slow=50`, `trail_percent=0.05` on BTC-USD from 2021-01-01 to 2023-01-01, as per your example setup) yielded the following results in one test:



Pasted image 20250424193637.png

Pasted image 20250424193637.png

```
Starting Portfolio Value: 10,000.00

2021-11-29 SELL CREATE (Hurst Trend Regime & SMA Cross < 0), H=0.740,
Close=57806.57
2021-11-30 SELL EXECUTED @ 57830.11, Size: -0.1643, Cost: -9503.87, Comm:
9.50
2021-11-30 Trailing Stop Placed for SELL order ref 8475 at 5.00% trail
2021-12-07 STOP BUY (Cover) EXECUTED @ 51660.74, Size: 0.1643, Cost: -
```

```
9503.87, Comm: 8.49
2021-12-07 OPERATION PROFIT, GROSS 1013.88, NET 995.89
2022-04-28 SELL CREATE (Hurst Trend Regime & SMA Cross < 0), H=0.712,
Close=39773.83
2022-04-29 SELL EXECUTED @ 39768.62, Size: -0.2626, Cost: -10444.73, Comm:
10.44
2022-04-29 Trailing Stop Placed for SELL order ref 8477 at 5.00% trail
2022-05-04 STOP BUY (Cover) EXECUTED @ 39600.62, Size: 0.2626, Cost: -
10444.73, Comm: 10.40
2022-05-04 OPERATION PROFIT, GROSS 44.12, NET 23.28
2022-07-31 BUY CREATE (Hurst Trend Regime & SMA Cross > 0), H=0.771,
Close=23336.90
2022-08-01 BUY EXECUTED @ 23336.72, Size: 0.4486, Cost: 10468.13, Comm: 10.47
2022-08-01 Trailing Stop Placed for BUY order ref 8479 at 5.00% trail
2022-08-18 STOP SELL (Exit Long) EXECUTED @ 23202.86, Size: -0.4486, Cost:
10468.13, Comm: 10.41
2022-08-18 OPERATION PROFIT, GROSS -60.04, NET -80.92
2022-08-31 SELL CREATE (Hurst Trend Regime & SMA Cross < 0), H=0.830,
Close=20049.76
2022-09-01 SELL EXECUTED @ 20050.50, Size: -0.5183, Cost: -10391.72, Comm:
10.39
2022-09-01 Trailing Stop Placed for SELL order ref 8481 at 5.00% trail
2022-09-09 STOP BUY (Cover) EXECUTED @ 19779.55, Size: 0.5183, Cost: -
10391.72, Comm: 10.25
2022-09-09 OPERATION PROFIT, GROSS 140.43, NET 119.78
2022-11-14 SELL CREATE (Hurst Trend Regime & SMA Cross < 0), H=0.702,
Close=16618.20
2022-11-15 SELL EXECUTED @ 16617.48, Size: -0.6321, Cost: -10504.68, Comm:
10.50
2022-11-15 Trailing Stop Placed for SELL order ref 8483 at 5.00% trail
2022-11-23 STOP BUY (Cover) EXECUTED @ 16576.65, Size: 0.6321, Cost: -
10504.68, Comm: 10.48
2022-11-23 OPERATION PROFIT, GROSS 25.81, NET 4.83
Final Portfolio Value:  11,062.86
Total Return: 10.63%

--- Performance Analysis ---
Sharpe Ratio: 0.028
Max Drawdown: 7.23%
Max Money Drawdown: 832.72
System Quality Number (SQN): 1.20
SQN Trades: 5

--- Trade Analysis ---
Total Closed Trades: 5
Total Open Trades: 0

Win Rate: 80.00% (4 wins)
Loss Rate: 20.00% (1 losses)
```

```
Total Net Profit/Loss: 1,062.86
Average Net Profit/Loss per Trade: 212.57

Average Winning Trade: 285.94
Average Losing Trade: -80.92
Max Winning Trade: 995.89
Max Losing Trade: -80.92

Profit Factor: 14.13

Average Bars in Trade: 9.00
Max Bars in Winning Trade: 8
Max Bars in Losing Trade: 17
```

Interpretation: In this specific run, the Hurst filter drastically reduced the number of trades and helped achieve excellent drawdown control compared to unfiltered strategies. However, the risk-adjusted returns (Sharpe) and system quality (SQN) remained very low. The very short average trade duration suggests the 5% trailing stop might have been too tight, cutting off winners before they could significantly contribute to profit, thus limiting the overall return despite the decent win rate and profit factor.

Conclusion:

Using the Hurst exponent as a regime filter shows potential for improving selectivity and managing risk in trend-following systems. This `backtrader` example provides a framework. However, the results highlight the critical need for extensive **parameter tuning** (`hurst_window`, `hurst_threshold`, trend indicator periods, `trail_percent`) through optimization. Finding the right balance to control risk while allowing profitable trends to run is key to developing this concept into a potentially viable strategy.

Can This Adaptive Average Boost Your Strategy Full VAMA and Backtrader Example

Backtrader is an exceptional Python framework for backtesting trading strategies, offering powerful building blocks. While its built-in indicators and functions cover many scenarios, true power comes from implementing your own custom logic – both in indicators and strategy rules, including vital risk management techniques.

This article presents a complete, runnable Backtrader example. We will:

1. Build a custom **Volatility-Adjusted Moving Average (VAMA)** indicator that adapts to market conditions.
2. Implement a **trading strategy** using a crossover between our custom VAMA and a standard SMA.

3. Integrate a crucial risk management tool: a **trailing stop loss**.
4. Set up and run the **full backtest** using data downloaded via `yfinance`.

This example showcases how to combine custom analytics with practical strategy execution and risk control, based on concepts detailed in **“Backtrader Essentials: Building Successful Strategies with Python”**.

Part 1: The Custom Indicator - Volatility-Adjusted Moving Average (VAMA)

First, we need our adaptive indicator. The VAMA adjusts its sensitivity based on recent market volatility (measured by Standard Deviation). It aims to be faster in volatile markets and smoother in calm ones. Here is the complete code for the `VolatilityAdjustedMovingAverage` indicator class:

Python

```python
import backtrader as bt
import math
# Ensure yfinance and matplotlib are installed: pip install yfinance
matplotlib
import yfinance as yf
import matplotlib.pyplot as plt

class VolatilityAdjustedMovingAverage(bt.Indicator):
    """
    Volatility Adjusted Moving Average (VAMA / VIDYA)
    Adapts EMA smoothing based on the ratio of current to average volatility.
    """
    lines = ('vama',)
    params = (
        ('period', 20),       # Base EMA period
        ('vol_period', 9),    # Volatility calculation period (StdDev)
        ('min_alpha_ratio', 0.1), # Floor for volatility ratio adjustment
        ('max_alpha_ratio', 2.0), # Ceiling for volatility ratio adjustment
    )

    def __init__(self):
        if self.p.vol_period <= 1:
             raise ValueError("vol_period must be greater than 1")
        if self.p.period <= 1:
             raise ValueError("period must be greater than 1")

        self.base_alpha = 2.0 / (self.p.period + 1.0)
        self.vol = bt.indicators.StandardDeviation(self.data.close,
period=self.p.vol_period)
        self.avg_vol = bt.indicators.SimpleMovingAverage(self.vol,
period=self.p.period)
        # Set minimum period based on nested indicators' requirements
        self.addminperiod(self.p.vol_period + self.p.period -1)

    def next(self):
        current_close = self.data.close[0]
        current_vol = self.vol[0]
        current_avg_vol = self.avg_vol[0]

        if current_avg_vol is not None and current_avg_vol != 0 and not
math.isnan(current_avg_vol):
            vol_ratio = current_vol / current_avg_vol
        else:
            vol_ratio = 1.0 # Neutral ratio

        # Clamp ratio and calculate adjusted alpha
        vol_ratio = max(self.p.min_alpha_ratio, min(vol_ratio,
self.p.max_alpha_ratio))
        adjusted_alpha = self.base_alpha * vol_ratio
        adjusted_alpha = max(1e-9, min(adjusted_alpha, 1.0)) # Ensure valid
alpha

        # Iterative VAMA calculation using adjusted alpha
        if len(self) > 1 and not math.isnan(self.lines.vama[-1]):
             prev_vama = self.lines.vama[-1]
             self.lines.vama[0] = (adjusted_alpha * current_close) + ((1 -
adjusted_alpha) * prev_vama)
        else:
             self.lines.vama[0] = current_close # Initialize
```

Key Points:

- It uses internal `StandardDeviation` and `SimpleMovingAverage` indicators.
- `addminperiod` ensures calculations wait for sufficient data.
- The `next` method performs the bar-by-bar adaptive EMA calculation, crucially depending on the previous VAMA value (`self.lines.vama[-1]`).

Part 2: The Strategy - VAMA Crossover with Trailing Stop

Now, we build a strategy (`VamaStrategy`) that utilizes our custom indicator. This strategy enters long when the faster VAMA crosses above a slower standard SMA and includes a trailing stop loss for risk management.

Python

```
class VamaStrategy(bt.Strategy):
    params = (
        ('vama_period', 20),       # Passed to VAMA indicator
        ('vama_vol_period', 9),    # Passed to VAMA indicator
        ('sma_period', 50),        # Period for the slow SMA
        ('trail_perc', None),     # Trailing stop percentage (e.g., 0.05).
None to disable.
    )

    def __init__(self):
        # Instantiate the custom VAMA indicator
        self.vama = VolatilityAdjustedMovingAverage(
            period=self.p.vama_period,
            vol_period=self.p.vama_vol_period
        )
        # Instantiate the standard SMA
        self.sma = bt.indicators.SimpleMovingAverage(
            self.data.close, period=self.p.sma_period
        )
        # Instantiate the Crossover detector
        self.crossover = bt.indicators.CrossOver(self.vama, self.sma)

        # Order tracking
        self.order = None

    def next(self):
        # Prevent placing new orders if one is already pending
        if self.order:
            return

        # Entry Logic: No position currently open
        if not self.position:
            # Buy signal: VAMA crosses above SMA
            if self.crossover > 0:
                print(f'{self.data.datetime.date(0)}: BUY SIGNAL - VAMA
{self.vama[0]:.2f} crossed above SMA {self.sma[0]:.2f}')
                # Place the market buy order
                self.order = self.buy()

                # --- Add Trailing Stop Loss ---
                # Check if trailing stop is enabled in parameters
                if self.p.trail_perc is not None and self.p.trail_perc > 0.0:
                    # Place a sell stop order that trails the price
                    self.sell(exectype=bt.Order.StopTrail,
                              trailpercent=self.p.trail_perc)
                    print(f'{self.data.datetime.date(0)}:    Trailing Stop
set at {self.p.trail_perc*100:.2f}%')
                # --- ---

        # Exit Logic: Position currently open
        else:
            # Sell signal: VAMA crosses below SMA
            if self.crossover < 0:
                print(f'{self.data.datetime.date(0)}: SELL SIGNAL (Crossover)
- VAMA {self.vama[0]:.2f} crossed below SMA {self.sma[0]:.2f}')
                # Close the position; this should also cancel the associated
trailing stop
                self.order = self.close()

    def notify_order(self, order):
        # Handle order status notifications
        if order.status in [order.Submitted, order.Accepted]:
            # Order is pending, nothing to do for basic logic
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                print(f'{self.data.datetime.date(0)}: BUY EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}, Cost:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
            elif order.issell():
                print(f'{self.data.datetime.date(0)}: SELL EXECUTED, Price:
{order.executed.price:.2f}, Size: {order.executed.size}, Cost:
{order.executed.value:.2f}, Comm: {order.executed.comm:.2f}')
                # Identify if the sell was due to the trailing stop
                if order.exectype == bt.Order.StopTrail:
                    print(f'{self.data.datetime.date(0)}: --- Trailing Stop
Loss Triggered ---')

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            print(f'{self.data.datetime.date(0)}: Order
Canceled/Margin/Rejected - Status: {order.getstatusname()}')

        # Reset order tracker allows checking signals again after order
resolution
        self.order = None
```

Strategy Highlights:

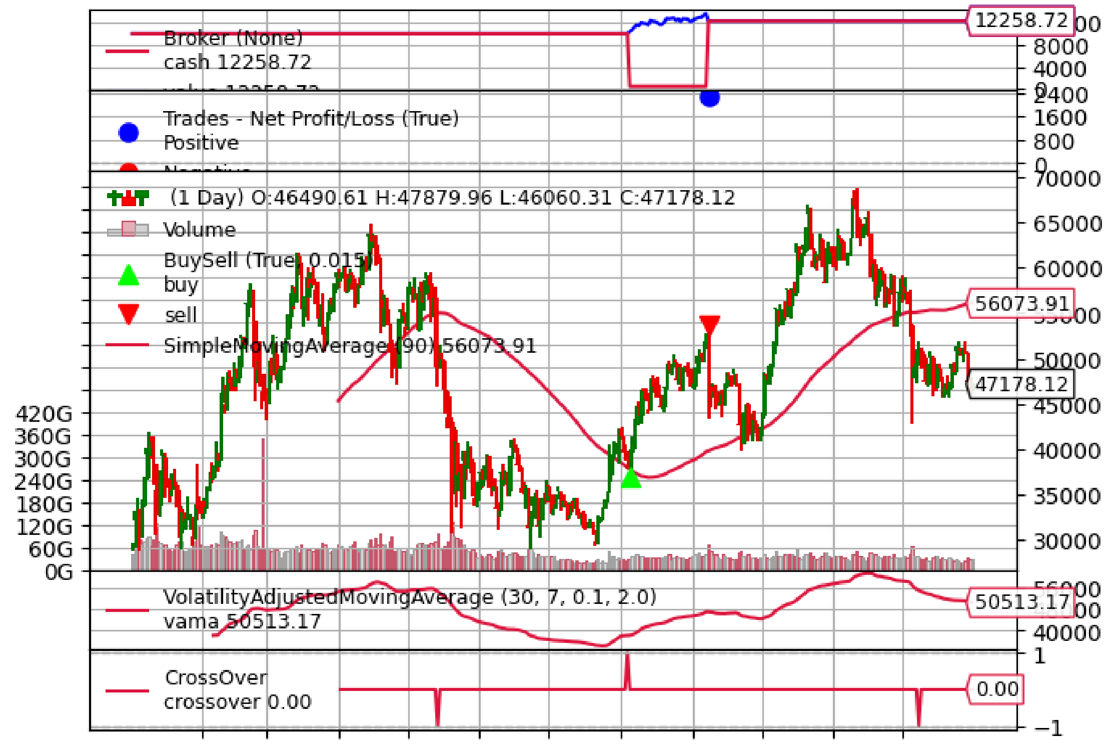
- **Indicators:** Uses the custom `VAMA`, a standard `SMA`, and `CrossOver`.
- **Entry:** Buys when VAMA crosses above SMA (`self.crossover > 0`).
- **Trailing Stop:** Immediately after buying, places a sell order (`exectype=bt.Order.StopTrail`) if `trail_perc` parameter is set. This order automatically trails the price upwards.
- **Exit:** Closes the position (`self.close()`) if the VAMA crosses back below the SMA (`self.crossover < 0`). This signal takes precedence if the trailing stop hasn't been hit yet. `self.close()` should cancel the pending trailing stop.
- **Risk Management:** The trailing stop provides downside protection that adapts as price moves favorably.

Part 3: Putting It All Together - The Backtest Setup

Finally, we need the main script block (`if __name__ == '__main__':`) to set up the Backtrader engine (Cerebro), load data, configure the test, run it, and plot the results.

Python

```
# --- Cerebro Setup ---
if __name__ == '__main__':
    # Create a Cerebro entity
    cerebro = bt.Cerebro()

    # --- Add data feed ---
    print("Downloading Data...")
    # Define parameters for data download
    ticker = 'BTC-USD'
    start_date = '2021-01-01'
    end_date = '2021-12-31' # Adjust as needed
    # Download data using yfinance
    data_df = yf.download(ticker, start=start_date, end=end_date,
progress=False)
    # Ensure the multi-level column index from yfinance is removed if present
(older versions)
    if isinstance(data_df.columns, pd.MultiIndex):
         data_df.columns = data_df.columns.droplevel(1)

    if data_df.empty:
        raise ValueError(f"Data download for {ticker} failed or returned
empty DataFrame.")

    print(f"Data Downloaded for {ticker}. Adding to Backtrader...")
    # Wrap the pandas DataFrame using Backtrader's feed
    data_feed = bt.feeds.PandasData(dataname=data_df)
    cerebro.adddata(data_feed)
    print("Data Added.")

    # --- Add strategy ---
    # Define strategy parameters
    trail_stop_percentage = 0.10 # Example: 10% Trailing Stop
    cerebro.addstrategy(VamaStrategy,
                        vama_period=30,         # Example VAMA period
                        vama_vol_period=7,      # Example VAMA volatility
period
                        sma_period=90,          # Example slow SMA period
                        trail_perc=trail_stop_percentage) # Pass the trailing
stop parameter
    print(f"Strategy Added with {trail_stop_percentage*100}% Trailing Stop.")

    # --- Configure Broker ---
    initial_cash = 10000.0
    commission_rate = 0.001 # 0.1%
    cerebro.broker.setcash(initial_cash)
    cerebro.broker.setcommission(commission=commission_rate)
    print(f"Broker configured: Cash={initial_cash},
Commission={commission_rate*100}%")

    # --- Add Sizer ---
    position_size_perc = 95 # Use 95% of equity per trade
    cerebro.addsizer(bt.sizers.PercentSizer, percents=position_size_perc)
    print(f"Sizer Added: {position_size_perc}% PercentSizer")

    # --- Run Backtest ---
    print(f'Starting Portfolio Value: {cerebro.broker.getvalue():.2f}')
    print("Running Backtest...")
    results = cerebro.run() # Run the backtest
    print("Backtest Finished.")
    print(f'Ending Portfolio Value: {cerebro.broker.getvalue():.2f}')

    # --- Plot Results ---
    print("Generating Plot...")
    # Use iplot=False for non-interactive environments
    cerebro.plot(style='candlestick', barup='green', bardown='red',
iplot=False)
    print("Plotting Complete.")
```



Pasted image 20250422031254.png

Pasted image 20250422031254.png

Setup Walkthrough:

1. **Cerebro:** The main Backtrader engine is created.
2. **Data:** Uses `yfinance` to download historical data for a specified ticker and date range, then wraps the resulting Pandas DataFrame with `bt.feeds.PandasData`.
3. **Strategy:** Adds the `VamaStrategy`, crucially passing the desired parameters including `trail_perc`.
4. **Broker:** Sets the initial cash and commission rate.
5. **Sizer:** Configures position sizing using `PercentSizer`.
6. **Run & Plot:** Executes the backtest using `cerebro.run()` and visualizes the results with `cerebro.plot()`.

Conclusion

This complete example demonstrates the process of creating a custom adaptive indicator (VAMA), incorporating it into a trading strategy with specific entry/exit rules (VAMA/SMA crossover), and adding essential risk management through a trailing stop loss. By using Backtrader's flexible framework, you can build and test sophisticated strategies tailored to your specific hypotheses about market behavior. Remember that parameter tuning and further testing across different market conditions are crucial steps before considering any strategy for live trading.

Crypto Trading Strategy using the Sharpe Ratio with Python Code

Cryptocurrencies like Ethereum and Bitcoin are known for their high volatility, presenting both opportunities and significant risks for traders. Navigating these turbulent markets often requires a disciplined approach. Quantitative trading strategies, which rely on mathematical models and historical data to make trading decisions, offer one such framework.

This article explores a specific quantitative strategy using **Sharpe Ratio** to generate long and short signals, aiming to capitalize on periods of strong risk-adjusted performance. We'll break down the logic, examine its implementation in Python step-by-step, and discuss crucial considerations for evaluating such a strategy.

1. Setup: Importing Libraries

First, we import the necessary Python libraries:

Python

```
import yfinance as yf       # To download historical market data from Yahoo
Finance
import pandas as pd         # For data manipulation and analysis (DataFrames)
import numpy as np          # For numerical operations (like square root)
import matplotlib.pyplot as plt # For plotting the results
```

- **Explanation:** We need `yfinance` to get the price data, pandas to work with it efficiently in a table-like structure (DataFrame), numpy for mathematical calculations, and `matplotlib` to visualize the strategy's performance.

2. Strategy Parameters

We define the core parameters that control the strategy's behavior:

Python

```
# Parameters
symbol = "ETH-USD"          # The asset we want to trade
start_date = "2018-01-01"   # Start date for historical data
end_date = "2025-04-17"     # End date for historical data (use current date
for latest)
window = 30                 # Rolling window (in days) for Sharpe Ratio
calculation
upper_threshold = 2.0       # Sharpe Ratio threshold to trigger a long signal
```

```
lower_threshold = -2.0      # Sharpe Ratio threshold to trigger a short
signal
hold_days = 7               # Maximum number of days to hold a position
stop_loss_pct = 0.20        # Stop-loss percentage (e.g., 0.20 = 20%)
```

- **Explanation:** These variables make it easy to adjust the strategy. We specify the crypto pair (ETH-USD), the date range for our backtest, the lookback period (window) for the Sharpe calculation, the thresholds that trigger trades, the maximum hold_days for any trade, and a stop_loss_pct for risk management.

3. Data Acquisition and Preparation

We download the historical price data and calculate basic returns:

Python

```
# 1) Download data
df = yf.download(symbol, start=start_date, end=end_date, progress=False)

# Clean column names (sometimes yfinance returns multi-level columns)
df.columns = [col.lower() for col in df.columns] # Make column names
lowercase

# 2) Calculate daily returns
df["return"] = df["close"].pct_change() # Calculate daily percentage change
in closing price
```

- **Explanation:** We use yf.download to fetch the Open, High, Low, Close, Adjusted Close, and Volume data for Ethereum. We simplify the column names and then calculate the daily return based on the percentage change in the close price from one day to the next. This return series is the basis for our Sharpe calculation.

4. Calculating the Rolling Sharpe Ratio

This is the core signal generation step. We calculate the Sharpe Ratio over a rolling window:

Python

```
# 3) Compute rolling Sharpe (annualized)
# Calculate rolling mean return and rolling standard deviation
rolling_mean = df["return"].rolling(window).mean()
rolling_std = df["return"].rolling(window).std()

# Calculate annualized Sharpe Ratio
# Note: np.sqrt(365) is arguably more appropriate for crypto (24/7 market)
df["rolling_sharpe"] = (rolling_mean / rolling_std) * np.sqrt(252)
```

- **Explanation:** For each day, we look back over the specified `window` (30 days). We calculate the average daily return (`rolling_mean`) and the standard deviation of daily returns (`rolling_std`) during that period. The Sharpe Ratio is then `rolling_mean / rolling_std`. We annualize it by multiplying by the square root of trading days per year (`np.sqrt(252)` used here; `np.sqrt(365)` is often better for crypto). A higher value suggests better risk-adjusted returns recently.

5. Generating Trading Signals

Based on the calculated Sharpe Ratio, we generate the raw entry signals:

Python

```
# 4) Generate entry signals based on thresholds
df["signal"] = 0 # Default signal is neutral (0)
df.loc[df["rolling_sharpe"] > upper_threshold, "signal"] = 1  # Go Long if
Sharpe > upper
df.loc[df["rolling_sharpe"] < lower_threshold, "signal"] = -1 # Go Short if
Sharpe < lower
```

- **Explanation:** We create a new `signal` column, initially all zeros. If a day's `rolling_sharpe` is above the `upper_threshold` (2.0), we set the signal to 1 (buy). If it's below the `lower_threshold` (-2.0), we set the signal to -1 (short). Otherwise, it remains 0 (do nothing).

6. Preparing for Backtesting

Before running the simulation, we adjust the signal timing and initialize variables:

Python

```
# 5) Prepare for Backtest: Shift signals to avoid lookahead bias
# We use yesterday's signal to trade on today's price
df['signal_shifted'] = df['signal'].shift(1).fillna(0)

# Initialize backtest variables
position = 0                # Current position: 1=long, -1=short, 0=flat
entry_price = 0.0           # Price at which the current position was entered
days_in_trade = 0           # Counter for days held in the current trade
equity = 1.0                # Starting equity (normalized to 1)
equity_curve = [equity]     # List to store equity values over time
```

- **Explanation:**
  - `df['signal'].shift(1)`: This is crucial. It shifts the signals forward by one day. This means the signal generated using data up to the *end* of Day T is used to make a trading decision on Day T+1. This prevents lookahead bias. `.fillna(0)` handles the first day which has no prior signal.

  - We initialize position to 0 (flat), track the entry_price and days_in_trade for the current position, start with a hypothetical equity of 1, and create a list equity_curve to record how this equity changes over time.

7. The Backtesting Engine Loop

This loop simulates the strategy day by day:

Python

```python
# 6) Backtest Engine Loop
# Start loop from 'window' index to ensure enough data for rolling
calculation and shift
for i in range(window, len(df)):
    idx = df.index[i]          # Current date
    row = df.iloc[i]           # Current day's data row
    price = row["close"]       # Use today's close price for trading action
(simplification)
    sig = row["signal_shifted"] # Use *yesterday's* calculated signal

    # --- Check Exits First ---
    if position != 0: # Are we currently in a trade?
        days_in_trade += 1
        # Calculate potential return if we closed the trade *right now*
        current_return_pct = (price / entry_price - 1) * position

        # Check Stop Loss: Did the trade lose more than stop_loss_pct?
        if current_return_pct <= -stop_loss_pct:
            equity *= (1 - stop_loss_pct) # Apply the max loss percentage
            position = 0 # Exit position
        # Check Holding Period: Have we held for the max number of days?
        elif days_in_trade >= hold_days:
            equity *= (1 + current_return_pct) # Realize the profit/loss
            position = 0 # Exit position

    # --- Check Entries Second ---
    # Can only enter if currently flat (position == 0) and there's a non-zero
signal
    if position == 0 and sig != 0:
        position = sig             # Enter long (1) or short (-1)
        entry_price = price        # Record entry price (using today's close)
        days_in_trade = 0          # Reset trade duration counter

    # Append the *current* equity value to the curve after all checks/trades
    equity_curve.append(equity)
```

- **Explanation:**

- The loop iterates through each day starting after the initial `window` period needed for calculations.
- Inside the loop, it first checks if we are *already in a trade* (`position != 0`).
- If yes, it increments `days_in_trade` and calculates the `current_return_pct` based on the `entry_price` and the `price` for the *current* day.
- It then checks the exit conditions:
  - **Stop-Loss:** If the `current_return_pct` shows a loss greater than or equal to `stop_loss_pct`, the position is closed, and equity is reduced by the `stop_loss_pct`.
  - **Holding Period:** If the `days_in_trade` reaches `hold_days`, the position is closed, and the realized profit or loss (`current_return_pct`) is applied to the `equity`.
- If we are *not* in a trade (`position == 0`), it checks if there's a new entry signal (`sig != 0`). If yes, it sets the `position`, records the `entry_price`, and resets `days_in_trade`.
- Finally, it appends the potentially updated `equity` value to the `equity_curve` list for that day.

8. Post-Processing and Alignment

We align the calculated equity curve with the corresponding dates in our DataFrame:

Python

```
# 7) Align Equity Curve with DataFrame
# Remove the initial equity point (1.0) as it corresponds to the day *before* the loop starts
equity_curve_series = pd.Series(equity_curve[1:], index=df.index[window:])

# Add the equity curve to the DataFrame
df["equity"] = equity_curve_series
```

- **Explanation:** The `equity_curve` list contains one extra point at the beginning (the initial 1.0). We remove this and create a pandas Series using the dates from the DataFrame that correspond to the backtest period (starting from the `window`-th index). This equity series is then added as a new column to our DataFrame `df`.

9. Plotting the Results

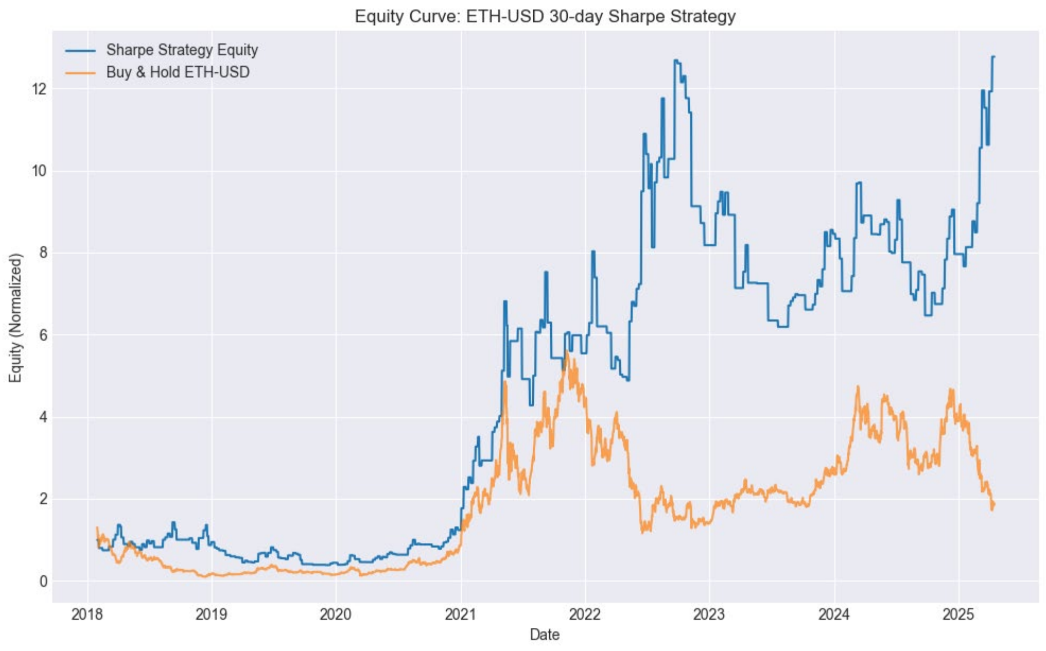
Finally, we visualize the performance:

Python

```
# 8) Plot Equity Curve vs. Buy-and-Hold
plt.figure(figsize=(12, 7)) # Create a figure for the plot
```

```
# Plot the strategy's equity curve
plt.plot(df.index, df["equity"], label="Sharpe Strategy Equity")

# Plot the normalized price of the asset (Buy and Hold)
# Normalize by dividing by the first closing price in the backtest period
buy_hold = df['close'] / df['close'].iloc[window] # Adjust starting point
plt.plot(buy_hold.index, buy_hold, label=f"Buy & Hold {symbol}", alpha=0.7)

# Add plot titles and labels
plt.title(f"Equity Curve: {symbol} {window}-day Sharpe Strategy")
plt.ylabel("Equity (Normalized)")
plt.xlabel("Date")
plt.legend() # Show the legend
plt.grid(True) # Add a grid for readability
plt.show() # Display the plot
```

- **Explanation:** This code generates a plot using `matplotlib`. It shows the strategy's equity curve over time. For comparison, it also plots the performance of simply buying and holding ETH-USD over the same period (normalized to start at 1). This visual comparison helps assess the strategy's relative performance and risk profile.



Pasted image 20250417003609.png

Pasted image 20250417003609.png

Interpreting the Results & Crucial Caveats

The output plot visualizes the hypothetical growth of capital. Comparing the strategy's equity curve to the normalized price of Ethereum helps assess whether the strategy added value (outperformed buy-and-hold) during the tested period, potentially with different risk characteristics (e.g., lower drawdowns).

However, it's vital to understand the limitations:

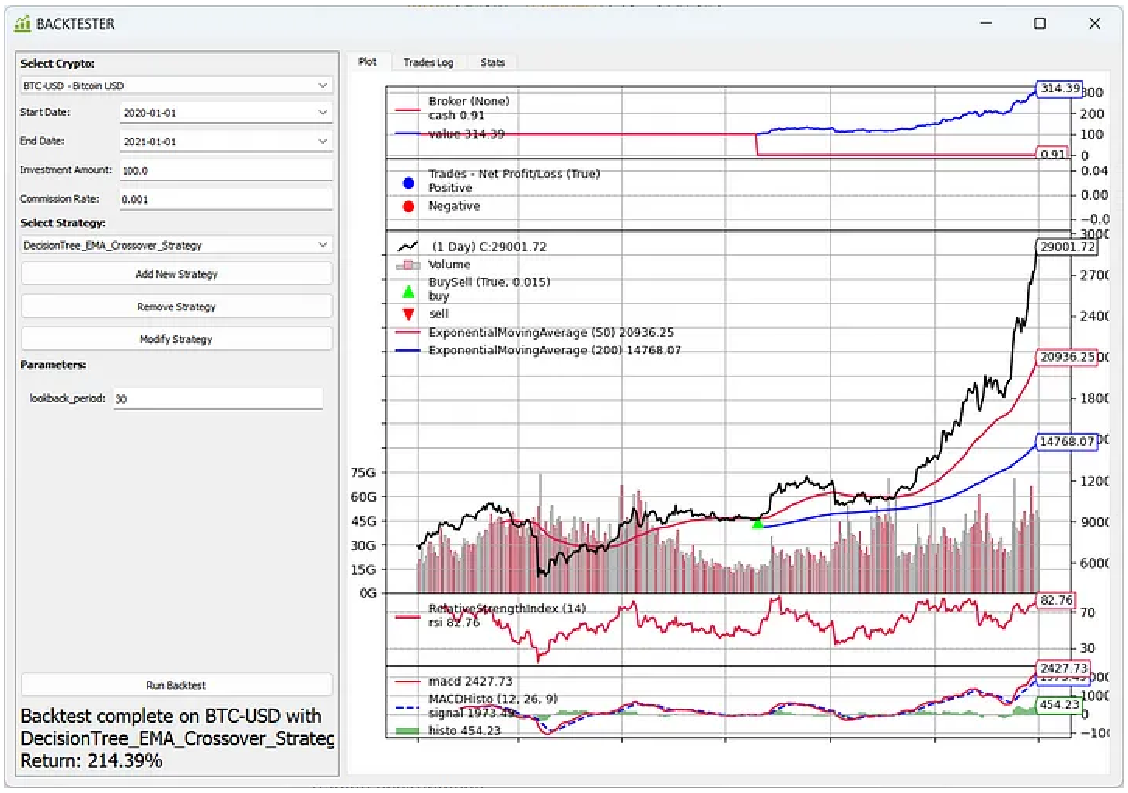
- **Transaction Costs & Slippage:** The code ignores trading fees and slippage, which reduce real-world profits.
- **Parameter Optimization (Overfitting):** The parameters might be tuned to past data and may not work well in the future.
- **Data Frequency:** Daily data masks intraday risk; stop-losses could trigger at worse prices than the close.
- **Annualization Factor:** Using `np.sqrt(365)` is likely more accurate for crypto.
- **Market Regimes:** Past performance doesn't guarantee future results; the strategy might fail in different market conditions.
- **Simplifications:** Using the closing price for execution is an approximation.

Conclusion

This Sharpe Ratio-based strategy provides a structured, quantitative approach to trading cryptocurrencies. While the backtest (with the corrected logic avoiding lookahead bias) offers insights, **this code and article are for educational purposes only and do not constitute financial advice.** Real-world trading requires incorporating costs, rigorous testing against overfitting, and robust risk management before committing capital.

Decision Trees and EMA Crossover 50% Average Annual Returns

I am working on trading strategies that blend traditional technical indicators with machine learning to generate buy and sell signals. In this article I try mixing a simple EMA crossover strategy with Decision Trees. First I will explain a bit on the theory of the methods and then share the Python implementation of the strategy with some backtest results. You can read this article on my website as well: https://www.aliazary.com/. You will find more articles and resources as well. You can also subscribe with your email address so that you get my newsletter and don't miss out on anything, especially my new backtesting app that I am working on. You can add your own strategies, modify the strategies and change the parameters and the asset and dates for backtesting the strategies to find the best strategies for trading:



Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2020-01-01
End Date: 2021-01-01
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: DecisionTree_EMA_Crossover_Strategy
Parameters:
lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strateg
Return: 214.39%

1_2Wv9CPpkmEQFwszVpkcv2g 1.webp

1_2Wv9CPpkmEQFwszVpkcv2g 1.webp

---

1. Theoretical Foundations

1.1 Exponential Moving Average (EMA)

The **Exponential Moving Average (EMA)** is a weighted moving average that gives more importance to recent prices, making it more responsive to new information. The formula is:

[*t = P_t + (1 -)* {t-1}]

where:

- (P_t) is the current price,
- (=) is the smoothing factor, and
- (n) is the number of periods.

In our strategy, we use two EMAs:

- **Short-term EMA** with a period of 50.

- **Long-term EMA** with a period of 200.

A bullish signal is generated when the short-term EMA crosses above the long-term EMA, while a bearish signal occurs when it crosses below.

1.2 Relative Strength Index (RSI)

The **Relative Strength Index (RSI)** is a momentum oscillator that measures the speed and change of price movements. Its formula is:

[= 100 -]

with

[RS =]

Typically, an RSI above 70 suggests that an asset may be overbought, while an RSI below 30 indicates oversold conditions.

1.3 Moving Average Convergence Divergence (MACD)

The **MACD** is a trend-following momentum indicator that shows the relationship between two EMAs of a security's price. It is calculated as:

[= {} - {}]

Usually, the short-term EMA is taken over 12 periods and the long-term EMA over 26 periods. A signal line, typically a 9-period EMA of the MACD, is also computed. In this strategy, the MACD histogram (the difference between the MACD line and its signal line) is used to capture momentum changes.

---

2. Decision Trees: Theory and Equations

2.1 Introduction to Decision Trees

A **Decision Tree** is a non-parametric supervised learning method used for both classification and regression. In classification, the goal is to assign a class label to a given input by learning decision rules inferred from the features.

2.2 Structure of a Decision Tree

A decision tree is composed of:

- **Root Node:** Represents the entire dataset.
- **Internal Nodes:** Each node represents a test on an attribute (feature).

- **Branches:** The outcome of the test.
- **Leaf Nodes:** Represent class labels or outcomes.

2.3 Splitting Criteria

To split the data at each node, decision trees typically use measures of impurity such as **Entropy** or the **Gini Index**.

Entropy

Entropy is a measure of the randomness or impurity in the data. For a binary classification, the entropy (H) is calculated as:

[H(p) = -p _2(p) - (1-p) _2(1-p)]

where (p) is the proportion of positive examples. A perfectly pure node (all examples of one class) has an entropy of 0.

Information Gain

Information Gain (IG) is used to measure the effectiveness of a split. It is defined as the difference between the entropy of the parent node and the weighted average of the entropies of the child nodes:

[= H() - _{i=1}^{k} H(_i)]

where:

- (N) is the total number of samples in the parent node,
- (N_i) is the number of samples in child (i), and
- (H(_i)) is the entropy of child (i).

Gini Index

The Gini Index is another measure of impurity:

[(p) = 1 - _{i=1}^{C} p_i^2]

where (p_i) is the probability of class (i) in the node. Lower values indicate higher purity.

2.4 Decision Trees in the Trading Strategy

In our strategy, the **Decision Tree Classifier** is used to predict whether the price will go up (represented by 1) or not (represented by 0). The steps include:

1. **Feature Extraction:**
   The classifier uses features derived from technical indicators (e.g., EMA values, RSI, MACD, signal values) over a defined lookback window.
2. **Training:**
   The decision tree is trained on historical data from the lookback window. The training involves splitting the data based on the feature values to minimize impurity (using either entropy or Gini Index).
3. **Prediction:**
   The latest feature vector is passed to the trained decision tree, which predicts the class label (up or down). This prediction is then used as one of the signals for trade execution.
4. **Model Adaptation:**
   The model is retrained continuously using a rolling window, ensuring that it adapts to new market conditions.

---

3. Strategy Implementation

3.1 Feature Engineering

In this strategy, features are generated from a lookback window (default 30 periods) including:

- Short-term EMA values (50 periods)
- Long-term EMA values (200 periods)
- RSI values (14 periods)
- MACD values and its signal line

These features are stacked into a matrix (X) for the decision tree to process. The target variable (y) is defined based on whether the price increased in the lookback window.

3.2 Training and Prediction Process

- **Training Data:**
  The feature matrix (X) is constructed from historical data (all rows except the last) and aligned with the target variable (y) (shifted by one period to maintain causality).
- **Prediction:**
  The most recent feature vector (last row of (X)) is fed into the decision tree to predict whether the price will increase.

3.3 Trade Execution Logic

The strategy combines the machine learning prediction with the EMA crossover condition:

- **Entry Signal:**
  If the decision tree predicts a price increase (1) and the short-term EMA is above the long-term EMA, a buy order is executed.

- **Position Sizing:**
  The size of the position is calculated based on available cash and the asset price, with a minor adjustment factor (0.99) for risk management.

- **Exit Signal:**
  If the short-term EMA falls below the long-term EMA, any open positions are closed, signaling a potential trend reversal.

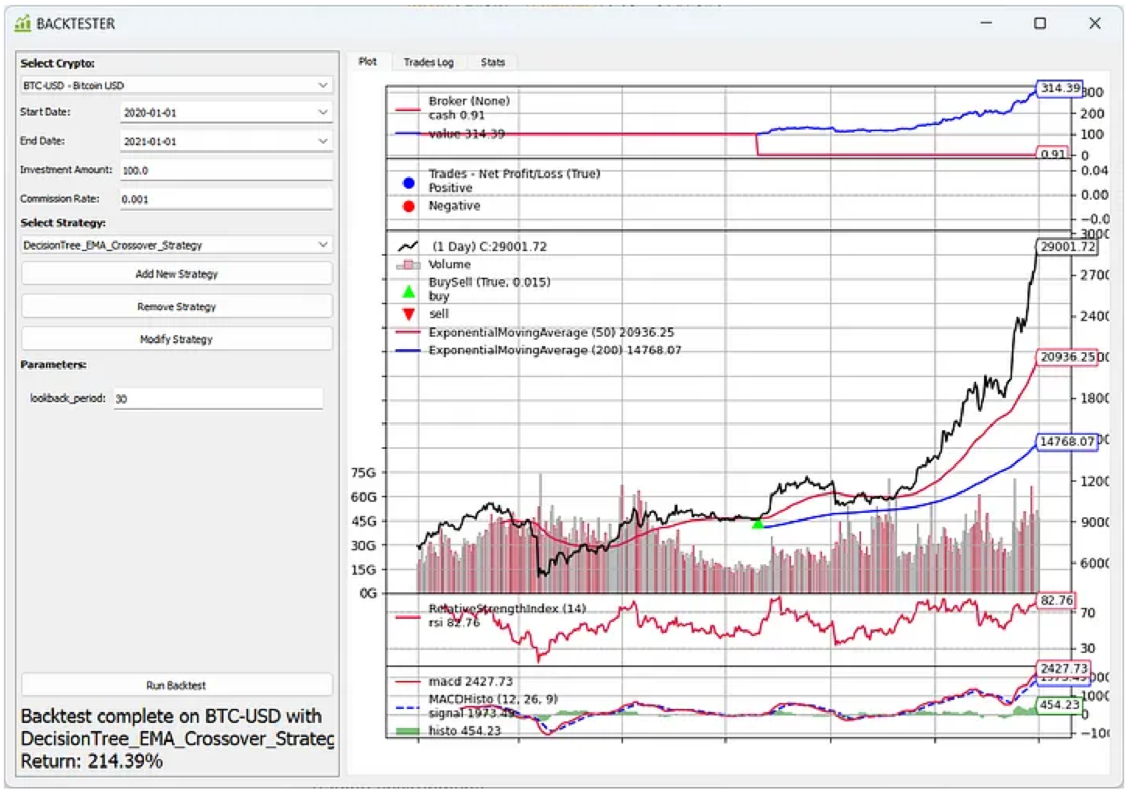
3.4 Code Walkthrough

Below is a Python implementation of the the strategy for use with backtrader library (or the **BACKTESTER** app) that integrates these concepts:

```
class DecisionTree_EMA_Crossover_Strategy(bt.Strategy):
    params = (("lookback_period", 30),)

    def __init__(self):
        # Data series and lookback window
        self.data_close = self.datas[0].close
        self.window = self.params.lookback_period

        # Decision Tree Classifier initialization
        self.model = DecisionTreeClassifier(random_state=42)

        # Technical indicators initialization
        self.emas = bt.indicators.ExponentialMovingAverage(self.data_close,
period=50)
        self.emal = bt.indicators.ExponentialMovingAverage(self.data_close,
period=200)
        self.rsi = bt.indicators.RelativeStrengthIndex(self.data_close,
period=14)
        self.macd = bt.indicators.MACDHisto(self.data_close,
                                            period_me1=12,
                                            period_me2=26,
                                            period_signal=9)
        self.order = None  # Track pending orders

    def next(self):
        # Ensure sufficient data is available for the lookback period
        if len(self) > self.window:
            # Extract indicator values over the lookback window
            emas_values = np.array(self.emas.get(size=self.window))
            emal_values = np.array(self.emal.get(size=self.window))
            rsi_values = np.array(self.rsi.get(size=self.window))
            macd_values = np.array(self.macd.macd.get(size=self.window))
            signal_values = np.array(self.macd.signal.get(size=self.window))

            # Construct feature matrix X
            X = np.column_stack((emas_values, emal_values, rsi_values,
macd_values, signal_values))

            # Define target variable: 1 if price increased, 0 otherwise
            prices = np.array(self.data_close.get(size=self.window + 1))
            y = np.where(np.diff(prices) > 0, 1, 0)

            # Prepare training and testing data
            X_train = X[:-1]
            y_train = y[1:]  # Shift target by one period to align with
features
            X_test = X[-1]

            # Train the decision tree on historical lookback data
            self.model.fit(X_train, y_train)

            # Predict the next move using the most recent features
            prediction = self.model.predict(X_test.reshape(1, -1))

            # Trade execution: enter position if conditions are met
            if not self.position:
                cash = self.broker.get_cash()
                asset_price = self.data_close[0]
                position_size = cash / asset_price * 0.99

                # Buy if prediction is 1 and the EMA crossover is bullish
                if prediction[0] == 1 and self.emas[0] > self.emal[0]:
                    self.buy(size=position_size)
                    self.log(f"Buy order placed at price: {asset_price:.2f}")
            else:
                # Close position if the EMA crossover indicates a bearish
trend
                if self.emas[0] < self.emal[0]:
                    self.close()
                    self.log(f"Position closed at price:
{self.data_close[0]:.2f}")

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        # Log order execution details
        if order.status == order.Completed:
            if order.isbuy():
                self.log(f"Buy executed: {order.executed.price:.2f}")
            elif order.issell():
                self.log(f"Sell executed: {order.executed.price:.2f}")
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log("Order canceled/margin/rejected")
        self.order = None

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f"{dt.isoformat()}, {txt}")
```

5. Backtests

let’s see the backtest results for trading Bitcoin for 5 consecutive years from 2020 to 2025. Since the strategy only takes long positions, it won’t make money in bearish markets. However, we can easily add short positions using opposite conditions so that we can make money in any market regime. If you are trading using a margin or futures account you can take short positions as well:

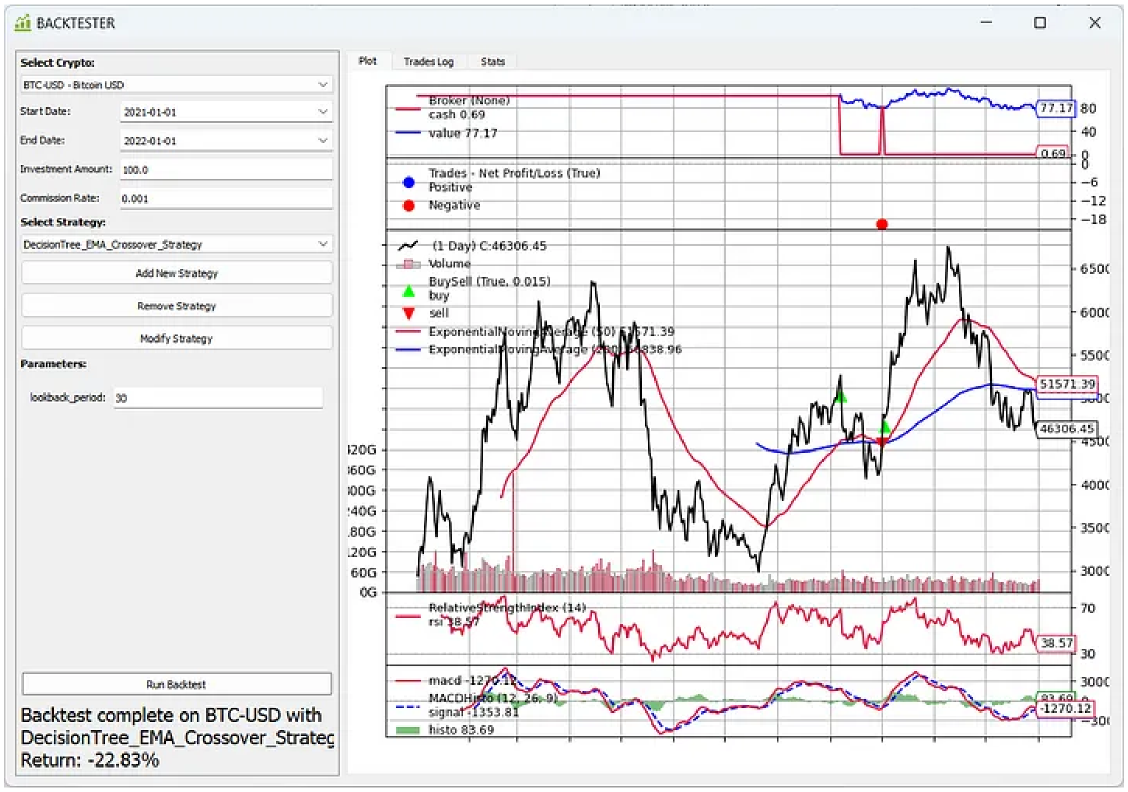


Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2020-01-01
End Date: 2021-01-01
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: DecisionTree_EMA_Crossover_Strategy
Parameters:
lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strateg
Return: 214.39%

1_2Wv9CPpkmEQFwszVpkcv2g.webp

1_2Wv9CPpkmEQFwszVpkcv2g.webp



Select Crypto: BTC-USD - Bitcoin USD

Start Date: 2021-01-01

End Date: 2022-01-01

Investment Amount: 100.0

Commission Rate: 0.001

Select Strategy: DecisionTree_EMA_Crossover_Strategy

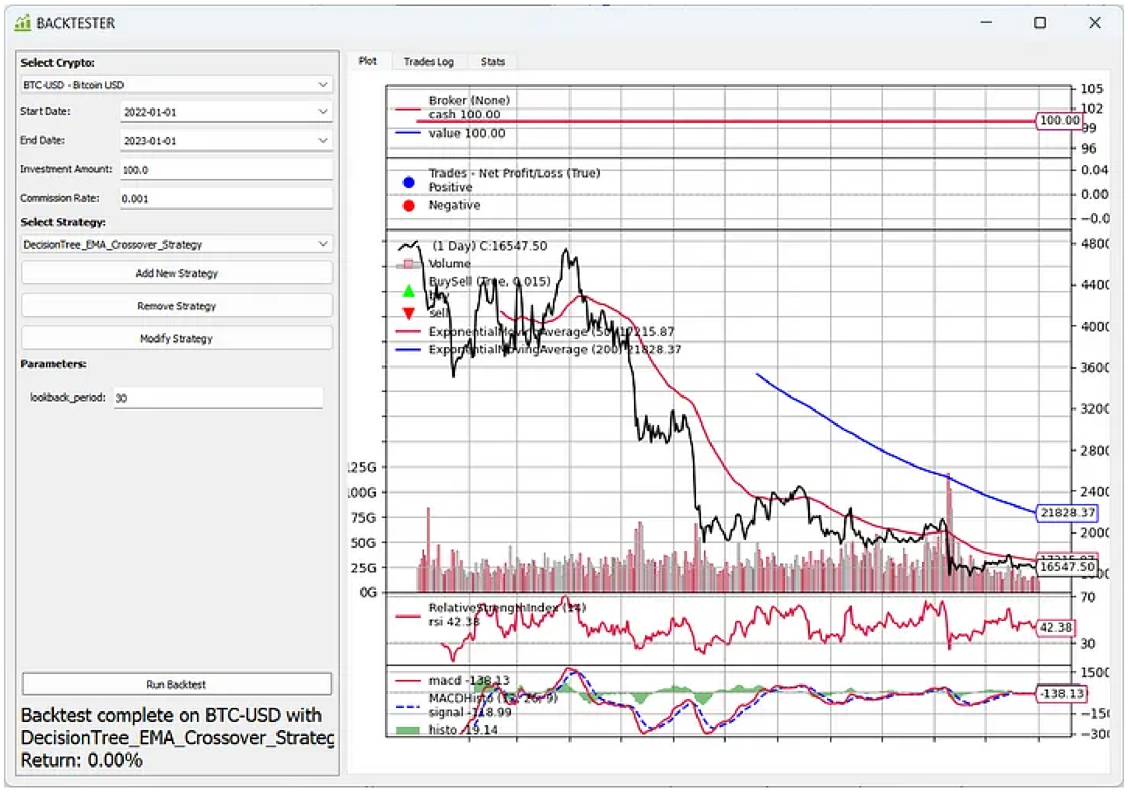
Parameters:

lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strateg
Return: -22.83%

1_ZhRVxa7aCOYNzouXc3pBCQ.webp

1_ZhRVxa7aCOYNzouXc3pBCQ.webp



BACKTESTER

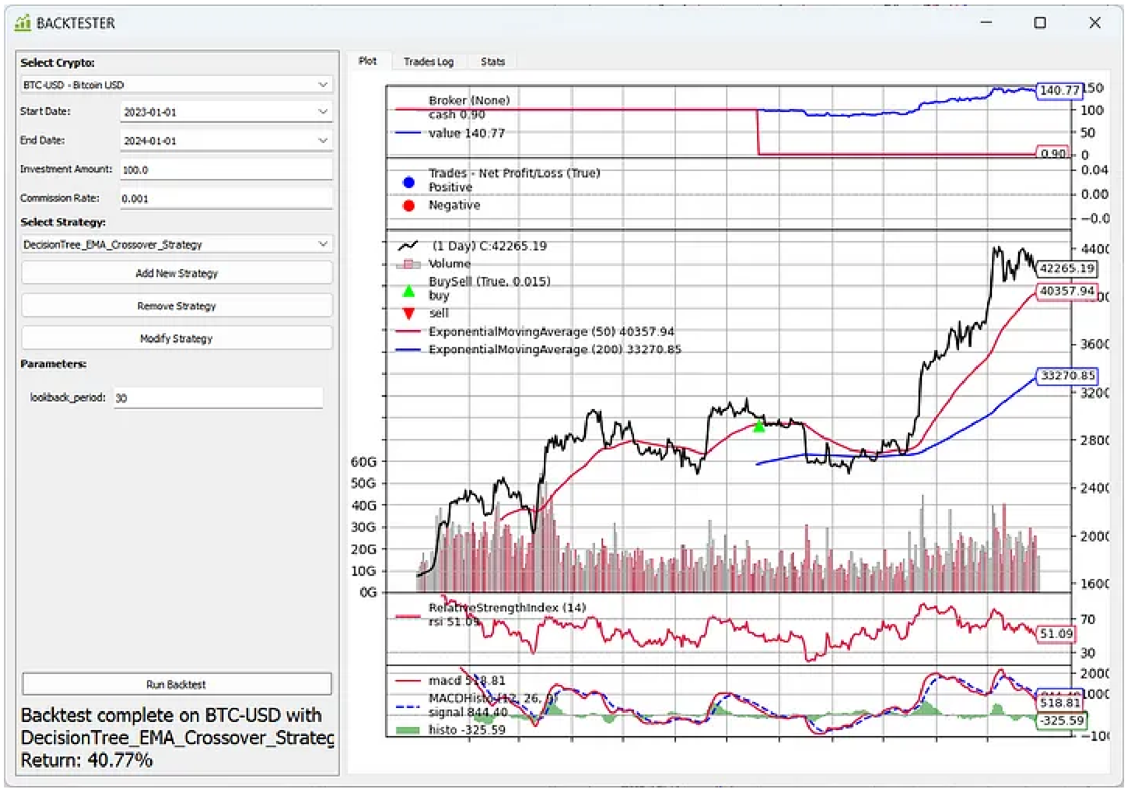
Select Crypto:
BTC-USD - Bitcoin USD
Start Date: 2022-01-01
End Date: 2023-01-01
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy:
DecisionTree_EMA_Crossover_Strategy

Parameters:
lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strateg
Return: 0.00%

1_IrSuMTr6fSl9BAbzhcbJNQ.webp

1_IrSuMTr6fSl9BAbzhcbJNQ.webp

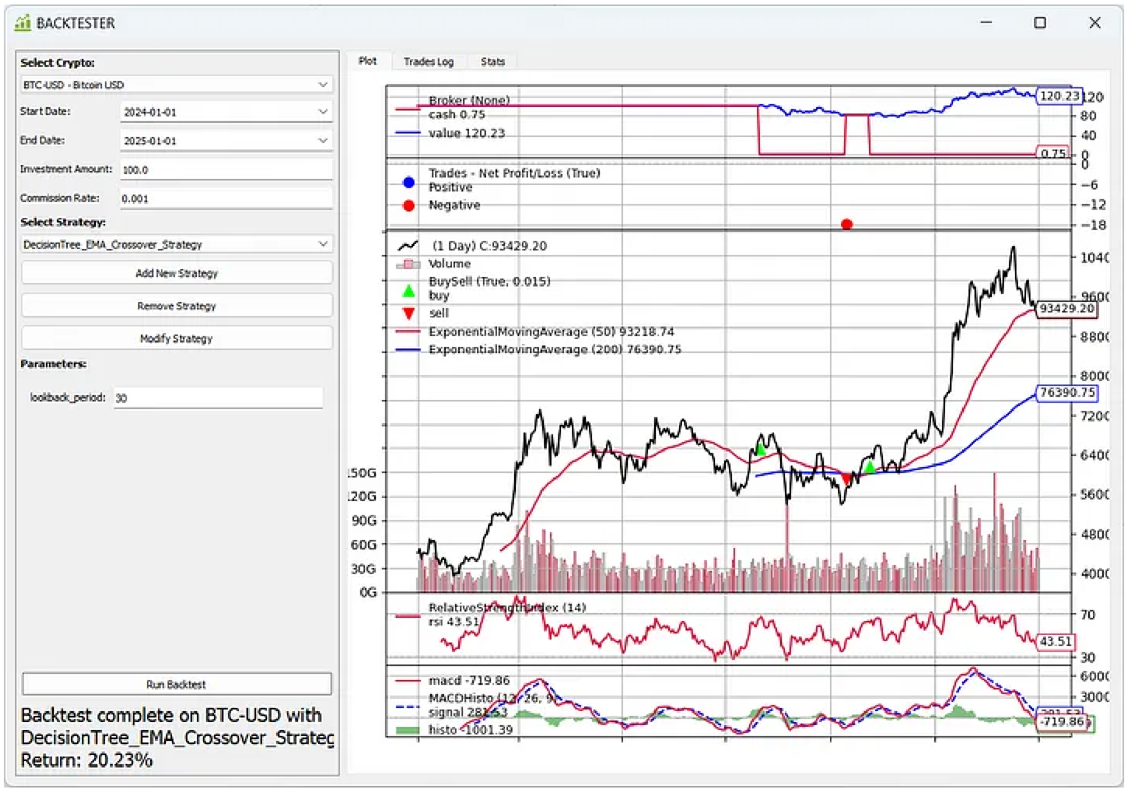


Select Crypto:
BTC-USD - Bitcoin USD
Start Date: 2023-01-01
End Date: 2024-01-01
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy:
DecisionTree_EMA_Crossover_Strategy
Parameters:
lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strateg
Return: 40.77%

1_NsD8a7tzNxEnXahla5Nakg.webp

1_NsD8a7tzNxEnXahla5Nakg.webp


BACKTESTER

Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2024-01-01
End Date: 2025-01-01
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: DecisionTree_EMA_Crossover_Strategy
Parameters:
lookback_period: 30

Backtest complete on BTC-USD with DecisionTree_EMA_Crossover_Strategy
Return: 20.23%

1_kQHBhBLCMohnNUWJrBUXbQ.webp

1_kQHBhBLCMohnNUWJrBUXbQ.webp

Overall the results seem promising. In the ranging market of 2021 we lost about 20%, which can be easily avoided with a stop-loss. The best case was the bullish market of 2020 where we made more than 200%. For a long-only strategy its performance is not so bad even for ranging or bearish markets. If we implement short selling and also put in place stop-loss conditions or any other risk management strategy, it has great potential as a consistently profitable strategy. In the end, please make sure to backtest it thoroughly for different periods and different assets to make sure its performance is what you expect. Also please make sure to go over the code carefully so that there are no mistakes. Always be careful, and try with a small account for real trading, so you make sure the real-life performance is good enough and you don't risk losing your money.

---

5. Conclusion

The **DecisionTree_EMA_Crossover_Strategy** represents a hybrid approach that integrates machine learning with traditional technical analysis. By employing technical indicators such as EMA, RSI, and MACD, the strategy gathers rich features that are fed into a decision tree classifier. The decision tree uses well-established splitting criteria—grounded in entropy, information gain, or the Gini Index—to predict future price movements. Coupled with the EMA crossover condition, this strategy aims to enhance trade execution by confirming machine-generated signals with trend-based indicators. As I mentioned before, you can make it even better adding short selling and implementing a simple risk management strategy like a stop-loss and end up with a very profitable trading bot that makes you money consistently.

This comprehensive overview provides both the theoretical background and the practical implementation details, offering a robust framework for adapting machine learning to dynamic trading environments. I hope you find it useful and I would also appreciate your ideas and comments if you have any.

Do Price Extremes Hold Trading Clues

In the world of financial markets, traders and analysts constantly seek methods to understand price movements and identify potential trading opportunities. Technical analysis offers a range of tools and strategies based on historical price data. One such simple strategy involves observing price breakouts relative to recent highs and lows. Let’s explore a basic “Rolling Max/Min Crossover” strategy, similar to concepts used in channel breakout systems.

The Strategy Explained

The core idea is to identify when the price moves significantly beyond its recent trading range, potentially signaling the start of a new trend or a reversal. We define this “recent range” using a rolling window – a specific number of past trading days (e.g., 20 days).

1. **Rolling Maximum:** The highest closing price observed during the specified rolling window period.
2. **Rolling Minimum:** The lowest closing price observed during the specified rolling window period.

The trading signals are generated as follows:

- **Buy Signal (+1):** Generated when the *current* closing price drops *below* the *Rolling Minimum* calculated over the *previous* window period. The intuition here is that the price has fallen significantly relative to its recent lows, potentially indicating an oversold condition or a point where buyers might step in.
- **Sell Signal (-1):** Generated when the *current* closing price rises *above* the *Rolling Maximum* calculated over the *previous* window period. This suggests the price has broken out above its recent highs, possibly indicating an overbought condition or a point where sellers might take profits.

Important Note: We calculate the Rolling Max/Min based on the data *before* the current day (`shift(1)`) to avoid lookahead bias – we make decisions based only on information available up to that point.

Implementation in Python

Let's see how this looks in Python using libraries like pandas and yfinance.

1. Getting Data: First, we fetch historical price data. We used Bitcoin (BTC-USD) as an example:

Python

```python
import yfinance as yf
import pandas as pd
import numpy as np

# Fetch Bitcoin data
ticker_symbol = "BTC-USD"
btc_data = yf.Ticker(ticker_symbol)
df = btc_data.history(period="1y") # Get 1 year of data

if df.empty:
    print("Could not fetch data.")
else:
    print(f"Fetched {len(df)} rows of data for {ticker_symbol}")
```

2. Calculating Signals: We then calculate the rolling minimum/maximum and generate the signals.

Python

```python
# Define the rolling window size (e.g., 20 days)
count = 20

# Create a DataFrame for signals
signals = pd.DataFrame(index=df.index)
signals['signal'] = 0.0 # 0 means hold/no signal
signals['trend'] = df['Close'] # Base the strategy on closing price

# Calculate Rolling Max/Min on *previous* data points
signals['RollingMax'] = signals.trend.shift(1).rolling(count).max()
signals['RollingMin'] = signals.trend.shift(1).rolling(count).min()

# Generate Buy signal (Price drops below Rolling Min)
signals.loc[signals['RollingMin'] > signals.trend, 'signal'] = 1

# Generate Sell signal (Price rises above Rolling Max)
signals.loc[signals['RollingMax'] < signals.trend, 'signal'] = -1

# See how many signals were generated
print("\nSignal distribution:")
print(signals['signal'].value_counts())
```

Simulating Trades (Backtesting)

Generating signals is one thing; seeing how they might have performed historically is another. This is done via backtesting. We created a function (buy_stock) to simulate making trades based on these signals, starting with an initial amount of capital and tracking hypothetical profits or losses. This function handles buying and selling (including fractional amounts for assets like Bitcoin) when a signal occurs, subject to certain constraints (like only buying if we don't already hold the asset).

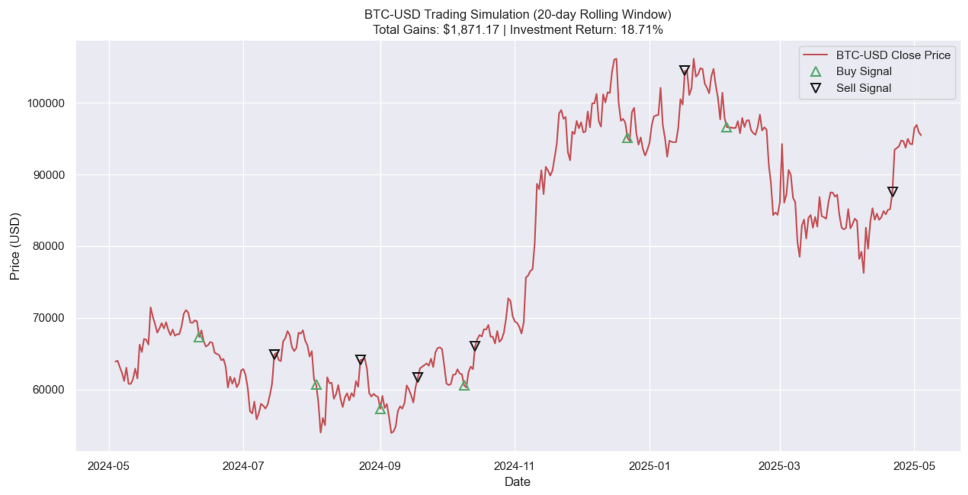
Python

```
# (Simplified representation of calling the backtest function)

# Assuming 'buy_stock' function is defined as in the previous example...
initial_money = 10000
states_buy, states_sell, total_gains, invest_percent = buy_stock(
    real_movement=df['Close'],
    signal=signals['signal'].values,
    initial_money=initial_money
)

print(f"Simulation Result: Gain ${total_gains:,.2f} ({invest_percent:.2f}%)")
```

Finally, we visualize the results, plotting the closing price and marking the points where buy and sell signals were executed.

Python

```
import matplotlib.pyplot as plt

# (Assuming states_buy/states_sell lists contain indices of trades)
if states_buy or states_sell:
    plt.figure(figsize=(15, 7))
    plt.plot(df['Close'], color='r', lw=1.5, label=f'{ticker_symbol} Close
Price')
    plt.plot(df['Close'], '^', markersize=8, color='g', label='Buy Signal',
markevery=states_buy, markerfacecolor='none')
    plt.plot(df['Close'], 'v', markersize=8, color='k', label='Sell Signal',
markevery=states_sell, markerfacecolor='none')
    plt.title(f'{ticker_symbol} - Rolling Max/Min ({count}-day) Strategy
Simulation')
    plt.ylabel("Price (USD)")
    plt.xlabel("Date")
    plt.legend()
    plt.grid(True)
    plt.show()
```



Pasted image 20250504181829.png

Pasted image 20250504181829.png

Interpretation and Caveats

This strategy provides a basic framework for identifying potential entry and exit points based on recent price ranges.

However, it's crucial to understand its limitations:

- **Simplicity:** This is a very basic strategy and doesn't incorporate many factors real traders use (volume, broader market trends, news, other indicators).
- **Whipsaws:** In sideways or choppy markets, the price might frequently cross the rolling max/min levels, leading to many small, potentially losing trades (whipsaws).
- **Parameter Dependent:** The strategy's performance heavily depends on the chosen rolling window (`count`). A different window might yield vastly different results.
- **No Risk Management (Explicit):** The basic signal generation doesn't include stop-loss or take-profit rules, which are vital for risk management.
  Our `buy_stock` function was also very basic in its execution logic.
- **Idealized Simulation:** Backtesting often ignores transaction costs (fees, slippage) which can significantly impact profitability.

Conclusion

The Rolling Max/Min Crossover strategy serves as a simple, understandable example of how technical indicators can be used to generate trading signals. While likely too basic for direct application without significant refinement and risk management, implementing and backtesting such strategies provides valuable educational insight into quantitative analysis, programming for finance, and the challenges of strategy development.

Remember, past performance is not indicative of future results, and this article is purely for educational purposes, not financial advice.

Easy Entry into Algorithmic Trading with Backtrader and Backtester

Algorithmic trading, the practice of using computer programs to execute trading strategies, has gained immense popularity. For Python enthusiasts looking to dive into this world, Backtrader stands out as a powerful and flexible open-source framework. It allows for the testing and implementation of trading ideas with historical or live data. To get started, understanding a minimal strategy template is crucial. This article dissections a fundamental Backtrader strategy, providing a clear path for beginners.

The provided code snippet offers a complete, albeit simple, example of a Backtrader strategy. Let's break it down step-by-step:

Python

```
import backtrader as bt
import yfinance as yf
import matplotlib.pyplot as plt
# %matplotlib qt5 # This line is specific to Jupyter environments for
interactive plots

class MyStrategy(bt.Strategy):
    # define any strategy parameters here (optional)
    params = (
        ('maperiod', 15),  # example parameter: moving average period
    )

    def __init__(self):
        # references to data feeds, indicators, etc.
        self.dataclose = self.datas[0].close # Accessing the closing price of
the primary data feed
        # simple moving average example
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=self.params.maperiod # Using the defined
'maperiod'
        )

    def next(self):
        # called on each new bar (data point)
        if not self.position:  # Check if not in the market
            if self.dataclose[0] > self.sma[0]: # If close is above SMA
                # enter long
                self.buy()
        else: # Already in the market
            if self.dataclose[0] < self.sma[0]: # If close is below SMA
                # exit long
                self.close()

if __name__ == '__main__':
    cerebro = bt.Cerebro() # Create a Cerebro engine instance
    cerebro.addstrategy(MyStrategy, maperiod=20)  # Add the strategy,
optionally overriding params

    # Fetch data using yfinance
    ticker = yf.Ticker('BTC-USD') # Example: Bitcoin USD data
    data_df = ticker.history(period='1y') # Fetch 1 year of historical data

    # Add a data feed (Pandas DataFrame)
    data_feed = bt.feeds.PandasData(dataname=data_df)
    cerebro.adddata(data_feed)

    cerebro.broker.setcash(100000.0) # Set initial trading capital
    print('Starting Portfolio Value:', cerebro.broker.getvalue())
    cerebro.run() # Execute the backtest
    print('Final Portfolio Value:', cerebro.broker.getvalue())

    cerebro.plot(iplot=False) # Plot the results (iplot=False for non-
interactive plots)
```

Core Components Explained:

1. **Imports:**
    - `backtrader as bt`: This is the primary import for the Backtrader library itself.
    - `yfinance as yf`: A popular library to download historical market data from Yahoo Finance. This is used here to fetch sample data for backtesting.
    - `matplotlib.pyplot as plt`: Used by Backtrader for plotting the results of the backtest.
    - `# %matplotlib qt5`: This is a magic command for Jupyter Notebook or IPython environments. It specifies that Matplotlib plots should be rendered in a separate, interactive Qt window. If you're running a standard Python script, this line might be unnecessary or could cause issues if the Qt backend isn't configured.
2. **Strategy Definition (`MyStrategy(bt.Strategy)`):**
    - Every trading strategy in Backtrader is a class that inherits from `bt.Strategy`.
    - **`params`:** This class attribute is a tuple of tuples, allowing you to define parameters for your strategy. These parameters can be easily tuned without modifying the core logic. In this example, `maperiod` is defined with a default value of 15, representing the period for a moving average.

- **`__init__(self)`:** This is the constructor of your strategy. It's called once when the strategy is initialized.
  - `self.datas[0]`: Backtrader can handle multiple data feeds. `self.datas[0]` refers to the first (primary) data feed added to Cerebro.
  - `self.dataclose = self.datas[0].close`: This creates a convenient alias to the 'close' price series of the primary data feed.
  - `self.sma = bt.indicators.SimpleMovingAverage(...)`: Here, a Simple Moving Average (SMA) indicator is instantiated. It's calculated on the primary data feed (`self.datas[0]`) using the period defined in `self.params.maperiod`. Backtrader has a rich library of built-in indicators.
- **`next(self)`:** This is the heart of the strategy. It's called for each new bar (e.g., daily, hourly) of data once all indicators have a calculated value.
  - `if not self.position:`: This checks if the strategy currently holds any open position (i.e., is "in the market"). `self.position` is a Backtrader object that provides information about the current market position.
  - `if self.dataclose[0] > self.sma[0]`: This is the entry condition for a long position. It checks if the current closing price (`self.dataclose[0]`) is greater than the current SMA value (`self.sma[0]`). The `[0]` refers to the current bar.
  - `self.buy()`: If the entry condition is met, this command places a market order to buy the asset.
  - `else: ... if self.dataclose[0] < self.sma[0]: self.sell()`: If the strategy is already in a position, this block checks the exit condition. If the current closing price falls below the SMA, `self.sell()` places a market order to sell the asset (closing the long position).

3. **Execution Block (`if __name__ == '__main__':`)**

   - This standard Python construct ensures the code within it only runs when the script is executed directly (not when imported as a module).
   - **`cerebro = bt.Cerebro()`:** Cerebro (Spanish for "brain") is the central engine of Backtrader. It orchestrates data feeds, strategies, brokers, analyzers, and more.
   - **`cerebro.addstrategy(MyStrategy, maperiod=20)`:** This line adds an instance of our `MyStrategy` to Cerebro. Notice that we can override the default `maperiod` (which was 15) directly here, setting it to 20 for this specific backtest.
   - **Data Fetching with `yfinance`:**

    - `ticker = yf.Ticker('BTC-USD')`: An instance of a yfinance `Ticker` object is created for Bitcoin against the US Dollar.
    - `data_df = ticker.history(period='1y')`: The `history` method fetches historical data for the specified period (here, ‘1y’ for one year) and returns it as a Pandas DataFrame.
  - **Adding Data Feed to Cerebro:**
    - `data_feed = bt.feeds.PandasData(dataname=data_df)`: Backtrader needs data in a specific format. `bt.feeds.PandasData` is a convenient way to feed data from a Pandas DataFrame. The `dataname` argument points to our fetched DataFrame. Backtrader automatically recognizes common column names like ‘Open’, ‘High’, ‘Low’, ‘Close’, ‘Volume’, and ‘OpenInterest’.
    - `cerebro.adddata(data_feed)`: The prepared data feed is added to Cerebro.
  - **Broker Setup:**
    - `cerebro.broker.setcash(100000.0)`: This sets the initial cash balance for the simulated broker.
  - **Running the Backtest:**
    - `print('Starting Portfolio Value:', cerebro.broker.getvalue())`: Prints the initial portfolio value (which will be the cash set).
    - `cerebro.run()`: This command starts the backtesting process. Cerebro will iterate through the historical data, calling the `next()` method of the strategy for each bar.
    - `print('Final Portfolio Value:', cerebro.broker.getvalue())`: After the backtest is complete, this prints the final value of the portfolio.
  - **Plotting Results:**
    - `cerebro.plot(iplot=False)`: Backtrader provides a convenient plotting facility. `iplot=False` is often used for non-Jupyter environments or when a static plot is preferred. If `iplot=True` (the default if `matplotlib` is configured for interactive backends), it would attempt to generate an interactive plot. The plot typically shows the price data, indicators (like our SMA), and buy/sell trades.

How This Simple Strategy Works:

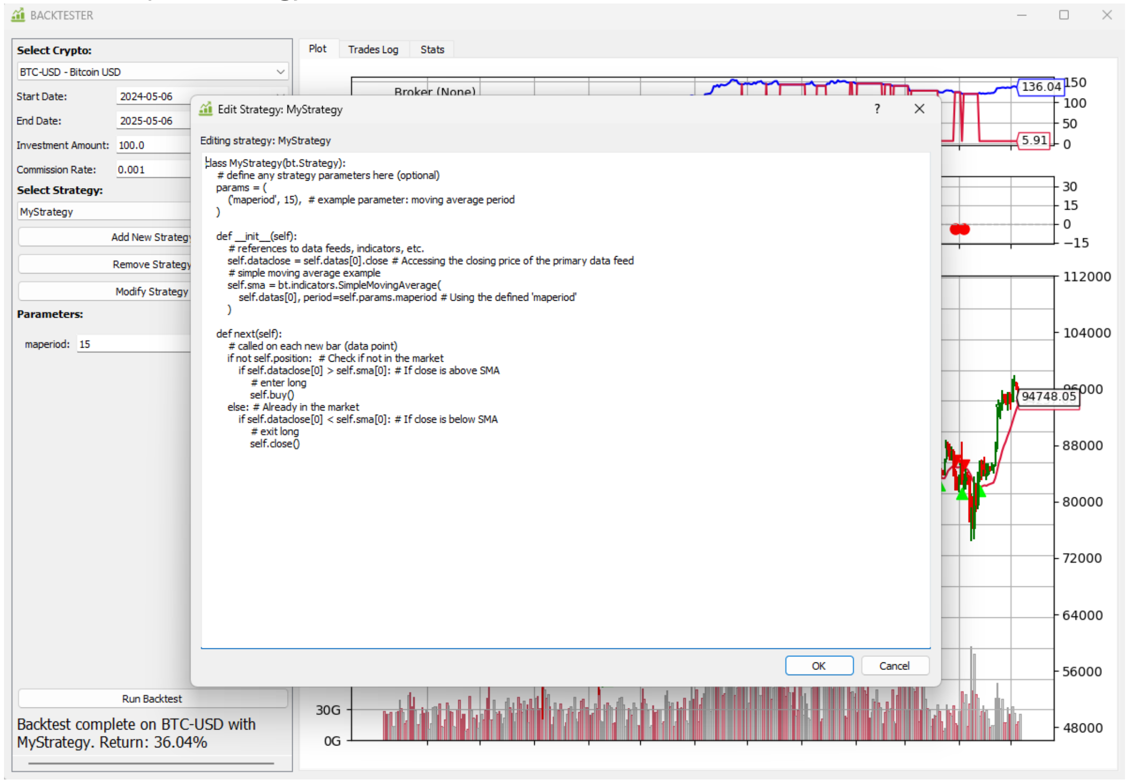
This template implements a basic trend-following strategy using a Simple Moving Average (SMA):

- **Entry Signal:** When the closing price of the asset crosses above its SMA, the strategy assumes an uptrend might be starting and enters a long position (buys).

- **Exit Signal:** If the strategy is already in a long position and the closing price crosses below the SMA, it assumes the uptrend might be ending and exits the position (sells).

This is a very straightforward strategy and serves primarily as an educational example of how to structure code within the Backtrader framework.

I added the simple strategy class into the "Backtester" app:



Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2024-05-06
End Date: 2025-05-06
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: MyStrategy
Parameters:
maperiod: 15

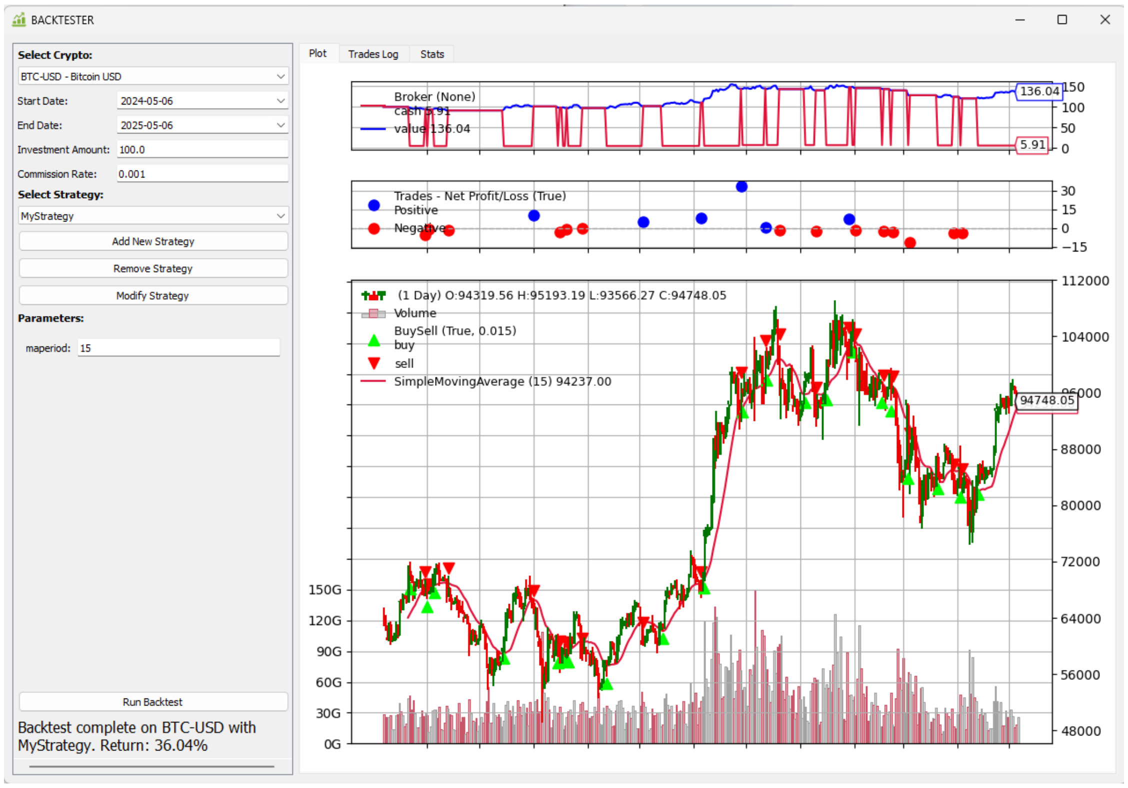
Backtest complete on BTC-USD with MyStrategy. Return: 36.04%

Edit Strategy: MyStrategy

Editing strategy: MyStrategy

```
class MyStrategy(bt.Strategy):
    # define any strategy parameters here (optional)
    params = (
        ('maperiod', 15),  # example parameter: moving average period
    )

    def __init__(self):
        # references to data feeds, indicators, etc.
        self.dataclose = self.datas[0].close # Accessing the closing price of the primary data feed
        # simple moving average example
        self.sma = bt.indicators.SimpleMovingAverage(
            self.datas[0], period=self.params.maperiod # Using the defined 'maperiod'
        )

    def next(self):
        # called on each new bar (data point)
        if not self.position:  # Check if not in the market
            if self.dataclose[0] > self.sma[0]: # If close is above SMA
                # enter long
                self.buy()
        else: # Already in the market
            if self.dataclose[0] < self.sma[0]: # If close is below SMA
                # exit long
                self.close()
```

and run backtest:



Select Crypto:
BTC-USD - Bitcoin USD
Start Date: 2024-05-06
End Date: 2025-05-06
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy:
MyStrategy
Parameters:
maperiod: 15

Backtest complete on BTC-USD with MyStrategy. Return: 36.04%

Pasted image 20250506222608.png

Pasted image 20250506222608.png

Expanding Beyond the Minimal:

This minimal template is a fantastic starting point. From here, you can:

- **Add More Indicators:** Explore Backtrader's extensive library of indicators (e.g., RSI, MACD, Bollinger Bands) or create your own.
- **Implement More Complex Logic:** Develop multi-condition entry/exit signals.
- **Introduce Risk Management:** Incorporate stop-loss, take-profit orders, or position sizing.
- **Handle Short Selling:** Add logic to profit from falling prices.
- **Use Different Data Sources:** Connect to CSV files, databases, or live brokers.
- **Optimize Parameters:** Use Backtrader's optimization capabilities to find the best parameter settings for your strategy.
- **Add Analyzers:** Utilize analyzers to get detailed statistics about your strategy's performance (e.g., Sharpe ratio, drawdown).

By understanding this foundational template, aspiring algorithmic traders can confidently begin building and testing their own sophisticated trading strategies using the versatile Backtrader library.

Enhancing ADX Trend Strategy with Ranging Filters, and Trailing Stops from 36% to 182% Profit

In algorithmic trading, trend-following strategies can capture profit from sustained market movements. One popular method for filtering trades is using the Average Directional Index (ADX) along with its related directional indicators, +DI and –DI. In this article, we will build a basic ADX trend strength strategy, then demonstrate step-by-step how to enhance it by filtering out sideways (ranging) markets with Bollinger Bands and by implementing trailing stops for improved risk management.

1. Basic ADX Trend Strength Strategy

Strategy Overview

The **ADX Trend Strength Strategy** relies on the following key principles:

- **Trend Strength Measurement:**
  The ADX indicator measures market trend strength. A high ADX value (e.g., above 25) suggests that the market is trending, while a low value points to weak or sideways movement.
- **Directional Signals:**
  The strategy uses the +DI (positive directional indicator) and -DI (negative directional indicator) to determine trade direction:
  - If **+DI > -DI**, the market is considered bullish and the strategy goes long.
  - If **-DI > +DI**, it is considered bearish and the strategy goes short.
- **Order Handling and Logging:**
  The strategy logs important events such as trade execution, order status updates, and overall performance once the backtest concludes.

Code Snippet

Below is the code for the basic ADX trend strategy:

```
import backtrader as bt
import yfinance as yf
import pandas as pd
import datetime

class ADXTrendStrengthStrategy(bt.Strategy):
    """
    ADX Trend Strength Strategy:
    - Trades only when ADX is above a specified threshold (default 25),
indicating a strong trend.
    - Trade direction is determined by the directional indicators:
      * If +DI > -DI, enter long.
      * If -DI > +DI, enter short.
    """
    params = (
        ('adx_period', 14),     # Period for ADX and directional indicators
        ('adx_threshold', 25),  # ADX threshold for strong trend
        ('printlog', True),     # Enable logging of events
    )

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.datetime(0)
            print(f'{dt.isoformat()} - {txt}')

    def __init__(self):
        self.dataclose = self.datas[0].close
        # Initialize ADX, PlusDI, and MinusDI indicators using the same
period.
        self.adx = bt.indicators.ADX(self.datas[0],
period=self.params.adx_period)
        self.plusdi = bt.indicators.PlusDI(self.datas[0],
period=self.params.adx_period)
        self.minusdi = bt.indicators.MinusDI(self.datas[0],
period=self.params.adx_period)
        self.order = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return  # Order is being processed

        if order.status == order.Completed:
            if order.isbuy():
                self.log(
                    f'BUY EXECUTED, Price: {order.executed.price:.2f}, '
                    f'Cost: {order.executed.value:.2f}, Comm:
{order.executed.comm:.2f}'
                )
            elif order.issell():
                self.log(
                    f'SELL EXECUTED, Price: {order.executed.price:.2f}, '
                    f'Cost: {order.executed.value:.2f}, Comm:
{order.executed.comm:.2f}'
                )
            self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order Canceled/Margin/Rejected, Status:
{order.getstatusname()}')
```

```
        self.order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}')

    def next(self):
        if self.order or len(self) < self.params.adx_period:
            return

        if self.adx[0] < self.params.adx_threshold:
            self.log(f"Market not trending strongly (ADX: {self.adx[0]:.2f}).
Skipping trade.")
            return

        # Check directional indicators and take trade positions accordingly.
        if self.plusdi[0] > self.minusdi[0]:
            if self.position and self.position.size < 0:
                self.log(
                    f'REVERSING TO LONG at Close {self.dataclose[0]:.2f} '
                    f'(+DI: {self.plusdi[0]:.2f} vs -DI:
{self.minusdi[0]:.2f})'
                )
                self.order = self.buy()  # Reverse short to long.
            elif not self.position:
                self.log(
                    f'GOING LONG at Close {self.dataclose[0]:.2f} '
                    f'(+DI: {self.plusdi[0]:.2f} vs -DI:
{self.minusdi[0]:.2f})'
                )
                self.order = self.buy()
        elif self.minusdi[0] > self.plusdi[0]:
            if self.position and self.position.size > 0:
                self.log(
                    f'REVERSING TO SHORT at Close {self.dataclose[0]:.2f} '
                    f'(-DI: {self.minusdi[0]:.2f} vs +DI:
{self.plusdi[0]:.2f})'
                )
                self.order = self.sell()  # Reverse long to short.
            elif not self.position:
                self.log(
                    f'GOING SHORT at Close {self.dataclose[0]:.2f} '
                    f'(-DI: {self.minusdi[0]:.2f} vs +DI:
{self.plusdi[0]:.2f})'
                )
                self.order = self.sell()
    def stop(self):
        strategy_params = f'ADX({self.params.adx_period},
threshold={self.params.adx_threshold})'
        self.log(f'({strategy_params}) Ending Portfolio Value
{self.broker.getvalue():.2f}', doprint=True)
```

Explanation

- **ADX, +DI, and -DI Initialization:**
  In the `__init__` method, three indicators are initialized with a 14-bar period. These drive the decision-making process by filtering trades based on trend strength and direction.

- **Order Execution Logic:**
  In the `next()` method, before executing any trades, the strategy checks that the ADX is above the threshold. Then, using simple if-else logic, it determines the appropriate action:

  - For a bullish signal (i.e., `+DI > -DI`), the strategy enters long positions.
  - For a bearish signal (i.e., `-DI > +DI`), it enters short positions.
  - If an existing position opposes the new signal, the strategy reverses the position.

2. Enhancing the Strategy

While the basic ADX trend strategy is effective for trending markets, two common issues often arise:

- **Trading in Ranging (Sideways) Markets:**
  When the market is not trending, directional signals may produce false entries and reversals.

- **Risk Management and Exits:**
  Fixed exits or immediate reversals may not protect profits adequately.

To address these, we implemented two enhancements:

1. **Ranging Market Filter Using Bollinger Bands:**
   By incorporating Bollinger Bands, the strategy checks the width of the bands to determine if the market is range-bound. A narrow band (e.g., less than 1% of the current close) suggests low volatility; in such cases, the strategy will skip trading to avoid whipsaws.

2. **Trailing Stop Exits:**
   Instead of using fixed exits or immediate reversals, a trailing stop is employed to

dynamically manage exits. This mechanism allows profits to run while protecting gains if the market reverses.

Enhanced Strategy Code

Below is the enhanced strategy that integrates these improvements:

```
import backtrader as bt
import datetime
import yfinance as yf
import pandas as pd

class ADXTrendStrengthWithFilters(bt.Strategy):
    params = (
        ('adx_period', 14),
        ('adx_threshold', 25),
        ('boll_period', 20),          # Period for Bollinger Bands
        ('boll_devfactor', 2),        # Deviation factor for Bollinger Bands
        ('confirmation_bars', 3),     # Number of bars to confirm reversal
signal
        ('trail_percent', 0.02),      # Trailing stop percentage (2% by
default)
        ('printlog', True),
    )

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.datetime(0)
            print(f"{dt.isoformat()} - {txt}")

    def __init__(self):
        self.dataclose = self.datas[0].close

        # Initialize ADX, PlusDI, and MinusDI
        self.adx = bt.indicators.ADX(self.datas[0],
period=self.params.adx_period)
        self.plusdi = bt.indicators.PlusDI(self.datas[0],
period=self.params.adx_period)
        self.minusdi = bt.indicators.MinusDI(self.datas[0],
period=self.params.adx_period)

        # Bollinger Bands to measure market range
        self.boll = bt.indicators.BollingerBands(self.datas[0],

period=self.params.boll_period,

devfactor=self.params.boll_devfactor)
```

```
        # Counter to confirm reversal signals
        self.reversal_counter = 0

        # Track market and trailing orders
        self.order = None
        self.trail_order = None

    def notify_order(self, order):
        # Skip processing if order is submitted or accepted
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f"BUY EXECUTED at {order.executed.price:.2f}")
            elif order.issell():
                self.log(f"SELL EXECUTED at {order.executed.price:.2f}")
            self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f"Order Canceled/Margin/Rejected: 
{order.getstatusname()}")

        # Reset order reference if it was our order.
        if order == self.order:
            self.order = None
        if order == self.trail_order:
            self.trail_order = None

    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f"Trade Profit: GROSS {trade.pnl:.2f}, NET 
{trade.pnlcomm:.2f}")

    def cancel_trail(self):
        if self.trail_order:
            self.log("Canceling active trailing stop order.")
            self.cancel(self.trail_order)
            self.trail_order = None

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

        # If there is an open position, ensure trailing stop order is active.
        if self.position:
            if not self.trail_order:
                if self.position.size > 0:
                    self.log(f"Placing trailing stop order for long position
at {self.dataclose[0]:.2f}")
                    self.trail_order = self.sell(
                        exectype=bt.Order.StopTrail,
                        trailpercent=self.params.trail_percent)
                elif self.position.size < 0:
                    self.log(f"Placing trailing stop order for short position
at {self.dataclose[0]:.2f}")
                    self.trail_order = self.buy(
                        exectype=bt.Order.StopTrail,
                        trailpercent=self.params.trail_percent)
            # If in position, do not open a new trade.
            return

        # Ensure sufficient data is available
        if len(self) < max(self.params.adx_period, self.params.boll_period):
            return

        # Check if the market is trending strongly via ADX
        if self.adx[0] < self.params.adx_threshold:
            self.log(f"Low ADX ({self.adx[0]:.2f}). Market trending weakly.
Skipping trade.")
            self.reversal_counter = 0  # Reset confirmation counter if market
weakens
            return

        # Check for range-bound market using Bollinger Band width.
        boll_width = self.boll.top[0] - self.boll.bot[0]
        if boll_width < 0.01 * self.dataclose[0]:  # Example threshold: 1% of
price
            self.log(f"Bollinger Bands narrow ({boll_width:.2f}). Market is
range-bound. Skipping trade.")
            self.reversal_counter = 0
            return

        # Define directional signal
        long_signal = self.plusdi[0] > self.minusdi[0]
        short_signal = self.minusdi[0] > self.plusdi[0]

        # Confirm the directional signal persists for a number of bars
        if (long_signal and self.position.size <= 0) or (short_signal and
self.position.size >= 0):
            self.reversal_counter += 1
        else:
            self.reversal_counter = 0

        # Check if confirmation condition is met
        if self.reversal_counter < self.params.confirmation_bars:
            return  # Wait for more confirmation

        # Cancel any active trailing stop before entering a new trade.
        self.cancel_trail()

        # Execute orders when confirmation condition is met
        if long_signal:
            if self.position and self.position.size < 0:
                self.log(f"Reversing to long at {self.dataclose[0]:.2f}")
                self.order = self.buy()  # Reverse position (buy closes short 
and opens long)
            elif not self.position:
                self.log(f"Going long at {self.dataclose[0]:.2f}")
                self.order = self.buy()
        elif short_signal:
            if self.position and self.position.size > 0:
                self.log(f"Reversing to short at {self.dataclose[0]:.2f}")
                self.order = self.sell()  # Reverse position (sell closes 
long and opens short)
            elif not self.position:
                self.log(f"Going short at {self.dataclose[0]:.2f}")
                self.order = self.sell()

    def stop(self):
        self.log(f"Ending Portfolio Value: {self.broker.getvalue():.2f}", 
doprint=True)
```

Explanation of Enhancements

1. **Filtering Ranging Markets with Bollinger Bands:**
    - The Bollinger Bands indicator (`self.boll`) is used to calculate the width between the upper and lower bands.
    - If the width is too narrow (less than 1% of the current close), it suggests the market is in a range-bound state; hence, the strategy skips taking new trades in order to avoid false signals.
2. **Trailing Stop Exit for Better Risk Management:**
    - A new parameter (`trail_percent`) defines the trailing stop distance (e.g., 2%).
    - When a position is open and no trailing stop order exists, the strategy creates a trailing stop order:
        - For a long position, a trailing stop sell order is issued.

        - For a short position, a trailing stop buy order is issued.

    - Prior to entering a new trade (especially when reversing positions), any active trailing stop orders are canceled using the `cancel_trail()` method. This ensures that exit orders from prior trades do not interfere with new trade logic.

3. **Reversal Confirmation Mechanism:**

    - The strategy uses a confirmation counter (`self.reversal_counter`) to ensure that the directional signal persists over a few bars (set by `confirmation_bars`). This helps avoid premature or false reversals.

3. Backtesting and Analysis

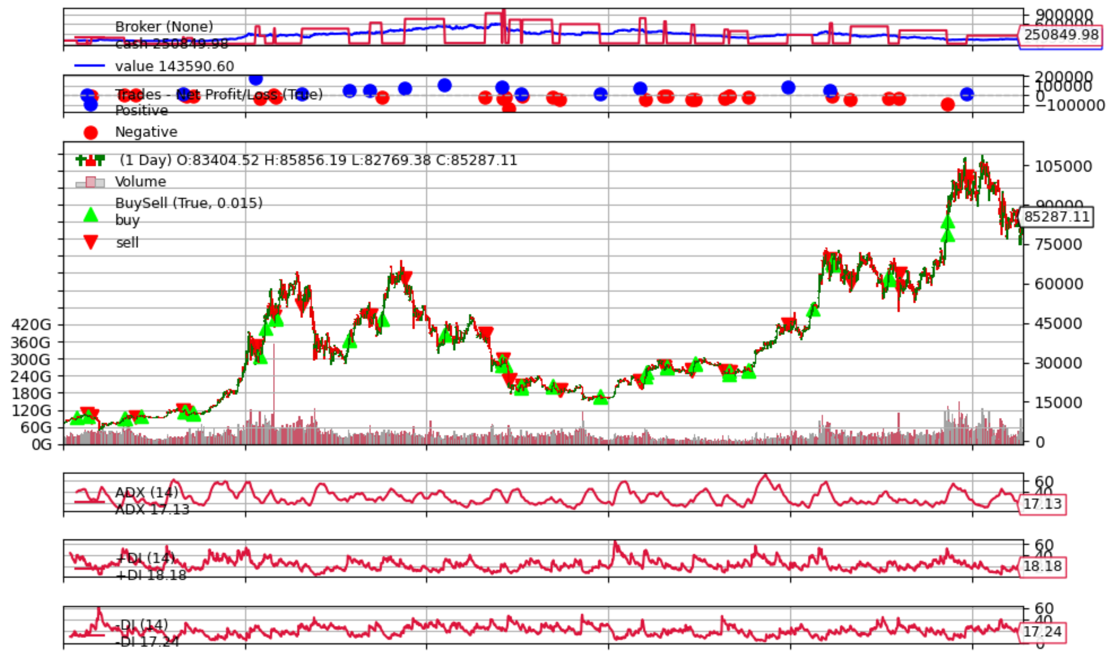
Both strategies are then fed to Backtrader for backtesting. In the provided code, historical price data for BTC-USD is downloaded using the `yfinance` library. The backtest sets up an initial portfolio, commissions, position sizing, and several analyzers such as the Sharpe Ratio, total return, maximum drawdown, and trade statistics to help evaluate the performance of the strategy.

Sample Backtest Setup

```
def run_backtest():
    cerebro = bt.Cerebro()
    cerebro.addstrategy(ADXTrendStrengthWithFilters, printlog=False)

    # Data Loading: Using BTC-USD as an example.
    ticker = 'BTC-USD'
    current_date = datetime.datetime(2025, 4, 14)
    start_date = datetime.datetime(2020, 1, 1)
    end_date = current_date - datetime.timedelta(days=1)

    print(f"Fetching data for {ticker} from {start_date.date()} to
{end_date.date()}")
    try:
        data_df = yf.download(ticker, start=start_date, end=end_date,
progress=False)
        if data_df.empty:
            raise ValueError("No data fetched. Check ticker symbol or date
range.")

        data_df.columns = data_df.columns.droplevel(1).str.lower()
        data_df.index = pd.to_datetime(data_df.index)
        data = bt.feeds.PandasData(dataname=data_df)
        cerebro.adddata(data)
    except Exception as e:
        print(f"Error fetching or processing data: {e}")
        return

    initial_cash = 100000.0
    cerebro.broker.setcash(initial_cash)
    cerebro.broker.setcommission(commission=0.001)
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

    # Add performance analyzers.
    cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe_ratio',
timeframe=bt.TimeFrame.Days, riskfreerate=0.0)
    cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')

    print(f"Starting Portfolio Value: {initial_cash:.2f}")
    results = cerebro.run()
    print(f"Final Portfolio Value: {cerebro.broker.getvalue():.2f}")

    # Print and analyze results...
    try:
        cerebro.plot(iplot=False, style='candlestick', subplot=True)
    except Exception as e:
        print(f"Could not generate plot. Error: {e}.")
```

3. Sample Results

Base Strategy



Pasted image 20250414071750.png

Pasted image 20250414071750.png

```
Starting Portfolio Value: 100000.00
Final Portfolio Value: 143590.60
--------------------------------------------------
Strategy Analysis (ADX Trend Strength Strategy - BTC-USD):
--------------------------------------------------
Sharpe Ratio: 0.021
Total Return: 36.18%
Annualized Return: 4.84%
Max Drawdown: 80.89%
Total Trades: 47
Winning Trades: 15
Losing Trades: 31
Win Rate: 31.91%
Avg Winning Trade: 56464.96
Avg Losing Trade: -26395.70
Profit Factor: 1.0350837315910437
--------------------------------------------------
```

Analysis

1. **Return Performance:**
   The overall portfolio experienced a 36% total return, which is positive. However, the annualized return of 4.84% is relatively modest. This suggests that while the strategy can capture gains, the growth is not aggressively outperforming market benchmarks on an annual basis.

2. **Risk-Adjusted Return:**
   The Sharpe Ratio is extremely low at 0.021, indicating that the strategy's returns are not significantly compensating for the risk taken. Investors typically look for higher Sharpe ratios to ensure the excess return adequately compensates for volatility.

3. **Drawdown Concerns:**
   A maximum drawdown of 80.89% is alarmingly high. This means there were periods when the portfolio value plunged significantly, which could be intolerable for many investors. High drawdowns can also lead to psychological stress and potential forced exits from the strategy.

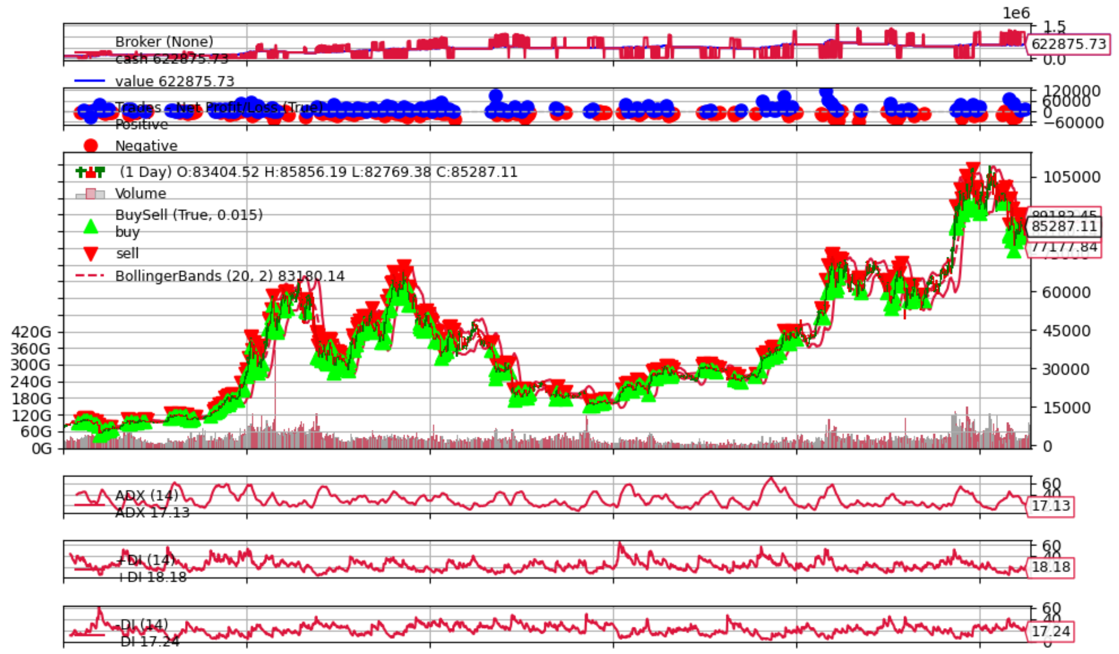
4. **Trade Frequency and Quality:**

   - **Trade Count and Win Rate:**
     With only 47 trades over the backtest period and a win rate of approximately 32%, the strategy generates fewer winning trades compared to losing ones.

   - **Profit vs. Loss:**
     On average, winning trades are much larger ($56K) than the losing trades ($26K), which helps the strategy break even overall. However, this reliance

on a small number of big winners can be problematic, as missing or delaying these trades can significantly hurt performance.

- **Profit Factor:**
  A profit factor of about 1.035 indicates that for every dollar lost, about $1.03 is gained. This slim margin suggests that the strategy is just barely profitable on a per-dollar risk basis. ### Enhanced Strategy



Pasted image 20250414072340.png

Pasted image 20250414072340.png

```
Starting Portfolio Value: 100000.00
Final Portfolio Value: 622875.73
--------------------------------------------------
Strategy Analysis (ADX Trend Strength with Trailing Stop - BTC-USD):
--------------------------------------------------
Sharpe Ratio: 0.054
Total Return: 182.92%
Annualized Return: 26.99%
Max Drawdown: 37.54%
Total Trades: 321
Winning Trades: 154
Losing Trades: 167
Win Rate: 47.98%
Avg Winning Trade: 15863.18
Avg Losing Trade: -11497.33
Profit Factor: 1.2723234247163178
--------------------------------------------------
```

Key Enhancements Impact

1. **Enhanced Returns and Lower Drawdowns:**
   - The significant boost in total and annualized returns is one of the most compelling outcomes.
   - The dramatic reduction in maximum drawdown—from over 80% in the basic strategy to 37.54%—shows that the introduction of trailing stops and filtering out ranging markets helps limit large losses during volatile periods.
2. **Improved Trade Consistency:**
   - The enhanced strategy executes more trades (321 vs. 47), which provides a more statistically meaningful performance record.
   - The win rate improvement from 31.91% to nearly 48% indicates better selection of trades, likely due to the extra filtering (via Bollinger Bands) and the confirmation mechanism that reduces false signals.
3. **Risk Management with Trailing Stops:**
   - Incorporating trailing stops has contributed to locking in profits while allowing winners to run, without exposing the portfolio to significant reversals.
   - By canceling trailing stops before reversing trades, the strategy avoids conflicts between exit orders and new trade signals.

Conclusion

The enhancement from a basic ADX Trend Strength Strategy to a more refined version with a Bollinger Band ranging filter and trailing stop exit represents a thoughtful approach to mitigating common trading pitfalls:

- **Filtering Out Ranging Markets:** Prevents false entries when the market lacks clear direction.
- **Using Trailing Stops:** Enables dynamic exit management, letting profits run while limiting losses.
- **Confirmation Mechanisms:** Ensure that trade signals are genuine and sustained.

By iterating through these enhancements and validating performance with thorough backtesting, traders can achieve a more robust strategy that adapts better to changing market conditions.

This approach shows how combining multiple technical indicators and advanced order management techniques can lead to a more refined and potentially profitable trading system.

Enhancing Bollinger Bands Mean-Reversion with ADX and RSI Transforming $7000 Loss into $57000 Profit

Basic Strategy

Mean-reversion is a popular trading strategy that assumes asset prices tend to revert to their historical mean. In the context of Bollinger Bands, the strategy often uses the lower and upper bands as triggers for trading:

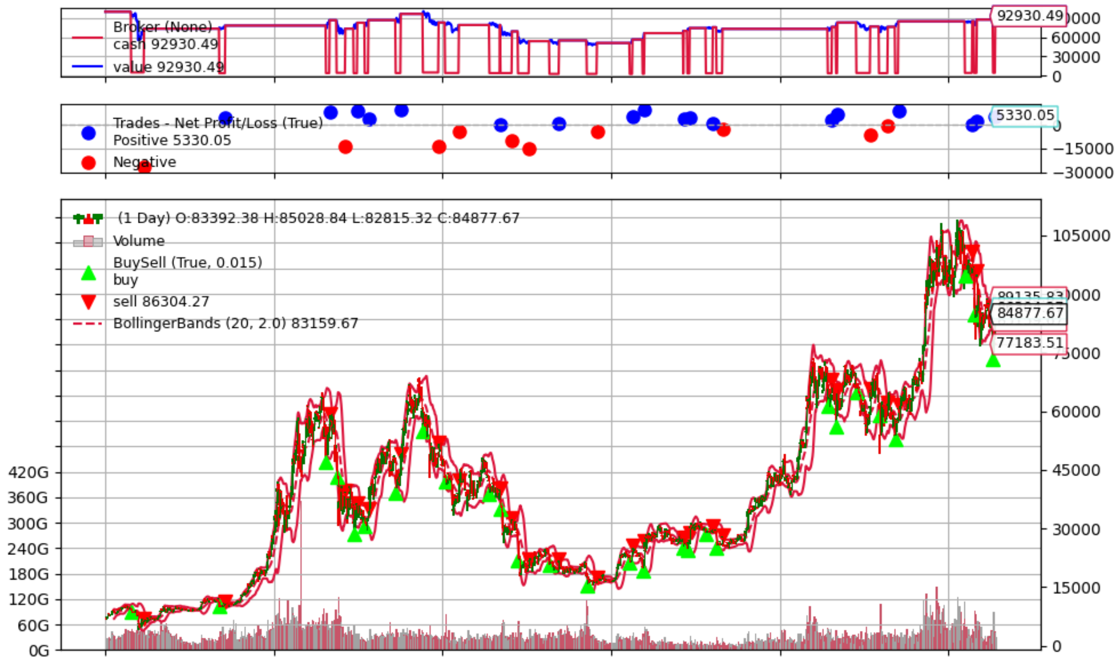
- **Entry:**
    - **Buy:** When the closing price falls below the lower Bollinger Band, the market is deemed "oversold" and is expected to revert toward the mean (the middle band).
    - **(Optional Short Entry):** Similarly, when the price rises above the upper Bollinger Band, the market is considered "overbought."
- **Exit:**
    - **Close Long:** The position is exited when the price moves back above the middle band.
    - **(Optional Cover Short):** For short positions, the exit occurs when the price falls back below the middle band.

Basic Strategy Code:

Here's how I implemented the basic mean-reversion strategy using Backtrader:

```
class BBMeanReversion(bt.Strategy):
    params = (
        ('bb_period', 20),
        ('bb_devfactor', 2.0),
        ('printlog', True),
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.bband = bt.indicators.BollingerBands(
            self.datas[0],
            period=self.params.bb_period,
            devfactor=self.params.bb_devfactor
        )

    def next(self):
        # Entry condition: Only enter if not in market
        if not self.position:
            if self.dataclose[0] < self.bband.lines.bot[0]:
                self.order = self.buy()
        # Exit condition: if in a long position and price crosses above the
mid band
        else:
            if self.dataclose[0] > self.bband.lines.mid[0]:
                self.order = self.close()
```

Performance Observations:
When applied on a one-year dataset, the basic Bollinger Bands mean-reversion strategy can yield a high return. However, over other years the same strategy might result in modest returns or even losses. This inconsistency is common with straightforward mean-reversion approaches because markets can experience strong trends during which price "extremes" do not reliably reverse. For example let's look at the cumulative performance from 2020 onward:



let's look at some statistics:

```
Starting Portfolio Value: 100000.00
2025-04-12 - (BB Period 20, BB DevFact 2.0) Ending Value 92930.49
Final Portfolio Value: 92930.49
Sharpe Ratio: 0.008
Total Return: -7.33%
Annualized Return: -0.95%
Max Drawdown: 53.27%
```

```
Total Trades: 28
Winning Trades: 18
Losing Trades: 10
Win Rate: 64.29%
Avg Winning Trade: 4947.69
Avg Losing Trade: -9612.79
Profit Factor: 0.9264571927698776
```

- **Overall Performance:**
  Starting with $100,000, the portfolio ended at approximately $92,930—reflecting a total return of -7.33% and an annualized loss of about -0.95%. This indicates that the strategy, despite a relatively high win ratio, did not manage to generate net profits over the period.

- **Risk Metrics:**
  The Sharpe Ratio is almost negligible at 0.008, suggesting that the returns were not sufficient to compensate for the risk taken. Additionally, the maximum drawdown reached over 53%, a significant decline that points to high volatility and periods of substantial losses.

- **Trade-Level Insights:**
  Although 18 out of 28 trades were winners (win rate of 64.29%), the average winning trade ($4,947.69) was considerably smaller than the average losing trade (-$9,612.79). This imbalance contributed to a profit factor of only 0.93, meaning that losing trades cost more than the gains from winning trades.

Enhancing the Strategy with Filters

To address the limitations of a basic mean-reversion strategy, multiple filters can be introduced:

- **ADX Filter:** The Average Directional Index (ADX) indicates the strength of a trend. By requiring the ADX to be below a threshold (e.g., 25), the strategy avoids taking positions during strong trending periods, where mean reversion is less likely.

- **RSI Filter:** The Relative Strength Index (RSI) adds an extra layer of confirmation. It ensures that long entries occur in an oversold state (RSI below 30) and, if short selling is enabled, short entries occur in an overbought condition (RSI above 70).

These filters help ensure that trades are initiated only when the conditions favor mean-reversion rather than trend continuation.

Enhanced Strategy Code:

Below is the modified version of the basic strategy. Notice that both the entry and exit conditions now incorporate the ADX and RSI filters:

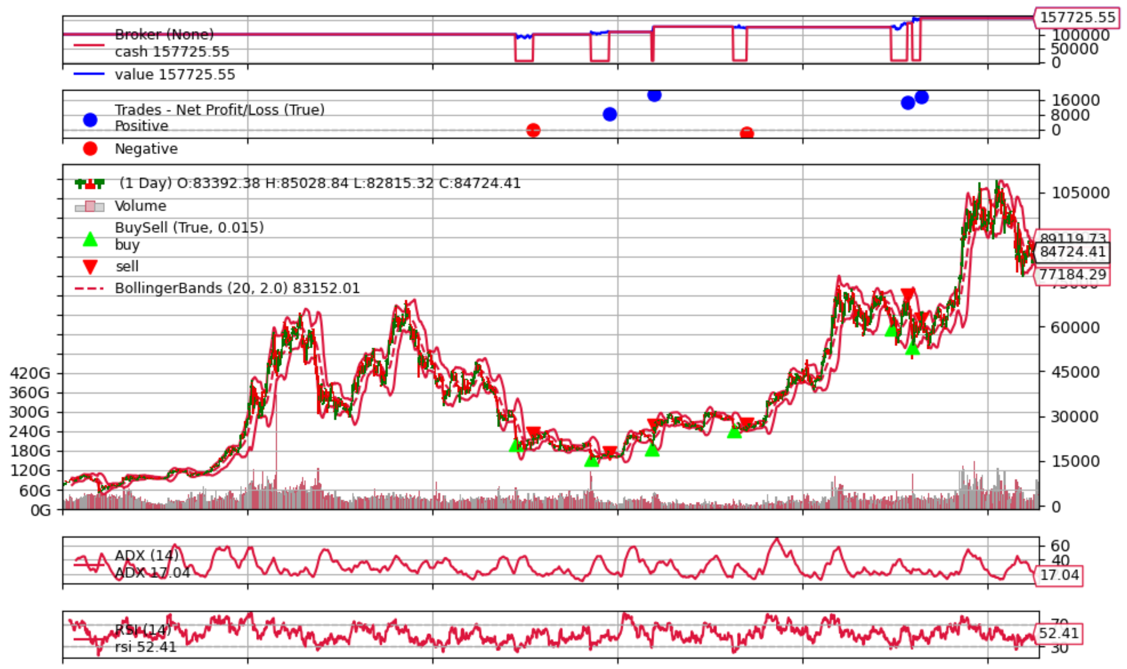
```
class EnhancedBBMeanReversion(bt.Strategy):
    params = (
        ('bb_period', 20),
        ('bb_devfactor', 2.0),
        ('adx_period', 14),
        ('adx_threshold', 25),
        ('rsi_period', 14),
        ('rsi_lower', 30),
        ('rsi_upper', 70),
        ('printlog', True),
    )

    def __init__(self):
        self.dataclose = self.datas[0].close
        self.bband = bt.indicators.BollingerBands(
            self.datas[0],
            period=self.params.bb_period,
            devfactor=self.params.bb_devfactor
        )
        # ADX Filter for non-trending conditions
        self.adx = bt.indicators.ADX(self.datas[0],
period=self.params.adx_period)
        # RSI Filter for overbought/oversold conditions
        self.rsi = bt.indicators.RSI(self.datas[0],
period=self.params.rsi_period)

    def next(self):
        # Skip if ADX indicates a trending market
        if self.adx[0] >= self.params.adx_threshold:
            self.log(f"Market trending (ADX {self.adx[0]:.2f} >=
{self.params.adx_threshold}), skipping new entries.")
            return

        # Long Entry Condition with multiple confirmations:
        # Price below lower band and RSI indicates oversold conditions.
        if not self.position:
            if (self.dataclose[0] < self.bband.lines.bot[0] and
                self.rsi[0] < self.params.rsi_lower):
                self.log(f'LONG ENTRY SIGNAL: Close: {self.dataclose[0]:.2f},
Lower BB: {self.bband.lines.bot[0]:.2f}, RSI: {self.rsi[0]:.2f}')
                self.order = self.buy()
        # Exit Condition: long positions exit when price crosses above the
middle band.
        else:
            if self.position.size > 0 and self.dataclose[0] >
self.bband.lines.mid[0]:
```

```
                self.log(f'EXIT LONG SIGNAL: Close: {self.dataclose[0]:.2f}, 
Mid BB: {self.bband.lines.mid[0]:.2f}')
                self.order = self.close()
```

Key Enhancements:

- **Refined Entry Criteria:**
  The enhanced strategy now incorporates both ADX and RSI filters to fine-tune entry signals. First, it verifies that `self.adx[0] < self.params.adx_threshold`, ensuring that trades are only considered when the market is not strongly trending. Then, it checks that the RSI is in an oversold condition (i.e., `self.rsi[0] < self.params.rsi_lower`) before executing a buy order. This dual-filter approach prevents entering trades in environments prone to prolonged trends or oversold conditions that might not revert as expected.
- **Optimized Exit Conditions:**
  The exit logic similarly benefits from the ADX filter by requiring that the market remains non-trending before closing a position. This helps avoid premature exits during volatile periods and reduces the risk of locking in losses if the trend temporarily reverses. In essence, the exit is triggered only when the price crosses above the middle Bollinger Band under favorable (i.e., non-trending) market conditions.

Overall Impact on Performance:
While the basic mean-reversion strategy can deliver high returns in periods conducive to reversion, it typically suffers from significant variability and higher risk during trending markets. The enhanced version, by integrating both ADX and RSI filters, may yield slightly lower returns during peak mean-reversion years, but it offers substantially improved stability. This results in smaller drawdowns and a more robust risk profile over time. Let’s examine the performance metrics from 2020 onward:



Pasted image 20250412171816.png

Pasted image 20250412171816.png

let’s check the new result statistics:

```
Starting Portfolio Value: 100000.00
2025-04-12 - (BB Period 20, BB DevFact 2.0, ADX Thresh 25, RSI [< 30 / > 70])
Ending Value 157725.55
Final Portfolio Value: 157725.55
Sharpe Ratio: 0.033
Total Return: 45.57%
Annualized Return: 6.13%
Max Drawdown: 14.98%
Total Trades: 6
Winning Trades: 4
Losing Trades: 2
Win Rate: 66.67%
Avg Winning Trade: 15023.39
Avg Losing Trade: -1184.00
Profit Factor: 25.377326687383313
```

- **Overall Return:**
  The ending portfolio value of $157,725.55 translates to a total return of 45.57% with an annualized return of 6.13%—a strong performance compared to the previous negative return.

- **Risk-Adjusted Return:**
  Although the Sharpe Ratio remains modest at 0.033, the dramatic reduction in

maximum drawdown—from 53.27% to 14.98%—indicates that the enhanced filtering not only improved returns but also significantly reduced downside risk.

- **Trade Efficiency:**
  With only 6 trades executed:
  - **Win Rate:** 66.67% (4 winners vs. 2 losers) suggests that the filters helped in identifying higher-quality trades.
  - **Trade Sizing:** The average winning trade delivered approximately $15,023, whereas the average loss was limited to about $1,184. This imbalance (reflected in a profit factor of 25.38) shows that the winners more than compensated for the losses.
- **Method Enhancements:**
  The additional filters (ADX threshold at 25 and RSI conditions at [< 30 / > 70]) appear to have refined entry and exit signals, allowing the strategy to avoid unfavorable conditions and capture higher conviction trades.

Conclusion

Introducing filters such as ADX and RSI to the Bollinger Bands mean-reversion strategy provides a more refined trading signal—one that avoids unfavorable conditions and captures high-conviction entries. While the basic version might produce high returns in the right market environment, it can also suffer substantial losses in trending periods. The enhanced strategy, on the other hand, demonstrates that combining multiple indicators can lead to a more stable and attractive risk-adjusted performance.

By integrating both ADX and RSI filters, traders can achieve a balance between capturing high-probability mean-reversion trades and maintaining strict risk control—resulting in fewer trades but with significantly improved outcomes.

Enhancing RSI Mean-Reversion with ATR and ADX from $48000 to $131000 Profits

Mean-reversion strategies hinge on the idea that prices will revert to an average level over time. One popular indicator for such a strategy is the Relative Strength Index (RSI). In this article, we start with a basic RSI mean-reversion strategy, then describe how we can improve performance by filtering out trades during trending conditions using the Average Directional Movement Index (ADX) and the Average True Range (ATR). Generally, you should always think of the market conditions that your strategy works well in and then try to reduce false signals by filtering out undesirable markets.

The Base RSI Mean-Reversion Strategy

Strategy Concept

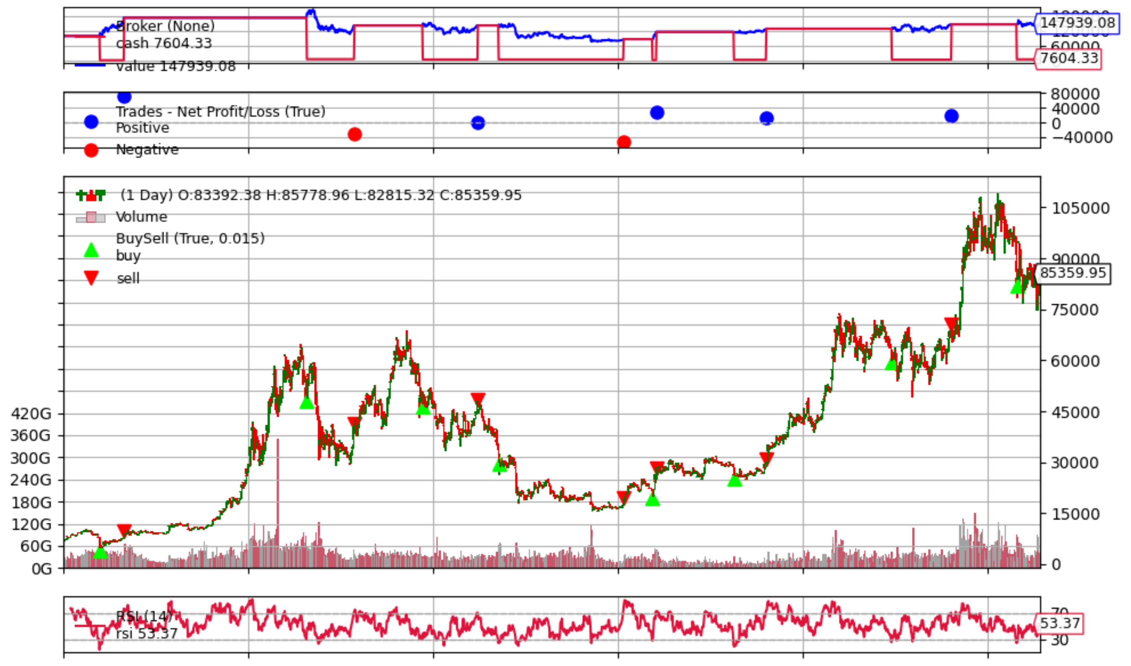
The basic RSI mean-reversion strategy is built around these core ideas:

- **RSI Indicator:**
  The RSI is a momentum indicator that measures the speed and change of price movements. It typically oscillates between 0 and 100.
- **Entry Signal:**
  When the RSI falls below an oversold threshold (commonly 30), the strategy interprets the market as oversold and a potential reversal is expected, triggering a buy signal.
- **Exit Signal:**
  When the RSI exceeds an overbought threshold (commonly 70), the market is considered overbought. The strategy then initiates a sell, capturing profits from the expected reversal.

Base Strategy Code

Below is the essential code for the basic RSI mean-reversion strategy written using Backtrader and yfinance libraries:

```
import backtrader as bt
import yfinance as yf
import datetime
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib qt5

class RsiStrategy(bt.Strategy):
    params = (
        ('rsi_period', 14),
        ('rsi_overbought', 70),
        ('rsi_oversold', 30),
        ('printlog', True),
    )

    def __init__(self):
        # Reference to the close price series
        self.dataclose = self.datas[0].close
        # For tracking orders
        self.order = None
        # Add RSI indicator
        self.rsi = bt.indicators.RSI(self.datas[0],
period=self.params.rsi_period)

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()} - {txt}')

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return  # Do nothing for pending orders

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f}, '
                         f'Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}')
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
'
                         f'Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}')
            self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None

    def next(self):
        # Skip new orders if an order is pending.
        if self.order:
            return

        # If we are not in the market, check for oversold condition for buy
entry.
        if not self.position:
            if self.rsi < self.params.rsi_oversold:
                self.log(f'BUY CREATE, {self.dataclose[0]:.2f}')
                self.order = self.buy()
        else:
            # If already in the market, check for overbought condition for
exit.
            if self.rsi > self.params.rsi_overbought:
                self.log(f'SELL CREATE, {self.dataclose[0]:.2f}')
                self.order = self.sell()

if __name__ == '__main__':
    cerebro = bt.Cerebro()
    cerebro.addstrategy(RsiStrategy, printlog=False)
    ticker = 'BTC-USD'
    start_date = datetime.datetime(2020, 1, 1)
    end_date = datetime.datetime(2025, 12, 31)

    data_df = yf.download(ticker, start=start_date, end=end_date,
progress=False)
    if data_df.empty:
        raise ValueError("No data fetched. Check ticker symbol or date
range.")

    # Adjust DataFrame for Backtrader if needed
    data_df.columns = data_df.columns.droplevel(1).str.lower()
    data_df.index = pd.to_datetime(data_df.index)
    data = bt.feeds.PandasData(dataname=data_df)
    cerebro.adddata(data)

    cerebro.broker.set_cash(100000.0)
    cerebro.broker.setcommission(commission=0.001)
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

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

    plt.rcParams["figure.figsize"] = (10, 6)
    figure = cerebro.plot(style='candlestick', barup='green', bardown='red',
iplot=False, show=False)[0][0]
    plt.tight_layout()
    figure.show()
```



###

Performance of the Base Strategy

Let’s take a look at backtest performance for bitcoin from 2020 onward:

```
Starting Portfolio Value: 100000.00
Final Portfolio Value: 147939.08
Sharpe Ratio: 0.020
Total Return: 39.16%
Annualized Return: 5.25%
Max Drawdown: 62.21%
Total Trades: 8
Winning Trades: 5
Losing Trades: 2
Win Rate: 62.50%
Avg Winning Trade: 26152.38
Avg Losing Trade: -42397.02
Profit Factor: 1.5421118842358195
```

These numbers indicate moderate growth, though the high drawdown signals the strategy’s vulnerability during trending markets where mean reversion assumptions fail.

Improved Strategy Logic: Filtering Out Trending Conditions

The Challenge with Trending Markets

Mean reversion strategies work best in range-bound markets. However, during trending phases, prices often continue in one direction rather than reverting, which leads to false RSI signals. To address this, we can use:

- **ADX (Average Directional Movement Index):**
  Measures the strength of a trend. A high ADX (above a set threshold) indicates a trending market.

- **ATR (Average True Range):**
  Gauges market volatility. A high ATR-to-price ratio indicates significant volatility.

Improved Strategy Concept

For a mean-reversion strategy, the logic now becomes:

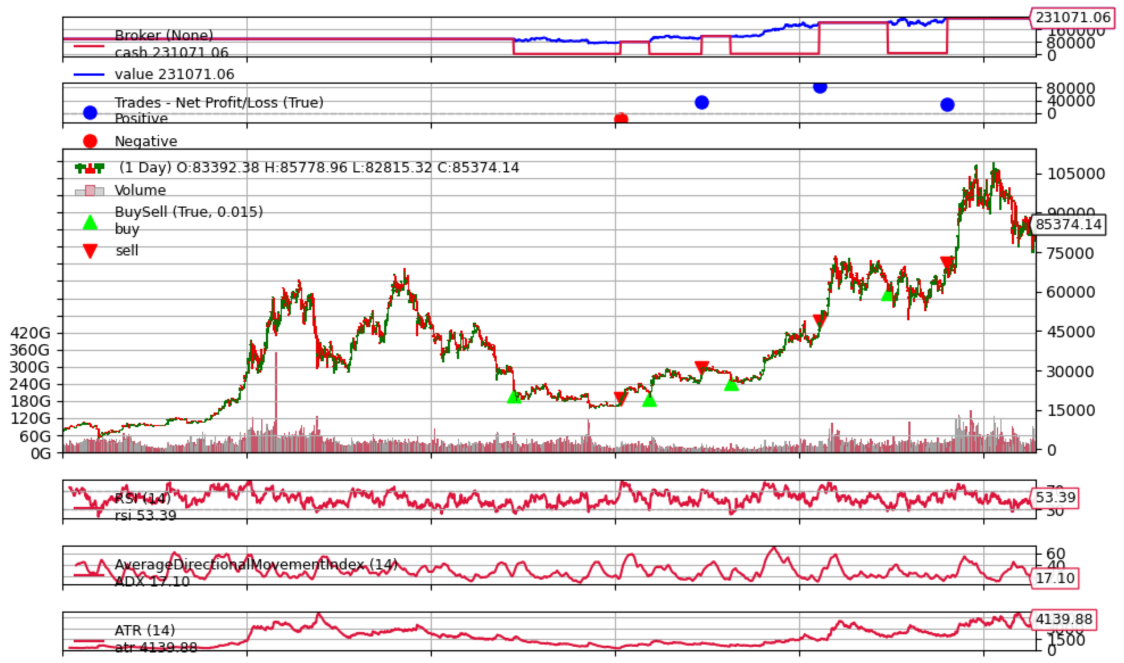
- **Skip trades** if both ADX and ATR confirm that the market is trending and volatile.

- **Take trades only** when the market is range-bound—i.e., when the ADX is below the threshold or the ATR/close ratio is low.

Enhanced Strategy Code

The following code integrates ADX and ATR filters to prevent entry or exit signals during trending conditions:

```
import backtrader as bt
import yfinance as yf
import datetime
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib qt5

class RsiStrategyWithTrendFilters(bt.Strategy):
    params = (
        ('rsi_period', 14),
        ('rsi_overbought', 70),
        ('rsi_oversold', 30),
        ('adx_period', 14),
        ('adx_threshold', 25),         # A high ADX indicates trending
conditions.
        ('atr_period', 14),
        ('atr_ratio_threshold', 0.02),   # ATR-to-Price ratio threshold.
        ('printlog', True),
    )
    def __init__(self):
        self.dataclose = self.datas[0].close
        self.order = None
        self.rsi = bt.indicators.RSI(self.datas[0],
period=self.params.rsi_period)
        self.adx =
bt.indicators.AverageDirectionalMovementIndex(self.datas[0],
period=self.params.adx_period)
        self.atr = bt.indicators.ATR(self.datas[0],
period=self.params.atr_period)

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()} - {txt}')

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f}, '
                         f'Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}')
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
'
                         f'Cost: {order.executed.value:.2f}, Comm
{order.executed.comm:.2f}')
            self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None

    def next(self):
        if self.order:
            return

        # Calculate the ATR-to-Price ratio for the current bar
        atr_ratio = self.atr[0] / self.dataclose[0]

        # Determine trending conditions:
        trend_confirmed = self.adx[0] >= self.params.adx_threshold
        volatile_enough = atr_ratio >= self.params.atr_ratio_threshold

        # For a mean-reversion strategy, skip trading in trending markets.
        if trend_confirmed and volatile_enough:
            return

        # If we're not in the market, consider buying when RSI is oversold.
        if not self.position:
            if self.rsi < self.params.rsi_oversold:
                self.log(f'BUY CREATE at {self.dataclose[0]:.2f} | RSI:
{self.rsi[0]:.2f}')
                self.order = self.buy()
        else:
            # If in the market, consider selling when RSI is overbought.
            if self.rsi > self.params.rsi_overbought:
                self.log(f'SELL CREATE at {self.dataclose[0]:.2f} | RSI:
{self.rsi[0]:.2f}')
                self.order = self.sell()

if __name__ == '__main__':
    cerebro = bt.Cerebro()
    cerebro.addstrategy(RsiStrategyWithTrendFilters, printlog=False)

    ticker = 'BTC-USD'
    start_date = datetime.datetime(2020, 1, 1)
    end_date = datetime.datetime(2025, 12, 31)
    data_df = yf.download(ticker, start=start_date, end=end_date,
progress=False)
    if data_df.empty:
        raise ValueError("No data fetched. Check ticker symbol or date
range.")

    data_df.columns = data_df.columns.droplevel(1).str.lower()
    data_df.index = pd.to_datetime(data_df.index)
    data = bt.feeds.PandasData(dataname=data_df)
    cerebro.adddata(data)

    cerebro.broker.set_cash(100000.0)
    cerebro.broker.setcommission(commission=0.001)
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

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

    plt.rcParams["figure.figsize"] = (10, 6)
    figure = cerebro.plot(style='candlestick', barup='green', bardown='red',
iplot=False, show=False)[0][0]
    plt.tight_layout()
    figure.show()
```



###

Improvements Observed

Now let’s see the performance of the enhanced strategy:

```
Starting Portfolio Value: 100000.00
Final Portfolio Value: 231071.06
Sharpe Ratio: 0.038
Total Return: 83.76%
Annualized Return: 11.56%
Max Drawdown: 33.76%
Total Trades: 4
Winning Trades: 3
Losing Trades: 1
Win Rate: 75.00%
Avg Winning Trade: 49901.91
Avg Losing Trade: -18634.66
Profit Factor: 8.03372435292996
```

The improved strategy increases returns and lowers risk by filtering out trades during trending and volatile conditions. It takes fewer trades, but they are more selective and in environments where mean reversion is more likely to be successful.

Conclusion

By starting with a basic RSI mean-reversion strategy and then improving it with ADX and ATR filters, we’ve demonstrated how to better adapt the strategy to market conditions. This approach helps avoid false signals in trending markets, leading to higher returns and lower drawdown. Such enhancements underscore the importance of tailoring your trading logic to the prevailing market environment for more robust strategy performance.

Whether you're refining an existing system or building a new one, understanding the interplay between indicators, market regimes, and risk management is key to optimizing trading strategies. Happy trading!

How to Properly Bakctest Trading Strategies with Backtrader in Python

Backtesting is more than simply checking if a trading strategy works on historical data — it's about rigorously stress-testing your approach across various market conditions. In this article, we delve into a robust methodology for running multiple randomized backtests over random assets and time periods. This enables you to derive the statistical distribution of both market and strategy returns, revealing insights into the true performance and risk of your trading ideas.

Why Randomized Backtesting?

Traditional backtests often run on a specific asset or a fixed timeframe. However, markets are dynamic, and a strategy that works on one asset or during one market cycle might not translate to another. Randomized backtesting addresses this by:

- **Diversifying Market Conditions:** Running tests over a variety of random time periods helps simulate different market environments.
- **Asset Variability:** By picking random assets from a defined list (for example, major cryptocurrencies), you can assess how robust your strategy is across different instruments.
- **Statistical Confidence:** Collecting a distribution of returns — from both the market and your strategy — allows for rigorous statistical analysis. This helps in understanding not just the average performance but also the variability and risk inherent in your approach.

The Methodology

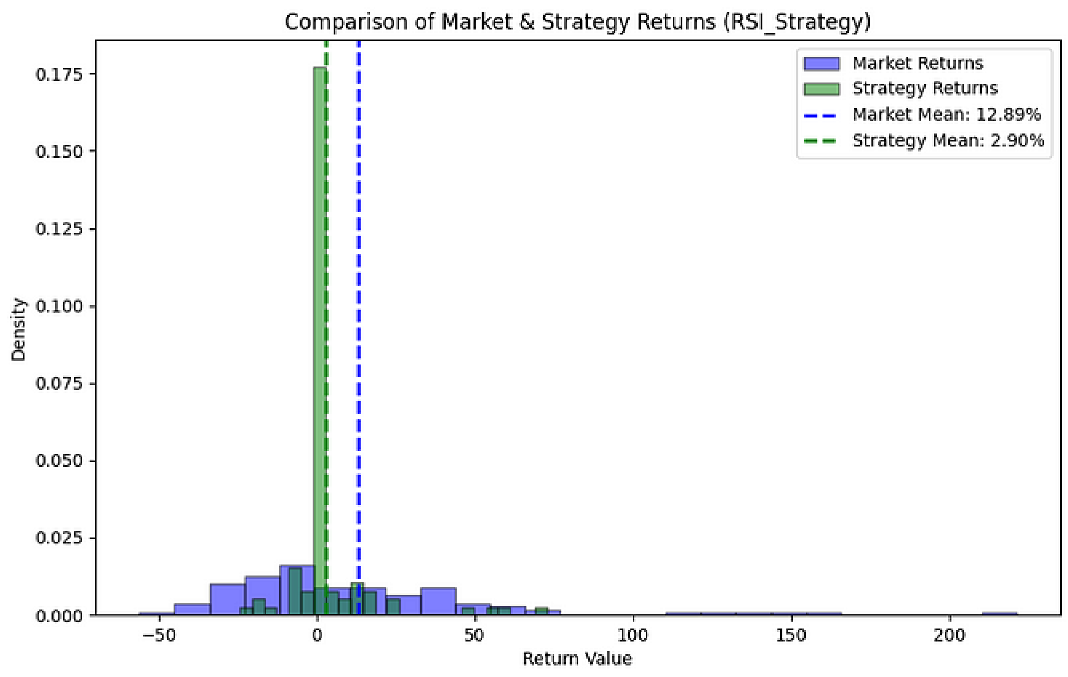
Our approach uses Python's Backtrader library for strategy testing, along with yfinance for historical data acquisition. We add a dash of randomness to both asset selection and window period, then run each simulation in a loop. The aggregated results of multiple backtests provide the density distribution of returns, which is then visualized with overlapping histograms and mean markers.

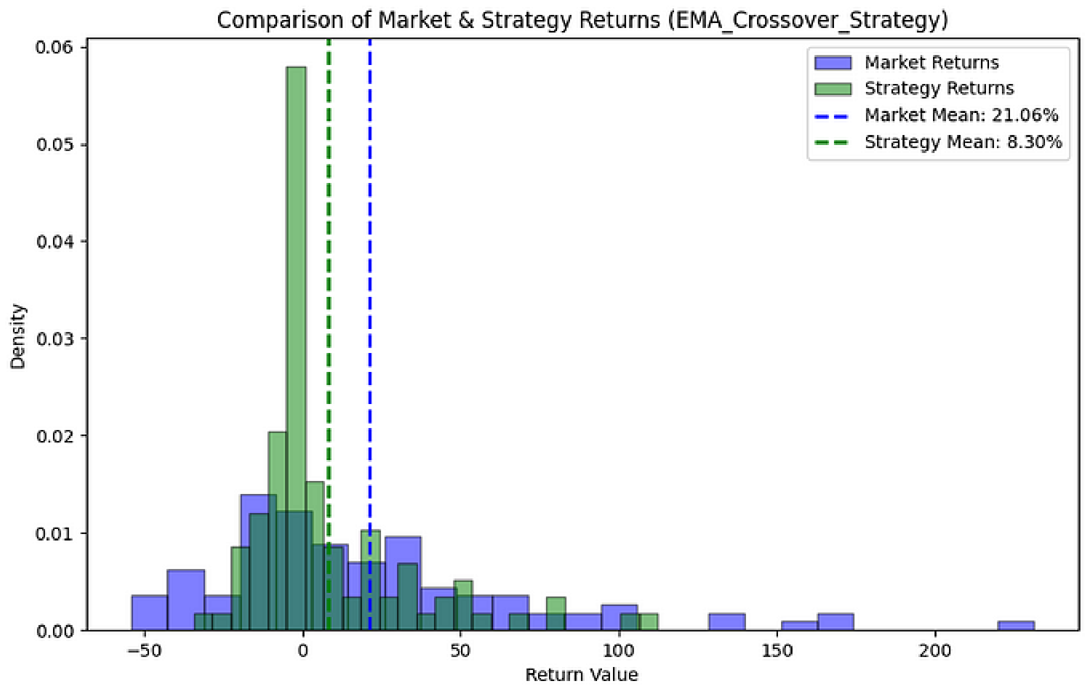
Code Walkthrough

Below is a comprehensive code example that demonstrates this process. The code:

- Randomly chooses a cryptocurrency from a given list.
- Picks a random 2-month window from a defined historical period.
- Runs a backtest with a predefined strategy (here, we use an RSI mean-reversion strategy).
- Collects market returns and strategy returns for each iteration.
- Visualizes the results with density histograms that display the statistical distribution and indicate mean returns.

```
import backtrader as bt
import yfinance as yf
import random
from datetime import datetime, timedelta
import warnings
warnings.filterwarnings(action='ignore')
from tqdm import tqdm
import sys
import os
import contextlib


# Context manager for suppressing verbose outputs from yfinance
@contextlib.contextmanager
def suppress_stdout_stderr():
    with open(os.devnull, 'w') as fnull:
        old_stdout = sys.stdout
        old_stderr = sys.stderr
        try:
            sys.stdout = fnull
            sys.stderr = fnull
            yield
        finally:
            sys.stdout = old_stdout
            sys.stderr = old_stderr

# EMA Crossover Strategy for comparison (optional)
class EMA_Crossover_Strategy(bt.Strategy):
    params = (
        ('short_window', 7),
        ('long_window', 30),
    )
    def __init__(self):
        self.data_close = self.datas[0].close
        self.emas = bt.indicators.ExponentialMovingAverage(self.data_close,
period=self.params.short_window)
        self.emal = bt.indicators.ExponentialMovingAverage(self.data_close,
period=self.params.long_window)
        self.order = None
    def next(self):
        if not self.position:
            cash = self.broker.get_cash()
            asset_price = self.data_close[0]
            position_size = cash / asset_price * 0.99
            if self.emas[0] > self.emal[0]:
                self.buy(size=position_size)
        else:
            if self.emas[0] < self.emal[0]:
                self.close()
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        self.order = None
# RSI-based Mean Reversion Strategy: buy when oversold, sell when overbought.
class RSI_Strategy(bt.Strategy):
    params = (
        ('rsi_period', 14),
        ('overbought', 70),
        ('oversold', 30),
    )
    def __init__(self):
        self.data_close = self.datas[0].close
        self.rsi = bt.indicators.RelativeStrengthIndex(self.data_close,
period=self.params.rsi_period)
        self.order = None
    def next(self):
        if not self.position:
            if self.rsi[0] < self.params.oversold:
                cash = self.broker.get_cash()
                asset_price = self.data_close[0]
                position_size = cash / asset_price * 0.99
                self.buy(size=position_size)
        else:
            if self.rsi[0] > self.params.overbought:
                self.close()
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        self.order = None

if __name__ == '__main__':

    market_returns = []     # List to store market returns for each test
    strategy_returns = []   # List to store strategy returns for each test
    cryptos = ['BTC-USD', 'ETH-USD', 'XRP-USD', 'LTC-USD', 'BCH-USD']
    overall_start = datetime.strptime("2020-01-01", "%Y-%m-%d")
    overall_end   = datetime.strptime("2025-12-31", "%Y-%m-%d")

    iterations = 100  # Number of backtest runs
    with tqdm(total=iterations) as pbar:
        i = 0
        while i < iterations:
            try:
                random_crypto = random.choice(cryptos)
                two_months_days = 60  # approx. 60 days for a 2-month window
                max_start_date = overall_end -
timedelta(days=two_months_days)
                random_start = overall_start + (max_start_date -
overall_start) * random.random()
                random_end = random_start + timedelta(days=two_months_days)
                start_str = random_start.strftime("%Y-%m-%d")
                end_str = random_end.strftime("%Y-%m-%d")

                with suppress_stdout_stderr():
                    data = yf.download(random_crypto, start=start_str,
end=end_str, progress=False)

                if hasattr(data.columns, 'droplevel'):
                    data.columns = data.columns.droplevel(1).str.lower()
                else:
                    data.columns = data.columns.str.lower()

                if data.empty:
                    i += 1
                    pbar.update(1)
                    continue

                cerebro = bt.Cerebro()
                data_feed = bt.feeds.PandasData(dataname=data)
                cerebro.adddata(data_feed)

                # Uncomment one of the strategies to test:
                cerebro.addstrategy(RSI_Strategy)
                # cerebro.addstrategy(EMA_Crossover_Strategy)

                cerebro.broker.setcash(100.)
                cerebro.broker.setcommission(commission=0.001)
                cerebro.run()

                market_return = 100 * (data.close[-1] / data.close[0] - 1)
                strategy_return = cerebro.broker.getvalue() - 100
                market_returns.append(market_return)
                strategy_returns.append(strategy_return)

                i += 1
                pbar.update(1)
            except Exception as e:
                i += 1
                pbar.update(1)
                continue
    # Visualize the distribution of returns using density histograms.
    import matplotlib.pyplot as plt
    import numpy as np
    plt.figure(figsize=(10, 6))
    bins = 25  # Number of histogram bins
    plt.hist(market_returns, bins=bins, density=True, alpha=0.5,
             label='Market Returns', color='blue', edgecolor='black')
    plt.hist(strategy_returns, bins=bins, density=True, alpha=0.5,
             label='Strategy Returns', color='green', edgecolor='black')
    mean_market = np.mean(market_returns)
    mean_strategy = np.mean(strategy_returns)
    plt.axvline(mean_market, color='blue', linestyle='dashed', linewidth=2,
                label=f'Market Mean: {mean_market:.2f}')
    plt.axvline(mean_strategy, color='green', linestyle='dashed',
linewidth=2,
                label=f'Strategy Mean: {mean_strategy:.2f}')
    plt.title("Statistical Distribution of Market & Strategy Returns")
    plt.xlabel("Return Value")
    plt.ylabel("Density")
    plt.legend()
    plt.show()
```





As you can see in the results, mean returns and also the distribution of returns show no obvious advantage over market returns for the simple classic strategies we tested for demonstration of the idea. You can test your strategies like this and analyze the results to see if they prove to be statistically significant returns over the market.

Key Takeaways

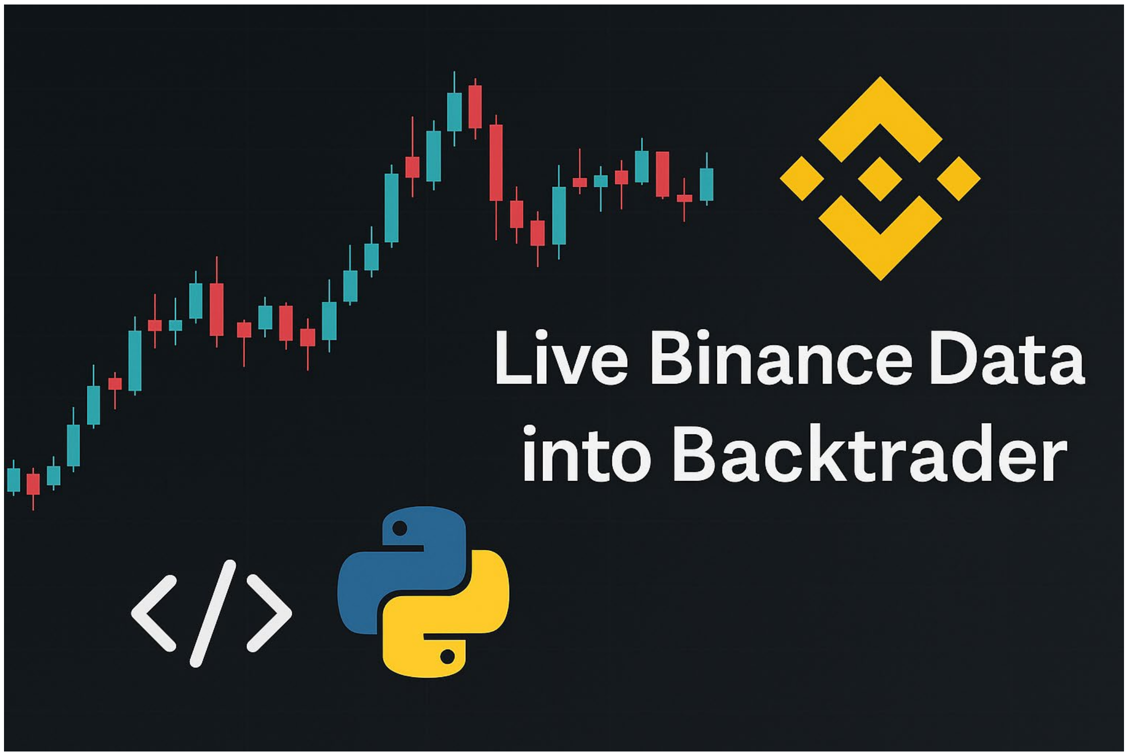
1. **Randomized Data Selections:**
   Varying both the asset choice and the timeframe for each backtest run helps simulate different market conditions and ensures your strategy's performance isn't just a product of a favorable dataset.
2. **Multiple Iterations & Aggregation:**
   Running many iterations and aggregating results gives you a statistical distribution of returns. This reveals not only average performance but also the variance and potential risk.
3. **Visual Analysis:**
   Overlapping density histograms — with indicators for mean returns — allow for a clear comparison between raw market performance and your strategy's performance. This visual approach aids in decision-making and risk assessment.
4. **Practical Setup with Backtrader:**
   Integrating tools like `tqdm` for progress tracking and custom context managers to suppress noisy output creates a clean, automated testing environment.

Conclusion

By embracing the randomness inherent in markets — randomized asset selection and time windows — you can perform a more comprehensive and robust evaluation of your strategy. This method gives you statistical insight into performance variability, paving the way for more confident decision-making and risk management. With Backtrader and the above framework, you're well-equipped to "crack the code" on how your strategy might perform across a myriad of market conditions.

Integrating Live Binance Data Feed into Backtrader for Real-Time Trading Strategies

Backtrader is a powerful Python framework for backtesting and live trading strategies. While its strength lies in historical analysis, integrating live data feeds opens up exciting possibilities for deploying trading algorithms in real-time. This article will guide you through the process of connecting Backtrader to Binance, a leading cryptocurrency exchange, to receive live market data. Let's see how we can do it.



Pasted image 20250507215909.png

Pasted image 20250507215909.png

The `BinanceLiveFeed` Class: Your Live Data Pipeline

This custom class, inheriting from Backtrader's `bt.feed.DataBase`, is responsible for continuously pulling live candlestick data from the Binance API and making it available to your trading strategies.

Python

```
import backtrader as bt
from binance.client import Client
import pandas as pd
import time
import pytz


# ─── Live Feed from Binance ───
class BinanceLiveFeed(bt.feed.DataBase):
    params = (
        ('symbol',         'BTCUSDT'),
        ('interval',       '1m'),        # '1m','5m','15m','1h', etc.
        ('lookback',       500),         # how many bars to fetch initially
        ('sleep',          2),           # seconds to wait if no new data
        ('api_key',        None),
        ('api_secret',     None),
    )
```

- **Imports**: We bring in the necessary tools: `backtrader` for the framework, `binance.client` for interacting with Binance, `pandas` for data handling, `time` for pausing, and `pytz` for timezone management.
- **`BinanceLiveFeed(bt.feed.DataBase)`**: We define our data feed class, telling Backtrader it's a source of market data.
- **`params = (...)`**: These are the settings you can adjust when you create an instance of this data feed:
    - `symbol`: The trading pair (e.g., 'BTCUSDT').
    - `interval`: The candlestick timeframe ('1m', '5m', etc.).
    - `lookback`: How many historical bars to grab at the start.
    - `sleep`: How long to wait (in seconds) if no new bars are available from Binance, helping to avoid overloading their API.
    - `api_key`, `api_secret`: Your Binance API credentials (needed for private data or trading).

Python

```
    def __init__(self):
        super().__init__()  # Initialize the base class
        self.client     = Client(self.p.api_key, self.p.api_secret)
        self._df        = None  # Will hold the fetched data as a DataFrame
        self._next_i    = 0     # Index of the next bar to send to Backtrader
        self._last_ts   = None  # Timestamp of the last bar sent (in
nanoseconds)
```

- **`__init__(self)`**: This sets up the data feed when you create it.
    - `super().__init__()`: Calls the initialization of the parent `bt.feed.DataBase` class.
    - `self.client = Client(...)`: Creates a connection to the Binance API using your provided keys.
    - `self._df = None`, `self._next_i = 0`, `self._last_ts = None`: Initializes internal variables to manage the data flow.

Python

```
    def islive(self):
        # Tells Backtrader this feed is continuous and live
        return True
```

- **`islive(self)`**: This method signals to Backtrader that this isn't a one-time historical dataset but a live stream of data.

Python

```
    def _load(self):
        while True:  # Keep trying to get new data
            if self._df is None or self._next_i >= len(self._df):
                klines = self.client.get_klines(
                    symbol=self.p.symbol,
                    interval=self.p.interval,
                    limit=self.p.lookback
                )
                df = pd.DataFrame(klines, columns=[
                    'open_time','Open','High','Low','Close','Volume',

'close_time','qav','num_trades','taker_base','taker_quote','ignore'
                ])
                df['open_time'] = pd.to_datetime(df['open_time'], unit='ms')
                df.set_index('open_time', inplace=True)
                df = df[['Open','High','Low','Close','Volume']].astype(float)

                if self._last_ts is not None:
                    df = df[df.index.view('int64') > self._last_ts]

                if df.empty:
                    time.sleep(self.p.sleep)
                    continue

                self._df     = df
                self._next_i = 0
                self._last_ts = int(df.index.view('int64')[-1])

            row = self._df.iloc[self._next_i]
            dt  = row.name.to_pydatetime().replace(tzinfo=pytz.utc)

            self.lines.datetime[0] = bt.date2num(dt)
            self.lines.open[0]   = row['Open']
            self.lines.high[0]   = row['High']
            self.lines.low[0]    = row['Low']
            self.lines.close[0]  = row['Close']
            self.lines.volume[0] = row['Volume']

            self._next_i += 1
            return True  # A new bar has been sent
```

- **`_load(self)`**: This is the engine that fetches and processes data:
    1. **Check for New Data Need**: It checks if we need to fetch a new batch of data from Binance (either the first time or if we've used all the data in our current buffer).
    2. **Fetch Klines**: It uses the Binance API client to get candlestick data (`klines`) for the specified symbol and interval, limited by the `lookback` parameter.

3. **Process Data**: The raw `klines` are converted into a Pandas DataFrame with meaningful column names. The 'open_time' is converted to datetime objects and set as the index. Only the OHLCV columns are kept and converted to the correct data type (float).
4. **Filter New Bars**: If we've already received data, it filters the new DataFrame to include only bars with timestamps later than the last one we sent, preventing duplicates.
5. **Handle No New Data**: If the DataFrame is empty after fetching and filtering, it means no new bars are available yet. The script waits for the specified `sleep` duration and tries again.
6. **Update Buffer**: If new data is fetched, the internal DataFrame (`_df`) is updated, the index for the next bar (`_next_i`) is reset, and the timestamp of the latest bar (`_last_ts`) is recorded.
7. **Emit a Bar**: It takes the next row from the DataFrame, converts its timestamp to a Backtrader-compatible format (with UTC timezone), and assigns the OHLCV values to the corresponding data lines in Backtrader (`self.lines`).
8. **Increment Index**: It moves to the next bar in the buffer.
9. **Return True**: Signals to Backtrader that a new data point is ready.

The `PrintBars` Strategy: Seeing the Live Data

This simple strategy demonstrates how to access the live data coming into Backtrader.

Python

```
# ─── Simple Printer Strategy ───
class PrintBars(bt.Strategy):
    def next(self):
        dt = self.datas[0].datetime.datetime(0)
        o  = self.data.open[0]
        h  = self.data.high[0]
        l  = self.data.low[0]
        c  = self.data.close[0]
        v  = self.data.volume[0]
        print(f"{dt.isoformat()} | O:{o:.2f} H:{h:.2f} L:{l:.2f} C:{c:.2f}
V:{v:.6f}")
```

- **`class PrintBars(bt.Strategy)`**: Defines a trading strategy within Backtrader.
- **`def next(self)`**: This method is called by Backtrader for each new bar of data it receives.
    - `dt = self.datas[0].datetime.datetime(0)`: Gets the timestamp of the current bar from the first data feed (`self.datas[0]`).
    - `o = self.data.open[0]`, etc.: Accesses the open, high, low, close, and volume values of the current bar. `self.data` is a shortcut for `self.datas[0]`. The `[0]` refers to the current bar.

  - `print(...)`: Prints the timestamp and the OHLCV values to the console, allowing you to see the live data flowing in.

Running the Live Feed: Putting It All Together

This is the section that sets up Backtrader and starts the live data stream.

Python

```
if __name__ == '__main__':
    # — Your Binance API keys (leave blank for public data) —
    BINANCE_API_KEY     = ''  # e.g. 'AbCdEfGh1234...'
    BINANCE_API_SECRET = ''  # e.g. 'XyZ9876...'

    cerebro = bt.Cerebro()
    cerebro.adddata(BinanceLiveFeed(
        symbol='BTCUSDT',
        interval='1m',
        lookback=200,
        sleep=1,
        api_key=BINANCE_API_KEY,
        api_secret=BINANCE_API_SECRET,
    ))
    cerebro.addstrategy(PrintBars)

    print("Starting live Binance feed…  press Ctrl+C to stop")
    try:
        cerebro.run()  # This will block and continuously process live data
    except KeyboardInterrupt:
        print("\nInterrupted by user. Exiting.")
```

- **API Key Input**: You'll need to replace the empty strings with your actual Binance API key and secret if required. Be careful with these!
- **`cerebro = bt.Cerebro()`**: Creates Backtrader's central engine.
- **`cerebro.adddata(BinanceLiveFeed(...))`**: Creates an instance of our `BinanceLiveFeed` with your chosen settings (symbol, interval, etc.) and adds it to Cerebro. This tells Backtrader where to get its data.
- **`cerebro.addstrategy(PrintBars)`**: Adds our simple `PrintBars` strategy to Cerebro. This tells Backtrader which logic to apply to the incoming data.
- **`cerebro.run()`**: Starts the Backtrader engine. When using a live feed, this will run continuously, processing new data as it arrives from Binance.
- **`try...except KeyboardInterrupt`**: This allows you to stop the live feed by pressing Ctrl+C in your terminal.

Important Considerations for Live Trading

- **API Rate Limits**: Be mindful of Binance's API rate limits. Excessive requests can lead to temporary blocking of your access. The `sleep` parameter in the `BinanceLiveFeed` helps mitigate this, but you might need to adjust it based on the interval and the complexity of your strategy.
- **Error Handling**: The provided code includes basic handling for no new data, but for a robust live trading system, you'll need to implement more comprehensive error handling for network issues, API errors, and unexpected data formats.
- **Latency**: Real-time data feeds inherently have some latency. Understand that the data your strategy receives might not be exactly the instantaneous market price.
- **Execution**: This code focuses on data ingestion. To execute trades based on your strategy's signals, you'll need to integrate with Binance's order placement API, which involves additional considerations for security, order types, and risk management.
- **Time Synchronization**: Ensure your system's clock is properly synchronized to avoid issues with timestamp comparisons and order execution.
- **Testing**: Thoroughly test your live trading strategy in a simulated environment or with a very small amount of capital before deploying it with significant funds.

Conclusion

Integrating live data from exchanges like Binance into Backtrader opens up a world of possibilities for algorithmic trading. The `BinanceLiveFeed` class provides a solid foundation for building real-time trading strategies. By understanding its components and considering the important factors for live trading, you can leverage Backtrader's powerful features to automate your trading decisions based on real-time market movements. Remember to always prioritize security, error handling, and thorough testing when working with live financial data and trading systems.

Intraday Trading Strategies

We will analyze some intraday trading strategies that many traders have explored. Keep in mind that no strategy guarantees profits, and it's crucial to rigorously backtest any approach before live trading. Also, always manage your risk appropriately. This information is for educational purposes only and is not financial advice.

---

1. Scalping

Overview:
Scalping involves making many small trades to capture minimal price movements throughout the day. Scalpers typically hold positions for only a few minutes.

Key Points:

- **High Frequency:** Execute many trades per day.
- **Small Targets:** Aim for small profit margins per trade.
- **Tight Risk Controls:** Use strict stop-loss orders to minimize losses.
- **Example Setup:**
    - **Entry:** Use a short-term indicator (like a 5-period moving average crossover) to signal quick entry.
    - **Exit:** Set profit targets of 0.1–0.2% per trade and stop-loss levels that match the risk/reward ratio.

Pros:

- Can capitalize on many small moves.
- Less exposure to market direction if executed well.

Cons:

- Requires fast execution and low latency.
- High trading costs can erode profit margins.

---

2. Breakout Trading

Overview:
Breakout strategies focus on price moving out of a defined range (support/resistance). The idea is that when the price breaks out, it tends to continue in that direction.

Key Points:

- **Identify Key Levels:** Plot support and resistance or use pivot points.
- **Volume Confirmation:** Look for increased volume to confirm the breakout.
- **Example Setup:**
    - **Entry:** Buy when the price breaks above a significant resistance level on strong volume; short when it breaks below support.
    - **Stop-Loss:** Place just below the breakout level (for longs) or above it (for shorts).
    - **Exit:** Use a trailing stop or predefined target based on risk-reward ratios.

Pros:

- Captures strong directional moves.
- Can be automated using technical indicators.

Cons:

- False breakouts (whipsaws) can lead to losses.
- Requires precise entry timing and confirmation.

---

3. Mean Reversion

Overview:
This strategy assumes that price tends to revert to its mean or average over time. When price deviates significantly, it's likely to return to the average.

Key Points:

- **Indicators:** Use Bollinger Bands, RSI, or moving averages to identify overbought/oversold conditions.
- **Example Setup:**
    - **Entry:** Buy when the price hits the lower Bollinger Band (indicating oversold conditions) and the RSI is below 30.
    - **Exit:** Sell when the price reverts to the 20-period moving average or upper Bollinger Band.
    - **Stop-Loss:** Set just below the recent swing low to manage risk.

Pros:

- Works well in range-bound markets.
- Clear entry and exit signals.

Cons:

- Can underperform in strongly trending markets.
- Requires quick decision-making as reversions can be short-lived.

---

4. Momentum Trading

Overview:
Momentum strategies focus on securities that are moving strongly in one direction. The idea is to ride the trend until signs of a reversal appear.

Key Points:

- **Indicators:** Use indicators like the Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), or rate-of-change.
- **Example Setup:**
    - **Entry:** Enter a long position when the MACD line crosses above the signal line and the RSI is rising but not overbought (e.g., between 40-70).

   - **Exit:** Exit when the momentum slows down, indicated by a divergence or when RSI reaches extreme levels (over 70 for longs).
   - **Stop-Loss:** Tight stop-loss orders just below recent support levels.

Pros:

- Captures extended moves.
- Can be very profitable during strong trends.

Cons:

- Timing the exit can be challenging.
- Trends can quickly reverse, leading to rapid losses if not managed well.

---

5. Algorithmic / Automated Strategies

Overview:
Many traders develop automated trading systems that combine elements of the above strategies. Algorithms can monitor markets continuously, executing trades based on predefined criteria.

Key Points:

- **Backtesting:** Rigorously test your strategy against historical data.
- **Execution Speed:** Automation can capitalize on opportunities faster than manual trading.
- **Example Setup:**
   - **Signal Generation:** Use technical indicators (like EMA crossovers, RSI thresholds) to generate trade signals.
   - **Risk Management:** Integrate dynamic stop-losses and take-profit rules.
   - **Monitoring:** Continuously monitor for slippage, latency, and changes in market conditions.

Pros:

- Removes emotional bias.
- Can operate 24/7 (especially useful in crypto markets).

Cons:

- Requires technical expertise to build and maintain.
- System failures or unexpected market conditions can result in losses.

Risk Management and Backtesting

No matter which strategy you choose:

- **Risk per Trade:** Limit your risk to a small percentage (typically 1-2% of your trading capital).
- **Backtesting:** Always test your strategies on historical data to understand potential pitfalls.
- **Diversification:** Avoid putting all your capital into a single strategy or asset.
- **Continuous Review:** Monitor and adjust strategies as market conditions evolve.

Final Thoughts

While there are multiple intraday strategies that have worked for some traders, the key is to combine a robust strategy with strict risk management and continuous evaluation. The most “profitable” strategy is one that fits your personality, risk tolerance, and market understanding.

Remember: Trading involves significant risk, and you should only trade with capital you can afford to lose.

MA Crossover with MACD Confirmation Building Custom Strategies with Backtester

This article demonstrates how to create a custom trading strategy using the backtrader library in Python for use with “Backtester” app. The strategy combines Moving Average (MA) crossover with MACD confirmation to generate buy and sell signals. We’ll break down the code, explain its components, and show how it can be used within the backtrader framework.

Strategy Overview: MACrossoverMACDConfirmStrategy

The strategy, named `MACrossoverMACDConfirmStrategy`, leverages two popular technical indicators:

- **Moving Average Crossover:** Uses a short-term MA and a long-term MA. A bullish crossover (short MA crossing above long MA) indicates a potential buy signal, while a bearish crossover (short MA crossing below long MA) suggests a sell signal.
- **MACD (Moving Average Convergence Divergence):** A momentum indicator that shows the relationship between two moving averages of prices.1 It helps confirm the trend indicated by the MA crossover.

The strategy's logic is as follows:

- **Long Entry:** A buy signal is generated when a bullish MA crossover occurs **AND** the MACD line is above the signal line, confirming bullish momentum.
- **Short Entry:** A sell signal is triggered when a bearish MA crossover occurs **AND** the MACD line is below the signal line, confirming bearish momentum.
- **Exit:** The strategy exits a position (long or short) when a reverse MA crossover occurs.

Code Breakdown

Let's dissect the Python code, explaining each section:

1. Import Statements and Parameters:

Python

```
import backtrader as bt
from datetime import datetime

class MACrossoverMACDConfirmStrategy(bt.Strategy):
    params = (
        # MA Params
        ('short_ma_period', 50), # Period for the short-term MA
        ('long_ma_period', 200), # Period for the long-term MA
        ('ma_type', 'EMA'),      # Type of Moving Average: 'SMA' or 'EMA'

        # MACD Params
        ('macd_fast', 12),       # Period for the fast EMA in MACD
        ('macd_slow', 26),       # Period for the slow EMA in MACD
        ('macd_signal', 9),      # Period for the signal line EMA in MACD

        ('printlog', True),      # Enable/Disable logging
    )
```

- The code imports the `backtrader` library as `bt` and the `datetime` module.
- The `MACrossoverMACDConfirmStrategy` class inherits from `bt.Strategy`, which is the base class for all backtrader strategies.
- The `params` tuple defines the strategy's configurable parameters. These include the periods for the short-term and long-term moving averages (`short_ma_period`, `long_ma_period`), the type of moving average to use (`ma_type` - either 'SMA' for Simple Moving Average or 'EMA' for Exponential Moving Average), and the parameters for the MACD calculation (`macd_fast`, `macd_slow`, `macd_signal`). The `printlog` parameter enables or disables logging of trades and strategy actions. The image of the backtester, for example, shows the parameters `short_ma_period` at 50 and `long_ma_period` at 90.

2. __init__ Method:

Python

```
    def __init__(self):
        # Keep references to data feeds
        self.dataclose = self.datas[0].close

        # To keep track of pending orders
        self.order = None
        self.buyprice = None
        self.buycomm = None

        # --- MA Indicators ---
        ma_indicator = bt.indicators.SimpleMovingAverage
        if self.p.ma_type == 'EMA':
            ma_indicator = bt.indicators.ExponentialMovingAverage
        self.short_ma = ma_indicator(self.datas[0],
period=self.p.short_ma_period)
        self.long_ma = ma_indicator(self.datas[0],
period=self.p.long_ma_period)
        self.ma_crossover = bt.indicators.CrossOver(self.short_ma,
self.long_ma)

        # --- MACD Indicator ---
        self.macd = bt.indicators.MACD(
            self.datas[0],
            period_me1=self.p.macd_fast,
            period_me2=self.p.macd_slow,
            period_signal=self.p.macd_signal
        )
```

- The `__init__` method is the constructor for the strategy. It's called once when the strategy is initialized.
- `self.dataclose = self.datas[0].close` gets the closing price data. `self.datas` is a list of data feeds provided to the strategy.
- `self.order`, `self.buyprice`, and `self.buycomm` are initialized to `None` and are used to track orders and trade details.
- The code then creates the Moving Average indicators. It dynamically chooses between `SimpleMovingAverage` and `ExponentialMovingAverage` based on the `ma_type` parameter. The periods are set using `short_ma_period` and `long_ma_period` from the parameters. The `bt.indicators.CrossOver` indicator detects when the short MA crosses the long MA.
- Similarly, the MACD indicator is created using `bt.indicators.MACD`, with its parameters also taken from the `params` tuple.

3. `log` Method:

Python

```
    def log(self, txt, dt=None, doprint=False):
        ''' Logging function for this strategy'''
        if self.params.printlog or doprint:
            try:
                 log_dt = bt.num2date(self.datas[0].datetime[0]).date()
            except IndexError:
                 dt = dt or self.datas[0].datetime.date(0) if
len(self.datas[0]) else datetime.now().date()
                 log_dt = dt
            print(f'{log_dt.isoformat()} - {txt}')
```

- This is a utility function for logging messages during the backtesting process.
- It checks the `printlog` parameter to determine whether to print the log message.
- It formats the output with a timestamp and the provided text message.

4. `notify_order` Method:

Python

```
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm {order.executed.comm:.2f}')
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm {order.executed.comm:.2f}')
            self.bar_executed = len(self)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')
        self.order = None
```

- This method is called by backtrader whenever an order's status changes.
- It handles different order statuses, such as `Submitted`, `Accepted`, `Completed`, `Canceled`, `Margin`, and `Rejected`.
- When an order is completed (filled), it logs the details of the execution (price, cost, commission).

5. `notify_trade` Method:

Python

```
    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}')
```

- This method is called when a trade is closed (i.e., when a position is exited).
- It logs the profit and loss (gross and net of commission) of the trade.

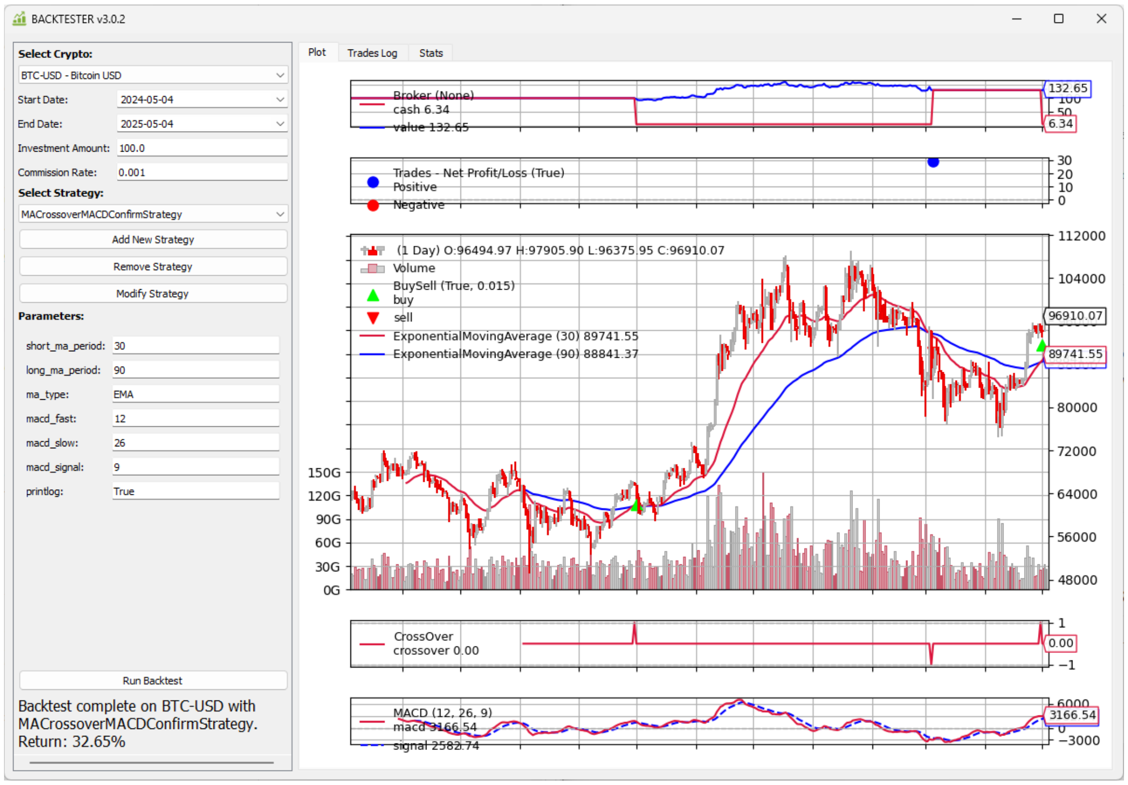
6. next Method:

Python

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

        # Check if indicators are ready
        if len(self) < self.p.long_ma_period or len(self) < (self.p.macd_slow
+ self.p.macd_signal):
              return
```

Conclusion

It is easy to create custom strategies to backtest with backtrader that you can add to the "Backtester" app. You can start with my book to learn fast or check out the strategies instruction manual for Backtester:
https://www.pyquantlab.com/apps/backtester_strategies_manual.pdf let's see the results with Backtester:


BACKTESTER v3.0.2

Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2024-05-04
End Date: 2025-05-04
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: MACrossoverMACDConfirmStrategy

Parameters:
short_ma_period: 30
long_ma_period: 90
ma_type: EMA
macd_fast: 12
macd_slow: 26
macd_signal: 9
printlog: True

Backtest complete on BTC-USD with MACrossoverMACDConfirmStrategy. Return: 32.65%

Pasted image 20250504033208.png

Pasted image 20250504033208.png

I changed parameters as you can see to find a combination that works better. I am updating the app to include many strategies to give you ideas and a base to build upon. Don't miss out!

Mastering Technical Indicators in backtrader RSI, MACD, Bollinger Bands

Technical analysis is a cornerstone of quantitative trading, and `backtrader` provides an excellent framework for implementing and backtesting strategies based on key technical indicators. Visualizing these indicators alongside price action is crucial for understanding strategy behavior and results.

In this article, we will delve into three of the most popular and widely used indicators – **Relative Strength Index (RSI)**, **Moving Average Convergence Divergence (MACD)**, and **Bollinger Bands** – and demonstrate how to implement *and plot* them using `backtrader`.

Visualizing Indicators with `backtrader` Plotting

One of `backtrader`'s most powerful features is its integrated plotting system, built on top of `matplotlib`. After running a backtest using the `Cerebro` engine, a single command (`cerebro.plot()`) generates comprehensive charts visualizing:

- Price data (OHLC bars or lines)
- Indicator values over time
- Buy/Sell signals generated by your strategy
- Portfolio value/equity curve

Crucially, standard indicators declared within your strategy are typically plotted *automatically* without needing extra plotting code within the strategy itself. `backtrader` intelligently places oscillators like RSI and MACD in separate subplots, while overlays like Moving Averages and Bollinger Bands are drawn directly on the main price chart.

1. Relative Strength Index (RSI)

The Relative Strength Index (RSI) is a momentum oscillator measuring the magnitude of recent price changes to evaluate overbought or oversold conditions (typically above 70 or below 30).

Implementing RSI in `backtrader`:

Python

```
import backtrader as bt

class RSIStrategy(bt.Strategy):
    params = (
        ('rsi_period', 14),  # Period for RSI calculation
        ('rsi_overbought', 70),
        ('rsi_oversold', 30),
    )

    def __init__(self):
        # Store the indicator reference - backtrader plots it automatically
        self.rsi = bt.indicators.RelativeStrengthIndex(
            self.data, period=self.params.rsi_period)
        # Optional: Add horizontal lines to the RSI plot
        self.rsi.plotinfo.plotyhlines = [self.params.rsi_oversold,
self.params.rsi_oversold]

    def next(self):
        if not self.position:  # Not in the market
            # Potential buy signal
            if self.rsi < self.params.rsi_oversold:
                self.log(f'RSI OVERSOLD, BUY CREATE
{self.data.close[0]:.2f}')
```

```
                self.order = self.buy()
        else:  # In the market
            # Potential sell signal
            if self.rsi > self.params.rsi_overbought:
                self.log(f'RSI OVERBOUGHT, SELL CREATE
{self.data.close[0]:.2f}')
                self.order = self.sell()

    def log(self, txt, dt=None):
        ''' Logging function for this strategy'''
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()} {txt}') # Print date and log message
```

Plotting Behavior: When plotted using `cerebro.plot()`, the RSI typically appears in its own subplot below the main price chart. `backtrader` often automatically draws horizontal lines at the 70 and 30 levels if detected, or you can explicitly add them using `plotinfo.plotyhlines` as shown above.

2. Moving Average Convergence Divergence (MACD)

The Moving Average Convergence Divergence (MACD) shows the relationship between two EMAs. It consists of the MACD line, a Signal line (EMA of MACD), and a Histogram (MACD - Signal). Crossovers are often used as signals.

Implementing MACD in `backtrader`:

Python

```
import backtrader as bt

class MACDStrategy(bt.Strategy):
    params = (
        ('fast_period', 12),
        ('slow_period', 26),
        ('signal_period', 9),
    )

    def __init__(self):
        # Store indicator references - backtrader plots them automatically
        self.macd_obj = bt.indicators.MACD(self.data,

period_me1=self.params.fast_period,

period_me2=self.params.slow_period,

period_signal=self.params.signal_period)
        # Optional: Define a crossover signal indicator for cleaner logic
        self.macd_crossover = bt.indicators.CrossOver(self.macd_obj.macd,
self.macd_obj.signal)
```

```
    def next(self):
        if not self.position:
            # Buy signal: MACD line crosses above Signal line
            if self.macd_crossover > 0: # Value is 1 for upward crossover
                self.log(f'MACD Crossover BUY, {self.data.close[0]:.2f}')
                self.order = self.buy()
        else:
            # Sell signal: MACD line crosses below Signal line
            if self.macd_crossover < 0: # Value is -1 for downward crossover
                self.log(f'MACD Crossover SELL, {self.data.close[0]:.2f}')
                self.order = self.sell()

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()} {txt}')
```

Plotting Behavior: In `backtrader` plots, the MACD line, signal line, and histogram are usually displayed together in a separate subplot beneath the price data, making it easy to visualize crossovers and histogram divergence.

3. Bollinger Bands

Bollinger Bands® consist of an SMA (middle band) and two outer bands plotted at +/- standard deviations. They help visualize volatility and potential price extremes relative to the recent trend.

Implementing Bollinger Bands in `backtrader`:

Python

```
import backtrader as bt

class BollingerBandsStrategy(bt.Strategy):
    params = (
        ('period', 20),    # Period for the SMA middle band
        ('devfactor', 2.0) # Standard Deviations for upper/lower bands
    )

    def __init__(self):
        # Store indicator reference - backtrader plots it automatically
        self.bbands = bt.indicators.BollingerBands(
            self.data, period=self.params.period,
devfactor=self.params.devfactor)
        # Optional: Store individual lines if needed for complex logic
        # self.mid_band = self.bbands.mid
        # self.top_band = self.bbands.top
        # self.bot_band = self.bbands.bot
    def next(self):
        if not self.position:
            # Buy signal: Price closes below the lower band
            if self.data.close[0] < self.bbands.lines.bot[0]:
                self.log(f'BBands Low, BUY CREATE {self.data.close[0]:.2f}')
                self.order = self.buy()
        else:
            # Sell signal: Price closes above the upper band
            if self.data.close[0] > self.bbands.lines.top[0]:
                self.log(f'BBands High, SELL CREATE
{self.data.close[0]:.2f}')
                self.order = self.sell()
            # Alternative Exit: Close crosses back below middle band
(example)
            # elif self.data.close[0] < self.bbands.lines.mid[0]:
            #     self.log(f'BBands Mid Cross, SELL CREATE
{self.data.close[0]:.2f}')
            #     self.order = self.sell()

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print(f'{dt.isoformat()} {txt}')
```

Important Considerations:

- These are simplified examples and should be further refined and optimized based on your specific trading style and risk tolerance.
- It's crucial to backtest these strategies thoroughly on historical data before risking real capital.
- Combining multiple indicators can often provide more robust trading signals.

This article provides a basic overview of implementing RSI, MACD, and Bollinger Bands within the `backtrader` framework. By mastering these fundamental indicators, you can build a solid foundation for developing and backtesting more complex algorithmic trading strategies.

Further Exploration:

For a deeper dive into technical analysis and quantitative trading with `backtrader`, I recommend exploring:

- **Backtrader's documentation:** Find detailed information on indicators, strategies, and other features.
- **Online resources and tutorials:** There are numerous online resources available, including blog posts, articles, and video tutorials, that can help you learn more about `backtrader` and technical analysis.

Remember, consistent learning and practice are key to becoming a successful quantitative trader.

Disclaimer: This information is for educational purposes only and should not be considered financial advice. Investing in financial markets carries significant risks, and you could lose all of your invested capital.

Predicting Bitcoin’s Weekly Moves with 68% Accuracy using Random Forests in Python

Predicting the direction of volatile assets like Bitcoin is a central challenge in quantitative finance. While daily noise can make short-term predictions resemble random walks, analyzing trends over slightly longer horizons, like a week, might offer more traction. This article details a Python-based approach using a Random Forest classifier and a rolling forecast methodology to predict whether Bitcoin’s price will be higher or lower seven days from the present, leveraging a pre-selected set of technical indicators. We’ll cover the theory, the implementation with code snippets, and how to interpret the results.

1. Theoretical Background

Before diving into the code, let’s understand the core concepts:

a) Random Forest Classifier

- **Ensemble Learning:** Random Forest is an ensemble machine learning method primarily used for classification and regression. It operates by constructing a multitude of individual decision trees during training.
- **How it Works:**
    1. **Bagging (Bootstrap Aggregating):** It creates multiple random subsets of the original training data (with replacement). A separate decision tree is trained on each subset.
    2. **Feature Randomness:** When splitting a node in a decision tree, the algorithm considers only a random subset of the available features, rather than all of them. This decorrelates the trees.
    3. **Voting:** For classification, the final prediction is determined by a majority vote among all the individual trees in the forest. The class predicted by the most trees wins.
- **Advantages:**
    - Handles non-linear relationships between features and the target well.
    - Generally robust to overfitting compared to individual decision trees, especially when well-tuned.
    - Can handle high-dimensional data (many features).

  - Provides useful estimates of **Feature Importance**, indicating which features contributed most to the model's decisions.
- **Equations:** While the implementation is complex, the core idea relies on aggregating simple decision trees. The prediction () for an input (x) is often represented conceptually as: (= _{b=1}^{B} { _b(x) })where B is the number of trees and (_b(x)) is the prediction of the (b^{th}) tree trained on a bootstrap sample and considering random feature subsets.

b) Feature Selection (Context)

This script assumes that a preliminary analysis has been performed to identify potentially predictive features. In our development process, Mutual Information scores were used to rank ~30 technical indicators based on their statistical relationship with the 1-day price direction. We will use the top 15 features identified in that analysis as inputs to our Random Forest model, assuming they might also hold relevance for the 7-day horizon.

c) Rolling Forecast Evaluation

- **Why Use It:** Financial markets evolve. A model trained on data from years ago might not perform well today. A simple train-test split doesn't capture this dynamic. A rolling forecast provides a more realistic simulation of how a model might perform when periodically retrained on recent data and used to predict the near future.
- **How it Works:**
  1. Define a fixed-size training window (e.g., the last 30 days).
  2. Train the model on the data within this window.
  3. Make a prediction for the target period (e.g., 7 days ahead).
  4. Slide the window forward by one time step (e.g., one day).
  5. Repeat steps 2-4 until the end of the dataset is reached.
  6. Evaluate the model based on the aggregated predictions made across all windows.

d) Classification Metrics

Since we're predicting direction (Up=1, Down=0), we use classification metrics:

- **Accuracy:** Overall percentage of correct predictions. Accuracy=TP+TN+FP+FNTP+TN
- **Precision (for class 1):** Of the times the model predicted 'Up', how often was it right? Minimizes False Positives (FP). Precision=TP+FPTP
- **Recall (Sensitivity, for class 1):** Of all the actual 'Up' movements, how many did the model catch? Minimizes False Negatives (FN). Recall=TP+FNTP
- **F1-Score (for class 1):** Harmonic mean of Precision and Recall, useful for imbalanced datasets. F1=2×Precision+RecallPrecision×Recall

- **AUC-ROC:** Area Under the Receiver Operating Characteristic Curve. Measures the model's ability to distinguish between classes across1 different probability thresholds (0.5 = random, 1.0 = perfect).
- **Confusion Matrix:** A table visualizing performance:

	Predicted Down (0)	**Predicted Up (1)**
Actual Down (0)	True Negative (TN)	False Positive(FP)
Actual Up (1)	False Negative(FN)	True Positive (TP)

2. Python Implementation Details

Let's walk through the key parts of the Python script.

a) Setup and Configuration

Import libraries and set up parameters. Critically, set `PREDICTION_HORIZON = 7` and define the `TRAINING_WINDOW_DAYS` and the list of `TOP_FEATURES` derived from previous analysis.

Python

```
#
============================================================================
=
# Imports
#
============================================================================
=
import pandas as pd
import numpy as np
import yfinance as yf
import talib # Make sure TA-Lib is installed
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix,
ConfusionMatrixDisplay,
                             roc_auc_score)
import warnings
# ... (warnings configuration) ...

#
============================================================================
=
# Configuration
#
============================================================================
=
TICKER = 'BTC-USD'
START_DATE = '2021-01-01' # Needs enough data for rolling
END_DATE = None
INTERVAL = '1d' # Daily data

# --- Rolling Window Parameters ---
TRAINING_WINDOW_DAYS = 30 # Approx 1 month training window
PREDICTION_HORIZON = 7    # Predict direction 7 days ahead

# --- Feature Selection ---
# Using Top 15 features identified previously from MI analysis
TOP_FEATURES = [
    'ROC_10', 'STOCHRSI_d', 'ADX_14', 'STOCHRSI_k', 'RSI_14',
    'STOCH_k', 'ATR_14', 'EMA_20', 'STOCH_d', 'MACD',
    'ULTOSC', 'BB_upper', 'SAR', 'Open_Close', 'MACD_hist'
]

# --- Random Forest Model Parameters ---
N_ESTIMATORS = 150
MAX_DEPTH = 8
MIN_SAMPLES_LEAF = 5
CLASS_WEIGHT = 'balanced'
RANDOM_STATE = 42
```

b) Data Loading and Indicator Calculation

Standard functions using `yfinance` and `talib` are used to fetch OHLCV data and compute the full set of ~30 technical indicators.

Python

```
# Function definitions for load_data and calculate_indicators
# (Use the full function definitions from the previous script response)

# In main execution block:
data = load_data(TICKER, START_DATE, END_DATE, INTERVAL)
if data is not None:
    data_indicators = calculate_indicators(data.copy())
```

c) Target Variable and Feature Preparation

The 7-day target variable (1 if price is higher 7 days later, 0 otherwise) is created. The data is cleaned of NaNs, and only the `TOP_FEATURES` columns are selected into the `X_all_features` DataFrame, while the `Target` column becomes `Y_all`.

Python

```
# Function definition for create_target (horizon=PREDICTION_HORIZON)
# (Use the function definition from the previous script response)
# In main execution block:
data_target = create_target(data_indicators, horizon=PREDICTION_HORIZON)
data_processed = data_target.dropna()

available_features = [f for f in TOP_FEATURES if f in data_processed.columns]
# ... (Error handling if features are missing) ...

X_all_features = data_processed[available_features]
Y_all = data_processed['Target']
Dates_all = data_processed.index # Keep dates for plotting results
```

d) The Rolling Forecast Loop

This is the core logic change from a simple train/test split.

Python

```
# --- Rolling Forecast Loop ---
all_predictions = []
all_actuals = []
all_predict_dates = []
all_probabilities = []

start_index = TRAINING_WINDOW_DAYS
end_index = len(X_all_features) - PREDICTION_HORIZON

print(f"\nStarting rolling forecast from index {start_index} to {end_index-
1}...")

for i in range(start_index, end_index):
    # 1. Define window boundaries
    train_start_idx = i - TRAINING_WINDOW_DAYS
    train_end_idx = i
    predict_feature_idx = i
    actual_target_idx = i

    # 2. Extract current window data
    X_train_window = X_all_features.iloc[train_start_idx:train_end_idx]
    Y_train_window = Y_all.iloc[train_start_idx:train_end_idx]
    X_predict_point = X_all_features.iloc[[predict_feature_idx]]
    Y_actual_point = Y_all.iloc[actual_target_idx]

    # 3. Scale features WITHIN the loop
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train_window)
    X_predict_scaled = scaler.transform(X_predict_point)

    # 4. Build and Train Model WITHIN the loop
    rf_model = RandomForestClassifier(
        n_estimators=N_ESTIMATORS,
        max_depth=MAX_DEPTH,
        min_samples_leaf=MIN_SAMPLES_LEAF,
        random_state=RANDOM_STATE,
        n_jobs=-1,
        class_weight=CLASS_WEIGHT
    )
    rf_model.fit(X_train_scaled, Y_train_window)

    # 5. Predict and Store Results
    prediction = rf_model.predict(X_predict_scaled)[0]
    probability = rf_model.predict_proba(X_predict_scaled)[0, 1] # Robust
extraction might be needed here too

    all_predictions.append(prediction)
    all_actuals.append(Y_actual_point)
    all_probabilities.append(probability)
    all_predict_dates.append(Dates_all[actual_target_idx])

    # ... (Optional progress print) ...

print("Rolling forecast complete.")
```

Crucially, the `StandardScaler` and `RandomForestClassifier` are initialized and fitted inside the loop on each window's data.

e) Aggregated Evaluation

After the loop completes, the collected predictions and actual values are used to calculate the overall performance metrics.

Python

```
# --- Evaluate Aggregated Results ---
if not all_actuals:
    print("No predictions were made.")
else:
    print("\n--- Aggregated Rolling Forecast Metrics ---")
    accuracy = accuracy_score(all_actuals, all_predictions)
    precision = precision_score(all_actuals, all_predictions,
zero_division=0)
    recall = recall_score(all_actuals, all_predictions, zero_division=0)
    f1 = f1_score(all_actuals, all_predictions, zero_division=0)
    try:
        roc_auc = roc_auc_score(all_actuals, all_probabilities)
    except ValueError:
        roc_auc = float('nan')
        # ... (print warning) ...
    print(f"Accuracy:         {accuracy:.4f}")
    print(f"Precision (for 1):{precision:.4f}")
    # ... (print other metrics) ...

    # Baseline comparison
    majority_class_overall = Y_all.value_counts().idxmax()
    baseline_accuracy = accuracy_score(all_actuals, np.full(len(all_actuals),
majority_class_overall))
    print(f"\nBaseline Accuracy (...): {baseline_accuracy:.4f}")

    # Confusion Matrix Plotting
    print("\n--- Confusion Matrix (Aggregated Rolling Forecast) ---")
    cm = confusion_matrix(all_actuals, all_predictions)
    print(cm)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
    # ... (Plotting code for CM) ...
    plt.show()

    # Optional: Plot actual vs predicted directions over time
    # ... (Plotting code for results_df) ...
    plt.show()
```

3. Results and Interpretation (Based on Your Last Run)

Your last run with this rolling Random Forest approach yielded:

- **Accuracy:** ~0.6793 (vs. Baseline ~0.5136)
- **Precision (Up):** ~0.6919
- **Recall (Up):** ~0.6772
- **F1-Score (Up):** ~0.6845
- **AUC-ROC:** ~0.7524
- **Confusion Matrix:** `[[122 57] / [ 61 128]]`

Interpretation:

These results show a clear improvement over random chance and the baseline of simply predicting the majority class. The model achieved ~68% accuracy in predicting the 7-day direction over the rolling test period. Precision and Recall are reasonably balanced (around 68-69%), indicating the model identifies 'Up' moves moderately well without excessively predicting 'Up' incorrectly. The AUC of ~0.75 suggests a decent discriminatory ability. While not perfect, these results indicate that the combination of selected features, the Random Forest model, and the rolling approach captured a statistically significant predictive signal in the historical data tested.

4. How to Use the Code

1. **Install Prerequisites:** Ensure Python, `pandas`, `numpy`, `yfinance`, `matplotlib`, `seaborn`, `scikit-learn`, and crucially, `TA-Lib` (C library + Python wrapper) are installed.
2. **Save:** Save the complete code as a Python file (e.g., `rolling_rf_btc.py`).
3. **Configure:** Modify settings like `TICKER`, `START_DATE`, `TRAINING_WINDOW_DAYS`, `PREDICTION_HORIZON`, or Random Forest parameters if desired.
4. **Run:** Execute from your terminal: `python rolling_rf_btc.py`. It will take some time as the model retrains repeatedly.
5. **Analyze:** Review the printed metrics and the confusion matrix plot. Compare accuracy to the baseline. Assess if the Precision/Recall/F1/AUC meet your requirements for considering a signal potentially useful.

5. Limitations and Conclusion

- **Historical Performance:** Success on past data doesn't guarantee future results. Markets change.
- **Not a Trading Strategy:** This analyzes predictive accuracy ONLY. It lacks entry/exit rules, risk management, cost simulation, etc.
- **Need for Tuning/Testing:** Results depend heavily on the chosen features, hyperparameters, and time period. Extensive testing and tuning are required for any real application.
- **Feature Stability:** The selected `TOP_FEATURES` might lose predictive power over time.

In conclusion, this script provides a robust framework for evaluating the predictive power of technical indicators for Bitcoin's weekly direction using a Random Forest model and a realistic rolling forecast method. The results achieved (~68% accuracy, ~0.75 AUC historically) demonstrate a potential edge worthy of further investigation, but require critical interpretation and significant further development before any practical trading application.

Regime-Filtered Risk-Adjusted Momentum Strategy with Inverse Volatility Weighting 12% to 655% Annual Returns

In this article, I present an advanced cryptocurrency trading strategy built around several robust components: market regime filtering, risk-adjusted momentum asset selection, inverse volatility weighting for portfolio construction, and a built-in stop-loss cap. The strategy aims to capture upside momentum during bullish periods while managing downside risk.

Overview:
The strategy focuses on trading a universe of cryptocurrencies by:

- **Market Regime Filter:** Only taking trades when the market (represented by BTC-USD) is in an upward trend as determined by its moving average.
- **Risk-Adjusted Momentum:** Evaluating assets based on their recent returns relative to volatility.
- **Inverse Volatility Weighting:** Allocating higher weights to assets that are less volatile.
- **Stop-Loss Cap:** Applying a post-hoc return floor to mitigate extreme drawdowns.

Strategy Components and Code Explanation

Below is the annotated code along with explanations that detail each part of the process.

1. Setting Up the Environment

The code begins by importing libraries such as `yfinance` for data retrieval, pandas for data manipulation, and `matplotlib` for visualization. It also sets key parameters for the backtest, including time periods, lookback windows, and transaction cost assumptions.

```
import requests
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt

# --- Parameters ---
START_DATE = '2022-01-01'
END_DATE = '2023-01-01'  # You can change these dates to run the backtest for
different years
LOOKBACK_DAYS = 30
HOLDING_DAYS = 7
N_TOP_ASSETS = 10
STOP_LOSS_THRESHOLD = -0.05  # The post-hoc analytical loss cap
MARKET_REGIME_MA_PERIOD = 50  # Moving Average period to filter the market
regime
MARKET_REGIME_TICKER = 'BTC-USD'
TRANSACTION_COST_PER_ROUND_TRIP = 0.002  # 0.2% cost per weekly trade round-
trip
```

Explanation:
These parameters give you control over the lookback window for calculating momentum and volatility, the number of assets to include, and the thresholds for risk management. Adjusting `START_DATE` and `END_DATE` allows you to run the strategy over different historical periods.

2. Defining the Asset Universe

We define a list of cryptocurrency pairs. In this example, placeholder tickers are used; however, you can replace these with your own data source (like BinanceSymbols).

```
print("Using placeholder ticker list. Replace with BinanceSymbols if
needed.")
crypto_pairs_binance_format = ['BTCUSDC', 'ETHUSDC', 'BNBUSDC', 'ADAUSDC',
'SOLUSDC',
                               'XRPUSDC', 'DOTUSDC', 'DOGEUSDC', 'AVAXUSDC',
'LUNA1-USD',
                               'MATICUSDC', 'LTCUSDC', 'LINKUSDC', 'TRXUSDC',
'ATOMUSDC']
# Exclude the market regime ticker from the asset selection list.
tickers_to_trade = [pair[:-4] + '-USD' for pair in
crypto_pairs_binance_format
                    if pair[:-4] + '-USD' != MARKET_REGIME_TICKER]

# Ensure the market regime ticker is part of the download.
all_tickers_to_download = tickers_to_trade + [MARKET_REGIME_TICKER]
```

Explanation:
The list `crypto_pairs_binance_format` holds sample tickers. We then convert these to a format compatible with `yfinance` (e.g., BTC-USD) and ensure that our market regime indicator remains separate.

3. Data Download and Preparation

Historical price data is fetched for the assets using `yfinance`. The code makes sure that only the closing prices are retained and that the tickers are sorted chronologically.

```
print(f"Downloading data for: {', '.join(all_tickers_to_download)}")
try:
    data = yf.download(all_tickers_to_download, start=START_DATE,
end=END_DATE, progress=False)
    if data.empty:
        raise ValueError("No data downloaded. Check tickers and date range.")
    data = data['Close']  # Use closing prices
    data = data.sort_index()
    data = data.dropna(axis=1, how='all')  # Drop columns with no data
    tickers_to_trade = [t for t in tickers_to_trade if t in data.columns]
    if MARKET_REGIME_TICKER not in data.columns:
        raise ValueError(f"Market regime ticker {MARKET_REGIME_TICKER} could
not be downloaded.")
    print(f"Successfully downloaded data for {len(data.columns)} tickers.")
except Exception as e:
    print(f"Error downloading data: {e}")
    exit()
```

Explanation:
This block downloads the data, ensures it is valid (non-empty), and extracts just the closing prices. It also checks that the market regime ticker is successfully downloaded.

4. Market Regime Filter

A 50-day moving average is computed for the market regime asset. The strategy will only trade if BTC-USD's current price is above its moving average, signaling a bullish trend.

```
data[f'{MARKET_REGIME_TICKER}_MA'] =
data[MARKET_REGIME_TICKER].rolling(window=MARKET_REGIME_MA_PERIOD).mean()
```

Explanation:
By adding a moving average column, the strategy creates a simple trend indicator to help avoid entering positions during bearish market conditions.

5. Backtesting Loop and Asset Selection

The core of the strategy is executed in a loop where for each holding period the strategy:

- Checks the market regime.
- Calculates the momentum (returns) and volatility for each asset over the lookback period.
- Computes a risk-adjusted metric (return/volatility).
- Selects the top N assets based on this metric.
- Applies inverse volatility weighting.
- Calculates the portfolio's weekly return and subtracts transaction costs.
- Applies a stop-loss cap.

```
# --- Backtesting ---
results = []
start_index = max(LOOKBACK_DAYS, MARKET_REGIME_MA_PERIOD)

print("Starting backtest...")
for i in range(start_index, len(data) - HOLDING_DAYS, HOLDING_DAYS):
    current_date = data.index[i]

    # Check market regime: Only trade when BTC-USD is above its moving
average
    try:
        is_bull_market = data[MARKET_REGIME_TICKER].iloc[i] >
data[f'{MARKET_REGIME_TICKER}_MA'].iloc[i]
    except IndexError:
        print(f"Warning: Market regime check failed for {current_date}.
Skipping week.")
        is_bull_market = False

    weekly_return = 0.0

    if is_bull_market:
        # Define the lookback period for calculating momentum and volatility
        current_prices = data.iloc[i]
        lookback_start_prices = data.iloc[i - LOOKBACK_DAYS]

        # Only consider assets with valid price data over the lookback period
        valid_tickers =
lookback_start_prices[tickers_to_trade].dropna().index
        valid_tickers = current_prices[valid_tickers].dropna().index

        if len(valid_tickers) >= N_TOP_ASSETS:
            # Calculate return over the lookback period
            lookback_returns = (current_prices[valid_tickers] /
lookback_start_prices[valid_tickers]) - 1

            # Calculate volatility over the lookback period (daily standard
deviation)
            daily_returns = data[valid_tickers].iloc[i - LOOKBACK_DAYS :
i].pct_change().dropna(how='all')
            volatilities = daily_returns.std().replace(0, 1e-9)

            # Calculate risk-adjusted returns as the ratio of return to
volatility
            risk_adjusted = lookback_returns.loc[volatilities.index] /
volatilities
            valid_risk_adjusted = risk_adjusted.dropna()

            # Select the top N assets based on the risk-adjusted metric
            if len(valid_risk_adjusted) >= N_TOP_ASSETS:
                topN_tickers =
valid_risk_adjusted.nlargest(N_TOP_ASSETS).index

                # Check that the required prices exist for the holding period
                if not current_prices[topN_tickers].isnull().any() and not
data.iloc[i + HOLDING_DAYS][topN_tickers].isnull().any():
                    # Weight allocation: Inverse volatility weighting
                    selected_vols = volatilities[topN_tickers]
                    inv_vols = 1 / selected_vols
                    weights = inv_vols / inv_vols.sum()

                    next_week_prices = data.iloc[i +
HOLDING_DAYS][topN_tickers]
```

```
                    individual_returns = (next_week_prices /
current_prices[topN_tickers]) - 1

                    # Re-align weights if some data is missing (should rarely
happen given previous checks)
                    individual_returns = individual_returns.dropna()
                    weights = weights.loc[individual_returns.index]
                    weights = weights / weights.sum()

                    gross_return = (individual_returns * weights).sum()
                    weekly_return = gross_return -
TRANSACTION_COST_PER_ROUND_TRIP
                else:
                    pass  # Missing price data for selected tickers; skip the
week.
            else:
                pass  # Not enough assets with valid risk-adjusted scores;
skip the week.

    # Apply the stop-loss cap to manage downside risk
    capped_return = max(weekly_return, STOP_LOSS_THRESHOLD)
    results.append((current_date, capped_return))

print("Backtest finished.")
```

Explanation:

- The **market regime check** ensures positions are only taken in a bull market.
- The **momentum calculation** uses a 30-day window.
- **Volatility is computed** from daily returns (you can choose to annualize if desired).
- The **risk-adjusted metric** ranks assets by return per unit of volatility.
- **Inverse volatility weighting** allocates more weight to assets with lower volatility.
- **Transaction costs** are deducted from the returns.
- A **stop-loss cap** is applied to limit weekly losses.

6. Post-Processing and Visualization

Once the backtest loop is complete, cumulative returns and performance metrics (including Sharpe ratio and maximum drawdown) are computed. The results are then plotted for visual analysis.

```
# --- Post-Processing & Plotting ---
df_results = pd.DataFrame(results, columns=['Week Start', 'Avg 1 Week
Return'])
df_results['Cumulative Return'] = (df_results['Avg 1 Week Return'] +
1).cumprod() - 1
df_results = df_results.set_index('Week Start')

plt.figure(figsize=(14, 7))
plt.plot(df_results.index, df_results['Cumulative Return'],
label=f'Strategy')

# Optional: Buy & Hold Benchmark for BTC-USD
if MARKET_REGIME_TICKER in data.columns:
     btc_buy_hold = data[MARKET_REGIME_TICKER].iloc[start_index:]
     btc_cum_ret = (btc_buy_hold / btc_buy_hold.iloc[0]) - 1
     plt.plot(btc_cum_ret.index, btc_cum_ret, label=f'{MARKET_REGIME_TICKER}
Buy & Hold', linestyle='--', alpha=0.7)

plt.xlabel('Date')
plt.ylabel('Cumulative Return')
plt.title('Inverse Volatility Weight with Regime Filter')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# --- Performance Metrics ---
print("\n--- Results ---")
print(f"Final Cumulative Return: {df_results['Cumulative Return'].iloc[-
1]:.2%}")

# Annualized Sharpe Ratio (using weekly returns, 52 weeks per year)
sharpe_ratio = (df_results['Avg 1 Week Return'].mean() / df_results['Avg 1
Week Return'].std()) * (52**0.5)
print(f"Annualized Sharpe Ratio: {sharpe_ratio:.2f}")

# Maximum Drawdown Calculation
def calculate_max_drawdown(cum_returns):
    roll_max = (cum_returns + 1).cummax()
    drawdown = (cum_returns + 1) / roll_max - 1
    return drawdown.min()

max_drawdown = calculate_max_drawdown(df_results['Cumulative Return'])
print(f"Maximum Drawdown: {max_drawdown:.2%}")
```

Explanation:

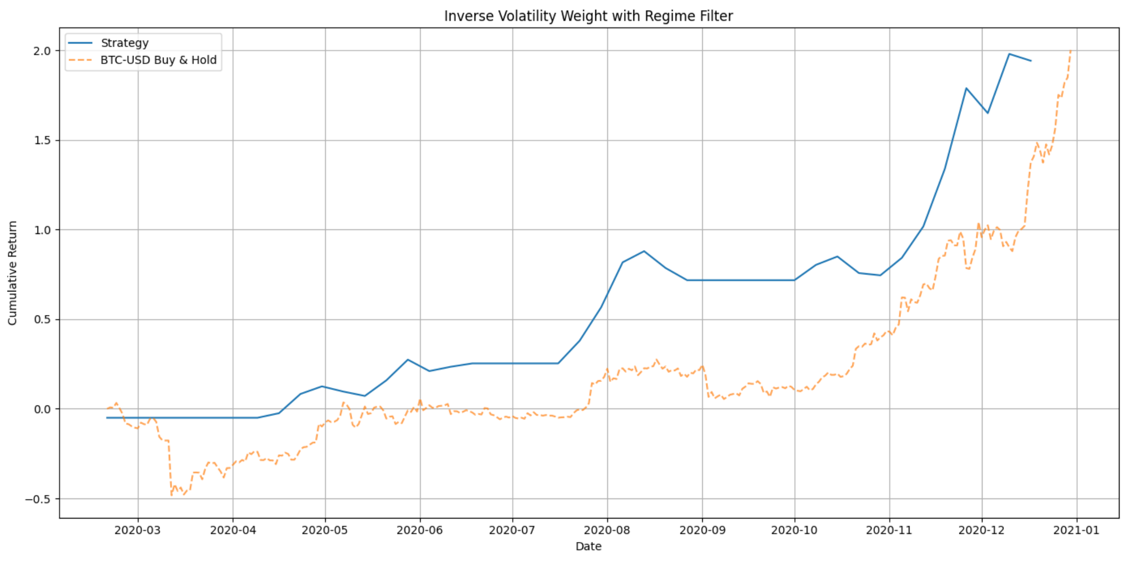
- **DataFrames creation:** Weekly returns and cumulative returns are calculated.

- **Visualization:**
  The cumulative return curves are plotted along with an optional benchmark (BTC-USD Buy & Hold) for context.

- **Performance Metrics:**
  The final cumulative returns, Sharpe ratios, and maximum drawdown figures give you quantitative insights into the strategy's performance.

Running the Backtest for Different Years

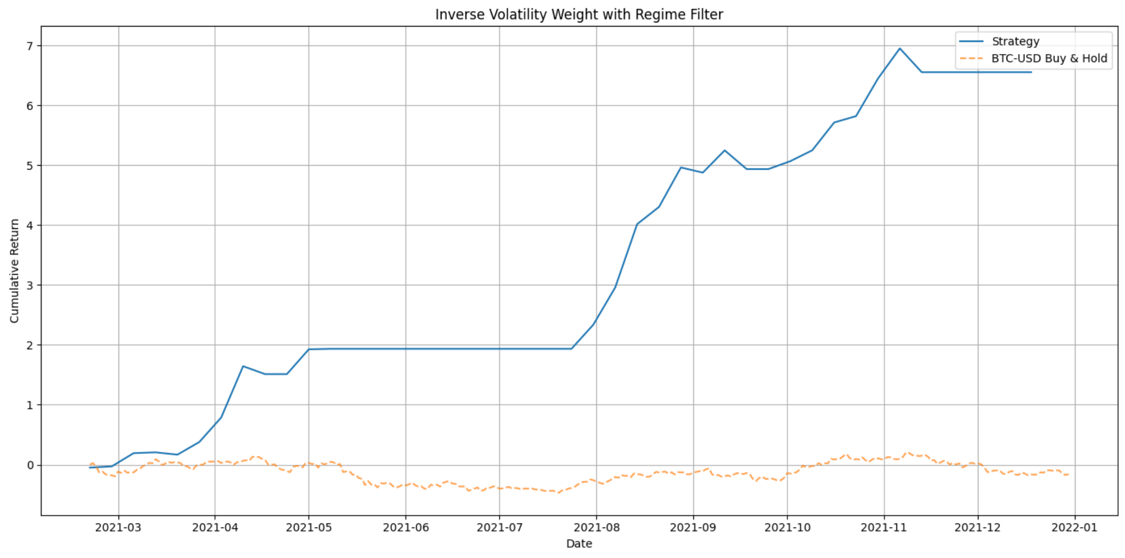
Let's see the performance of the strategy for 2020 to 2025:



Pasted image 20250414183511.png

Pasted image 20250414183511.png

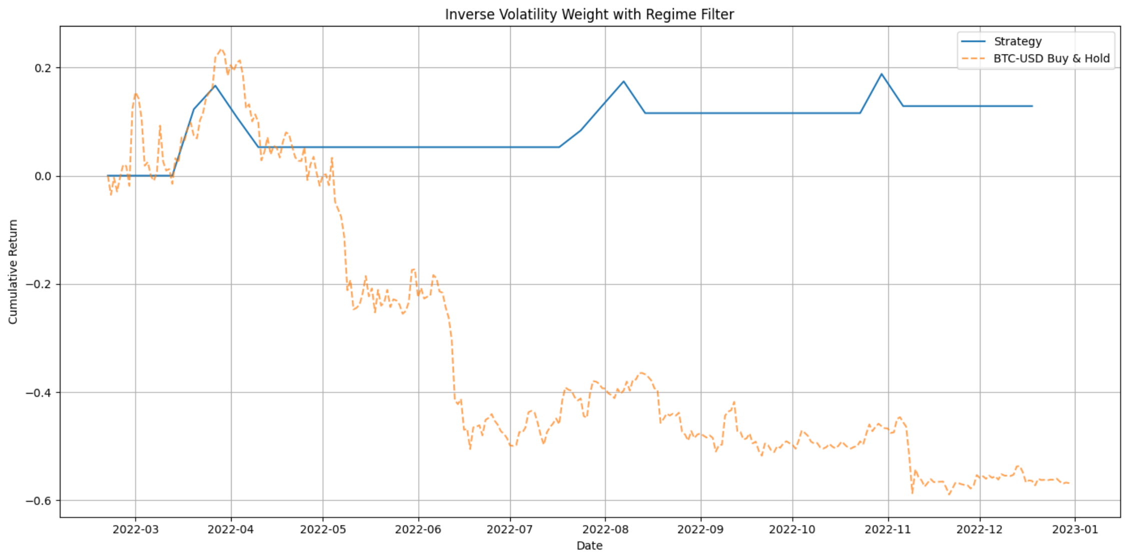
```
--- Results ---
Final Cumulative Return: 194.15%
Annualized Sharpe Ratio: 3.10
Maximum Drawdown: -8.61%
```



Pasted image 20250414183632.png

Pasted image 20250414183632.png

```
--- Results ---
Final Cumulative Return: 655.19%
Annualized Sharpe Ratio: 3.49
Maximum Drawdown: -5.00%
```

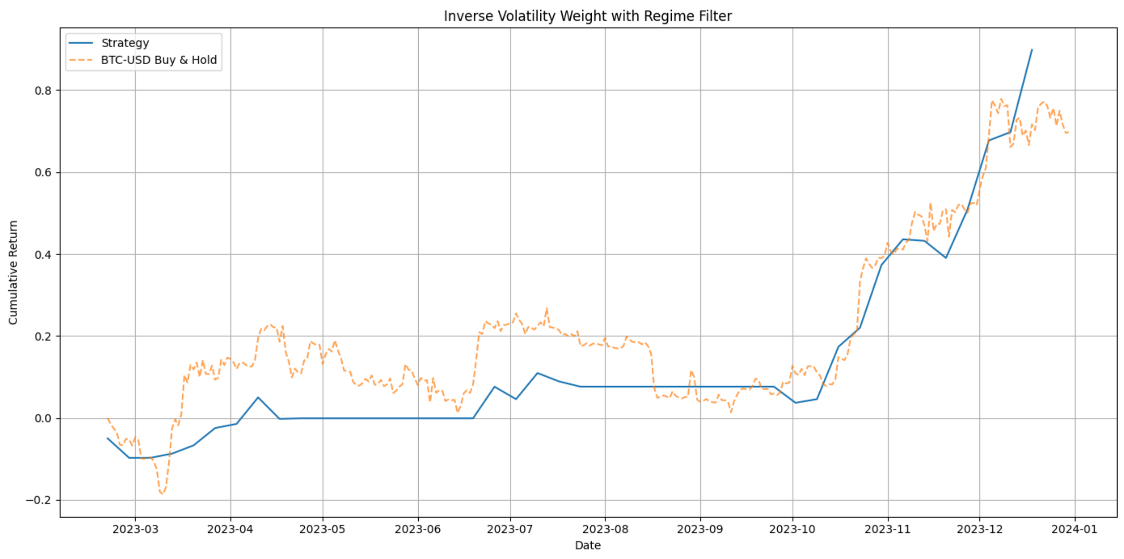


Pasted image 20250414183339.png

Pasted image 20250414183339.png

```
--- Results ---
Final Cumulative Return: 12.86%
```

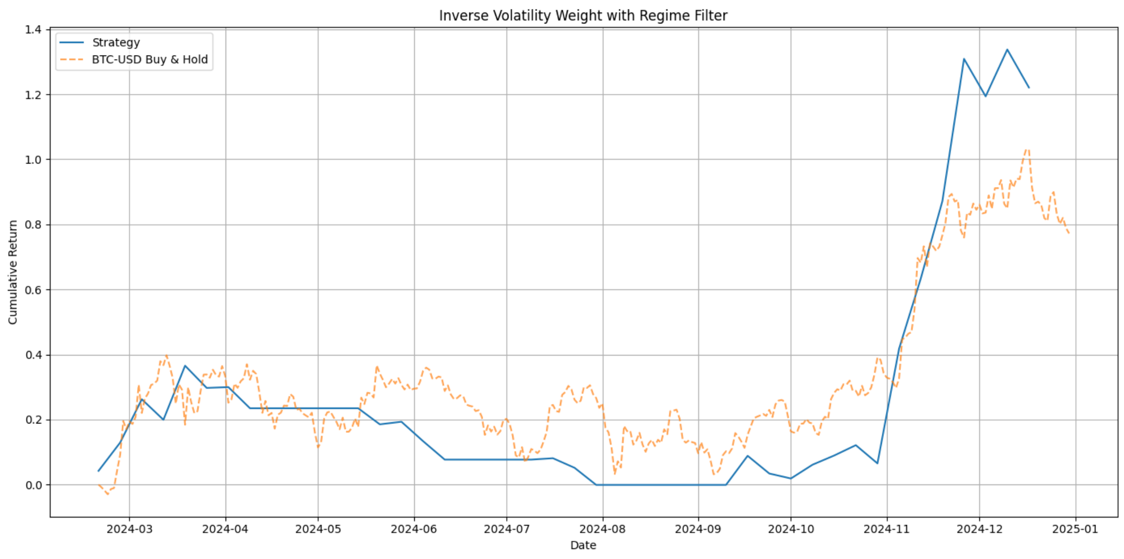
```
Annualized Sharpe Ratio: 0.80
Maximum Drawdown: -9.75%
```



Pasted image 20250414183744.png

Pasted image 20250414183744.png

```
--- Results ---
Final Cumulative Return: 89.81%
Annualized Sharpe Ratio: 2.52
Maximum Drawdown: -6.54%
```



Pasted image 20250414183842.png

Pasted image 20250414183842.png

```
--- Results ---
Final Cumulative Return: 122.05%
Annualized Sharpe Ratio: 1.90
Maximum Drawdown: -26.83%
```

Conclusion

The **Regime-Filtered Risk-Adjusted Momentum Strategy with Inverse Volatility Weighting** uses a combination of trend detection, risk-adjusted momentum ranking, and volatility-based weighting to navigate the volatile cryptocurrency market. With a market regime filter and stop-loss protection in place, the strategy aims to harness trends during bull markets while reducing exposure during less favorable conditions. We saw the impressive annual returns in the backtests from 12% up to 655%. Keep in mind that we only took long positions so we didn't make money when the market was bearish.

By understanding and adjusting each component of the code, you can tailor the backtest to various time periods and refine the strategy further with your results. This modular approach not only provides robust insights into historical performance but also sets the foundation for future enhancements and live trading implementations.

RSI Trend Confirmation Strategy in Backtester

Trading strategies often rely on technical indicators to generate buy or sell signals. However, individual indicators can sometimes produce "false" signals, especially in choppy or ranging markets. A common technique to improve signal quality is to use multiple indicators in conjunction, where one indicator confirms the signal generated by another.

The `RSITrendConfirmationStrategy` is a perfect example of this approach. It uses a primary trend-following signal, such as a Moving Average (MA) Crossover, but only acts on that signal if the momentum, measured by the Relative Strength Index (RSI), confirms the direction of the potential trade.

This article serves as a demonstration, breaking down the Python code for this strategy to show how you can create your own modular, parameter-driven strategies. The goal is to build code suitable for testing within the `backtrader` framework itself, or for easy integration into the "Backtester" application. We'll focus on creating a self-contained, parameterizable strategy class using `backtrader` conventions.

Strategy Logic Explained

The core idea is simple yet powerful:

1. **Primary Signal:** The strategy first identifies a potential trend direction using a primary indicator. In the provided code, this is a Dual Moving Average Crossover.
    - **Bullish Signal:** The shorter-term MA crosses above the longer-term MA.

   - **Bearish Signal:** The shorter-term MA crosses below the longer-term MA.
2. **Momentum Confirmation (RSI Filter):** Before entering a trade based on the primary signal, the strategy checks the RSI:
   - For a **long entry** (after a bullish MA crossover), the RSI must be above a predefined midline (typically 50), indicating bullish momentum.
   - For a **short entry** (after a bearish MA crossover), the RSI must be below the midline, indicating bearish momentum.
3. **Entry:** If both the primary signal and the RSI confirmation align, a trade is entered (buy for bullish, sell for bearish). If the RSI does not confirm the primary signal, the signal is ignored, potentially filtering out trades against the prevailing momentum.
4. **Exit:** The exit signal is based *solely* on the reversal of the primary indicator.
   - A long position is closed when the shorter-term MA crosses back below the longer-term MA.
   - A short position is closed when the shorter-term MA crosses back above the longer-term MA.
   - Crucially, the RSI filter is **not** applied to exit signals. This ensures that the position is closed based on the trend change indicated by the MAs, regardless of the RSI reading at that specific moment.

Implementation

Let’s break down the Python code which implements this strategy using the popular backtrader library.

Python

```python
import backtrader as bt

class RSITrendConfirmationStrategy(bt.Strategy):
    """
    Uses RSI > 50 (or < 50) to confirm momentum before taking signals
    from another indicator (e.g., MA crossover).
    - Enters long on primary bullish signal only if RSI > rsi_midline.
    - Enters short on primary bearish signal only if RSI < rsi_midline.
    - Primary Signal Example: Dual Moving Average Crossover.
    - Exit based on the reverse crossover of the primary signal.
    """
    params = (
        # RSI Filter Params
        ('rsi_period', 14),      # Period for RSI calculation
        ('rsi_midline', 50.0),   # RSI level to confirm trend direction

        # Primary Signal Params (Example: Dual MA Crossover)
        ('short_ma_period', 50), # Period for the short-term MA
        ('long_ma_period', 200), # Period for the long-term MA
        ('ma_type', 'SMA'),      # Type of Moving Average: 'SMA' or 'EMA'
        ('printlog', True),      # Enable/Disable logging
    )

    def __init__(self):
        # Keep a reference to the closing price
        self.dataclose = self.datas[0].close

        # To keep track of pending orders
        self.order = None
        self.buyprice = None
        self.buycomm = None

        # --- Indicators ---
        # RSI Indicator
        self.rsi = bt.indicators.RelativeStrengthIndex(
            period=self.p.rsi_period
        )

        # Primary Signal Indicators (MA Crossover)
        # Choose MA type based on parameter
        ma_indicator = bt.indicators.SimpleMovingAverage
        if self.p.ma_type == 'EMA':
            ma_indicator = bt.indicators.ExponentialMovingAverage

        # Instantiate the Moving Averages
        self.short_ma = ma_indicator(self.datas[0],
period=self.p.short_ma_period)
        self.long_ma = ma_indicator(self.datas[0],
period=self.p.long_ma_period)

        # Crossover indicator for the primary signal
        # This indicator returns:
        # +1 if short_ma crosses above long_ma
        # -1 if short_ma crosses below long_ma
        #  0 otherwise
        self.ma_crossover = bt.indicators.CrossOver(self.short_ma,
self.long_ma)

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

    def notify_order(self, order):
        # Standard order notification logic
        if order.status in [order.Submitted, order.Accepted]:
            return # Await completion

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(
                    f'BUY EXECUTED, Price: {order.executed.price:.2f}, Cost:
{order.executed.value:.2f}, Comm {order.executed.comm:.2f}'
                )
                self.buyprice = order.executed.price
                self.buycomm = order.executed.comm
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Cost: {order.executed.value:.2f}, Comm {order.executed.comm:.2f}')
            self.bar_executed = len(self) # Record bar number when order was
executed

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('Order Canceled/Margin/Rejected')

        # Reset order status
        self.order = None

    def notify_trade(self, trade):
        # Standard trade notification logic (when a position is closed)
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}')

    def next(self):
        # Core strategy logic executed on each bar
        # 1. Check if an order is pending; if so, can't send another
        if self.order:
            return

        # 2. Get the current RSI value
        current_rsi = self.rsi[0]

        # 3. Check if we are currently *not* in the market
        if not self.position:
            # Check for primary long entry signal (MA crossover bullish)
            if self.ma_crossover[0] > 0:
                # Confirm with RSI momentum
                if current_rsi > self.p.rsi_midline:
                    self.log(f'BUY CREATE, {self.dataclose[0]:.2f} (RSI
{current_rsi:.2f} > {self.p.rsi_midline:.1f}, MA Cross)')
                    # Place the buy order
                    self.order = self.buy()
                # else: # Optional log for ignored signals
                #    self.log(f'Long signal ignored, RSI {current_rsi:.2f} <=
{self.p.rsi_midline:.1f}')

            # Check for primary short entry signal (MA crossover bearish)
            elif self.ma_crossover[0] < 0:
                 # Confirm with RSI momentum
                if current_rsi < self.p.rsi_midline:
                    self.log(f'SELL CREATE, {self.dataclose[0]:.2f} (RSI
{current_rsi:.2f} < {self.p.rsi_midline:.1f}, MA Cross)')
                    # Place the sell order (short entry)
                    self.order = self.sell()
                # else: # Optional log for ignored signals
                #    self.log(f'Short signal ignored, RSI {current_rsi:.2f}
>= {self.p.rsi_midline:.1f}')

        # 4. Else (if we *are* in the market), check for exit conditions
        else:
            # Exit Long: Primary signal reverses (MA crossover bearish)
            # RSI filter is NOT used for exit
            if self.position.size > 0 and self.ma_crossover[0] < 0:
                   self.log(f'CLOSE LONG CREATE, {self.dataclose[0]:.2f} (MA
Cross)')
                   # Place the closing order
                   self.order = self.close()

            # Exit Short: Primary signal reverses (MA crossover bullish)
            # RSI filter is NOT used for exit
            elif self.position.size < 0 and self.ma_crossover[0] > 0:
                   self.log(f'CLOSE SHORT CREATE, {self.dataclose[0]:.2f} (MA
Cross)')
                   # Place the closing order
                   self.order = self.close()

    def stop(self):
        # Executed at the end of the backtest
        self.log(f'(RSI Period {self.p.rsi_period}, RSI Mid
{self.p.rsi_midline:.1f}, MA
{self.p.short_ma_period}/{self.p.long_ma_period}) Ending Value
{self.broker.getvalue():.2f}', doprint=True)
```

Code Breakdown:

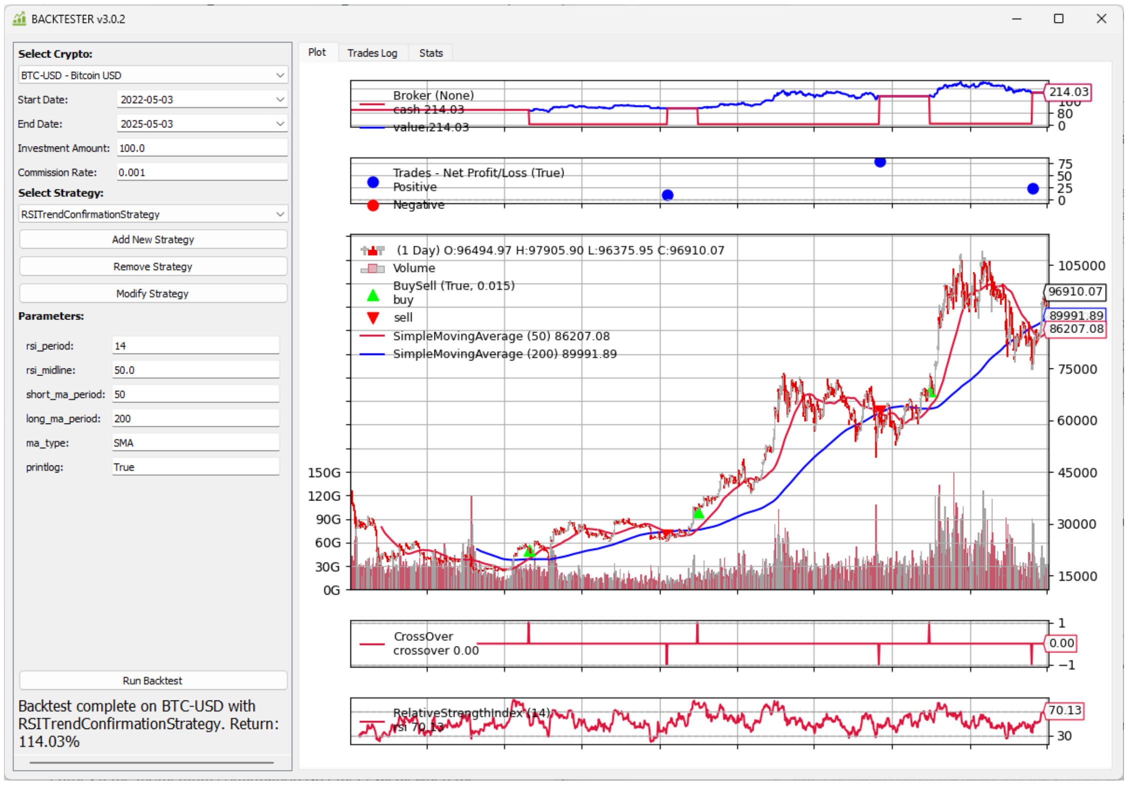
1. **params:** This tuple defines the strategy's configurable parameters:
    - rsi_period and rsi_midline control the RSI filter.
    - short_ma_period, long_ma_period, and ma_type control the primary MA Crossover signal.
    - printlog toggles detailed logging.

2. **`__init__(self)`:** The constructor sets up the necessary components:
    - It stores a reference to the closing price (`self.dataclose`).
    - Initializes variables for order tracking (`self.order`, etc.).
    - Instantiates the `RelativeStrengthIndex` indicator using the specified period.
    - Selects and instantiates the `SimpleMovingAverage` or `ExponentialMovingAverage` indicators based on the `ma_type` parameter.
    - Instantiates the `CrossOver` indicator, which directly compares the short and long MAs. This indicator simplifies checking for crossovers in the `next` method.
3. **`log`, `notify_order`, `notify_trade`:** These are standard `backtrader` methods for handling logging, order status updates, and closed trade notifications. They provide visibility into the strategy's execution during a backtest.
4. **`next(self)`:** This is the heart of the strategy, executed for each bar of data:
    - It first checks if an order is already pending (`if self.order:`).
    - It retrieves the latest RSI value (`current_rsi = self.rsi[0]`).
    - **Entry Logic (`if not self.position:`):**
        - Checks if the `ma_crossover` indicator shows a bullish cross (`> 0`). If yes, it *then* checks if `current_rsi` is above the `rsi_midline`. Only if *both* are true does it create a buy order.
        - Similarly, it checks for a bearish cross (`< 0`) and confirms with `current_rsi` below the `rsi_midline` before creating a sell order.
    - **Exit Logic (`else:`):**
        - If holding a long position (`self.position.size > 0`), it checks *only* for a bearish MA crossover (`self.ma_crossover[0] < 0`) to trigger a close order.
        - If holding a short position (`self.position.size < 0`), it checks *only* for a bullish MA crossover (`self.ma_crossover[0] > 0`) to trigger a close order.
5. **`stop(self)`:** This method is called when the backtest finishes, typically used to print final results or summaries, like the final portfolio value and the parameters used.

Why Use This Strategy?

- **Noise Reduction:** The RSI filter helps to avoid entering trades based on MA crossovers that occur during periods of weak or contradictory momentum (e.g., a bullish crossover when RSI is still below 50). This can potentially reduce whipsaws in ranging markets.

- **Momentum Confirmation:** It ensures that entries align with the underlying momentum, potentially increasing the probability of the trade moving in the desired direction initially.
- **Flexibility:** The primary signal generator can be swapped. Instead of MA Crossover, one could use MACD signals, Donchian channel breakouts, or other trend indicators, while still applying the RSI confirmation layer.


BACKTESTER v3.0.2

Select Crypto: BTC-USD - Bitcoin USD
Start Date: 2022-05-03
End Date: 2025-05-03
Investment Amount: 100.0
Commission Rate: 0.001
Select Strategy: RSITrendConfirmationStrategy

Parameters:
rsi_period: 14
rsi_midline: 50.0
short_ma_period: 50
long_ma_period: 200
ma_type: SMA
printlog: True

Backtest complete on BTC-USD with RSITrendConfirmationStrategy. Return: 114.03%

Pasted image 20250503154553.png

Pasted image 20250503154553.png

Considerations

- **Lag:** Both MAs and RSI are lagging indicators. Combining them adds complexity and potentially more lag compared to simpler strategies.
- **Missed Trades:** The RSI filter might occasionally screen out valid entries if the momentum confirmation isn’t met exactly when the primary signal occurs, even if the trend subsequently develops strongly.
- **Parameter Optimization:** The performance is sensitive to the chosen periods for the MAs and RSI, and the RSI midline value. These often require optimization for specific assets and timeframes.
- **Exit Condition:** Exiting solely on the MA crossover might lead to giving back significant profits if the trend reverses sharply before the MAs cross back.

Incorporating other exit mechanisms (e.g., trailing stops, profit targets) could be considered.

Conclusion

The `RSITrendConfirmationStrategy` offers a methodical way to enhance trend signals using momentum confirmation. As implemented here, it serves as a clear template for combining indicators within the `backtrader` framework.

Also, its structure, relying on `backtrader`'s conventions (inheriting `bt.Strategy`, using `params`, standard methods) and clear parameterization, makes it an excellent example of how to code strategies for straightforward integration and efficient testing within the "Backtester" application. Once coded this way, the strategy code can be simply added, allowing users to leverage the easy GUI for configuration, execution, and analysis.

Streamlit App for Backtesting Trading Strategies

This article provides a comprehensive guide to building a Streamlit application for backtesting trading strategies using Backtrader, yFinance, and Matplotlib.

Code Overview

Imports and Setup

The following Python libraries are used: - `streamlit` for the web interface. - pandas for data manipulation. - `backtrader` for strategy backtesting. - `yfinance` for financial data retrieval. - `matplotlib` for plotting.

```
import streamlit as st
import pandas as pd
import backtrader as bt
import yfinance as yf
import matplotlib
# Use a backend that doesn't display the plot to the user
matplotlib.use('Agg')
import matplotlib.pyplot as plt
```

Backtest Function

This function initializes Backtrader, sets up the trading environment, and executes the backtest.

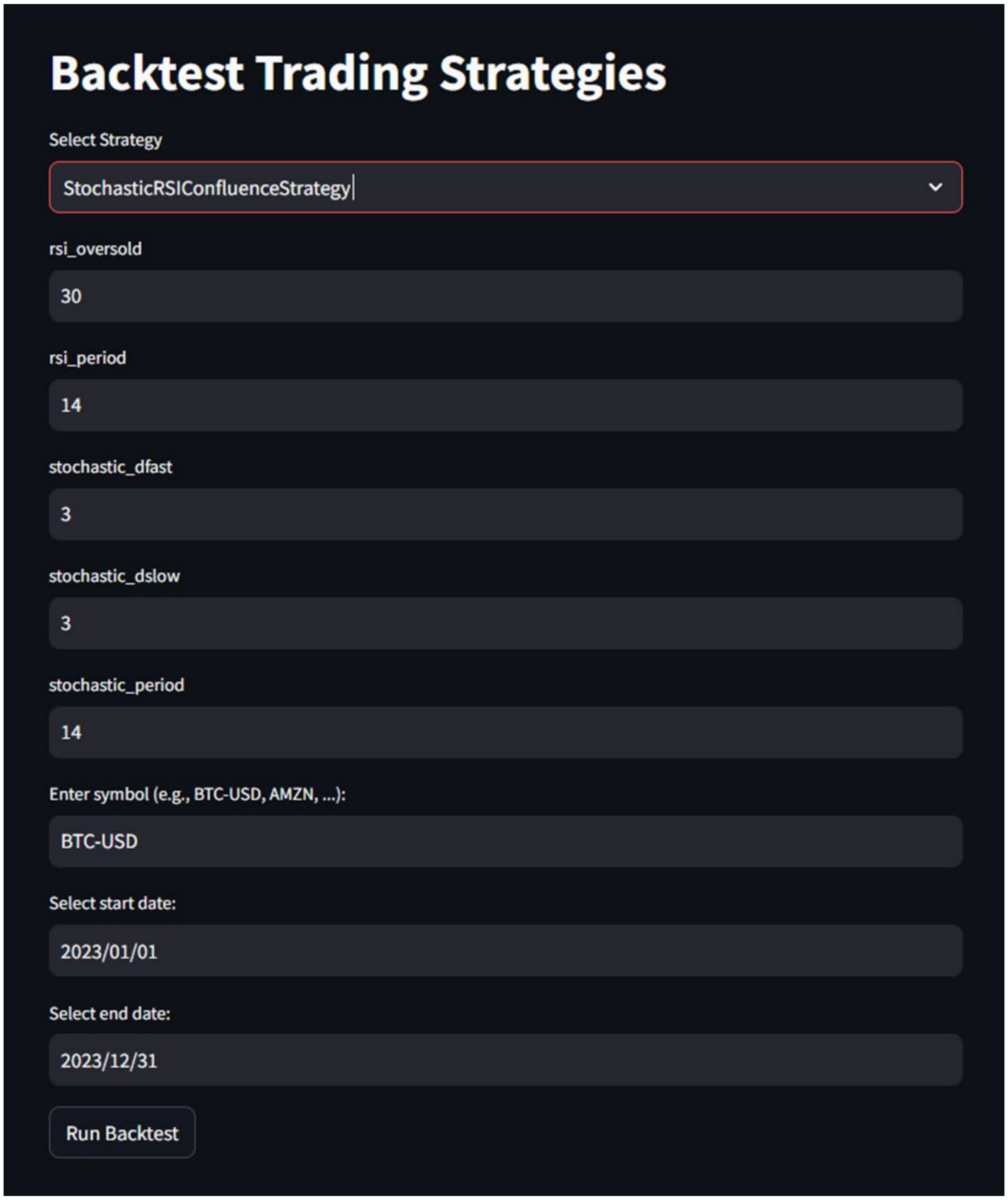
```
def run_backtest(strategy_class, symbol, start_date, end_date, **params):
    cerebro = bt.Cerebro()
    cerebro.broker.setcommission(commission=0.00)
    data = bt.feeds.PandasData(dataname=yf.download(symbol, start=start_date,
end=end_date, interval='1d'))
    cerebro.adddata(data)

    cerebro.addstrategy(strategy_class, **params)

    cerebro.addanalyzer(bt.analyzers.TimeReturn, _name='returns')

    cerebro.broker.setcash(100.)
    results = cerebro.run()
    strat = results[0]

    returns = strat.analyzers.returns.get_analysis()
    returns_df = pd.DataFrame(list(returns.items()), columns=['Date',
'Return'])
    returns_df['Date'] = pd.to_datetime(returns_df['Date'])
    returns_df.set_index('Date', inplace=True)

    plt.rcParams["figure.figsize"] = (10, 6)
    fig = cerebro.plot()
    return fig[0][0]
```

Streamlit Application

The Streamlit app allows users to interactively select trading strategies, set parameters, and visualize backtest results.

```
def main():
    st.title('Backtest Trading Strategies')

    import strategies
    strategy_names = [name for name in dir(strategies) if
name.endswith('Strategy')]
    selected_strategy = st.selectbox('Select Strategy', strategy_names)

    selected_strategy_class = getattr(strategies, selected_strategy)

    def to_number(s):
        n = float(s)
        return int(n) if n.is_integer() else n

    strategy_params = {}
    for param_name in dir(selected_strategy_class.params):
        if not param_name.startswith("_") and param_name not in ['isdefault',
'notdefault']:
            param_value = getattr(selected_strategy_class.params, param_name)
            strategy_params[param_name] = st.text_input(f'{param_name}',
value=param_value)
```

```
    strategy_params = {param_name: to_number(strategy_params[param_name]) for
param_name in strategy_params}

    symbol = st.text_input('Enter symbol (e.g., BTC-USD, AMZN, ...):', 'BTC-
USD')
    start_date = st.date_input('Select start date:', pd.to_datetime('2023-01-
01'))
    end_date = st.date_input('Select end date:', pd.to_datetime('2023-12-
31'))

    if st.button('Run Backtest'):
        st.write(f"Running backtest for {symbol} from {start_date} to
{end_date} with {selected_strategy} strategy")
        fig = run_backtest(selected_strategy_class, symbol, start_date,
end_date, **strategy_params)

        st.pyplot(fig)

if __name__ == '__main__':
    main()
```



Backtest Trading Strategies

Select Strategy

StochasticRSIConfluenceStrategy

rsi_oversold

30

rsi_period

14

stochastic_dfast

3

stochastic_dslow

3

stochastic_period

14

Enter symbol (e.g., BTC-USD, AMZN, ...):

BTC-USD

Select start date:

2023/01/01

Select end date:

2023/12/31

image

image

Summary

This application provides a user-friendly interface to backtest various trading strategies using Backtrader. Users can select strategies, input parameters, and visualize the results interactively. Integrating Streamlit with Backtrader and yFinance facilitates easy experimentation with different trading strategies and data.

Taming the RSI From a 60% Loss to 15% Profit by Adding Trend Filters

The Relative Strength Index (RSI) is one of the most popular technical indicators, beloved by traders for its apparent simplicity in identifying potentially overbought or oversold conditions. Coupled with the intuitive concept of mean reversion – the idea that prices tend to return to their average over time – it forms the basis of many automated trading strategies. Buy when the RSI dips below 30 (oversold), sell when it climbs above 70 (overbought). Simple, right?

Maybe too simple. As a recent backtesting experiment dramatically illustrates, a basic RSI mean reversion strategy, while appealing on the surface, can lead to disastrous results when faced with real-world market dynamics, particularly strong trends. However, by adding a layer of intelligence – a trend filter – the exact same core idea was transformed from a catastrophic failure into a potentially viable strategy.

The Basic Strategy: Simple Logic, Flawed Premise?

The initial strategy was straightforward:

1. Go Long: Buy when the 14-period RSI crossed below the 30 level (indicating oversold conditions).
2. Go Short: Sell short when the 14-period RSI crossed above the 70 level (indicating overbought conditions).
3. Exit Long: Sell the long position if the RSI crossed back above 50 (neutral) or 70 (overbought).
4. Exit Short: Cover the short position if the RSI crossed back below 50 (neutral) or 30 (oversold).

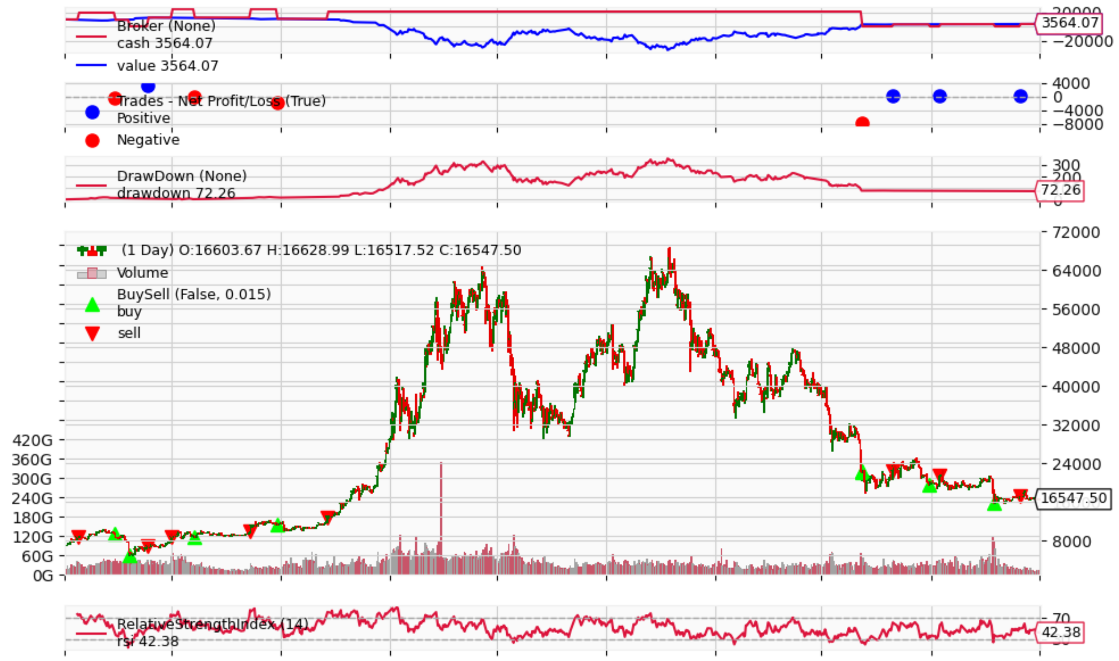
The logic seems sound for a market oscillating back and forth. Buy the dips, sell the rips.

Here’s how this basic strategy (including short selling) might be implemented using the backtrader Python library:

Python

```
import backtrader as bt

class RsiMeanReversion(bt.Strategy):
    """
    Implements an RSI Mean Reversion strategy WITH SHORT SELLING.
    Buys when RSI goes below oversold level.
    Sells (shorts) when RSI goes above overbought level.
    Exits long when RSI goes above overbought or neutral level.
    Exits short (covers) when RSI goes below neutral or oversold level.
    """
    params = (
        ('rsi_period', 14),      # Period for the RSI calculation
        ('oversold', 30),        # RSI level considered oversold (for buying)
        ('overbought', 70),      # RSI level considered overbought (for
shorting)
        ('neutral', 50),         # RSI level considered neutral for exit
        ('printlog', False),     # Enable/disable logging (set False for
cleaner article)
    )

    def __init__(self):
        self.rsi =
bt.indicators.RelativeStrengthIndex(period=self.params.rsi_period)
        self.order = None
        self.dataclose = self.datas[0].close

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()} {txt}')

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f},
Size: {order.executed.size:.4f}', doprint=True)
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Size: {order.executed.size:.4f}', doprint=True)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order Canceled/Margin/Rejected: Status
{order.getstatusname()}', doprint=True)
        self.order = None

    def notify_trade(self, trade):
         if not trade.isclosed:
             return
         self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}', doprint=True)

    def next(self):
        if self.order:
            return

        current_position_size = self.position.size

        # --- Logic when FLAT (No position) ---
        if current_position_size == 0:
            if self.rsi < self.params.oversold:
                self.log(f'BUY CREATE (Long Entry), RSI={self.rsi[0]:.2f}',
doprint=True)
                self.order = self.buy()
            elif self.rsi > self.params.overbought:
                self.log(f'SELL CREATE (Short Entry), RSI={self.rsi[0]:.2f}',
doprint=True)
                self.order = self.sell()

        # --- Logic when LONG (Position > 0) ---
        elif current_position_size > 0:
            if self.rsi > self.params.overbought or self.rsi >
self.params.neutral:
                exit_condition = "Overbought" if self.rsi >
self.params.overbought else "Neutral Reversion"
                self.log(f'SELL CREATE (Long Exit - {exit_condition}),
RSI={self.rsi[0]:.2f}', doprint=True)
                self.order = self.sell()

        # --- Logic when SHORT (Position < 0) ---
        elif current_position_size < 0:
            if self.rsi < self.params.neutral or self.rsi <
self.params.oversold:
                 exit_condition = "Oversold" if self.rsi <
self.params.oversold else "Neutral Reversion"
                 self.log(f'BUY CREATE (Short Cover - {exit_condition}),
RSI={self.rsi[0]:.2f}', doprint=True)
                 self.order = self.buy()
```

The Harsh Reality: Unfiltered Backtest Results



When this RsiMeanReversion strategy is backtested on historical data (3 years of BTC-USD daily data from 2020 to 2022), the results are devastating. Instead of generating profits, it produces:

- Total Return: -60% (A significant loss of capital)
- Maximum Drawdown: 350%

A 60% loss is bad enough, but a 350% drawdown is catastrophic. Drawdown measures the largest peak-to-trough decline in account equity. A drawdown exceeding 100% implies the use of leverage and indicates that at its worst point, the strategy lost not only all its initial capital but also a significant amount of borrowed funds, leading to margin calls and account wipeout.

Why does it fail so spectacularly? Because markets don't always revert to the mean promptly. They often trend, sometimes strongly. The basic RSI strategy blindly bought dips in powerful downtrends (catching falling knives) and shorted peaks in roaring uptrends (standing in front of a freight train). Each signal fought the prevailing momentum, leading to accumulating losses and crippling drawdowns.

The Enhancement: Adding Market Context with Trend Filters

Recognizing that fighting the trend was the strategy's Achilles' heel, the next step is to introduce filters to gauge the market's underlying direction and strength. The goal is to adapt the strategy: trade with the trend when it's strong, and revert to mean reversion only when the market is directionless or ranging.

The following filters can be used:

1. Trend Direction: A 200-period Exponential Moving Average (EMA). Price trading above the EMA suggests an uptrend; below suggests a downtrend.
2. Trend Strength: The 14-period Average Directional Index (ADX). An ADX value above 25 is used to indicate a strong trend (either up or down).

The core RSI logic was then modified based on these filters:

- If Strong Uptrend (Price > 200 EMA and ADX > 25): ONLY take LONG signals (RSI < 30). Ignore short signals (RSI > 70).
- If Strong Downtrend (Price < 200 EMA and ADX > 25): ONLY take SHORT signals (RSI > 70). Ignore long signals (RSI < 30).
- If Weak/Ranging Trend (ADX <= 25): Allow BOTH long (RSI < 30) and short (RSI > 70) signals, reverting to the original mean-reversion logic.

The exit rules remain the same mean-reversion style.

This enhanced logic is implemented in a new backtrader class:

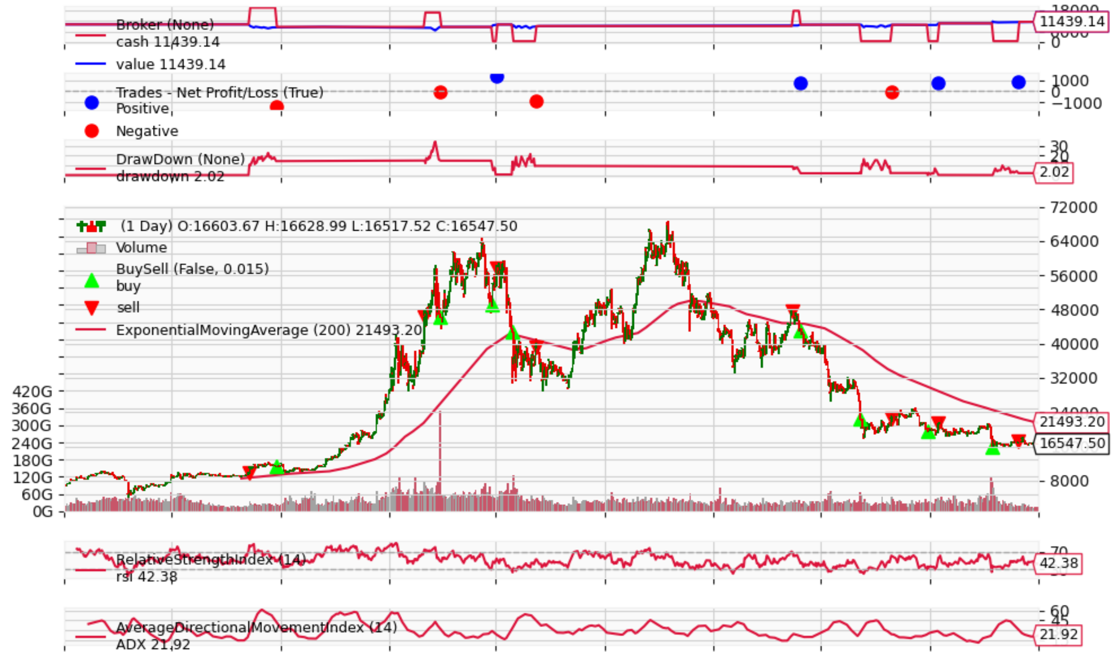
Python

```
import backtrader as bt

class RsiMeanReversionTrendFiltered(bt.Strategy):
    """
    Implements an RSI Mean Reversion strategy filtered by trend strength
(ADX)
    and direction (EMA).

    - Strong Trend (ADX > threshold):
        - Uptrend (Close > EMA): Only take LONG entries on RSI oversold.
        - Downtrend (Close < EMA): Only take SHORT entries on RSI overbought.
    - Weak/Ranging Trend (ADX <= threshold):
        - Take BOTH LONG (RSI oversold) and SHORT (RSI overbought) entries.
    - Exits remain mean-reversion based (RSI crossing neutral/opposite
threshold).
    """
    params = (
        ('rsi_period', 14),      # Period for the RSI calculation
        ('oversold', 30),        # RSI level considered oversold (for buying)
        ('overbought', 70),      # RSI level considered overbought (for
shorting)
        ('neutral', 50),         # RSI level considered neutral for exit

        ('ema_period', 200),     # Period for the EMA trend filter
        ('adx_period', 14),      # Period for ADX calculation
        ('adx_threshold', 25),   # ADX level to distinguish strong trend
        ('printlog', False),     # Enable/disable logging (set False for
cleaner article)
    )

    def __init__(self):
        # Indicators
        self.rsi =
bt.indicators.RelativeStrengthIndex(period=self.params.rsi_period)
        self.ema =
bt.indicators.ExponentialMovingAverage(period=self.params.ema_period)
        self.adx =
bt.indicators.AverageDirectionalMovementIndex(period=self.params.adx_period)

        # Order tracking and price references
        self.order = None
        self.dataclose = self.datas[0].close

    def log(self, txt, dt=None, doprint=False):
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()} {txt}')

    def notify_order(self, order):
        # (Keep the notify_order method exactly the same as the previous
version)
        if order.status in [order.Submitted, order.Accepted]:
            return
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(f'BUY EXECUTED, Price: {order.executed.price:.2f},
Size: {order.executed.size:.4f}', doprint=True)
            elif order.issell():
                self.log(f'SELL EXECUTED, Price: {order.executed.price:.2f},
Size: {order.executed.size:.4f}', doprint=True)
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log(f'Order Canceled/Margin/Rejected: Status
{order.getstatusname()}', doprint=True)
        self.order = None

    def notify_trade(self, trade):
        # (Keep the notify_trade method exactly the same as the previous
version)
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS {trade.pnl:.2f}, NET
{trade.pnlcomm:.2f}', doprint=True)
```

```
    def next(self):
        if self.order:
            return

        current_rsi = self.rsi[0]
        # Add checks to ensure indicators have calculated enough values
        if len(self.adx.adx) < 1 or len(self.ema.ema) < 1:
             return # Wait for indicators to have data

        current_adx = self.adx.adx[0]
        current_close = self.dataclose[0]
        current_ema = self.ema[0]
        current_position_size = self.position.size

        is_strong_trend = current_adx > self.params.adx_threshold
        is_uptrend = current_close > current_ema
        trend_status_log = ""

        # --- ENTRY LOGIC (only if flat) ---
        if current_position_size == 0:
            if is_strong_trend:
                if is_uptrend:
                    trend_status_log = f"Strong Uptrend
(ADX={current_adx:.2f})"
                    if current_rsi < self.params.oversold:
                        self.log(f'{trend_status_log} - BUY CREATE (Long
Entry), RSI={current_rsi:.2f}', doprint=True)
                        self.order = self.buy()
                else: # Strong Downtrend
                    trend_status_log = f"Strong Downtrend
(ADX={current_adx:.2f})"
                    if current_rsi > self.params.overbought:
                        self.log(f'{trend_status_log} - SELL CREATE (Short
Entry), RSI={current_rsi:.2f}', doprint=True)
                        self.order = self.sell()
            else: # Weak/Ranging Trend
                trend_status_log = f"Weak/Ranging Trend
(ADX={current_adx:.2f})"
                if current_rsi < self.params.oversold:
                    self.log(f'{trend_status_log} - BUY CREATE (Long Entry),
RSI={current_rsi:.2f}', doprint=True)
                    self.order = self.buy()
                elif current_rsi > self.params.overbought:
                    self.log(f'{trend_status_log} - SELL CREATE (Short
Entry), RSI={current_rsi:.2f}', doprint=True)
                    self.order = self.sell()

        # --- EXIT LOGIC (Based on RSI, independent of trend filter for exit)
---
```

```
        else: # Already in a position
            if current_position_size > 0: # Exit LONG
                if current_rsi > self.params.overbought or current_rsi >
self.params.neutral:
                    exit_condition = "Overbought" if current_rsi >
self.params.overbought else "Neutral Reversion"
                    self.log(f'SELL CREATE (Long Exit - {exit_condition}),
RSI={current_rsi:.2f}', doprint=True)
                    self.order = self.sell()
            elif current_position_size < 0: # Exit SHORT
                if current_rsi < self.params.neutral or current_rsi <
self.params.oversold:
                     exit_condition = "Oversold" if current_rsi <
self.params.oversold else "Neutral Reversion"
                     self.log(f'BUY CREATE (Short Cover - {exit_condition}),
RSI={current_rsi:.2f}', doprint=True)
                     self.order = self.buy()
```

The Transformation: Backtesting the Filtered Strategy



The RsiMeanReversionTrendFiltered strategy, now enhanced with trend awareness, is backtested on the same historical data. The results, now, showe a remarkable turnaround:

- Total Return: +15% (Turning a significant loss into a respectable profit)
- Maximum Drawdown: 35%

The improvement is staggering. The 60% loss turns into a 15% gain. More importantly, the catastrophic 350% drawdown is slashed by 90% to a much more manageable (though still significant because we don't have even a basic risk management!) 35%. By simply avoiding entries that directly fight strong trends, the strategy preserves capital during unfavourable periods and capitalizes on opportunities more aligned with the market's flow.

Comparing the Results: Side-by-Side

Metric	Basic RSI Mean Reversion (RsiMeanReversion)	Trend-Filtered RSI (RsiMeanReversionTrendFiltered)	Improvement
Total Return	-60%	+15%	Loss -> Profit
Max Drawdown	350%	35%	Reduced by 90%

Key Lessons and Caveats

This comparison underscores several critical points for strategy developers and traders:

1. Context is Crucial: Indicators like RSI rarely work well in isolation. Understanding the broader market context, especially the prevailing trend, is vital.
2. Simplicity Can Be Deceptive: While simple rules are appealing, they often lack the robustness to handle diverse market conditions.
3. Filters Can Add Value: Intelligent filters, like the EMA and ADX used here, can significantly improve a strategy's risk-adjusted returns by adapting its behaviour.
4. Drawdown Matters Immensely: A strategy is only viable if its drawdowns are survivable, both financially and psychologically. Reducing drawdown is often more important than maximizing raw returns.
5. Backtesting is Essential (But Not Perfect): This experiment highlights the power of backtesting to reveal flaws and test enhancements. However, remember that these results are specific to the tested period, asset, and parameters. There's always a risk of overfitting, and past performance doesn't guarantee future results.

Conclusion

The journey from a -60% return and 350% drawdown to a +15% return and 35% drawdown showcases the power of adding adaptive logic to a simple trading concept. The basic RSI mean reversion strategy, applied blindly, was a recipe for disaster in trending markets, as shown in our hypothetical backtest. By incorporating basic trend direction and strength filters, the strategy was able to navigate market regimes more intelligently, dramatically improving its performance and survivability in this specific example. It serves as a powerful reminder that successful trading often lies not just in finding signals, but in understanding when and when not to act on them.**

Trading Bitcoin with Adaptive Volatility

Trading volatile assets like Bitcoin presents unique challenges. Prices can exhibit strong trends but also experience rapid changes in volatility and sharp reversals. A rigid trading strategy might struggle in such dynamic conditions. This article explores a Python strategy using the Backtrader framework designed to adapt – employing Kaufman's Adaptive Moving Average (KAMA) combined with volatility bands and a trailing stop-loss.

The Strategy Concept: Adapting to the Flow

The core idea is to build a system that adjusts its parameters based on market conditions:

1. **Adaptive Trend Following:** Instead of a simple moving average (SMA) or exponential moving average (EMA) with fixed lookback periods, we use Kaufman's Adaptive Moving Average (KAMA). KAMA automatically adjusts its smoothing based on market volatility (price chop). It becomes more sensitive during strong trends and less sensitive during choppy periods, potentially reducing whipsaws.
2. **Dynamic Volatility Bands:** Standard deviation bands are created, but instead of being based on the price's standard deviation (like Bollinger Bands), they are based on the standard deviation *of the KAMA line itself*. The idea is that when KAMA is trending smoothly (low internal volatility), the bands will be tighter, and when KAMA is choppy (high internal volatility), the bands will widen.
3. **Flexible Entry:** The strategy incorporates two distinct entry modes:
    - `'pullback'`: Enter when the price crosses the KAMA line, aiming to catch moves as price returns to the dynamic average.
    - `'breakout'`: Enter when the price breaks outside the calculated volatility bands, aiming to capture strong momentum moves.
4. **Dual Exit Logic:** Positions are exited based on whichever condition occurs first:
    - The price crossing back over the KAMA line against the direction of the trade.
    - A percentage-based trailing stop-loss being hit, designed to lock in profits or cut losses.

Implementation in Backtrader

Let's look at the key code sections within the `AdaptiveMAVolatilityStrategy` class.

1. Initialization and Indicator Setup (`__init__`)

Python

```
def __init__(self):
    self.dataclose = self.datas[0].close # Reference to close price

    # KAMA: Adaptive Moving Average (uses internal fast/slow periods)
    self.kama = bt.indicators.KAMA(period=self.p.kama_period)
```

```
        # Standard Deviation of the KAMA line itself
        self.kama_stddev = bt.indicators.StandardDeviation(self.kama,
period=self.p.vol_period)

        # Calculate adaptive volatility bands
        self.upper_band = self.kama + self.p.vol_mult * self.kama_stddev
        self.lower_band = self.kama - self.p.vol_mult * self.kama_stddev

        # Indicator to detect price crossing KAMA
        self.kama_cross = bt.indicators.CrossOver(self.dataclose, self.kama)

        # Order tracking variables
        self.order = None         # Tracks main entry/exit orders
        self.order_trail = None   # Tracks the trailing stop order
        # ... other initializations ...
```

Here, we set up the core indicators: KAMA, its standard deviation, the upper/lower bands based on that deviation, and a `CrossOver` indicator for easy signal detection against KAMA. We also initialize variables to keep track of pending orders.

2. Core Trading Logic (`next`)

The `next` method runs on each bar and contains the primary decision-making logic.

Python

```
    def next(self):
        # Check if any order is pending; if so, do nothing
        if self.order or self.order_trail:
            return

        # --- Entry Logic ---
        if not self.position: # Only enter if not already in the market
            entry_signal = False
            # ** Pullback Mode Entry **
            if self.p.trade_mode == 'pullback':
                if self.kama_cross[0] > 0: # Buy on cross above KAMA
                    self.log(f'PULLBACK BUY SIGNAL...')
                    self.order = self.buy()
                    entry_signal = True
                elif self.kama_cross[0] < 0: # Sell on cross below KAMA
                    self.log(f'PULLBACK SELL SIGNAL...')
                    self.order = self.sell()
                    entry_signal = True

            # ** Breakout Mode Entry **
            elif self.p.trade_mode == 'breakout':
                # Buy on close above Upper Band (after being below/on it)
                if self.dataclose[0] > self.upper_band[0] and
```

```
self.dataclose[-1] <= self.upper_band[-1]:
                     self.log(f'BREAKOUT BUY SIGNAL...')
                     self.order = self.buy()
                     entry_signal = True
                # Sell on close below Lower Band (after being above/on it)
                elif self.dataclose[0] < self.lower_band[0] and
self.dataclose[-1] >= self.lower_band[-1]:
                     self.log(f'BREAKOUT SELL SIGNAL...')
                     self.order = self.sell()
                     entry_signal = True

            if entry_signal: return # Stop processing if entry order placed

        # --- KAMA-Based Exit Logic ---
        else: # Already in the market
            # Close Long if price crosses below KAMA
            if self.position.size > 0 and self.kama_cross[0] < 0:
                 self.log(f'KAMA CLOSE LONG SIGNAL...')
                 self.order = self.close()
            # Close Short if price crosses above KAMA
            elif self.position.size < 0 and self.kama_cross[0] > 0:
                 self.log(f'KAMA CLOSE SHORT SIGNAL...')
                 self.order = self.close()
```

This section first checks if we can trade (no pending orders, not already in a position for entries). Based on the chosen `trade_mode`, it looks for either KAMA crosses or band breakouts. If an entry occurs, it exits the next method for that bar. If already in a position, it checks for the KAMA cross exit condition.

3. Risk Management: The Trailing Stop (`notify_order`)

A crucial part is managing the trailing stop. This logic resides primarily within the `notify_order` method, which Backtrader calls whenever an order's status changes.

Python

```
# Inside notify_order, when an ENTRY order completes successfully:
        # ... (logic to detect entry completion) ...
        if order.isbuy(): # If entry was a BUY
             self.log(f'BUY EXECUTED ...')
             if self.p.trail_perc: # If trailing stop is enabled
                  self.log(f'PLACING TRAILING SELL ORDER...')
                  # Place the sell trailing stop order
                  self.order_trail = self.sell(exectype=bt.Order.StopTrail,

trailpercent=self.p.trail_perc)
        elif order.issell(): # If entry was a SELL
             self.log(f'SELL EXECUTED ...')
             if self.p.trail_perc: # If trailing stop is enabled
```

```
                self.log(f'PLACING TRAILING BUY ORDER...')
                # Place the buy trailing stop order
                self.order_trail = self.buy(exectype=bt.Order.StopTrail,
                                            trailpercent=self.p.trail_perc)

# Also in notify_order, if the KAMA-based EXIT order completes:
        # ... (logic to detect KAMA exit completion) ...
        if not self.position and self.order_trail is not None: # If position
closed by KAMA exit
             self.log(f'KAMA EXIT EXECUTED. CANCELLING PENDING TRAIL
ORDER...')
             self.cancel(self.order_trail) # Cancel the now redundant trail
order
             self.order_trail = None
```

This logic ensures that *after* an entry order is successfully filled, the corresponding trailing stop order (`bt.Order.StopTrail`) is placed. It also handles canceling this trailing stop if the position is closed by the KAMA cross signal first.

Backtesting Setup

The `if __name__ == '__main__':` block sets up and runs the backtest. The provided code uses specific settings for a test run:

Python

```
if __name__ == '__main__':
    cerebro = bt.Cerebro()

    # Strategy Configuration for this run
    selected_trade_mode = 'breakout'
    trailing_stop_percentage = 0.10 # 10% trailing stop

    cerebro.addstrategy(AdaptiveMAVolatilityStrategy,
                        trade_mode=selected_trade_mode,
                        kama_period=30,   # Slower KAMA
                        vol_period=7,     # Faster Volatility calc
                        vol_mult=3.0,     # Wider Bands
                        trail_perc=trailing_stop_percentage)

    # Data Fetching (BTC-USD for 2021-2023)
    ticker = 'BTC-USD'
    start_date = '2021-01-01'
    end_date = '2023-12-31'
    # ... (yf.download and data feed creation) ...
    cerebro.adddata(data_feed)

    # Realistic Broker Simulation Settings
    cerebro.broker.setcash(100000.0)     # Starting capital
    cerebro.broker.setcommission(commission=0.001) # 0.1% commission
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95) # Position sizing
(Note: 95% is very aggressive)

    # Analyzers for performance metrics
    cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe_ratio')
    # ... (add Returns, DrawDown, TradeAnalyzer) ...

    # Run, Print Results, Plot
    print('Starting Portfolio Value: %.2f' % cerebro.broker.getvalue())
    results = cerebro.run()
    print('Final Portfolio Value: %.2f' % cerebro.broker.getvalue())
    # ... (print analyzer results) ...
    # ... (plotting code) ...
```

This setup tests the strategy in `'breakout'` mode with specific parameters (KAMA 30, Vol Period 7, Vol Mult 3.0, Trail 10%) on Bitcoin data from 2021 to 2023, using realistic starting capital, commission, and percentage-based position sizing (although 95% is very high risk).

Complete Code

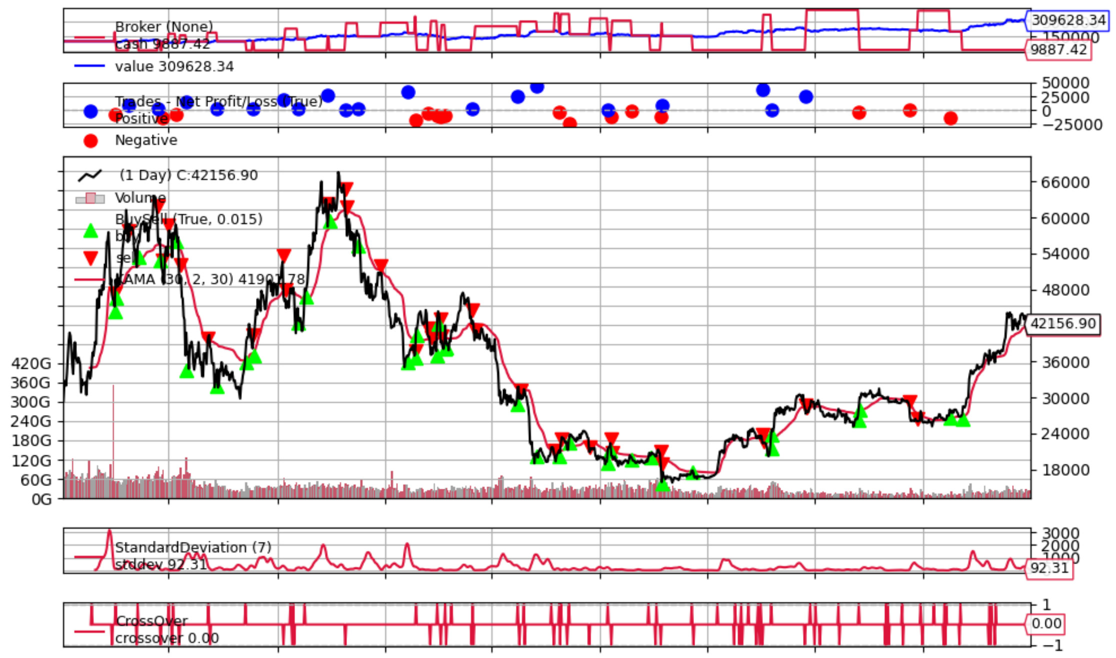
```
import backtrader as bt
import yfinance as yf
import datetime
import matplotlib.pyplot as plt
%matplotlib qt5

class AdaptiveMAVolatilityStrategy(bt.Strategy):
    """
    Strategy using KAMA with volatility bands and a trailing stop-loss.
    Trades pullbacks or breakouts, exits on KAMA cross OR trailing stop.
    """
    params = (
        ('kama_period', 20),
        ('fast_ema', 2),      # Only relevant if using custom KAMA
        ('slow_ema', 30),     # Only relevant if using custom KAMA
        ('vol_period', 20),
        ('vol_mult', 2.0),
        ('trade_mode', 'pullback'), # 'pullback' or 'breakout'
        ('trail_perc', 0.05), # Trailing stop percentage (e.g., 0.05 = 5%)
        ('printlog', True),
    )

    def log(self, txt, dt=None, doprint=False):
        ''' Logging function '''
        if self.params.printlog or doprint:
            dt = dt or self.datas[0].datetime.date(0)
            print(f'{dt.isoformat()} - {txt}')
    def __init__(self):
        self.dataclose = self.datas[0].close
        self.kama = bt.indicators.KAMA(period=self.p.kama_period) # Uses
internal fast/slow
        self.kama_stddev = bt.indicators.StandardDeviation(self.kama,
period=self.p.vol_period)
        self.upper_band = self.kama + self.p.vol_mult * self.kama_stddev
        self.lower_band = self.kama - self.p.vol_mult * self.kama_stddev
        self.kama_cross = bt.indicators.CrossOver(self.dataclose, self.kama)

        # Order tracking
        self.order = None         # For entry or KAMA-based exit orders
        self.order_trail = None   # For the trailing stop order
        self.buyprice = None
        self.buycomm = None

        if self.p.trade_mode not in ['pullback', 'breakout']:
            raise ValueError("trade_mode parameter must be 'pullback' or
'breakout'")
        if self.p.trail_perc is not None and self.p.trail_perc <= 0:
            raise ValueError("trail_perc must be positive or None")

        self.log(f"Strategy Initialized: KAMA(Period={self.p.kama_period}), "
                 f"VolBands(StdDev({self.p.vol_period}),
Mult={self.p.vol_mult}), "
                 f"TradeMode={self.p.trade_mode},
TrailPerc={self.p.trail_perc}", doprint=True)
        self.log(f"Note: Standard bt.indicators.KAMA uses internal fast/slow
periods (typically 2 and 30).", doprint=True)


    def notify_order(self, order):
        # --- Order Status ---
        if order.status == order.Submitted:
            self.log(f'ORDER SUBMITTED: Ref:{order.ref},
Type:{order.ordtypename()}, Size:{order.size}, Price:{order.price}')
            return
        if order.status == order.Accepted:
             self.log(f'ORDER ACCEPTED: Ref:{order.ref},
Type:{order.ordtypename()}')
             return

        # --- Order Completion/Rejection ---
        if order.status in [order.Completed, order.Canceled, order.Margin,
order.Rejected]:
            otype = order.ordtypename()
            # Get execution details safely, providing defaults for non-
```

```
completed orders
            price = order.executed.price if order.status == order.Completed
else None
            # Use executed size if available, otherwise submitted size
(submitted size might be more informative for rejected orders)
            size = order.executed.size if order.status == order.Completed and
order.executed.size is not None else order.size
            comm = order.executed.comm if order.status == order.Completed
else 0.0
            pnl = order.executed.pnl if order.status == order.Completed and
order.executed.pnl else 0.0 # Defaults to 0.0

            # Prepare formatted strings to handle None values gracefully for
logging
            exec_price_str = f"{price:.2f}" if price is not None else "N/A"
            # Handle order.price which might be 0 or None for Market
orders/some brokers
            req_price_val = order.price if order.price is not None and
order.price != 0.0 else None
            req_price_str = f"{req_price_val:.2f}" if req_price_val is not
None else "Market/None"
            comm_str = f"{comm:.2f}" # Safe as it defaults to 0.0
            pnl_str = f"{pnl:.2f}"   # Safe as it defaults to 0.0

            # --- CORRECTED LOG LINE ---
            # Uses the pre-formatted safe strings (exec_price_str,
req_price_str, etc.)
            self.log(f'ORDER COMPLETE/CANCEL/REJECT: Ref:{order.ref},
Type:{otype}, Status:{order.getstatusname()}, '
                     f'Size:{size}, ExecPrice:{exec_price_str}
(Req:{req_price_str}), Comm:{comm_str}, Pnl:{pnl_str}')

            # --- Order Tracking Logic --- (Rest of the function is the same)
            is_entry = order.ref == getattr(self.order, 'ref', None)
            is_trail = order.ref == getattr(self.order_trail, 'ref', None)

            if is_entry:
                if order.status == order.Completed:
                    # --- ENTRY COMPLETED ---
                    if order.isbuy():
                        self.log(f'BUY EXECUTED @ {price:.2f}, Size: {size}')
# Use 'price' here as it's confirmed not None
                        self.buyprice = price
                        self.buycomm = comm
                        if self.p.trail_perc:
                            stop_price = price * (1.0 - self.p.trail_perc)
                            self.log(f'PLACING TRAILING SELL ORDER: Trail %:
{self.p.trail_perc * 100:.2f}, Initial Stop: {stop_price:.2f}')
                            self.order_trail =
self.sell(exectype=bt.Order.StopTrail, trailpercent=self.p.trail_perc)
                            self.order_trail.addinfo(name="TrailStopSell")
                    elif order.issell():
                        self.log(f'SELL EXECUTED @ {price:.2f}, Size:
{size}') # Use 'price' here as it's confirmed not None
                        self.buyprice = price
                        self.buycomm = comm
                        if self.p.trail_perc:
                             stop_price = price * (1.0 + self.p.trail_perc)
                             self.log(f'PLACING TRAILING BUY ORDER: Trail %:
{self.p.trail_perc * 100:.2f}, Initial Stop: {stop_price:.2f}')
                             self.order_trail =
self.buy(exectype=bt.Order.StopTrail, trailpercent=self.p.trail_perc)
                             self.order_trail.addinfo(name="TrailStopBuy")

                # --- KAMA-EXIT COMPLETED ---
                if not self.position and self.order_trail is not None:
                     # Check status just to be sure the order causing
flatness was completed, not rejected.
                     if order.status == order.Completed:
                          self.log(f'KAMA EXIT EXECUTED. CANCELLING PENDING
TRAIL ORDER Ref: {self.order_trail.ref}')
                          self.cancel(self.order_trail)
                          self.order_trail = None

                # Reset main order tracker only if the main order event
finished (completed, canceled, rejected)
                if order.status in [order.Completed, order.Canceled,
order.Margin, order.Rejected]:
                    self.order = None

            elif is_trail:
                 # --- TRAILING STOP EVENT (EXECUTED, CANCELED, REJECTED) ---
                 if order.status == order.Completed:
                      # Use 'price' here as it's confirmed not None
                      self.log(f'TRAILING STOP EXECUTED @ {price:.2f}, Size:
{size}, Pnl: {pnl:.2f}')
                 else:
                      self.log(f'TRAILING STOP CANCELED/REJECTED Status:
{order.getstatusname()}')

                 # Reset trail order tracker regardless of completion status
                 self.order_trail = None

            else:
                 # Order completion notification for an unknown order ref?
Should not happen often.
                 # Could be an manually cancelled order if broker had such
API interaction?
```

```
                 self.log(f"WARN: notify_order received for unrecognized
order Ref: {order.ref}, Status: {order.getstatusname()}")

        # --- End notify_order -----



    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log(f'OPERATION PROFIT, GROSS: {trade.pnl:.2f}, NET:
{trade.pnlcomm:.2f}')

    def next(self):
        # Check if any order is pending
        if self.order or self.order_trail:
            # self.log(f"Skipping next(): Pending order exists. self.order:
{self.order}, self.order_trail: {self.order_trail}")
            return

        # --- Entry Logic ---
        if not self.position:
            entry_signal = False
            # ** Pullback Mode **
            if self.p.trade_mode == 'pullback':
                if self.kama_cross[0] > 0: # Buy Signal
                    self.log(f'PULLBACK BUY SIGNAL: Close
{self.dataclose[0]:.2f} > KAMA {self.kama[0]:.2f}')
                    self.order = self.buy()
                    entry_signal = True
                elif self.kama_cross[0] < 0: # Sell Signal (optional
shorting)
                    self.log(f'PULLBACK SELL SIGNAL: Close
{self.dataclose[0]:.2f} < KAMA {self.kama[0]:.2f}')
                    self.order = self.sell()
                    entry_signal = True

            # ** Breakout Mode **
            elif self.p.trade_mode == 'breakout':
                if self.dataclose[0] > self.upper_band[0] and
self.dataclose[-1] <= self.upper_band[-1]: # Buy Signal
                     self.log(f'BREAKOUT BUY SIGNAL: Close
{self.dataclose[0]:.2f} > Upper Band {self.upper_band[0]:.2f}')
                     self.order = self.buy()
                     entry_signal = True
                elif self.dataclose[0] < self.lower_band[0] and
self.dataclose[-1] >= self.lower_band[-1]: # Sell Signal (optional shorting)
                     self.log(f'BREAKOUT SELL SIGNAL: Close
{self.dataclose[0]:.2f} < Lower Band {self.lower_band[0]:.2f}')
```

```
                    self.order = self.sell()
                    entry_signal = True

            # If an entry order was placed, stop processing for this bar
            if entry_signal:
                return

        # --- KAMA-Based Exit Logic ---
        # Only consider KAMA exit if we are in a position AND no trailing
stop order is currently active
        # Note: We place the KAMA exit even if a trail order exists, letting
the broker handle which executes first.
        # The cancellation logic is handled in notify_order.
        else: # Already in the market
            # Exit Long: Price crosses below KAMA
            if self.position.size > 0 and self.kama_cross[0] < 0:
                 self.log(f'KAMA CLOSE LONG SIGNAL: Close
{self.dataclose[0]:.2f} < KAMA {self.kama[0]:.2f}')
                 self.order = self.close() # Close position via KAMA cross

            # Exit Short: Price crosses above KAMA
            elif self.position.size < 0 and self.kama_cross[0] > 0:
                 self.log(f'KAMA CLOSE SHORT SIGNAL: Close
{self.dataclose[0]:.2f} > KAMA {self.kama[0]:.2f}')
                 self.order = self.close() # Close position via KAMA cross

    def stop(self):
        trail_info = f"TrailPerc={self.p.trail_perc}" if self.p.trail_perc
else "NoTrail"
        self.log(f'(KAMA Period {self.p.kama_period}, {trail_info}) Ending
Value {self.broker.getvalue():.2f}', doprint=True)


# --- Main Execution ---
if __name__ == '__main__':
    # Create a Cerebro entity
    cerebro = bt.Cerebro()

    # selected_trade_mode = 'pullback'
    selected_trade_mode = 'breakout'
    # Set trail_perc=None to disable trailing stop
    trailing_stop_percentage = 0.1 # e.g., 5% trailing stop

    cerebro.addstrategy(AdaptiveMAVolatilityStrategy,
                        trade_mode=selected_trade_mode,
                        kama_period=30,
                        vol_period=7,
                        vol_mult=3.0,
                        trail_perc=trailing_stop_percentage,
                        printlog=False)

    # --- Data Loading --- (Same as before)
    ticker = 'BTC-USD'
    start_date = '2021-01-01'
    end_date = '2023-12-31'

    print(f"Fetching data for {ticker} from {start_date} to {end_date}")

    # Fetch data using yfinance
    try:
        data_df = yf.download(ticker, start=start_date, end=end_date,
progress=False)

        if data_df.empty:
             print(f"No data fetched for {ticker}. Check ticker symbol or
date range.")
             exit()

        data_df.columns = data_df.columns.droplevel(1)


        # Create a Backtrader data feed
        data_feed = bt.feeds.PandasData(dataname=data_df)

        # Add the Data Feed to Cerebro
        cerebro.adddata(data_feed)

    except Exception as e:
        print(f"Error fetching or processing data: {e}")
        exit()


    # --- Backtest Configuration ---
    # Set starting cash
    cerebro.broker.setcash(100000.0)
    cerebro.broker.setcommission(commission=0.001)
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

    # --- Analyzers --- (Same as before)
    cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe_ratio',
timeframe=bt.TimeFrame.Days)
    cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trade_analyzer')
    # --- Run Backtest ---
    print('Starting Portfolio Value: %.2f' % cerebro.broker.getvalue())
    results = cerebro.run()
    print('Final Portfolio Value: %.2f' % cerebro.broker.getvalue())

    # --- Print Analyzer Results --- (Same as before)
    strat = results[0]
    print('\n--- Analyzer Results ---')
    print(f"Sharpe Ratio:
{strat.analyzers.sharpe_ratio.get_analysis().get('sharperatio', 'N/A')}")
    returns_dict = strat.analyzers.returns.get_analysis()
    print(f"Total Return: {returns_dict.get('rtot', 'N/A'):.4f}")
    print(f"Average Annual Return: {returns_dict.get('ravg', 'N/A'):.4f}")
    print(f"Max Drawdown:
{strat.analyzers.drawdown.get_analysis().max.drawdown:.2f}%")
    trade_analysis = strat.analyzers.trade_analyzer.get_analysis()
    if trade_analysis and trade_analysis.total.total > 0:
         print("\n--- Trade Analysis ---")
         print(f"Total Trades: {trade_analysis.total.total}")
         print(f"Winning Trades: {trade_analysis.won.total}")
         print(f"Losing Trades: {trade_analysis.lost.total}")
         print(f"Win Rate: {trade_analysis.won.total /
trade_analysis.total.total * 100:.2f}%")
         print(f"Average Win ($): {trade_analysis.won.pnl.average:.2f}")
         print(f"Average Loss ($): {trade_analysis.lost.pnl.average:.2f}")
         pf = abs(trade_analysis.won.pnl.total /
trade_analysis.lost.pnl.total) if trade_analysis.lost.pnl.total != 0 else
float('inf')
         print(f"Profit Factor: {pf:.2f}")
    else:
        print("\n--- Trade Analysis ---")
        print("No trades executed.")

    # --- Plotting ---
    try:
        # To plot in environments like Jupyter, add %matplotlib inline at the
top
        # or configure matplotlib backend appropriately.
        # Set plot volume=False if volume data is noisy or not needed on main
chart
        plt.rcParams['figure.figsize'] = [10, 6]
        cerebro.plot(style='line', barup='green', bardown='red', volume=True,
iplot=False)
        plt.tight_layout()
    except Exception as e:
        print(f"\nCould not plot results: {e}")
        print("Plotting requires matplotlib installed and a suitable display
environment.")
```

```
Fetching data for BTC-USD from 2021-01-01 to 2023-12-31
Starting Portfolio Value: 100000.00
2023-12-30 - Strategy Initialized: KAMA(Period=30), VolBands(StdDev(7),
Mult=3.0), TradeMode=breakout, TrailPerc=0.1
2023-12-30 - Note: Standard bt.indicators.KAMA uses internal fast/slow
periods (typically 2 and 30).
2023-12-30 - (KAMA Period 30, TrailPerc=0.1) Ending Value 309628.34
Final Portfolio Value: 309628.34

--- Analyzer Results ---
Sharpe Ratio: 0.053511740895852936
Total Return: 1.1302
Average Annual Return: 0.0010
Max Drawdown: 39.60%

--- Trade Analysis ---
Total Trades: 36
Winning Trades: 19
Losing Trades: 16
Win Rate: 52.78%
Average Win ($): 13470.19
Average Loss ($): -9680.71
Profit Factor: 1.65
```



Pasted image 20250427140413.png

Pasted image 20250427140413.png

Potential Use Cases

This type of adaptive strategy could be useful in several scenarios:

- **Volatile Assets:** Designed specifically for markets like cryptocurrencies or volatile stocks where fixed-parameter strategies might struggle.
- **Trend Following with Adaptability:** Aims to capture trends (via KAMA) while adjusting band width and smoothing based on volatility.
- **Breakout or Pullback Systems:** The flexible `trade_mode` allows testing both common entry styles within the same adaptive framework.
- **Component for Larger Systems:** The adaptive MA or bands could be used as filters or components within more complex multi-strategy systems.

Suggestions for Improvement

While the strategy incorporates adaptive elements, backtesting often reveals areas for enhancement:

1. **Parameter Optimization:** The chosen parameters (`kama_period`, `vol_period`, `vol_mult`, `trail_perc`) heavily influence performance. Systematic optimization (e.g., using Backtrader's built-in optimizer or manual grid search) across different market periods is crucial. Remember to optimize *after* setting realistic sizing.
2. **Risk Management Refinement:**
   - **Sizing:** The 95% `PercentSizer` is extremely aggressive. Test much lower percentages (e.g., 2%, 5%, 10%, 20%) for more realistic risk control and reduced impact of losing streaks.
   - **Trailing Stop Method:** A fixed percentage trail might not be optimal for Bitcoin's volatility. Consider an ATR (Average True Range)-based trailing stop (`bt.Order.StopTrailATR` or manual calculation) which adapts the stop distance to market volatility.
3. **Entry/Exit Signal Filters:**
   - **Trend Filter:** Add a longer-term moving average (e.g., SMA100 or SMA200) and only allow long entries above it / short entries below it to avoid trading against the dominant trend.
   - **Volatility Filter:** Avoid entries if volatility (e.g., ATR percentage) is excessively high (indicating potential blow-offs) or too low (indicating potential chop).
   - **Confirmation:** Require price to close outside the band or across the KAMA for *two* bars instead of one to reduce false signals.
   - **Alternative Exits:** Test different exit criteria, such as reaching the opposite volatility band or fixed profit targets.
4. **Band Calculation:** Experiment with using price ATR for bands (similar to Keltner Channels) instead of the KAMA's standard deviation. Compare performance.

5. **Regime Filtering:** Advanced: Attempt to classify the market (e.g., high-trend/low-volatility vs. low-trend/high-volatility) and potentially adjust strategy parameters or enable/disable trading based on the detected regime.
6. **Robustness Testing:** Test across different assets, timeframes, and market periods (including bear markets) to assess how robust the strategy is. Perform walk-forward optimization for a more realistic performance expectation.

Conclusion

This Backtrader strategy provides a framework for trading volatile assets like Bitcoin by adapting to market conditions using KAMA and dynamic volatility bands derived from KAMA's own behavior. The inclusion of a trailing stop adds a layer of risk management. However, like any trading strategy, its "out-of-the-box" performance depends heavily on parameter choices, market conditions, and realistic configuration (especially position sizing and costs). Thorough backtesting, optimization, and refinement are essential steps before considering any strategy for live deployment. This code serves as a solid starting point for exploring adaptive trading concepts in Python.

Which Bitcoin Indicators Actually Predict the Next Move

In the quest to build profitable trading strategies, particularly in volatile markets like Bitcoin, identifying *which* technical indicators genuinely provide predictive information is a crucial first step. With dozens of indicators available, how do you sift through them to find those most relevant to future price movements?

This article guides you through a Python script designed for exactly this purpose. It provides a sophisticated approach to analyze the historical predictive power of approximately 30 different technical indicators for forecasting Bitcoin's next-day price direction. We'll break down the code step-by-step, explain how to run it, and discuss how to interpret the results using two distinct feature importance methods: **Mutual Information** and **Random Forest Importance**.

Objective:

The goal of this script is **not** to create a trading bot, but rather to perform **feature analysis**. It helps answer the question: "Based on historical data, which of these technical indicators had the strongest relationship with whether Bitcoin's price went up or down the next day?"

Prerequisites:

Before running the script, you need a Python environment with the following libraries installed:

- pandas: For data manipulation.

- `numpy`: For numerical operations.
- `yfinance`: To download market data from Yahoo Finance.
- `matplotlib` & `seaborn`: For plotting the results.
- `scikit-learn`: For preprocessing, feature selection (Mutual Information), and the Random Forest model.
- `TA-Lib`: A crucial library for calculating technical indicators. **Important:** Installing TA-Lib can sometimes be tricky as it requires the underlying TA-Lib C library to be installed first. Follow the official instructions carefully: https://mrjbq7.github.io/ta-lib/install.html

You can typically install the Python wrappers (once the C library is set up) using pip:

```
pip install pandas numpy yfinance matplotlib seaborn scikit-learn TA-Lib
```

How the Script Works: Step-by-Step Breakdown

The script follows a logical workflow from data acquisition to analysis:

A. Configuration

The script begins with a configuration section where you can easily modify key parameters for your analysis.

Python

```
#
=============================================================================
=
# Configuration
#
=============================================================================
=
TICKER = 'BTC-USD'          # Asset to analyze (e.g., 'ETH-USD', 'AAPL')
START_DATE = '2018-01-01'   # Start date for historical data
END_DATE = None             # End date (None uses latest data) or 'YYYY-MM-
DD'
PREDICTION_HORIZON = 1      # How many days ahead to predict direction (e.g.,
1 = next day)
TEST_SIZE = 0.2             # Proportion of data reserved (chronologically)
for potential later testing
                            # Note: This analysis primarily uses the training
portion
```

- **`TICKER`**: The symbol recognized by Yahoo Finance for the asset you want to analyze.
- **`START_DATE/END_DATE`**: Defines the period for historical data. A longer period (several years) is generally better for robustness.
- **`PREDICTION_HORIZON`**: Sets the timeframe for the direction prediction (e.g., 1 means predicting if the close price *tomorrow* will be higher than today's close).

- **`TEST_SIZE`**: Reserves the final 20% of the data chronologically. While this script focuses importance analysis on the *training* part (the first 80%), reserving a test set is good practice for *later* model validation if you build upon this analysis.

B. Data Loading

The `load_data` function fetches the necessary OHLCV (Open, High, Low, Close, Volume) data using `yfinance`.

Python

```
# In main execution block:
data = load_data(TICKER, START_DATE, END_DATE)
```

It includes error handling and basic column name standardization.

C. Indicator Calculation

The `calculate_indicators` function is the workhorse for feature engineering. It takes the raw OHLCV data and computes approximately 30 different technical indicators using the installed `TA-Lib` library.

Python

```
# Example snippets from inside the calculate_indicators function:

# Trend
df['SMA_20'] = talib.SMA(close, timeperiod=20)
df['ADX_14'] = talib.ADX(high, low, close, timeperiod=14)

# Momentum
df['RSI_14'] = talib.RSI(close, timeperiod=14)
df['MACD'], df['MACD_signal'], df['MACD_hist'] = talib.MACD(close,
fastperiod=12, slowperiod=26, signalperiod=9)

# Volatility
df['ATR_14'] = talib.ATR(high, low, close, timeperiod=14)
df['BB_upper'], df['BB_middle'], df['BB_lower'] = talib.BBANDS(close,
timeperiod=20, nbdevup=2, nbdevdn=2, matype=0)

# Volume (if available)
if volume is not None and not (volume == 0).all():
    df['OBV'] = talib.OBV(close, volume.astype(float))

# Other
df['High_Low'] = df['high'] - df['low']

# ... plus many others covering different indicator types ...
```

This function generates a wide range of potential predictor variables.

D. Target Variable Definition

The `create_target` function defines what we are trying to predict. It creates a binary `Target` column.

Python

```
# Inside create_target function:
def create_target(df, horizon=1):
    """Creates binary target variable: 1 if future price > current, 0
otherwise."""
    df['Future_Close'] = df['close'].shift(-horizon) # Look ahead 'horizon'
days
    # Target is 1 if the future price increased, 0 otherwise
    df['Target'] = (df['Future_Close'] > df['close']).astype(int)
    print(f"Target variable created for {horizon}-day future direction.")
    return df
```

Here, `Target` = 1 if the closing price `PREDICTION_HORIZON` days later is higher than the current day's closing price, and 0 otherwise.

E. Preprocessing

This stage prepares the data for analysis:

Python

```
# In main execution block:

# Drop rows with NaNs (essential after indicator/target calculation)
print(f"Shape before dropping NaNs: {data_target.shape}")
data_processed = data_target.dropna()
print(f"Shape after dropping NaNs: {data_processed.shape}")

# Separate Features (X) and Target (Y)
original_cols = ['open', 'high', 'low', 'close', 'adj_close', 'volume',
'Future_Close', 'Target']
features = [col for col in data_processed.columns if col not in
original_cols]
X = data_processed[features]
Y = data_processed['Target']

# Split data chronologically (using first 80% for importance analysis)
split_index = int(len(X) * (1 - TEST_SIZE))
X_train, X_test = X[:split_index], X[split_index:]
Y_train, Y_test = Y[:split_index], Y[split_index:]

# Scale features (important for some analyses, good practice)
```

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
# X_test_scaled = scaler.transform(X_test) # Scale test set if needed later
```

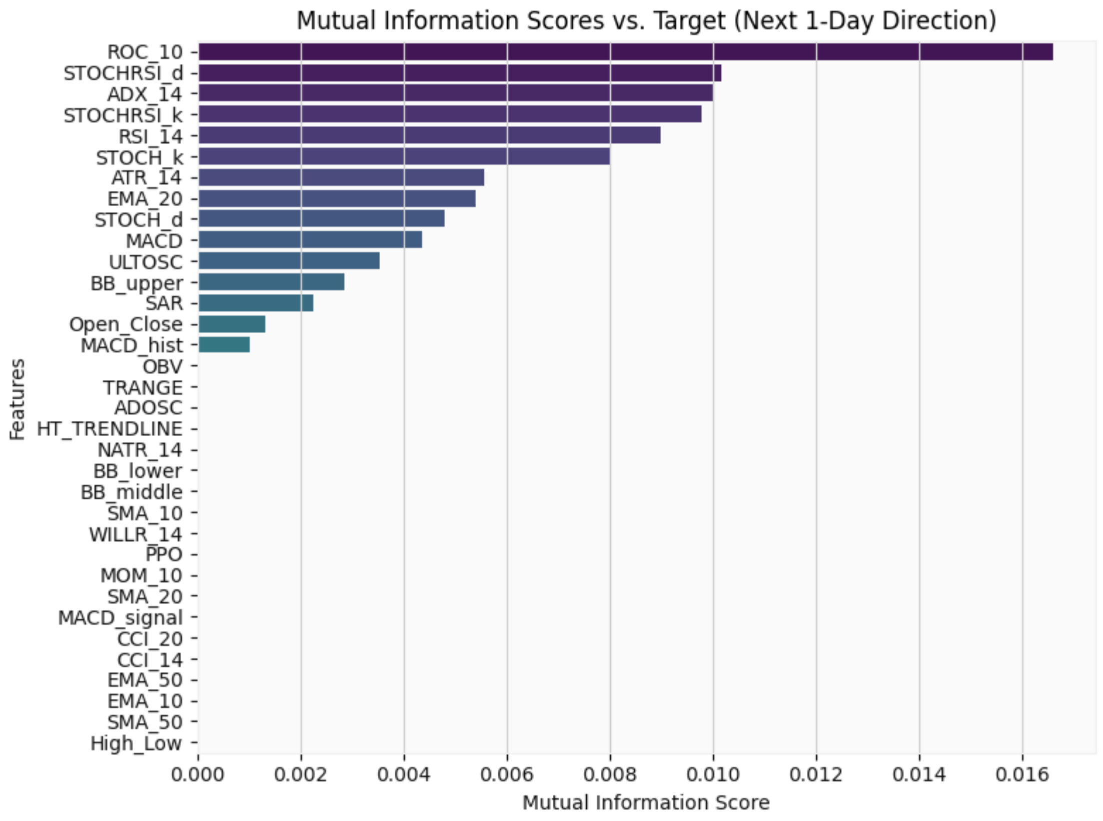
- **dropna()**: Removes rows containing NaN values, which are introduced by indicators requiring a lookback period (like moving averages) and by the target variable's shift.
- **Separating X/Y**: Isolates the indicator columns (X) from the Target column (Y).
- **Splitting Data**: Divides the data chronologically into a training set (used for this analysis) and a test set (reserved for future model validation).
- **Scaling**: Uses StandardScaler to standardize the features (mean=0, variance=1). This is helpful for mutual information calculation and often beneficial for machine learning models.

F. Predictive Power Analysis 1: Mutual Information

This analysis uses mutual_info_classif from scikit-learn to estimate the mutual information between each scaled feature and the binary target variable on the training data. Mutual information measures the reduction in uncertainty about the target variable given knowledge of the feature, capturing both linear and non-linear dependencies.

Python

```
# In main execution block:
print("\n--- Analyzing Feature Importance using Mutual Information ---")
try:
    # Ensure Y_train has enough samples and variance
    if len(Y_train.unique()) > 1 and len(Y_train) > 5:
        mi_scores = mutual_info_classif(X_train_scaled, Y_train,
discrete_features=False, random_state=42)
        mi_series = pd.Series(mi_scores,
index=features).sort_values(ascending=False)

        # Plotting using seaborn
        plt.figure(figsize=(12, 10))
        sns.barplot(x=mi_series.values, y=mi_series.index, palette='viridis')
        plt.title(f'Mutual Information Scores vs. Target (Next
{PREDICTION_HORIZON}-Day Direction)')
        plt.xlabel('Mutual Information Score')
        # ... rest of plotting ...
        plt.show()
        print("Top 15 Features (Mutual Information):\n", mi_series.head(15))
    # ... error handling ...
```



Pasted image 20250419171111.png

Pasted image 20250419171111.png

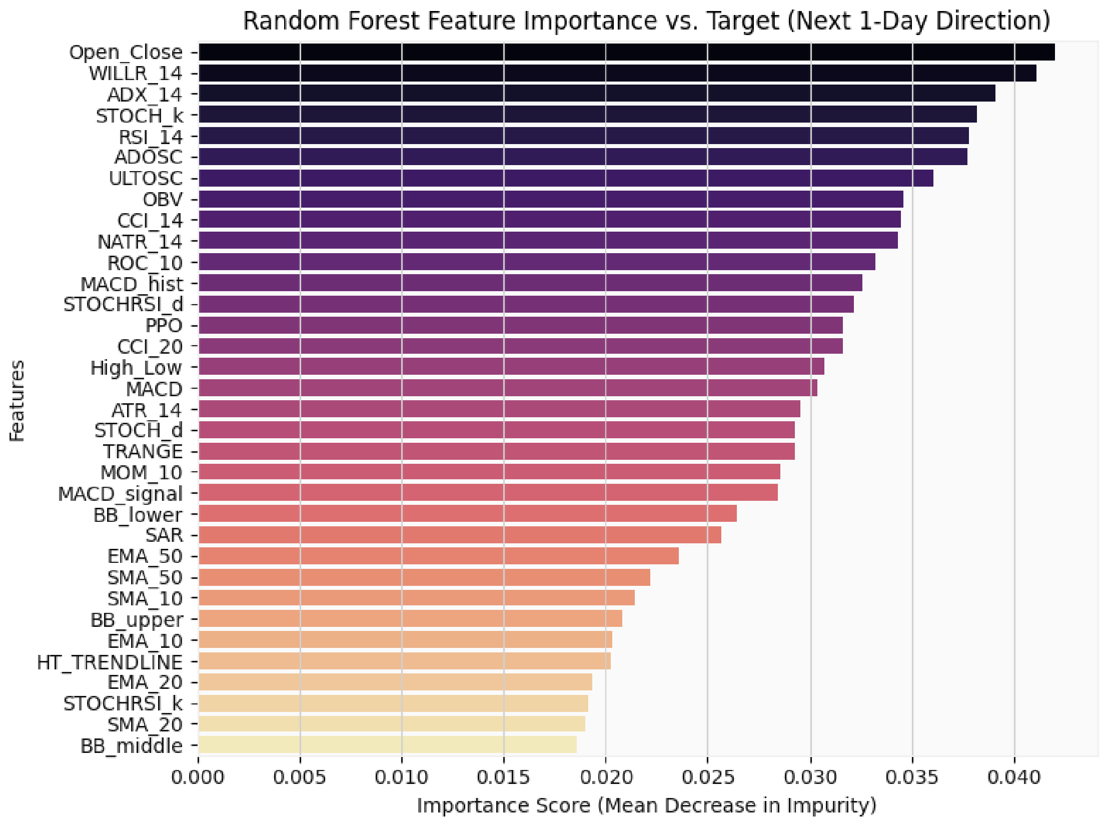
The bar chart visualizes these scores. Higher scores suggest a stronger statistical relationship between the indicator and the next day's price direction in the training data.

G. Predictive Power Analysis 2: Random Forest Importance

This method trains a `RandomForestClassifier` model on the scaled training data and then extracts the feature importances calculated by the model itself. For Random Forests, this is typically the "mean decrease in impurity" (Gini importance) – it measures how much, on average, splitting on a particular feature reduces the impurity (improves the classification) across all the trees in the forest.

Python

```
# In main execution block:
print("\n--- Analyzing Feature Importance using Random Forest ---")
try:
    if len(Y_train.unique()) > 1 and len(Y_train) > 5:
        rf_model = RandomForestClassifier(n_estimators=200,
                                          random_state=42,
                                          n_jobs=-1,
                                        max_depth=10,
                                        min_samples_leaf=5,
                                        class_weight='balanced') # Helps if
Ups/Downs are imbalanced
        rf_model.fit(X_train_scaled, Y_train)

        rf_importances = rf_model.feature_importances_
        rf_series = pd.Series(rf_importances,
index=features).sort_values(ascending=False)

        # Plotting using seaborn
        plt.figure(figsize=(12, 10))
        sns.barplot(x=rf_series.values, y=rf_series.index, palette='magma')
        plt.title(f'Random Forest Feature Importance vs. Target (Next
{PREDICTION_HORIZON}-Day Direction)')
        plt.xlabel('Importance Score (Mean Decrease in Impurity)')
        # ... rest of plotting ...
        plt.show()
        print("Top 15 Features (Random Forest):\n", rf_series.head(15))
    # ... error handling ...
```



Pasted image 20250419171146.png

Pasted image 20250419171146.png

The bar chart visualizes these model-specific importance scores. Higher scores mean the Random Forest relied more heavily on that feature to make its predictions on the training data.

How to Use the Script

1. **Install Prerequisites:** Ensure Python and all required libraries (including TA-Lib C library and Python wrapper) are installed.
2. **Save the Code:** Save the entire Python script to a file (e.g., `indicator_analysis.py`).
3. **Configure:** Open the script and modify the variables in the "Configuration" section (`TICKER`, `START_DATE`, etc.) for your desired analysis.
4. **Run:** Execute the script from your terminal: `python indicator_analysis.py`
5. **Observe Output:** The script will print messages about data loading, indicator calculation, and data shapes. Finally, it will display two plots: one for Mutual Information scores and one for Random Forest feature importances. It will also print the top 15 features according to each method.

Interpreting the Results

- **Focus on Top Features:** Look at the features consistently appearing near the top of *both* bar charts. These are likely the most robust candidates for having predictive power based on the historical data analyzed.
- **Mutual Information (MI Plot):** This plot shows the *statistical dependency* (linear and non-linear) between each indicator and the next day's direction. Higher MI suggests the indicator contains more information about the target. It's model-agnostic.
- **Random Forest Importance (RFI Plot):** This plot shows how *useful* each feature was *to the Random Forest model* in classifying the training data. Features that help create purer nodes (better splits) in the decision trees get higher scores.
- **Compare Plots:** Sometimes features rank differently. A feature might have high MI but lower RFI if its information is redundant with other features the RF model preferred, or if its relationship is complex in a way the RF didn't fully exploit with the chosen hyperparameters. Features high on both lists are generally the most interesting.
- **Magnitude Matters:** Pay attention not just to the rank but also the *value* of the scores. Is there a clear drop-off after the top few features, or are many features similarly important?

Limitations and Critical Next Steps

- **HISTORICAL DATA ONLY:** This analysis reveals correlations and importances based *purely on past data*. **There is absolutely no guarantee these relationships will hold in the future.** Markets evolve.

- **THIS IS NOT A TRADING STRATEGY:** This script performs **feature analysis ONLY**. It does not generate buy/sell signals, manage risk, or execute trades. Using this analysis directly for trading is highly discouraged and likely unprofitable.
- **Overfitting Risk:** Importance calculated on the training set can be overly optimistic. The model might have learned noise specific to that data period.
- **Next Steps (Crucial):**
  1. **Feature Selection:** Choose a subset of the top-performing indicators identified here.
  2. **Model Building:** Train various *predictive* models (Logistic Regression, SVM, Gradient Boosting, Neural Networks, etc.) using the selected features on the training data.
  3. **Validation:** Tune model hyperparameters using the validation set (or preferably, Time-Series Cross-Validation like `TimeSeriesSplit` within the training data).
  4. **Rigorous Backtesting:** Test the *complete strategy* (model predictions + entry/exit rules + risk management) on the **hold-out test set**. Analyze performance metrics critically (Sharpe Ratio, Max Drawdown, Profit Factor, etc.).
  5. **Risk Management:** Implement robust risk management rules (stop-losses, position sizing).

Conclusion

This Python script offers a sophisticated starting point for quantitatively assessing which technical indicators might hold predictive value for Bitcoin's short-term price direction. By using both Mutual Information and Random Forest importance, it provides two valuable perspectives. However, remember that this is an **analytical tool for research**, not a trading system. The insights gained must be rigorously validated through proper model building and backtesting before ever considering real-world application.