#tushare获取数据

import tushare as ts

import pandas as pd

ts.set\_token('22357e3a846ee863a7d7ab44011d99e434e8c19e9e08bd76ae8ef79e')

pro = ts.pro\_api()

# 获取沪深300指数日线数据

df = pro.index\_daily(

ts\_code='000300.SH',

start\_date='20100101',

end\_date=pd.Timestamp.today().strftime('%Y%m%d')

)

df = df.sort\_values('trade\_date')

df['trade\_date'] = pd.to\_datetime(df['trade\_date'])

df = df.rename(columns={

'ts\_code': 'stock code',

'trade\_date': 'trade date',

'close': 'closing price',

'open': 'at the opening',

'high': 'top price',

'low': 'floor price',

'pre\_close': 'previous cds',

'change': 'rise and fall',

'vol': 'turnover',

'amount': 'volume of transaction'

})

# 保存到Excel文件

file\_path = r"C:\Users\sysws\Desktop\300.csv"

df.to\_csv(file\_path, index=False)

#金融文本爬虫

mport pyautogui

import time

import subprocess

startNum = 834

endNum = 871

def open\_program(path):

print(f"Opening program: {path}")

subprocess.Popen(path)

time.sleep(20) # Pause to wait for the program to load

def click\_at(x, y, description):

print(f"Clicking at ({x}, {y}) - {description}")

pyautogui.click(x, y)

time.sleep(4) # Pause to wait for the action to complete

def type\_text(text):

print(f"Typing text: {text}")

pyautogui.typewrite(str(text))

time.sleep(4) # Pause to wait for the typing to complete

def press\_enter():

print("Pressing Enter key")

pyautogui.press('enter')

time.sleep(4) # Pause to wait for the action to complete

def check\_and\_click\_image(image\_path, region):

print(f"Checking for image: {image\_path} in region: {region}")

try:

location = pyautogui.locateOnScreen(image\_path, confidence=0.8, region=region)

if location is not None:

print("Image found, clicking on it.")

pyautogui.click(location)

time.sleep(4) # Pause to wait for the action to complete

except pyautogui.ImageNotFoundException:

print("Image not found.")

time.sleep(3)

check\_and\_click\_image(image\_path, region)

def check(image\_path, region):

print(f"Checking for image: {image\_path} in region: {region}")

try:

location = pyautogui.locateOnScreen(image\_path, confidence=0.7, region=region)

if location is not None:

return True

except pyautogui.ImageNotFoundException:

return False

def main():

program\_path = "F:/Choice/Choice.exe"

open\_program(program\_path)

click\_at(55, 505, "资讯")

click\_at(165, 175, "财经快讯")

click\_at(73, 428, "财经聚焦")

click\_at(1088, 138, "高级搜索")

click\_at(1299, 864, "搜索")

if startNum != 1:

check\_and\_click\_image("F:/SCSElearning/lab/Time-Series-stock/index.png", region=(1350, 901, 1490, 945))

pyautogui.press('right')

time.sleep(2)

pyautogui.press("backspace") # backspace

time.sleep(2)

type\_text(startNum)

time.sleep(3)

press\_enter()

time.sleep(4)

for number in range(startNum, endNum):

click\_at(1841, 231, "导出Excel")

time.sleep(2) # Extra wait for export dialog

type\_text(number)

time.sleep(3)

# click\_at(1056, 673, "保存Excel文件到电脑")

press\_enter()

# time.sleep(2) # Extra wait for save action

while check("F:/SCSElearning/lab/Time-Series-stock/existAlready.png", region=(728, 380, 1175, 555)):

click\_at(1114, 525, "否")

time.sleep(2) # Extra wait for export dialog

type\_text(number)

time.sleep(3)

press\_enter()

print("Closing the program.")

pyautogui.hotkey('alt', 'f4')

time.sleep(3)

check\_and\_click\_image("F:/SCSElearning/lab/Time-Series-stock/next.png",

region=(1140, 900, 1215, 950)) # region左上右下定位法

if \_\_name\_\_ == "\_\_main\_\_":

main()

#结构化行情数据处理

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from factor\_analyzer import calculate\_bartlett\_sphericity, calculate\_kmo

# 1. 加载数据

df = pd.read\_csv("C:\\Users\\sysws\\Desktop\\2015hs300.csv")

# 2. 数据清洗 - 缺失值填充（一次性完成前后填充）

df\_cleaned = df.ffill().bfill()

# 3. 特征筛选 - 生成滞后特征

lags = [1, 2, 3, 5] # 1天、2天、3天、5天滞后

features = ['closing price', 'volume', 'turnover']

for feature in features:

for lag in lags:

df\_cleaned[f'{feature}\_lag{lag}'] = df\_cleaned[feature].shift(lag)

# 删除因滞后特征产生的缺失值

df\_cleaned = df\_cleaned.dropna()

# 4. 准备分析数据

numeric\_cols = df\_cleaned.select\_dtypes(include=[np.number]).columns.tolist()

X = df\_cleaned[numeric\_cols]

# 标准化数据

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

X\_df = pd.DataFrame(X\_scaled, columns=numeric\_cols)

# 5. 删除高度相关的特征（减少矩阵接近奇异）

corr\_matrix = X\_df.corr().abs()

upper = corr\_matrix.where(np.triu(np.ones(corr\_matrix.shape), k=1).astype(bool))

to\_drop = [column for column in upper.columns if any(upper[column] > 0.95)]

if to\_drop:

print(f"删除高度相关的特征: {to\_drop}")

X\_df = X\_df.drop(columns=to\_drop)

# 6. 计算KMO值和Bartlett球形检验

kmo\_all, kmo\_model = calculate\_kmo(X\_df)

chi\_square\_value, p\_value = calculate\_bartlett\_sphericity(X\_df)

print("="\*50)

print("因子分析适用性检验:")

print(f"KMO统计量: {kmo\_model:.4f}")

print(f"Bartlett球形检验卡方值: {chi\_square\_value:.2f}")

print(f"Bartlett球形检验p值: {p\_value:.4e}")

# 7. 主成分分析

pca = PCA()

principal\_components = pca.fit\_transform(X\_df)

cumulative\_variance = np.cumsum(pca.explained\_variance\_ratio\_)

n\_components = np.argmax(cumulative\_variance >= 0.85) + 1

print("\n" + "="\*50)

print("主成分分析结果:")

print(f"总特征数量: {len(pca.explained\_variance\_ratio\_)}")

print(f"保留的主成分数量: {n\_components}")

print(f"累积解释方差: {cumulative\_variance[n\_components-1]:.4f}")

# 8. 使用选定主成分重新拟合

pca\_selected = PCA(n\_components=n\_components)

X\_pca = pca\_selected.fit\_transform(X\_df)

# 9. 保存结果

pca\_columns = [f'PC{i+1}' for i in range(n\_components)]

df\_pca = pd.DataFrame(X\_pca, columns=pca\_columns)

df\_pca['trade date'] = df\_cleaned['trade date'].values

print("\n" + "="\*50)

print("处理完成！前5行主成分数据:")

print(df\_pca.head())

df\_pca.to\_csv('processed\_hs300\_pca.csv', index=False)

#情感分析

import os

os.environ['HF\_HUB\_DISABLE\_SYMLINKS\_WARNING'] = '1' # 禁用符号链接警告

import torch

import pandas as pd

from transformers import pipeline, BertTokenizer

from tqdm import tqdm # 添加进度条支持

import logging

# 配置日志

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

logger = logging.getLogger(\_\_name\_\_)

# 检查GPU是否可用并配置设备

if torch.cuda.is\_available():

device = torch.device("cuda")

logger.info(f"使用GPU: {torch.cuda.get\_device\_name(0)}")

device\_id = 0 # GPU设备ID

else:

device = torch.device("cpu")

logger.warning("未检测到可用GPU，将使用CPU")

device\_id = -1 # CPU设备ID

def load\_model():

"""加载模型和分词器"""

try:

logger.info("正在加载模型...")

tokenizer = BertTokenizer.from\_pretrained("yiyanghkust/finbert-tone-chinese")

sentiment\_analyzer = pipeline(

"sentiment-analysis",

model="yiyanghkust/finbert-tone-chinese",

tokenizer=tokenizer,

device=device\_id, # 使用GPU（0）或CPU（-1）

top\_k=1, # 只返回得分最高的一个结果

batch\_size=32, # 批次大小

truncation=True, # 启用截断

padding=True # 启用填充

)

# 将模型移动到指定设备

sentiment\_analyzer.model.to(device)

logger.info("模型加载完成")

return sentiment\_analyzer

except Exception as e:

logger.error(f"加载模型失败: {e}")

raise

def analyze\_sentiment\_batch(texts, analyzer):

"""批量分析中文金融文本情感"""

results = []

confidences = []

# 根据设备设置批处理大小

batch\_size = 64 if torch.cuda.is\_available() else 16

logger.info(f"使用批处理大小: {batch\_size}")

for i in tqdm(range(0, len(texts), batch\_size), desc="情感分析进度"):

batch = texts[i:i+batch\_size]

try:

# 分析批处理数据

batch\_results = analyzer(batch)

for result in batch\_results:

# 处理每个文本的结果

if result and isinstance(result, list) and len(result) > 0:

top\_result = result[0] # 取最高分结果

results.append(top\_result['label'])

confidences.append(top\_result['score'])

else:

results.append('unknown')

confidences.append(0.0)

except Exception as e:

logger.error(f"分析第 {i} 到 {i+batch\_size} 条数据时出错: {e}")

# 为出错批次添加占位符

results.extend(['error'] \* min(batch\_size, len(texts) - i))

confidences.extend([0.0] \* min(batch\_size, len(texts) - i))

return results, confidences

def main():

# 加载模型

analyzer = load\_model()

try:

# 读取Excel文件

logger.info("正在读取Excel文件...")

data = pd.read\_excel(r"C:\Users\sysws\Desktop\shuju\merged\_news.xlsx")

logger.info(f"成功读取 {len(data)} 条数据")

# 提取标题列并转换为列表

texts = data["标题"].tolist()

# 情感分析

logger.info("开始情感分析...")

labels, confidences = analyze\_sentiment\_batch(texts, analyzer)

# 添加结果列

data['label'] = labels

data['conf'] = confidences

# 保存结果

output\_path = "C:\\Users\sysws\Desktop\shuju\qggqgqgqgqg.xlsx"

logger.info(f"正在保存结果到 {output\_path}")

data.to\_excel(output\_path, index=False, engine='openpyxl')

logger.info("结果保存成功")

except Exception as e:

logger.error(f"处理过程中出错: {e}")

# 可以在这里添加错误恢复或通知逻辑

if \_\_name\_\_ == "\_\_main\_\_":

main()

#情感处理

import pandas as pd

import numpy as np

from datetime import datetime, timedelta

import logging

# 配置日志

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler('alignment\_log.log'),

logging.StreamHandler()

]

)

def align\_sentiment\_with\_trading\_days(sentiment\_data, trading\_days, max\_lookahead=90):

"""

将情感数据与股票交易日对齐，非交易日数据累积到下一交易日

增强版：修复pandas.np错误，增加详细日志和错误处理

参数:

sentiment\_data: DataFrame，包含情感数据，需有'时间'和'sentiment\_score'列

trading\_days: 列表或DataFrame，包含所有交易日的日期，日期列应为'时间'

max\_lookahead: 寻找下一交易日的最大天数，默认90天

返回:

aligned\_data: DataFrame，对齐后的交易日情感数据

"""

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

# 1. 输入数据验证与预处理

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

try:

# 检查输入数据是否为空

if sentiment\_data.empty:

raise ValueError("情感数据为空，请检查输入数据")

if isinstance(trading\_days, list):

if not trading\_days:

raise ValueError("交易日列表为空，请检查输入数据")

trading\_days = pd.DataFrame({'时间': pd.to\_datetime(trading\_days)})

else:

if trading\_days.empty:

raise ValueError("交易日数据为空，请检查输入数据")

# 确保列名正确

required\_columns = ['时间', 'sentiment\_score']

if not all(col in sentiment\_data.columns for col in required\_columns):

missing = [col for col in required\_columns if col not in sentiment\_data.columns]

raise ValueError(f"情感数据缺少必要列: {missing}")

# 日期格式转换与验证

sentiment\_data = sentiment\_data.copy()

trading\_days = trading\_days.copy()

# 转换日期列并处理错误

sentiment\_data['时间'] = pd.to\_datetime(sentiment\_data['时间'], errors='coerce')

trading\_days['时间'] = pd.to\_datetime(trading\_days['时间'], errors='coerce')

# 检查日期转换错误

sentiment\_invalid\_dates = sentiment\_data[sentiment\_data['时间'].isna()]

if not sentiment\_invalid\_dates.empty:

logging.warning(f"情感数据中有 {len(sentiment\_invalid\_dates)} 条记录日期格式无效，已移除")

sentiment\_data = sentiment\_data.dropna(subset=['时间'])

trading\_invalid\_dates = trading\_days[trading\_days['时间'].isna()]

if not trading\_invalid\_dates.empty:

logging.warning(f"交易日数据中有 {len(trading\_invalid\_dates)} 条记录日期格式无效，已移除")

trading\_days = trading\_days.dropna(subset=['时间'])

# 再次检查清洗后的数据是否为空

if sentiment\_data.empty:

raise ValueError("情感数据在清洗无效日期后为空")

if trading\_days.empty:

raise ValueError("交易日数据在清洗无效日期后为空")

# 打印日期范围信息

min\_sent\_date = sentiment\_data['时间'].min()

max\_sent\_date = sentiment\_data['时间'].max()

min\_trade\_date = trading\_days['时间'].min()

max\_trade\_date = trading\_days['时间'].max()

logging.info(f"情感数据日期范围: {min\_sent\_date.strftime('%Y-%m-%d')} 至 {max\_sent\_date.strftime('%Y-%m-%d')}")

logging.info(f"交易日数据日期范围: {min\_trade\_date.strftime('%Y-%m-%d')} 至 {max\_trade\_date.strftime('%Y-%m-%d')}")

# 检查日期范围是否重叠

if max\_sent\_date < min\_trade\_date or min\_sent\_date > max\_trade\_date:

raise ValueError("情感数据与交易日数据的日期范围没有重叠")

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

# 2. 数据处理

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

# 排序交易日并创建集合用于快速查找

trading\_days = trading\_days.sort\_values('时间').reset\_index(drop=True)

trading\_dates = set(trading\_days['时间'].dt.date) # 使用date对象便于比较

# 按日期分组计算基础指标

daily\_sentiment = sentiment\_data.groupby('时间').agg(

total\_score=('sentiment\_score', 'sum'),

news\_count=('sentiment\_score', 'count')

).reset\_index()

# 提取日期部分用于比较

daily\_sentiment['date\_only'] = daily\_sentiment['时间'].dt.date

logging.info(f"共处理 {len(daily\_sentiment)} 天的情感数据")

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

# 3. 对齐逻辑

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

aligned\_results = []

processed\_dates = set()

for \_, row in daily\_sentiment.iterrows():

current\_date = row['date\_only']

current\_datetime = row['时间']

if current\_date in processed\_dates:

continue

# 检查是否为交易日

if current\_date in trading\_dates:

aligned\_results.append({

'时间': current\_datetime,

'total\_score': row['total\_score'],

'news\_count': row['news\_count'],

'original\_dates': [current\_date]

})

processed\_dates.add(current\_date)

logging.debug(f"已处理交易日: {current\_date}")

else:

# 寻找下一交易日

next\_trading\_date = None

days\_ahead = 1

while next\_trading\_date is None and days\_ahead <= max\_lookahead:

check\_date = current\_date + timedelta(days=days\_ahead)

if check\_date in trading\_dates:

next\_trading\_date = check\_date

else:

days\_ahead += 1

if next\_trading\_date:

# 累积连续非交易日数据

total\_score = row['total\_score']

news\_count = row['news\_count']

original\_dates = [current\_date]

processed\_dates.add(current\_date)

# 检查后续非交易日

check\_date = current\_date + timedelta(days=1)

while check\_date < next\_trading\_date:

if check\_date in daily\_sentiment['date\_only'].values:

temp = daily\_sentiment[daily\_sentiment['date\_only'] == check\_date].iloc[0]

total\_score += temp['total\_score']

news\_count += temp['news\_count']

original\_dates.append(check\_date)

processed\_dates.add(check\_date)

check\_date += timedelta(days=1)

# 找到对应的datetime

next\_trading\_datetime = trading\_days[trading\_days['时间'].dt.date == next\_trading\_date]['时间'].iloc[0]

aligned\_results.append({

'时间': next\_trading\_datetime,

'total\_score': total\_score,

'news\_count': news\_count,

'original\_dates': original\_dates

})

logging.debug(f"已处理非交易日，累积到 {next\_trading\_date}，包含 {len(original\_dates)} 天数据")

else:

logging.warning(f"无法为 {current\_date} 找到{max\_lookahead}天内的下一交易日，数据被跳过")

# 处理无结果的情况

if not aligned\_results:

logging.warning("没有生成任何对齐结果，可能是因为所有数据都无法找到对应的交易日")

return pd.DataFrame(columns=['时间', 'total\_score', 'news\_count', 'original\_dates'])

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

# 4. 结果处理与扩展

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

aligned\_df = pd.DataFrame(aligned\_results)

aligned\_df = aligned\_df.sort\_values('时间').reset\_index(drop=True)

# 合并重复的交易日

aligned\_df = aligned\_df.groupby('时间').agg(

total\_score=('total\_score', 'sum'),

news\_count=('news\_count', 'sum'),

original\_dates=('original\_dates', lambda x: [date for sublist in x for date in sublist])

).reset\_index()

# 计算额外指标（根据需要扩展）

aligned\_df['avg\_score'] = aligned\_df['total\_score'] / aligned\_df['news\_count'].replace(0, 1)

# 添加滞后指标

for lag in [1, 2, 3]:

aligned\_df[f'lag\_{lag}\_total\_score'] = aligned\_df['total\_score'].shift(lag)

aligned\_df[f'lag\_{lag}\_avg\_score'] = aligned\_df['avg\_score'].shift(lag)

# 填充可能的缺失值

aligned\_df = aligned\_df.fillna(0)

logging.info(f"对齐完成，共生成 {len(aligned\_df)} 条交易日数据")

return aligned\_df

except Exception as e:

logging.error(f"处理过程中发生错误: {str(e)}", exc\_info=True)

# 返回包含所有预期列的空DataFrame，便于后续处理

columns = [

'时间', 'total\_score', 'news\_count', 'original\_dates', 'avg\_score',

'lag\_1\_total\_score', 'lag\_1\_avg\_score',

'lag\_2\_total\_score', 'lag\_2\_avg\_score',

'lag\_3\_total\_score', 'lag\_3\_avg\_score'

]

return pd.DataFrame(columns=columns)

# 使用示例

if \_\_name\_\_ == "\_\_main\_\_":

# 创建示例数据

try:

# 生成示例情感数据

dates = pd.date\_range(start='2023-01-01', end='2023-01-10')

data = []

for date in dates:

# 随机生成1-10条新闻（使用numpy代替pandas.np）

num\_news = np.random.randint(1, 11)

for \_ in range(num\_news):

# 随机情感得分（使用numpy代替pandas.np）

score = np.random.uniform(-1, 1)

data.append({

'时间': date,

'sentiment\_score': score

})

sentiment\_df = pd.DataFrame(data)

# 定义交易日（排除周末）

trading\_days = []

for date in pd.date\_range(start='2023-01-01', end='2023-01-15'):

# 只包含周一到周五

if date.weekday() < 5:

trading\_days.append(date)

trading\_df = pd.DataFrame({'时间': trading\_days})

# 执行对齐

result = align\_sentiment\_with\_trading\_days(sentiment\_df, trading\_df)

if not result.empty:

print("对齐结果预览:")

print(result[['时间', 'total\_score', 'news\_count', 'avg\_score']].head())

# 保存结果

result.to\_excel('enhanced\_aligned\_data.xlsx', index=False)

print("结果已保存到 'enhanced\_aligned\_data.xlsx'")

else:

print("未生成任何对齐结果，请查看日志了解详情")

except Exception as e:

print(f"示例运行失败: {str(e)}")

#决策树代码

import pandas as pd

import numpy as np

from sklearn.model\_selection import GridSearchCV, TimeSeriesSplit

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report, roc\_auc\_score

from sklearn.preprocessing import StandardScaler

from imblearn.over\_sampling import SMOTE

import warnings

warnings.filterwarnings('ignore')

class AdvancedStockPredictor:

"""决策树股指趋势预测器（含数据清洗、特征工程、时间序列验证、SMOTE）"""

def \_\_init\_\_(self):

self.data = None

self.X\_train = None

self.X\_test = None

self.y\_train = None

self.y\_test = None

self.model = None

self.scaler = StandardScaler()

self.best\_params = None

self.feature\_names = []

self.core\_cols = {

'close': 'closing price',

'open': 'at the opening',

'high': 'top price',

'low': 'floor price',

'change': 'price change',

'volume': 'volume',

'sentiment': '最终情感得分',

'date': '原始日期时间'

}

def \_handle\_extreme\_values(self, df, low\_q=0.01, high\_q=0.99):

"""处理无穷值和极端值"""

df = df.replace([np.inf, -np.inf], np.nan)

numeric\_cols = df.select\_dtypes(include=['float64', 'int64']).columns

for col in numeric\_cols:

if col in ['target', 'target\_raw', 'next\_day\_change']:

continue

upper = df[col].quantile(high\_q)

lower = df[col].quantile(low\_q)

df[col] = np.clip(df[col], lower, upper)

df[col] = df[col].fillna(df[col].median())

return df

def load\_data(self, file\_path):

"""加载数据并按日期排序"""

try:

df = pd.read\_excel(file\_path)

df[self.core\_cols['date']] = pd.to\_datetime(df[self.core\_cols['date']])

df.sort\_values(by=self.core\_cols['date'], inplace=True)

self.data = df

print(f" 数据加载成功，时间范围: {df[self.core\_cols['date']].min()} 至 {df[self.core\_cols['date']].max()}")

return True

except Exception as e:

print(f" 数据加载失败: {e}")

return False

def create\_target(self, threshold=0.005):

"""生成二分类目标变量，过滤横盘"""

self.data['next\_day\_change'] = self.data[self.core\_cols['change']].shift(-1)

self.data['target\_raw'] = np.where(

self.data['next\_day\_change'] > threshold, 1,

np.where(self.data['next\_day\_change'] < -threshold, 0, -1)

)

self.data = self.data[self.data['target\_raw'] != -1].copy()

self.data['target'] = self.data['target\_raw'].astype(int)

print(f"目标变量生成: 上涨={sum(self.data['target']==1)}, 下跌={sum(self.data['target']==0)}")

def advanced\_feature\_engineering(self):

"""生成特征集并清洗"""

df = self.data.copy()

c = self.core\_cols

# 核心价格特征

df['price\_range\_ratio'] = (df[c['high']] - df[c['low']]) / df[c['close']]

df['close\_open\_ratio'] = df[c['close']] / df[c['open']]

df['change\_abs'] = abs(df[c['change']])

# 短期趋势

for w in [3, 5]:

df[f'close\_ma\_{w}'] = df[c['close']].rolling(w).mean()

df[f'change\_ma\_{w}'] = df[c['change']].rolling(w).mean()

# 情感特征

df['sentiment\_norm'] = (df[c['sentiment']] - df[c['sentiment']].mean()) / (df[c['sentiment']].std() + 1e-8)

df['sentiment\_change'] = df[c['sentiment']].pct\_change().replace([np.inf, -np.inf], 0).fillna(0)

df['sentiment\_price\_interact'] = df['sentiment\_norm'] \* df[c['change']]

# 滞后特征

df['lag1\_close'] = df[c['close']].shift(1)

df['lag1\_change'] = df[c['change']].shift(1)

df['lag1\_sentiment'] = df[c['sentiment']].shift(1)

# 成交量特征

df['volume\_log'] = np.log1p(df[c['volume']])

df['volume\_change'] = df[c['volume']].pct\_change().replace([np.inf, -np.inf], 0).fillna(0)

# 最终保留列

keep\_cols = [c['close'], c['change'], c['volume'], c['sentiment'],

'price\_range\_ratio', 'close\_open\_ratio', 'change\_abs',

'close\_ma\_3', 'change\_ma\_3', 'close\_ma\_5',

'sentiment\_norm', 'sentiment\_change', 'sentiment\_price\_interact',

'lag1\_close', 'lag1\_change', 'lag1\_sentiment',

'volume\_log', 'volume\_change', 'target', c['date']]

df = df[keep\_cols].dropna().copy()

df = self.\_handle\_extreme\_values(df)

self.data = df

self.feature\_names = [col for col in keep\_cols if col not in ['target', c['date']]]

print(f" 特征工程完成，共 {len(self.feature\_names)} 个特征")

def time\_series\_split(self, test\_ratio=0.2):

"""按时间顺序分割数据并标准化"""

train\_size = int(len(self.data) \* (1 - test\_ratio))

train, test = self.data.iloc[:train\_size], self.data.iloc[train\_size:]

X\_train, y\_train = train[self.feature\_names], train['target']

X\_test, y\_test = test[self.feature\_names], test['target']

X\_train = self.\_handle\_extreme\_values(X\_train)

X\_test = self.\_handle\_extreme\_values(X\_test)

smote = SMOTE(random\_state=42)

X\_train\_res, y\_train\_res = smote.fit\_resample(X\_train, y\_train)

self.X\_train = self.scaler.fit\_transform(X\_train\_res)

self.X\_test = self.scaler.transform(X\_test)

self.y\_train, self.y\_test = y\_train\_res, y\_test

print(f"数据分割完成, 训练集={len(self.X\_train)}, 测试集={len(self.X\_test)}")

print(f" 训练集类别分布: {np.bincount(self.y\_train)}, 测试集: {np.bincount(self.y\_test)}")

def optimize\_tree(self):

"""使用网格搜索优化决策树"""

if len(self.X\_train) < 50:

print(" 数据量过少，跳过参数优化")

self.model = DecisionTreeClassifier(random\_state=42)

self.model.fit(self.X\_train, self.y\_train)

return

param\_grid = {

'criterion': ['entropy'],

'max\_depth': [6, 7, 8],

'min\_samples\_split': [4, 5],

'min\_samples\_leaf': [2, 3],

'max\_features': ['sqrt'],

'class\_weight': ['balanced']

}

tscv = TimeSeriesSplit(n\_splits=5)

self.model = GridSearchCV(

DecisionTreeClassifier(random\_state=42),

param\_grid, cv=tscv, n\_jobs=-1, verbose=0

)

self.model.fit(self.X\_train, self.y\_train)

self.best\_params = self.model.best\_params\_

print(f" 最佳参数: {self.best\_params}, CV准确率: {self.model.best\_score\_:.4f}")

def evaluate\_performance(self):

"""输出模型在训练集和测试集的表现"""

y\_pred\_train = self.model.predict(self.X\_train)

y\_pred\_test = self.model.predict(self.X\_test)

print("\n===== 模型评估 =====")

print(f"训练集准确率: {accuracy\_score(self.y\_train, y\_pred\_train):.4f}")

print(f"测试集准确率: {accuracy\_score(self.y\_test, y\_pred\_test):.4f}")

if len(set(self.y\_test)) == 2:

print(f"测试集ROC-AUC: {roc\_auc\_score(self.y\_test, y\_pred\_test):.4f}")

cm = confusion\_matrix(self.y\_test, y\_pred\_test)

print("\n混淆矩阵:\n", cm)

print("\n分类报告:\n", classification\_report(self.y\_test, y\_pred\_test))

importances = self.model.best\_estimator\_.feature\_importances\_ \

if hasattr(self.model, 'best\_estimator\_') else self.model.feature\_importances\_

fi\_df = pd.DataFrame({

'Feature': self.feature\_names,

'Importance': importances

}).sort\_values('Importance', ascending=False)

print("\n重要特征 Top 10:\n", fi\_df.head(10))

def run\_optimized\_pipeline(self, file\_path):

print("\n 启动优化后的股指预测流程")

if not self.load\_data(file\_path):

return

self.create\_target(threshold=0.005)

self.advanced\_feature\_engineering()

self.time\_series\_split()

self.optimize\_tree()

self.evaluate\_performance()

if \_\_name\_\_ == "\_\_main\_\_":

predictor = AdvancedStockPredictor()

data\_path = r"C:\Users\sysws\Desktop\shuju\awax.xlsx"

predictor.run\_optimized\_pipeline(data\_path)