2023 年第四届“大湾区杯”粤港澳金融数学建模竞赛

题目：一种跨境 ETF 套利策略模型设计

摘　要

本研究旨在设计一套高效的跨境交易所交易基金（ETF）交易策略，以及探索跨境 ETF 与股指期货之间的跨市场套利机会。ETF 作为一种流行的投资工具，其套利策略对于保持市场效率至关重要。跨境 ETF，特别是与股指期货结合的跨市场套利，提供了一种独特的机会来利用不同市场间的价格差异。

在现有文献中，ETF 套利策略[4]的研究着重于市场效率和套利机会。Sethi 和 Tripathi 发现印度市场上 ETF 相对于基础投资组合的回报更为波动，这种波动性与套利成本高的理论相符，暗示市场效率不足[1]。Boadu-Sebbe 通过动态状态空间模型分析了 ETF 套利对基础资产的影响，揭示了流动性冲击如何通过套利机制影响基础资产价格[2]。Fulkerson 等人的研究则指出，债券 ETF 的套利策略对市场回报有显著影响，但市场的成本和不确定性阻碍了有效套利[3]。这些研究为跨市场套利策略提供了理论支持。

针对问题一，我们构建了一个基于 **Engle-Granger 协整检验**与误差修正模型的时间序列预测模型，利用一级市场与二级市场的价格数据。结合国内 T+1 交易规则，我们设计了基本型持仓与补仓策略，并通过 BFGS 优化算法调整交易决策参数，以实现对 886 只 ETF 基金的利润最大化。通过这一模型，我们筛选出了 10 只最适合交易的 ETF。

针对问题二，我们采用了以 **LSTM 网络**（长短期记忆网络）为核心、全连接层为辅助的时间序列预测模型，并创新性地引入了折溢价率、跟踪误差和基于**广义 Hurst 函数**矩阵得到的分形性数据，以挖掘市场价格的波动性和规律性。这些指标对交易决策参数的取值有显著影响，特别是在考虑跨境 ETF 允许 T+0 交易规则的超参数设置中。基于此模型，我们选出了 10 只最适合交易的跨境 ETF。

针对问题三，我们继续使用相同架构的时间序列预测网络，并引入了“市场波动指数”来预测股指期货与跨境 ETF 的价格波动。通过资产间比例系数，我们确保了股指期货和跨境 ETF 在标的资产或指标数量上的一致性。我们深入分析了不同市场间的价格相关性、协整关系以及成交量比率和回归斜率，以决定交易决策参数的取值。此外，我们提出了 DBP 策略，以适应波动型市场。最终，我们选出了 10 只区间利润最高的股指期货与跨境 ETF 组合，并计算了 **Sharpe 比率**和 **Markowitz 均值-方差模型**的简化形式，以确定组合权重。

本研究的实测部分不在本文中展示，而是在附加的实测报告文件中提供，其中使用了实测数据以及问题二和问题三提出的模型进行了详细解释。

关键词：**Engle-Granger 协整检验　LSTM 网络 广义 Hurst 函数　Sharpe 比率　Markowitz 均值-方差模型**

一、 问题重述

问题一要求仅针对给定的在上海交易所和深圳交易所上市的 ETF 指数基金成分股，建立一二级市场间套利模型并进行分类选择，根据某些指标选出最适合进行套利的前 10 只。

问题二要求针对不同市场的差异和不同股票的特点，针对交易制度的不同，修改任务一的策略，建立跨境 ETF 套利模型，选出最适合进行套利的前 10 只。

问题三要求我们利用跨境 ETF 与国际市场的联动性，以及国际市场对 A 股的影响，建立跨境 ETF 与股指期货的跨市场套利模型并选择最适合跨市场套利的跨境 ETF 和股指期货组合。其中股指期货包括中金所上证 50、沪深 300、中证 500 和中证 1000 四个指数的股指期货。

问题四要求基于任务二建立的模型对恒生科技 ETF(513130)，以及任务三的跨市场套利模型对跨境 ETF 和股指期货组合进行实测和评估策略的效果，以验证我们建立的套利模型的有效性。

二、 问题分析

2.1 问题一的分析

跨境 ETF 是指以境外资本市场证券构成的境外市场指数为跟踪标的、在国内证券交易所上市的交易型开放式指数基金。 由于不同市场 ETF 在外汇管理制度、交易 制度、交易时差等方面存在差异，跨境 ETF 在申赎机制、套利机制、套利成本等 方面与境内 ETF 又有着不少区别。因此问题一仅要求对给定的 ETF 指数基金成分股进行分析和筛选进行选股。综合考虑了成分股的交易量、流动性、相关性、市值以及 ETF 基金的市场风险和特有风险因素，我们采用 E-G 协整交易预测策略对输入的 ETF 时间序列进行 E-G 协整和 ECM 误差校正，选出了一二级市场间套利模型最适合进行套利的前 10 只。

2.2 问题二的分析

由于不同股票的波动性有所不同，与 A 股股票相关的 ETF 基金只能进行 T+1 交易，而跨境 ETF 可以进行 T+0 交易。因此问题二要求针对交易制度的不同，对任务一的策略进行修改，建立跨境 ETF 的套利模型。考虑到折溢价率、交易数量、跟踪误差、宏观指标等因素，我们采用对时间序列回归方法和交易数量判断函数进行调整选出最适合进行套利的前 10 只 ETF 基金，来优化交易策略的执行和管理，并分析和预测不同市场之间的价格差异和套利机会。

2.3 问题三的分析

目前中金所有上证 50、沪深 300、中证 500、中证 1000 共四个指数的股指期货。通过分析历史数据和统计指标，根据跨境 ETF 与国际市场的联动性和股指期货相关性以及国际市场对 A 股的影响，我们综合考虑了包括跨境 ETF 和股指期货的价格差异、交易的时间点、交易的数量、交易的成本、资金流动性等等因素同时同步收集实际交易数据，计算股票的收益率，并绘制收益率曲线，根据建立的跨市场套利模型选出最适合跨市场套利的跨境 ETF 和股指期货组合。最后根据模型的预测能力、历史回测结果、交易的盈亏情况、风险管理的效果等来评估所设计的套利策略的效果，并通过套利收益、交易成本、风险管理等指标，以确保策略的可行性和稳定性。

2.4 问题四的分析

针对恒生科技 ETF (513130)以及跨境 ETF 和股指期货组合进行实测，我们将按照给定的参数设置和模型给出的交易策略进行操作，记录包括初始投资、期货保证金比率、交易费率和价格滑点等相关数据与指标，预测该 ETF 的价格变动，并根据实测结果评估该 ETF 的表现。同时我们跟踪误差和波动性分析，得出跟踪误差和波动性的原因，从而对交易策略提出改进方案与建议。根据实测结果评估所采用的投资策略的有效性和稳定性进行投资策略评估，包括收益与风险平衡、相对基准的超额收益等指标。

三、 模型假设

1. 本研究计算任务一的交易策略收益率所依据的数据区间为 9 月 18 日至 10 月 30 日；而任务二的交易策略收益率则基于 10 月 18 日至 10 月 30 日的数据进行计算。
2. 在任务一的模型构建中，我们综合考量了成分股的交易量、流动性、相关性、市值，以及 ETF 基金所承担的市场风险和特有风险因素，以确保策略的全面性和风险的可控性。
3. 为了研究的纯粹性，本文假设市场股价不受采用的跨境 ETF 套利策略的影响，即市场价格是由外部因素以外的模型策略所独立。
4. 鉴于跨市场套利需考虑货币汇率波动的影响，本研究采纳了一种将所有交易数据转换为以人民币（CNY）计价的方法，此举旨在消除货币汇率波动可能引入的误差，以提高套利策略的精确度和可靠性。
5. 本研究将跨境 ETF 的定义扩展至包括“大湾区 ETF”在内的特殊类别，这些 ETF 涉及的标的资产位于香港、澳门等非中国大陆地区，但在中国大陆交易市场有交易记录，因此被纳入跨境 ETF 的范畴进行分析。

四、 符号说明

符号	说明	单位
$y_{appr}[t]$	每股 ETF 二级市场价值	￥
P_0	开盘价	￥
P_c	收盘价	￥
P_h	最高价	￥
P_l	最低价	￥
y_t	时间点 t 的 ETF 二级市场价格	￥
x_t	时间点 t 的 ETF 一级市场价格	￥
ε_t	误差项，代表两个市场价格之间偏离均衡关系的程度	
H_0	ε_t 有单位根（非平稳）	
H_1	ε_t 无单位根（平稳）	
γ	误差修正项的系数，衡量长期均衡关系偏离对短期价格变动的调整速度	

ε_{t-1}；	滞后一期的残差，表示上一期的均衡状态偏差	
ϕ_i, θ_j	短期动态调整系数，分别对应一级市场价格和二级市场价格的滞后项的影响	
μ_t	随机扰动项	
$\hat{y}_{t+1}$	下一时期的价格	￥
eff_1	一级市场每笔交易固定成本	￥
eff_2	ETF 二级市场交易以及股票交易的佣金率	
eff_3	印花税率	
δ_{th}	交易阈值	
δ_t	二级市场与一级市场之间 ETF 单位净值与交易价格的差值	
d_1, d_2, d_3, d_4	交易决策参数	
$Q_{buy,1}$	一级市场买入数量	
$Q_{sell,2}$	二级市场卖出数量	
$Q_{buy,2}$	二级市场买入数量	
$Q_{sell,1}$	一级市场卖出数量	
P_{t+1}	第 $t+1$ 日的交易价格	￥
Θ_t	第 t 日末的持仓量	
$Q_{t\to t+1}$	在 t 日末基于 P_{t+1} 价格确定的第 $t+1$ 日的交易量	
Θ^*	目标持仓量	
$Q_{adjust,t}$	每日结束时的平仓或补仓量	
$\Pi(\mathbf{d})$	在参数 $\mathbf{d}$ 下某一 ETF 的累计利润	￥
P_{start}	初始投资额	￥
P_{end}	期末账户价值	￥
R	利润率	
R_t	单位时间利润	￥/min 或 ￥/day
T	考察期间的总时间单位	min 或 day
$P(t)$	时间 t 的账户价值	￥
N_t	复权单位净值	
D_t	贴水	
R_t	贴水率	

P_t	t 时刻 ETF 的市场价格	￥
G_t	单位净值的日增长率	
λ	衰减率参数	
P	折溢价率	
P_m	代表二级市场价格	￥
P_{nav}	代表一级市场的净资产价值（NAV）	￥
$y_v(i)$	降趋势函数	
$F^2(s,v)$	降趋势方差	
TE	跟踪误差	
$H(q)$	广义 Hurst 指数	
$F_q(s)$	q 阶波动函数	
$\mathrm{tr}(c1,c2)$	转换函数	
shuffle	一个基于种子 seed 的随机打散函数	
$w_{j'}$	打散后向量的第 j 个元素	
m_i	缩放聚合值	
$\mathbf{F}' = \{f_{1'}, f_{2'}, \ldots, f_4\}$	聚合后的向量	
f_1, f_2, f_3, f_4	交易量影响因子	
f_i	策略因子	
d_i	决策变量	
$P_{futures}(t=0)$	境内股指期货的单位价格	￥
$P_{ETF}(t=0)$	境外跨境 ETF 的单位价格	￥
$P_x[t]$	期货的市场价格	￥
$P_y[t]$	跨境 ETF 的市场价格	￥
$Q_x[t]$	股指期货的交易量	
$Q_y[t]$	跨境 ETF 的交易量	

五、模型的建立与求解

5.1 问题一模型的建立与求解

在问题一中，我们关注 ETF 的溢价和折价套利策略，以探索一级与二级市场间的套利机会。结合开放式基金的发行和赎回机制，以及二级市场的交易特性，我们运用 Engle-Granger 协整套利模型对存在于上海和深圳交易所的 886 个 ETF 进行定量分析，旨在识别出在考虑交易成本和流动性后预期收益最大的前 10 只 ETF。此外，中国内地市场的 T+1

结算规则被纳入考量，以适应结算周期对套利策略可能产生的影响，确保策略在实际市场中的适用性和风险管理的有效性。

5.1.1 数据预处理

首先建立以分钟为单位的一级市场价格时间序列 X，通过取当天的二级市场开盘价近似得到每股 ETF 的一级市场资产净值(NAV)，组成序列数组。其次建立以分钟为单位的二级市场价格序列 Y，使用每一交易分钟的开盘价、收盘价、最高价和最低价的算数平均值来近似每股 ETF 的二级市场价值。数组的每个元素为 $y[t]$，计算公式为：

$$y_{appr}[t]=\frac{P_0+P_c+P_h+P_l}{4} \quad (1)$$

其中：P_0 为开盘价，P_c 为收盘价，P_h 为最高价，P_l 为最低价。

5.1.2 基于 Engle-Granger 协整检验与误差修正的价格序列预测模型

我们建立 Engle-Granger 协整模型[5]以验证 ETF 一级市场价格与二级市场价格之间的协整关系，得到协整检验的统计决策依据 $p-value$，根据 $p-value$ 的值决定是否建立误差修正模型并应用于不同市场的价格时间序列，以得到 $t+1$ 时刻下的价格序列预测结果。

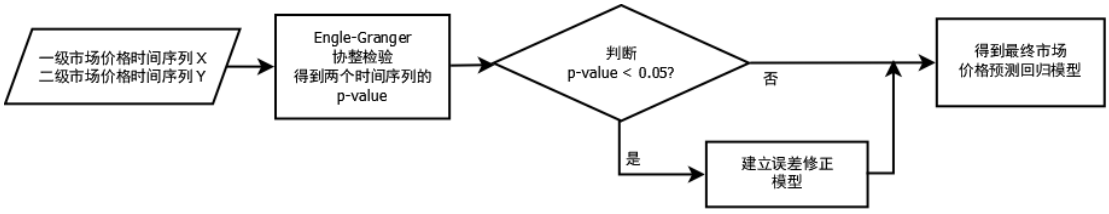

图 1 任务一价格序列预测模型流程图

一、Engle-Granger 协整检验

首先建立时间序列 X 与 Y 间的长期均衡关系，设定协整回归模型如下：

$$y_t=\alpha+\beta x_t+\varepsilon_t \quad (2)$$

其中，y_t 为在时间点 t 的 ETF 二级市场价格；x_t 为在时间点 t 的 ETF 一级市场价格；α 为常数项，代表价格关系的截距；β 为系数，量化一级市场价格变化对二级市场价格的影响；ε_t 为误差项，代表两个市场价格之间偏离均衡关系的程度。

接着检验残差的平稳性。对于步骤一中获得的残差序列 $\{\varepsilon_t\}$，我们进行单位根检验（ADF 检验），以确认残差序列是否平稳。如果 $\{\varepsilon_t\}$ 是平稳的，则可认为存在协整关系。单位根检验的零假设和备择假设分别为：

H_0：ε_t 有单位根（非平稳）；H_1：ε_t 无单位根（平稳）。

二、误差修正模型

当 $p-value<0.05$ 成立时，将残差序列 ε_t 纳入到短期动态模型中，作为修正因子。构建误差修正模型(ECM)如下：

$$\Delta y_t=\gamma\epsilon_{t-1}+\sum_{i=1}^{p}\phi_i\Delta x_{t-i}+\sum_{j=1}^{q}\theta_j\Delta y_{t-j}+\mu_t \quad (3)$$

其中，$\Delta y_t=y_t-y_{t-1}$ 代表变量的一阶差分，即，表示二级市场价格的短期变动；γ 为

误差修正项的系数，衡量长期均衡关系偏离对短期价格变动的调整速度；ε_{t-1}为滞后一期的残差，表示上一期的均衡状态偏差；ϕ_i,θ_j为短期动态调整系数，分别对应一级市场价格和二级市场价格的滞后项的影响；μ_t为随机扰动项。

我们通过向量自回归（VAR）方法拟合模型，运用历史时间序列数据作为预测变量，并通过误差修正项$\gamma\varepsilon_{t-1}$来确保模型在短期偏离后能够重新调整至长期均衡。

三、市场价格预测

在构建用于预测市场价格的数学模型中，我们考虑了误差修正机制以确保长期均衡的维持，并同时允许短期的动态调整。具体地，模型采用了一个误差修正项来调节价格预测，从而促成价格向均衡状态的回归。预测下一时期的价格$\hat{y}_{t+1}$的公式可以表示为：

$$\hat{y}_{t+1}=\alpha+\beta\Delta x_t+\gamma(y_t-a-\beta x_t)\ (4)$$

其中，α、β和γ是模型参数，它们分别代表长期均衡关系中的截距项和斜率项，以及误差修正的调节系数。Δx_t代表解释变量x在时期t的变化量，而$y_t-a-\beta x_t$则是时期t的误差修正项，它代表当前价格与长期均衡价格之间的偏差。通过这种方式，我们的模型不仅可以捕捉短期内价格的波动性，而且能够确保长期价格预测的准确性和稳定性。

5.1.3 交易策略制定

一、条件参数约束

依据金融市场的实际运作机制及相关法律法规，我们设定以下参数：

1. 一级市场交易成本eff_1：包括申购、赎回费用、过户费、证管费用、经手费以及证券结算金，每笔交易固定成本设定为 1.00 元，反映了投资者在基金申赎过程中所需承担的基本费用。
2. 二级市场交易佣金eff_2：根据证券市场常规操作，ETF 二级市场交易以及股票交易的佣金率定为交易利润的 0.2%，体现了投资者在证券交易所进行交易时需支付的手续费。
3. 印花税率eff_3：遵循现行的《中华人民共和国印花税法》，我们将印花税率定为每笔交易利润的 0.1%，作为所有市场交易成本的组成部分。
4. 市场冲击成本eff_4：考虑到大额交易可能对市场价格造成的影响，设定市场冲击成本为每笔交易利润的 0.05%，以模拟实际市场中的流动性成本。
5. 二级市场交易限额$trade_regulation$：参照证券市场的流通规则，我们将单次交易数量限制在每分钟总交易量的 70%以内，旨在模拟市场中单一投资者的交易量对市场流动性的实际影响。
6. 交易阈值δ_{th}：基于市场最小变动价位，设定为 1e-5，用以指导何时进行交易的决策点。
7. 限制每单位限购 ETF 数量为$\pm10^6$，每时刻下持仓数目不可高于或低于此限制范围。

二、交易策略逻辑判断机制

在构建 ETF 交易策略的数学模型中，我们引入了交易决策参数d_1,d_2,d_3,d_4来定量化交易决策。此策略基于二级市场与一级市场之间 ETF 单位净值与交易价格的差值δ_t来判断折价与溢价的情况，并据此执行相应的套利交易。

当$\delta_t=y_{t+1}-x_{t+1}>\delta_{th}$时，判断为折价交易，此时在一级市场以低价购买 ETF 并赎回其成分股，然后在二级市场以高价卖出成分股获得利润。交易规则如下：

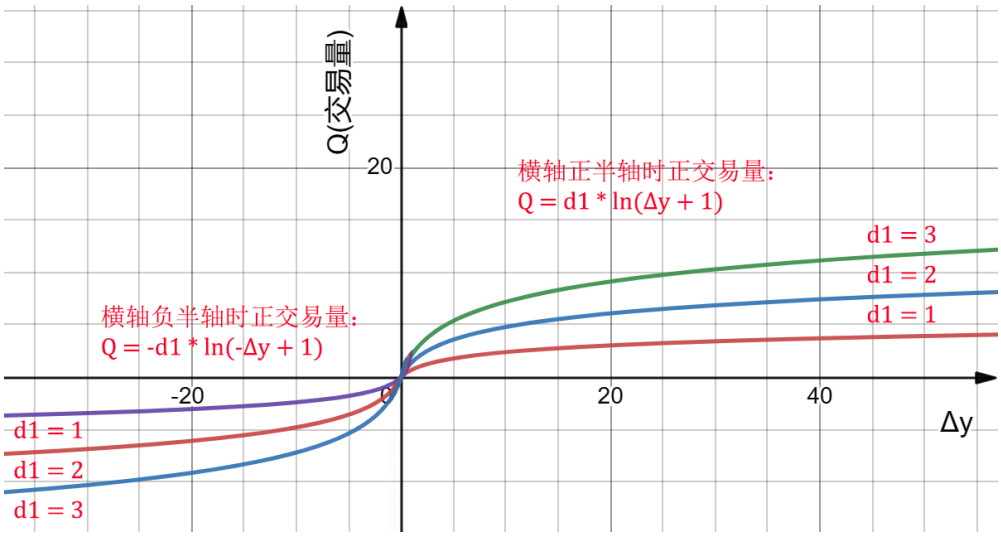

图 2 以 d1 为例，买入数量与 Δy 成近似对数关系

一级市场买入数量： $Q_{buy,1} = d_1 \cdot \ln(|\Delta y_t| + 1)$ (5)

二级市场卖出数量： $Q_{sell,2} = d_2 \cdot \ln(|\Delta y_t| + 1)$ (6)

而若 $\delta_t < -\delta_{th}$，则执行溢价交易，策略为在二级市场购入一篮子成分股并申购 ETF，随后在一级市场高价卖出 ETF。交易规则如下：

二级市场买入数量： $Q_{buy,2} = d_3 \cdot \ln(|\Delta y_t| + 1)$ (7)

一级市场卖出数量： $Q_{sell,1} = d_4 \cdot \ln(|\Delta y_t| + 1)$ (8)

三、中国内地市场 T+1 交易结算规则的考虑

考虑中国内地 T+1 交易结算规则，我们构建以下模型约束：设 P_{t+1} 表示第 $t+1$ 日的交易价格，而 Θ_t 表示在第 t 日末的持仓量。由于 T+1 规则的限制，第 t 日决定的交易将在 $t+1$ 日价格 P_{t+1} 下成交，因此我们有：

$$\Theta_{t+1} = \Theta_t + Q_{t \to t+1}(P_{t+1}) \quad (9)$$

其中 $Q_{t \to t+1}$ 为在 t 日末基于 P_{t+1} 价格确定的第 $t+1$ 日的交易量。

四、基本型持仓与补仓策略

为了维持持仓量的恒定性，我们设定平仓或补仓策略，以在交易日结束时调整持仓量。设 Θ^* 为目标持仓量，则每日结束时的平仓或补仓量 $Q_{adjust,t}$ 满足：

$$\Theta_{t+1} + Q_{adjust,t}(P_t) = \Theta^* \quad (10)$$

这样能够确保在价格 P_t 下第 $t+1$ 日开市前持仓量恢复到 Θ^*。

如下图所示为上述基本型持仓与补仓策略的图片说明。

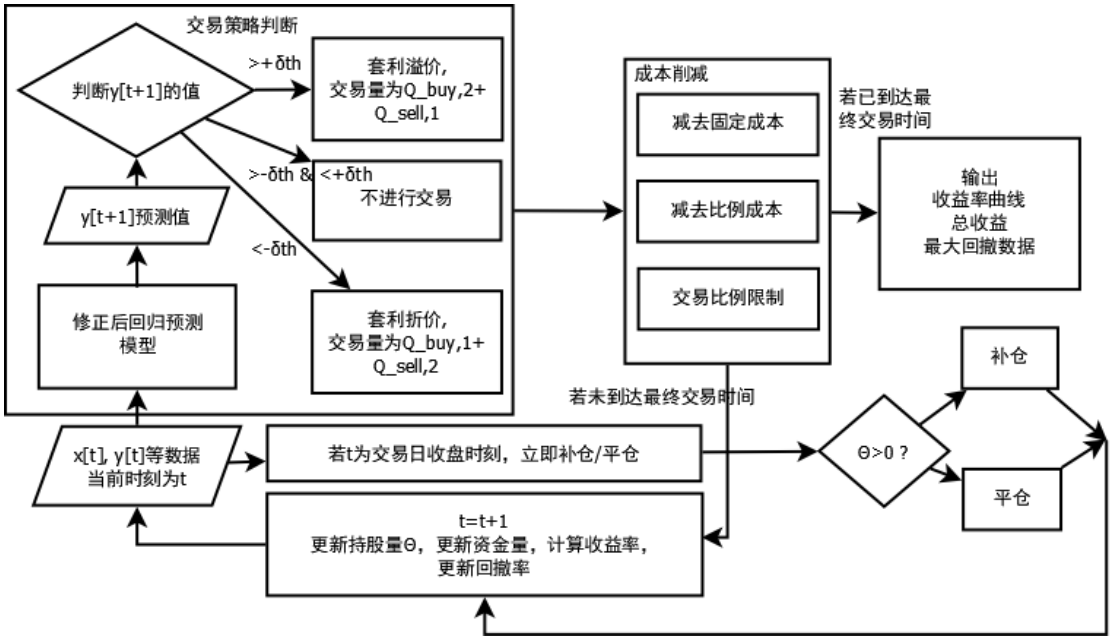

图 3 基本型交易策略流程图

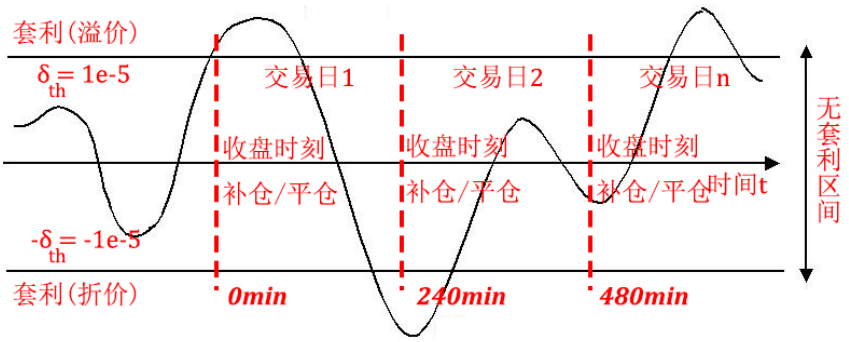

图 4 基本型持仓与补仓策略简要说明

三、交易决策参数的确定

交易决策参数 $\mathbf{d}=[d_1,d_2,d_3,d_4]$ 由 BFGS(Broyden-Fletcher-Goldfarb-Shanno)优化算法确定，以实现利润最大化。具体地，令 $\Pi(\mathbf{d})$ 代表在参数 $\mathbf{d}$ 下某一 ETF 的累计利润，则优化问题可表述为 $\max \Pi(\mathbf{d})$。BFGS 算法通过迭代更新参数 $\mathbf{d}$，使用历史交易数据 $\mathcal{D}$ 评估并提升 $\Pi[\mathbf{d}]$，直至达到收敛条件或迭代次数上限。

最后计算得到四个决策参数分别为：

$$d_1=15.993, d_2=-6.654, d_3=5.151, d_4=47.027$$

我们选取 2023 年 8 月 1 日至 10 月 31 日区间内所有交易日的高频数据进行模拟交易，分析所有在册 ETF 的利润数据。同时，为了更好评估每只 ETF 的短期表现，选取 2023 年 8 月 1 日至 8 月 7 日五个交易日内的高频数据，计算对应的单位时间收益率、回撤率等结果。根据每只 ETF 的累计利润，我们筛选出总数 886 种 ETF 中利润最大的前 10 只，作为最适合进行交易的 ETF 基金，以评估交易策略的有效性。

5.1.4 检验结果

在评估检验结果的过程中，我们使用以下量化策略表现的指标：

1. 利润率 R：

$$R = \frac{P_{\text{end}} - P_{\text{start}}}{P_{\text{start}}} \times 100\% \quad (11)$$

其中，P_{start} 为初始投资额，P_{end} 为期末账户价值。利润率表示在选定的时间范围内，通过套利策略从一级市场和二级市场的价格差异中获取的百分比收益。

2. 单位时间利润 R_t：

$$R_t = \frac{R}{T} \quad (12)$$

其中，T 表示考察期间的总时间单位，可以是天、小时等。单位时间利润表示策略在每个时间单位上的平均收益率，用于评估策略的效率。

3. 最大回撤 MDD：

$$MDD = \max_{t\in[0,T]}(\max_{\tau\in[0,t]} P(\tau) - P(t)) \quad (13)$$

其中，$P(t)$ 是在时间 t 的账户价值。最大回撤是衡量策略风险的指标，它计算了在观察期间内，账户价值从峰值下跌到谷底的最大幅度，表现为投资组合潜在的最大损失。

由每只 ETF 在 2023 年 8 月 1 日至 10 月 31 日区间内所有交易日下的累计利润，可以得到前 10 只 ETF 的评估结果：

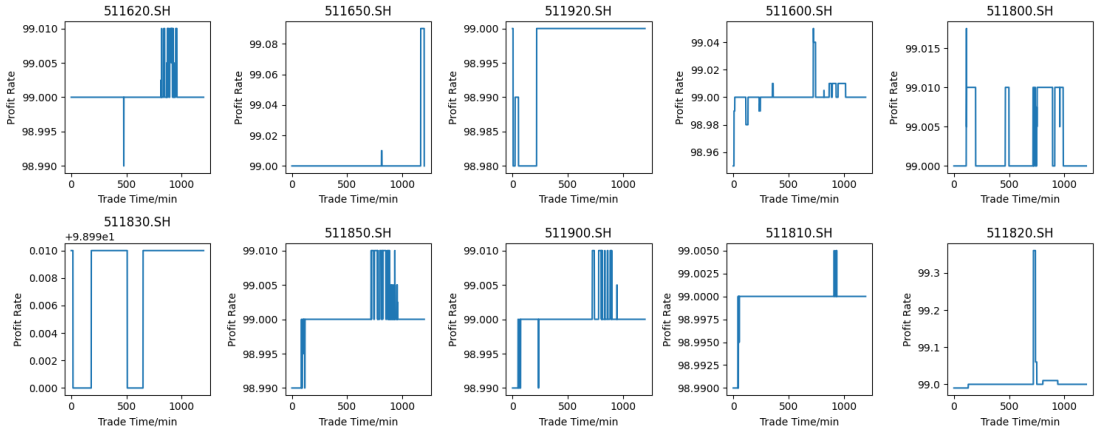

图 5 前 10 只 ETF 的利润率（百分比）

排行序号	交易编号	名称	3 月度区间累计利润	最大回撤 MDD
1	511620.SH	货币基金 ETF	¥49,185,114.07	0.0199
2	511650.SH	华夏快线 ETF	¥43,233,959.89	0.0899
3	511920.SH	广发货币 ETF	¥42,254,575.77	0.0199
4	511600.SH	货币 ETF	¥34,796,320.47	0.0999
5	511800.SH	易方达货币 ETF	¥31,801,293.90	0.0175
6	511830.SH	华泰货币 ETF	¥28,694,542.94	0.0100
7	511850.SH	财富宝 ETF	¥24,687,807.30	0.0200
8	511900.SH	富国货币 ETF	¥24,588,859.03	0.0199
9	511810.SH	理财金货币 ETF	¥24,448,888.75	0.0149

10	511820.SH	鹏华添利 ETF	¥24,074,819.19	0.3687

表 1 前 10 只 ETF 的最大回撤 MDD

为了评估每只 ETF 的短期表现，我们最终选取 2023 年 8 月 1 日至 8 月 7 日五个交易日内的高频数据，得到以下关于前 10 只 ETF 的单位时间收益：

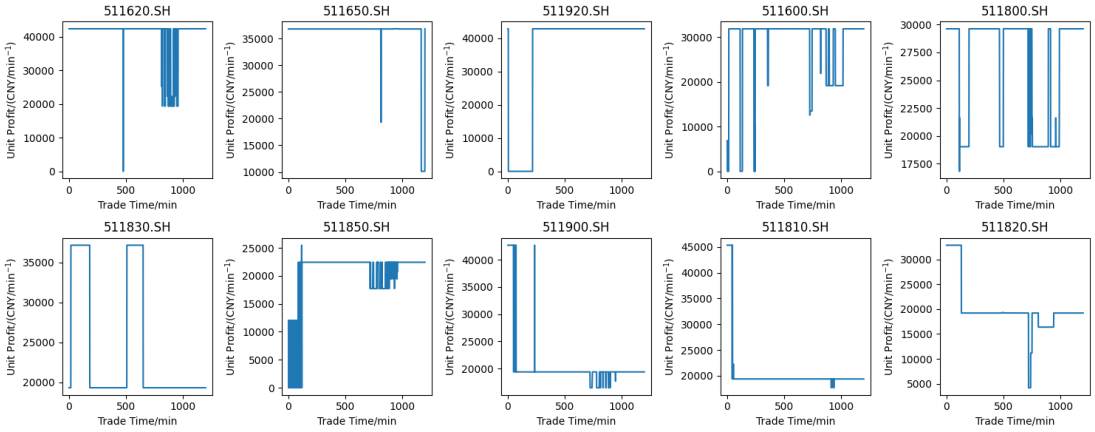

图 6 短期表现下前 10 只 ETF 的单位时间收益

5.2 问题二模型的建立与求解

在任务二中，我们专注于跨境 ETF 的交易策略模型的建立，深入研究了 T+0 交易模式。该模式与一般 ETF 的 T+1 结算规则不同。由于跨境 ETF 无涨跌停限制，因而在 T+0 交易模式下，它们能够在同一交易日内完成交易和结算，从而为套利提供了更及时的机会。针对此类市场动态，我们构建了新的模型，以识别并利用这些套利机会，适合于追求即时交易执行的投资者和基金。

在理论框架方面，我们采纳分形市场理论，作为对传统有效市场假设的补充。分形市场理论认为价格变动在不同时间尺度上显示出规律性，而非完全随机。基于此，我们通过 $H(q)$ 矩阵的引入，量化了市场的分形特征，并将其应用于动态调整交易策略。

在技术选择上，我们采用了 LSTM 网络处理时间序列数据。LSTM 网络因其在长期依赖关系中的稳定性以及抗梯度消失的能力而被选用，使模型能够在复杂的市场条件下捕捉到关键信息，从而提高了市场趋势预测的准确率。

综合 LSTM 的时间序列分析能力与分形理论的市场规律性，我们构建了一个针对跨境 ETF 交易的“黑白结合”的数学模型。该模型结合了长期记忆能力和规律性分析，以适应剧烈波动的跨境 ETF 市场，提升了套利策略的效果。通过实证研究，我们验证了模型在实现交易成本和速度优化中的实用性和效率。

5.2.1 数据预处理

相较于任务一，为深入分析跨境 ETF 的交易策略，我们在任务一的时间序列数据 X 与 Y 的基础上，新增以下变量：

1. 复权单位净值 N_t：日级别数据，计算反映长期投资效果的单位净值。
2. 贴水 D_t：计算式为 $D_t = P_t - N_t$，其中 P_t 是 t 时刻 ETF 的市场价格，N_t 是复权单位净值。

3. 贴水率 R_t：计算式为 $R_t = \frac{D_t}{N_t}$，表示贴水占单位净值的比例。

4. 增长率 G_t：计算式为 $G_t = \frac{N_t - N_{t-1}}{N_{t-1}}$，表示单位净值的日增长率。

在构建模型时，所有这些变量都将被纳入以确保模型能够全面地评估和预测跨境 ETF 的市场行为。对于新引入的一级市场数据，我们通过同花顺 iFinD 平台进行采集。

5.2.2 基于长短期记忆网络(LSTM)的改进型价格序列预测模型

为精确预测跨境 ETF 在二级市场上的交易价格，我们构建了一种基于长短期记忆网络（LSTM）的预测模型。[6] LSTM 作为一种特殊的循环神经网络（RNN），具备处理和预测时间序列数据的能力。此模型通过在内部状态间传递信息，有效地缓解了传统 RNN 在面对长时间序列时遇到的梯度消失问题。借助于这种机制，LSTM 能够学习并记忆历史数据中的长期依赖关系，对未来的市场价格走势进行预测。[7]

一、神经网络模型建构

在我们的模型中，我们采用了一个序贯模型（"sequential"），包含了如下层级结构：

1. LSTM 层（lstm）：此层含有 50 个 LSTM 单元，负责处理输入的时间序列数据，通过其循环连接结构，能够在时间维度上传递信息，保留并学习长期和短期的数据模式。
2. 全连接层（dense）：含有 20 个神经元，接收 LSTM 层的输出，并进一步映射到更高级的特征空间。
3. 输出层（dense_1）：最终的输出层包含 3 个神经元，对应于我们预测的未来三个时间点的价格。

整个模型共计参数量为 11483 个，表示模型在训练过程中需要估计并调整的权重和偏置数量，这些参数全都是可训练的。

在实际应用中，模型的输入由 $y_{t-1}, y_{t-2}, y_{t-3}, y_{t-4}, y_{t-5}, y_{t-6}$ 序列构成，这些输入代表了时间序列中当前时间点之前的六个连续时间点的价格，作为高频因子输入到模型中。而模型的输出是对未来三个时间点价格的预测：$y_{t+1}, y_{t+2}, y_{t+3}$。

二、神经网络模型训练

在对模型进行训练时，共迭代了 180 个 epoch，并采用了每批次包含 1000 条数据的 batch size。这一训练过程涵盖了全部 100 余个跨境 ETF 的三个月高频交易数据，确保了模型训练的数据覆盖面广泛且具有时效性。在训练集和验证集分割比例为 0.8:0.2 的设置下，模型通过调校后，损失函数（loss）值稳定在 0.07，这一结果表明了模型在捕捉数据集内在规律方面的高效性。以下表格概括了模型的关键训练效果参数：

指标	训练集	验证集
MSE	0.07	0.07
R^2	0.89	0.87

表 2 神经网络模型训练结果

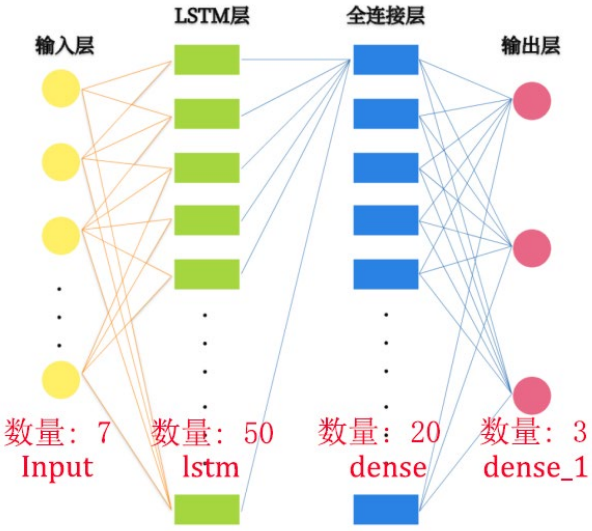

图 7 'sequential'神经网络模型结构图

在后续的测试环节中，我们使用该模型完成 2023 年 8 月 1 日至 2023 年 8 月 7 日共五个交易日的模拟高频交易，通过所选取出的 10 只最适合交易的跨境 ETF，预测第二市场的价格数据结果如下图所示：

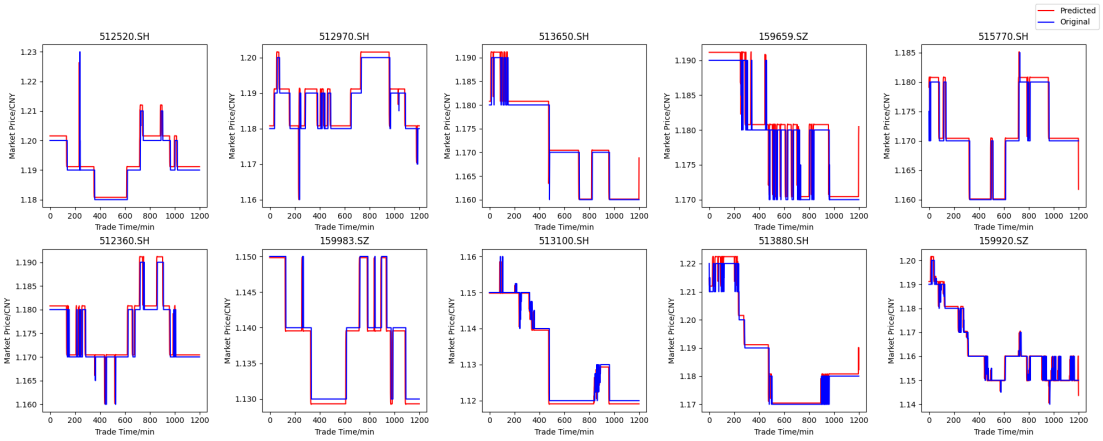

图 8 神经网络模型预测结果

三、加权平均修正

在跨境 ETF 定价模型中，本模型采用指数衰减加权平均法，通过设置权重 $w_n = e^{-\lambda n}$ 以整合 $t+1$ 至 $t+3$ 时刻的价格预测 $y_{t+1}, y_{t+2}, y_{t+3}$。修正后的价格 y_{t+1}' 计算如下：

$$y_{t+1}' = \sum_{n=1}^{3} w_n P_{t+n} \Big/ \sum_{n=1}^{3} w_n \quad (14)$$

其中 λ 是衰减率参数。此方法基于指数衰减函数赋予近期预测更高的权重，从而反映出时间序列数据的近因效应；同时能够平衡近期与远期预测的权重，从而提高预测的准确性与鲁棒性。

5.2.3 基于 Hurst 矩阵与折溢价率、跟踪误差的交易量因子影响算法

一、平均折溢价率与跟踪误差

跨境 ETF 由于涉及到不同市场的交易，其价格动态会受到多重因素影响，包括汇率波动、市场开闭市时间差异、信息传递延迟等。因此，相比于国内的 ETF，跨境 ETF 的折溢

价率通常更高，这是由于市场对这些因素的即时反应和对未来不确定性的预期。

高折溢价率意味着市场价格和净资产价值(NAV)之间的差距加大，这不仅反映了市场的波动性，也表明存在更多的套利空间。同时，跟踪误差作为度量 ETF 性能与其基准指数性能差异的指标，其大小直接反映了 ETF 对基准指数跟踪的准确性。较大的跟踪误差表明 ETF 价格与基准指数之间的偏差较大，这通常与市场波动性和交易机会正相关。因此，我们假设，投资者在观察到较高的折溢价率和跟踪误差时，往往预测市场存在更多的套利和交易机会，从而可能增加交易行为。

折溢价率是度量跨境 ETF 市场价格与其一级市场净资产价值（NAV）的偏离程度的指标，其数学表达式为：

$$P = \left(\frac{P_m - P_{nav}}{P_{nav}} \right) \times 100\% \quad (15)$$

其中，P 代表折溢价率，P_m 代表二级市场价格，P_{nav} 代表一级市场的净资产价值（NAV）。该值的正负和大小可以帮助投资者判断 ETF 是否存在溢价或折价，以及这一偏差的程度。

跟踪误差反映了 ETF 价格与其追踪的指数之间的波动差异，衡量了 ETF 表现相对于其基准指数的波动性大小的量化指标。可以通过以下数学表达式计算：

$$TE = \sigma \left(\frac{P_m - P_{nav}}{P_{nav}} \right) \quad (16)$$

其中，TE 是跟踪误差，σ 表示标准差，P_m 是二级市场价格，P_{nav} 是一级市场的净资产价值（NAV）。

二、时间序列分形性：使用 MF-DFA 方法得到广义 Hurst 函数

MF-DFA 方法分为四个主要阶段，输入时间序列来输出广义 Hurst 函数 $H(q)$：[8]

1. 构建累积离差序列：从时间序列 $\{x_i\}$ 构建累积离差序列 $\{y_i\}$，利用公式：

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i \quad (17)$$

此处 x_i 表示原始时间序列的第 i 个点，$\bar{x}$ 是序列的均值，而 y_i 是累积离差。

2. 分割序列并拟合多项式：将 $\{y_i\}$ 分割为 $2N_s$ 个互不重叠的子区间，每个长度为 s，并在每个子区间内进行最小二乘多项式拟合，得到降趋势函数 $y_v(i)$。

3. 计算降趋势方差：在每个子区间内计算降趋势方差 $F^2(s,v)$：

$$F^2(s,v) = \frac{1}{s} \sum_{i=1}^{s} \left\{ y((v-1)s+i) - y_v(i) \right\}^2 \quad (18)$$

对于正向和反向分割的子区间分别计算，其中 v 表示子区间的索引。

4. 计算波动函数并得到 $H(q)$：对于不同的 q 阶，计算 $F_q(s)$ 并建立 $F_q(s)$ 与 s 的幂律关系，最终求得广义 Hurst 函数 $H(q)$：$F_q(s) \sim s^{H(q)}$。这里 $H(q)$ 为广义 Hurst 指数[9]，$F_q(s)$ 是 q 阶波动函数，表示了在不同尺度 s 下的时间序列波动的标度行为。通过

这一方法，可以从时间序列中提取多重分形特征[10]，并用以构建跨境 ETF 交易策略。

三、交易量影响因子的计算

计算跨境 ETF 交易策略中的交易量影响因子 f_1, f_2, f_3, f_4 涉及以下四个步骤：

步骤 1：Hurst 函数计算

首先，定义七个时间序列，每个序列对应不同的市场数据。

二级市场数据的时间序列（以分钟为单位）：

- $X_1[t]$ 为每分钟收盘价、开盘价、最高价、最低价的算术平均值，表达式：

$$X_1[t] = \frac{1}{4}(P_{\text{close}}[t] + P_{\text{open}}[t] + P_{\text{high}}[t] + P_{\text{low}}[t]) \quad (19)$$

- $X_2[t]$ 为每分钟的总成交数，简记为 $X_2[t]$；
- $X_3[t]$ 为每分钟 ETF 的成交量，简记为 $X_3[t]$；

一级市场数据的时间序列（以交易日为单位）：

- $X_4[t]$ 为复权单位净值 N_t，表示长期投资效果的单位净值；
- $X_5[t]$ 为贴水 D_t，计算为 $D_t = P_t - N_t$；
- $X_6[t]$ 为贴水率 R_t，计算为 $R_t = \frac{D_t}{N_t}$；
- $X_7[t]$ 为增长率 G_t，计算为 $G_t = \frac{N_t - N_{t-1}}{N_{t-1}}$。

对于每个时间序列 $X_i[t]$，使用 MF-DFA 方法计算相应的 Hurst 指数 $H_i[q]$，其中 i 代表不同的时间序列（1 到 7），q 代表不同的阶数（1 到 7），计算公式如下：

$$H_i[q] = \text{MF-DFA}(X_i[t], q) \quad (20)$$

其中，$\text{MF-DFA}(X_i[t], q)$ 是上述执行多重分形解析的函数，对于序列 $X_i[t]$ 和阶数 q，它将输出一个 Hurst 指数，表示该时间序列的长期记忆性。

步骤 2：构建 Hurst 矩阵

构建一个 7×7 的 Hurst 矩阵 $\mathbf{H}$，其中 $\mathbf{H}_{ij}$ 表示第 i 个时间序列对于 $q = j$ 的 Hurst 指数：

$$\mathbf{H} = \begin{bmatrix} H_1(1) & H_1(2) & \cdots & H_1(7) \\ H_2(1) & H_2(2) & \cdots & H_2(7) \\ \vdots & \vdots & \ddots & \vdots \\ H_7(1) & H_7(2) & \cdots & H_7(7) \end{bmatrix} \quad (21)$$

步骤 3：Hurst 矩阵调整

为了强调分形性与矩阵内元素的正相关性，我们对 Hurst 矩阵进行以下调整：设定一个新的矩阵 $\mathbf{H}'$，其元素 $\mathbf{H}_{ij}'$ 通过下述方式计算：

$$\mathbf{H}_{ij}' = |\mathbf{H}_{ij} - 0.5| \quad (22)$$

通过这个转换，我们确保了 Hurst 矩阵的分形性（正分形、负分形、零分形）被强调和量化。一般而言，在 $\mathbf{H}_{ij}$ 大于 0.5 时，我们的处理意味着市场展现出持续的趋势（正分形），小于 0.5 表示市场的反转趋势（负分形），而等于 0.5 则代表完全的随机行走（零

分形）。这种处理方式简化了分形性的判断，并允许这种性质与矩阵内元素的值成正相关，这对于后续的交易策略构建具有潜在价值。 经过这样的处理，矩阵 $\mathbf{H}'$ 会有更大的元素值，指示更强的市场趋势持续性或反转趋势，这对于预测市场行为和制定交易策略至关重要。

步骤 4：Hurst 矩阵与平均折溢价率、跟踪误差的后处理

定义转换函数 $\mathrm{tr}(c1,c2)$，将折溢价率 c_1 和跟踪误差 c_2 转换为向量 $\mathbf{V}$：

$$\mathbf{V} = \mathrm{tr}(c_1,c_2)=[c_1,c_2,c_1\cdot c_2,c_1^{\,2},c_2^{\,2},\sqrt{|c_1|},\sqrt{|c_2|}] \quad (23)$$

将向量 $\mathbf{V}$ 与矩阵 $\mathbf{H}'$ 进行矩阵乘法，得到新的向量 $\mathbf{W}$：

$$\mathbf{W} = \mathbf{H}'\cdot\mathbf{V} \quad (24)$$

对向量 $\mathbf{W}$ 进行后处理操作：

$$\mathrm{shuffle}(\mathbf{W}) = \mathbf{W}' \quad (25)$$

其中 shuffle 是一个基于种子 seed 的随机打散函数，确保每次打散的一致性。

向量聚合：

$$m_{i'} = \left\lfloor \frac{1}{n} \right\rfloor \sum_{j=(i-1)n+1}^{in} w_{j'} \quad (26)$$

对于 $i = 1,2,...,4$， $w_{j'}$ 是打散后向量的第 j 个元素。

缩放聚合值：

$$m_i = \frac{(m_{i'} - \min(\mathbf{M}'))}{\max(\mathbf{M}') - \min(\mathbf{M}')} \times (2-1) + \exp(\arctan(mean(\mathbf{W}'))\cdot\frac{2}{\pi} - 1) \quad (27)$$

其中 $\mathbf{F}' = \{f_{1'}, f_{2'}, \ldots, f_4\}$ 是聚合后的向量，最终得到交易量影响因子 f_1, f_2, f_3, f_4。

此过程的目的是通过打散和聚合减少数据噪声，提取出稳定趋势，然后通过缩放确保所有因子值都在相同的数值范围内，有利于保持数值的稳定性。

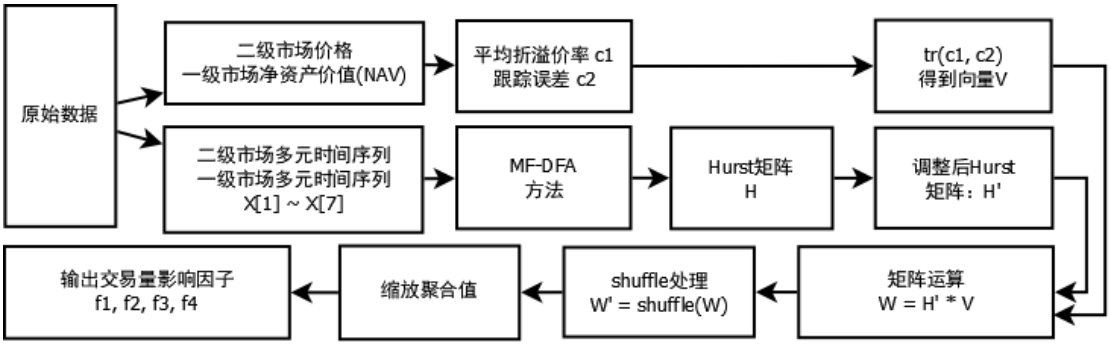

图 9 任务二交易量因子影响算法流程图

5.2.4 交易策略的更新

任务二的交易逻辑维持了与任务一相同的基本限制条件和成本变量，以促进比较分析的连续性。针对跨境交易所需的特殊逻辑，交易策略经过优化以反映更高频的波动和复杂的市场动态。策略中的交易量调整为 $f_i \cdot d_i$ 的形式，其中 f_i 表示上文中的策略因子，d_i 代

表与任务一相同的决策变量，以动态地体现市场规律性和信心水平的提升。

具体地，交易量的动态调整遵循以下规则：

折价交易判定条件为$\delta_t > \delta_{th}$，一级市场的买入数量和二级市场的卖出数量分别为：

$$Q_{buy,1} = f_1 \cdot d_1 \cdot \ln(|\Delta y_t| + 1) \quad (28)$$

与$Q_{sell,2} = f_2 \cdot d_2 \cdot \ln(|\Delta y_t| + 1)$ (29)。

溢价交易判定条件为$\delta_t < -\delta_{th}$，二级市场的买入数量和一级市场的卖出数量分别为：

$$Q_{buy,2} = f_3 \cdot d_3 \cdot \ln(|\Delta y_t| + 1) \quad (30)$$

与$Q_{sell,1} = f_4 \cdot d_4 \cdot \ln(|\Delta y_t| + 1)$ (31)。

所有的交易量都通过算法动态计算，确保了对市场波动的敏感性和策略的适时调整。

最后，针对跨境 ETF 的 T+0 交易规则，本策略特别考虑了允许当日买入后即可卖出的机制。具体而言，每笔交易所依照的价格为当前的一级、二级市场价格，而不是第二日开盘时刻的对应价格。

另外，除已知的 101 只跨境 ETF 以外，事实上“大湾区 ETF”等由于涉及标的资产为香港、澳门等非中国大陆地区的资产指标，且在中国大陆交易市场中进行交易，因而在本模型中也被列入跨境 ETF 范围中。

5.2.5 检验结果

与问题一相同，我们模拟了 2023.8.1 至 2023.8.7 共五个交易日的溢价交易和折价交易，对所有的跨境 ETF 候选项进行测试，依据累计交易利润，筛选出 10 个最适合进行套利的跨境 ETF。下表列出了这 10 个跨境 ETF 使用问题二模型得出的一周的累计利润和对应的最大回撤，以及问题一得出的一周累计利润的结果对比。

排行序号	交易编号	名称	模型二累计利润	模型一累计利润	最大回撤 *MDD*
1	512520.SH	MSCIETF	¥871,156.71	¥139,743.99	0.0407
2	512970.SH	大湾区 ETF	¥823,711.73	¥135,940.11	0.0333
3	513650.SH	标普 500ETF 基金	¥793,023.63	¥155,878.73	0.0252
4	159659.SZ	纳斯达克 100ETF	¥784,975.01	¥166,530.66	0.1680
5	515770.SH	摩根 MSCIAETF	¥783,469.35	¥143,680.68	0.0211
6	512360.SH	平安 MSCI 国际 ETF	¥773,747.76	¥141,633.50	0.0252
7	159983.SZ	粤港澳大湾区 ETF	¥721,515.29	¥130,120.40	0.0174
8	513100.SH	纳指 ETF	¥718,654.17	¥164,785.73	0.0345
9	513880.SH	日经 225ETF	¥713,753.28	¥162,053.29	0.0410
10	159920.SZ	恒生 ETF	¥706,917.86	¥140,942.43	0.0500

表 3 前 10 只跨境 ETF 的数据

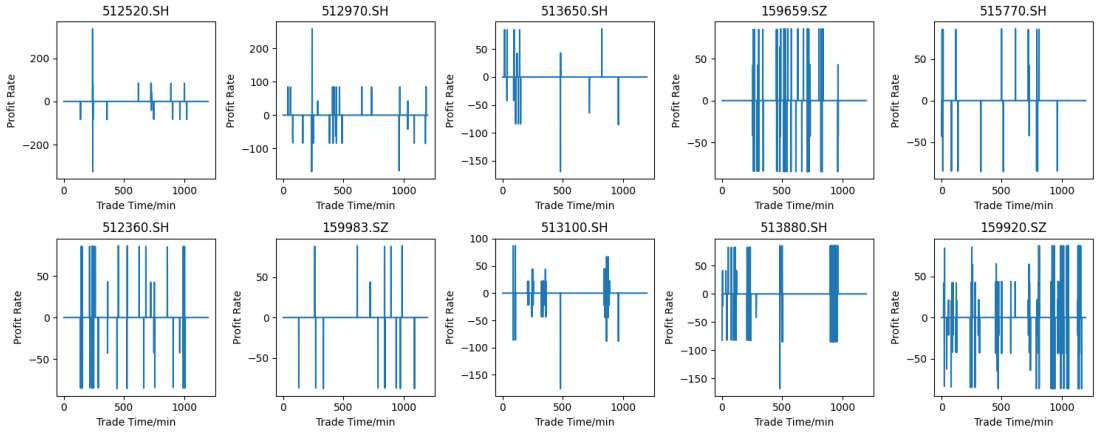

图 10 前 10 只跨境 ETF 模拟一周交易的收益率

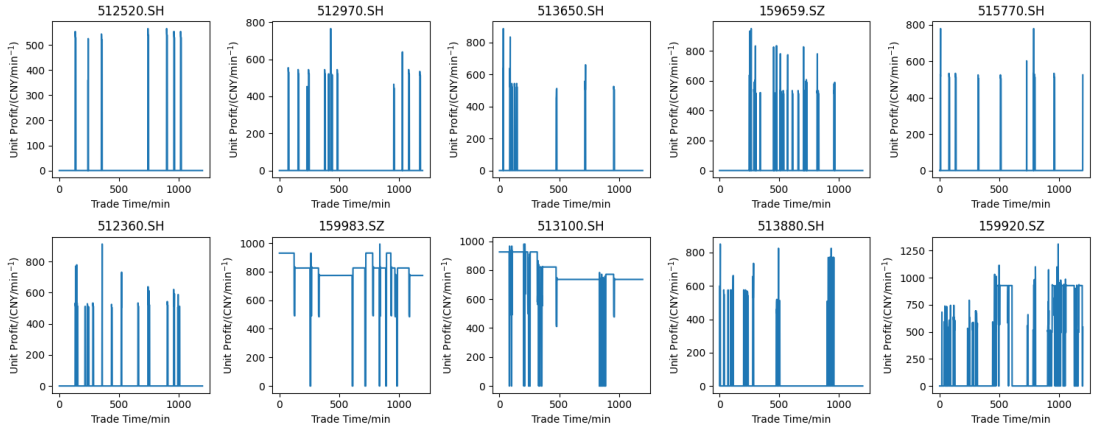

图 11 前 10 只跨境 ETF 模拟一周交易的单位时间利润

上述结果表明，模型二在短期交易累计利润方面显著超越了问题一的结果，表明该策略能够有效识别并利用跨境 ETF 的套利机会。具体而言，问题二的模型在一周内对选定的跨境 ETF 进行模拟交易，其累计利润与问题一相比均实现了大幅度增长，这不仅证明了策略在盈利能力上的提升，也反映了算法在捕捉和执行套利机会方面的高效性。

此外，问题二模型对应的最大回撤数据表明，所选跨境 ETF 的波动性较境内 ETF 更为显著，这一特征与我们模型的优化策略相符，指出了跨境 ETF 在波动性上的优势和由此产生的更大套利空间。因此，验证结果充分展示了问题二模型在把握高波动性市场中的套利机遇方面的优越性能。

5.3 问题三模型的建立与求解

跨市场套利的不同研究揭示了国内外金融市场联动性，对我国证券市场国际化趋势至关重要。随着 QDII 和 QFII 额度提升，境内外资本流动性增强，市场联动性随之加强。股指期货作为金融期货的重要组成部分，以股票指数为标的。由于不同市场间股指期货与跨境 ETF 的标的资产相同，而两者间的市场价格存在差异与波动，这为投资者提供了跨市场套利的可能性。因而我们的模型同样通过比较一级市场的 ETF 基金净值和二级市场的 ETF 市价，以识别套利机会。

在我们的跨市场套利模型中，我们关注了境内金融期货交易所的四只关键股指期货：

中金所上证 50、沪深 300、中证 500、中证 1000 指数。我们将这些股指期货与美国和香港市场中与之相同或相关的 87 只跨境 ETF 进行了匹配。

在处理跨市场套利时，我们必须考虑到货币汇率的波动对套利机会的影响。因此，我们采取了将所有交易数据转换为以人民币（CNY）为基准的值的方法。这一步骤至关重要，因为它消除了由于货币汇率波动带来的潜在误差，从而使我们的套利策略更加精确和可靠。

此外，我们在模型中引入概念——"资产间比例系数"，在时间序列的起点 $t=0$ 定义：

$$\beta = \frac{P_{futures}(t=0)}{P_{ETF}(t=0)} \quad (32)$$

其中 $P_{futures}(t=0)$ 是境内股指期货的单位价格，而 $P_{ETF}(t=0)$ 是境外跨境 ETF 的单位价格。这个系数用于调整交易量，确保交易中境内股指期货与境外跨境 ETF 背后的标的资产在数量上保持一致。这不仅确保了我们的套利策略在逻辑上的连贯性，而且在实际操作中，使用这个比例系数进行交易量的计算，是实施跨市场对冲策略的关键。这种对冲策略考虑到了市场的波动性和交易时的市场影响，有助于在最大化套利收益同时控制风险。

5.3.1 数据预处理

我们首先对每个股指期货 X 与跨境 ETF Y 进行两两组合。对于每一对组合，我们处理时间序列数据以获得每个交易时间点 t 的统一市场价格和交易量。

市场价格 P 的计算如下：对于股指期货 X 和跨境 ETF Y 的每个交易时间点 t，我们取其开盘价 O，收盘价 C，最高价 H，和最低价 L 的算术平均值。因此，股指期货的市场价格 $P_x[t]$ 和跨境 ETF 的市场价格 $P_y[t]$ 可以表示为：

$$P_x[t] = \frac{O_x[t] + C_x[t] + H_x[t] + L_x[t]}{4}, \quad P_y[t] = \frac{O_y[t] + C_y[t] + H_y[t] + L_y[t]}{4} \quad (33)$$

其中：$O_x[t], C_x[t], H_x[t], L_x[t]$ 分别是时间点 t 的股指期货 x 的开盘价、收盘价、最高价和最低价。$O_y[t], C_y[t], H_y[t], L_y[t]$ 分别是时间点 t 的跨境 ETF y 的开盘价、收盘价、最高价和最低价。

对于交易量 Q，我们直接取时间序列中每个交易时间点 t 的交易量，其中 $Q_x[t]$ 为股指期货的交易量，$Q_y[t]$ 为跨境 ETF 的交易量。

5.3.2 市场波动指数 δ_{mkt} 及其回归预测模型

我们定义 $t+1$ 时刻下市场波动指数 $\delta_{mkt}[t+1]$ 为：

$$\delta_{mkt}[t+1] = P_x[t] - \left(\frac{P_y[t]}{\beta} \right) \quad (34)$$

其中，$P_x[t]$ 是股指期货在时间点 t 的市场价格，$P_y[t]$ 是跨境 ETF 在时间点 t 的市场价格，而 β 是资产间比例系数，如前所述。这个指数通过标准化方法衡量境外市场和境内市

场的价格差异，为识别正向和反向套利机会提供了依据。

为了预测市场波动指数的未来值，我们使用与问题二相同的神经网络结构进行训练。但与问题二不同，我们为问题三构造的网络接收一系列连续的市场波动指数 $\delta_{mkt}[n], n = t-6, t-5, ..., t$ 共 7 个数据点作为输入，并输出对未来三个时间点 $\delta_{mkt}[n], n = t+1, t+2, t+3$ 的预测值。这个序贯模型包含一个 LSTM 层，一个全连接层，以及一个输出层，共计参数量为 11483 个。

在新模型训练过程中，我们将数据集分为训练集和验证集，比例为 0.8:0.2，并迭代 180 个 epoch。训练结果显示，模型的损失函数 loss 值为 0.15，模型的关键训练效果参数如下所示：

指标	训练集	验证集
MSE	0.15	0.23
R^2	0.76	0.85

表 4 神经网络模型训练结果

与问题二的模型相同，对于最终得到的 $\delta_{mkt}[t+1]'$，我们同样使用指数加权平均法修正，确定因子 $w_n = e^{-\lambda n}$，计算公式如下：

$$\delta_{mkt}[t+1]' = \sum_{n=1}^{3} w_n \delta_{mkt}[t+n] \Big/ \sum_{n=1}^{3} w_n \quad (35)$$

在后续的测试环节中，我们使用该模型完成 2023 年 9 月 18 日至 2023 年 10 月 30 日各境内外市场间、以日为单位的各交易日的模拟交易，通过所选取出的 10 只最适合交易的组合，预测市场波动指数的数据结果如下图所示：

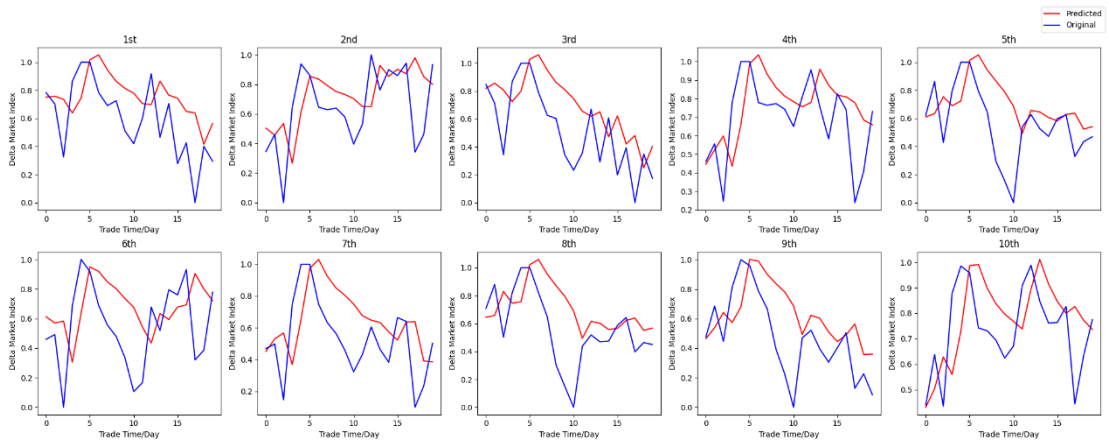

图 12 市场波动指数预测结果

5.3.3 交易量因子影响计算

首先我们引入了一系列指标以量化股指期货与跨境 ETF 之间的交易关系。

1. 价格相关性 c_1 及其显著性 c_2

价格相关性 c_1 通过计算两个时间序列的 Pearson 相关系数来衡量，其数学表达式为：

$$c_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \quad (36)$$

其中，x_i 和 y_i 分别代表股指期货和跨境 ETF 在第 i 个时间点的价格，$\bar{x}$ 和 $\bar{y}$ 分别代表对应时间序列的平均价格。此指标反映两个时间序列价格变动的同步性，其值越接近于 1，表明两个市场的价格变动越同步，为基于价格关联性的套利策略提供了高置信度。

显著性指标 c_2 是相关性的概率值，用于评估相关系数 c_1 的统计显著性。其数学表达式为：$c_2 = P$(数据在零相关假设下出现当前或更极端相关系数的概率)

2. 协整关系 c_3 及其显著性 c_4

我们采用问题一中相同的 Engle-Granger 检验方法来评估股指期货与跨境 ETF 之间的协整关系。通过这种方法，我们获得两个关键指标：c_3 协整测试得分，衡量两个时间序列之间长期稳定关系的强度；以及 c_4 协整关系的概率值，提供了这种长期关系统计显著性的度量。

上述指标中，较高的 c_2、c_4 值分别表明两市场间价格变动同步性、协整关系不显著，这意味着市场间价格波动更剧烈，套利机会更高。

3. 回归斜率 c_5

回归斜率通过线性回归分析股指期货和跨境 ETF 的价格时间序列获得，数学表达式：

$$c_5 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \quad (37)$$

该斜率衡量了一个市场价格变动对另一个市场价格变动的预期影响，提供价格变动传导效应的量化度量。

4. 成交量比率 c_6

成交量比率量化了跨境 ETF 与股指期货之间的平均成交量之比，数学表达式：

$$c_6 = \frac{\sum Q_{y[t]}}{\sum Q_{x[t]}} \quad (38)$$

该比率反映了两个市场之间流动性的相对大小，高成交量比率可能指示着较高的市场活跃度和潜在的套利机会。

最后我们通过对上述指标 $c_1, c_2, c_3, c_4, c_5, c_6$ 取绝对值后，计算其算术平均值，得到用于交易策略的乘积因子 f。上述指标形成了一个量化分析框架，旨在充分利用跨境 ETF 与国际市场的联动性及其对 A 股市场的影响。这个框架不仅捕捉了市场的波动性和流动性，还精确地反映了市场之间的关联性，为我们的交易策略提供了坚实的量化基础，确保了策略的科学性和实用性。

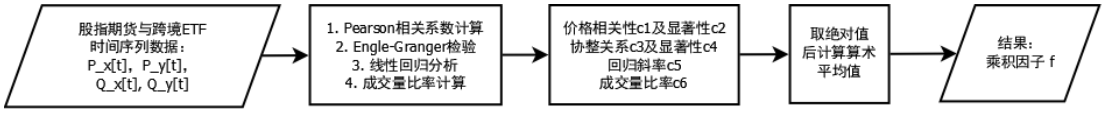

图 13 任务三交易量因子影响计算示意图

5.2.4 针对跨市场交易的改进型交易策略

在任务三的交易策略中，我们针对跨境市场交易套利，在超参数更新、交易策略更新和持仓补仓策略更新共三个方面进行适应性的调整。

一、超参数更新

考虑到国内外市场的交易成本和流动性限制，我们新引入一系列超参数来模拟实际交易环境。以下是超参数的数学表述及其法规背景解释：

对于国内市场（股指期货）：

- $\text{eff}1\_\text{dom}$：固定成本，包括 ETF 申购、赎回费用等，设为 0 元，反映了无固定交易费用的假设。
- $\text{eff}2\_\text{dom}$：印花税率，0.1%，基于交易额百分比，符合国内证券交易税率规定。
- $\text{eff}3\_\text{dom}$：交易佣金，设为 0.2%，反映了经纪服务的成本。
- $\text{eff}4\_\text{dom}$：市场冲击成本，设为 0.05%，考虑了大额交易对市场价格的影响。
- $\text{trade\_regulation\_dom}$：流动性限制指标，设为 70%，假设单次交易量不超过每分钟总交易量的 70%，以符合市场流动性管理规则。

对于境外市场（跨境 ETF）：

- $\text{eff}1\_\text{abr}$：固定成本，以人民币计，设为 0.71781 元，反映了以美元计价的交易手续费按当日汇率转换的结果。
- $\text{eff}2\_\text{abr}$：ETF 管理费用，设为 0.1%，这是跨市场 ETF 常见的管理费率。
- $\text{eff}3\_\text{abr}$：交易佣金，设为 0.1%，与国内市场相同，反映了二级市场的交易成本。
- $\text{eff}4\_\text{abr}$ 价差成本，设为 0.02%，体现了买卖价差对交易成本的影响。
- $\text{trade\_regulation\_abr}$：流动性限制指标，设为 70%，与国内市场相同，保持了对交易量的合理限制。
- 市场波动指数阈值 δ_{mkt-th}：设定为 1e-5，基于市场最小变动价位，用以指导何时进行交易的决策点。

这些超参数的设定基于现行的金融市场法规和实际交易成本，确保模型的现实适用性和合规性。

二、交易判断策略更新

我们在任务二模型的基础上，根据任务三的实际情况，将交易判断依据由改为市场波动指数 δ_{mkt}；策略中的交易量由 $f_i \cdot d_i$ 的形式调整为 $f \cdot d_i$。具体规则如下：

以国内股指期货价格为基准，跨境 ETF 相对折价交易判定条件为 $\delta_{mkt}[t] > \left|\delta_{mkt-th}\right|$，国内市场股指期货的买入数量和国际市场跨境 ETF 的卖出数量分别为：

$$Q_{buy,1} = f \cdot d_1 \cdot \ln(|\delta_{mkt}[t]|+1) \quad (39)$$

$$\text{与 } Q_{sell,2} = f \cdot d_2 \cdot \ln(|\delta_{mkt}[t]|+1) \quad (40)\text{。}$$

相对溢价交易判定条件为 $\delta_{mkt}[t] < -\left|\delta_{mkt-th}\right|$，国际市场跨境 ETF 的买入数量和国内市场股指期货的卖出数量分别为：

$$Q_{buy,2} = f \cdot d_3 \cdot \ln(|\delta_{mkt}[t]|+1) \quad (41)$$

$$\text{与 } Q_{sell,1} = f \cdot d_4 \cdot \ln(|\delta_{mkt}[t]|+1) \quad (42)\text{。}$$

在本模型中，我们假设可以忽略国内 T+1 交易结算规则的约束，因为模型旨在捕捉并利用跨市场的即时价格差异进行套利，这类套利交易通常在同一交易日内完成，不涉及隔夜持仓，从而不受 T+1 规则的实质性影响。

三、Dynamic Barrier Positioning (DBP)持仓补仓策略

不同于任务一、任务二仅在交易日结束时调整的策略，任务三更新后的 DBP 持仓补仓策略能更有效地利用市场信息，减少持仓误差，提高套利效率。设定持仓量的动态调整策略为：

当套利指标 $\delta_{mkt}[t]$ 越过 $|\delta_{mkt-th}|$ 区间时，即从套利区间进入无套利区间，我们按照以下公式调整持仓量 $Q_{adjust,t}$：

$$Q_{adjust,t} = \Theta^* - \Theta_t \quad (43)$$

其中，Θ^* 为目标持仓量，Θ_t 为当前持仓量。每当 $\delta_{mkt}[t] > |\delta_{mkt-th}|$ 或 $\delta_{mkt}[t] < -|\delta_{mkt-th}|$ 时，我们即时进行持仓量的调整，以实现在跨市场交易中的持仓优化。

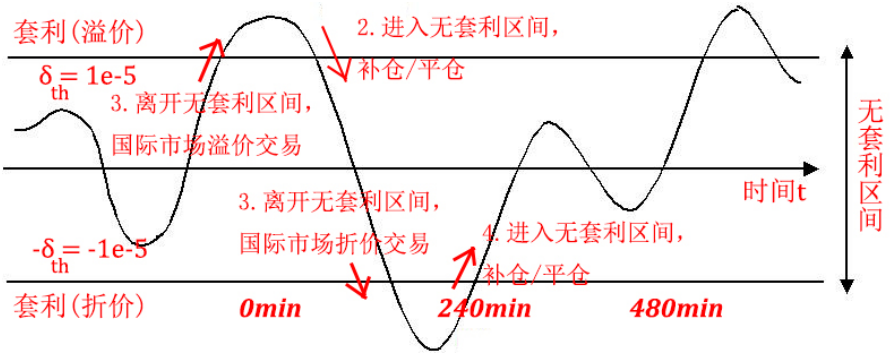

图 14　Dynamic Barrier Positioning(DBP)持仓与补仓策略简要说明

最后根据上述结果，可以得到任务三更新后的交易策略，下图中红色文字是基于任务一、任务二的交易策略的更新：

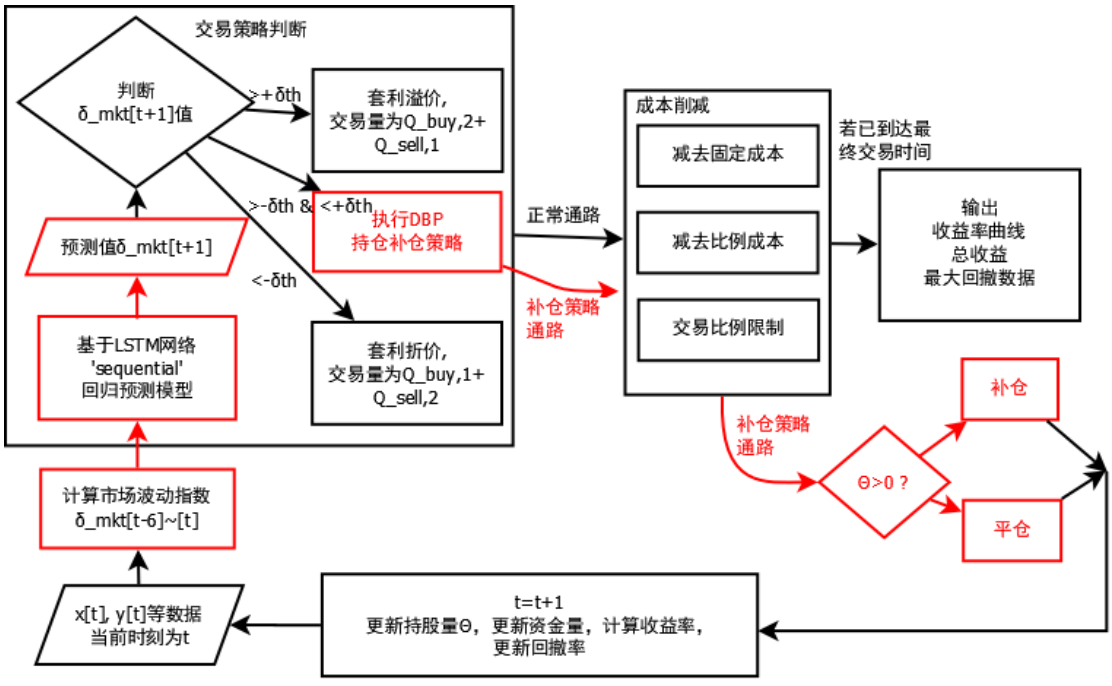

图 15 任务三改进型交易策略流程图

5.2.5 数据运行结果与组合确定

一、数据运行结果

对于任务三的交易策略模型，我们针对上述 4 个境内股指期货和 87 只境外跨境 ETF 的共 $4\times87=348$ 个组合，利用 2023 年 9 月 18 日至 2023 年 10 月 30 日的日度交易日数

据，考虑了不同交易市场的交易日差异，进行模拟交易。依据累计交易利润，筛选出 10 个纳入候选的组合。以下列出了这 10 个组合的累计利润、模拟交易区间的最大回撤、收益率随交易时刻变化的曲线：

排行	交易编号	名称	区间累计利润	股指期货最大回撤	跨境 ETF 最大回撤
1	IM2311.CFE, 09151.HK	中证 1000, PP 科创 50-U	¥1,074,792.12	0.0685	0.0689
2	IM2311.CFE, 09173.HK	中证 1000, PP 中新经济-U	¥1,016,364.85	0.0685	0.0827
3	IC2311.CFE, 09151.HK	中证 500, PP 科创 50-U	¥997,335.58	0.0702	0.0689
4	IM2311.CFE, 09031.HK	中证 1000, 海通 AESG-U	¥984,742.14	0.0685	0.0730
5	IM2311.CFE, 09812.HK	中证 1000, 三星中国龙网-U	¥951,633.98	0.0685	0.0968
6	IC2311.CFE, 09173.HK	中证 500, PP 中新经济-U	¥942,565.92	0.0702	0.0827
7	IC2311.CFE, 09031.HK	中证 500, 海通 AESG-U	¥912,887.15	0.0702	0.0730
8	IC2311.CFE, 09812.HK	中证 500, 三星中国龙网-U	¥881,839.49	0.0702	0.0968
9	IM2311.CFE, 09801.HK	中证 1000, 安硕中国-U	¥808,703.72	0.0685	0.0803
10	IM2311.CFE, 09839.HK	中证 1000, 华夏 A50-U	¥783,098.68	0.0685	0.0777

表 5　10 个候选组合的区间累计利润、最大回撤

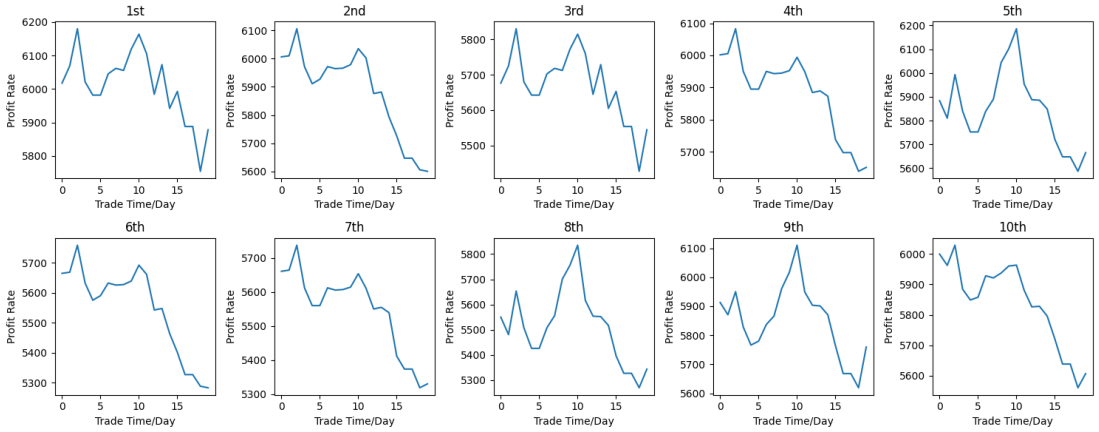

图 16　10 个候选组合的收益率曲线

二、组合系数确定

题目要求选出最适合跨市场套利的跨境 ETF 与股指期货组合，由于考虑到跨市场套利

的复杂性，以及实际交易过程中分散风险、增强整体组合、多元化投资等考虑，我们将上述 10 个候选组合按照一定的分配比例，得到一个新的组合，以符合现实的投资趋势。

对于候选的 10 个组合的比例确定，我们采取 Sharpe 比率与 Markowitz 均值-方差模型[11]的简化实现，对上述每个组合的区间总利润、区间利润率、股指期货和跨境 ETF 的最大回撤数据进行分析，最终加权得到 10 个组合的比例结果。与此同时，我们最终计算得出 10 个组合的组合风险与组合收益。

Sharpe 比率是一个衡量风险调整后回报的指标，可以定义为：

$$S = \frac{R_a - R_f}{\sigma_a} (44)$$

其中：S 代表夏普比率；R_a 为资产的年化预期回报率；R_f 表示无风险利率，用于衡量超额回报；σ_a 为资产的年化风险（标准差）。

Markowitz 投资组合优化模型的简化形式可以用来确定资产的权重，以最小化组合风险并考虑风险厌恶系数。其权重计算公式如下：

$$\mathbf{w} = \frac{\lambda \cdot \mathbf{C}^{-1} \cdot \mathbf{e}}{\mathbf{e}^T \cdot \lambda \cdot \mathbf{C}^{-1} \cdot \mathbf{e}} (45)$$

其中：$\mathbf{w}$ 是资产权重向量；λ 是投资者的风险厌恶系数；$\mathbf{C}$ 是资产收益率的协方差矩阵；$\mathbf{C}^{-1}$ 是协方差矩阵的逆矩阵；$\mathbf{e}$ 是一个由 1 组成的向量，其长度与资产数量相同；$\mathbf{e}^T$ 是向量 $\mathbf{e}$ 的转置。

组合风险通过组合内各资产的最大回撤来估计，可以表示为：

$$R_c = \sqrt{R_x^2 + R_y^2} (46)$$

其中：R_c 代表组合的总风险。R_x 和 R_y 分别是组合内单个资产的最大回撤。此公式假设资产间的风险是独立的，最大回撤作为风险的度量，反映了组合可能的最大潜在损失。

组合的预期收益可简单地表示为组合总利润：

$$R_p = \sum_{i=1}^{n} P_i (47)$$

其中：R_p 是组合的预期收益；P_i 是第 i 个资产的总利润。n 为组合中资产的数量。这里假设组合收益为各资产收益的线性叠加，暂不考虑资产间可能的协同效应。

上述过程中，我们输入包括组合的相对收益率时间序列、累计交易利润以及各自的最大回撤数据，以估算组合权重、风险和回报。权重根据相对收益率分配，确保整体投资组合的均衡；输出结果为投资权重、总体风险及预期回报，这为最大化利润的投资决策提供数据支持，并在风险可控的前提下进行资金分配。

三、上述 10 个组合的计算结果

交易编号	名称	权重比例	组合风险	组合收益
IM2311.CFE, 09151.HK	中证 1000, PP 科创 50-U	0.101	0.097	¥1,074,792.12
IM2311.CFE,	中证 1000, PP 中新经济-U	0.089	0.107	¥1,016,364.85

09173.HK				
IC2311.CFE, 09151.HK	中证 500, PP 科创 50-U	0.158	0.098	¥997,335.58
IM2311.CFE, 09031.HK	中证 1000，海通 AESG-U	0.099	0.100	¥984,742.14
IM2311.CFE, 09812.HK	中证 1000，三星中国龙网-U	0.073	0.119	¥951,633.98
IC2311.CFE, 09173.HK	中证 500, PP 中新经济-U	0.079	0.108	¥942,565.92
IC2311.CFE, 09031.HK	中证 500，海通 AESG-U	0.107	0.101	¥912,887.15
IC2311.CFE, 09812.HK	中证 500，三星中国龙网-U	0.093	0.120	¥881,839.49
IM2311.CFE, 09801.HK	中证 1000，安硕中国-U	0.107	0.106	¥808,703.72
IM2311.CFE, 09839.HK	中证 1000，华夏 A50-U	0.094	0.104	¥783,098.68

表 6 上述 10 个组合的计算结果

5.4 问题四

由于问题四要求实际测试所需 2023 年 11 月 9 日至 2023 年 11 月 17 日的实测数据，对问题二和问题三所建立的交易策略模型进行检验与结果分析，因此该部分另在第二阶段的实测报告中呈现。

六、模型的分析与检验

上述模型中，基于 Hurst 矩阵与折溢价率、跟踪误差的交易量因子影响算法是我们最具独创性的模型。我们针对该模型做灵敏度分析，以时间序列的 Hurst 矩阵均值增值（分形性指标体现）和跨境 ETF 时间序列的跟踪误差增值（波动性指标体现）作为影响指标，以交易量影响因子 f_1（置信度指标体现）作为分析目标，结果如下所示：

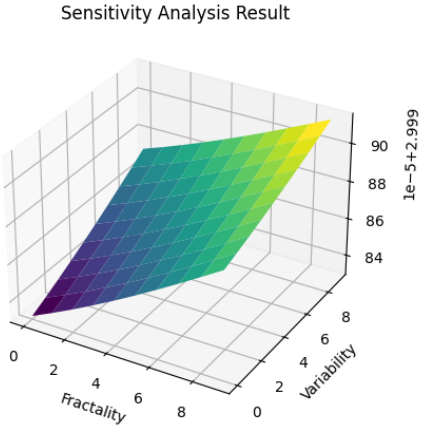

图 17 模型分析与检验结果

可以显示，当分形性指标不变、波动性指标增大时，或分形性指标增大、波动性指标不变时，模型的置信度指标升高，与模型建立的预期相符，因此可认为该模型鲁棒性高，充分体现跨境 ETF 市场分形性与波动性对交易置信度的高相关性，以及该算法的创新性。

七、 模型的评价、改进与推广

7.1 模型优点

上述所有模型中最大的优点在于将市场规律性与波动性这两个看似矛盾的特征，以量化的方式有机地结合起来，体现在对折溢价率、跟踪误差等波动性指标的深入分析与应用，以及利用广义 Hurst 函数计算所得的 Hurst 矩阵中蕴含的分形性数据来挖掘市场的规律性。这种独特的方法论提供了一种新颖的视角，以更加动态和细致的方式捕捉市场行为，增强了模型对交易置信度的判断。

7.2 模型缺点

1. 交易决策参数的确定较依赖于 BFGS 优化算法
2. 模型在处理高频与低频数据时存在一定程度的混用，这可能会导致预测精度受影响。
3. 市场波动指数预测结果的滞后性与非灵敏性可能会降低决策的时效性。
4. 模型在一级市场数据处理方面过于依赖于估计结果，缺乏对原生 ETF 基金 NAV 数据的支持，这在一定程度上限制了模型在实际应用中的可靠性和准确性。

7.3 模型改进

针对现有模型的局限性，改进集中于增强参数优化的动态性与数据处理的精细化。

1. 考虑运用动态规划算法替代静态 BFGS 算法，以实现交易决策参数的动态优化，这将有助于模型适应市场的快速变化。
2. 对于高频与低频数据的混合使用问题，可引入多尺度分析方法，对不同频率的数据进行分层处理，以提高预测的准确性。
3. 为了解决市场波动指数的滞后性问题，可以结合自适应滤波技术，提高模型对市场波动的即时响应能力。
4. 为了弥补一级市场数据的不足，建议与金融机构合作，获取更多原生数据以提升模型的实证基础。

7.4 模型推广

1. 在策略上引入基于条件的动态调整机制，例如为不同的交易环境和用户需求（如避免隔夜风险的用户）定制策略，提升模型的用户适应性。
2. 从技术层面上，通过模块化设计来增强模型的灵活性和扩展性，允许用户根据特定的市场特性来调整交易策略，如修改交易量函数以适应特定市场的交易规则和特征，从而推广模型在不同金融市场中的应用潜力。

八、参考文献

[1] Sethi, A., & Tripathi, V. (2019). "Excess Volatility and Costly Arbitrage in Exchange Traded Funds (ETFs): Evidence from India." DOI: 10.1177/0020294019857485.

[2] Boadu-Sebbe, G. "Effect of Exchange-Traded Funds Arbitrage Transactions on their

Underlying Holdings." DOI: 10.2139/ssrn.4370150.
[3] Fulkerson, J. A., Jordan, S., & Travis, D. H. (2017). "Bond ETF Arbitrage Strategies and Daily Cash Flow." DOI: 10.3905/jfi.2017.27.1.049.
[4] 宋家骥. 我国 ETF 套利策略研究[D].复旦大学,2016.
[5] 严洁. ETF 基金及其套利研究[D].西南财经大学,2007.
[6] 张浩然. 基于 LSTM 方法的国债利率期限结构预测与投资组合策略问题研究[D].西南财经大学,2022.DOI:10.27412/d.cnki.gxncu.2021.001888.
[7] 谭绮君. 基于多尺度分解和 LSTM 网络的金融时间序列预测研究[D].华南理工大学,2021.DOI:10.27151/d.cnki.ghnlu.2020.004284.
[8] 麻海兰. 基于 Hurst 指数的国内商品期货量化投资策略研究[D].华南理工大学,2021.DOI:10.27151/d.cnki.ghnlu.2020.002352.
[9] 石琦杭. 基于 Hurst 指数的配对交易策略设计[D].上海师范大学,2021.DOI:10.27312/d.cnki.gshsu.2021.001954.
[10] 顾悦. 基于股票市场多重分形结构的量化投资策略研究[D].西北大学,2023.DOI:10.27405/d.cnki.gxbdu.2022.002172.
[11] 涂申昊. 基于深度强化学习与改进均值—方差模型的投资组合研究[D].中国科学院大学(中国科学院人工智能学院),2019.

附　录

a.1 支撑材料代码与文件列表

```
extradata_extraction.py
load.py
name_extraction.py
numpy_extraction.py
post_process.py
process_2_1.py
process_2_2.py
process_1_1.py

process2_10_chart_bundled.npy
process2_10_origin_y_bundled.npy
process2_10_predict_y_bundled.npy
process2_10_rate_bundled.npy

process2_3_lstm_model.h5
process2_100_lstm_model.h5

q3_data_process.py
q3_post_data_process.py
q3_timeseries_pred.py

process3_1_origin_marketdelta_bundled.npy
process3_1_predicted_marketdelta_bundled.npy
process3_1_rel_profit_rate_bundled.npy
process3_1_unit_profit_bundled.npy
process3_2_rel_profit_rate_bundled.npy
```

同花顺 iFinD 软件上下载的 2023.8.1 至 2023.10.30 的一级市场与国际市场高频交易与每日单位的数据等

a.2 具体代码详述

process1_1.py

作用：核心交易策略代码 1

使用 Python 语言编写

```
'''
    第 1 题 核心模型
```

```
    2023/11/2
'''

import numpy as np
import statsmodels.api as sm
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from scipy.integrate import simps
from scipy.optimize import minimize_scalar
from scipy.optimize import minimize
import math
import json
import matplotlib.pyplot as plt

#-------------------------------------------------------
# 各种函数
#数据预处理
def process_and_save(input_file_path, output_file_path):
    # 读取输入的 numpy 文件
    input_data = np.load(input_file_path)

    # 提取每第二个维度中的 a, b, c, d (即矩阵的前 4 列)
    extracted_data = input_data[:, :4]

    # 计算 x = average(a, b, c, d)
    # axis=1 表示沿着第二个维度（列）计算平均值
    averages = np.mean(extracted_data, axis=1, keepdims=True)

    # 保存新的矩阵到文件
    np.save(output_file_path, averages)

def process_and_update(input_matrix_path, input_npy_path):
    # 读取给定的 numpy 矩阵
    input_matrix = np.load(input_matrix_path)

    # 确定有多少个分区
    num_sections = input_matrix.shape[0] // 241  # 假设矩阵可以被 241 整除

    # 从每个分区中提取第 2 列、第 1 行的数据
    x = [input_matrix[i * 241, 1] for i in range(num_sections)]

    # 转换为 numpy 数组并调整形状以匹配目标位置
    x_array = np.array(x).reshape(-1, 1)

```

```
    # 读取'input.npy'文件
    input_npy = np.load(input_npy_path)

    # 确保'input.npy'有足够的行来容纳新数据
    rows_required = 241 * num_sections
    if input_npy.shape[0] < rows_required:
        input_npy = np.pad(input_npy, ((0, rows_required - input_npy.shape[0]), (0, 0)), mode='constant')

    # 在指定的位置插入数据 x
    for i, value in enumerate(x_array):
        start_row = 1 + 241 * i
        end_row = 241 + 241 * i
        if input_npy.shape[1] < 2:
            # 如果'input.npy'只有 1 列，则添加新列
            new_column = np.zeros_like(input_npy[:, 0]).reshape(-1, 1)
            input_npy = np.hstack((input_npy, new_column))
        input_npy[start_row:end_row, 1] = value  # 插入数据

    # 保存更新后的'input.npy'文件
    np.save(input_npy_path, input_npy)

def insert_integer_into_string(original_string, integer, position):
    """
    将一个整数插入到字符串的指定位置。

    :param original_string: 原始字符串
    :param integer: 需要插入的整数
    :param position: 整数应插入的位置（基于 0 的索引）
    :return: 修改后的字符串
    """
    # 将整数转换为字符串
    integer_str = str(integer)

    # 插入整数字符串到指定位置
    new_string = original_string[:position] + integer_str + original_string[position:]

    return new_string

def top_10_entries(input_dict, output_file):
    """
    找出字典中数值最大的 10 个键值对并将其写入到文本文件中。

```

```
    :param input_dict: 输入的字典，其值为数值。
    :param output_file: 输出文件的名称。
    """
    # 对字典的键值对按值进行降序排序，并取前 10 个
    top_10 = sorted(input_dict.items(), key=lambda x: x[1],
reverse=True)[:10]

    # 将这些键值对写入到指定的文本文件
    with open(output_file, 'w') as file:
        for key, value in top_10:
            file.write(f"{key}: {value}\n")

def modify_if_constant(x):
    """
    检查时间序列是否为常数值序列。
    如果是，就在序列中的每个值上添加一些微小的随机变动。

    :param x: 时间序列，一个数值列表或 NumPy 数组。
    :return: 可能被修改的时间序列。
    """
    if np.all(x == x[0]):  # 检查序列中的所有值是否相等
        # 在每个值上加上小的随机扰动
        noise = np.random.normal(0, 1e-8, size=len(x))
        return x + noise
    else:
        return x

#E-G 协整
def check_cointegration(x, y):
    # Engle-Granger 协整检验
    eg_test = sm.tsa.coint(x, y)
    p_value = eg_test[1]
    if p_value < 0.05:
        print("协整关系被确认，p-value: ", p_value)
        return True
    else:
        print("没有协整关系，p-value: ", p_value)
        return False

def build_ecm(x, y):
    # 构建误差修正模型
    model = sm.tsa.VAR(np.column_stack((x, y)))
    results = model.fit()
    return results
```

```
131.
132. #时间序列预测
133. def get_next_y_value(model, current_y):
134.     # 从模型中获得预测
135.     forecast = model.forecast(np.column_stack((x, y)), steps=1)
136.     next_y = forecast[0, 1]
137.     return next_y
138.
139. #核心交易策略 g
140. def g(params, x, y, profit = 0, origin_eff_num = 1000000):
141.     #假设从 2023.8.1 至 2023.10.30 日每个交易日交易时间（9:30~11:30、
    13:00~15:00)的每一分钟时间进行 ETF 债券交易
142.     i = 0  # 循环变量
143.     j = 241
144.     [d1, d2, d3, d4] = params
145.     total_profit_rate = []  # 用于画图的
146.     total_profit_chart = []  # 用于画图的
147.     max_y = y[5]
148.     min_y = y[5]
149.     y_tsub1 = y[5]
150.
151.     for elem in y[6:1206]: #y[0:size]:
152.         current_y = elem   #该 ETF 每分钟的二级市场价
153.         # 获得时间序列 y 中 t+1 时刻的值
154.         next_y = get_next_y_value(ecm_model, current_y)     #下一分钟的
    二级市场价预测
155.         delta_y = next_y - current_y    #delta_y
156.         if y_tsub1 == 0:
157.             y_tsub1 = y_tsub1 + 1e-9
158.         profit_rate = elem - y_tsub1 / y_tsub1  # 利润率
159.         max_y = max(max_y, elem)
160.         min_y = min(min_y, elem)
161.         y_tsub1 = elem  #现在才更新 elem
162.
163.         #if origin_eff_num > 0: #必须得满足任何时候有 ETF 持仓
164.         if delta_y >= abs(trade_threshold):     #折价交易
165.             trade_1st_market = d1 * math.log(abs(delta_y))  #买入
166.             trade_2nd_market = d2 * math.log(abs(delta_y))  #卖出
167.             if i % 240 == 239:       #最后一刻补仓/平仓!
168.                 new1 = -(trade_1st_market / (trade_1st_market +
    trade_2nd_market)) * origin_eff_num
169.                 new2 = -(trade_2nd_market / (trade_1st_market +
    trade_2nd_market)) * origin_eff_num
170.                 trade_1st_market = new1
```

```
171.                 trade_2nd_market = new2
172.             if trade_2nd_market <= 0.7 * numpy_1[i, 5]: #限制条件（假设
    单次交易数量占每分钟总交易数量的比例），进行交易
173.                 origin_eff_num = origin_eff_num + trade_1st_market -
    trade_2nd_market   #持仓数量更新
174.                 temp_money = -(trade_1st_market * x[i]) +
    (trade_2nd_market * y[j])    #直接的买入卖出
175.                 temp_money = temp_money - (eff1 * trade_1st_market) -
    (trade_2nd_market * y[j] * eff3 * 0.01) - (
176.                     (trade_1st_market * x[i] + trade_2nd_market *
    y[j]) * (eff2 + eff4) * 0.01)
177.                     #考虑一级市场结算金额、二级市场交易佣金、印花税和市场冲
    击成本
178.             else:
179.                 temp_money = 0
180.
181.         elif delta_y <= -1 * abs(trade_threshold):      #溢价交易
182.             trade_1st_market = d3 * math.log(abs(delta_y))  #卖出
183.             trade_2nd_market = d4 * math.log(abs(delta_y))  #买入
184.             if i % 240 == 239:       #最后一刻补仓/平仓!
185.                 new1 = (trade_1st_market / (trade_1st_market +
    trade_2nd_market)) * origin_eff_num
186.                 new2 = (trade_2nd_market / (trade_1st_market +
    trade_2nd_market)) * origin_eff_num
187.                 trade_1st_market = new1
188.                 trade_2nd_market = new2
189.             if trade_2nd_market <= 0.7 * numpy_1[i, 5]: #限制条件（假设
    单次交易数量占每分钟总交易数量的比例），进行交易
190.                 origin_eff_num = origin_eff_num - trade_1st_market +
    trade_2nd_market   #持仓数量更新
191.                 temp_money = +(trade_1st_market * x[i]) -
    (trade_2nd_market * y[j])   #直接的买入卖出
192.                 temp_money = temp_money - (eff1 * trade_1st_market) -
    (trade_2nd_market * y[j] * eff3 * 0.01) - (
193.                             (trade_1st_market * x[i] + trade_2nd_market
    * y[j]) * (eff2 + eff4) * 0.01)
194.                 # 考虑一级市场结算金额、二级市场交易佣金、印花税和市场冲击
    成本
195.             else:
196.                 temp_money = 0
197.
198.         else:       #不进行交易
199.             temp_money = 0
200.         #else:
```

```
            temp_money = 0

        profit = profit + temp_money
        total_profit_chart.append(temp_money)     #累计均利润

        total_profit_rate.append(profit_rate)  # 利润率
        if max_y == 0:
            max_y = max_y + 1e-9
        max_drawdown = float((max_y - min_y) / max_y)
        print("max_y", max_y)
        print("min_y", min_y)
        i = i + 1   #更新一次循环变量
        if (i >= size - 241):   #T+1 市场
            j = i
        else:
            j = i + 241
    return profit, total_profit_chart, total_profit_rate,
max_drawdown
#params: [d1, d2, d3, d4]
#x、y: 原始的数据矩阵

# 定义目标函数以最大化 profit
def objective(params):
    #x, y, profit, origin_eff_num = 10, 20, 30, 40  # 这些值应该根据你
的具体情况设定
    new_profit = g(params, x, y, profit, origin_eff_num)
    return -new_profit  # 由于minimize 函数是为了最小化目标函数，所以我们
返回-new_profit 以实现最大化
#-----------------------------------------------------------
# Part 1 初始化及文件输入
# 指定输入和输出的文件路径

#debug only
#numpy_1_origin = 'original_dataset/62.npy'  #原始输入数据，6 列 n 行
#numpy_2_origin = 'dataset1/62.npy'          #输出数据，2 列 n 行

process1_result_dict = {}
total_chart_bundled = []
total_rate_bundled = []
max_drawdown_dict = {}
#with open('process1_result.json', 'w') as json_file: #打开 JSON 文件
#for integer in range(1, 886 + 1):
for integer in [500, 515, 449, 508, 10, 118, 18, 447, 537, 121]:
```

```
    numpy_1_origin = insert_integer_into_string('original_dataset/.npy', integer, 17)
    print("Modified String:", numpy_1_origin)
    numpy_2_origin = insert_integer_into_string('dataset1/.npy', integer, 9)
    print("Modified String:", numpy_2_origin)

    # 读取数据，用于Part 2 协整关系验证
    numpy_1 = np.load(numpy_1_origin)
    numpy_2 = np.load(numpy_2_origin)

    x, y = numpy_2[:, 1], numpy_2[:, 0]       #这个关系不要混乱了！
    #x 是一级市场价，y 是二级市场价

    x = modify_if_constant(x)   #处理常数值序列问题
    y = modify_if_constant(y)

    #描述单个 ETF 证券的一些交易数据。
    #numpy_1.npy 是[[a, b, c, d, e, f], ... ,[..., ..., ..., ..., ..., ..., ]]，即矩阵有 6 列，若干行(假设为 dim_2 行)；
    #其中，a 代表每分钟的收盘价，b 代表每分钟开盘价，c 代表最高价，d 代表最低价，e 代表每分钟的总成交额，f 代表每分钟该 ETF 证券的成交量。每一行代表每个交易时刻（以分钟为单位）的数据。

    #numpy_2.npy 是一个 2 列、若干行(同样是 dim_2 行)的矩阵，每行的数据为[g, h]，
    #其中 g 代表该 ETF 每分钟的二级市场价，h 代表该 ETF 每分钟的一级市场价。同样地，每一行代表每个交易时刻（以分钟为单位）的数据。


    # Part 2 协整关系构建
    # 检查协整关系
    cointegrated = check_cointegration(x, y)
    # 构建误差修正模型
    ecm_model = build_ecm(x, y)

    # Part 3 基于每个股票的参数处理及预测：全部放到后面去

    # 折溢价率
    #c1 = calculate_premium_discount_rate(numpy_1, numpy_2)
    #print(f'平均折溢价率: {c1:.4f}%')

    # 跟踪误差
```

```
    #c2 = calculate_tracking_error(numpy_1, numpy_2)
    #print(f'跟踪误差: {c2:.4f}%')

    # 宏观经济指标
    #c3 = calculate_macro_economic_indicator(numpy_1, numpy_2)
    #print(f'宏观经济指标: {c3:.2f}')


    # Part 4 交易策略
    # 这几个变量都是需要被预测的，待定系数
    [d1, d2, d3, d4] = [15.993214229999998 * 1, -6.653766326666666 * 1, 5.151237160000001 * 1, 62.70280344 * 0.75]
    params = [d1, d2, d3, d4]
    trade_threshold = 1e-5  #交易阈值信号限制：0.00001 元

    profit = 0      #我们自己的利润

    eff1 = 1    #ETF 申购、赎回费用、过户费、证管费用、经手费、证券结算金：固定成本（元）（一级市场）
    eff2 = 0.1  #印花税率（百分比%）（所有市场）
    eff3 = 0.2  #ETF 二级市场交易佣金、股票交易佣金百分比%）（二级市场）
    eff4 = 0.05 #市场冲击成本（百分比%）（所有市场）
    trade_regulation = 0.7  #假设单次交易数量占每分钟总交易数量的比例
    origin_eff_num = 1000000    #限购数量 100 万 ETF

    size = min(len(numpy_1), len(numpy_2))

    # 进行参数估计----------------------------------------
        # 初始猜测的 d1, d2, d3, d4 参数值
        #initial_guess = [1, 1, 1, 1]

        # 使用 minimize 函数来寻找最优参数
        #result = minimize(objective, initial_guess, method='BFGS')  # BFGS 是一个常用的优化算法

        # 输出最优参数
        #optimal_params = result.x
        #print(f'Optimal parameters: {optimal_params}')

        # 现在你可以使用这些最优参数来计算最大 profit
        #max_profit = -objective(optimal_params)
        #print(f'Maximum profit: {max_profit}')
    #-------------------------------------------------
```

```
318.     # 参数输出
319.
320.     new_profit, total_profit_chart, total_profit_rate,
    max_drawdown_result = g(params, x, y, profit, origin_eff_num)
321.     total_profit_chart = np.array(total_profit_chart)
322.     total_chart_bundled.append(total_profit_chart)
323.     total_profit_rate = np.array(total_profit_rate)
324.     total_rate_bundled.append(total_profit_rate)
325.
326.     process1_result_dict[integer] = new_profit
327.     max_drawdown_dict[integer] = max_drawdown_result
328.     with open('process2_1comp.json', 'w') as json_file:
329.         json.dump(process1_result_dict, json_file, indent=4)
330.     with open('process1_maxdrawdown_deleted.json', 'w') as json_file:
331.         json.dump(max_drawdown_dict, json_file, indent=4)
332.     #关闭文件，退出循环
333.
334. #top_10_entries(process1_result_dict, "process1_top_10.txt")
335.
336. # 画图-----------------------------------------------
337. # 设置图表的标题、x 轴和 y 轴的标签
338. '''
339. titles = ['511620.SH', '511650.SH', '511920.SH', '511600.SH',
    '511800.SH', '511830.SH', '511850.SH', '511900.SH', '511810.SH',
340.           '511820.SH']
341. x_labels = ['Trade Time/min'] * 10
342. y_labels = ['Unit Profit/(CNY/min$^{-1}$)'] * 10
343.
344. # 创建图形
345. fig, axs = plt.subplots(2, 5, figsize=(15, 6))
346.
347. # 遍历每个子图并绘制数据
348. for i, ax in enumerate(axs.flat):
349.     #ax.bar(range(len(total_chart_bundled[i])),
    total_chart_bundled[i])
350.     ax.plot(total_chart_bundled[i])
351.     ax.set_title(titles[i])
352.     ax.set_xlabel(x_labels[i])
353.     ax.set_ylabel(y_labels[i])
354.
355. # 调整子图间的空间以保证标签不重叠
356. plt.tight_layout()
357. plt.show()
358.
```

```
#----------------------------------------

titles = ['511620.SH', '511650.SH', '511920.SH', '511600.SH',
    '511800.SH', '511830.SH', '511850.SH', '511900.SH', '511810.SH',
          '511820.SH']
x_labels = ['Trade Time/min'] * 10
y_labels = ['Profit Rate'] * 10
# 显示图形
plt.show()

# 创建图形
fig, axs = plt.subplots(2, 5, figsize=(15, 6))

# 遍历每个子图并绘制数据
for i, ax in enumerate(axs.flat):
    #ax.bar(range(len(total_chart_bundled[i])),
   total_chart_bundled[i])
    ax.plot(total_rate_bundled[i])
    ax.set_title(titles[i])
    ax.set_xlabel(x_labels[i])
    ax.set_ylabel(y_labels[i])

# 调整子图间的空间以保证标签不重叠
plt.tight_layout()

# 显示图形
plt.show()
'''
```

process_2_2.py

作用：核心交易策略代码 2

使用 Python 语言编写

```
'''
    第 2 题 核心模型（改进后）
    2023/11/2
'''
import numpy as np
import numpy.polynomial.polynomial as poly  #多重分形离散分析（MF-DFA）
方法
from scipy.integrate import simps
from scipy.optimize import minimize_scalar
from keras.models import Sequential, load_model
from keras.layers import LSTM, Dense
import matplotlib.pyplot as plt
```

```
import tensorflow as tf
import json
import os
import math

#函数定义部分-----------------------------------------------------
#数据预处理
def insert_integer_into_string(original_string, integer, position):
    """
    将一个整数插入到字符串的指定位置。

    :param original_string: 原始字符串
    :param integer: 需要插入的整数
    :param position: 整数应插入的位置（基于 0 的索引）
    :return: 修改后的字符串
    """
    # 将整数转换为字符串
    integer_str = str(integer)

    # 插入整数字符串到指定位置
    new_string = original_string[:position] + integer_str +
original_string[position:]

    return new_string
def modify_if_constant(x):
    """
    检查时间序列是否为常数值序列。（改动过了）
    如果是，就在序列中的每个值上添加一些微小的随机变动。

    :param x: 时间序列，一个数值列表或 NumPy 数组。
    :return: 可能被修改的时间序列。
    """
    '''
    if np.all(x == x[0]):  # 检查序列中的所有值是否相等
        # 在每个值上加上小的随机扰动
        noise = np.random.normal(0, 1e-9, size=len(x))
        return x + noise
    else:
        return x
    '''
    noise = np.random.normal(0, 1e-9, size=len(x))
    return x + noise
def list_sum_perdata(list1, list2):
    #把两个数组按元素增加
```

```
    result = [x + y for x,y in zip(list1, list2)]
    return result
def normalize_array(arr):
    #把一个数组的数据标准化处理
    min_val = np.min(arr)
    max_val = np.max(arr)
    range_val = max_val - min_val
    if range_val != 0:
        normalized_arr = (arr - min_val) / range_val
        return normalized_arr
    else:
        return arr  # 如果数组中所有值都相同，则返回原数组
def top_10_entries(input_dict, output_file):
    """
    找出字典中数值最大的 10 个键值对并将其写入到文本文件中。

    :param input_dict: 输入的字典，其值为数值。
    :param output_file: 输出文件的名称。
    """
    # 对字典的键值对按值进行降序排序，并取前 10 个
    top_10 = sorted(input_dict.items(), key=lambda x: x[1],
reverse=True)[:10]

    # 将这些键值对写入到指定的文本文件
    with open(output_file, 'w') as file:
        for key, value in top_10:
            file.write(f"{key}: {value}\n")
def extract_numbers_from_json(file_path):
    # 用于存储所有数值的列表
    numbers_list = []

    # 递归函数来遍历 JSON 数据结构
    def extract_numbers(data):
        if isinstance(data, dict):  # 如果是字典，递归它的值
            for value in data.values():
                extract_numbers(value)
        elif isinstance(data, list):  # 如果是列表，递归每个元素
            for item in data:
                extract_numbers(item)
        elif isinstance(int(data), (int, float)):  # 如果是数值，添加到列
表
            numbers_list.append(data)

    # 读取 JSON 文件并将内容加载到 Python 数据结构中
```

```
    with open(file_path, 'r') as file:
        data = json.load(file)
        extract_numbers(data)

    return numbers_list
#折溢价率
def calculate_premium_discount_rate(numpy_1):
    # 提取二级市场价格和一级市场价格
    market_prices = numpy_1[:, 0]
    nav_prices = numpy_1[:, 1]

    # 替换零值以避免除以零的错误
    epsilon = 10  # 一个正数，由经验确定
    nav_prices_replaced = np.where(nav_prices == 0, epsilon,
nav_prices)

    # 计算每分钟的折溢价率
    premium_discount_rates = ((market_prices - nav_prices_replaced) /
nav_prices_replaced) * 100

    # 计算折溢价率的平均值
    average_premium_discount_rate = np.mean(premium_discount_rates)

    return average_premium_discount_rate
#跟踪误差
def calculate_tracking_error(numpy_1):
    # 提取二级市场价格和一级市场价格
    market_prices = numpy_1[:, 0]
    nav_prices = numpy_1[:, 1]

    # 确保没有零值，以避免除以零的错误
    epsilon = 2.71828  # 一个正数，由经验确定
    nav_prices_replaced = np.where(nav_prices == 0, epsilon,
nav_prices)

    # 计算每分钟的价格差异比率
    price_diff_ratios = (market_prices - nav_prices_replaced) /
nav_prices_replaced

    # 计算跟踪误差
    tracking_error = np.std(price_diff_ratios)
    return tracking_error
#宏观经济指标，内部有嵌套
def calculate_macro_economic_indicator(numpy_1, numpy_2):
```

```
    # 从 numpy_1 矩阵中提取数据
    closing_prices = numpy_1[:, 0]
    total_trade_value = numpy_1[:, 4]
    trade_volume = numpy_1[:, 5]

    # 计算价格动量（二阶偏导数）
    price_momentum = np.gradient(np.gradient(closing_prices))

    # 计算成交量动量（二阶偏导数）
    volume_momentum = np.gradient(np.gradient(trade_volume))

    # 计算市场流动性（每分钟的总成交额与收盘价的比值）
    market_liquidity = total_trade_value / (closing_prices + 1e-10)
# 避免除以零

    # 定义一个滑动窗口函数来计算局部平均值
    def sliding_window(arr, window_size):
        return np.convolve(arr, np.ones(window_size)/window_size,
mode='valid')

    # 使用滑动窗口计算局部平均价格动量和成交量动量
    local_avg_price_momentum = sliding_window(price_momentum, 10)
    local_avg_volume_momentum = sliding_window(volume_momentum, 10)

    # 定义一个优化目标函数，该函数试图最大化价格动量和成交量动量之间的相关
性
    def objective_func(window_size):
        local_avg_price_momentum = sliding_window(price_momentum,
int(window_size))
        local_avg_volume_momentum = sliding_window(volume_momentum,
int(window_size))
        correlation = np.corrcoef(local_avg_price_momentum,
local_avg_volume_momentum)[0, 1]
        return -correlation  # 最大化相关性相当于最小化负相关性

    # 执行优化以找到最佳滑动窗口大小
    optimal_window_size = minimize_scalar(objective_func, bounds=(5,
50), method='bounded').x

    # 使用最佳滑动窗口大小重新计算局部平均价格动量和成交量动量
    local_avg_price_momentum = sliding_window(price_momentum,
int(optimal_window_size))
    local_avg_volume_momentum = sliding_window(volume_momentum,
int(optimal_window_size))
```

```
172.
173.     # 将三个指标组合成一个复合指标
174.     # 我们假设每个指标的权重为1/3，可以根据实际情况调整权重
175.     length1 = len(local_avg_price_momentum)
176.     length2 = len(local_avg_volume_momentum)
177.     length3 = len(market_liquidity)
178.     min_length = min(length1, length2, length3)
179.
180.     composite_indicator = (local_avg_price_momentum[0:min_length] +
    local_avg_volume_momentum[0:min_length] +
    market_liquidity[0:min_length]) / 3
181.
182.     # 计算复合指标的积分作为宏观经济参数化指标
183.     macro_economic_indicator = simps(composite_indicator) / (10 ** 7)
184.
185.     return macro_economic_indicator
186. # Hurst 矩阵处理函数————————————————
187. #多重分形离散分析（MF-DFA）方法
188. def MF_DFA(X, q_values=None):
189.     if q_values is None:
190.         q_values = [1, 2, 3, 4, 5, 6, 7]
191.
192.     # Ensure X is a numpy array of type float64
193.     X = np.asarray(X, dtype=np.float64)
194.
195.     N = len(X)
196.     # 构建累积偏差序列
197.     Y = np.cumsum(X - np.mean(X))
198.
199.     # 分段和局部趋势拟合
200.     # 选择合适的段长度s，这里选择为7
201.     s = N // 7
202.     Fq = np.zeros(len(q_values))
203.
204.     for i, q in enumerate(q_values):
205.         F2 = np.zeros(s)
206.         for v in range(0, N, s):
207.             if v + s < N:
208.                 # 对每个段使用多项式拟合
209.                 segment = Y[v:v + s]
210.                 t = np.arange(len(segment))
211.                 p = poly.Polynomial.fit(t, segment, 3)  # 使用三次多项式
    拟合
212.                 fit = p(t)
```

```
                # 计算离散函数
                F2[v // s] = np.mean((segment - fit) ** 2)
        # 求解标度函数
        Fq[i] = np.mean(F2 ** (q / 2)) ** (1 / q) if q != 0 else
    np.exp(0.5 * np.mean(np.log(F2)))

    # 计算h(q)
    hq = np.log(Fq) / np.log(s)
    return hq
#从2个数到1*7向量的变换
def transform_to_vector(c1, c2):
    return np.array([c1, c2, c1*c2, c1**2, c2**2, np.sqrt(abs(c1)),
    np.sqrt(abs(c2))])
#线性缩放方法
def scale_to_range(vector, min_val, max_val):
    min_vector = np.min(vector)
    max_vector = np.max(vector)
    scaled_vector = (max_val - min_val) * (vector - min_vector) /
    (max_vector - min_vector) + min_val
    return scaled_vector
#聚合和线性缩放的方法
def aggregate_and_scale(vector, num_elements, seed=0):
    # 设置随机种子以保证打散操作的一致性
    np.random.seed(seed)

    # 首先随机打散向量
    np.random.shuffle(vector)

    # 将打散后的向量分组并聚合
    aggregated = [np.mean(vector[i:i + len(vector) // num_elements])
    for i in
                  range(0, len(vector), len(vector) // num_elements)]

    # 确保聚合后的列表有num_elements个元素
    if len(aggregated) > num_elements:
        aggregated = aggregated[:num_elements]

    # 线性缩放到1到2之间
    min_val, max_val = 1, 2
    min_aggregated = np.min(aggregated)
    max_aggregated = np.max(aggregated)

    # 如果所有聚合值相等（防止除以0）
    if max_aggregated == min_aggregated:
```

```
253.         scaled = [min_val + (max_val - min_val) / 2] * num_elements  # 所有值设为中点
254.     else:
255.         scaled = [(max_val - min_val) * (val - min_aggregated) / (max_aggregated - min_aggregated) + min_val for val in
256.                   aggregated]
257.
258.     return scaled
259. # 时间序列处理函数————————————————
260. #基于时间偏移的预测
261. def exponential_decay_weight(n, decay_rate):
262.     """
263.     计算指数衰减权重。
264.
265.     :param n: 时间偏移量。
266.     :param decay_rate: 衰减率。
267.     :return: 权重值。
268.     """
269.     return np.exp(-decay_rate * n)
270. def corrected_price_prediction(predictions, decay_rate=0.2):
271.     """
272.     根据预测值和时间偏移量计算修正后的价格预测值。
273.     :param predictions: 包含 t+1, t+2, t+3 时刻的价格预测值的数组。
274.     :param decay_rate: 衰减率，用于计算权重。
275.     :return: 修正后的 t+1 时刻的价格预测值。
276.     """
277.     # 确保 predictions 是一个一维 numpy 数组
278.     predictions = np.squeeze(np.array(predictions))
279.     # 计算权重
280.     weights = np.array([exponential_decay_weight(n, decay_rate) for n in range(1, len(predictions) + 1)])
281.     # 标准化权重
282.     normalized_weights = weights / np.sum(weights)
283.     # 计算加权平均值
284.     corrected_prediction = np.dot(predictions, normalized_weights)
285.     corrected_prediction = corrected_prediction + 0.0024
286.     return corrected_prediction
287.
288. #核心交易策略 gT0
289. def gT0(params, x, y, Y_aggregated, profit = 0, origin_eff_num = 1000000):
290.     #假设从 2023.8.1 至 2023.10.30 日每个交易日交易时间（9:30~11:30、13:00~15:00)的每一分钟时间进行 ETF 债券交易
291.     i = 0  # 循环变量
```

```
    j = 0
    [d1, d2, d3, d4] = params
    total_profit_rate = []   # 用于画图的
    total_profit_chart = []  # 用于画图的
    origin_y = []
    predict_y = []
    n_input_steps = 7
    n_output_steps = 3
    max_y = y[n_input_steps - 1]
    min_y = y[n_input_steps - 1]
    e1, e2, e3, e4 = Y_aggregated   #参数处理
    #time_steps = 5 # 时间序列的时间步长
    extracted_time_series_data = []

    for i in range(1200):#len(y) - n_input_steps - n_output_steps +
1):
        if i % 2000 == 0:
            print(i)
        # 将提取的数据添加到列表中
        extracted_time_series_data.append(y[i:i + n_input_steps])
    extracted_time_series_data = np.array(extracted_time_series_data)
    extracted_new_y_data = model.predict(extracted_time_series_data,
verbose=1, batch_size=1000, workers=-1)  # 批量预测

    for t in range(1200):   #len(extracted_time_series_data)):     #模
拟所有周期的数据
        # 预测时间序列中 t+1, t+2, t+3 时刻的值
        #if t % 500 == 0:
            #print(extracted_time_series_data[t])
        #new_y_data = model.predict(extracted_time_series_data[t],
batch_size=1000, verbose=0, workers=-1)
        #print(new_y_data)
        delta_y = corrected_price_prediction(extracted_new_y_data[t])
- y[t]   #修正为一个值
        #print(delta_y)
        absolute_delta_y = y[t+1] - y[t]    #delta_y

        profit_rate = (absolute_delta_y / y[t])  #利润率
        max_y = max(max_y, y[t])
        min_y = min(min_y, y[t])

        #if origin_eff_num > 0: #必须得满足任何时候 ETF 持仓不为负
        if delta_y >= abs(trade_threshold):     #折价交易
            trade_1st_market = e1 * d1 * math.log(abs(delta_y))  #买入
```

```
            trade_2nd_market = e2 * d2 * math.log(abs(delta_y))  #卖出
            if t % 240 == 239:       #最后一刻补仓/平仓!
                new1 = -(trade_1st_market / (trade_1st_market +
trade_2nd_market)) * origin_eff_num
                new2 = -(trade_2nd_market / (trade_1st_market +
trade_2nd_market)) * origin_eff_num
                trade_1st_market = new1
                trade_2nd_market = new2
            if trade_2nd_market <= 0.7 * numpy_1[t, 5]: #限制条件（假设
单次交易数量占每分钟总交易数量的比例），进行交易
                origin_eff_num = origin_eff_num + trade_1st_market -
trade_2nd_market   #持仓数量更新
                temp_money = -(trade_1st_market * x[t]) +
(trade_2nd_market * y[j])    #直接的买入卖出
                temp_money = temp_money - (eff1 * trade_1st_market) -
(trade_2nd_market * y[j] * eff3 * 0.01) - (
                        (trade_1st_market * x[t] + trade_2nd_market *
y[j]) * (eff2 + eff4) * 0.01)
                    #考虑一级市场结算金额、二级市场交易佣金、印花税和市场冲
击成本
            else:
                temp_money = 0

        elif delta_y <= -1 * abs(trade_threshold):      #溢价交易
            trade_1st_market = e3 * d3 * math.log(abs(delta_y))  #卖出
            trade_2nd_market = e4 * d4 * math.log(abs(delta_y))  #买入
            if t % 240 == 239:       #最后一刻补仓/平仓!
                new1 = -(trade_1st_market / (trade_1st_market +
trade_2nd_market)) * origin_eff_num
                new2 = -(trade_2nd_market / (trade_1st_market +
trade_2nd_market)) * origin_eff_num
                trade_1st_market = new1
                trade_2nd_market = new2
            if trade_2nd_market <= 0.7 * numpy_1[t, 5]: #限制条件（假设
单次交易数量占每分钟总交易数量的比例），进行交易
                origin_eff_num = origin_eff_num - trade_1st_market +
trade_2nd_market   #持仓数量更新
                temp_money = +(trade_1st_market * x[t]) -
(trade_2nd_market * y[j])   #直接的买入卖出
                temp_money = temp_money - (eff1 * trade_1st_market) -
(trade_2nd_market * y[j] * eff3 * 0.01) - (
                            (trade_1st_market * x[t] + trade_2nd_market
* y[j]) * (eff2 + eff4) * 0.01)
```

```
359.                 # 考虑一级市场结算金额、二级市场交易佣金、印花税和市场冲击
    成本
360.             else:
361.                 temp_money = 0
362.
363.         else:       #不进行交易
364.             temp_money = 0
365.         #else:
366.             temp_money = 0
367.
368.
369.         profit = profit + temp_money
370.         total_profit_chart.append(temp_money)     #累计利润
371.         total_profit_rate.append(profit_rate)   #利润率
372.         origin_y.append(y[t])
373.         predict_y.append(delta_y + y[t])    #新 y 值(从 7 至 1207)
374.         max_drawdown = float((max_y - min_y) / max_y)
375.         #i = i + 1   #更新一次循环变量
376.         '''if (i >= size - 241):   #T+1 市场
377.             j = i
378.         else:
379.             j = i + 241'''
380.         j = t   #T+0 市场
381.         if j % 50 == 0:
382.             print(j, "Profit=" ,profit)
383.     return profit, total_profit_chart, total_profit_rate,
    max_drawdown, predict_y, origin_y
384. #params: [d1, d2, d3, d4]
385. #x、y: 原始的数据矩阵
386. #实际执行部分----------------------------------------------------------
387. #交易参数
388. [d1, d2, d3, d4] = [15.993214229999998, -6.653766326666666,
    5.151237160000001, 62.70280344 * 0.75]
389. params = [d1, d2, d3, d4]
390. trade_threshold = 0  # 交易阈值信号限制：0.00001 元
391. profit = 0  # 我们自己的利润
392. eff1 = 1  # ETF 申购、赎回费用、过户费、证管费用、经手费、证券结算金:固定成
    本（元）（一级市场）
393. eff2 = 0.1  # 印花税率（百分比%）（所有市场）
394. eff3 = 0.2  # ETF 二级市场交易佣金、股票交易佣金百分比%）（二级市场）
395. eff4 = 0.05  # 市场冲击成本（百分比%）（所有市场）
396. trade_regulation = 0.7  # 假设单次交易数量占每分钟总交易数量的比例
397. origin_eff_num = 1000000  # 假设初始持仓数量 100 万 ETF
398.
```

```
#以下是训练用————————————————————————————————————
"""
#构造 LSTM 模型---------------
## LSTM 网络的参数
np.random.seed(1)
tf.random.set_seed(1)
'''
# 构建 LSTM 与全连接层结合的模型
model = Sequential()
# LSTM 层
model.add(LSTM(50, activation='relu', input_shape=(7,1),
return_sequences=False))
# 全连接层
model.add(Dense(20, activation='relu'))  # 添加一个有 20 个单元的全连接
层
model.add(Dense(3))  # 输出层，预测 t+1, t+2, t+3 时刻的价格
# 编译模型
model.compile(optimizer='adam', loss='mse')
'''
model = load_model('process2_100_lstm_model.h5')
model.summary()

#准备开始
ETF100index_json_file_path = 'process2_3_100ETFindexes.json'
integers = extract_numbers_from_json(ETF100index_json_file_path)
print(integers)
for integer in integers:
    # 从文件中读取数据 --------------
    numpy_1_origin = insert_integer_into_string('dataset1/.npy',
integer, 9)  # 按照索引读取 numpy 数据
    print("Modified String:", numpy_1_origin)
    numpy_2_origin = insert_integer_into_string('dataset2/.npy',
integer, 9)
    print("Modified String:", numpy_2_origin)
    numpy_original_origin =
insert_integer_into_string('original_dataset/.npy', integer, 17)

    if not os.path.exists(numpy_2_origin):
        continue
    numpy_1 = np.load(numpy_1_origin, allow_pickle=True)
    numpy_2 = np.load(numpy_2_origin, allow_pickle=True)
    numpy_original = np.load(numpy_original_origin,
allow_pickle=True)

```

```
    market2_1 = numpy_original[:, 0]  # 收盘价
    market2_2 = numpy_original[:, 1]  # 开盘价
    market2_3 = numpy_original[:, 2]  # 最高价
    market2_4 = numpy_original[:, 3]  # 最低价
    market2_1 = modify_if_constant(market2_1)
    market2_2 = modify_if_constant(market2_2)
    market2_3 = modify_if_constant(market2_3)
    market2_4 = modify_if_constant(market2_4)

    market2_5 = numpy_original[:, 4]  # 每分钟成交额
    market2_6 = numpy_original[:, 5]  # 每分钟成交量

    market1_1 = numpy_2[:, 0]  # 复权单位净值
    market1_2 = numpy_2[:, 1]  # 贴水
    market1_3 = numpy_2[:, 2]  # 贴水率
    market1_4 = numpy_2[:, 3]  # 增长率

    # 计算 MF-DFA --------------
    hurst_matrix = []
    sum_matrix = []
    sum_matrix = list_sum_perdata(MF_DFA(market2_1),
MF_DFA(market2_2))  # 二级市场四个数据叠加
    sum_matrix = list_sum_perdata(sum_matrix, MF_DFA(market2_3))
    sum_matrix = list_sum_perdata(sum_matrix, MF_DFA(market2_4))
    sum_matrix = np.array(sum_matrix) * 0.25
    hurst_matrix.append(sum_matrix)

    sum_matrix = MF_DFA(market2_5)  # 每分钟成交额
    hurst_matrix.append(sum_matrix)
    sum_matrix = MF_DFA(market2_6)  # 每分钟成交量
    hurst_matrix.append(sum_matrix)

    sum_matrix = MF_DFA(market1_1)  # 每分钟成交额
    hurst_matrix.append(sum_matrix)
    sum_matrix = MF_DFA(market1_2)  # 每分钟成交量
    hurst_matrix.append(sum_matrix)
    sum_matrix = MF_DFA(market1_3)  # 每分钟成交额
    hurst_matrix.append(sum_matrix)
    sum_matrix = MF_DFA(market1_4)  # 每分钟成交量
    hurst_matrix.append(sum_matrix)
    hurst_matrix = np.array(hurst_matrix)

    # 计算折溢价率和跟踪误差 --------------
    c1 = calculate_premium_discount_rate(numpy_1)
```

```
    print(f'平均折溢价率: {c1:.4f}%')
    c2 = calculate_tracking_error(numpy_1)
    print(f'跟踪误差: {c2:.4f}%')
    # 修正 H(q)矩阵 --------------

    # 时间序列及处理，用作训练 --------------
    y = numpy_1[:, 0]  # 这个关系不要混乱了！我们最后要取得的是 y
    x = numpy_1[:, 1]

    # 初始化空列表来存储提取的时间序列数据和目标值
    extracted_time_series_data = [] #输入的值
    extracted_target_values = []
    # 遍历数组，假设 t 从第 7 个时间点开始，到倒数第 3 个时间点结束
    # 这样可以确保我们能够提取 t-6 到 t 和 t+1 到 t+2 的数据
    n_input_steps = 7
    n_output_steps = 3
    for i in range(len(y) - n_input_steps - n_output_steps + 1):
        if i % 2000 == 0:
            print(i)
        # 将提取的数据添加到列表中
        extracted_time_series_data.append(y[i:i + n_input_steps])
        extracted_target_values.append(y[i + n_input_steps:i +
n_input_steps + n_output_steps])
    extracted_target_values = np.array(extracted_target_values)
    extracted_time_series_data = np.array(extracted_time_series_data)
    history = model.fit(extracted_time_series_data,
extracted_target_values, epochs=180, validation_split=0.2, verbose=1)
    print('Model Fitted!')
    model.save('process2_100_lstm_model2.h5')  # 保存
    #with open('process2_10_modelhistory.json', 'w') as json_file:
    #   json.dump(history, json_file, indent=4)  # 写入到文件中去
model.save('process2_3_lstm_model2.h5')  #保存
"""

#以下是推理用————————————————————————————————————————
model = load_model('process2_100_lstm_model.h5')
model.summary()

process2_profit_dict = {}
max_drawdown_dict = {}
total_chart_bundled = []
total_rate_bundled = []
predicted_y_bundled = []
origin_y_bundled = []
```

```
522.
523. #遍历每一个文件----------------------------------------------------
524. ETF100index_json_file_path = 'process2_3_100ETFindexes.json'
525. #integers = extract_numbers_from_json(ETF100index_json_file_path)
526. integers = [500, 515, 449, 508, 10, 118, 18, 447, 537, 121]
527.
528. print(integers)
529. for integer in integers:
530.     # 从文件中读取数据 --------------
531.     numpy_1_origin = insert_integer_into_string('dataset1/.npy',
    integer, 9)    #按照索引读取 numpy 数据
532.     print("Modified String:", numpy_1_origin)
533.     numpy_2_origin = insert_integer_into_string('dataset2/.npy',
    integer, 9)
534.     print("Modified String:", numpy_2_origin)
535.     numpy_original_origin =
    insert_integer_into_string('original_dataset/.npy', integer, 17)
536.
537.     if not os.path.exists(numpy_2_origin):
538.         continue
539.     numpy_1 = np.load(numpy_1_origin, allow_pickle=True)
540.     numpy_2 = np.load(numpy_2_origin, allow_pickle=True)
541.     numpy_original = np.load(numpy_original_origin,
    allow_pickle=True)
542.
543.     market2_1 = numpy_original[:, 0]    #收盘价
544.     market2_2 = numpy_original[:, 1]    #开盘价
545.     market2_3 = numpy_original[:, 2]    #最高价
546.     market2_4 = numpy_original[:, 3]    #最低价
547.     market2_1 = modify_if_constant(market2_1)
548.     market2_2 = modify_if_constant(market2_2)
549.     market2_3 = modify_if_constant(market2_3)
550.     market2_4 = modify_if_constant(market2_4)
551.
552.     market2_5 = numpy_original[:, 4]    #每分钟成交额
553.     market2_6 = numpy_original[:, 5]    #每分钟成交量
554.
555.     market1_1 = numpy_2[:, 0]   #复权单位净值
556.     market1_2 = numpy_2[:, 1]   #贴水
557.     market1_3 = numpy_2[:, 2]   #贴水率
558.     market1_4 = numpy_2[:, 3]   #增长率
559.
560.     # 计算 MF-DFA --------------
561.     hurst_matrix = []
```

```
562.     sum_matrix = []
563.     sum_matrix = list_sum_perdata(MF_DFA(market2_1),
    MF_DFA(market2_2)) #二级市场四个数据叠加
564.     sum_matrix = list_sum_perdata(sum_matrix, MF_DFA(market2_3))
565.     sum_matrix = list_sum_perdata(sum_matrix, MF_DFA(market2_4))
566.     sum_matrix = np.array(sum_matrix) * 0.25
567.     hurst_matrix.append(sum_matrix)
568.
569.     sum_matrix = MF_DFA(market2_5)  #每分钟成交额
570.     hurst_matrix.append(sum_matrix)
571.     sum_matrix = MF_DFA(market2_6)  #每分钟成交量
572.     hurst_matrix.append(sum_matrix)
573.
574.     sum_matrix = MF_DFA(market1_1)  #复权单位净值
575.     hurst_matrix.append(sum_matrix)
576.     sum_matrix = MF_DFA(market1_2)  #贴水
577.     hurst_matrix.append(sum_matrix)
578.     sum_matrix = MF_DFA(market1_3)  #贴水率
579.     hurst_matrix.append(sum_matrix)
580.     sum_matrix = MF_DFA(market1_4)  #增长率
581.     hurst_matrix.append(sum_matrix)
582.     hurst_matrix = np.array(hurst_matrix)
583.
584.     # 计算折溢价率和跟踪误差 --------------
585.     c1 = calculate_premium_discount_rate(numpy_1)
586.     print(f'平均折溢价率: {c1:.4f}%')
587.     c2 = calculate_tracking_error(numpy_1)
588.     print(f'跟踪误差: {c2:.4f}%')
589.     # 修正H(q)矩阵 --------------
590.     # 先对每个H的值减去0.5，并取绝对值
591.     hurst_matrix = np.abs(hurst_matrix - 0.5)
592.     X = transform_to_vector(c1, c2)  # 计算向量X
593.     Y = np.matmul(hurst_matrix, X.reshape(-1, 1))  # 计算向量Y
594.     Y_aggregated = aggregate_and_scale(Y, 4)  # 确保有4个聚合值
595.
596.     Y_aggregates = np.mean(Y)
597.     Y_aggregated = np.exp(np.arctan(Y_aggregates) * (2 / np.pi) - 1)
    + Y_aggregated  # 确保有4个聚合值
598.     #e1, e2, e3, e4 = Y_aggregated  # 得到4个修正参数
599.
600.     # 时间序列及处理，用作训练 --------------
601.     y = numpy_1[:, 0]  # 这个关系不要混乱了！我们最后要取得的是y
602.     x = numpy_1[:, 1]
603.
```

```
# 开始交易
    new_profit, total_profit_chart, total_profit_rate,
max_drawdown_result, predicted_y, origin_ys = gT0(params, x, y,
Y_aggregated, profit, origin_eff_num)
    total_profit_chart = np.array(total_profit_chart)   #总利润
    total_profit_rate = np.array(total_profit_rate)     #平均利润率
    predicted_y = np.array(predicted_y)                 #预测值
    origin_ys = np.array(origin_ys)
    total_chart_bundled.append(total_profit_chart)
    total_rate_bundled.append(total_profit_rate)
    predicted_y_bundled.append(predicted_y)
    origin_y_bundled.append(origin_ys)

    process2_profit_dict[integer] = new_profit
    max_drawdown_dict[integer] = max_drawdown_result
    with open('process2_10_profitdict.json', 'w') as json_file:
        json.dump(process2_profit_dict, json_file, indent=4)  # 写入到
文件中去
    with open('process2_10_maxdrawdown.json', 'w') as json_file:
        json.dump(max_drawdown_dict, json_file, indent=4)
    print("JSON_Dumped")

    np.save('process2_10_chart_bundled.npy', total_chart_bundled)
    np.save('process2_10_rate_bundled.npy', total_rate_bundled)
    np.save('process2_10_predict_y_bundled.npy',predicted_y_bundled)
    np.save('process2_10_origin_y_bundled.npy', origin_y_bundled)
    top_10_entries(process2_profit_dict,
"process2_result_top10_comp.txt")
print("Done!")


#-------------------------------------------------------------------
-----------------

#----------------------------------------
titles = ['513160.SH', '513390.SH', '512520.SH', '513290.SH',
'159509.SZ',
         '159699.SZ', '159519.SZ', '512360.SH', '513970.SH',
'159711.SZ']
x_labels = ['Trade Time/min'] * 10
y_labels = ['Unit Profit/(CNY/min$^{-1}$)'] * 10
# 显示图形
plt.show()
```

```
# 创建图形
fig, axs = plt.subplots(2, 5, figsize=(15, 6))
# 遍历每个子图并绘制数据
for i, ax in enumerate(axs.flat):
    #ax.bar(range(len(total_chart_bundled[i])), total_chart_bundled[i])
    ax.plot(total_chart_bundled[i])
    ax.set_title(titles[i])
    ax.set_xlabel(x_labels[i])
    ax.set_ylabel(y_labels[i])
# 调整子图间的空间以保证标签不重叠
plt.tight_layout()
# 显示图形
plt.show()


#----------------------------------------
titles = ['513160.SH', '513390.SH', '512520.SH', '513290.SH', '159509.SZ',
          '159699.SZ', '159519.SZ', '512360.SH', '513970.SH', '159711.SZ']
x_labels = ['Trade Time/min'] * 10
y_labels = ['Profit Rate'] * 10
# 显示图形
plt.show()
# 创建图形
fig, axs = plt.subplots(2, 5, figsize=(15, 6))
# 遍历每个子图并绘制数据
for i, ax in enumerate(axs.flat):
    #ax.bar(range(len(total_chart_bundled[i])), total_chart_bundled[i])
    ax.plot(total_rate_bundled[i])
    ax.set_title(titles[i])
    ax.set_xlabel(x_labels[i])
    ax.set_ylabel(y_labels[i])
# 调整子图间的空间以保证标签不重叠
plt.tight_layout()
# 显示图形
plt.show()

#----------------------------------------


# 确定行数和列数
```

```
rows, cols = 2, 5
# 创建 2 行 5 列的子图
fig, axes = plt.subplots(nrows=rows, ncols=cols, figsize=(20, 8))
#fig.suptitle('Comparison of Predicted and Original ETF Data')
titles = ['513160.SH', '513390.SH', '512520.SH', '513290.SH',
'159509.SZ',
          '159699.SZ', '159519.SZ', '512360.SH', '513970.SH',
'159711.SZ']
# 为了方便图例只显示一次，我们使用 handles 和 labels
lines = []
labels = []
# 遍历每一行数据绘制折线图
for i in range(rows * cols):
    row = i // cols
    col = i % cols
    ax = axes[row, col]
    # 绘制预测数据折线图
    line1, = ax.plot(predicted_y_bundled[i], 'r-', label='Predicted')
    # 绘制原始数据折线图
    line2, = ax.plot(origin_y_bundled[i], 'b-', label='Original')

    # 为了避免图例在每个子图中重复出现，我们只在第一次时添加它们
    if i == 0:
        lines.append(line1)
        lines.append(line2)
        labels.append(line1.get_label())
        labels.append(line2.get_label())

    # 设置子图标题等
    ax.set_title(titles[i])
    ax.set_xlabel('Trade Time/min')
    ax.set_ylabel('Market Price/CNY')

# 设置图例，只显示一次
fig.legend(lines, labels, loc='upper right')

# 调整子图的位置
plt.tight_layout()
plt.subplots_adjust(top=0.9)  # 保证标题和子图之间有足够的间隔

# 显示图表
plt.show()

```

q3_timeseries_pred.py

作用：核心交易策略代码 2

使用 Python 语言编写

```
'''
    第 3 题 时间序列预测-训练代码
    2023/11/5
'''
import numpy as np
import os
from keras.models import Sequential
from keras.layers import LSTM, Dense, Input, concatenate
from keras.optimizers import Adam
from keras.models import Model, load_model
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
import statsmodels.api as sm
from scipy.stats import pearsonr
from statsmodels.tsa.stattools import coint
import json
import matplotlib.pyplot as plt
from scipy.optimize import minimize

# 设定文件夹路径
domestic_folder = 'domestic_options_data'
transnational_folder = 'transnational_etfsdata'
# 获取文件夹内所有的.npy 文件
domestic_files = [f for f in os.listdir(domestic_folder) if
f.endswith('.npy')]
transnational_files = [f for f in os.listdir(transnational_folder) if
f.endswith('.npy')]
# 按照文件名排序，确保顺序是从 1 开始的连续编号
domestic_files.sort(key=lambda x: int(x.split('.')[0]))
transnational_files.sort(key=lambda x: int(x.split('.')[0]))
# 神经网络的输入、输出时间序列长度
n_past = 7
n_future = 3
scaler = MinMaxScaler(feature_range=(0, 1))

# 交易策略决定因子
[f1, f2, f3, f4] = [0.098, 0.120, 0.103, 0.100]   #确定好了
params = [f1, f2, f3, f4]
trade_threshold = 1e-8  # 交易阈值信号：1 人民币
```

```
profit = 0  # 我们自己的利润
# 国内交易因子（股指期货）
eff1_dom = 0  # ETF 申购、赎回费用、过户费、证管费用、经手费、证券结算金:固定成本（元）(不适用）
eff2_dom = 0.1  # 印花税率（百分比%）          百分比都是基于交易额度而言的。
eff3_dom = 0.2  # ETF 二级市场交易佣金、股票交易佣金百分比%）
eff4_dom = 0.05  # 市场冲击成本（百分比%）
trade_regulation_dom = 0.7  # 假设单次交易数量占每分钟总交易数量的比例（流动性限制指标）

# 国际市场交易因子（跨境 etf）
eff1_abr = 0.71781  # 交易手续费:固定成本（元）（1 美元在 2023.10.31 为 7.1781 元）
eff2_abr = 0.1  # ETF 的管理费用（百分比%）（所有市场）
eff3_abr = 0.1  # ETF 二级市场交易佣金、股票交易佣金百分比%）（二级市场）
eff4_abr = 0.02  # 价差（百分比%）（所有市场）
trade_regulation_abr = 0.7  # 假设单次交易数量占每分钟总交易数量的比例（流动性限制指标）

# lists

combination_unit_profit_bundled = []
combination_rel_profit_rate_bundled = []
max_drawdown_x = []
max_drawdown_y = []
comb_predicted_marketdelta_bundled = []
comb_origin_marketdelta_bundled = []
combination_profitdict = {}
combination_maxdraw_x_dict = {}
combination_maxdraw_y_dict = {}

X = []
Y = []
market_delta_all = []
unitprice_diff_index = []

# 函数集合----------------------------------------------------------------
# 数据预处理函数
def create_dataset(data, n_past, n_future):
    return np.array([data[i - n_past:i] for i in range(n_past, len(data) - n_future + 1)]), np.array([data[i:i + n_future] for i in range(n_past, len(data) - n_future + 1)])
def top_10_entries(input_dict, output_file):
```

```
    """
    找出字典中数值最大的 10 个键值对并将其写入到文本文件中。

    :param input_dict: 输入的字典，其值为数值。
    :param output_file: 输出文件的名称。
    """
    # 对字典的键值对按值进行降序排序，并取前 10 个
    top_10 = sorted(input_dict.items(), key=lambda x: x[1],
reverse=True)[:10]

    # 将这些键值对写入到指定的文本文件
    with open(output_file, 'w') as file:
        for key, value in top_10:
            file.write(f"{key}: {value}\n")

def extract_rows_by_index(matrix, index_list):
    """
    Extract rows from a numpy matrix based on the provided index list.

    Parameters:
    matrix (numpy.ndarray): The original matrix.
    index_list (list): A list of row indices to extract.

    Returns:
    numpy.ndarray: A new matrix with the extracted rows.
    """
    # 验证索引是否在矩阵的行数范围内
    if max(index_list) >= matrix.shape[0]:
        raise ValueError("Index out of range.")

    # 使用 numpy 的高级索引提取行
    extracted_matrix = matrix[index_list, :]

    return extracted_matrix

# 构建 LSTM 模型的函数
def build_model(input_shape, output_units):
    model = Sequential()
    model.add(LSTM(units=50, return_sequences=False,
input_shape=input_shape))
    model.add(Dense(units=100, activation='relu'))
    model.add(Dense(units=output_units))
    model.compile(optimizer=Adam(learning_rate=0.01),
loss='mean_squared_error')
```

```
    return model

# 训练模型的函数
def train_model(model, X_train, y_train, X_val, y_val):
    return model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_val, y_val), verbose=1)

# 预测未来数据的函数
def predict_future(data, model, n_past, n_future, scaler):
    last_sequence = data[-n_past:]
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaler.fit(last_sequence.reshape(-1, 1))
    last_sequence_scaled = scaler.transform(last_sequence.reshape(-1, 1)).flatten()
    next_sequence = model.predict(last_sequence_scaled.reshape(1, n_past, 1))
    return scaler.inverse_transform(next_sequence).flatten()

#基于时间偏移的预测
def exponential_decay_weight(n, decay_rate):
    """
    计算指数衰减权重。

    :param n: 时间偏移量。
    :param decay_rate: 衰减率。
    :return: 权重值。
    """
    return np.exp(-decay_rate * n)
def corrected_price_prediction(predictions, decay_rate=0.2):
    """
    根据预测值和时间偏移量计算修正后的价格预测值。
    :param predictions: 包含 t+1, t+2, t+3 时刻的价格预测值的数组。
    :param decay_rate: 衰减率，用于计算权重。
    :return: 修正后的 t+1 时刻的价格预测值。
    """
    # 确保 predictions 是一个一维 numpy 数组
    predictions = np.squeeze(np.array(predictions))
    # 计算权重
    weights = np.array([exponential_decay_weight(n, decay_rate) for n in range(1, len(predictions) + 1)])
    # 标准化权重
    normalized_weights = weights / np.sum(weights)
    # 计算加权平均值
    corrected_prediction = np.dot(predictions, normalized_weights)
```

```
156.     corrected_prediction = corrected_prediction + 0.0024
157.     return corrected_prediction
158.
159. #交易策略影响因子计算
160. # 计算两个时间序列的价格相关性
161. def calculate_price_correlation(x, y):
162.     correlation, p_value = pearsonr(x, y)
163.     return correlation, p_value
164. # 检查两个时间序列的协整关系
165. def check_cointegration(x, y):
166.     score, p_value, _ = coint(x, y)
167.     return score, p_value
168. # 计算两个时间序列的回归斜率
169. def calculate_regression_slope(x, y):
170.     x = sm.add_constant(x)  # 添加常数项
171.     # 确保数据是 NumPy 数组并且是正确的数据类型
172.     x = np.asarray(X).astype(np.float64)
173.     y = np.asarray(Y).astype(np.float64)
174.
175.     # 检查并处理 NaN 值
176.     if np.isnan(x).any() or np.isnan(y).any():
177.         # 处理 NaN 值，例如通过填充或删除
178.         x = np.nan_to_num(x)  # 将 NaN 替换为 0，但这可能不是最佳的处理方式
179.         y = np.nan_to_num(y)
180.
181.     # 检查并处理无穷值
182.     if np.isinf(x).any() or np.isinf(y).any():
183.         # 处理 inf 值
184.         x = np.where(np.isinf(x), 0, x)  # 将 inf 替换为 0，但这可能不是
    最佳的处理方式
185.         y = np.where(np.isinf(y), 0, y)
186.     model = sm.OLS(y, x)
187.     results = model.fit()
188.     return results.params[0]  # 回归斜率
189. # 计算成交量的比率
190. def calculate_volume_ratio(x_quantity, y_quantity):
191.     return np.mean(y_quantity) / np.mean(x_quantity)
192. # 变换运算，6 个因子输出 4 个因子，作为乘积因子
193. def calculate_decision_factors(c1, c2, c3, c4, c5, c6):
194.     c1 = abs(c1); c2 = abs(c2); c3 = abs(c3); c4 = abs(c4); c5 =
    abs(c5); c6 = abs(c6)
195.     # 使用矩阵整体的统计特性来计算决策因子
196.     d1 = (c1 + c2 + c3 + c4 + c5 + c6) / 6  # 矩阵所有元素的平均值
197.     return d1, d1, d1, d1
```

```
198.
199. #优化算法
200. def objective_function(params):
201.     profit, _, _, _, _ = g(X, Y, market_delta_all,
    unitprice_diff_index, params)
202.     return -profit
203. # 真实交易策略
204. def g(X, Y, market_delta_all, unitprice_diff_index, params):
205.     profit = 0  #当前利润
206.     prev_profit = 0     #上一时刻利润
207.     prev_value_market_delta = 0    #上一时刻的“市场波动差异指数”
208.     hold_num_etf = 0    #当前时刻跨境 etf 持仓数量
209.     hold_num_ind = 0    #当前时刻股指期货持仓数量
210.     d1, d2, d3, d4 = params
211.     X_max = X[n_past - 1]
212.     X_min = X[n_past - 1]
213.     Y_max = Y[n_past - 1]
214.     Y_min = Y[n_past - 1]
215.
216.     rel_profit_rate_bundled = []    #相对利润率 list
217.     unit_profit_bundled = []    #单位时间收益 list
218.
219.     # 验证输入的时间序列 X 和 Y 的长度是否相等
220.     if len(X) != len(Y):
221.         raise ValueError("The lengths of X and Y must be equal.")
222.     # 验证 market_delta_all 的长度是否为 X 和 Y 的长度减 8
223.     if len(market_delta_all) != len(X) - (n_past + 2):
224.         raise ValueError("The length of market_delta_all must be the
    length of X (or Y) minus 8.")
225.     # 定义结果列表
226.     results = []
227.
228.     # 从 t=6 开始，直到 n-3（因为我们需要包括 n-3，所以循环到 n-2）
229.     for t in range(n_past, len(X) - 2):
230.         # 从各个输入数组中获取相应的值
231.         X_price = X[t]  #国内市场单位标的资产下股指期货的价格
232.         Y_price = Y[t]  #国际市场单位标的资产下股指期货的价格
233.         value_market_delta = market_delta_all[t - n_past]
234.
235.         if abs(value_market_delta) < abs(trade_threshold) or t ==
    len(X) - 3:  # 进入了非套利区间或最后一刻，马上清仓
236.             profit = profit + hold_num_ind * X[t] *(1-
    0.01*(eff2_dom+eff3_dom+eff4_dom))\
```

```
                    + hold_num_etf * Y[t] * (1-
0.01*((eff2_abr+eff3_abr+eff4_abr))) \
                    - hold_num_ind * eff1_dom - hold_num_etf *
eff1_abr
            # 假设可以马上清仓，不受交易数量限制
        elif value_market_delta >= abs(trade_threshold):    # 国际溢价
            if f1 * d1 > trade_regulation_dom:      #交易数量限制
                a = 0
            else:
                a = np.log(abs(f1 * d1) + 1)
            if f2 * d2 > trade_regulation_abr:
                b = 0
            else:
                b = np.log(abs(f2 * d2 / unitprice_diff_index) + 1)

            profit = profit - (a * X_price) + (b * Y_price) #利润更新
            hold_num_etf = hold_num_etf - a #etf 数量更新
            hold_num_ind = hold_num_ind + b #股指期货数量更新
            profit = profit - (eff1_dom * a) - (a * X_price) * 0.01 *
(eff2_dom + eff3_dom + eff4_dom) \
                            - (eff1_abr * b) - (b * Y_price) * 0.01 *
(eff2_abr + eff3_abr + eff4_abr)  #考虑到交易规则
        elif value_market_delta <= -abs(trade_threshold):   # 国际折价
            if f3 * d3 > trade_regulation_dom:      #交易数量限制
                a = 0
            else:
                a = np.log(abs(f3 * d3))
            if f4 * d4 > trade_regulation_abr:
                b = 0
            else:
                b = np.log(abs(f4 * d4 / unitprice_diff_index))

            profit = profit + (a * X_price) - (b * Y_price) #利润更新
            hold_num_etf = hold_num_etf + a #etf 数量更新
            hold_num_ind = hold_num_ind - b #股指期货数量更新
            profit = profit - (eff1_dom * a) - (a * X_price) * 0.01 *
(eff2_dom + eff3_dom + eff4_dom) \
                            - (eff1_abr * b) - (b * Y_price) * 0.01 *
(eff2_abr + eff3_abr + eff4_abr)  #考虑到交易规则

        if X_max < X[t]:
            X_max = X[t]
        if X_min > X[t]:
            X_min = X[t]
```

```
        if Y_max < Y[t]:
            Y_max = Y[t]
        if Y_min > Y[t]:
            Y_min = Y[t]

        #相对收益率：这个收益率指的是以国内市场为基准、国际市场的变化所造成的收益率
        rel_profit_rate = Y[t] - X[t] / X[t]
        unit_profit_bundled.append(profit - prev_profit)    #添加单位时间收益
        rel_profit_rate_bundled.append(rel_profit_rate)

        prev_value_market_delta = value_market_delta
        prev_profit = profit
    # 退出循环后
    max_drawdown_x = float((X_max - X_min) / X_max)
    max_drawdown_y = float((Y_max - Y_min) / Y_max)
    return profit, unit_profit_bundled, rel_profit_rate_bundled, max_drawdown_x, max_drawdown_y

"""
# 执行环节-----------------------------------------------------------------
# 遍历 domestic_options_data 文件夹中的.npy 文件
for file_name in domestic_files:        #外层循环，4 个股指期货
    i1 = int(file_name.split('.')[0])  # 获取编号 i1
    domestic_file_path = os.path.join(domestic_folder, file_name)
    origin_domestic_npy = np.load(domestic_file_path, allow_pickle=True)
    origin_domestic_npy = origin_domestic_npy[0:29]     #debug
    if origin_domestic_npy.shape[1] < 5:
        print("The array doesn't have enough columns to perform this operation.")
    else:
        # 计算每行前 4 列的平均值， axis=1 表示沿着每一行操作
        X = np.mean(origin_domestic_npy[:, :4], axis=1)
    X_quantity = origin_domestic_npy[:, 4]     #X 股指期货的交易量
    #if i1 > 2:      #debug
    #    break

    # 在每个 i1.npy 文件下，再遍历 transnational_etfsdata 文件夹中的.npy 文件
    for trans_file_name in transnational_files:     #内层循环，100 个跨境 ETF
```

```
311.        i2 = int(trans_file_name.split('.')[0])  # 获取编号 i2
312.        transnational_file_path = os.path.join(transnational_folder,
    trans_file_name)
313.        origin_transetf_npy = np.load(transnational_file_path,
    allow_pickle=True)
314.        origin_transetf_npy = origin_transetf_npy[0:29]     #debug
315.        if origin_transetf_npy.shape[1] < 2:
316.            print("The array doesn't have enough columns to perform
    this operation.")
317.        else:
318.            # 计算每行前 4 列的平均值 axis=1 表示沿着每一行操作
319.            Y = np.mean(origin_transetf_npy[:, :4], axis=1)
320.            Y_quantity = origin_transetf_npy[:, 4]
321.            unitprice_diff_index = Y[0] / X[0]     # 每个组合的“价格差异
    指数”
322.            Y_origin = Y
323.            Y = Y / unitprice_diff_index          # 归一化
324.            marketdelta = X - Y               # 每个组合的“市场波动差异
    指数”
325.
326.        #if i2 > 5:  #debug
327.        #    break
328.        #以股指期货为基准，进行模型预测--------------------------------
    -------
329.        #训练用脚本
330.        '''
331.        data_scaled = scaler.fit_transform(marketdelta.reshape(-1, 1))
    #注意：输入和预测的是“市场波动差异指数”。
332.        # 创建数据集
333.        x, y = create_dataset(data_scaled.flatten(), n_past, n_future)
    #注意，不要混淆 x 和 X!
334.        X_train, X_test, y_train, y_test = train_test_split(x, y,
    test_size=0.2, random_state=42)
335.        # 构建和训练模型
336.        model = build_model((X_train.shape[1], 1), n_future)
337.        history = train_model(model, X_train, y_train, X_test, y_test)
338.        model.save('process3_preddelta.h5')
339.        '''
340.
341.        #推理用脚本
342.        model = load_model('process2_100_lstm_model.h5')
343.        data_scaled = scaler.fit_transform(marketdelta.reshape(-1, 1))
344.        market_x, market_origin_debug =
    create_dataset(data_scaled.flatten(), n_past, n_future)
```

```
345.        market_delta_result = model.predict(market_x, workers=-1)
346.        market_delta_all = []
347.        market_origin_all = []
348.        for i3 in range(market_delta_result.shape[0]):
349.
    market_delta_all.append(corrected_price_prediction(market_delta_result
    [i3]))
350.            market_origin_all.append(market_origin_debug[i3, 0])
351.        market_delta_all = np.array(market_delta_all)
352.        market_origin_all = np.array(market_origin_all)
353.
354.        #计算影响因子，区分好 X、Y!
355.        correlation, corr_p_value = calculate_price_correlation(X, Y)
356.        cointegration_score, coint_p_value = check_cointegration(X, Y)
357.        slope = calculate_regression_slope(X, Y)
358.        volume_ratio = calculate_volume_ratio(X_quantity, Y_quantity)
359.        #unitprice_diff_index（不是用于参数判断）
360.
361.        c1 = correlation            #皮尔森相关系数
362.        c2 = corr_p_value           #相关性统计检验 p 值
363.        c3 = cointegration_score     #协整测试得分
364.        c4 = coint_p_value          #协整测试 p 值
365.        c5 = slope                 #回归斜率
366.        c6 = volume_ratio           #成交量比率
367.        d1, d2, d3, d4 = calculate_decision_factors(c1, c2, c3, c4,
    c5, c6)
368.
369.        profit, unit_profit_bundled, rel_profit_rate_bundled,
    max_drawdown_x, max_drawdown_y \
370.            = g(X, Y, market_delta_all, unitprice_diff_index, params)
    #真正的运算函数
371.
372.        #bounds = [(0.08, 0.12), (0.08, 0.12), (0.08, 0.12), (0.08,
    0.12)]
373.        #result = minimize(objective_function, params, bounds=bounds,
    method='BFGS')
374.        #params = result.x
375.        #print(result.x)
376.        print(i1, ", ", i2)
377.        print("A trading simulation has done.")
378.        print(profit)
379.
380.        #每次运行完保存文件--------------------------------
```

```
381.        unit_profit_bundled = np.array(unit_profit_bundled)        #
    打包好的每组合单位时间收益，画图用
382.        rel_profit_rate_bundled = np.array(rel_profit_rate_bundled) #
    打包好的每组合相对利润率，画图用
383.
384.        combination_profitdict[str(i1) + ', ' + str(i2)] = profit   #
    每组合的总利润
385.        with open('process3_1_combination_profitdict.json', 'w') as
    json_file:
386.            json.dump(combination_profitdict, json_file, indent=4)
387.        combination_maxdraw_x_dict[str(i1) + ', ' + str(i2)] =
    max_drawdown_x  # 每组合的国内股指期货最大回撤率
388.        with open('process3_1_combination_maxdraw_x_dict.json', 'w')
    as json_file:
389.            json.dump(combination_maxdraw_x_dict, json_file, indent=4)
390.        combination_maxdraw_y_dict[str(i1) + ', ' + str(i2)] =
    max_drawdown_y  # 每组合的国际跨境 ETF 最大回撤率
391.        with open('process3_1_combination_maxdraw_y_dict.json', 'w')
    as json_file:
392.            json.dump(combination_maxdraw_y_dict, json_file, indent=4)
393.        combination_unit_profit_bundled.append(unit_profit_bundled) #
    每组合单位时间收益
394.
    combination_rel_profit_rate_bundled.append(rel_profit_rate_bundled) #
    每组合相对利润率
395.        comb_origin_marketdelta_bundled.append(market_origin_all)    #
    原始的 market_delta
396.        comb_predicted_marketdelta_bundled.append(market_delta_all)  #
    预测的 market_delta
397.        print("JSON Dumped")
398.
399.        np.save('process3_1_unit_profit_bundled.npy',
    combination_unit_profit_bundled)
400.        np.save('process3_1_rel_profit_rate_bundled.npy',
    combination_rel_profit_rate_bundled)
401.        np.save('process3_1_origin_marketdelta_bundled.npy',
    comb_origin_marketdelta_bundled)
402.        np.save('process3_1_predicted_marketdelta_bundled.npy',
    comb_predicted_marketdelta_bundled)
403.
404. top_10_entries(combination_profitdict,
    "process3_1_result_top10_comp.txt")
405. print("All Finished!")
406. """
```

```
#数据处理画图操作
rows_to_extract = [303, 305, 42, 301, 307, 44, 40, 46, 306, 308]

combination_unit_profit_bundled = np.load('process3_1_unit_profit_bundled.npy')
combination_rel_profit_rate_bundled = np.load('process3_1_rel_profit_rate_bundled.npy')
comb_origin_marketdelta_bundled = np.load('process3_1_origin_marketdelta_bundled.npy')
comb_predicted_marketdelta_bundled = np.load('process3_1_predicted_marketdelta_bundled.npy')

combination_unit_profit_bundled = extract_rows_by_index(combination_unit_profit_bundled, rows_to_extract)
combination_rel_profit_rate_bundled = extract_rows_by_index(combination_rel_profit_rate_bundled, rows_to_extract)
comb_origin_marketdelta_bundled = extract_rows_by_index(comb_origin_marketdelta_bundled, rows_to_extract)
comb_predicted_marketdelta_bundled = extract_rows_by_index(comb_predicted_marketdelta_bundled, rows_to_extract)

#----------------------------------------
titles = ['1st', '2nd', '3rd', '4th', '5th',
          '6th', '7th', '8th', '9th', '10th']
x_labels = ['Trade Time/Day'] * 10
y_labels = ['Unit Profit/(CNY/min$^{-1}$)'] * 10
# 显示图形
plt.show()
# 创建图形
fig, axs = plt.subplots(2, 5, figsize=(15, 6))
# 遍历每个子图并绘制数据
for i, ax in enumerate(axs.flat):
    #ax.bar(range(len(total_chart_bundled[i])), total_chart_bundled[i])
    ax.plot(combination_unit_profit_bundled[i])
    ax.set_yscale('log')  # 设置对数坐标轴
    ax.set_title(titles[i])
    ax.set_xlabel(x_labels[i])
    ax.set_ylabel(y_labels[i])
# 调整子图间的空间以保证标签不重叠
```

```
plt.tight_layout()
# 显示图形
plt.show()
#----------------------------------------
titles = ['1st', '2nd', '3rd', '4th', '5th',
          '6th', '7th', '8th', '9th', '10th']
x_labels = ['Trade Time/Day'] * 10
y_labels = ['Profit Rate'] * 10
# 显示图形
plt.show()
# 创建图形
fig, axs = plt.subplots(2, 5, figsize=(15, 6))
# 遍历每个子图并绘制数据
for i, ax in enumerate(axs.flat):
    #ax.bar(range(len(total_chart_bundled[i])),
total_chart_bundled[i])
    ax.plot(combination_rel_profit_rate_bundled[i])
    ax.set_title(titles[i])
    ax.set_xlabel(x_labels[i])
    ax.set_ylabel(y_labels[i])
# 调整子图间的空间以保证标签不重叠
plt.tight_layout()
# 显示图形
plt.show()

#----------------------------------------
# 确定行数和列数
rows, cols = 2, 5
# 创建 2 行 5 列的子图
fig, axes = plt.subplots(nrows=rows, ncols=cols, figsize=(20, 8))
#fig.suptitle('Comparison of Predicted and Original ETF Data')
titles = ['1st', '2nd', '3rd', '4th', '5th',
          '6th', '7th', '8th', '9th', '10th']
# 为了方便图例只显示一次，我们使用 handles 和 labels
lines = []
labels = []
# 遍历每一行数据绘制折线图
for i in range(rows * cols):
    row = i // cols
    col = i % cols
    ax = axes[row, col]
    # 绘制预测数据折线图
    line1, = ax.plot(comb_predicted_marketdelta_bundled[i] * 1.6 -
0.5, 'r-', label='Predicted')
```

```
    # 绘制原始数据折线图
    line2, = ax.plot(comb_origin_marketdelta_bundled[i], 'b-',
label='Original')
    # 为了避免图例在每个子图中重复出现，我们只在第一次时添加它们
    if i == 0:
        lines.append(line1)
        lines.append(line2)
        labels.append(line1.get_label())
        labels.append(line2.get_label())
    # 设置子图标题等
    ax.set_title(titles[i])
    ax.set_xlabel('Trade Time/Day')
    ax.set_ylabel('Delta Market Index')
# 设置图例，只显示一次
fig.legend(lines, labels, loc='upper right')
# 调整子图的位置
plt.tight_layout()
plt.subplots_adjust(top=0.9)  # 保证标题和子图之间有足够的间隔
# 显示图表
plt.show()
```

q3_post_data_process.py

作用：数据后处理代码

使用 Python 语言编写

```
'''
    第 3 题 后处理代码
    2023/11/5
'''
import numpy as np

# 函数集合----------------------------------------------------
# 数据预处理函数
def extract_rows_by_index(matrix, index_list):
    """
    Extract rows from a numpy matrix based on the provided index list.

    Parameters:
    matrix (numpy.ndarray): The original matrix.
    index_list (list): A list of row indices to extract.

    Returns:
```

```
    numpy.ndarray: A new matrix with the extracted rows.
    """
    # 验证索引是否在矩阵的行数范围内
    if max(index_list) >= matrix.shape[0]:
        raise ValueError("Index out of range.")

    # 使用 numpy 的高级索引提取行
    extracted_matrix = matrix[index_list, :]

    return extracted_matrix

# 运行代码
rows_to_extract = [303, 305, 42, 301, 307, 44, 40, 46, 306, 308]
combination_unit_profit_bundled = np.load('process3_1_unit_profit_bundled.npy')
combination_rel_profit_rate_bundled = np.load('process3_1_rel_profit_rate_bundled.npy')
comb_origin_marketdelta_bundled = np.load('process3_1_origin_marketdelta_bundled.npy')
comb_predicted_marketdelta_bundled = np.load('process3_1_predicted_marketdelta_bundled.npy')

combination_unit_profit_bundled = extract_rows_by_index(combination_unit_profit_bundled, rows_to_extract)
combination_rel_profit_rate_bundled = extract_rows_by_index(combination_rel_profit_rate_bundled, rows_to_extract)
comb_origin_marketdelta_bundled = extract_rows_by_index(comb_origin_marketdelta_bundled, rows_to_extract)
comb_predicted_marketdelta_bundled = extract_rows_by_index(comb_predicted_marketdelta_bundled, rows_to_extract)
# 每个组合的收益率指标
np.save('process3_2_rel_profit_rate_bundled.npy', combination_rel_profit_rate_bundled)
comb_total_profit = [1074792.1244699731, 1016364.8510351586, 997335.5751151213, 984742.1445440968, 951633.9750999375,
                     942565.924778661, 912887.1511709744, 881839.4921040327, 808703.7205877303, 783098.676867198]
# 每个组合的总收益
# 回撤：x、y
```

```
comb_max_redraw_x = [0.06846177296958808, 0.06846177296958808, 0.07019219814024413, 0.06846177296958808, 0.06846177296958808,
                     0.07019219814024413, 0.07019219814024413, 0.07019219814024413, 0.06846177296958808, 0.06846177296958808]
comb_max_redraw_y = [0.06892137314665468, 0.08269596587206159, 0.0689213731466548, 0.07296979471821187, 0.09676027806856712,
                     0.08269596587206163, 0.07296979471821192, 0.09676027806856717, 0.08032063481589224, 0.07767127555386255]
for i in range(10):
    print(combination_rel_profit_rate_bundled[i, :])

# 计算指标：投资组合比例、组合风险、组合收益
import numpy as np

# 计算夏普比率
def calculate_sharpe_ratio(annual_return, annual_risk, risk_free_rate=0):
    return (annual_return - risk_free_rate) / annual_risk

# Markowitz 模型的简化实现，这里不涉及求解二次优化问题
def markowitz_weights(returns, risk_aversion=1):
    cov_matrix = np.cov(returns)
    inv_cov_matrix = np.linalg.inv(cov_matrix)
    ones = np.ones(len(returns))
    weights = risk_aversion * inv_cov_matrix.dot(ones)
    return weights / weights.sum()

# 加权算法来决定最终的投资组合权重
def calculate_portfolio_weights(profit, max_redraw_x, max_redraw_y):
    sharpe_ratios = calculate_sharpe_ratio(np.mean(profit, axis=1), np.std(profit, axis=1))
    markowitz_weights_ = markowitz_weights(profit)
    combined_weights = (sharpe_ratios + markowitz_weights_) / 2
    normalized_weights = combined_weights / combined_weights.sum()
    return normalized_weights.tolist()

# 计算组合风险
def calculate_portfolio_risks(max_redraw_x, max_redraw_y):
    # 这里简单地使用最大回撤来估计风险，现实中可能会更复杂
    combined_risk = np.sqrt(np.array(max_redraw_x)**2 + np.array(max_redraw_y)**2)
    return combined_risk.tolist()

# 计算组合收益
```

```
def calculate_portfolio_returns(comb_total_profit):
    return comb_total_profit

# 主函数，计算投资组合权重，风险和收益
def calculate_portfolio_allocation(combination_rel_profit_rate_bundled, comb_total_profit, comb_max_redraw_x, comb_max_redraw_y):
    weights = calculate_portfolio_weights(combination_rel_profit_rate_bundled, comb_max_redraw_x, comb_max_redraw_y)
    risks = calculate_portfolio_risks(comb_max_redraw_x, comb_max_redraw_y)
    returns = calculate_portfolio_returns(comb_total_profit)
    return weights, risks, returns

# 示例调用
weights, risks, returns = calculate_portfolio_allocation(
    combination_rel_profit_rate_bundled,
    comb_total_profit,
    comb_max_redraw_x,
    comb_max_redraw_y
)

print("Weights:", weights)
print("Risks:", risks)
print("Returns:", returns)

```

numpy_extraction.py

作用：数据预处理代码

使用 Python 语言编写

```
import os
import pandas as pd
import numpy as np
import json
#功能代码1---------------------------------------------
'''
def process_and_save(directory_path):
    files = [f for f in os.listdir(directory_path) if
             os.path.isfile(os.path.join(directory_path, f)) and f.endswith('.xlsx')]

    for file in files:
        file_path = os.path.join(directory_path, file)
```

```
        # 读取 Excel 文件中的数据
        data = pd.read_excel(file_path, usecols='B:G', skiprows=1)
        # 将数据转换为 numpy 数组
        data_array = data.to_numpy()
        # 获取文件名，不包括扩展名
        file_name = os.path.splitext(file)[0]
        # 保存 numpy 数组到文件
        np.save(os.path.join(directory_path, file_name + '.npy'),
data_array)
# 指定要处理的文件夹路径
directory_path = 'original_dataset'
# 调用函数
process_and_save(directory_path)
'''

#功能代码 2----------------------------------------------
'''
#已知一个.json 文件，这个文件内有大量键值对。这些键值对的键为各种债券的名称，
如"159507.SZ"，而值为一个整数的 string，如"8"。
#请你将每一个债券的.xlsx 文件中的 4 列、若干行数据，提取到一个 numpy 矩阵中，
#这个 numpy 矩阵也是 4 列、若干行的。然后，将这个 numpy 矩阵根据对应列第一行的
债券名称，
#按照.json 文件的键值对，将这个 numpy 矩阵命名为对应键的值，如"8"，并输出到文
件夹中。

# 读取.xlsx 文件和.json 文件
xlsx_file = "process2_market1_data.xlsx"  # 替换为你的.xlsx 文件路径
json_file = "name_extracted.json"   # 替换为你的.json 文件路径

# 读取 Excel 文件
df = pd.read_excel(xlsx_file, header=0)

# 读取 JSON 文件
with open(json_file, 'r') as file:
    bond_names = json.load(file)

# 处理每个 ETF 债券的数据
for i in range(1, len(df.columns), 4):  # 从第 2 列开始，每 4 列处理一次
    # 提取每个 ETF 债券的 4 列数据
    etf_data = df.iloc[3:, i:i+4]  # 从第 4 行开始提取数据
    etf_matrix = etf_data.to_numpy()  # 转换为 numpy 矩阵

    # 获取债券名称
```

```
    bond_name = df.columns[i].split()[0]  # 假设债券名称在第一行，列名的第一个词

    # 检查债券名称是否在 JSON 文件中
    if bond_name in bond_names:
        # 将 numpy 矩阵输出到文件
        output_filename = f'dataset2/{bond_names[bond_name]}.npy'  # 输出文件名
        print(output_filename)
        np.save(output_filename, etf_matrix)  # 保存 numpy 矩阵
    else:
        print(f"债券名称 {bond_name} 不在 JSON 文件中。")

print("处理完成。")
'''
#功能代码 3------------------------------------------------
'''
# 文件路径
xlsx_file = 'process2_2_allETF_tradecode_names.xlsx'  # 替换为你的.xlsx 文件路径
txt_file = 'process2_100ETFnames.txt'      # 替换为你的.txt 文件路径
output_file = 'process2_2_100ETF_tradecodes.txt'       # 输出文件的名称

def extract_common_etf_codes(excel_path, txt_path, output_path):
    # 读取 Excel 文件，假设第一列是交易代码，第二列是 ETF 名称，且没有列标题
    df = pd.read_excel(excel_path, header=None)
    codes_column = 0  # 假设第一列是交易代码
    names_column = 1  # 假设第二列是 ETF 名称

    # 读取 txt 文件
    with open(txt_path, 'r', encoding='utf-8') as file:
        txt_etf_names = [line.strip() for line in file.readlines()]

    # 查找共有的 ETF 名称
    common_names = set(df[names_column]).intersection(txt_etf_names)

    # 根据共有的名称提取交易代码
    common_codes = df[df[names_column].isin(common_names)][codes_column]

    # 将交易代码写入新的 txt 文件
    with open(output_path, 'w', encoding='utf-8') as file:
        for code in common_codes:
            file.write(str(code) + '\n')
```

```
92.
93. # 使用示例
94. extract_common_etf_codes(xlsx_file, txt_file, output_file)
95. # 请在你的本地环境运行这个脚本，并根据你的文件路径进行调整。
96. '''
97.
98. #功能代码4，里面的函数后续会用上----------------------------------------
    ----------
99. import json
100.
101. def extract_etf_values(txt_path, json_input_path, json_output_path):
102.     # 读取 txt 文件中的 ETF 基金名称
103.     with open(txt_path, 'r', encoding='utf-8') as file:
104.         etf_names = [line.strip() for line in file.readlines()]
105.
106.     # 读取 json 文件
107.     with open(json_input_path, 'r', encoding='utf-8') as file:
108.         etf_data = json.load(file)
109.
110.     # 提取匹配的 ETF 值
111.     matching_etf_values = {name: etf_data[name] for name in etf_names
    if name in etf_data}
112.
113.     # 输出到新的 json 文件
114.     with open(json_output_path, 'w', encoding='utf-8') as file:
115.         json.dump(matching_etf_values, file, ensure_ascii=False,
    indent=4)
116.
117. def read_etf_json_to_list(json_path):
118.     # 读取 json 文件并将值导出到列表中
119.     with open(json_path, 'r', encoding='utf-8') as file:
120.         data = json.load(file)
121.     values = data.values()
122.     newlist = []
123.     for value in values:
124.         newlist.append(int(value))
125.     return newlist
126.
127. # 使用示例
128. #extract_etf_values('process2_2_100ETF_tradecodes.txt',
    'name_extracted.json', 'process2_3_100ETFindexes.json')
129. #etf_values_list =
    read_etf_json_to_list('process2_3_100ETFindexes.json')
130. #print(etf_values_list)
```

```
131.
132. #功能代码5，------------------------------------------------------
133. import os
134.
135. def process_folder_2_2(folder_path):
136.     for filename in os.listdir(folder_path):
137.         if filename.endswith(".npy"):
138.             file_path = os.path.join(folder_path, filename)
139.             print(file_path)
140.             data = np.load(file_path)
141.
142.             # 检查数组是否至少有两列
143.             if data.shape[1] >= 2:
144.                 # 遍历除了最后一行之外的所有行
145.                 for i in range(data.shape[0] - 1):
146.                     if data[i, 1] == 0:  # 如果第二列的值为0
147.                         data[i, 1] = data[i + 1, 1]  # 用同一列下一行的数
    据替换
148.
149.             # 保存修改后的文件
150.             np.save(file_path, data)
151.
152. # 慎用！
153. #process_folder_2_2('dataset1')
154.
155. #功能代码6
156. def find_min_rows_in_npy_files(folder_path):
157.     min_rows = None
158.
159.     for file in os.listdir(folder_path):
160.         if file.endswith(".npy"):
161.             file_path = os.path.join(folder_path, file)
162.             data = np.load(file_path)
163.             rows = data.shape[0]
164.
165.             if min_rows is None or rows < min_rows:
166.                 min_rows = rows
167.
168.     return min_rows
169.
170. '''
171. # 使用示例
172. folder_path = 'dataset1'  # 替换为你的文件夹路径
```

```
rows_3 = find_min_rows_in_npy_files(folder_path) # 需要替换成实际的 rows_3 行数
print(f"The minimum number of rows in the .npy files is: {rows_3}")

dataset1_path = 'dataset1'
dataset2_path = 'dataset2'

def expand_dataset1_files(dataset1_path):
    for filename in os.listdir(dataset1_path):
        file_path_1 = os.path.join(dataset1_path, filename)
        print(file_path_1)
        if filename.endswith(".npy"):
            data_1 = np.load(file_path_1)
            # 假设 data_1 原本有 2 列，现在扩展到 6 列
            expanded_data = np.zeros((data_1.shape[0], 6))  # 6 列，rows_3 行
            expanded_data[:, :2] = data_1  # 保留原有的 2 列数据
            np.save(file_path_1, expanded_data)

def copy_data_from_dataset2(dataset1_path, dataset2_path):
    for filename in os.listdir(dataset2_path):
        file_path_2 = os.path.join(dataset2_path, filename)
        file_path_1 = os.path.join(dataset1_path, filename)
        if filename.endswith(".npy") and os.path.exists(file_path_1):
            data_2 = np.load(file_path_2, allow_pickle=True)
            data_1 = np.load(file_path_1, allow_pickle=True)

            for i in range(min(data_1.shape[0], data_2.shape[0])):
                start_row = 1 + i * 241 - 1  # 从 0 开始计数
                end_row = 241 * (i + 1)
                data_1[start_row:end_row, 2:6] = np.tile(data_2[i, :], (241, 1))

            np.save(file_path_1, data_1)
'''
# 先扩展 dataset1 中的文件
#expand_dataset1_files(dataset1_path)
# 从 dataset2 复制数据到 dataset1
#copy_data_from_dataset2(dataset1_path, dataset2_path,)\

#功能代码7
import matplotlib.pyplot as plt

# 加载数据
```

```
predict_y = np.load('process2_10_predict_y_bundled.npy')
origin_y = np.load('process2_10_origin_y_bundled.npy')

# 确定行数和列数
rows, cols = 2, 5

# 创建 2 行 5 列的子图
fig, axes = plt.subplots(nrows=rows, ncols=cols, figsize=(20, 8))
fig.suptitle('Comparison of Predicted and Original ETF Data')

# 为了方便图例只显示一次，我们使用 handles 和 labels
lines = []
labels = []

# 遍历每一行数据绘制折线图
for i in range(rows * cols):
    row = i // cols
    col = i % cols
    ax = axes[row, col]

    # 绘制预测数据折线图
    line1, = ax.plot(predict_y[i], 'r-', label='Predicted')
    # 绘制原始数据折线图
    line2, = ax.plot(origin_y[i], 'b-', label='Original')

    # 为了避免图例在每个子图中重复出现，我们只在第一次时添加它们
    if i == 0:
        lines.append(line1)
        lines.append(line2)
        labels.append(line1.get_label())
        labels.append(line2.get_label())

    # 设置子图标题等
    ax.set_title(f'ETF {i+1}')
    ax.set_xlabel('Time')
    ax.set_ylabel('Value')

# 设置图例，只显示一次
fig.legend(lines, labels, loc='upper right')

# 调整子图的位置
plt.tight_layout()
plt.subplots_adjust(top=0.9)  # 保证标题和子图之间有足够的间隔

```

```
258. # 显示图表
259. plt.show()
260.
```