QF 635 Market Microstructure and Algorithmic Trading

BTC Future API Strategy

Group Member:

Yu Lingfeng

Liu Yisi

Liu Jing

Lee Kaishan

Wang Wenjie

Zhang Keke

Contents

1. Abstract:

This paper provides an overview of a comprehensive API trading system through test.net Binance gateway, which includes functionalities such as order book management, data recording, credential management, model training, real-time trading, and retrieval of trade history records. Regarding model training, the system employs various statistical methods and regression models for financial data analysis and development of trading strategies.

To begin with, statistical methods like thresholding techniques are utilized to identify and filter outliers in the dataset, enhancing data quality and analysis accuracy.

Subsequently, for model training, the system selects linear regression and LASSO regression models. During the training process, these models use historical financial data to calculate trading signals, such as those based on predefined periods and thresholds, to generate trading signals. Additionally, the system evaluates the profitability of trades signal using threshold as some multiple of current order book spread. After model training, the system conducts simulated trading to evaluate the performance of developed trading strategies. During simulated trading, the system executes trades based on signals generated by the models and tracks the profitability of the trades, providing insights into strategy effectiveness in a controlled environment.

Finally, the system performs online testing using real-time data to validate the robustness of the models. During online testing, the models are applied to real-time market data, and their trading performance is analyzed in real-time. This step helps identify differences between simulated and actual trading outcomes and ensures the reliability and practicality of the developed strategies.

2. Gateway Setting Up

2.1 Class OrderBook

Obtaining Depth Messages: Periodically retrieve depth messages by calling the exchange's API. Depth messages typically include information about buy and sell limit orders on the current market, along with relevant prices and quantities.

Parsing Depth Messages: Utilize the parse function to decode the retrieved depth messages. This function converts JSON-formatted depth messages into data structures that the program can handle internally, specifically OrderBook objects. During the parsing process, extract the prices and quantities of buy and sell orders separately, and organize them into arrays of Tier objects.

Constructing Order Book Objects: Create an OrderBook object using the parsed data. This object will contain all the information from the depth messages, including buy and sell orders, as well as additional details such as the message timestamp.

Extracting Desired Information: Retrieve the necessary information from the OrderBook object. In this program, the main focus is on obtaining the best bid and best ask prices, along with other market depth-related metrics such as the mid-price and spread.

2.2 Per-Second Data Recording

Parsing JSON data from API responses and converting it into a manageable data structure: During this process, the data obtained from API responses is initially represented in JSON format. JSON is a common data exchange format, but it's typically in textual form, which is not convenient for direct data processing and analysis. Therefore, it's necessary to parse JSON data into data structures in Python for subsequent processing and analysis. This typically involves converting JSON objects into dictionaries or lists in Python, allowing easy access to the data.

Parsing information about buy and sell orders' prices and quantities: When parsing JSON data from API responses, the focus is primarily on information regarding buy and sell orders' prices and quantities. This information is usually found in the "bids" (buy orders) and "asks" (sell orders) fields of the order book. For each price level, there is typically a corresponding quantity (size), indicating the number of orders at that price level.

Key information in the order book:

- Timestamp: Represents the timestamp of the data, used to record the time of data collection.
- Mid-price: Half of the difference between the buy and sell prices, typically used to gauge the overall market trend.
- Spread: The difference between the sell and buy prices, used to measure market liquidity.
- Best bid and ask prices: Represents the best bid and ask prices in the current order book, i.e., the highest bid price and the lowest ask price. Best N tiers of limit orders on both bid side and ask side.
- Best N-tiers: In our strategy, we specifically extend the Class OrderBook to capture 10 best tiers of bids and asks.
- Converting the recorded data into a DataFrame: When the number of entries in the list reaches a certain threshold (batchN), the recorded data is converted into a DataFrame and saved as a CSV file. Here, batchN represents the threshold for the number of entries recorded each time. Once this threshold is reached, the current recorded data is converted into a DataFrame and saved, and then the process starts again to record new data.

2.3 Credentials for live trading

We obtain credentials by loading sensitive information stored in an environment variable file, such as API keys. Firstly, we determine the path of the current working directory and then combine it with a relative path to locate the .env file containing the credential information. Subsequently, we load the environment variables from the .env file to acquire API keys and other necessary details. This approach adheres to best practices by separating sensitive information from the code, enhancing security, and facilitating the management and maintenance of credentials across various environments.

3. Model Training

We employ statistical methods and regression models to analyze financial data and develop trading strategies. Initially, we utilize thresholding techniques based on statistical information in the dataset, aiming to purify the data for further analysis. Subsequently, we select linear regression and LASSO regression models for model training. During the training process, these models calculate trading signals using predefined periods and thresholds and evaluate the profitability of trades. Following model training, we conduct simulated trading to assess the effectiveness of the developed strategies and determine their profitability. Finally, we perform online testing of the models using real-time data and analyze their trading performance.

3.1 Methodology

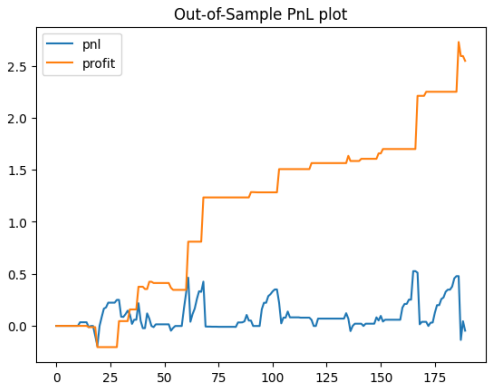
Data Preprocessing: We begin by preprocessing the financial data to remove outliers using statistical methods such as thresholding. This step aims to enhance the quality of the dataset by eliminating erroneous or irrelevant data points.

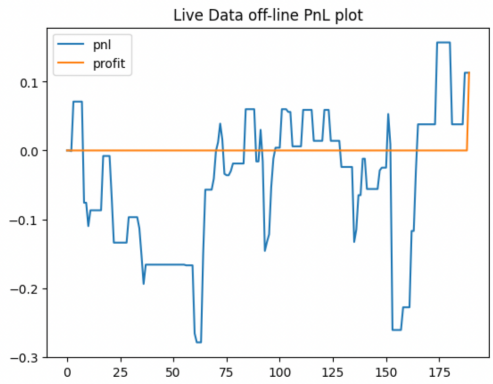
Model Selection: For model training, we choose two regression models: linear regression and LASSO regression. These models are commonly used in financial analysis for their simplicity and interpretability. Linear regression provides a straightforward approach to modeling the relationship between variables, while LASSO regression offers a regularization technique that helps prevent overfitting.

Model Training: During the training process, the selected regression models are trained using historical financial data. The models calculate trading signals based on predefined periods and thresholds, which serve as indicators for initiating trades. Additionally, we evaluate the profitability of trades using metrics such as mean squared error and R-squared.

Simulated Trading: After model training, we conduct simulated trading to assess the performance of the developed trading strategies. Simulated trading involves executing trades based on the trading signals generated by the models and tracking the resulting profitability. This step provides insights into the effectiveness of the strategies in a controlled environment.

Online Testing: To validate the robustness of the models, we perform online testing using real-time data. During online testing, the models are applied to live market data, and their trading performance is analyzed in real-time. This step helps identify any discrepancies between simulated and actual trading outcomes and ensures the reliability of the developed strategies.





4. Live Trading

4.1 Recent Trade

To retrieve the latest trading data, we take in three parameters: an API key, the API key's secret key, and the currency pair. Firstly, it constructs the currency pair into a parameter dictionary and converts it into a query string. Next, it uses the secret key to hash the query string, generating a digital signature. Subsequently, this signature and the query string are added to the URL to form an API request. Following that, the API key is used as a request header to send a GET request to the server for fetching the latest trading data. Ultimately, the function returns the obtained trading data in JSON format.

4.2 Send Market Order and Limit Order

To send trading orders to the exchange, we have defined two functions: one for sending market orders and one for sending limit orders. Market orders are utilized for stop-loss, while limit orders are employed for profit-taking. In the function for sending market orders, it first computes the current timestamp and constructs an order parameter dictionary based on the passed parameters. Then, it converts the parameter dictionary into a query string and uses the HMAC algorithm to hash it, generating a digital signature. Subsequently, it appends the digital signature and the query string to the URL to form an API request. Following this, it uses a POST request to send this API request to the exchange. Finally, it retrieves the order ID from the response and returns it.

The function for sending limit orders is similar to the one for sending market orders, with the distinction that the order type is a limit order, hence requiring parameters for price and time in force. The logic for other parts is similar to that of sending market orders, involving the construction of order parameters, generation of signatures, sending requests, and returning order IDs.

This design is intended to issue buy or sell instructions to the exchange for trading operations. The use of encrypted signatures ensures the integrity and security of the requests, while the timestamp

prevents replay attacks. Finally, returning the order ID helps track the status and results of the transactions.

4.3 Get Open Positions

First, it sends a GET request to the Binance exchange to fetch the current server time. Then, it extracts the server timestamp from the response. To ensure the accuracy of the request timestamp, the function uses the obtained server timestamp directly without any adjustments.

Next, it constructs a parameter dictionary for the position query and converts it into a query string. It then generates a digital signature by hashing the query string using the private key. The digital signature and the query string are concatenated to form the URL for the API request.

The API key is added to the request headers to identify the request source as Binance. Subsequently, a GET request is sent to the API, and the response is parsed as JSON format.

Finally, it checks the response status code. If it's 200, indicating a successful request, it parses the response and returns the position information. If not, it prints an error message and returns None.

This design allows users to stay updated on their positions on the exchange in real-time, facilitating appropriate trading decisions. The use of encrypted signatures ensures the integrity and security of the requests, while the addition of the API key and the accurate retrieval of the timestamp ensure the effectiveness of the requests.

4.4 Get Open Orders and Cancel All Open Orders

First, we obtain the current timestamp and construct a parameter dictionary containing the currency pair and timestamp. Then, we convert the parameter dictionary into a query string and hash it to generate a digital signature. Next, we append the digital signature and the query string to the URL to form an API request. Afterwards, we send a GET request to retrieve all open orders. The function returns the response in JSON format, which includes information about all open orders. Following this, we send a DELETE request to cancel all open orders. In this function, we also include error handling to deal with potential request errors. If the request is successful (status code 200), it prints the message "All open orders cancelled successfully"; otherwise, it prints a failure message and returns the corresponding status code.

This design enables users to retrieve all open orders and cancel them. The use of encrypted signatures ensures the integrity and security of the requests, while the addition of the API key and the accurate retrieval of the timestamp ensure the effectiveness of the requests. The ability to cancel all orders aids in promptly adjusting trading strategies and risk management.

4.5 Stop Loss

Our goal is to close positions based on market or limit orders when the stop-loss condition is triggered. Still, limit order is preferred due to maker transaction charge is lower than the taker

transaction charge. First, we cancel all outstanding orders to ensure no interference from other orders before setting the stop-loss. Then, we retrieve the current position information. Within a while loop, the function attempts to set the stop-loss. It first fetches a snapshot of the order book for the trading pair and selects the best buy or sell price as the limit price based on the current position direction. Then, using step b), it sends a limit order to attempt to close the position. If the limit order fails to execute within a certain number of attempts (controlled by max_trial), the function attempts to close the position using a market order. In this case, the function iteratively cancels all outstanding orders and then sends a market order for immediate execution. The purpose of this function is to close positions as quickly as possible when the stop-loss condition is reached to control risk. Using both limit and market orders allows for flexibility to adapt to market conditions while ensuring order execution.

4.6 Trade History

First, it obtains the current timestamp and constructs a parameter dictionary containing the trading pair and timestamp. Then, it converts the parameter dictionary into a query string and hashes it to generate a digital signature. Next, it appends the digital signature and the query string to the URL to form an API request. When sending the GET request, it includes the API key as a request header to allow Binance to identify the request's source. Then, it retrieves the transaction history data in JSON format from the response.

The purpose is to allow users to retrieve transaction history records for specific trading pairs. It uses encrypted signatures to ensure the integrity and security of the request, while the addition of the API key ensures the request's validity.

5. Trading Logic Analysis

We outline a trading strategy characterized by a systematic process. The trading logic involves following steps:

Offline database training: The process begins by accessing an offline database and utilizing previously fine-tuned machine learning model parameters to train a model. The aim is to generate the best-performing model based on past testing results.

Live market Data Analysis: Subsequently, live market data, specifically the limit order book, is accessed. Using the trained model, the next 10-second mid-price is predicted.

Trade signal generation: A signal is generated based on the difference between the predicted mid-price and the current mid-price, provided it is tradable. If the signal indicates a trading opportunity:

- The direction (buy or sell) of the trade is determined.
- Entry price and take profit price levels are calculated.

Order placement: Using a trading gateway, both buy and sell orders are sent to the exchange.

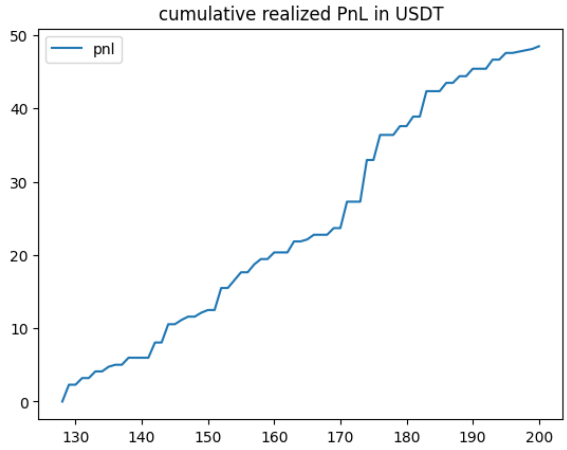
Order monitoring: The system continuously monitors the execution status of the orders. If an order fails to execute within a specified time frame, it is canceled, as signals are only valid for the predicted 10-second window.

Position management: Upon successful execution of an order, the system holds the position and waits for the take profit condition to be met.

Risk management: At regular intervals, the system checks the unrealized Profit and Loss (PnL) of the position. If the PnL exceeds a predefined threshold of 10 USDT, a stop-loss order is triggered to mitigate potential losses.

Position closure: Once the take profit or stop-loss conditions are met, the position is closed, and the entire process is repeated.

This systematic approach aims to exploit short-term market inefficiencies while employing risk management measures to mitigate potential losses. By leveraging machine learning models and real-time market data, the strategy seeks to optimize trading decisions and maximize profitability in dynamic market conditions.



6. Backtesting Success and Limitation

Finally, it's worth noting that while the backtesting results on test.net show a 100% success rate, there are several limitations to consider:

Virtual testing environment: It's important to recognize that our current testing environment is virtual, which means there is zero market impact. Real market conditions may differ significantly, affecting trading outcomes.

Order filling logic in demo trading: The order filling logic observed in demo trading relies on the

capabilities of the test.net service, rather than actual market execution. Thus, the results may not accurately reflect real-world performance.

Limit model training database: The model's training database may not encompass all relevant data, potentially leading to biased or incomplete training. In real-world scenarios, market conditions can vary widely, and the model's effectiveness may be influenced by the comprehensiveness of the training data. Improving the quality and quantity of the dataset is crucial for enhancing model performance and accuracy. This limitation can be addressed by continuously expanding the dataset to include more market scenarios and time periods.

Exhaustiveness of model fine-tuning: While efforts have been made to fine-tune the model, it's essential to acknowledge that model tuning is an ongoing process and may not capture all potential variations in market behavior.

Uncertainty in stop loss levels: Stop loss levels implemented in the trading strategy are not guaranteed to prevent losses under all market conditions. Market volatility and unexpected events can lead to stop loss levels being triggered differently from what was anticipated during backtesting.

Time constraints in backtesting: Backtesting is limited by time constraints and may not encompass all possible market fluctuations. Consequently, the performance of the API trading strategy under various market conditions may not be fully evaluated.

Connectivity stability: all such API instructions are based on stable connections to the server, with very limited error handling. In worst scenario of unstable connections, manual interfering is still essential.

Acknowledging these limitations is crucial for understanding the potential risks and uncertainties associated with deploying the trading strategy in live market conditions. Ongoing monitoring, refinement, and adaptation of the strategy based on real-time market feedback are essential for optimizing performance and managing risk effectively.