Fahrenheit Black

Michael Chase

12/01/2023

Fahrenheit Black Quantitative Trading System

Project Overview

This document provides a comprehensive architecture and design layout for a Quantitative Trading System aimed at executing multiple trading strategies simultaneously. The system will initially focus on trading stocks and options but is designed for easy extensibility to other asset types.

Objective

To build a modular, scalable, and efficient trading system that can:

- Fetch and manage trading data in real-time
- Execute multiple trading strategies concurrently
- Evaluate and manage risks at both strategy and portfolio levels
- Backtest strategies using historical data
- Execute orders in real-time
- Monitor performance and adapt dynamically

System Architecture

The system is divided into seven main modules:

1. Data Management Module

`DataRetrieval`

- **Public Properties:**
  - `provider`: The data provider (e.g., Interactive Brokers).
  - `symbol`: The trading symbol for which data is to be retrieved.
- **Functions:**
  - `.fetch_realtime_data(symbol: str) -> DataFrame`: Fetch real-time data for the specified symbol.
  - `.fetch_historical_data(symbol: str, start_date: datetime, end_date: datetime) -> DataFrame`: Fetch historical data for a specified time frame.

`DataStorage`

- **Public Properties:**
  - `data_path`: The file path where data is stored.
- **Functions:**
  - `.save_to_csv(data: DataFrame, filename: str) -> None`: Save DataFrame to a CSV file.
  - `.load_from_csv(filename: str) -> DataFrame`: Load data from a CSV file into a DataFrame.

`DataProcessing`

- **Public Properties:**
  - `data`: The raw data to be processed.
- **Functions:**
  - `.clean_data(data: DataFrame) -> DataFrame`: Clean and preprocess data.
  - `.transform_data(data: DataFrame) -> DataFrame`: Apply necessary transformations or calculations.

2. Trading Strategies Module

`StrategyInterface`

- **Public Properties:**
  - `data`: The trading data.
  - `signals`: Generated trading signals.
- **Functions:**
  - `.analyze(data: DataFrame) -> Series`: Analyze data and generate trading signals.
  - `.execute(signals: Series) -> List[Order]`: Execute trading signals and generate orders.

`StrategyExecutor`

- **Public Properties:**
  - `strategy`: The trading strategy to be executed.
- **Functions:**
  - `.run_strategy(strategy: Strategy) -> None`: Run a specific strategy.
  - `.scale_strategy(strategy: Strategy, factor: float) -> None`: Scale a strategy based on performance.

3. Risk Management Module

`StrategyLevelRisk`

- **Public Properties:**
  - `stop_loss_level`: The stop-loss level for the strategy.
  - `take_profit_level`: The take-profit level for the strategy.

- **Functions:**
  - `.apply_stop_loss(strategy: Strategy, level: float) -> None`: Apply stop-loss levels to a strategy.
  - `.apply_take_profit(strategy: Strategy, level: float) -> None`: Apply take-profit levels to a strategy.

`PortfolioLevelRisk`

- **Public Properties:**
  - `max_drawdown`: The maximum allowed drawdown for the portfolio.
- **Functions:**
  - `.calculate_max_drawdown(portfolio: Portfolio) -> float`: Calculate the maximum drawdown for the portfolio.
  - `.halt_trading(if max_drawdown > threshold: float) -> None`: Halt trading activities if risk thresholds are breached.

4. Backtesting Module

`CustomBacktester`

- **Public Properties:**
  - `strategy`: The strategy to be backtested.
  - `data`: The historical data for backtesting.
- **Functions:**
  - `.run_backtest(strategy: Strategy, data: DataFrame) -> DataFrame`: Run backtest for a specific strategy and return results.
  - `.calculate_performance_metrics(results: DataFrame) -> Dict`: Calculate performance metrics after backtesting.

`PerformanceAnalysis`

- **Public Properties:**
  - `results`: The results of the backtest.
- **Functions:**
  - `.visualize_results(results: DataFrame) -> None`: Generate visualizations for backtesting results.

5. Order Execution Module

`BrokerIntegration`

- **Public Properties:**
  - `broker`: The broker for order execution (e.g., Interactive Brokers).
- **Functions:**
  - `.execute_order(order: Order) -> Confirmation`: Execute order and return confirmation.
  - `.query_open_orders() -> List[Order]`: Return a list of open orders.
  - `.query_positions() -> List[Position]`: Return a list of current positions.
  - `.query_account_details() -> Dict`: Return account-related details.

`OrderManagement`

- **Public Properties:**
  - `orders`: A list of current orders.
- **Functions:**
  - `.create_order(signal: Signal) -> Order`: Create a new order based on trading signal.
  - `.monitor_order(order: Order) -> Status`: Monitor the status of an open order.
  - `.cancel_order(order: Order) -> Confirmation`: Cancel an open order.
  - `.modify_order(order: Order, modifications: Dict) -> Confirmation`: Modify an existing order.

6. Performance Metrics and Evaluation Module

`MetricsCalculation`

- **Public Properties:**
  - `metrics`: The calculated performance metrics.
- **Functions:**
  - `.calculate_real_time_metrics(data: DataFrame, orders: List[Order]) -> Dict`: Calculate real-time performance metrics.
  - `.calculate_post_trade_metrics(trades: List[Trade]) -> Dict`: Calculate post-trade metrics.

`Optimization`

- **Public Properties:**
  - `portfolio`: The current portfolio.
- **Functions:**
  - `.apply_half_kelly(portfolio: Portfolio) -> Dict`: Apply the Half Kelly Criterion for portfolio rebalancing.

7. Maintenance and Monitoring Module

`UpdatesManagement`

- **Public Properties:**
  - `updates`: The updates or patches to be applied.
- **Functions:**
  - `.apply_update(update: Update) -> Confirmation`: Apply system updates or bug fixes.

`Diagnostics`

- **Public Properties:**
  - `alerts`: The generated alerts.
- **Functions:**
  - `.send_alert(alert: Alert) -> Confirmation`: Send real-time alerts via SMS or email.
  - `.log_activity(activity: Activity) -> None`: Log system activities and performance.

Data Flow

1. Data Retrieval and Preparation

1. `DataRetrieval.fetch_realtime_data(symbol)` retrieves real-time data for a specified symbol from the `provider`.
2. `DataRetrieval.fetch_historical_data(symbol, start_date, end_date)` gets historical data for a specified time frame.
3. The fetched data are stored in `DataStorage.data_path` using `DataStorage.save_to_csv(data, filename)`.
4. Data are loaded into a DataFrame with `DataStorage.load_from_csv(filename)`.

2. Data Processing

5. `DataProcessing.clean_data(data)` cleans and preprocesses the loaded data.
6. `DataProcessing.transform_data(data)` applies necessary transformations or calculations.

3. Strategy Execution

7. `StrategyInterface.analyze(data)` analyzes the processed data to generate `StrategyInterface.signals`.
8. `StrategyExecutor.run_strategy(strategy)` executes the specific strategy using the generated signals.
9. If needed, `StrategyExecutor.scale_strategy(strategy, factor)` scales the strategy based on performance.

4. Risk Management

10. `StrategyLevelRisk.apply_stop_loss(strategy, level)` and `StrategyLevelRisk.apply_take_profit(strategy, level)` apply stop-loss and take-profit levels, modifying `StrategyLevelRisk.stop_loss_level` and `StrategyLevelRisk.take_profit_level`.
11. `PortfolioLevelRisk.calculate_max_drawdown(portfolio)` calculates the maximum drawdown and updates `PortfolioLevelRisk.max_drawdown`.
12. If the drawdown exceeds the threshold, `PortfolioLevelRisk.halt_trading()` is invoked to stop trading activities.

5. Order Creation and Execution

13. `StrategyInterface.execute(signals)` generates orders based on valid signals.
14. `OrderManagement.create_order(signal)` creates a new order and adds it to `OrderManagement.orders`.
15. `BrokerIntegration.execute_order(order)` sends the order to the broker for execution and returns a confirmation.
16. `OrderManagement.monitor_order(order)` monitors the status of the open order.

6. Performance Metrics

17. `MetricsCalculation.calculate_real_time_metrics(data, orders)` calculates real-time performance metrics and updates `MetricsCalculation.metrics`.
18. `MetricsCalculation.calculate_post_trade_metrics(trades)` calculates post-trade metrics after the orders are executed.

7. Backtesting (if needed)

19. `CustomBacktester.run_backtest(strategy, data)` performs backtesting for a specific strategy and returns results.
20. `CustomBacktester.calculate_performance_metrics(results)` calculates performance metrics based on backtesting results.
21. `PerformanceAnalysis.visualize_results(results)` generates visualizations for the backtesting results.

8. Portfolio Optimization

22. `Optimization.apply_half_kelly(portfolio)` applies the Half Kelly Criterion and suggests rebalancing actions.

9. Monitoring and Maintenance

23. `Diagnostics.send_alert(alert)` sends real-time alerts if there are any issues or significant events, utilizing the `Diagnostics.alerts` property.
24. `Diagnostics.log_activity(activity)` logs system activities and performance metrics.
25. If there are any updates, `UpdatesManagement.apply_update(update)` applies system updates or bug fixes, updating the `UpdatesManagement.updates` property.

10. Continuous Monitoring

26. `BrokerIntegration.query_open_orders()`, `BrokerIntegration.query_positions()`, and `BrokerIntegration.query_account_details()` can be used continuously to monitor the current state of orders, positions, and account details.