香 港 大 學
THE UNIVERSITY OF HONG KONG

ELEC7079 Investment and trading for engineering students

Group Project

Group 3

Department of Electrical and Electronic Engineering

Student Name: Yang Yifan, Li Chaolu,

Guo Yongzhi, Xu Yunxiang, Xu Yimeng

26 November 2024

Contents

List of Tables and Figures

1. Introduction

As the complexity of financial markets continues to increase, traditional investment strategies have gradually become insufficient to meet investors' dual demands for returns and risk management. In this context, the project aims to establish a systematic investment and trading analysis framework based on multi-factor models. By employing systematic data processing, building XGBoost models, and executing trades, the project significantly enhances investment efficiency and controllability.

Additionally, the project is equipped with a multi-layered risk control and position management mechanism to ensure effective risk management in a dynamic market environment. Through a rigorous backtesting process, the project will evaluate the strategy's performance on historical data, verifying its feasibility and effectiveness for practical application, thus achieving the goal of capturing market fluctuations using high-frequency trading data and multi-factor models.

2. Market Analysis

2.1 Global Market Environment

The global financial markets have been influenced by multiple factors in recent years, including rising interest rates, geopolitical tensions, and economic slowdown. Firstly, the frequent interest rate hikes by the Federal Reserve have led to a tightening of global liquidity, with international capital gradually withdrawing from high-risk markets and shifting towards more stable asset allocations.[1] As an international financial hub, Hong Kong’s monetary policy is pegged to the US dollar, meaning that Hong Kong’s interest rates are directly influenced by Federal Reserve policies. This has significantly impacted market liquidity. Rising interest rates not only increase corporate financing costs but also compress investor risk appetite, further intensifying market uncertainty.[2]

At the same time, international capital flows have become more volatile and directional. With lower risk appetite, investors have increasingly shifted towards safe-haven assets, putting significant pressure on sectors with high foreign capital participation in the Hong Kong stock market, such as technology, finance, and consumer stocks. However, the heightened volatility has also provided more opportunities for quantitative trading strategies, especially in the application of volatility and volume factors.

These factors help capture short-term market fluctuations and potential risks more effectively.[2]

Globally, emerging technology sectors have attracted substantial funds. Relevant listed companies in Hong Kong have become key targets for investment. Stocks in these industries have exhibited stronger trend characteristics, offering sustained investment opportunities for trend and momentum factors in quantitative trading.[3]

2.2 Structural Characteristics of Hong Kong Market

Due to the high participation of foreign capital in the Hong Kong market, stock prices are highly susceptible to international capital flows. Overseas investors account for a significant proportion of trading volumes in the Hong Kong market, making momentum and reversal factors particularly significant. [4]

At the same time, the Hong Kong market is interconnected with mainland China's A-shares through the Stock Connect programs (Shanghai-Hong Kong Stock Connect and Shenzhen-Hong Kong Stock Connect), creating a bridge for capital and information flows. Northbound inflows influence A-shares, while southbound capital allocation directly determines the liquidity and performance of Hong Kong stocks. This dual interconnectivity enhances the predictability of cross-market capital rotation, providing multi-factor strategies with richer trading signals.

In terms of trading characteristics, the Hong Kong market exhibits significant volatility, with uneven liquidity distribution. Large-cap blue-chip stocks tend to have better liquidity, while small-cap stocks often face higher risks due to short-term capital fluctuations.[5] This characteristic makes volatility and volume factors particularly prominent in the Hong Kong market. Furthermore, Hong Kong's trading system offers greater flexibility and intraday arbitrage opportunities for quantitative strategies.

However, the limited market depth can result in higher slippage costs for large-scale transactions, which must be carefully taken into account in strategy execution.

2.3 Stock Selection

The dataset for this project includes minute-level trading data for 100 stocks over an extended period, featuring five core variables: opening price (open_px), closing price (close_px), highest price (high_px), lowest price (low_px), and trading volume (volume). The data is organized by timestamp (ts) at 5-minute intervals, providing rich high-frequency information for factor extraction and strategy research. Each stock is distinguished by a unique identifier (symbol), and data records for each time point offer comprehensive market performance metrics for each stock.

	A	B	C	D	E	F	G	H	I	
1	symbol	STOCK_1							STOCK_2	
2		close_px	high_px	low_px	open_px	volume	close_px	high_px	low_px	ope
3	ts									
4	2019-01-02 09:30:00	49.75031	49.75031	49.75031	49.75031	3010.705	183.2435	183.2435	183.2435	183.
5	2019-01-02 09:35:00	49.48474	50.01588	49.48474	50.01588	60303.18	182.2718	183.8801	181.6017	183.
6	2019-01-02 09:40:00	49.48474	49.75031	49.48474	49.57326	29447.91	183.277	184.2821	181.8027	182.
7	2019-01-02 09:45:00	49.75031	49.75031	49.48474	49.57326	27577.35	182.2718	183.344	182.2718	183.
8	2019-01-02 09:50:00	49.57326	49.66179	49.57326	49.66179	31051.24	182.2383	182.3723	181.9367	182.
9	2019-01-02 09:55:00	49.39622	49.66179	49.39622	49.57326	54718.24	180.5965	182.3388	180.0939	182.
10	2019-01-02 10:00:00	49.30769	49.39622	49.21917	49.39622	55644.61	179.9599	180.8645	179.9599	180.
11	2019-01-02 10:05:00	49.48474	49.57326	49.30769	49.30769	28485.91	180.1274	180.1274	179.5913	180.
12	2019-01-02 10:10:00	49.30769	49.48474	49.30769	49.48474	20567.22	179.7253	180.1274	179.5243	180.
13	2019-01-02 10:15:00	49.39622	49.39622	49.30769	49.39622	11597.45	179.8594	179.8594	179.2228	179.
14	2019-01-02 10:20:00	49.39622	49.39622	49.30769	49.39622	21564.85	179.7588	180.0939	179.5913	179.
15	2019-01-02 10:25:00	49.30769	49.48474	49.30769	49.39622	50273.44	179.5578	179.9264	179.3233	179.
16	2019-01-02 10:30:00	49.30769	49.39622	49.13065	49.21917	45374.36	179.4238	179.7588	179.4238	179.
17	2019-01-02 10:35:00	49.30769	49.39622	49.21917	49.21917	16478.71	178.7202	179.4238	178.5191	179.
18	2019-01-02 10:40:00	49.21917	49.30769	49.13065	49.21917	25252.52	178.3181	179.0217	178.2511	178.
19	2019-01-02 10:45:00	49.30769	49.30769	49.13065	49.13065	37838.69	178.9882	179.2228	178.2846	178.
20	2019-01-02 10:50:00	49.21917	49.30769	49.21917	49.30769	14038.08	179.0217	179.2228	178.8542	179.
21	2019-01-02 10:55:00	49.39622	49.39622	49.21917	49.30769	7820.708	178.9882	179.2563	178.9547	179.
22	2019-01-02 11:00:00	49.39622	49.39622	49.21917	49.39622	12755.42	178.8877	179.2563	178.5526	178.

Figure 1 Structure of The Dataset.

The dataset includes both high-volatility stocks and stable growth stocks, providing comprehensive sample support for the study of momentum and reversal factors. For instance, in cyclical industries, stock prices may exhibit significant fluctuations, while stable industries are suitable for observing the performance of low-volatility factors. Additionally, the selected stock data possesses a high level of completeness, reducing the negative impact of missing or outlier values on factor extraction and strategy backtesting.

The minute-level structure of this dataset allows for the study of market microstructure and the development of short-term trading strategies, enabling the capture of subtle market changes within the Hong Kong trading system. The rich time span and wide range of stocks ensure that the model can effectively test the performance of both long-term and short-term factors.

The return distribution chart provides an intuitive visualization of quantitative analysis, showcasing the return characteristics of a specific stock during the analysis period. Through the statistical analysis of historical return data, it helps identify potential investment opportunities and risks. Here, the first four stocks are presented as representatives.

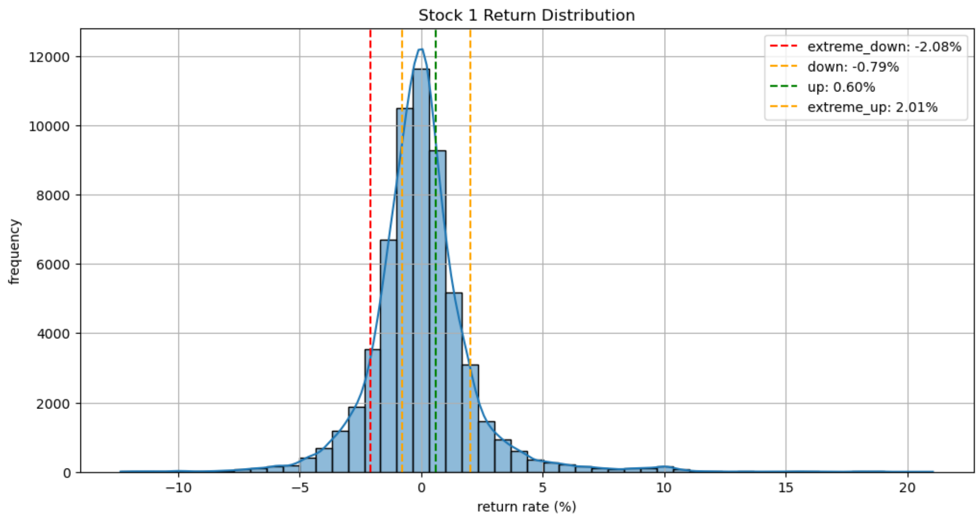

Figure 2 Stock 1 Return Distribution

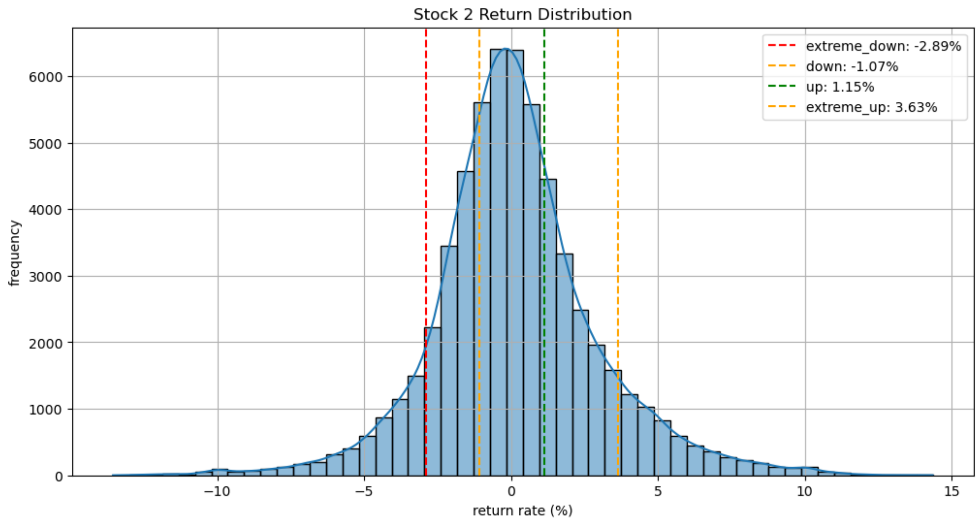

Figure 3 Stock 2 Return Distribution

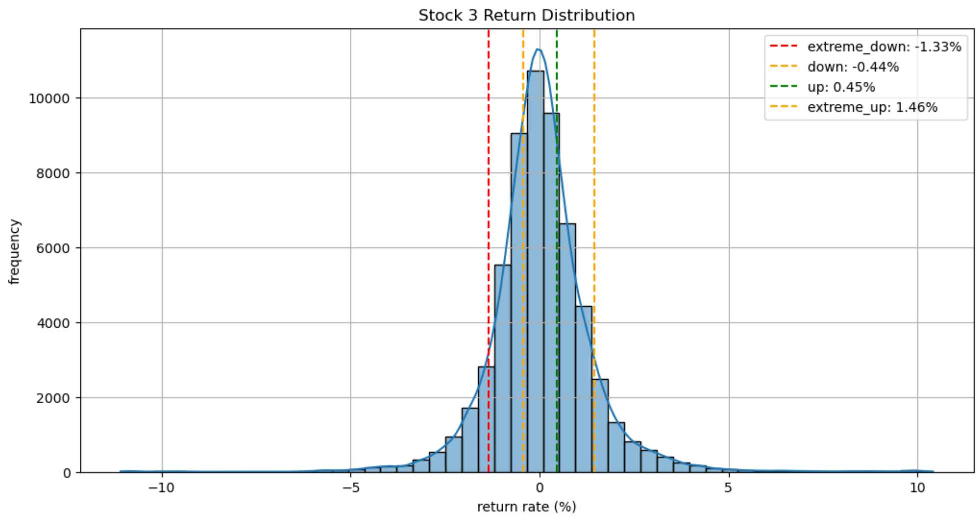

Figure 4 Stock 3 Return Distribution

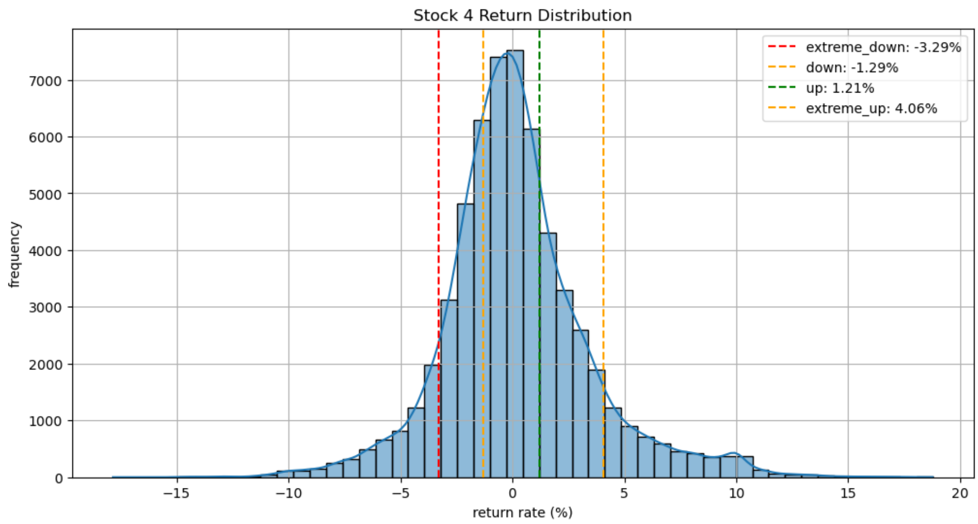

Figure 5 Stock 4 Return Distribution

In the chart, the horizontal axis represents the return rate (displayed as a percentage), reflecting the future return of the stock within a given prediction window. The vertical axis indicates the frequency, which is the number of samples that fall within specific return ranges, allowing for a quick identification of which return intervals occur most frequently, thereby revealing the return characteristics of the stock.

Additionally, the chart includes dynamic threshold lines, which are calculated based on historical return data and represent the boundaries of different return levels. For example, thresholds for extreme down, down, neutral, up, and extreme up are indicated with different colored dashed lines.

The return distribution graphs exhibit distinct peaks and troughs, reflecting the concentration of stock returns within specific ranges. This suggests that stock returns are not entirely random but instead exhibit a certain degree of regularity, such as relatively concentrated distributions with smaller fluctuations or higher levels of volatility. Furthermore, the skewness and kurtosis of the return distributions highlight the risk characteristics of the stock returns.

3. Data preprocessing

3.1 Overall Strategy

In the process of building a quantitative trading strategy, the first step is to clarify the investment objectives. This includes setting the expected returns, risk tolerance, and investment horizon for the strategy, as well as establishing specific performance metrics.

During the data preprocessing phase, it is essential to ensure that the dataset consists of high-quality historical market data, guaranteeing the integrity and accuracy of the data. Subsequently, the data should be cleaned by handling missing values and outliers, and normalized to ensure it is suitable for further analysis.

3.2 Data Cleaning

1) Outlier detection and correction.

For the price fields in the stock data (such as opening price, closing price, etc.) and volume fields, statistical methods are used to detect outliers. Specifically, the interquartile range (IQR) method is employed to identify outliers. For example, if volume data exceeds $Q1 - 1.5 \times IQR$ or $Q3 +$

$1.5 \times IQR$, it is considered an outlier.

For the detected outliers, the code offers logical corrections or interpolation methods to adjust the values, ensuring the reasonableness and continuity. Logical corrections are primarily used for rule-based anomalies, while interpolation is suitable for time series data with strong continuity.

```
# 计算四分位数
Q1 = volume.quantile(0.25)
Q3 = volume.quantile(0.75)
IQR = Q3 - Q1

# 定义异常值的界限
volume_lower = Q1 - 2 * IQR
volume_upper = Q3 + 2 * IQR

# 检测异常值
volume_outliers = (volume < volume_lower) | (volume > volume_upper)
```

2) Missing Value Handling

Count missing values in each field to assess data quality to assess the severity of data loss and guide the selection of appropriate filling methods.

Linear Interpolation: This method is suitable for prices or volumes with sporadic gaps, assuming smooth changes that occur gradually over time.

Forward Fill: Fills missing values with the last observed value, which is ideal for stable data with clear trends.

Backward Fill: Replaces missing values with the next observed value, useful when future data points are relevant for determining trends.

```
# 1. 线性插值处理少量缺失
before_interpolation = df.isna().sum().sum()
df.interpolate(method='linear', limit=self.config.interpolation_window, inplace=True)
missing_stats['interpolated_count'] = before_interpolation - df.isna().sum().sum()

# 2. 对剩余缺失使用前向填充
before_ffill = df.isna().sum().sum()
df.fillna(method='ffill', inplace=True)
df.fillna(method='bfill', inplace=True)
missing_stats['forward_filled_count'] = before_ffill - df.isna().sum().sum()
```

3) Price Logic Validation

Perform logical consistency checks on the price fields of stocks. The high price is always greater than or equal to the low price.

The high price must be greater than or equal to both the open and close prices. If inconsistencies are found, the high price is adjusted to the maximum of the open, close, and current high.

Low Price Validation: The low price should be less than or equal to both the open and close prices. Inconsistencies are corrected by setting the low price to the minimum of the open, close, and current low.

4) Zero value analysis

This analysis helps identify potential data quality issues, such as non-trading days or data entry errors. Based on the findings, further filtering of the data can be performed.

4. Factors

4.1 The Reasons We Chose Factor-based Investing

In the machine learning stock selection project, factor-based stock analysis plays a crucial role. Through comprehensive consideration of multiple factors, the investment value of stocks can be evaluated more accurately from multiple dimensions, providing a strong basis for stock selection.

4.2 Factors in The Project

1) Momentum Factor

The Momentum Factor is based on the assumption of “price trend continuation,” where assets with strong past performance may continue to perform well in the short term, while poorly performing assets may continue to decline. It is typically measured using price changes or logarithmic returns. An increase in the Momentum Factor often indicates heightened market sentiment, with investors chasing upward trends. However, an excessively high Momentum Factor may signal a price peak and potential reversal. Since momentum effects require a sufficiently long observation period to capture stable trends, we have set the period to 10 hours.

Logarithmic Returns:

$$\boxed{R_t = \ln\left(\frac{P_t}{P_{t-1}}\right)}$$

where P_t is the closing price at time t.

Exponentially Weighted Momentum:

$$\boxed{\mathrm{Momentum}_t = \mathrm{EWM}(R_t, \mathrm{span} = \mathrm{period})[-1]}$$

where EWM is the exponentially weighted moving average.

Normalization:

$$\boxed{\mathrm{Signal} = \tanh(10 \cdot \mathrm{Momentum}_t)}$$

2) RSI Factor (Relative Strength Index)

RSI is used to measure whether an asset is overbought or oversold, with values ranging from 0 to 100. An RSI above 70 indicates an overbought condition, while an RSI below 30 indicates an oversold condition. An increase in the RSI coefficient suggests that the market may be in an overbought state, potentially leading to a price pullback; a low RSI coefficient indicates oversold conditions, where prices may rebound. To balance short-term fluctuations and trends, we have set the period to 12 hours.

Price Change:

$$\boxed{\Delta P_t = P_t - P_{t-1}}$$

Gains and Losses:

$$\boxed{G_t = \max(0, \Delta P_t), \quad L_t = \max(0, -\Delta P_t)}$$

Average Gains and Losses:

$$\boxed{\mathrm{AvgG}_t = \mathrm{MA}(G_t, \mathrm{period}), \quad \mathrm{AvgL}_t = \mathrm{MA}(L_t, \mathrm{period})}$$

RSI Calculation:

$$\boxed{RS_t = \frac{\mathrm{AvgG}_t}{\mathrm{AvgL}_t}, \quad \mathrm{RSI}_t = 100 - \frac{100}{1 + RS_t}}$$

Normalization:

$$\boxed{\mathrm{Signal} = \begin{cases} -1, & \text{if RSI} > 70, \\ 1, & \text{if RSI} < 30, \\ \frac{\mathrm{RSI}-50}{20}, & \text{otherwise.} \end{cases}}$$

3) MACD Factor (Moving Average Convergence Divergence)

The MACD factor reflects trend changes by analyzing the difference between short-term and long-term moving averages. It is commonly used to capture reversal signals in price trends. An increase in the MACD factor indicates that short-term prices are rising faster than the long-term trend, potentially forming an upward trend. Conversely, a decrease suggests a weakening trend or a potential reversal. The fast line is set to 2 hours, the slow line to 4.3 hours, and the signal line to 1.5 hours. These parameters are based on classic settings, adjusted after multiple trials and slightly modified according to high-frequency data.

EMA (Exponential Moving Average) for Fast and Slow Lines:

$$\boxed{\begin{aligned}\text{EMA}_{\text{fast}}(t) &= \frac{P_t \cdot 2}{\text{fast\_period}+1} + \left(1 - \frac{2}{\text{fast\_period}+1}\right)\text{EMA}_{\text{fast}}(t-1)\\ \text{EMA}_{\text{slow}}(t) &= \frac{P_t \cdot 2}{\text{slow\_period}+1} + \left(1 - \frac{2}{\text{slow\_period}+1}\right)\text{EMA}_{\text{slow}}(t-1)\end{aligned}}$$

MACD Line:

$$\boxed{\text{MACD}_t = \text{EMA}_{\text{fast}}(t) - \text{EMA}_{\text{slow}}(t)}$$

Signal Line:

$$\boxed{\text{Signal}_t = \text{EMA}_{\text{MACD}}(\text{signal\_period})}$$

MACD Histogram:

$$\boxed{\text{Hist}_t = \text{MACD}_t - \text{Signal}_t}$$

Combined Score:

$$\boxed{\text{CombinedScore} = 0.7 \cdot (\text{MACD}_t - \text{Signal}_t) + 0.3 \cdot (\text{Hist}_t - \text{Hist}_{t-1})}$$

Normalization:

$$\boxed{\text{Signal} = \tanh(10 \cdot \text{CombinedScore})}$$

4) Volatility Factor

The volatility factor reflects the intensity of asset price changes. High volatility indicates uncertainty, risk, or significant market events, while low volatility suggests a relatively stable market. An increase in the volatility factor reflects market panic or major events, leading to sharp price fluctuations, while a decrease indicates market stability with smaller price movements. As a sufficiently long window is needed to compare short-term and long-term volatility, we have set the period to 20 hours.

Logarithmic Returns:

$$\boxed{R_t = \ln\left(\frac{P_t}{P_{t-1}}\right)}$$

Short-Term and Long-Term Volatility:

$$\boxed{\sigma_{\text{short}} = \text{STD}(R_t[-\text{period}/4:]), \quad \sigma_{\text{long}} = \text{STD}(R_t[-\text{period}:])}$$

Volatility Ratio:

$$\boxed{\text{VolRatio} = \frac{\sigma_{\text{short}}}{\sigma_{\text{long}}} - 1}$$

Normalization:

$$\boxed{\text{Signal} = -\tanh(3 \cdot \text{VolRatio})}$$

5) Trend Factor

The Trend Factor determines the long-term direction of prices by analyzing their deviation from moving averages (e.g., moving averages). It reflects whether the market is in an upward trend or a downward trend. An increase in the Trend Factor indicates that the asset price is in an upward trend, while a decrease suggests the price is in a downward trend. It requires a long window to compare short-term and long-term volatility, so we set the period to 25 hours.

EMA:

$$\boxed{\mathrm{EMA}_t = \frac{P_t \cdot 2}{\mathrm{period}+1} + \left(1 - \frac{2}{\mathrm{period}+1}\right)\mathrm{EMA}_{t-1}}$$

Price to EMA Ratio:

$$\boxed{\mathrm{PriceRatio} = \frac{P_t}{\mathrm{EMA}_t} - 1}$$

EMA Slope:

$$\boxed{\mathrm{EMASlope} = \frac{\mathrm{EMA}_t}{\mathrm{EMA}_{t-1}} - 1}$$

Combined Score:

$$\boxed{\mathrm{TrendScore} = 0.7 \cdot \mathrm{PriceRatio} + 0.3 \cdot \mathrm{EMASlope}}$$

Normalization:

$$\boxed{\mathrm{Signal} = \tanh(5 \cdot \mathrm{TrendScore})}$$

6) Reversal Factor

The Reversal Factor uses the degree of price deviation from the moving average and RSI signals to identify mean-reversion characteristics. It indicates whether prices are in extreme conditions. A high Reversal Factor suggests that asset prices have deviated significantly and may revert to the mean. A low Reversal Factor indicates that prices are close to the mean, with less likelihood of a major correction. We have set the period to 20 hours to ensure reversal signals are supported by sufficient data.

Moving Average:

$$\boxed{\mathrm{MA}_t = \mathrm{Mean}(P_t[-\mathrm{period}\ :])}$$

Deviation:

$$\boxed{\mathrm{Deviation} = \frac{P_t}{\mathrm{MA}_t} - 1}$$

RSI:

$$\boxed{RSI = \frac{\mathrm{PositiveReturns}}{\mathrm{PositiveReturns} + \mathrm{NegativeReturns}}}$$

Combined Score:

$$\boxed{\mathrm{ReversalScore} = -0.6 \cdot \mathrm{Deviation} - 0.4 \cdot (RSI - 0.5) \cdot 2}$$

Normalization:

$$\boxed{\mathrm{Signal} = \mathrm{clip}(\mathrm{ReversalScore}, -1, 1)}$$

7) Volume Factor

The Volume Factor measures the relative change in current trading volume compared to historical averages, while also considering price-volume relationships. Volume is often viewed as a confirmation of price trends. An increase in the Volume Factor indicates active market trading, which may support the current price trend. A decrease suggests weakening momentum or market hesitation. In high-frequency data, a shorter period can reflect changes in volume more quickly, so we set the period to 1.5 hours.

Relative Volume Ratio:

$$\boxed{\text{VolRatio} = \frac{V_t}{\text{Mean}(V_t[-\text{period} :])} - 1}$$

Price-Volume Relationship:

$$\boxed{\text{PriceVolRatio} = \frac{V_t}{|R_t|} - \text{Mean}\left(\frac{V_t}{|R_t|}[-\text{period} :]\right)}$$

Combined Score:

$$\boxed{\text{VolumeScore} = 0.6 \cdot \tanh(2 \cdot \text{VolRatio}) + 0.4 \cdot \tanh(2 \cdot \text{PriceVolRatio})}$$

5. Trading Strategy

5.1 XGBoost Algorithm Principles

XGBoost is an advanced boosting algorithm built upon existing boosting methods such as AdaBoost and GBDT (Gradient Boosting Decision Trees). Like all data mining algorithms, XGBoost consists of three key components: models, parameters, and objective functions. The model is a combination of base functions and weights, while parameters are the final results to be derived from model building. Taking decision trees as an example, parameters include the path q from root node to leaf nodes and the expected weight w of each leaf node. Generally, a model's accuracy depends on how well the objective function is optimized - better optimization leads to predictions closer to true values and better generalization capability. These goals can be achieved by minimizing the loss function while adding a penalty term for model complexity.

The objective function Obj is defined as:

$$\boldsymbol{Obj}(\boldsymbol{\theta}) = \boldsymbol{L}(\boldsymbol{\theta}) + \boldsymbol{O}(\boldsymbol{\theta})$$

The objective function Obj(θ) consists of two parts: the loss function L(θ) and the regularization function O(θ). Machine learning ultimately aims to achieve low bias while maintaining reasonable function complexity. High complexity can lead to overfitting risks. The key focus is finding the optimal balance between bias and variance. Bias represents the error between predicted and actual values, while variance indicates how changes in the dataset affect learning capability.

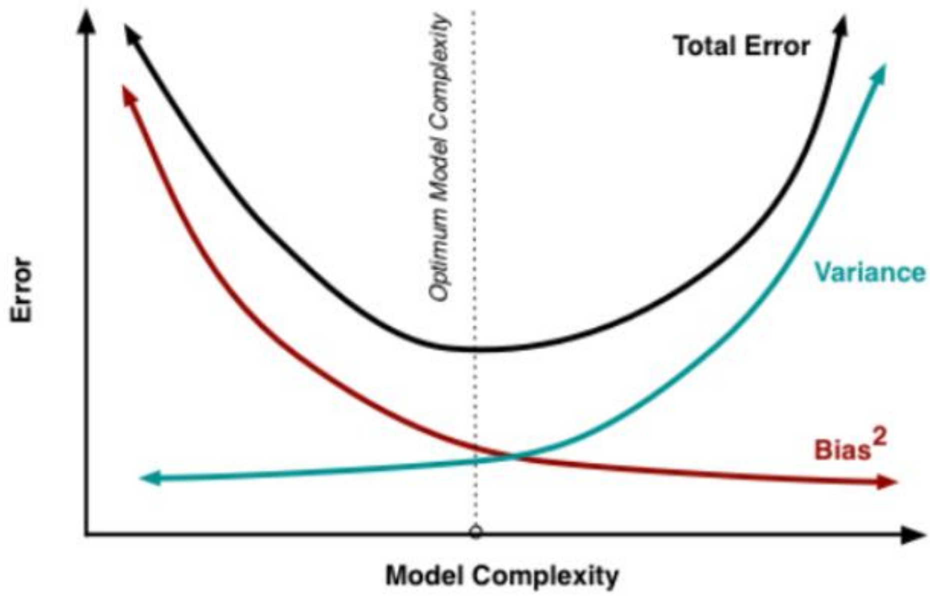

Figure 6 Bias-Variance Tradeoff Diagram

As shown in Figure 6, the intersection of bias and variance curves represents the model's optimal point. At this point, both bias and variance are relatively low, resulting in optimal model prediction capability. To the left of this vertical line, the model is underfitting; to the right, it's overfitting.

5.2 Loss Function

XGBoost is an ensemble boosting algorithm composed of a series of base classifiers (also called weak classifiers). Common choices for base classifiers include decision trees and logistic regression, though efficient classifiers like SVM can also serve as base classifiers.

The principle behind XGBoost is to divide the original dataset into multiple subsets, randomly assign each subset to base classifiers for prediction, and then combine their results using specific weights to make the final prediction. In simpler terms, boosting algorithms work like a selection process where each round picks the best performers from many candidates, similar to selecting the fastest runners who together win the race.

Base classifiers can only achieve good results through collaboration. This differs from traditional SGD models, which aim to find a function that minimizes the squared loss function. Models that add base classifier results are typically called additive models, expressed as:

$$\boldsymbol{F} = \sum_{\boldsymbol{i}=\boldsymbol{1}}^{\boldsymbol{m}-\boldsymbol{1}} \boldsymbol{f} + \boldsymbol{f}_{\boldsymbol{m}}$$

Additive models predict by performing backward stacking of each base classifier's results through spline smoothing, achieving a balance between fitting error (bias) and complexity (variance).

When using CART (Classification and Regression Trees) as base classifiers for XGBoost, tree depth must be carefully managed. While deeper trees can be very effective classifiers, they risk overfitting. Although Random Forests also combine decision trees using SGD and Bootstrap approaches and can select sample features to largely avoid overfitting, XGBoost's additive model structure proves more powerful.

For example, when choosing CART as the base function, the prediction result for the M-th single CART is:

$$\boldsymbol{f}_{\boldsymbol{m}} = \boldsymbol{T}(\boldsymbol{X}; \boldsymbol{\theta}_{\boldsymbol{m}})$$

With the base function determined, where T represents the decision tree, m represents the number of base classifiers, and theta represents the decision tree's splitting path, the final prediction result is the sum of the previous prediction and the current decision tree. The error term can be expressed as:

$$\boldsymbol{L(y,\hat{y}) = L\big(y, f_{m-1}(x)\big) + T(X,\theta_m)}$$

where $L(y,\hat{y})$ is the sum of differences between true value y and predicted value ŷ.

5.3 Complexity Control

The Gini index, pruning, and tree depth control are crucial classification tools in CART. These methods control model variance and bias to enhance model generalization capability. Following this principle, we can define the structural complexity function using the number of leaf nodes T and the L2 norm squared of LeafScore:

$$\emptyset(\boldsymbol{\theta}) = \boldsymbol{\gamma T} + \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{\lambda}\sum_{\boldsymbol{J=1}}^{\boldsymbol{T}} ||\boldsymbol{W}||_{\boldsymbol{j}}^{\mathbf{2}}$$

Here, γ is the coefficient controlling tree complexity, effectively performing pre-pruning on XGBoost trees, while λ determines the proportion of regularization term modification, penalizing complex models to prevent overfitting.

Combining bias and variance functions yields the following objective function:

$$\boldsymbol{Obj(\theta)} = \sum_{i} \boldsymbol{l(y_t + y_i)} + \boldsymbol{\gamma T} + \frac{\mathbf{1}}{\mathbf{2}}\lambda \sum_{J=1}^{T} \|\boldsymbol{W}\|_j^2$$

5.4 Gradient Optimization

Traditional approaches to Gradient Boosting Decision Trees repeatedly calculate objective function errors to minimize them using gradient descent. This method ensures optimal results by selecting weak classifiers and combining them additively.

The gradient descent formula is:

$$-\left[\frac{\partial L(y, f(x_i))}{\partial f(x_i)}\right] f(x) = f_{m-1}(x)$$

After computing the loss function derivative, both the child nodes and node weights of the tree can be determined, thus defining the tree structure.

Given XGBoost's use of multiple base classifiers, a more general algorithm for gradient descent is needed. The inventor of XGBoost replaced the original first-order derivative with Taylor second-order expansion, making the algorithm more universal.

The objective function with Taylor second-order expansion becomes:

$$Obj(\theta)\sum_{i=1}^{n} l\left(\left(y_t, y_i^{m-1}\right) + f_m(x_i)\right) + \emptyset(\theta)$$

where n represents the number of samples used, m represents the current iteration count, and l(m) represents the current iteration error.

Using Taylor expansion:

$$f(x + \Delta x) \cong f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2$$

Defining:

$$g_i = \partial_{y(m-1)} l\left(y_i, y^{(m-1)}\right), h_j = \partial^2_{y(m-1)}$$

The final computation becomes:

$$Obj \cong \sum_{i=1}^{n} \left[l\left(y_i, y^{(m-1)}\right) + g_i f_m(x_i)\right] + \emptyset(f_m)$$

This formulation enables more efficient and effective optimization of the XGBoost algorithm.

5.5 XGboost Implementation

The XGBoost model our project implemented employs a practical five-category prediction scheme based on dynamic thresholds. Unlike traditional fixed-threshold classification methods, our project implemented a dynamic threshold determination mechanism through return analyzing based on percentiles: using the 10th, 30th, 70th, and 90th percentiles of return distribution as classification boundaries to divide returns into five categories - extreme_down (<10th percentile), down (10th-30th percentile), fluctuation (30th-70th percentile), up (70th-90th percentile), and extreme_up (>90th percentile). This dynamic threshold design effectively accounts for variations in return distributions across different stocks, allowing better adaptation to individual stock characteristics and changing market conditions.

In terms of model training configuration, our project adopted a time series cross-validation approach, setting a training window of 120 periods and a prediction window of 49 periods, with 5-fold cross-validation for model performance evaluation. This configuration ensures sufficient training data while capturing an appropriate prediction timeframe through the prediction window setting. Notably, in sample processing, our project implemented a class weight-based balancing strategy by calculating sample weights for each category (total_samples / (count * len(class_counts))) to address class imbalance issues, which is crucial for improving the model's predictive capability for minority classes.

The core parameters of the XGBoost model are configured as follows:

1. Basic Parameter Settings:
    - learning_rate = 0.05: A relatively small learning rate to ensure model stability

    - n_estimators = 500: Sufficient number of trees to ensure adequate learning
    - objective = 'multi:softmax': Multi-classification task setting
    - num_class = 5: Corresponding to the five-category prediction task
2. Tree Structure Parameters:
    - max_depth = 6: Controls tree depth to prevent overfitting
    - min_child_weight = 2: Controls minimum sample weight sum for child nodes, preventing excessive subdivision
    - gamma = 0.1: Controls minimum gain for node splitting
3. Sampling and Regularization Parameters:
    - subsample = 0.9: Randomly selects 90% of samples to build trees
    - colsample_bytree = 0.9: Randomly selects 90% of features to build trees
    - reg_alpha = 0.1: L1 regularization parameter
    - reg_lambda = 1: L2 regularization parameter
4. Evaluation Metric Settings:
    - eval_metric = ['mlogloss', 'merror']: Monitors both logarithmic loss and classification error rate

These parameter configurations demonstrate fine-grained control over model training, achieving a good balance between model performance and generalization capability through multi-level adjustments. The regularization parameter settings, combined with sampling strategies and tree depth control, form an effective overfitting prevention mechanism. Meanwhile, the choice of evaluation metrics ensures the monitoring of the training process, facilitating timely detection and adjustment of training issues.

6. Trading Execution

This trading system is designed to implement trading strategy execution through predictive modeling. The entire trading process is as follows.

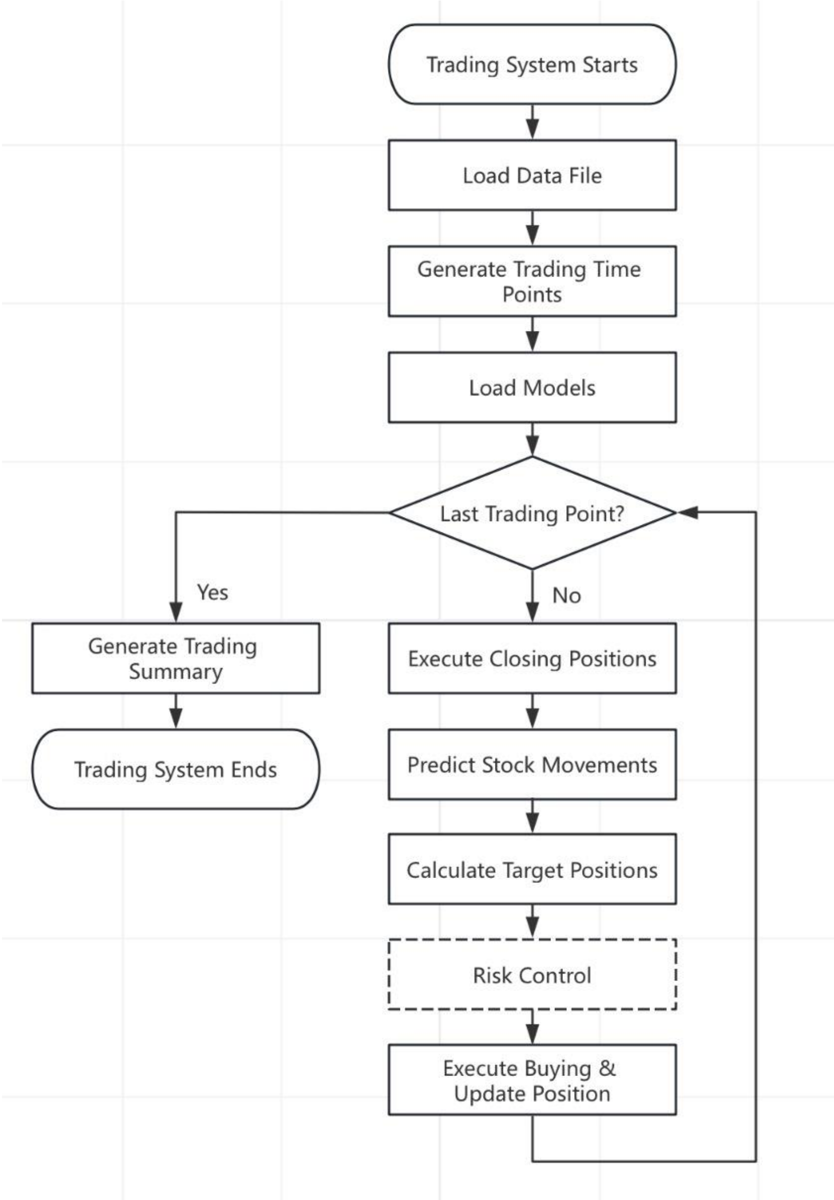

Figure 7 The Flow Chart of Entire Trading System

6.1 Initialization and Core Structure Setup

The system begins with an initialization phase, which includes setting up logging and defining core data structures. The logging module uses a standardized format to record critical information during runtime, facilitating debugging and post-analysis.

```
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
```

Simultaneously, the system employs dataclass to define several essential data structures, such as TradingPoint and StockPrediction, which describe the input and output data formats of the trading system.

```
@dataclass
class TradingPoint:
    """交易点数据结构"""
    position: int                 #
    historical_data: pd.DataFrame
    current_price: float         #
    future_price: float          #
```

```
@dataclass
class StockPrediction:
    """股票预测结果数据结构"""
    stock_code: str
    prediction: PredictionResult
    confidence: float
    position: int   # 替换timestamp
```

To support flexible handling of prediction results, an enumeration class PredictionResult categorizes possible market changes, including "Extreme Down," "Down," "Neutral," "Up," and "Extreme Up."

```
class PredictionResult(Enum):
    """预测结果枚举"""
    EXTREME_DOWN = 0
    DOWN = 1
    NEUTRAL = 2
    UP = 3
    EXTREME_UP = 4
```

6.2 Data Management and Model Loading

The system manages trading data by loading historical records and extracting features necessary for predictive modeling. A warm-up mechanism ensures sufficient historical data is available during initialization, which is essential for reducing errors in high-frequency or real-time trading scenarios.

The trained model drives the system's decision-making, processing historical data features to predict market direction and confidence levels. Predictions are stored in StockPrediction objects, providing structured and actionable insights.

6.3 Close Positions Before Opening New Ones

In the dataset, stock prices are recorded every five minutes, totaling 49 data points per day. Our trading interval is set to 1 day (Window = 49). For each stock, the system executes close positions based on the time window length (49 data points) and sells the currently held stocks once it reaches the closing time point. This means that the holding period for each stock is 1 day.

```python
def __init__(self, warmup_period: int = 300, prediction_window: int = 49):
    self.warmup_period = warmup_period
    self.prediction_window = prediction_window
    self.logger = logging.getLogger(__name__)
```

After selling a stock, the system predicts the stock trend for the next day. If the predicted trend is upward, the system will choose to buy the stock again; if the trend is down, the system will temporarily refrain from buying, waiting until the prediction indicates an upward trend before executing the buy operation. This cycle repeats itself.

The system follows a "sell before buy" process: it first checks the existing positions and sells the positions that have been held for one day, then, based on the predicted results, decides whether to open new positions for today. This mechanism ensures that buy and sell operations occur in pairs within each trading cycle, with each position being held for only one day.

6.4 Trend Prediction

The system inputs data into the xgboost model, which uses a pre-trained ensemble of decision trees to analyze the factor feature data (such as momentum, trend, volatility, etc.) and outputs prediction results (classification) based on the feature patterns.

For example, if the predicted category is **UP** or **EXTREME UP**, it indicates that the data's predicted result shows an upward trend. If the prediction falls into other categories (such as **DOWN, EXTREME DOWN** or **NEUTRAL**), the system will not buy the stock until the trend changes.

The system can further evaluate the reliability of the prediction based on the confidence value. Each trading signal is accompanied by a confidence value that reflects the system's trust in the prediction results. The higher the confidence, the stronger the reliability of the trading signal.

6.5 Trading Mechanism of Window

The fixed trading window mechanism ensures that even if the predicted results for certain stocks do not meet the expects, the system will still strictly liquidate the held stocks on time, resulting in a consistent number of buy and sell operations.

The capital management restriction mechanism causes the system to limit stock purchases based on the maximum position ratio for the single stock and the total position. This may lead to some stocks reaching limits, thereby controlling risk.

7. Risk Control

7.1 Position Control

In the investment world, position control is a core component of risk management, which involves how money is allocated to minimize potential losses and increase portfolio stability. This section explores in detail two position control strategies: single stock maximum position ratio and total position ratio, as well as their implementation and role in the code.

7.1.1 Single Stock Maximum Position Ratio

The single stock maximum position ratio is limited to minimize the portfolio's dependence on any single stock. This control strategy helps to diversify unsystematic risk, i.e., risk specific to a single stock or sector. By limiting the maximum percentage of a single stock in a portfolio, we can ensure that even if an unfavorable event occurs in one stock, it will not have a catastrophic effect on the entire portfolio.

In the code implementation, the max_single_position parameter of the PortfolioManager class is the parameter that controls this ratio. Specifically, after debugging the trading system several times, we chose to set max_single_position to 0.0065, meaning that the maximum investment in any single stock in the portfolio cannot exceed 0.65% of the total capitalization. This setting helps us to protect the portfolio from excessive volatility in a single stock by maintaining exposure to individual stocks without over-concentration.

7.1.2 Total Position Ratio

The opposite of the maximum position ratio in a single stock is the total position ratio control, which involves the sum of all positions in the portfolio. The purpose of this control strategy is to limit our overall share of the market in order to avoid significant losses when the market as a whole is unfavorable. By controlling the proportion of total positions, we can ensure that there is sufficient liquidity to cope with market uncertainty, while also reserving space for capturing new investment opportunities.

In the code, the max_total_position parameter of the PortfolioManager class is used to implement the control of the total position ratio. After several debugging sessions with different values of the parameter in the code, we set max_total_position to 0.6, which means that the total percentage of all positions in the portfolio must not exceed 60%. This measure ensures that we do not invest more than 60% of our capital in the market, thus pursuing returns while retaining enough capital for market volatility.

7.2 Pre-processing Controls

The filter_stocks function in the trading system code will filter the stocks involved in the trading process before the trading behavior, this function can screen out the nature of the stocks that are more in line with the requirements of low-risk, to reduce the risk of the trading process. filter_stocks function is based on the following three key indicators to screen the stocks: the minimum turnover (min_volume), the minimum price (min_price), and the maximum volatility (max_volatility). After the stock is entered into the function, only stocks that meet the requirements of the three key indicators can be retained: average volume greater than or equal to the threshold (100), the average price is greater than or equal to the threshold (1.0), volatility is less than or equal to the threshold (0.05). This process helps to reduce portfolio risk at the data level by ensuring that only stocks that meet specific risk and return characteristics are selected.

7.3 Trading Controls

Effective control of trading behavior reduces the risk associated with market uncertainty. Setting the values of extreme upside and downside weights as well as setting the confidence level of the forecast can fulfill this control function.

7.3.1 Adjustment of Extreme Upside and Upside Weights

In the PortfolioManager class of the model, we adjust the extreme upside and upside weights based on the confidence of the market forecast. This strategy helps us find a balance between risk and reward and minimize the risk of overtrading.

The extreme_up_weight parameter has a value of 1.5 and the up_weight parameter has a value of 1.0. Adjusting these two parameters essentially dynamically adjusts the sensitivity of the system to market movements. In times of high market volatility or increased uncertainty, the size of trades can be reduced by lowering these weights, thereby reducing risk. Conversely, when market conditions are stable and forecast confidence is high, we can modestly increase the weights to capture more market opportunities. The values of these two parameters set in the report (1.5 vs. 1.0) are the relatively appropriate results after many debugging sessions.

7.3.2 Forecast Confidence

Prediction confidence plays an important role in trading decisions. A confidence threshold can be preset in the code, and we will only execute a trade if the model's prediction confidence for a stock is higher than this threshold. This measure helps to reduce uncertainty and improve the quality of trades.

The factor_trainer.py in the data training system, after running, produces predictions that include the confidence parameter, the value of which is the confidence threshold set in the trading system. In the implementation of the trading system, the StockPrediction object contains the prediction results and the confidence level. In specific operations, once the value of the confidence level derived from the StockPrediction is greater than the preset confidence threshold, the transaction will be terminated immediately. This is the risk control of the trading system in terms of confidence.

7.4 Capital Management

7.4.1 Initial capital

In the PortfolioManager class, the initial capital is set as the starting capital of the trading system, and the corresponding parameter in the code is initial_capital, which is required by the title, and the value of this parameter in the system is 10000000.

Initial_capital provides the basis for portfolio capital allocation, allowing PortfolioManager to dynamically adjust the position size of each stock according to market conditions and forecasts, thus achieving risk control.

7.4.2 Calculate Target Position Size

The calculate_position_sizes method dynamically calculates target position sizes, dynamically adjusts positions based on predictions and current prices, and ensures rational capital allocation. This method contains two sets of parameters, predictions and current_prices.

predictions is a list of StockPrediction objects, each of which contains a prediction for a particular stock, such as the prediction category (up, down, etc.) and confidence level. This parameter is the key basis for the method to determine which stocks are worth investing in and the confidence with which to do so.

current_prices is a dictionary with the key being the stock code and the value being the current market price of the corresponding stock. This parameter ensures that the calculated target position size is based on the latest market information, thus improving the timeliness and accuracy of the decision.

7.4.3 Forced Position Liquidation

In extreme market conditions, we force liquidation of positions at the last trading point to protect capital and exit the market in a timely manner. This measure is implemented in the execute_trading method in the code and is controlled by the force_clear parameter. When force_clear is set to True, the system automatically closes all positions, regardless of the current market conditions.

In the code implementation, the force_clear parameter is used in conjunction with the _execute_close_positions method. When force_clear is activated, the method ignores other trading logic and performs a sell operation directly on all positions. This process is not only fast, but ensures that all positions are cleared before market conditions deteriorate further.

8. Model Performance Evaluation

In the financial field, building an effective forecasting model is an important basis for investment decisions. This model evaluates the performance of a stock prediction model based on a multi-factor model and XGBoost classification algorithm. We will use different performance metrics, including accuracy, accuracy, recall, F1 score, Sharpe rate, etc., to fully evaluate the performance of the model.

In the model, for each category (extreme decline, decline, shock, rise, extreme rise), we calculate the accuracy, accuracy, recall rate, F1 score and other indicators of the category in the forecast results by constructing the category mask, and then calculate the overall accuracy. Then, for the uptick (both up and extreme up categories), measures such as prediction accuracy, recall rate, and F1 score are calculated.

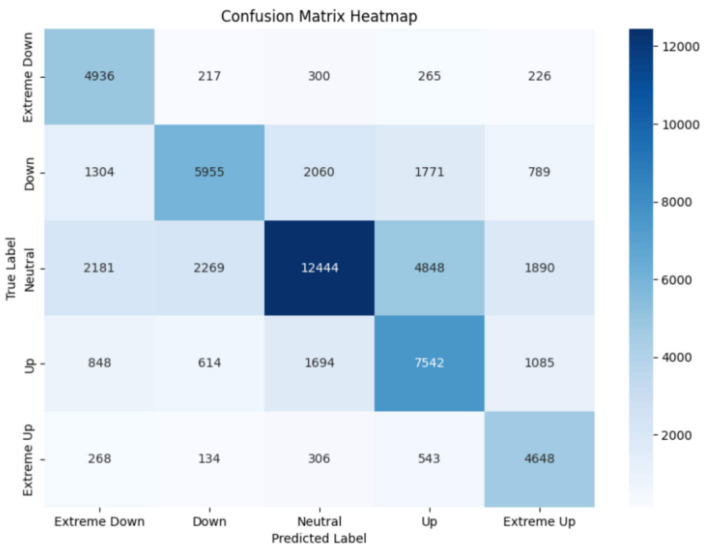

Figure 8 Confusion Matrix Heat Map of Stocks

For the training model, we generated some images to evaluate the model performance, such as the confusion matrix graph. By looking at this image, we find that the overall prediction accuracy of the model is very high. The model has a higher number of correct predictions for the "Neutral", "Up" and "Down" categories, but there are still many "Up" Neutral samples. This means that the model's judgment of the neutral state is relatively unclear. In addition, there are some differences in the number of samples of different categories, and the sample of "Neutral" category is significantly more than that of other categories.

After constantly adjusting the code parameters, our trading system got this result:

```
Total number of transactions: 6258
Number of buy trades: 3129
Number of sell trades: 3129
Cumulative return: 47.078817848164256
Winning percentage: 77.76%
Sharpe rate: 0.4966
Maximum drawdown: 0.31%
```

In this result, the total Number of transaction is 6258, which is the sum of the Number of buy trades (3129) and the number of sell trades (3129). This is because according to our trading strategy, the system sets the trading interval to 1 day and will sell all the stocks held at the trading point and buy the stocks with a rising trend.

The stocks in the data set are recorded every five minutes and 49 data points a day, so for stocks, when the trading interval of the day is reached (Window=49), a sell operation is performed and a prediction is made. Stocks that tend to rise are bought and sold the next day. Otherwise, it will not be bought until it is again predicted to rise. This is the reason why the Number of sell trades is exactly the same as Number of buy trades, and the sum of these two quantities represents the Total number of transactions.

In this result, we can also see that the Cumulative return, Sharpe rate, Maximum drawdown and other data have good results. And the trading win rate was 77.76%, which was greatly improved compared with the accuracy rate in training (68.97%).

At the same time, we also generated visualizations to show trends in the data. The Cumulative Return showed an upward trend, indicating good long-term returns.

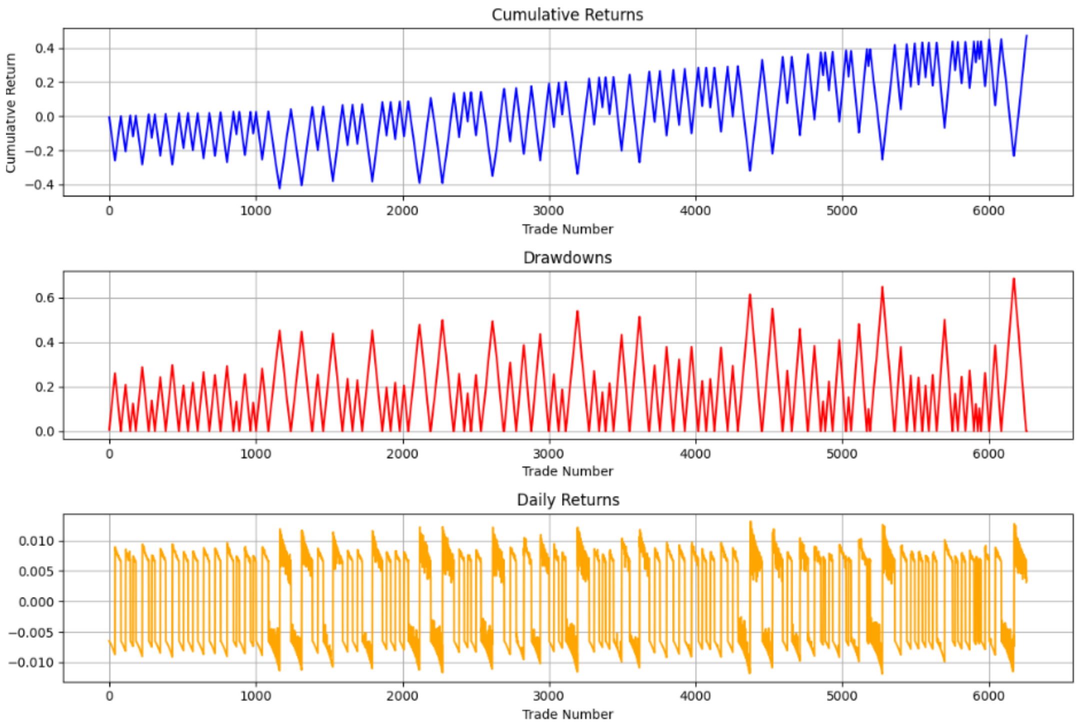

Figure 9 Trends of Data in Trading System

The line chart reflects the cumulative returns over time (per day). The horizontal axis values “0, 1000, 2000, ..., 6000” represent the time of that point in the dataset. (This is the position index; for example, the point 6000 corresponds to the time of the 6000th row of data in the Excel dataset, which is 2024-07-08.)

5998	2024-07-08 10:50:00	32.91725
5999	2024-07-08 10:55:00	32.91725
6000	2024-07-08 11:00:00	33.11261
6001	2024-07-08 11:05:00	33.01493
6002	2024-07-08 11:10:00	33.01493

8.1 Overall Accuracy

Definition:

Overall accuracy is for all categories, and it measures the percentage of the total sample that the model predicts correctly across the entire data set. It provides a rough estimate of how accurate a model is across all categories of combined predictions. This is a more intuitive global evaluation indicator, which can quickly understand the overall performance of the model.

Calculation formula:

$$\boldsymbol{Overall\ Accuracy = \frac{Correctly\ predicted\ samples\ in\ total}{Total\ \ samples}}$$

8.2 Accuracy

Definition: Accuracy refers to the percentage of the total sample that the model predicts correctly. It is an intuitive evaluation indicator used to measure the prediction correctness of the model. Can give the approximate degree to which the model is correct on all prediction samples. However, accuracy can be misleading when data categories are unbalanced. More data is therefore needed for assessment.

Calculation formula:

$$\boldsymbol{Accuracy = \frac{TP + TN}{TP + TN + FP + FN}}$$

TP: True Positive, which is the number of samples correctly predicted by the model to be positive.

TN: True Negative, which is the number of samples that the model correctly predicts to be negative.

FP: False Positive, which is the number of samples in which the model incorrectly marks false cases as positive cases.

FN: False Negative, which is the number of samples in which the model incorrectly predicts the positive class to be negative.

8.3 Precision

Definition: Accuracy refers to the proportion of samples that are actually positive among the samples that the model predicts to be positive. The accuracy rate focuses on the accuracy of the model predictions as positive. In predicting stock price movements, if we care about the "up" category (think of it as positive), high accuracy means that when the model predicts that the stock price will rise, it has a higher probability of being correct. Therefore, in the stock analysis, accuracy rate is one of the data that greatly affects the user's judgment.

Calculation formula:

$$\boldsymbol{Precision = \frac{TP}{TP + FP}}$$

8.4 Recall Rate

Definition: Recall rate, also known as recall rate, refers to the proportion of samples that are correctly predicted to be positive by the model. The recall rate focuses on the degree of coverage of the model to the positive sample. In predicting stock price movements, for the "up" category, a high recall rate indicates that the model correctly identifies most stocks that will rise.

The figure below shows the changes in accuracy and recall rates when the training model is trained on 100 stocks. Overall, the accuracy is volatile, but the trend tends to rise in the last few iterations, usually staying above 0.7. The recall rate is indicated by orange dots and lines, and the fluctuation is also large, but it is generally maintained at about 0.6, which is lower than the accuracy rate.

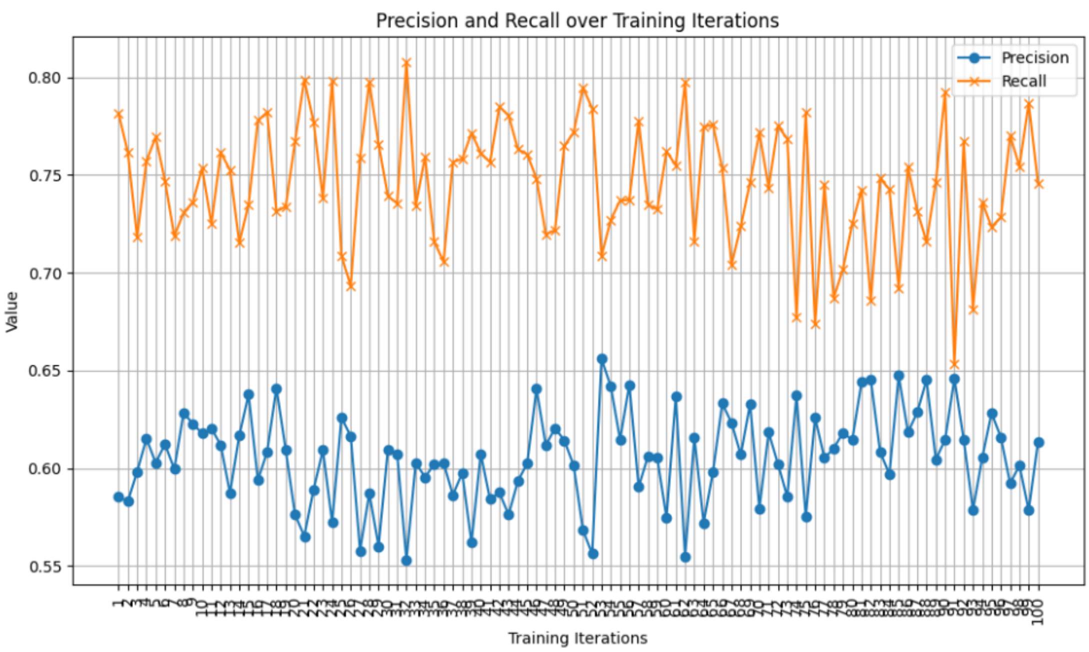

Figure 10 Precision and Recall Over Training Iterations

Calculation formula:

$$\boldsymbol{Recall} = \frac{\boldsymbol{TP}}{\boldsymbol{TP + FN}}$$

8.5 F1 score

Definition: The F1 score is the harmonic average of accuracy and recall, which takes both accuracy and recall into account to provide a more comprehensive assessment of model performance. When accuracy and recall are both important, F1 scores are a good metric. Especially in predicting stock price movements, we want the model to be as accurate as possible in predicting the rise (high accuracy), but also want it to find as many rising stocks as possible (high recall), and the F1 score balances these two aspects.

Calculation formula:

$$\boldsymbol{F1} = \frac{\boldsymbol{2 \times Precision \times Recall}}{\boldsymbol{Precision + Recall}}$$

8.6 Returns

Rate of return is a direct measure of investment return. By calculating future yields, models can help investors understand how a particular stock or asset will perform within the forecast window, allowing them to make more informed investment decisions. By analyzing the distribution of returns, investors can understand the maximum possible loss (such as the maximum retracement), to formulate the corresponding risk control strategy. In this model, by dividing the future price by the current price, the relative change multiple of the price is expressed, that is, how much the future price increases or decreases relative to the current price. The code for calculating future returns is as follows.

```
def calculate_returns(self, df: pd.DataFrame) -> pd.Series:
    """计算未来收益率"""
    future_prices = df['close'].shift(-self.prediction_window)
    current_prices = df['close']
    returns = (future_prices / current_prices - 1) * 100  # 转换为百分比
    return returns
```

8.7 Maximum retracement

Maximum retracement is an important index to measure investment risk. It reflects the maximum loss that can be suffered during the holding period. For investors, knowing the potential maximum retracement helps to assess the risk level of the investment, so as to make a more reasonable investment decision.

8.8 Sharpe ratio

It reflects the additional return that an asset can earn over and above the risk-free return if it takes unit risk. The higher the Sharpe ratio, the higher the return of the portfolio under the same risk; Or take less risk if you get the same return.

Calculation formula:

$$\boldsymbol{SharpeRatio} = \frac{\boldsymbol{E(R_P) - R_f}}{\boldsymbol{\sigma_p}}$$

$E(R_P)$ represents the expected rate of return of the portfolio, and R_f is the interest rate that an investor can get without any risk. Generally, the yield of national debt is used as the approximate representative of the risk-free interest rate. σ_p is the standard deviation of the return.

Reference

[1] S. An, ‘The Impact of Continuous Fed Rate Hikes on the Chinese Economy’, presented at the 2023 5th International Conference on Economic Management and Cultural Industry (ICEMCI 2023), Atlantis Press, Feb. 2024, pp. 765–775. doi: 10.2991/978-94-6463-368-9_91.

[2] X. Liu, ‘The Global Impact of and Response to the Fed’s Interest Rate Hike’, Highlights Bus. Econ. Manag., vol. 28, pp. 287–294, Apr. 2024, doi: 10.54097/gqe7gc61.

[3] K. Yan and Y. Li, ‘Machine learning-based analysis of volatility quantitative investment strategies for American financial stocks’, Quant. Finance Econ., vol. 8, no. 2, pp. 364–386, 2024, doi: 10.3934/QFE.2024014.

[4] W. Chen, R. Mamon, H. Xiong, and P. Zeng, ‘How do foreign investors affect China’s stock return volatility? Evidence from the Shanghai-Hong Kong Stock Connect Program’, Asia-Pac. J. Account. Econ., vol. 31, no. 1, pp. 1–24, Jan. 2024, doi: 10.1080/16081625.2022.2156360.

[5] M. R. Riyadh, S. T. Wahyudi, and F. Fadli, ‘Capital Resilience: Dynamics Movements of Large Cap and Small-Mid Cap Stock in Global Macroeconomic Uncertainty’, Futur. Econ., vol. 4, no. 4, pp. 107–120, Oct. 2024, doi: 10.57125/FEL.2024.12.25.06.

Appendix

Training Code

```
import numpy as np
import pandas as pd
from abc import ABC, abstractmethod
from enum import Enum

class Factor(Enum):
    MOMENTUM = "momentum"
    VOLATILITY = "volatility"
    TREND = "trend"
    VOLUME = "volume"
    REVERSAL = "reversal"
    RSI = "rsi"
    MACD = "macd"

class FactorCalculator(ABC):
    @abstractmethod
    def calculate(self, df: pd.DataFrame) -> float:
        pass

class MomentumFactor(FactorCalculator):
    def __init__(self, period: int = 12):
        self.period = period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period:
            return 0.0
        try:
            prices = df['close'].tail(self.period)
            returns = np.log(prices / prices.shift(1))
            weighted_return = returns.ewm(span=self.period).mean().iloc[-1]
            return float(np.tanh(weighted_return * 10))
        except Exception as e:
            print(f"动量因子计算出错: {str(e)}")
            return 0.0

class RSIFactor(FactorCalculator):
    def __init__(self, period: int = 14):
        self.period = period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period + 1:
            return 0.0
        try:
            price_diff = df['close'].diff()
            gains = price_diff.copy()
            gains[gains < 0] = 0
            losses = -price_diff.copy()
            losses[losses < 0] = 0
            avg_gains = gains.rolling(window=self.period).mean()
            avg_losses = losses.rolling(window=self.period).mean()
            rs = avg_gains.iloc[-1] / avg_losses.iloc[-1] if avg_losses.iloc[-1] != 0 else 0
            rsi = 100 - (100 / (1 + rs))
            normalized_rsi = -1.0 if rsi > 70 else (1.0 if rsi < 30 else (rsi - 50) / 20)
            return float(np.clip(normalized_rsi, -1, 1))
        except Exception as e:
            print(f"RSI因子计算出错: {str(e)}")
            return 0.0

class MACDFactor(FactorCalculator):
    def __init__(self, fast_period: int = 12, slow_period: int = 26, signal_period: int = 9):
        self.fast_period = fast_period
        self.slow_period = slow_period
        self.signal_period = signal_period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < max(self.fast_period, self.slow_period) + self.signal_period:
            return 0.0
        try:
            close_prices = df['close']
            fast_ema = close_prices.ewm(span=self.fast_period, adjust=False).mean()
            slow_ema = close_prices.ewm(span=self.slow_period, adjust=False).mean()
            macd_line = fast_ema - slow_ema
            signal_line = macd_line.ewm(span=self.signal_period, adjust=False).mean()
            macd_histogram = macd_line - signal_line
            current_macd = macd_line.iloc[-1]
            current_signal = signal_line.iloc[-1]
            current_hist = macd_histogram.iloc[-1]
            diff_score = (current_macd - current_signal) / close_prices.iloc[-1]
            hist_change = current_hist - macd_histogram.iloc[-2]
            hist_score = np.sign(hist_change) * min(abs(hist_change) / close_prices.iloc[-1], 1)
            combined_score = 0.7 * diff_score + 0.3 * hist_score
            return float(np.tanh(combined_score * 10))
        except Exception as e:
            print(f"MACD因子计算出错: {str(e)}")
            return 0.0

class VolatilityFactor(FactorCalculator):
    def __init__(self, period: int = 24):
        self.period = period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period:
            return 0.0
        try:
            returns = np.log(df['close'] / df['close'].shift(1)).tail(self.period)
            current_vol = returns.tail(self.period // 4).std()
```

```python
            hist_vol = returns.std()
            if hist_vol == 0:
                return 0.0
            vol_ratio = (current_vol / hist_vol) - 1
            return float(-np.tanh(vol_ratio * 3))
        except Exception as e:
            print(f"波动率因子计算出错: {str(e)}")
            return 0.0

class TrendFactor(FactorCalculator):
    def __init__(self, period: int = 36):
        self.period = period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period:
            return 0.0
        try:
            prices = df['close'].tail(self.period)
            ema = prices.ewm(span=self.period).mean()
            price_ratio = (prices.iloc[-1] / ema.iloc[-1]) - 1
            ema_slope = (ema.iloc[-1] / ema.iloc[-2]) - 1
            trend_score = 0.7 * price_ratio + 0.3 * ema_slope
            return float(np.tanh(trend_score * 5))
        except Exception as e:
            print(f"趋势因子计算出错: {str(e)}")
            return 0.0

class ReversalFactor(FactorCalculator):
    def __init__(self, period: int = 24):
        self.period = period
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period:
            return 0.0
        try:
            prices = df['close'].tail(self.period)
            ma = prices.mean()
            deviation = (prices.iloc[-1] / ma) - 1
            returns = prices.pct_change()
            pos_returns = returns[returns > 0].sum()
            neg_returns = abs(returns[returns < 0].sum())
            if pos_returns + neg_returns == 0:
                rsi = 0.5
            else:
                rsi = pos_returns / (pos_returns + neg_returns)
            reversal_score = -0.6 * deviation - 0.4 * (rsi - 0.5) * 2
            return float(np.clip(reversal_score, -1, 1))
        except Exception as e:
            print(f"反转因子计算出错: {str(e)}")
            return 0.0

class VolumeFactor(FactorCalculator):
    def __init__(self, period: int = 18):
        self.period = period
    def _process_volume(self, volumes: pd.Series) -> pd.Series:
        volumes = volumes.replace(0, np.nan)
        volumes = volumes.fillna(method='ffill').fillna(method='bfill')
        Q1 = volumes.quantile(0.25)
        Q3 = volumes.quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 2.0 * IQR
        upper_bound = Q3 + 2.0 * IQR
        volumes = np.clip(volumes, lower_bound, upper_bound)
        volumes = np.log1p(volumes)
        vol_min = volumes.min()
        vol_max = volumes.max()
        if vol_max > vol_min:
            volumes = 2 * (volumes - vol_min) / (vol_max - vol_min) - 1
        return volumes
    def calculate(self, df: pd.DataFrame) -> float:
        if len(df) < self.period:
            return 0.0
        try:
            window_data = df.tail(self.period).copy()
            volumes = self._process_volume(window_data['volume'])
            current_vol = volumes.iloc[-1]
            avg_vol = volumes.iloc[:-1].mean()
            vol_ratio = (current_vol - avg_vol) if avg_vol != 0 else 0
            returns = window_data['close'].pct_change()
            abs_returns = returns.abs()
            current_ratio = (volumes.iloc[-1] / abs_returns.iloc[-1]) if abs_returns.iloc[-1] != 0 else 0
            hist_ratio = (volumes.iloc[:-1] / abs_returns.iloc[:-1]).mean()
            price_vol_ratio = (current_ratio - hist_ratio) if hist_ratio != 0 else 0
            score = 0.6 * np.tanh(vol_ratio * 2) + 0.4 * np.tanh(price_vol_ratio * 2)
            return float(np.clip(score, -1, 1))
        except Exception as e:
            print(f"成交量因子计算出错: {str(e)}")
            return 0.0
```

```python
# data_processor_copy.py
import pandas as pd
from typing import Dict
import warnings
warnings.filterwarnings('ignore')

def process_raw_data(df: pd.DataFrame) -> Dict[str, pd.DataFrame]:
    stock_dict = {}
    try:
        stock_columns = [col for col in df.columns.levels[0]
                         if isinstance(col, str) and col.startswith('STOCK_')]
        print(f"发现股票数量: {len(stock_columns)}")
        for stock in stock_columns:
            try:
                stock_df = pd.DataFrame({
                    'close': pd.to_numeric(df[stock]['close_px'], errors='coerce'),
                    'high': pd.to_numeric(df[stock]['high_px'], errors='coerce'),
                    'low': pd.to_numeric(df[stock]['low_px'], errors='coerce'),
                    'open': pd.to_numeric(df[stock]['open_px'], errors='coerce'),
                    'volume': pd.to_numeric(df[stock]['volume'], errors='coerce')
                }, index=df.index)
                stock_df = stock_df.dropna()
                if not stock_df.empty:
                    stock_num = stock.split('_')[1]
                    stock_dict[stock_num] = stock_df
            except Exception as e:
                print(f"处理股票 {stock} 时出错: {str(e)}")
                continue
    except Exception as e:
        print(f"数据处理过程出错: {str(e)}")
        raise
    print(f"处理完成，共成功处理 {len(stock_dict)} 只股票的数据")
    return stock_dict

def filter_stocks(data_dict: Dict[str, pd.DataFrame],
                  min_volume: float = 100,
                  min_price: float = 1.0,
                  max_volatility: float = 0.05) -> Dict[str, pd.DataFrame]:
    filtered_stocks = {}
    for code, df in data_dict.items():
        try:
            volume_ma = df['volume'].rolling(window=3).mean()
            returns = df['close'].pct_change()
            volatility = returns.rolling(window=6).std()
            if (volume_ma.mean() >= min_volume and
                    df['close'].mean() >= min_price and
                    volatility.mean() <= max_volatility):
                filtered_stocks[code] = df
        except Exception as e:
            print(f"处理股票 {code} 时出错: {str(e)}")
            continue
    print(f"筛选后剩余股票数量: {len(filtered_stocks)}")
    return filtered_stocks
```

```python
# factor_trainer_test.py
import pandas as pd
import numpy as np
from typing import Dict, Tuple
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import TimeSeriesSplit
import xgboost as xgb
from factors.base_copy import Factor, FactorCalculator
from data_processor_copy import process_raw_data, filter_stocks
from return_analysis import ReturnAnalyzer
import os
import json
import joblib
import logging
from datetime import datetime
from typing import Dict, Tuple, Optional
import xgboost as xgb
from sklearn.preprocessing import StandardScaler

class ModelManager:
    def __init__(self, base_path="models"):
        self.base_path = base_path
        os.makedirs(base_path, exist_ok=True)
    def _get_model_names(self, timestamp: str) -> Dict[str, str]:
        return {
            "model": f"model_{timestamp}.joblib",
            "scaler": f"scaler_{timestamp}.joblib",
            "metrics": f"metrics_{timestamp}.json"
        }
    def save_model(self, stock_code: str, model: xgb.XGBClassifier,
                   scaler: StandardScaler, metrics: dict) -> bool:
        try:
            stock_path = os.path.join(self.base_path, f"stock_{stock_code}")
            os.makedirs(stock_path, exist_ok=True)
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            file_names = self._get_model_names(timestamp)
            model_path = os.path.join(stock_path, file_names["model"])
            joblib.dump(model, model_path, protocol=4)
            scaler_path = os.path.join(stock_path, file_names["scaler"])
            joblib.dump(scaler, scaler_path, protocol=4)
            metrics_path = os.path.join(stock_path, file_names["metrics"])
            with open(metrics_path, 'w', encoding='utf-8') as f:
                json.dump(metrics, f, ensure_ascii=False, indent=4)
            latest_path = os.path.join(stock_path, "latest.json")
            latest_info = {
                "timestamp": timestamp,
                "model_name": file_names["model"],
                "scaler_name": file_names["scaler"],
                "metrics_name": file_names["metrics"]
            }
            with open(latest_path, 'w', encoding='utf-8') as f:
                json.dump(latest_info, f, ensure_ascii=False, indent=4)
            return True
        except Exception as e:
            print(f"保存股票 {stock_code} 的模型失败：{str(e)}")
            return False
    def load_model(self, stock_code: str,
                   timestamp: str = None) -> Optional[Tuple[xgb.XGBClassifier, StandardScaler, dict]]:
        try:
            stock_path = os.path.join(self.base_path, f"stock_{stock_code}")
            if not os.path.exists(stock_path):
                print(f"股票 {stock_code} 的模型文件夹不存在")
                return None
            if timestamp is None:
                latest_path = os.path.join(stock_path, "latest.json")
                if not os.path.exists(latest_path):
                    print(f"股票 {stock_code} 的latest.json不存在")
                    return None
                with open(latest_path, 'r', encoding='utf-8') as f:
                    latest_info = json.load(f)
                    timestamp = latest_info["timestamp"]
                    model_name = latest_info["model_name"]
                    scaler_name = latest_info["scaler_name"]
                    metrics_name = latest_info["metrics_name"]
            else:
                file_names = self._get_model_names(timestamp)
                model_name = file_names["model"]
                scaler_name = file_names["scaler"]
                metrics_name = file_names["metrics"]
            model_path = os.path.join(stock_path, model_name)
            scaler_path = os.path.join(stock_path, scaler_name)
            metrics_path = os.path.join(stock_path, metrics_name)
            if not all(os.path.exists(p) for p in [model_path, scaler_path, metrics_path]):
                print(f"股票 {stock_code} 的某些模型文件不存在")
                return None
            print(f"正在加载股票 {stock_code} 的模型文件...")
            model = joblib.load(model_path)
            print(f"模型加载成功")
            scaler = joblib.load(scaler_path)
            print(f"标准化器加载成功")
            with open(metrics_path, 'r', encoding='utf-8') as f:
                metrics = json.load(f)
            print(f"指标加载成功")
            return model, scaler, metrics
        except Exception as e:
            print(f"加载股票 {stock_code} 的模型时发生错误：{str(e)}")
            return None
    def list_models(self, stock_code: str) -> list:
        stock_path = os.path.join(self.base_path, f"stock_{stock_code}")
        if not os.path.exists(stock_path):
```

```python
        if not os.path.exists(stock_path):
            return []
        models = []
        for file in os.listdir(stock_path):
            if file.startswith("metrics_"):
                timestamp = file.replace("metrics_", "").replace(".json", "")
                try:
                    with open(os.path.join(stock_path, file), 'r', encoding='utf-8') as f:
                        metrics = json.load(f)
                    models.append({
                        "timestamp": timestamp,
                        "accuracy": metrics.get("overall_accuracy", 0),
                        "up_prediction": metrics.get("up_prediction", {})
                    })
                except Exception as e:
                    print(f"读取股票 {stock_code} 的模型信息时出错: {str(e)}")
                    continue

        return sorted(models, key=lambda x: x["timestamp"], reverse=True)

class XGBoostFactorModel:
    def __init__(self,
                 train_window: int = 120,
                 pred_window: int = 49,
                 n_splits: int = 5):
        self.train_window = train_window
        self.pred_window = pred_window
        self.n_splits = n_splits
        self.scaler = StandardScaler()
        self.return_analyzer = ReturnAnalyzer(prediction_window=pred_window)

    def _prepare_data(self, df: pd.DataFrame,
                      factors: Dict[Factor, FactorCalculator],
                      stock_code: str) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
        warmup_periods = {
            'momentum': 120,
            'volatility': 240,
            'trend': 300,
            'rsi': 144,
            'macd': max(24 + 52 + 18, 52),
            'reversal': 240
        }
        max_warmup = max(warmup_periods.values())
        if len(df) < max_warmup + self.pred_window:
            raise ValueError(f"数据长度不足, 需要至少 {max_warmup + self.pred_window} 个时间点")
        analysis_result = self.return_analyzer.analyze_single_stock(df, stock_code)
        if analysis_result is None:
            raise ValueError(f"股票 {stock_code} 数据分析失败")
        factor_values = {}
        for factor_name, calculator in factors.items():
            try:
                signals = []
                for i in range(max_warmup, len(df) - self.pred_window):
                    historical_df = df.iloc[:i + 1]
                    signal = calculator.calculate(historical_df)
                    signals.append(signal)
                warmup_signals = [np.nan] * max_warmup
                factor_values[factor_name.value] = warmup_signals + signals
            except Exception as e:
                print(f"计算因子 {factor_name.value} 出错: {str(e)}")
                continue
        X = pd.DataFrame(factor_values)
        returns = self.return_analyzer.calculate_returns(df)
        thresholds = analysis_result['stats']['thresholds']
        y = pd.Series([self.return_analyzer.get_return_class(r, thresholds=thresholds)
                       for r in returns.iloc[:-self.pred_window]])
        valid_data_start = max_warmup
        X = X.iloc[valid_data_start:]
        y = y.iloc[valid_data_start:]
        valid_mask = ~(X.isna().any(axis=1) | y.isna())
        X = X[valid_mask]
        y = y[valid_mask]
        class_counts = y.value_counts()
        total_samples = len(y)
        class_weights = {cls: total_samples / (count * len(class_counts))
                         for cls, count in class_counts.items()}
        sample_weights = np.array([class_weights[label] for label in y])
        return X.values, y.values, sample_weights
    def _train_model(self, X: np.ndarray, y: np.ndarray, sample_weights: np.ndarray) -> xgb.XGBClassifier:
        X_scaled = self.scaler.fit_transform(X)
        tscv = TimeSeriesSplit(n_splits=self.n_splits)
        model = xgb.XGBClassifier(
            learning_rate=0.05,
            n_estimators=500,
            max_depth=6,
            subsample=0.9,
            colsample_bytree=1.0,
            min_child_weight=2,
            gamma=0.1,
            reg_alpha=0.1,
            reg_lambda=1,
            objective='multi:softmax',
            num_class=5,
            eval_metric=['mlogloss', 'merror'],
            use_label_encoder=False,
            random_state=42
        )
        eval_results = {}
        class_names = {
            0: "极端下跌",
```

```python
            0: "极端下跌",
            1: "下跌",
            2: "震荡",
            3: "上涨",
            4: "极端上涨"
        }
        for fold, (train_idx, val_idx) in enumerate(tscv.split(X_scaled)):
            X_train, X_val = X_scaled[train_idx], X_scaled[val_idx]
            y_train, y_val = y[train_idx], y[val_idx]
            weights_train = sample_weights[train_idx]
            print(f"\nFold {fold + 1}:")
            class_dist_train = pd.Series(y_train).value_counts()
            class_dist_val = pd.Series(y_val).value_counts()
            print("\n训练集类别分布:")
            for cls in sorted(class_dist_train.index):
                print(f"{class_names[cls]}: {class_dist_train[cls]}")
            print("\n验证集类别分布:")
            for cls in sorted(class_dist_val.index):
                print(f"{class_names[cls]}: {class_dist_val[cls]}")
            model.fit(
                X_train, y_train,
                sample_weight=weights_train,
                eval_set=[(X_val, y_val)],
                verbose=True
            )
        return model
    def _get_feature_importance(self, model: xgb.XGBClassifier,
                                factors: Dict[Factor, FactorCalculator]) -> Dict[str, float]:
        importance_dict = {}
        for factor_name, importance in zip(factors.keys(), model.feature_importances_):
            importance_dict[factor_name.value] = float(importance)
        return importance_dict

class FactorTrainer:
    def __init__(self, prediction_window: int = 49):
        self.prediction_window = prediction_window
        self.xgb_model = XGBoostFactorModel(pred_window=prediction_window)
        self.model_manager = ModelManager()
    def analyze_factor_metrics(self, model: xgb.XGBClassifier, X: np.ndarray, y: np.ndarray,
                               factors: Dict[Factor, FactorCalculator]) -> Dict[str, Dict[str, float]]:
        X_scaled = self.xgb_model.scaler.transform(X)
        predictions = model.predict(X_scaled)
        class_names = {
            0: "极端下跌",
            1: "下跌",
            2: "震荡",
            3: "上涨",
            4: "极端上涨"
        }
        class_metrics = {}
        for class_idx in range(5):
            class_mask = y == class_idx
            if np.any(class_mask):
                class_pred = predictions[class_mask]
                accuracy = np.mean(class_pred == class_idx)
                precision = np.sum((predictions == class_idx) & (y == class_idx)) / \
                            (np.sum(predictions == class_idx) + 1e-10)
                recall = np.sum((predictions == class_idx) & (y == class_idx)) / \
                         (np.sum(y == class_idx) + 1e-10)
                f1 = 2 * (precision * recall) / (precision + recall + 1e-10)

                class_metrics[class_names[class_idx]] = {
                    'accuracy': accuracy,
                    'precision': precision,
                    'recall': recall,
                    'f1': f1
                }
        overall_accuracy = np.mean(predictions == y)
        up_mask = (y == 3) | (y == 4)
        pred_up_mask = (predictions == 3) | (predictions == 4)
        up_precision = np.sum((pred_up_mask) & (up_mask)) / (np.sum(pred_up_mask) + 1e-10)
        up_recall = np.sum((pred_up_mask) & (up_mask)) / (np.sum(up_mask) + 1e-10)
        up_f1 = 2 * (up_precision * up_recall) / (up_precision + up_recall + 1e-10)
        feature_importance = self.xgb_model._get_feature_importance(model, factors)
        return {
            'feature_importance': feature_importance,
            'class_metrics': class_metrics,
            'overall_accuracy': overall_accuracy,
            'up_prediction': {
                'precision': up_precision,
                'recall': up_recall,
                'f1': up_f1
            }
        }
    def train(self, file_path: str, factors: Dict[Factor, FactorCalculator]) -> Dict[str, Dict]:
        print("\n开始训练五分类XGBoost模型...")
        print("读取数据...")
        df = pd.read_excel(file_path, header=[0, 1]) if file_path.endswith('.xlsx') else \
            pd.read_csv(file_path, header=[0, 1])
        stock_dict = process_raw_data(df)
        filtered_stocks = filter_stocks(stock_dict)
        all_metrics = {}
        sorted_stock_codes = sorted(filtered_stocks.keys(), key=lambda x: int(x))
        for stock_code in sorted_stock_codes:
            stock_data = filtered_stocks[stock_code]
            print(f"\n处理股票 {stock_code}...")
            try:
                X, y, sample_weights = self.xgb_model._prepare_data(stock_data, factors, stock_code)

                if len(X) == 0 or len(y) == 0:
```

```python
                if len(X) == 0 or len(y) == 0:
                    print(f"股票 {stock_code} 没有有效数据")
                    continue
                unique_classes = np.unique(y)
                if len(unique_classes) < 2:
                    print(f"股票 {stock_code} 的数据只包含单一类别 {unique_classes}, 跳过训练")
                    continue
                try:
                    print("训练XGBoost模型...")
                    model = self.xgb_model._train_model(X, y, sample_weights)
                    metrics = self.analyze_factor_metrics(model, X, y, factors)
                    all_metrics[stock_code] = metrics
                    self.model_manager.save_model(
                        stock_code,
                        model,
                        self.xgb_model.scaler,
                        metrics
                    )
                    print(f"\n股票 {stock_code} 的模型表现:")
                    print(f"总体准确率: {metrics['overall_accuracy']:.2%}")
                    print("\n上涨预测性能:")
                    up_pred = metrics['up_prediction']
                    print(f"精确率: {up_pred['precision']:.4f}")
                    print(f"召回率: {up_pred['recall']:.4f}")
                    print(f"F1分数: {up_pred['f1']:.4f}")
                    print("\n各类别指标:")
                    for class_name, class_metric in metrics['class_metrics'].items():
                        print(f"\n{class_name}:")
                        for metric_name, value in class_metric.items():
                            print(f"{metric_name}: {value:.4f}")
                    print("\n特征重要性:")
                    for factor, importance in metrics['feature_importance'].items():
                        print(f"{factor}: {importance:.4f}")
                except Exception as e:
                    print(f"模型训练出错: {str(e)}")
                    continue
            except Exception as e:
                print(f"处理股票 {stock_code} 时出错: {str(e)}")
                continue
        return all_metrics

if __name__ == "__main__":
    from factors.base_copy import (MomentumFactor, VolatilityFactor, TrendFactor,
                                   RSIFactor, MACDFactor, ReversalFactor)
    factors = {
        Factor.MOMENTUM: MomentumFactor(period=120),
        Factor.VOLATILITY: VolatilityFactor(period=240),
        Factor.TREND: TrendFactor(period=300),
        Factor.RSI: RSIFactor(period=144),
        Factor.MACD: MACDFactor(
            fast_period=24,
            slow_period=52,
            signal_period=18
        ),
        Factor.REVERSAL: ReversalFactor(period=240)  # 20小时
    }
    trainer = FactorTrainer(prediction_window=49)
    metrics = trainer.train("./train.xlsx", factors)
```

```python
# return_analysis.py
import pandas as pd
import numpy as np
from typing import Dict, Tuple, List
from data_processor import process_raw_data, filter_stocks
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict

class ReturnAnalyzer:
    def __init__(self, prediction_window: int = 49):
        self.prediction_window = prediction_window
        self.threshold_cache = {}
    def calculate_returns(self, df: pd.DataFrame) -> pd.Series:
        future_prices = df['close'].shift(-self.prediction_window)
        current_prices = df['close']
        returns = (future_prices / current_prices - 1) * 100
        return returns
    def get_dynamic_thresholds(self, returns: pd.Series) -> Dict[str, float]:
        thresholds = {
            'extreme_down': np.percentile(returns, 10),
            'down': np.percentile(returns, 30),
            'up': np.percentile(returns, 70),
            'extreme_up': np.percentile(returns, 90)
        }
        return thresholds
    def get_return_class(self, return_value: float, stock_code: str = None,
                         thresholds: Dict[str, float] = None) -> int:
        if thresholds is None:
            if stock_code is None or stock_code not in self.threshold_cache:
                raise ValueError("必须提供thresholds或有效的stock_code")
            thresholds = self.threshold_cache[stock_code]
        if return_value <= thresholds['extreme_down']:
            return 0
        elif return_value <= thresholds['down']:
            return 1
        elif return_value <= thresholds['up']:
            return 2
        elif return_value <= thresholds['extreme_up']:
            return 3
        else:
            return 4
    def analyze_single_stock(self, df: pd.DataFrame, stock_code: str) -> Dict:
        returns = self.calculate_returns(df)
        returns = returns.dropna()
        if len(returns) == 0:
            return None
        thresholds = self.get_dynamic_thresholds(returns)
        self.threshold_cache[stock_code] = thresholds
        stats = {
            'count': len(returns),
            'mean': returns.mean(),
            'std': returns.std(),
            'min': returns.min(),
            'max': returns.max(),
            'skew': returns.skew(),
            'kurt': returns.kurtosis(),
            'thresholds': thresholds,
            'percentiles': {
                '1%': np.percentile(returns, 1),
                '5%': np.percentile(returns, 5),
                '10%': np.percentile(returns, 10),
                '25%': np.percentile(returns, 25),
                '50%': np.percentile(returns, 50),
                '75%': np.percentile(returns, 75),
                '90%': np.percentile(returns, 90),
                '95%': np.percentile(returns, 95),
                '99%': np.percentile(returns, 99),
            }
        }
        labels = ['极端下跌', '下跌', '震荡', '上涨', '极端上涨']
        class_counts = [0] * 5
        for ret in returns:
            class_idx = self.get_return_class(ret, thresholds=thresholds)
            class_counts[class_idx] += 1
        distribution = {label: count for label, count in zip(labels, class_counts)}
        distribution_pct = {label: count / len(returns) * 100
                            for label, count in distribution.items()}
        return {
            'stats': stats,
            'distribution': distribution,
            'distribution_pct': distribution_pct,
            'returns': returns
        }
    def analyze_stock_returns(self, stock_dict: Dict[str, pd.DataFrame]) -> Dict[str, Dict]:
        analysis_results = {}
        for stock_code, df in stock_dict.items():
            result = self.analyze_single_stock(df, stock_code)
            if result is not None:
                analysis_results[stock_code] = result
        return analysis_results
    def print_analysis_results(self, results: Dict[str, Dict]):
        print("\n=== 股票收益率分析结果 ===")
```

```python
        print("\n=== 股票收益率分析结果 ===")
        total_distribution = defaultdict(int)
        total_count = 0
        for stock_code, result in sorted(results.items(), key=lambda x: int(x[0])):
            print(f"\n股票代码: {stock_code}")
            stats = result['stats']
            print("\n基本统计量:")
            print(f"样本数: {stats['count']}")
            print(f"平均收益率: {stats['mean']:.2f}%")
            print(f"标准差: {stats['std']:.2f}%")
            print(f"最小值: {stats['min']:.2f}%")
            print(f"最大值: {stats['max']:.2f}%")
            print(f"偏度: {stats['skew']:.2f}")
            print(f"峰度: {stats['kurt']:.2f}")
            print("\n分位数:")
            percentiles = stats['percentiles']
            for pct, value in percentiles.items():
                print(f"{pct}: {value:.2f}%")
            print("\n动态阈值:")
            thresholds = stats['thresholds']
            for threshold_name, value in thresholds.items():
                print(f"{threshold_name}: {value:.2f}%")
            print("\n收益率分布:")
            distribution_pct = result['distribution_pct']
            for label, percentage in distribution_pct.items():
                count = result['distribution'][label]
                print(f"{label}: {percentage:.2f}% ({count}个样本)")
                total_distribution[label] += count
                total_count += count
        print("\n=== 所有股票的综合分布情况 ===")
        for label, count in total_distribution.items():
            percentage = (count / total_count) * 100
            print(f"{label}: {percentage:.2f}% ({count}个样本)")
    def plot_return_distribution(self, returns: pd.Series, thresholds: Dict[str, float],
                                 title: str = "收益率分布"):
        plt.figure(figsize=(12, 6))
        sns.histplot(returns, bins=50, kde=True)
        colors = ['red', 'orange', 'green', 'orange', 'red']
        for (name, value), color in zip(thresholds.items(), colors):
            plt.axvline(x=value, color=color, linestyle='--',
                        label=f'{name}: {value:.2f}%')
        plt.title(title)
        plt.xlabel("收益率 (%)")
        plt.ylabel("频数")
        plt.legend()
        plt.grid(True)
        plt.show()
    def save_thresholds(self, filename: str):
        threshold_df = pd.DataFrame.from_dict(self.threshold_cache, orient='index')
        threshold_df.to_csv(filename)
    def load_thresholds(self, filename: str):
        threshold_df = pd.read_csv(filename, index_col=0)
        self.threshold_cache = threshold_df.to_dict('index')

def main():
    print("读取数据...")
    df = pd.read_excel("train.xlsx", header=[0, 1])
    print("处理数据...")
    stock_dict = process_raw_data(df)
    filtered_stocks = filter_stocks(stock_dict)
    print("分析收益率...")
    analyzer = ReturnAnalyzer()
    results = analyzer.analyze_stock_returns(filtered_stocks)
    analyzer.print_analysis_results(results)
    analyzer.save_thresholds("stock_thresholds.csv")
    stock_code = '1'
    if stock_code in results:
        analyzer.plot_return_distribution(
            results[stock_code]['returns'],
            results[stock_code]['stats']['thresholds'],
            f"股票 {stock_code} 收益率分布"
        )

if __name__ == "__main__":
    main()
```

Trading Code

```python
from datetime import datetime
import logging
import numpy as np
from trading_system import TradingSystem
import json
import os
import matplotlib.pyplot as plt
import pandas as pd

def setup_logging():
    if not os.path.exists('logs'):
        os.makedirs('logs')
    log_filename = f'logs/trading_{datetime.now().strftime("%Y%m%d_%H%M")}.log'
    log_format = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    logging.basicConfig(
        level=logging.INFO,
        format=log_format,
        handlers=[
            logging.FileHandler(log_filename, encoding='utf-8'),
            logging.StreamHandler()
        ]
    )
    return logging.getLogger(__name__)

def save_results(results: dict, filename: str):
    logger = logging.getLogger(__name__)
    try:
        if "final_metrics" not in results:
            logger.warning("结果中缺少final_metrics, 添加空字典")
            results["final_metrics"] = {}
        def convert_numpy(obj):
            if isinstance(obj, np.integer):
                return int(obj)
            elif isinstance(obj, np.floating):
                return float(obj)
            elif isinstance(obj, np.ndarray):
                return obj.tolist()
            return obj
        results_converted = json.loads(
            json.dumps(results, default=convert_numpy)
        )
        with open(filename, 'w', encoding='utf-8') as f:
            json.dump(results_converted, f, ensure_ascii=False, indent=4)
        logger.info(f"结果已保存至: {filename}")
    except Exception as e:
        logger.error(f"保存结果时出错: {str(e)}")

def print_trading_summary(summary: dict):
    print("\n=== 当日交易摘要 ===")
    print(f"总资产: {summary['total_capital']:,.2f}")
    print(f"当前持仓比例: {summary['position_ratio'] * 100:.2f}%")
    print(f"今日交易数: {summary['trades']['total_trades']}")
    if summary['trades']['trades_detail']:
        print("\n交易详情:")
        for trade in summary['trades']['trades_detail']:
            action = "买入" if trade['action'] == "buy" else "卖出"
            print(f"股票 {trade['stock_code']}: {action} {trade['shares']}股, "
                  f"价格: {trade['price']:.2f}, 金额: {trade['amount']:,.2f}")

def print_performance_metrics(metrics: dict):
    print("\n=== 绩效指标 ===")
    for metric, value in metrics.items():
        print(f"{metric}: {value}")


def calculate_performance_metrics(trade_history):
    cumulative_returns = []
    daily_returns = []
    total_capital = 10000000
    previous_capital = total_capital
    for trade in trade_history:
        if trade.action == 'buy':
            total_capital -= trade.price * trade.shares
        else:
            total_capital += trade.price * trade.shares
        daily_return = (total_capital - previous_capital) / previous_capital if previous_capital else 0
        daily_returns.append(daily_return)
        previous_capital = total_capital
        cumulative_return = (total_capital - 10000000) / 10000000
        cumulative_returns.append(cumulative_return)
    return cumulative_returns, daily_returns

def plot_performance(cumulative_returns, daily_returns):
    plt.figure(figsize=(12, 8))
    plt.subplot(3, 1, 1)
    plt.plot(cumulative_returns, color='blue')
    plt.title('Cumulative Returns')
    plt.xlabel('Trade Number')
    plt.ylabel('Cumulative Return')
    plt.grid()
    plt.subplot(3, 1, 2)
    if cumulative_returns:
        max_so_far = 0
```

```python
            drawdowns = []
            for i in range(len(cumulative_returns)):
                max_so_far = max(max_so_far, cumulative_returns[i])
                drawdown = max_so_far - cumulative_returns[i]
                drawdowns.append(drawdown)
            plt.plot(drawdowns, color='red')
            plt.title('Drawdowns')
        else:
            plt.title('Drawdowns')
            plt.ylabel('Drawdown')
            plt.text(0.5, 0.5, 'No data available.', horizontalalignment='center', verticalalignment='center', fontsize=12)
        plt.xlabel('Trade Number')
        plt.grid()
        plt.subplot(3, 1, 3)
        plt.plot(daily_returns, color='orange')
        plt.title('Daily Returns')
        plt.xlabel('Trade Number')
        plt.ylabel('Daily Return')
        plt.grid()
        plt.tight_layout()
        plt.show()

def main():
    logger = setup_logging()
    try:
        params = {
            'data_file': 'test.xlsx',
            'initial_capital': 10000000,
            'model_path': 'models',
            'extreme_up_weight': 1.5,
            'up_weight': 1.0,
            'max_single_position': 0.0065,
            'max_total_position': 0.6
        }
        logger.info("初始化交易系统...")
        trading_system = TradingSystem(**params)
        if not trading_system.initialize():
            logger.error("交易系统初始化失败")
            return
        logger.info("开始执行交易策略...")
        results = trading_system.execute_trading()
        cumulative_returns, daily_returns = calculate_performance_metrics(trading_system.trade_history)
        plot_performance(cumulative_returns, daily_returns)
        summary = {
            "总交易次数": len(trading_system.trade_history),
            "买入交易次数": sum(1 for order in trading_system.trade_history if order.action == "buy"),
            "卖出交易次数": sum(1 for order in trading_system.trade_history if order.action == "sell"),
        }
        if "final_metrics" not in results:
            results["final_metrics"] = {}
        if "error" in results:
            logger.error(f"交易执行失败: {results['error']}")
            return
        if "final_metrics" not in results:
            logger.error("结果中缺少final_metrics")
            return
        metrics = results["final_metrics"]
        print("\n=== 交易统计总结 ===")
        print(f"可用指标: {list(metrics.keys())}")
        if "总交易次数" not in metrics:
            logger.warning("无法获取交易次数指标, 交易历史可能为空")
        print(f"Total number of transactions: {metrics.get('总交易次数', '未知')}")
        print(f"Number of buy trades: {metrics.get('买入交易次数', '未知')}")
        print(f"Number of sell trades: {metrics.get('卖出交易次数', '未知')}")
        print(f"Cumulative return: {metrics.get('累计收益率', '未知')}")
        if '胜率' in metrics:
            print(f"Winning percentage: {metrics['胜率']}")
        if '夏普比率' in metrics:
            print(f"Sharpe rate: {metrics['夏普比率']}")
            print(f"Maximum drawdown: {metrics.get('最大回撤', '未知')}")
        if os.path.exists('results'):
            timestamp = datetime.now().strftime("%Y%m%d_%H%M")
            results_filename = f'results/trading_results_{timestamp}.json'
            save_results(results, results_filename)
        else:
            logger.error("results目录不存在")
    except Exception as e:
        logger.error(f"程序执行出错: {str(e)}", exc_info=True)

if __name__ == "__main__":
    main()
```

```python
#trading_system.py
from datetime import datetime
from dataclasses import dataclass
from typing import Dict, List, Tuple, Optional
import pandas as pd
import numpy as np
import logging
from enum import Enum
from trading_data_manager import TradingDataManager, TradingPoint
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

@dataclass
class TradingPoint:
    position: int
    historical_data: pd.DataFrame
    current_price: float
    future_price: float

class PredictionResult(Enum):
    EXTREME_DOWN = 0
    DOWN = 1
    NEUTRAL = 2
    UP = 3
    EXTREME_UP = 4

@dataclass
class StockPrediction:
    stock_code: str
    prediction: PredictionResult
    confidence: float
    position: int
    class TradingDataManager:
        def __init__(self, warmup_period: int = 300, prediction_window: int = 49):
            self.warmup_period = warmup_period
            self.prediction_window = prediction_window
            self.logger = logging.getLogger(__name__)
    def prepare_trading_sequence(self,
                                 stock_data: Dict[str, pd.DataFrame]
                                 ) -> Dict[str, List[TradingPoint]]:
        trading_sequences = {}
        for stock_code, df in stock_data.items():
            try:
                sequence = []
                for pos in range(self.warmup_period,
                                 len(df) - self.prediction_window,
                                 self.prediction_window):
                    historical_data = df.iloc[pos - self.warmup_period:pos + 1].copy()
                    historical_data = historical_data.reset_index(drop=True)
                    current_price = float(df.iloc[pos]['close'])
                    future_price = float(df.iloc[pos + self.prediction_window]['close'])
                    trading_point = TradingPoint(
                        position=pos,
                        historical_data=historical_data,
                        current_price=current_price,
                        future_price=future_price
                    )
                    sequence.append(trading_point)
                if sequence:
                    trading_sequences[stock_code] = sequence
                    self.logger.info(f"股票 {stock_code} 生成了 {len(sequence)} 个交易点")
            except Exception as e:
                self.logger.error(f"处理股票 {stock_code} 时出错: {str(e)}")
                continue
        return trading_sequences
    def get_next_trading_point(self,
                               trading_sequences: Dict[str, List[TradingPoint]],
                               current_position: Optional[int] = None
                               ) -> Dict[str, Optional[TradingPoint]]:
        next_points = {}
        if current_position is None:
            for stock_code, sequence in trading_sequences.items():
                next_points[stock_code] = sequence[0] if sequence else None
            return next_points
        for stock_code, sequence in trading_sequences.items():
            if not sequence:
                next_points[stock_code] = None
                continue
            try:
                current_index = next(
                    (i for i, point in enumerate(sequence)
                     if point.position == current_position),
                    None
                )
                if current_index is None:
                    next_points[stock_code] = None
                    continue
                next_index = current_index + 1
                if next_index < len(sequence):
                    next_points[stock_code] = sequence[next_index]
                else:
                    next_points[stock_code] = None
            except Exception as e:
                self.logger.error(f"获取股票 {stock_code} 下一个点时出错: {str(e)}")
                next_points[stock_code] = None
        return next_points


import xgboost as xgb
from sklearn.preprocessing import StandardScaler
```

```python
from sklearn.preprocessing import StandardScaler
from base import (Factor, FactorCalculator, MomentumFactor, VolatilityFactor,
                  TrendFactor, RSIFactor, MACDFactor, ReversalFactor)
from factor_trainer import ModelManager

class PredictionManager:
    def __init__(self, model_path: str = "models"):
        self.model_manager = ModelManager(model_path)
        self.models: Dict[str, Tuple[xgb.XGBClassifier, StandardScaler]] = {}
        self.factors: Dict[Factor, FactorCalculator] = {
            Factor.MOMENTUM: MomentumFactor(period=120),
            Factor.VOLATILITY: VolatilityFactor(period=240),
            Factor.TREND: TrendFactor(period=300),
            Factor.RSI: RSIFactor(period=144),
            Factor.MACD: MACDFactor(
                fast_period=24,
                slow_period=52,
                signal_period=18
            ),
            Factor.REVERSAL: ReversalFactor(period=240)
        }
        self.logger = logging.getLogger(__name__)
    def load_models(self, stock_codes: List[str]):
        sorted_codes = sorted(stock_codes, key=lambda x: int(x))
        for stock_code in sorted_codes:
            try:
                result = self.model_manager.load_model(stock_code)
                if result is not None:
                    model, scaler, _ = result
                    self.models[stock_code] = (model, scaler)
            except Exception as e:
                self.logger.error(f"加载股票 {stock_code} 的模型时发生错误: {str(e)}")
                continue
    def prepare_features(self, df: pd.DataFrame) -> Optional[np.ndarray]:
        try:
            factor_values = {}
            for factor_name, calculator in self.factors.items():
                try:
                    signal = calculator.calculate(df)
                    factor_values[factor_name.value] = [signal]
                except Exception as e:
                    self.logger.error(f"计算因子 {factor_name.value} 失败: {str(e)}")
                    return None
            X = pd.DataFrame(factor_values)
            return X.values
        except Exception as e:
            self.logger.error(f"特征准备失败: {str(e)}")
            return None

    def predict(self, stock_code: str, df: pd.DataFrame, position: int) -> Optional[StockPrediction]:
        if stock_code not in self.models:
            self.logger.warning(f"股票 {stock_code} 的模型未加载")
            return None
        try:
            X = self.prepare_features(df)
            if X is None:
                return None
            model, scaler = self.models[stock_code]
            X_scaled = scaler.transform(X)
            prediction = model.predict(X_scaled)[0]
            probabilities = model.predict_proba(X_scaled)[0]
            confidence = probabilities[prediction]
            return StockPrediction(
                stock_code=stock_code,
                prediction=PredictionResult(prediction),
                confidence=confidence,
                position=position
            )
        except Exception as e:
            self.logger.error(f"预测股票 {stock_code} 失败: {str(e)}")
            return None

    def predict_all(self, historical_data: Dict[str, pd.DataFrame], current_position: int) -> List[StockPrediction]:
        predictions = []
        for stock_code, df in historical_data.items():
            if stock_code in self.models:
                pred = self.predict(stock_code, df, current_position)
                if pred is not None:
                    predictions.append(pred)
                    self.logger.info(
                        f"股票 {stock_code} 预测结果: {pred.prediction.name}, "
                        f"置信度: {pred.confidence:.4f}, 位置: {pred.position}"
                    )
            else:
                self.logger.warning(f"股票 {stock_code} 没有可用的模型")
        self.logger.info(f"共完成 {len(predictions)} 只股票的预测")
        if len(predictions) > 0:
            up_predictions = [p for p in predictions if
                              p.prediction in [PredictionResult.UP, PredictionResult.EXTREME_UP]]
            self.logger.info(f"其中看涨股票数量: {len(up_predictions)}")
        return predictions

class TradeStatus(Enum):
    PENDING = "pending"
    EXECUTED = "executed"
    FAILED = "failed"
    CANCELLED = "cancelled"

@dataclass
class TradeOrder:
```

```python
class TradeOrder:
    stock_code: str
    action: str
    shares: int
    price: float
    status: TradeStatus
    position: int
    prediction: Optional[PredictionResult] = None
    confidence: Optional[float] = None

@dataclass
class Position:
    stock_code: str
    shares: int
    entry_price: float
    entry_position: int
    prediction: PredictionResult
    confidence: float

class PortfolioManager:
    def __init__(self,
                 initial_capital: float,
                 extreme_up_weight: float = 1.5,
                 up_weight: float = 1.0,
                 max_single_position: float = 0.2,
                 max_total_position: float = 0.8):
        self.initial_capital = initial_capital
        self.current_capital = initial_capital
        self.extreme_up_weight = extreme_up_weight
        self.up_weight = up_weight
        self.max_single_position = max_single_position
        self.max_total_position = max_total_position
        self.positions: Dict[str, Position] = {}
        self.current_prices = {}
        self.daily_returns = []
        self.logger = logging.getLogger(__name__)
    def update_current_prices(self, prices: Dict[str, float]):
        self.current_prices = prices
    def calculate_position_sizes(self,
                                 predictions: List[StockPrediction],
                                 current_prices: Dict[str, float]) -> Dict[str, float]:
        self.update_current_prices(current_prices)
        bullish_predictions = [p for p in predictions
                               if p.prediction in [PredictionResult.UP, PredictionResult.EXTREME_UP]]
        if not bullish_predictions:
            return {}
        weights = []
        for pred in bullish_predictions:
            base_weight = (self.extreme_up_weight
                           if pred.prediction == PredictionResult.EXTREME_UP
                           else self.up_weight)
            weight = base_weight * pred.confidence
            weights.append(weight)
        total_weight = sum(weights)
        if total_weight > 0:
            weights = [w / total_weight for w in weights]
        available_capital = self.current_capital * self.max_total_position
        position_sizes = {}
        for pred, weight in zip(bullish_predictions, weights):
            target_amount = min(
                available_capital * weight,
                self.current_capital * self.max_single_position
            )
            position_sizes[pred.stock_code] = target_amount
        return position_sizes
    def update_positions(self,
                         buys: Dict[str, Tuple[int, float]],
                         sells: Dict[str, Tuple[int, float]],
                         predictions: List[StockPrediction]):
        daily_pnl = 0
        for stock_code, (shares, price) in sells.items():
            if stock_code in self.positions:
                pos = self.positions[stock_code]
                realized_pnl = shares * (price - pos.entry_price)
                daily_pnl += realized_pnl
                self.current_capital += shares * price
                if shares >= pos.shares:
                    del self.positions[stock_code]
                else:
                    pos.shares -= shares
        for stock_code, (shares, price) in buys.items():
            pred = next((p for p in predictions if p.stock_code == stock_code), None)
            if pred is None:
                continue
            position = Position(
                stock_code=stock_code,
                shares=shares,
                entry_price=price,
                entry_position=pred.position,
                prediction=pred.prediction,
                confidence=pred.confidence
            )
            self.positions[stock_code] = position
            self.current_capital -= shares * price
        if self.initial_capital > 0:
            daily_return = daily_pnl / self.initial_capital
            self.daily_returns.append(daily_return)
    def get_position_ratio(self) -> float:
        if not self.positions:
            return 0.0
```

```python
                return 0.0
            total_position_value = 0.0
            for stock_code, pos in self.positions.items():
                price = self.current_prices.get(stock_code, pos.entry_price)
                total_position_value += pos.shares * price
            return min(max(total_position_value / self.initial_capital, 0), 1)
        def get_position_summary(self) -> Dict:
            return {
                'total_capital': self.current_capital,
                'position_ratio': self.get_position_ratio() * 100,  # 转换为百分比
                'cumulative_return': (self.current_capital / self.initial_capital - 1) * 100,
                'daily_returns_mean': np.mean(self.daily_returns) * 100 if self.daily_returns else 0,
                'daily_returns_std': np.std(self.daily_returns) * 100 if self.daily_returns else 0,
                'positions': [
                    {
                        'stock_code': pos.stock_code,
                        'shares': pos.shares,
                        'entry_price': pos.entry_price,
                        'current_price': self.current_prices.get(pos.stock_code, pos.entry_price),
                        'prediction': pos.prediction.name,
                        'confidence': pos.confidence,
                        'entry_position': pos.entry_position,
                        'unrealized_pnl': (self.current_prices.get(pos.stock_code,
                                                                    pos.entry_price) - pos.entry_price) * pos.shares
                    }
                    for pos in self.positions.values()
                ]
            }
        def _calculate_max_drawdown(self, returns: np.ndarray) -> float:
            cumulative = (1 + returns).cumprod()
            running_max = np.maximum.accumulate(cumulative)
            drawdown = (running_max - cumulative) / running_max
            return np.max(drawdown) if len(drawdown) > 0 else 0

    from data_processor import process_raw_data, filter_stocks

    class TradingSystem:
        def __init__(self, data_file: str, initial_capital: float,
                     model_path: str = "models",
                     extreme_up_weight: float = 1.5,
                     up_weight: float = 1.0,
                     max_single_position: float = 0.2,
                     max_total_position: float = 0.8):
            self.logger = logging.getLogger(__name__)
            self.data_file = data_file
            self.prediction_manager = PredictionManager(model_path)
            self.portfolio_manager = PortfolioManager(
                initial_capital=initial_capital,
                extreme_up_weight=extreme_up_weight,
                up_weight=up_weight,
                max_single_position=max_single_position,
                max_total_position=max_total_position
            )
            self.data_manager = TradingDataManager()
            self.trading_sequences = {}
            self.current_position = None
            self.trade_history = []

        def initialize(self) -> bool:
            try:
                df = pd.read_excel(self.data_file, header=[0, 1]) if self.data_file.endswith('.xlsx') \
                    else pd.read_csv(self.data_file, header=[0, 1])
                self.logger.info("原始数据信息：")
                self.logger.info(f"索引类型: {type(df.index)}")
                self.logger.info(f"时间范围: {df.index[0]} 到 {df.index[-1]}")
                self.logger.info(f"数据形状: {df.shape}")
                self.logger.info(f"列名: {df.columns.tolist()[:10]}...")
                raw_data = process_raw_data(df)
                for stock_code, stock_df in raw_data.items():
                    self.logger.info(f"\n股票 {stock_code} 处理后数据信息：")
                    self.logger.info(f"时间范围: {stock_df.index[0]} 到 {stock_df.index[-1]}")
                    self.logger.info(f"数据点数: {len(stock_df)}")
                    self.logger.info(f"列名: {stock_df.columns.tolist()}")
                    break
                processed_data = filter_stocks(raw_data)
                self.trading_sequences = self.data_manager.prepare_trading_sequence(processed_data)
                for stock_code, sequence in list(self.trading_sequences.items())[:1]:  # 只打印第一只股票的信息
                    self.logger.info(f"\n股票 {stock_code} 交易序列信息：")
                    self.logger.info(f"序列长度: {len(sequence)}")
                    if sequence:
                        first_point = sequence[0]
                        self.logger.info(
                            f"第一个点数据范围: 从位置 {first_point.position - self.data_manager.warmup_period} "
                            f"到 {first_point.position}"
                        )
                        self.logger.info(f"历史数据形状: {first_point.historical_data.shape}")
                        self.logger.info(f"历史数据列名: {first_point.historical_data.columns.tolist()}")
                stock_codes = list(self.trading_sequences.keys())
                self.prediction_manager.load_models(stock_codes)
                return True
            except Exception as e:
                self.logger.error(f"交易系统初始化失败: {str(e)}")
                return False
        def _log_time_point_start(self, time_point_count: int, total_points: int):
            self.logger.info(f"\n{'=' * 50}")
            self.logger.info(f"开始处理时间点 [{time_point_count}/{total_points}] "
                             f"{self.current_time.strftime('%Y-%m-%d %H:%M:%S')}")
            self.logger.info(f"{'=' * 50}")
        def _log_predictions(self, predictions: List[StockPrediction]):
            self.logger.info("\n预测结果汇总:")
```

```python
        self.logger.info("\n预测结果汇总:")
        self.logger.info(f"总计预测股票数量：{len(predictions)}")
        bullish_predictions = [p for p in predictions
                               if p.prediction in [PredictionResult.UP, PredictionResult.EXTREME_UP]]
        self.logger.info(f"看涨股票数量：{len(bullish_predictions)}")
        if predictions:
            self.logger.info("\n各股票预测详情:")
            for pred in sorted(predictions, key=lambda x: int(x.stock_code)):
                self.logger.info(
                    f"股票 {pred.stock_code:>3}: {pred.prediction.name:<11} "
                    f"置信度：{pred.confidence:.4f}")
    def _log_trading_execution(self, orders: List[TradeOrder]):
        if not orders:
            self.logger.info("\n本交易点无交易执行")
            return
        self.logger.info("\n交易执行情况:")
        by_action = {"buy": [], "sell": []}
        for order in orders:
            by_action[order.action].append(order)
        if by_action["sell"]:
            self.logger.info("\n平仓交易:")
            for order in sorted(by_action["sell"], key=lambda x: int(x.stock_code)):
                self.logger.info(
                    f"股票 {order.stock_code:>3} 平仓：{order.shares:>6}股，"
                    f"价格：{order.price:.2f}，金额：{order.shares * order.price:,.2f}")
        if by_action["buy"]:
            self.logger.info("\n开仓交易:")
            for order in sorted(by_action["buy"], key=lambda x: int(x.stock_code)):
                self.logger.info(
                    f"股票 {order.stock_code:>3} 开仓：{order.shares:>6}股，"
                    f"价格：{order.price:.2f}，金额：{order.shares * order.price:,.2f}")

    def _log_time_point_summary(self, summary: Dict):
        self.logger.info("\n当前交易点总结:")
        self.logger.info(f"位置：{self.current_position}")  # 使用position替代timestamp
        self.logger.info(f"总资产：{summary['total_capital']:,.2f}")
        self.logger.info(f"持仓比例：{summary['position_ratio'] * 100:.2f}%")
        if summary['positions']:
            self.logger.info("\n当前持仓:")
            for pos in sorted(summary['positions'], key=lambda x: int(x['stock_code'])):
                self.logger.info(
                    f"股票 {pos['stock_code']:>3}: {pos['shares']:>6}股，"
                    f"成本价：{pos['entry_price']:.2f}，"
                    f"当前盈亏：{pos.get('unrealized_pnl', 0):,.2f}")
    def _execute_trades(self,
                        target_positions: Dict[str, float],
                        current_prices: Dict[str, float],
                        exit_prices: Dict[str, float],
                        predictions: List[StockPrediction],
                        force_clear: bool = False) -> List[TradeOrder]:
        orders = []
        current_positions = self.portfolio_manager.positions
        for stock_code, position in list(current_positions.items()):
            if position.shares > 0:
                if force_clear or stock_code in exit_prices:
                    exit_price = current_prices.get(stock_code) or exit_prices.get(stock_code)
                    if exit_price:
                        order = TradeOrder(
                            stock_code=stock_code,
                            action="sell",
                            shares=position.shares,
                            price=exit_price,
                            status=TradeStatus.EXECUTED,
                            timestamp=self.current_time
                        )
                        orders.append(order)
                        self.logger.info(
                            f"股票 {stock_code} {'强制清仓' if force_clear else '达到平仓时间'}，"
                            f"以 {exit_price:.2f} 价格平仓 {position.shares} 股"
                        )
        if not force_clear:
            for stock_code, target_amount in target_positions.items():
                if stock_code in current_prices:
                    price = current_prices[stock_code]
                    shares = int(target_amount / price)
                    if shares > 0:
                        pred = next((p for p in predictions if p.stock_code == stock_code), None)
                        order = TradeOrder(
                            stock_code=stock_code,
                            action="buy",
                            shares=shares,
                            price=price,
                            status=TradeStatus.EXECUTED,
                            timestamp=self.current_time,
                            prediction=pred.prediction if pred else None,
                            confidence=pred.confidence if pred else None
                        )
                        orders.append(order)
                        self.logger.info(f"股票 {stock_code} 开仓，以 {price:.2f} 价格买入 {shares} 股")
        return orders
    def execute_trading(self) -> Dict:
        try:
            all_summaries = []
            first_stock_code = next(iter(self.trading_sequences))
            total_points = len(self.trading_sequences[first_stock_code])
            point_count = 0
            current_points = self.data_manager.get_next_trading_point(self.trading_sequences)
            if not current_points:
                return {"error": "没有可交易的时间点"}
            first_point = next(iter(current_points.values()))
```

```python
            first_point = next(iter(current_points.values()))
            self.current_position = first_point.position
            while True:
                point_count += 1
                self.logger.info(f"\n=== 处理交易点 [{point_count}/{total_points}] ===")
                historical_data = {
                    stock_code: point.historical_data
                    for stock_code, point in current_points.items()
                    if point is not None
                }
                current_prices = {
                    stock_code: point.current_price
                    for stock_code, point in current_points.items()
                    if point is not None
                }
                exit_prices = {
                    stock_code: point.future_price
                    for stock_code, point in current_points.items()
                    if point is not None
                }
                self.portfolio_manager.update_current_prices(current_prices)
                is_last_point = (point_count >= total_points)
                self.logger.info("\n=== 执行平仓操作 ===")
                close_orders = self._execute_close_positions(
                    current_prices, exit_prices, force_clear=is_last_point)
                if close_orders:
                    self._log_trading_execution(close_orders)
                    self._update_portfolio(close_orders, [])
                if not is_last_point:
                    predictions = self.prediction_manager.predict_all(
                        historical_data,
                        self.current_position
                    )
                    self._log_predictions(predictions)
                    target_positions = self.portfolio_manager.calculate_position_sizes(
                        predictions, current_prices)
                    open_orders = self._execute_open_positions(
                        target_positions, current_prices, predictions)
                    if open_orders:
                        self._log_trading_execution(open_orders)
                        self._update_portfolio(open_orders, predictions)
                else:
                    self.logger.info("\n=== 最终清仓完成 ===")
                current_summary = self.get_trading_summary()
                self._log_time_point_summary(current_summary)
                all_summaries.append(current_summary)
                if is_last_point:
                    break
                next_points = self.data_manager.get_next_trading_point(
                    self.trading_sequences, self.current_position)

                next_point = next(point for point in next_points.values()
                                  if point is not None)
                self.current_position = next_point.position
                current_points = next_points
            final_metrics = self.get_performance_metrics()
            self.logger.info("最终交易指标:")
            for key, value in final_metrics.items():
                self.logger.info(f"{key}: {value}")
            return {
                "summaries": all_summaries,
                "final_metrics": final_metrics
            }
        except Exception as e:
            self.logger.error(f"交易执行失败: {str(e)}", exc_info=True)
            return {"error": str(e)}
    def _execute_open_positions(self,
                                target_positions: Dict[str, float],
                                current_prices: Dict[str, float],
                                predictions: List[StockPrediction]) -> List[TradeOrder]:
        open_orders = []
        for stock_code, target_amount in target_positions.items():
            if stock_code in current_prices:
                price = current_prices[stock_code]
                shares = int(target_amount / price)
                if shares > 0:
                    pred = next((p for p in predictions if p.stock_code == stock_code), None)
                    order = TradeOrder(
                        stock_code=stock_code,
                        action="buy",
                        shares=shares,
                        price=price,
                        status=TradeStatus.EXECUTED,
                        position=self.current_position,
                        prediction=pred.prediction if pred else None,
                        confidence=pred.confidence if pred else None
                    )
                    open_orders.append(order)
                    self.logger.info(f"股票 {stock_code} 开仓，以 {price:.2f} 价格买入 {shares} 股")
        return open_orders

    def _execute_close_positions(self,
                                 current_prices: Dict[str, float],
                                 exit_prices: Dict[str, float],
                                 force_clear: bool = False) -> List[TradeOrder]:
        close_orders = []
        current_positions = self.portfolio_manager.positions
        for stock_code, position in list(current_positions.items()):
            if position.shares > 0:
                if force_clear or stock_code in exit_prices:
```

```python
                if force_clear or stock_code in exit_prices:
                    exit_price = current_prices.get(stock_code) or exit_prices.get(stock_code)
                    if exit_price:
                        order = TradeOrder(
                            stock_code=stock_code,
                            action="sell",
                            shares=position.shares,
                            price=exit_price,
                            status=TradeStatus.EXECUTED,
                            position=self.current_position,  # 使用position替代timestamp
                            prediction=None,
                            confidence=None
                        )
                        close_orders.append(order)
                        self.logger.info(
                            f"股票 {stock_code} {'强制清仓' if force_clear else '达到平仓时间'}, "
                            f"以 {exit_price:.2f} 价格平仓 {position.shares} 股"
                        )
        return close_orders

    def _update_portfolio(self, orders: List[TradeOrder],
                          predictions: List[StockPrediction]):
        buys = {}
        sells = {}
        for order in orders:
            if order.status != TradeStatus.EXECUTED:
                continue
            if order.action == "buy":
                buys[order.stock_code] = (order.shares, order.price)
            elif order.action == "sell":
                sells[order.stock_code] = (order.shares, order.price)
        self.portfolio_manager.update_positions(buys, sells, predictions)
        self.trade_history.extend(orders)
    def get_trading_summary(self) -> Dict:
        summary = self.portfolio_manager.get_position_summary()
        current_trades = [order for order in self.trade_history
                          if order.position == self.current_position]
        summary['trades'] = {
            'position': self.current_position,
            'total_trades': len(current_trades),
            'buy_trades': len([t for t in current_trades if t.action == "buy"]),
            'sell_trades': len([t for t in current_trades if t.action == "sell"]),
            'trades_detail': [
                {
                    'stock_code': t.stock_code,
                    'action': t.action,
                    'shares': t.shares,
                    'price': t.price,
                    'amount': t.shares * t.price,
                    'prediction': t.prediction.name if t.prediction else None,
                    'confidence': t.confidence if t.confidence else None
                }
                for t in current_trades
            ]
        }
        return summary
    def get_performance_metrics(self) -> Dict:
        metrics = {
        "总交易次数": len(self.trade_history),
        "买入交易次数": sum(1 for t in self.trade_history if t.action == "buy"),
        "卖出交易次数": sum(1 for t in self.trade_history if t.action == "sell"),
        "累计收益率": (self.portfolio_manager.current_capital / self.portfolio_manager.initial_capital - 1) * 100,
        }
        point_returns = self.portfolio_manager.daily_returns
        if point_returns:
            returns_array = np.array(point_returns)
            metrics.update({
                "平均点收益率": f"{np.mean(returns_array) * 100:.4f}%",
                "收益率标准差": f"{np.std(returns_array) * 100:.4f}%",
                "夏普比率": f"{np.mean(returns_array) / np.std(returns_array) if np.std(returns_array) > 0 else 0:.4f}",
                "最大回撤": f"{self.portfolio_manager._calculate_max_drawdown(returns_array) * 100:.2f}%",
            })
        if self.trade_history:
            entry_prices = {}
            profitable_trades = 0
            total_closed_trades = 0
            for trade in self.trade_history:
                stock_code = trade.stock_code
                if trade.action == "buy":
                    entry_prices[stock_code] = trade.price
                elif trade.action == "sell" and stock_code in entry_prices:
                    entry_price = entry_prices[stock_code]
                    if trade.price > entry_price:
                        profitable_trades += 1
                    total_closed_trades += 1
                    del entry_prices[stock_code]
            if total_closed_trades > 0:
                win_rate = profitable_trades / total_closed_trades
                metrics["胜率"] = f"{win_rate * 100:.2f}%"
                metrics["盈利交易数"] = profitable_trades
                metrics["总平仓交易数"] = total_closed_trades
        return metrics
```

```python
#trading_data_manager.py
from typing import Dict, List, Optional
import pandas as pd
import numpy as np
import logging
from dataclasses import dataclass

@dataclass
class TradingPoint:
    position: int
    historical_data: pd.DataFrame
    current_price: float
    future_price: float

class TradingDataManager:
    def __init__(self, warmup_period: int = 300, prediction_window: int = 49):
        self.warmup_period = warmup_period
        self.prediction_window = prediction_window
        self.logger = logging.getLogger(__name__)
    def prepare_trading_sequence(self,
                                 stock_data: Dict[str, pd.DataFrame]
                                 ) -> Dict[str, List[TradingPoint]]:
        trading_sequences = {}

        for stock_code, df in stock_data.items():
            try:
                sequence = []
                for pos in range(self.warmup_period,
                                 len(df) - self.prediction_window,
                                 self.prediction_window):
                    historical_data = df.iloc[pos - self.warmup_period:pos + 1].copy()
                    historical_data = historical_data.reset_index(drop=True)
                    current_price = float(df.iloc[pos]['close'])
                    future_price = float(df.iloc[pos + self.prediction_window]['close'])
                    trading_point = TradingPoint(
                        position=pos,
                        historical_data=historical_data,
                        current_price=current_price,
                        future_price=future_price
                    )
                    sequence.append(trading_point)
                if sequence:
                    trading_sequences[stock_code] = sequence
                    self.logger.info(
                        f"股票 {stock_code} 生成了 {len(sequence)} 个交易点"
                    )
                    self.logger.info(f"第一个点位置: {sequence[0].position}")
                    self.logger.info(f"最后一个点位置: {sequence[-1].position}")
            except Exception as e:
                self.logger.error(f"处理股票 {stock_code} 时出错: {str(e)}")
                continue
        return trading_sequences
    def get_next_trading_point(self,
                               trading_sequences: Dict[str, List[TradingPoint]],
                               current_position: Optional[int] = None
                               ) -> Dict[str, Optional[TradingPoint]]:
        next_points = {}
        if current_position is None:
            for stock_code, sequence in trading_sequences.items():
                next_points[stock_code] = sequence[0] if sequence else None
                if sequence:
                    self.logger.debug(
                        f"股票 {stock_code} 初始点位置: {sequence[0].position}"
                    )
            return next_points
        for stock_code, sequence in trading_sequences.items():
            if not sequence:
                next_points[stock_code] = None
                continue
            try:
                current_index = next(
                    (i for i, point in enumerate(sequence)
                     if point.position == current_position),
                    None
                )
                if current_index is None:
                    next_points[stock_code] = None
                    self.logger.warning(
                        f"股票 {stock_code} 找不到当前位置 {current_position}"
                    )
                    continue
                next_index = current_index + 1
                if next_index < len(sequence):
                    next_points[stock_code] = sequence[next_index]
                    self.logger.debug(
                        f"股票 {stock_code} 下一个点位置: {sequence[next_index].position}"
                    )
                else:
                    next_points[stock_code] = None
                    self.logger.debug(f"股票 {stock_code} 没有下一个点")
            except Exception as e:
                self.logger.error(f"获取股票 {stock_code} 下一个点时出错: {str(e)}")
                next_points[stock_code] = None
        return next_points
```