#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()

爬取文本合并

import pandas as pd

import os

from datetime import datetime

folder\_path = r"F:\wenben"

output\_file = r"F:\wenben\11111"

def merge\_excel\_files(folder\_path, output\_file):

# 创建空的DataFrame存储合并后的数据

merged\_data = pd.DataFrame()

# 遍历文件夹中的所有文件

for filename in os.listdir(folder\_path):

if filename.endswith(('.xls', '.xlsx')):

file\_path = os.path.join(folder\_path, filename)

try:

# 读取Excel文件（假设所有文件结构相同）

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

# 确保包含时间列（检查不同可能的列名）

time\_col = None

for col in ['时间', '日期', 'Date', 'date']:

if col in df.columns:

time\_col = col

break

if time\_col is None:

print(f"警告: 文件 {filename} 未找到时间列，跳过")

continue

# 转换时间列为datetime类型

df[time\_col] = pd.to\_datetime(df[time\_col], errors='coerce')

# 添加到合并数据

merged\_data = pd.concat([merged\_data, df], ignore\_index=True)

print(f"已加载: {filename}")

except Exception as e:

print(f"处理文件 {filename} 时出错: {str(e)}")

if merged\_data.empty:

print("未找到有效数据")

return

# 按时间排序

merged\_data = merged\_data.sort\_values(by=time\_col)

# 重置索引

merged\_data.reset\_index(drop=True, inplace=True)

# 保存合并后的文件

merged\_data.to\_excel(output\_file, index=False)

print(f"合并完成! 共 {len(merged\_data)} 条记录已保存至: {output\_file}")

# 使用示例

folder\_path = r"F:\wenben" # 替换为你的文件夹路径

output\_file = r"F:\wenben\merged\_news.xlsx" # 输出文件路径

merge\_excel\_files(folder\_path, output\_file)

#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)

LSTM

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, r2\_score, mean\_absolute\_percentage\_error

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Input, LSTM, Dense, Dropout, BatchNormalization

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

import tensorflow as tf

import os

import datetime

import matplotlib.font\_manager as fm

from platform import system

# 设置中文字体 - 增强版，优先选择支持更多符号的字体

def setup\_chinese\_fonts():

"""配置Matplotlib以正确显示中文字符和特殊符号"""

system\_name = system()

# 调整字体优先级，优先选择支持更多符号的字体

font\_paths = []

if system\_name == "Windows":

font\_paths = [

'C:\\Windows\\Fonts\\microsoftyahei.ttf', # 微软雅黑，对特殊符号支持较好

'C:\\Windows\\Fonts\\msyh.ttc', # 微软雅黑

'C:\\Windows\\Fonts\\simhei.ttf', # 黑体

'C:\\Windows\\Fonts\\simsun.ttc', # 宋体

'C:\\Windows\\Fonts\\simkai.ttf', # 楷体

'C:\\Windows\\Fonts\\Arial Unicode MS.ttf' # 支持多语言和符号

]

elif system\_name == "Darwin": # macOS

font\_paths = [

'/System/Library/Fonts/PingFang.ttc', # 苹方

'/Library/Fonts/Microsoft YaHei.ttf', # 微软雅黑

'/Library/Fonts/SimHei.ttf', # 黑体

'/Library/Fonts/Arial Unicode.ttf' # 支持多语言和符号

]

elif system\_name == "Linux": # Linux

font\_paths = [

'/usr/share/fonts/opentype/noto/NotoSansCJK-Regular.ttc',

'/usr/share/fonts/truetype/wqy/wqy-microhei.ttc',

'/usr/share/fonts/truetype/wqy/wqy-zenhei.ttc',

'/usr/share/fonts/truetype/freefont/FreeSans.ttf'

]

# 尝试加载系统字体

for font\_path in font\_paths:

if os.path.exists(font\_path):

try:

font\_prop = fm.FontProperties(fname=font\_path)

font\_name = font\_prop.get\_name()

plt.rcParams["font.family"] = font\_name

plt.rcParams['axes.unicode\_minus'] = False # 解决负号显示问题

sns.set(font=font\_name)

print(f"成功加载中文字体: {font\_name}")

return True

except Exception as e:

print(f"加载字体 {font\_path} 出错: {str(e)}")

continue

# 尝试使用已安装的中文字体

installed\_fonts = [f.name for f in fm.findSystemFonts()]

# 优先选择支持更多符号的字体

chinese\_fonts = ['Microsoft YaHei', 'Arial Unicode MS', 'PingFang',

'SimHei', 'WenQuanYi Micro Hei', 'Heiti TC']

for font in chinese\_fonts:

if font in installed\_fonts:

plt.rcParams["font.family"] = font

plt.rcParams['axes.unicode\_minus'] = False

sns.set(font=font)

print(f"使用已安装的中文字体: {font}")

return True

# 最终备选方案：使用支持Unicode的字体

plt.rcParams["font.family"] = ["Arial Unicode MS", "DejaVu Sans", "sans-serif"]

plt.rcParams['axes.unicode\_minus'] = False

print("使用默认备选字体方案，确保特殊符号正常显示")

return False

# 替换文本中的特殊字符和上标

def replace\_special\_chars(text):

"""替换可能引起字体问题的特殊字符和上标"""

# 替换Unicode减号为普通减号

text = text.replace('\u2212', '-')

# 替换上标2（²）为"平方"或"^2"

text = text.replace('²', '平方') # 对于MSE等指标，用"平方"替代上标2

text = text.replace('R²', 'R方') # 专门处理R²，改为"R方"

return text

setup\_chinese\_fonts()

class StockPricePredictor:

def \_\_init\_\_(self, time\_steps=30, epochs=200, batch\_size=32, close\_price\_col='closing\_price'):

"""初始化预测器参数"""

self.time\_steps = time\_steps

self.epochs = epochs

self.batch\_size = batch\_size

self.scaler = MinMaxScaler(feature\_range=(0, 1))

self.target\_scaler = MinMaxScaler(feature\_range=(0, 1)) # 单独为目标变量设置缩放器

self.model = None

self.data = None

self.features = None

self.target = None

self.close\_price\_col = close\_price\_col

self.X\_train = None

self.X\_test = None

self.y\_train = None

self.y\_test = None

self.train\_predictions = None

self.test\_predictions = None

self.history = None

self.test\_metrics = None

self.train\_metrics = None

def load\_data(self, file\_path, file\_type='csv'):

"""加载数据并添加特征工程步骤"""

if file\_path is None:

# 生成示例数据

print("使用示例数据进行演示...")

dates = pd.date\_range(start='2015-07-01', periods=1000, freq='B')

np.random.seed(42)

base\_trend = np.linspace(3000, 5000, 1000)

cycle = 50 \* np.sin(np.linspace(0, 40\*np.pi, 1000))

noise = np.random.normal(0, 30, 1000)

closing\_prices = base\_trend + cycle + noise

data = pd.DataFrame({

'date': dates,

self.close\_price\_col: closing\_prices,

'opening\_price': closing\_prices + np.random.normal(0, 10, 1000),

'high\_price': closing\_prices + np.random.normal(15, 5, 1000),

'low\_price': closing\_prices - np.random.normal(15, 5, 1000),

'volume': np.random.randint(200000000, 700000000, 1000),

})

else:

try:

if file\_type.lower() == 'excel' or file\_path.endswith('.xlsx'):

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

else:

data = pd.read\_csv(file\_path)

print("\n数据列名:", data.columns.tolist())

# 日期处理

date\_columns = ['date', 'datetime', '日期', '交易日期', '原始日期时间', '情感数据日期']

found\_date\_col = next((col for col in date\_columns if col in data.columns), None)

if found\_date\_col:

data['date'] = pd.to\_datetime(data[found\_date\_col])

if found\_date\_col != 'date':

data = data.drop(found\_date\_col, axis=1)

else:

print("警告：未找到日期列，创建默认日期序列")

data['date'] = pd.date\_range(start='2015-07-01', periods=len(data), freq='B')

# 收盘价列处理

if self.close\_price\_col not in data.columns:

possible\_close = ['closing price', '收盘价', 'close', '收盘']

matched = next((col for col in possible\_close if col in data.columns), None)

if matched:

self.close\_price\_col = matched

print(f"自动匹配收盘价列: {matched}")

else:

raise ValueError(f"未找到收盘价列，可用列: {data.columns.tolist()}")

data = data.sort\_values('date')

data = data.drop\_duplicates(subset=['date']) # 去重

data = data.bfill().ffill() # 缺失值处理

print(f"数据加载完成，共 {len(data)} 条记录")

except Exception as e:

raise e

# 添加技术指标作为新特征

data = self.\_add\_technical\_indicators(data)

self.data = data

return data

def \_add\_technical\_indicators(self, data):

"""添加技术指标特征"""

# 移动平均线

data['ma5'] = data[self.close\_price\_col].rolling(window=5).mean()

data['ma20'] = data[self.close\_price\_col].rolling(window=20).mean()

data['ma60'] = data[self.close\_price\_col].rolling(window=60).mean()

# 动量指标

data['rsi'] = self.\_calculate\_rsi(data[self.close\_price\_col], window=14)

# 波动率指标

data['volatility'] = data[self.close\_price\_col].rolling(window=10).std()

# 量价关系

if 'volume' in data.columns:

data['price\_volume'] = data[self.close\_price\_col] \* data['volume']

else:

data['price\_volume'] = data[self.close\_price\_col] # 没有成交量数据时的备选方案

# 填充计算指标产生的NaN

return data.bfill().ffill()

def \_calculate\_rsi(self, prices, window=14):

"""计算相对强弱指数(RSI)"""

delta = prices.diff()

gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()

loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()

rs = gain / loss

return 100 - (100 / (1 + rs))

def preprocess\_data(self, feature\_columns=None):

"""改进的数据预处理，过滤非数值特征"""

if self.data is None:

raise ValueError("请先加载数据")

# 显示目标变量的基本统计

print(f"\n{self.close\_price\_col} 统计信息:")

print(self.data[self.close\_price\_col].describe())

# 自动选择相关特征（排除日期、目标变量和非数值列）

if feature\_columns is None:

exclude\_cols = ['date', self.close\_price\_col]

# 先筛选出候选特征列

candidate\_cols = [col for col in self.data.columns if col not in exclude\_cols]

# 过滤掉非数值型列

numeric\_cols = []

string\_cols = []

for col in candidate\_cols:

if pd.api.types.is\_numeric\_dtype(self.data[col]):

numeric\_cols.append(col)

else:

string\_cols.append(col)

# 输出过滤信息

if string\_cols:

print(f"已过滤非数值特征: {string\_cols}")

feature\_columns = numeric\_cols

print(f"使用的数值特征列: {feature\_columns}")

# 检查是否有可用特征

if not feature\_columns:

raise ValueError("没有可用的数值特征列，请检查数据")

# 提取特征和目标变量

self.features = self.data[feature\_columns].values

self.target = self.data[self.close\_price\_col].values.reshape(-1, 1)

# 分别缩放特征和目标变量

scaled\_features = self.scaler.fit\_transform(self.features)

scaled\_target = self.target\_scaler.fit\_transform(self.target)

# 创建序列数据

X, y = [], []

for i in range(self.time\_steps, len(scaled\_features)):

X.append(scaled\_features[i-self.time\_steps:i, :]) # 过去time\_steps天的特征

y.append(scaled\_target[i, 0]) # 第i天的目标值（已缩放）

X, y = np.array(X), np.array(y)

if len(X) == 0:

raise ValueError(f"序列数据为空，时间步长({self.time\_steps})大于数据长度({len(self.data)})")

# 划分训练集和测试集

train\_size = int(len(X) \* 0.8)

self.X\_train, self.X\_test = X[:train\_size], X[train\_size:]

self.y\_train, self.y\_test = y[:train\_size], y[train\_size:]

print(f"训练集大小: {self.X\_train.shape}, 测试集大小: {self.X\_test.shape}")

return self.X\_train, self.X\_test, self.y\_train, self.y\_test

def build\_model(self, input\_shape):

"""构建模型结构"""

model = Sequential([

Input(shape=input\_shape),

LSTM(128, return\_sequences=True, kernel\_regularizer='l2', recurrent\_dropout=0.1),

BatchNormalization(),

LSTM(64, return\_sequences=False, kernel\_regularizer='l2', recurrent\_dropout=0.1),

BatchNormalization(),

Dense(32, activation='relu'),

Dropout(0.2),

Dense(1)

])

optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0005)

model.compile(

optimizer=optimizer,

loss='mean\_squared\_error',

metrics=['mae']

)

self.model = model

return model

def train\_model(self, X\_train, y\_train, X\_test, y\_test, save\_path='best\_model.keras'):

"""训练模型"""

if self.model is None:

input\_shape = (X\_train.shape[1], X\_train.shape[2])

self.build\_model(input\_shape)

# 早停机制

early\_stop = EarlyStopping(

monitor='val\_loss',

patience=15,

restore\_best\_weights=True

)

# 学习率调度器

lr\_scheduler = ReduceLROnPlateau(

monitor='val\_loss',

factor=0.7,

patience=5,

min\_lr=1e-6

)

checkpoint = ModelCheckpoint(

save\_path,

monitor='val\_loss',

save\_best\_only=True,

mode='min'

)

# 训练模型

self.history = self.model.fit(

X\_train, y\_train,

batch\_size=self.batch\_size,

epochs=self.epochs,

validation\_data=(X\_test, y\_test),

callbacks=[early\_stop, checkpoint, lr\_scheduler],

verbose=1

)

# 预测后反归一化

self.train\_predictions = self.target\_scaler.inverse\_transform(

self.model.predict(X\_train)

).flatten()

self.test\_predictions = self.target\_scaler.inverse\_transform(

self.model.predict(X\_test)

).flatten()

return self.history

def predict\_next\_day(self, last\_days\_data=None):

if self.model is None:

raise ValueError("请先构建和训练模型")

if last\_days\_data is None:

last\_days\_data = self.features[-self.time\_steps:]

if len(last\_days\_data) != self.time\_steps:

raise ValueError(f"需要提供 {self.time\_steps} 天的数据")

scaled\_data = self.scaler.transform(last\_days\_data)

input\_data = scaled\_data.reshape(1, self.time\_steps, -1)

# 预测后反归一化

next\_day\_pred\_scaled = self.model.predict(input\_data)

next\_day\_pred = self.target\_scaler.inverse\_transform(next\_day\_pred\_scaled)[0][0]

print(f"\n下一个交易日的预测收盘价: {next\_day\_pred:.2f}")

if hasattr(self, 'test\_predictions') and hasattr(self, 'y\_test'):

actual\_test = self.target\_scaler.inverse\_transform(self.y\_test.reshape(-1, 1)).flatten()

recent\_errors = np.abs(self.test\_predictions[-10:] - actual\_test[-10:])

avg\_error = np.mean(recent\_errors)

print(f"预测误差范围估计: ±{avg\_error:.2f}")

return next\_day\_pred

def plot\_predictions(self):

"""优化的预测可视化，合并训练集和测试集展示，并添加预测区间"""

if self.model is None or self.train\_predictions is None:

raise ValueError("请先训练模型")

# 反归一化实际值

actual\_train = self.target\_scaler.inverse\_transform(self.y\_train.reshape(-1, 1)).flatten()

actual\_test = self.target\_scaler.inverse\_transform(self.y\_test.reshape(-1, 1)).flatten()

# 获取日期范围

train\_dates = self.data['date'][self.time\_steps : self.time\_steps + len(actual\_train)]

test\_dates = self.data['date'][self.time\_steps + len(actual\_train) : self.time\_steps + len(actual\_train) + len(actual\_test)]

# 计算预测误差范围

train\_errors = np.abs(self.train\_predictions - actual\_train)

test\_errors = np.abs(self.test\_predictions - actual\_test)

avg\_train\_error = np.mean(train\_errors)

avg\_test\_error = np.mean(test\_errors)

plt.figure(figsize=(16, 10))

# 合并展示训练集和测试集预测结果

plt.plot(train\_dates, actual\_train, label='实际值（训练）', color='blue')

plt.plot(train\_dates, self.train\_predictions, label='预测值（训练）', color='red', alpha=0.7)

# 添加训练集预测区间

plt.fill\_between(train\_dates,

self.train\_predictions - avg\_train\_error,

self.train\_predictions + avg\_train\_error,

color='red', alpha=0.2,

label=replace\_special\_chars(f'训练预测区间 (±{avg\_train\_error:.2f})'))

plt.plot(test\_dates, actual\_test, label='实际值（测试）', color='green')

plt.plot(test\_dates, self.test\_predictions, label='预测值（测试）', color='orange', alpha=0.7)

# 添加测试集预测区间

plt.fill\_between(test\_dates,

self.test\_predictions - avg\_test\_error,

self.test\_predictions + avg\_test\_error,

color='orange', alpha=0.2,

label=replace\_special\_chars(f'测试预测区间 (±{avg\_test\_error:.2f})'))

# 添加垂直线区分训练集和测试集

split\_date = train\_dates.iloc[-1]

plt.axvline(x=split\_date, color='gray', linestyle='--', label='训练/测试分割点')

plt.title('股票价格预测 vs 实际值', fontsize=14)

plt.xlabel('日期', fontsize=12)

plt.ylabel('价格', fontsize=12)

plt.legend(loc='best')

plt.grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

def calculate\_accuracy\_metrics(self, actual, predicted):

"""计算准确度指标"""

actual = self.target\_scaler.inverse\_transform(actual.reshape(-1, 1)).flatten()

mse = mean\_squared\_error(actual, predicted)

rmse = np.sqrt(mse)

mae = mean\_absolute\_error(actual, predicted)

r2 = r2\_score(actual, predicted)

mape = mean\_absolute\_percentage\_error(actual, predicted) \* 100

return {

'mse': mse,

'rmse': rmse,

'mae': mae,

'r2': r2,

'mape': mape

}

def evaluate\_model(self, X\_test, y\_test, X\_train=None, y\_train=None):

"""评估模型性能并可视化关键指标"""

self.test\_metrics = self.calculate\_accuracy\_metrics(y\_test, self.test\_predictions)

print("\n===== 模型评估 =====")

print(f"测试集 R方 分数: {self.test\_metrics['r2']:.4f}") # 将R²改为R方

print(f"测试集 MAPE: {self.test\_metrics['mape']:.2f}%")

print(f"测试集 RMSE: {self.test\_metrics['rmse']:.2f}")

print(f"测试集 MAE: {self.test\_metrics['mae']:.2f}")

print(f"测试集 MSE（均方误差）: {self.test\_metrics['mse']:.2f}") # 明确写出均方误差

if X\_train is not None and y\_train is not None:

self.train\_metrics = self.calculate\_accuracy\_metrics(y\_train, self.train\_predictions)

print(f"\n训练集 R方 分数: {self.train\_metrics['r2']:.4f}") # 将R²改为R方

print(f"训练集 MAPE: {self.train\_metrics['mape']:.2f}%")

if self.train\_metrics['r2'] - self.test\_metrics['r2'] > 0.2:

print("警告: 可能存在过拟合")

# 可视化关键评估指标

self.plot\_metrics\_comparison()

return self.test\_metrics

def plot\_metrics\_comparison(self):

"""可视化展示关键评估指标的对比"""

if self.test\_metrics is None:

raise ValueError("请先评估模型")

# 准备绘图数据，替换特殊字符和上标

metrics\_names = [

replace\_special\_chars('RMSE（均方根误差）'),

replace\_special\_chars('MAE（平均绝对误差）'),

replace\_special\_chars('MAPE（平均绝对百分比误差）'),

replace\_special\_chars('R方（决定系数）') # 将R²改为R方

]

metrics\_keys = ['rmse', 'mae', 'mape', 'r2']

# 测试集指标

test\_values = [self.test\_metrics[key] for key in metrics\_keys]

# 训练集指标（如果有）

train\_values = None

if self.train\_metrics is not None:

train\_values = [self.train\_metrics[key] for key in metrics\_keys]

# 创建图形

x = np.arange(len(metrics\_names)) # 标签位置

width = 0.35 # 柱状图宽度

fig, ax = plt.subplots(figsize=(12, 6))

# 绘制柱状图

rects\_test = ax.bar(x - width/2, test\_values, width, label='测试集', color='#3498db')

if train\_values is not None:

rects\_train = ax.bar(x + width/2, train\_values, width, label='训练集', color='#e74c3c')

# 添加标签、标题和自定义x轴刻度

ax.set\_ylabel('指标值', fontsize=12)

ax.set\_title('模型性能评估指标', fontsize=14)

ax.set\_xticks(x)

ax.set\_xticklabels(metrics\_names, rotation=15, ha='right')

ax.legend()

# 在柱状图上添加数值标签

def autolabel(rects):

"""为柱状图添加数值标签"""

for rect in rects:

height = rect.get\_height()

ax.annotate(f'{height:.4f}',

xy=(rect.get\_x() + rect.get\_width() / 2, height),

xytext=(0, 3), # 3点垂直偏移

textcoords="offset points",

ha='center', va='bottom', fontsize=10)

autolabel(rects\_test)

if train\_values is not None:

autolabel(rects\_train)

# 添加指标解释，替换特殊字符

metrics\_explanations = [

replace\_special\_chars("衡量预测值与实际值的平均偏差，值越小越好"),

replace\_special\_chars("反映预测误差的绝对值均值，对异常值更稳健"),

replace\_special\_chars("以百分比形式展示误差，便于跨尺度比较"),

replace\_special\_chars("越接近1，说明模型对数据的解释能力越强")

]

# 在图表下方添加解释文本

plt.figtext(0.5, -0.05, "\n".join([f"{name}: {desc}" for name, desc in zip(metrics\_names, metrics\_explanations)]),

ha="center", fontsize=10, bbox={"facecolor":"white", "alpha":0.8, "pad":5})

plt.tight\_layout()

plt.subplots\_adjust(bottom=0.3) # 调整底部边距以容纳解释文本

plt.show()

def plot\_model\_loss(self):

"""优化的损失曲线可视化"""

if self.history is None:

raise ValueError("请先训练模型")

plt.figure(figsize=(12, 6))

plt.plot(self.history.history['loss'], label='训练损失', color='blue', linewidth=2)

plt.plot(self.history.history['val\_loss'], label='验证损失', color='red', linewidth=2)

# 添加最佳迭代点标记

best\_epoch = np.argmin(self.history.history['val\_loss'])

best\_loss = np.min(self.history.history['val\_loss'])

plt.scatter(best\_epoch, best\_loss, color='green', s=100, zorder=5,

label=replace\_special\_chars(f'最佳迭代点: Epoch {best\_epoch+1}'))

plt.title('模型训练与验证损失曲线', fontsize=14)

plt.xlabel('Epoch（迭代次数）', fontsize=12)

plt.ylabel('损失 (均方误差)', fontsize=12) # 将MSE改为均方误差

plt.yscale('log') # 使用对数刻度更清晰地展示损失变化

plt.legend()

plt.grid(True, alpha=0.3)

plt.tight\_layout()

plt.show()

def main():

# 设置收盘价列名

close\_col = "closing price"

# 创建预测器实例

predictor = StockPricePredictor(

time\_steps=30,

epochs=200,

batch\_size=32,

close\_price\_col=close\_col

)

# 加载数据

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

try:

predictor.load\_data(excel\_path, file\_type='excel')

except Exception as e:

print(f"加载数据出错: {str(e)}")

return

# 预处理数据

try:

X\_train, X\_test, y\_train, y\_test = predictor.preprocess\_data()

except Exception as e:

print(f"预处理出错: {str(e)}")

return

# 训练模型

history = predictor.train\_model(X\_train, y\_train, X\_test, y\_test)

# 绘制损失曲线

predictor.plot\_model\_loss()

# 评估模型（包含指标可视化）

predictor.evaluate\_model(X\_test, y\_test, X\_train, y\_train)

# 可视化预测结果

predictor.plot\_predictions()

# 预测下一天

predictor.predict\_next\_day()

print("\n预测完成！")

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

main()