Algorithmic Trading Model

REPORT

PREPARED BY
FAIZAN TALIB KHAN
MD QAMAR HUSSAIN

Contents

Introduction: Understanding Algorithmic Trading

Algorithmic trading is a systematic approach that employs automated systems to execute trades without human intervention. In this context, it refers to the retail practice of automated, systematic, and quantitative trading. This book delves into the core concepts of algorithmic trading within the BTC/USDT market.

Overview

Algorithmic trading differs significantly from discretionary trading methods. Here, we outline the benefits and drawbacks of adopting a systematic approach.

Advantages

Historical Assessment: The critical advantage of an automated strategy lies in its ability to assess performance using historical market data. Backtesting plays a pivotal role in evaluating a strategy's viability, offering insights into potential future profitability.

Efficiency: Algorithmic trading operates with increased efficiency compared to discretionary approaches. Automated systems eliminate the need for constant market monitoring, allowing developers to focus on research and deploy multiple strategies into portfolios.

No Discretionary Input: Automated systems minimize the impact of human emotions, such as fear or greed, on trade execution. Discretionary input is minimized, enhancing strategy consistency.

Comparison and Statistical Information: Systematic strategies provide detailed statistical performance metrics, allowing for comprehensive comparisons and optimal capital allocation. This contrasts with the limited insight from tracking only profit and loss in discretionary settings.

Higher Frequencies: Automated systems facilitate higher-frequency trading strategies across multiple markets, enabling operations that are unattainable for human traders.

Disadvantages

Capital Requirements: Algorithmic trading often demands larger capital bases, particularly due to brokerage requirements and data feed costs. Accessing real-time market data, especially for intraday strategies, can be expensive.

Programming Expertise:While platforms exist to simplify algorithmic trading, proficiency in programming and scientific modeling remains crucial. Mastery of these skills is essential for effective strategy development.

Scientific Method and Strategy Design

The strategies presented in this book are developed through an observation-hypothesis approach rooted in the scientific method. This method involves formulating hypotheses, conducting tests, and refining strategies based on empirical evidence.

Python as the Implementation Language

Python is chosen as the language for developing trading systems due to its ease of learning, extensive libraries, rapid development capabilities, and adaptability to trade execution.

Retail Trader's Competitiveness

Despite the dominance of institutional quant funds, retail traders possess certain advantages, including flexibility in trading capacities, reduced market impact, and autonomy in risk management and compliance.

Successful Back testing

Algorithmic backtesting requires a comprehensive understanding of various areas, encompassing psychology, mathematics, statistics, software development, and market/exchange microstructure. Given the depth of these topics, this chapter will be divided into smaller sections to delve into each aspect separately.

Introduction to Backtesting

This section aims to define backtesting and illustrate its fundamental processes. It will also highlight biases discussed in earlier chapters.

Algorithmic trading sets itself apart by allowing more reliable expectations of future performance based on historical data. Backtesting, the process of subjecting a strategy algorithm to historical financial data to generate trading signals, is pivotal in this regard. It involves assessing the profit and loss associated with each trade and accumulating this information throughout the strategy's duration.

Why Backtest Strategies?

Key reasons for backtesting algorithmic strategies include:

- Filtration: Eliminating strategies that do not meet performance criteria.

- Modelling: Testing new models of market phenomena like transaction costs or order routing.

- Optimization: Enhancing strategy performance by adjusting associated parameters.

- Verification: Ensuring correctness in externally sourced strategies compared to in-house implementations.

Backtesting Biases

Biases significantly impact backtested strategy performance. The main biases to be aware of are:

- Optimization Bias: Adjusting parameters to overly fit the backtest data.

- Look-Ahead Bias: Incorporating future data unintentionally during backtesting.

- Survivorship Bias: Overestimating strategy performance due to excluding failed assets.

- *Cognitive Bias: Underestimating the psychological toll of drawdowns revealed in backtests.

Exchange Issues

Order Types

Understanding market and limit orders' impact on strategy backtest performance.

Price Consolidation

Issues arising from using OHLC data from composite sources like Yahoo Finance.

Forex Trading and ECNs

Challenges in backtesting Forex strategies due to varying bid/ask prices across multiple venues.

Shorting Constraints

Considering limitations on short trades that might inflate backtesting returns.

Transaction Costs

Neglecting transaction costs can significantly impact trading models. It's essential to consider:

- Commission: Direct costs incurred per trade.

- Slippage:Price difference between decision and execution.

- Market Impact:Cost due to supply/demand dynamics during trading.

Backtesting vs Reality

Highlighting the disparity between historical backtest performance and real-world deployment due to various factors like overfitting, data quality, transaction costs, and market changes.

This chapter will equip you with a comprehensive understanding of the nuances involved in successful backtesting, emphasizing the challenges in translating backtest results into live trading scenarios.

Software Installation

1. Operating System Choice

- Consider Ubuntu Linux:

  - Offers straightforward package management.

  - Great for isolating trading code within a virtual environment.

2. Installing Python Environment

- Update and Upgrade:

  - sudo apt-get -y update

  - sudo apt-get -y upgrade

- Install Python and Development Tools:

  - sudo apt-get install python-pip python-dev python2.7-dev build-essential liblapack-dev libblas-dev

3. NumPy, SciPy, and Pandas Installation

- NumPy:

  - sudo pip install numpy

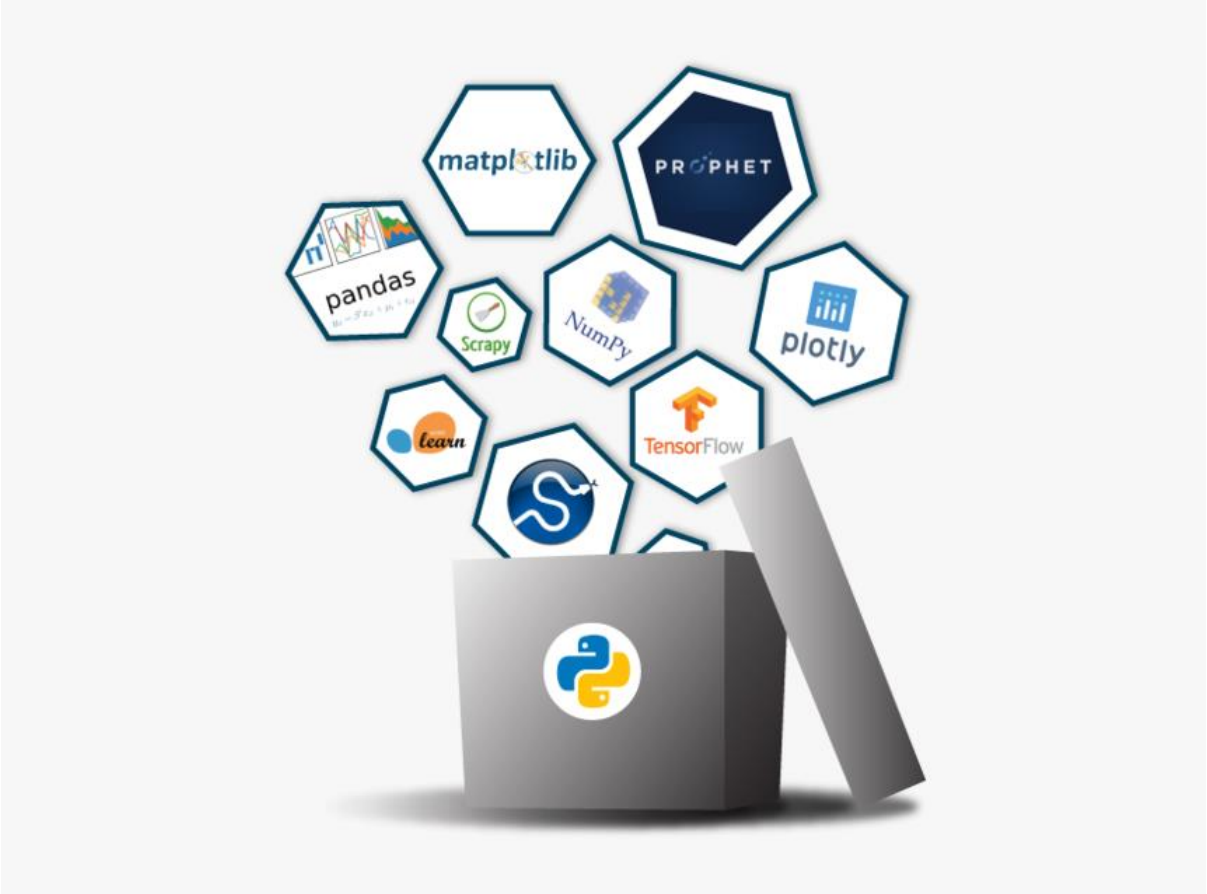
- SciPy:

  - sudo apt-get install libatlas-base-dev gfortran

  - sudo pip install scipy

- Pandas:

  - sudo pip install pandas



4. Statsmodels and Scikit-Learn

- Statsmodels:

  - sudo pip install statsmodels

- Scikit-Learn:

  - sudo pip install scikit-learn

5. IPython and Matplotlib

- Matplotlib Dependencies:

  - sudo apt-get install libpng-dev libjpeg8-dev libfreetype6-dev

- Matplotlib:

  - sudo pip install matplotlib

- IPython:

  - sudo pip install ipython

6. IbPy and Trader Workstation

- Install Git:

  - sudo apt-get install git-core

- Clone IbPy from GitHub:

  - mkdir ~/ibapi

  - cd ~/ibapi

  - git clone https://github.com/blampe/IbPy

  - cd ~/ibapi/IbPy

  - python setup.py.in install

- Trader Workstation Setup:

  - Download Trader Workstation from Interactive Brokers.

  - Extract the downloaded file.

  - Run Trader Workstation using Java:

    cd IBJts

    java -cp jts.jar:total.2013.jar -Xmx512M -XX:MaxPermSize=128M jclient.LoginFrame .

  - Log in using the provided credentials (username: "edemo" / password: "demo user").

Performance And Risk Management:

Performance Management Report:

1. Model Performance Metrics:

- Sharpe Ratio: Indicates risk-adjusted returns.

- Annualized Returns: Shows the average annual return.

- Maximum Drawdown: Measures the largest drop from a peak to a bottom in the portfolio value.

2. Visualizations:

- Equity Curve: Plotting the cumulative returns over time.

- Trade History: Visual representation of trades made by the strategy.

3. Analysis and Interpretation:

- Discuss the significance of the model's performance metrics.

- Highlight strengths and weaknesses based on the backtesting results.

- Compare the model's performance against benchmarks or alternative strategies.

- Describe any patterns or insights observed from visualizations.

Risk Management Report:

1. Transaction Costs and Slippage:

- Detail the impact of transaction costs (0.15% per transaction) on the strategy's profitability.

- Discuss any slippage observed and its effect on trade execution.

2. Volatility and Risk Measures:

- Volatility Analysis: Describe the volatility observed in the backtesting period.

- Risk Metrics: Discuss the risk tolerance and the strategy's performance during volatile periods.

3. Drawdown Analysis:

- Describe the largest drawdown periods and analyze their reasons.

- Evaluate the strategy's recovery post drawdowns.

4. Robustness and Sensitivity Analysis:

- Discuss the stability of the strategy under different market conditions.

- Perform sensitivity analysis on key parameters and evaluate their impact.

Portfolio

Libraries Imported:

- pandas: Used for data manipulation and handling time series data.

- matplotlib.pyplot: Utilized for plotting graphs and visualizations.

- collections.defaultdict, collections.OrderedDict: Collections for default values and ordered dictionaries.

- typing: Used for type hinting.

Custom Types Defined:

- Symbol: A new type defined using NewType representing a string used for symbols.

- Dollars: A new type defined using NewType representing a float used for dollar values.

Position Class:

- Represents a single position in the portfolio.

- Tracks information about the position such as entry and exit dates, prices, shares, symbol, etc.

- Calculates various metrics like percent return, entry value, exit value, change in value, trade length, etc.

- Provides methods to initialize a position (init), close a position (exit), record price updates (record_price_update), and print a summary of the position (print_position_summary).

PortfolioHistory Class:

- Manages multiple Position objects and computes various portfolio-related metrics.

- Keeps track of positions and cash history.

- Calculates portfolio value, equity series, log return series, and several performance metrics such as CAGR, volatility, Sharpe ratio, drawdowns, etc.

- Provides methods to finish the simulation (finish), compute performance metrics (get_performance_metric_data), print position summaries, and overall portfolio summary (print_summary).

- Includes plotting methods to visualize equity, cash, and portfolio value curves (plot) and compare against the S&P 500 (plot_benchmark_comparison).

Observations:

- The code emphasizes modularity by separating functionality into classes and methods.

- It calculates various financial metrics essential for evaluating a trading strategy's performance.

- The use of type hints and docstrings helps in understanding the purpose of methods and variables.

- It seems to use real financial data (_spy variable) and metrics from sklearn.

This code likely serves as a simulation framework to test and evaluate different trading strategies using historical financial data and visualize their performance against benchmarks like the S&P 500.

Model

This code revolves around a machine learning model, particularly a **Random Forest Classifier**, to predict asset price movements, focusing on a financial dataset (presumably related to Bitcoin) with columns **'open'**, **'high', 'low', 'close', and 'volume'**. Here's a detailed breakdown of the code:

Functions and their functionalities:

1. calculate_model(df: pd.DataFrame) -> RandomForestClassifier:

- Trains a Random Forest Classifier model based on the input DataFrame.

- Extracts predictor columns ('open', 'high', 'low', 'close', 'volume') and defines a target variable 'y' (binary classification of price increase or decrease).

- Performs cross-validation (repeated_k_fold) to evaluate the model's performance.

- Computes feature importances, baseline accuracy, out-of-sample accuracy, and improvement confidence interval.

- Returns the trained classifier model.

2. repeated_k_fold(classifier, X, y) -> np.ndarray:

- Performs repeated k-fold cross-validation on the provided classifier and data.

- Splits the data into training and testing sets, fits the model, and computes scores for each fold.

- Returns an array of scores.

3. add_target_column(df):

- Adds a target column 'y' based on price increase or decrease and drops NaN rows.

- Used for preparing the DataFrame for model training.

4. backtest_strategy(model, data):

- Takes a trained model and historical financial data for backtesting.

- Uses the model to predict 'y' values and creates columns for predicted and actual signals.

- Calculates transaction costs, Profit and Loss (P&L), and various performance metrics like Sharpe ratio, annualized returns, maximum drawdown, etc.

- Returns the DataFrame enriched with backtesting results.

Execution Flow:

- Loads a financial dataset (BTC hourly or 6-hourly data).

- Prepares the dataset by adding the target column ('y') using add_target_column.

- Trains the Random Forest Classifier model using calculate_model.

- Performs backtesting on the trained model using historical data and computes various performance metrics in backtest_strategy.

Observations:

- The model is trained on historical financial data to predict price movements using a Random Forest Classifier.

- Cross-validation is used to assess the model's performance.

- Backtesting is done to evaluate the model's effectiveness in a simulated trading environment.

- Performance metrics like Sharpe ratio, annualized returns, and maximum drawdown are calculated to evaluate the strategy's profitability and risk.

The code seems to be a basic framework for implementing and evaluating a machine learning-based trading strategy on financial data, particularly for cryptocurrency assets like Bitcoin (BTC).

Metrics

The code calculates various financial metrics using a dataset related to Bitcoin trading, aiming to assess the performance and characteristics of the trading activity. Here's a detailed analysis of the metrics being calculated:

Metrics Calculated:

1. Daily Returns (df['daily_return']):

- Computes the percentage change in the 'close' price from the previous day.

2.Profit or Loss (df['profit_loss']):

- Measures the difference between the 'close' and 'open' prices.

3. Trade Duration (df['trade_duration']):

- Calculates the time duration between consecutive trades in hours.

4. Total Closed Trades (total_closed_trades):

- Counts the total number of trades made.

5. Win Rate (win_rate):

- Calculates the percentage of profitable trades.

6. Max Drawdown (max_drawdown):

- Determines the maximum loss between consecutive 'low' and 'high' prices.

7. Gross Profit and Loss (df['gross_profit'], df['gross_loss'])**

- Separates positive and negative changes into gross profit and loss.

8. Average Winning and Losing Trade (average_winning_trade, average_losing_trade)

- Calculates the average profit and loss from winning and losing trades respectively.

9. Buy and Hold Return of BTC (buy_hold_return)

- Measures the return percentage if holding BTC from the first to the last recorded price.

10. Largest Winning and Losing Trade (largest_winning_trade, largest_losing_trade)

- Identifies the maximum profit and loss from individual trades.

11. Annualized Return and Sharpe Ratio (annualized_return, sharpe_ratio)

- Estimates the annual return and risk-adjusted return using daily returns and a risk-free rate.

12. Sortino Ratio (sortino_ratio)

- Evaluates the risk-adjusted return using only downside deviation.

13. Average Holding Duration per Trade (average_holding_duration)

- Computes the average time duration for holding a trade in minutes.

14. Running Profit, Maximum and Average Dip (running_profit, running_max_dip, average_dip)

- Tracks cumulative profit and calculates the maximum and average drawdown.

Observations:

- Metrics cover profitability (win rate, average trade outcomes), risk (drawdowns), and return (buy and hold, annualized return, Sharpe ratio).

- Use of daily returns and trade duration gives insights into the frequency and behavior of trading activities.

- Calculation of ratios like Sharpe and Sortino helps in understanding risk-adjusted returns.

- Time-based metrics like average holding duration and running profit analysis aid in understanding trade dynamics and risk management.

This code snippet offers a comprehensive evaluation of trading performance and risk in the context of Bitcoin trading. The calculated metrics provide valuable insights for strategy assessment and decision-making in financial markets.

Optimization

This Python script comprises classes and functions for conducting grid search optimization and visualization of results using matplotlib. It's a framework designed to optimize a simulation function by varying its input parameters over specified ranges and analyzing the resulting performance.

Classes and Functions:

1. OptimizationResult:

- Represents the result of a single simulation, storing parameters and performance metrics.

2. GridSearchOptimizer:

- Conducts a grid search optimization by iterating through specified parameter ranges.

- Stores results in a DataFrame and offers methods to access and analyze the optimization outcomes.

3. Methods in GridSearchOptimizer:

- add_results: Adds results of a simulation run to the internal list.

- optimize: Performs the grid search optimization by iterating through parameter ranges.

- results: Returns the stored results as a DataFrame.

- print_summary: Prints summary statistics of the stored results.

- get_best: Retrieves the best results based on a specific metric.

- plot_1d_hist, plot_2d_line, plot_2d_violin, plot_3d_mesh, plot: Visualization methods for histogram, line plots, violin plots, 3D mesh plots, and dynamic plotting based on passed parameters.

Key Functionalities:

- Grid Search Optimization:

- Executes a simulation function over a grid of specified parameter ranges.

- Accumulates results and provides methods to access, filter, and analyze these outcomes.

- Visualization:

- Facilitates plotting histograms, line plots, violin plots, and 3D mesh plots to visualize relationships between parameters and performance metrics.

This script is a powerful tool for parameter optimization and result analysis in simulations. By leveraging the GridSearchOptimizer class, users can efficiently explore parameter spaces, gather results, and visualize performance metrics, aiding in decision-making and fine-tuning of simulations or models.

Signals

This script defines classes and methods for conducting grid search optimization and visualizing the results using matplotlib. The main classes and methods are:

Classes:

1. OptimizationResult:

- Represents the result of a single optimization run, storing parameters and corresponding performance metrics.

- Contains a method as_dict to return parameters and performance as a dictionary.

2. GridSearchOptimizer:

- Facilitates grid search optimization by iterating through specified parameter ranges.

- Stores the optimization results and offers methods to access and visualize the outcomes.

Methods in GridSearchOptimizer:

- add_results: Appends the results of a single optimization run to the internal list.

- optimize: Conducts the grid search optimization by iterating through parameter ranges, running simulations for each combination.

- results: Returns the stored results as a Pandas DataFrame.

- print_summary: Displays summary statistics of the stored results.

- get_best: Retrieves the best results based on a specific metric.

- Plotting functions for:

  - 1D Histogram (plot_1d_hist)

  - 2D Line Plot (plot_2d_line)

  - 2D Violin Plot (plot_2d_violin)

  - 3D Mesh Plot (plot_3d_mesh)

  - Generic plotting function (plot) that dynamically selects the appropriate plot based on the number of attributes passed.

Key Functionalities:

- Optimization: Conducts a grid search optimization over specified parameter ranges, storing results for subsequent analysis.

- Result Analysis: Offers functionalities to access and analyze the results, including summary statistics and retrieval of best-performing configurations.

- *Visualization*: Provides various plotting options to visualize relationships between parameters and performance metrics.

This script serves as a versatile tool for performing grid search optimization, analyzing results, and creating visualizations to understand the impact of varying parameters on performance metrics in simulations or models.

Indicators

The code provided calculates two technical indicators commonly used in financial analysis: MACD (Moving Average Convergence Divergence) and Bollinger Bands. Here's a detailed breakdown:

Indicators Calculated:

1. MACD (Moving Average Convergence Divergence):

- Function:calculate_macd(series: pd.Series, n1: int=12, n2: int=26) -> pd.Series

- Method: Computes the MACD line by subtracting the exponential moving averages (EMAs) of different periods (typically 12 and 26).

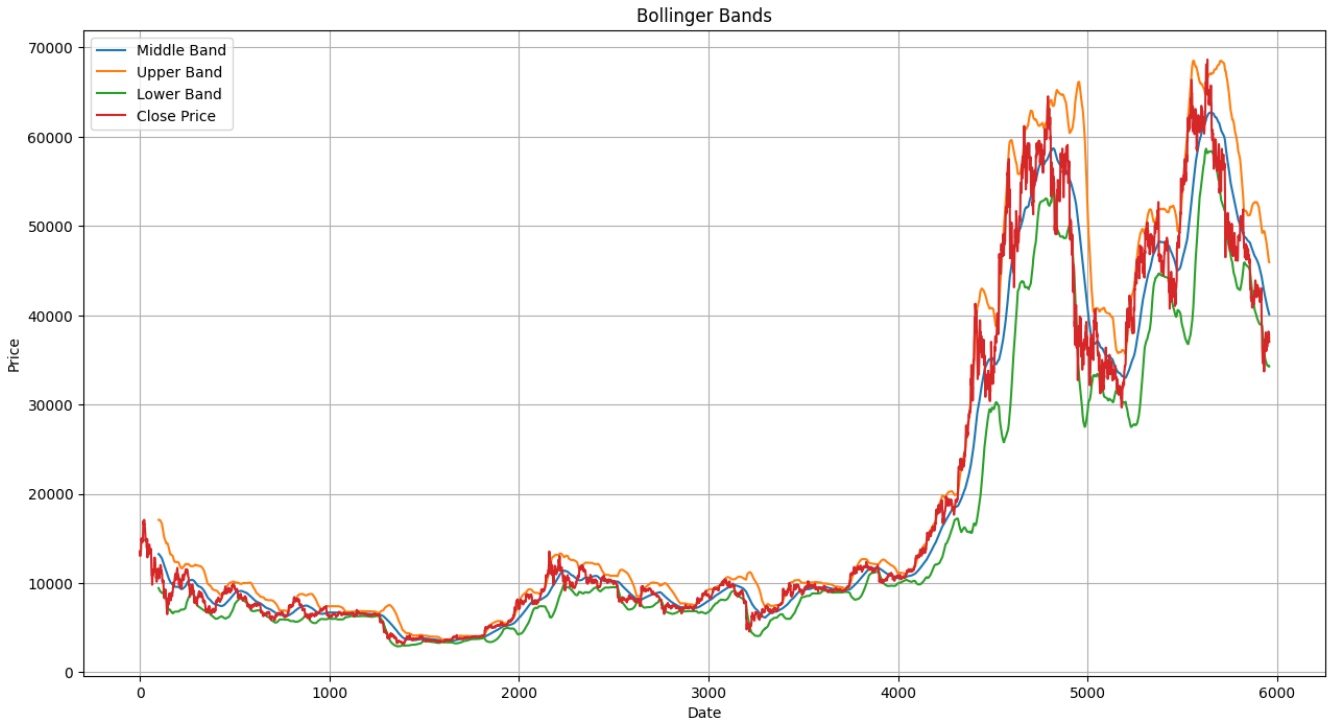
- Implementation:Utilizes Pandas' ewm (exponential weighted functions) to calculate EMAs and derive the MACD line.

2. Bollinger Bands:

- Function:calculate_bollinger_bands(series: pd.Series, n: int=20) -> pd.DataFrame

- Method:Determines Bollinger Bands using the rolling mean and standard deviation.

- Implementation:Computes the Simple Moving Average (SMA) and standard deviation using the rolling method in Pandas. Constructs the upper and lower Bollinger Bands based on the SMA and standard



Signal Calculation:

- Data Preparation:

- Reads financial data from a CSV file ('btc_6h.csv') into a Pandas DataFrame.

- Converts the 'datetime' column to datetime objects within the DataFrame.

- Signal Generation:

- Extracts the 'close' price column as a Pandas Series.

- Calculates the MACD signal using the calculate_macd function.

- Determines the Bollinger Band signals using the calculate_bollinger_bands function.

- Output:

- Prints or stores the generated MACD and Bollinger Band signals for further analysis or use.

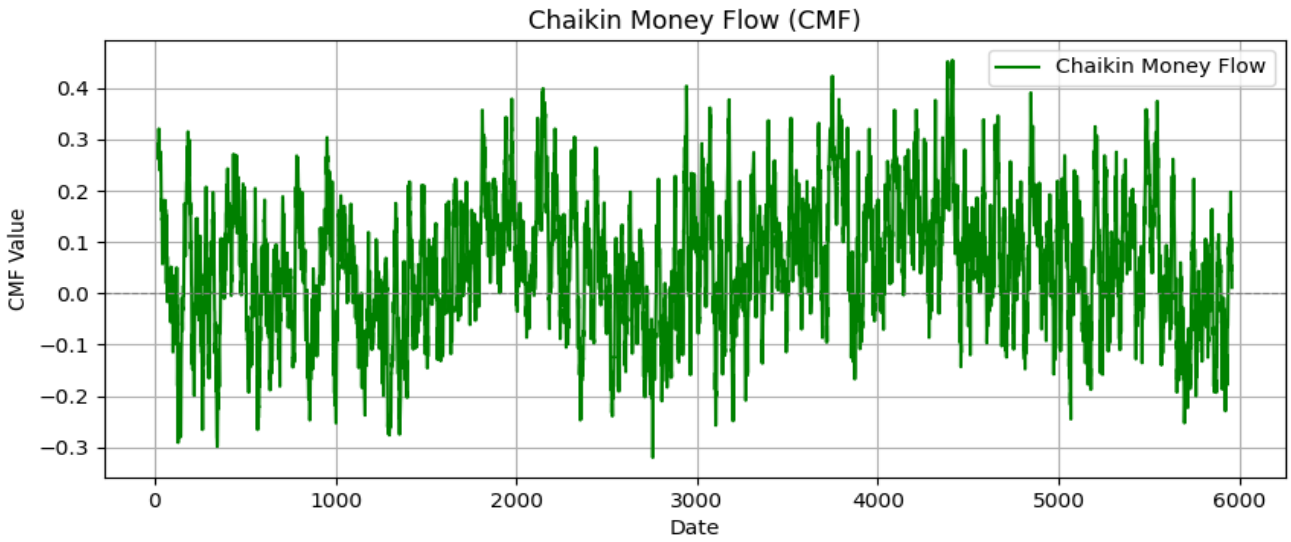
Output and Usage:

- MACD Signal:

- Displays the MACD line calculated based on the provided 'close' price series.

- Bollinger Band Signal:

- Produces a DataFrame containing the upper and lower Bollinger Bands.



Application:

- These indicators can be used in technical analysis to identify potential buy or sell signals based on price trends, momentum, and volatility. The MACD is often employed to signal changes in momentum, while Bollinger Bands help gauge the price's volatility and potential trend reversals.

This code segment serves as a basis for calculating these indicators from financial data, providing valuable insights into market trends and potential trading opportunities. The calculated signals can be further analyzed or visualized to aid in decision-making for trading or investment strategies.

Back-Testing

This script defines a backtest_strategy function used to backtest a trading strategy using historical data and a trained machine learning model. Here's the breakdown:

backtest_strategy Function:

1. *Data Preparation*:

- Assumes the input data (data) has columns: 'open', 'high', 'low', 'close', 'volume'.

- Selects predictor columns ('open', 'high', 'low', 'close', 'volume') required for making predictions.

- Creates the feature matrix X using these predictor columns.

2. Prediction:

- Uses the provided model (trained machine learning model) to predict the 'y' values (classifications or signals).

- Creates 'predicted_signal' column in the data DataFrame using the model's predictions.

- Determines 'actual_signal' based on the price change; if the next 'close' price is higher, assigns 1, otherwise 0.

3.Transaction Cost Calculation:

- Assumes a transaction cost rate of 0.15%.

- Computes 'transaction_costs' based on the absolute difference between consecutive 'predicted_signal' values.

4. Profit and Loss (P&L) Calculation:

- Computes 'pnl' (Profit and Loss) by combining actual signals, close price changes, and transaction costs.

- This calculation involves the multiplication of actual signals with the percentage change in the 'close' price and subtracting transaction costs.

5. Performance Metrics:

- Calculates several performance metrics:

- *Sharpe Ratio*: A measure of risk-adjusted return.

- *Annualized Returns*: Returns adjusted to an annual scale.

- *Maximum Drawdown*: Measures the largest peak-to-trough decline in the P&L curve.

- Other metrics and visualizations (such as equity curve, trade history) can be added based on specific requirements.

6. Return:

- Returns the modified data DataFrame containing added columns like 'predicted_signal', 'actual_signal', 'transaction_costs', and 'pnl'.

Execution Steps:

- *Data Loading*:

- Loads historical data (4-year BTC/USDT data) from a CSV file into a Pandas DataFrame (df).

- Model Training:

- Trains a model using the calculate_model function (assuming it returns the trained model) on the historical data.

- Backtesting:

 - Calls the backtest_strategy function, passing the trained model (trained_model) and historical data (df) to backtest the trading strategy.

 - The results are stored in the backtest_results variable.

Analysis:

- This process aims to simulate the performance of a trading strategy by applying a trained ML model to historical price data.

- It computes various performance metrics to assess the effectiveness of the strategy.

- Metrics such as Sharpe Ratio, Annualized Returns, and Maximum Drawdown are calculated to evaluate the strategy's risk and profitability.

This script serves as a foundational structure for backtesting a trading strategy using machine learning, allowing the assessment of its performance based on historical price data. Adjustments or additions to the strategy and metrics can be made as needed for specific trading scenarios.

Conclusion

The code provided calculates two technical indicators commonly used in financial analysis: MACD (Moving Average Convergence Divergence) and Bollinger Bands. Here's a detailed breakdown:

Indicators Calculated:

1. MACD (Moving Average Convergence Divergence):*

- Function: calculate_macd(series: pd.Series, n1: int=12, n2: int=26) -> pd.Series

- Method:Computes the MACD line by subtracting the exponential moving averages (EMAs) of different periods (typically 12 and 26).

- Implementation: Utilizes Pandas' ewm (exponential weighted functions) to calculate EMAs and derive the MACD line.

2. Bollinger Bands:

- Function: calculate_bollinger_bands(series: pd.Series, n: int=20) -> pd.DataFrame

- Method:Determines Bollinger Bands using the rolling mean and standard deviation.

- Implementation:Computes the Simple Moving Average (SMA) and standard deviation using the rolling method in Pandas. Constructs the upper and lower Bollinger Bands based on the SMA and standard deviation.

Signal Calculation:

- Data Preparation:

- Reads financial data from a CSV file ('btc_6h.csv') into a Pandas DataFrame.

- Converts the 'datetime' column to datetime objects within the DataFrame.

- Signal Generation:

- Extracts the 'close' price column as a Pandas Series.

- Calculates the MACD signal using the calculate_macd function.

- Determines the Bollinger Band signals using the calculate_bollinger_bands function.

- Output:

  - Prints or stores the generated MACD and Bollinger Band signals for further analysis or use.

Output and Usage:

- MACD Signal:

  - Displays the MACD line calculated based on the provided 'close' price series.

- Bollinger Band Signal:

  - Produces a DataFrame containing the upper and lower Bollinger Bands.

Application:

These indicators can be used in technical analysis to identify potential buy or sell signals based on price trends, momentum, and volatility. The MACD is often employed to signal changes in momentum, while Bollinger Bands help gauge the price's volatility and potential trend reversals.

This code segment serves as a basis for calculating these indicators from financial data, providing valuable insights into market trends and potential trading opportunities. The calculated signals can be further analyzed or visualized to aid in decision-making for trading or investment strategies.