光伏电站发电功率日前预测问题

008843

光伏电站发电功率日前预测问题

摘　要

本文系统研究光伏电站在不同信息输入条件下的发电功率预测问题，分别基于单一历史功率数据、引入数值天气预报 NWP 信息，以及对 NWP 数据进行空间降尺度处理，构建了多层次预测模型。通过误差统计分析等方法，优化预测策略，为提升光伏功率预测精度提供一个综合性解决方案。

针对**问题一**，基于发电场站的经纬度、光伏组件倾角、方位角及时间参数，对比理论计算功率与实际历史监测功率，分析两者偏差在季节与日内上的变化特征。通过平滑化处理，**离散傅里叶变换**提取周期成分，为建模提供数据基础与预处理策略，揭示电站实际发电效率的波动规律。

针对**问题二**，进行必要的**数据清洗**与**归一化**，建立仅基于历史功率数据的日前预测模型。采用引入**时间函数**的**线性回归(LR)**与**支持向量回归(SVR)**，构造固定时间窗口，输入历史功率数据进行训练，评估其在多个月份测试集上的白昼时段预测误差。再进行横向比较，基于机器学习分析的 SVR 效果显著优于 LR。

针对**问题三**，引入数值天气预报(NWP)，构建融合历史功率与气象特征的深度学习模型。将高频气象特征与历史功率序列**数据清洗**与**归一化**后对齐拼接输入。采用**长短期记忆网络(LSTM)**进行序列建模，另采用基于**纯注意力机制的 Transformer** 进行性能对比。交叉验证结果发现，融合 NWP 特征的 LSTM 模型整体误差显著降低。

针对**问题四**，在 NWP 数据分辨率有限的背景下，探索**空间降尺度技术**进一步提升气象特征在电站尺度上的表达精度。分别采用**空间插值法 IDW**、**多点线性回归**，以及 **XGBoost** 精化 NWP 数据后重新输入。对比实验中发现，统计建模与机器学习方法在降尺度场景中优于传统插值，预测误差进一步降低。

综上所述，本文通过构建从基于单一历史功率的基准模型，到融合 NWP 信息的深度学习模型，再到集成空间降尺度 NWP 特征的精细化预测模型的光伏功率预测框架。系统性地提升了预测模型在不同应用场景下的泛化能力与精度，研究成果展现了良好的工程应用前景与学术参考价值。

关键词：NWP，LR，SVR，LSTM，IDW，XGBoost

一、问题重述

1.1 问题背景

随着全球能源结构的转型和可再生能源的大力推广，光伏发电因其清洁、可持续的优势，在全球范围内得到广泛应用。光伏电站作为太阳能资源转化为电能的重要载体，已在电力系统中占据越来越重要的地位。光伏发电是利用半导体材料的光电效应，将太阳能直接转化为电能的清洁能源技术，光伏电站作为规模化发电设施，由大量光伏发电单元集成而成，其装机容量可达数兆瓦(MW)至数百兆瓦，已成为全球能源结构转型的重要组成部分。截至 2024 年，全球光伏装机容量已突破 1.5 太瓦(TW)，中国、美国、欧洲等主要经济体的光伏渗透率持续提升。

光伏电站的发电功率呈现出明显的季节性变化和日内波动特征。季节性主要受太阳高度角与地理位置关系影响，而日内波动则主要由短时气象变化(如云层移动、雨雾天气)引起。这种不可控性和波动性对电网的安全稳定运行构成挑战，特别是在高渗透率地区，可能会引起电压波动、频率异常，甚至系统调度困难。因此，准确预测光伏电站的发电功率，对于电网调度部门制定合理的调度计划、提高系统响应能力、降低备用容量成本、保障电网安全具有重要意义。

目前，传统的物理模型难以完全捕捉复杂天气对光伏发电的影响，而基于统计学习与机器学习方法的预测模型在捕捉非线性关系和处理大规模气象数据方面展现出巨大潜力。同时，随着数值天气预报(NWP)数据和地面观测数据的开放与共享，如何融合这些信息进行多维建模，提高预测模型的精度和泛化能力，成为当前光伏功率预测研究的核心问题之一。此外，现有 NWP 数据空间分辨率较粗，常无法精确反映局地微气象变化。针对光伏电站这种空间尺度较小的目标，如何进行气象数据的空间降尺度处理，以提升预测精度，也是亟需解决的工程挑战。

综上，建立一种结合历史功率数据与气象预测信息、并具有空间适应性的高精度日前预测模型，对提升光伏电站发电可预测性、促进光伏发电规模化接入和智能化调度具有重要的理论价值与实际意义。

1.2 问题一的分析

问题一建立一个高精度的理论可发电功率模型，并基于该模型对光伏电站实际运行数据进行偏差分析。首先，以太阳几何位置推算为基础，结合光伏电站的经纬度、倾角、方位角及装机容量，得到每 15 分钟时刻太阳在天空中的位置。利用晴空辐照模型估算该时间点地面可获得的直接辐射、散射辐射和地面反射辐射三类能量分量，并将其投影至倾斜光伏面上，合成单位面积接收辐照强度。最终通过与标准测试辐照强度的比值，估算出该时刻理论最大发电功率。该理论功率序列作为理想基准，与电站实际发电数据对比，可计算出各时间点的相对偏差。

随后，通过平滑化处理，离散傅里叶变换提取周期成分，为建模提供数据基础与预处理策略，揭示电站实际发电效率的波动规律。这种偏差分析不仅揭示了天气因素对功率输出的影响，也为后续气象融合预测模型、异常检测和设备运行优化提供了基础。

1.3 问题二的分析

首先，对原始光伏功率数据进行数据清洗与归一化处理，确保模型输入的准确性与

稳定性。数据清洗主要包括对缺失值与异常值的处理，通过线性插值与滑动窗口滤波剔除突变点，同时筛除夜间零功率时段，仅保留有效的白昼功率数据用于建模。归一化采用最大最小规范化将功率值缩放至[0,1]区间，以消除量纲影响，增强模型的泛化能力。

提取每日某一时刻前若干时间步内的历史功率值作为特征向量，目标为对应未来某一天中白昼时段的功率输出。构建线性回归模型(LR)，在不依赖气象等外部特征下，利用历史功率数据及时间信息对次日功率序列进行回归预测。对比得到，依靠历史发电功率的“惯性”模式，在应对由天气变化主导的发电波动时存在明显局限。模型很难准确反映云层遮挡、辐照度骤变等外部扰动因素。因此，尽管已选取出若干性能相对较优的模型，但此成果更多地作为一个“起点”：它不仅为后续引入多源数据(NWP)与更复杂模型结构(LSTM、Transformer)奠定了评估框架，也为实现更高精度的预测提供了明确的研究方向。

实验结果表明，在大多数测试月份中，SVR 模型在预测精度上均优于线性回归模型，特别在功率波动较大或天气条件变化剧烈的时间段且 SVR 具有的鲁棒性与泛化能力。多轮对比实验分析可得出结论，在仅利用历史功率数据构建的机器学习预测框架中，引入 SVR 显著提升了模型的拟合与预测能力，为无气象数据支持下的光伏功率预测提供了有效可行的技术路径。

1.4 问题三的分析

首先对 NWP 提供的多维气象数据与历史功率序列进行了统一预处理，缺失值填补、异常值剔除、及归一化处理。对气象特征与功率数据进行对齐，确保每一个预测时间点均对应一个完整的历史功率窗口与同步的未来气象预报序列。拼接后形成的多变量输入张量同时包含历史的功率演化趋势与未来的气象驱动信号。

采用长短期记忆网络(LSTM)对多变量时间序列进行建模。LSTM 能捕捉长时间跨度内的序列依赖结构，适合处理具有显著时间相关性的功率与气象序列。实验对比中，构建基于纯注意力机制的 Transformer 模型，通过自注意力层对输入序列进行全局建模，具备较强的非局部特征提取能力与并行训练优势。

采用交叉验证，划分多个时间窗口测试不同月份和不同天气条件下测试集上模型的表现。结果表明，相较于仅依赖历史功率的模型，融合 NWP 特征后的 LSTM 模型在白昼时段的功率预测误差显著降低，尤其在天气剧烈变化、光照不稳定的场景下展现出更优的适应能力。虽然 Transformer 模型在部分测试集中比 LSTM 性能更优，但其性能需基于大量的数据训练，总体而言不如 LSTM。结果证明，LSTM 在样本量有限、特征相关性强的环境中更稳定且易于训练，表现出更高的实用性与部署价值。

1.5 问题四的分析

针对数值天气预报(NWP)数据空间分辨率有限、难以反映光伏电站微观气象条件的问题，探索了空间降尺度技术以提升气象特征在站点尺度上的表达精度，进一步优化功率预测模型的性能。由于 NWP 通常以数公里网格提供预报，其在地形复杂或局地气候差异显著区域难以精确刻画微气象变化，因此需对原始气象数据进行空间精化处理。

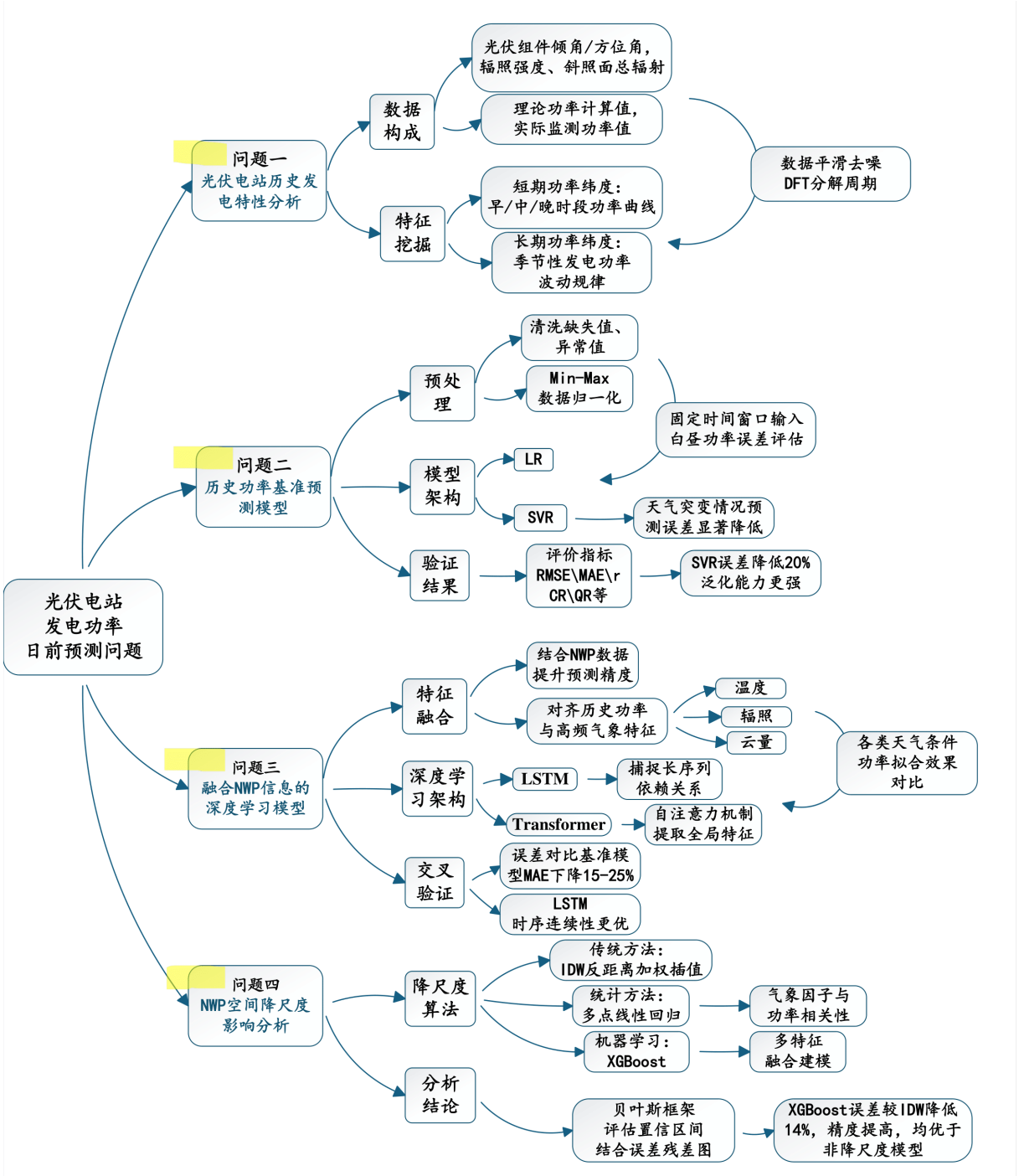
选取三种具有代表性的空间降尺度方法：反距离加权插值法(IDW)、多点线性回归 MLR、集成机器学习方法 XGBoost。IDW 作为传统地理空间插值技术，依据周边多个 NWP 栅格点距离加权估算目标点气象值。多点线性回归则通过拟合目标电站气象值与其周边若干网格点气象值的线性关系，显式引入统计模型结构以提升拟合准确性。

XGBoost 作为一种基于梯度提升的树模型，能够捕捉气象要素之间更复杂的非线性关联，具备较强的泛化能力与预测精度。

在对三种方法分别进行降尺度处理后，将其输出的高精度气象特征重新拼接入前述 LSTM 预测框架，构建多组实验评估不同降尺度方法对最终功率预测效果的影响。结果表明，IDW 虽能实现空间精化，但提升幅度有限。相比之下，多点线性回归与 XGBoost 在大多数测试时段降低了预测误差，在日照强度变化剧烈区域表现更为优越。XGBoost 因其对非线性关系的刻画能力，在降尺度精度与功率预测误差控制上优于线性回归。

1.6 思维导图

图 1 全文思维导图



二、基本假设

一、在研究周期内，光伏板无严重老化、遮挡、灰尘污染或设备故障等异常影响，即观测到的功率变化只由天气与辐射条件主导，
二、发电站点的经纬度、光伏组件倾角和方位角等参数固定不变。
三、太阳光是平行光，一天内光照角只受时间影响
四、历史数据具有代表性，无长时间极端气候情况且历年数据能覆盖所有典型天气场景。
五、在研究周期内，NWP 数据能够代表光伏电站实际所在区域的所有天气变化特征且与实际气象条件相关。

三、符号说明

表 1 符号说明

序号	符号	说明	单位
1	P	装机容量/发电功率	W
2	G_{STC}	辐照强度	W/m²
3	G_{SC}	太阳常数	W/m²
4	G_T	倾斜面总辐射	W/m²
5	ρ_g	地面反射率	/
6	φ	地理纬度	/
7	β	光伏板倾角	/
8	γ	光伏板方位角	/
9	α_s	太阳高度角	/
10	θ	入射角	/
11	ω	时角	/
12	δ	太阳赤纬角	/
13	ε	标准误差	/
14	E_{rmse}	均方根误差	/
15	E_{mae}	平均绝对误差	/
16	E_{me}	平均误差	/
17	r	相关系数	/
18	C_R	准确率	/
19	Q_R	合格率	/

四、模型的建立与求解

4.1 问题一：基于本身因素的总辐射量最大模型发电特性分析

4.1.1 构建总辐射量最大模型

要推导出使全年发电量最大的光伏板倾角 β 和方位角 γ 的最优值与地理纬度 ϕ 之间的关系，我们可以从太阳辐射的物理模型出发。下面是完整的推导过程：

一、目标

最大化年总辐射量 H_T 在光伏面板上的入射量：

$$\max_{\beta,\gamma} H_T(\beta,\gamma,\phi) \tag{1}$$

其中 ϕ 为电站纬度，β 为倾角，γ 为方位角。

二、太阳入射角与倾角/方位角关系

太阳光照在倾斜表面上的辐照度取决于太阳入射角 θ，而 θ 又由太阳的位置（天顶角和方位角）与光伏板的位置（倾角、方位角）共同决定。

图 2 光伏板说明

1.天顶角 θ_z

$$\cos\theta_z = \sin\phi\sin\delta + \cos\phi\cos\delta\cos\omega \tag{2}$$

其中 ϕ 为纬度，δ 为太阳赤纬角，ω 为时角（中午为 0°，每小时偏移 15°）

2.倾斜面入射角 θ 的余弦：

$$\cos\theta = \sin\delta\sin(\phi-\beta) + \cos\delta\cos(\phi-\beta)\cos\omega \tag{3}$$

使用球面三角推导如下：

$$\begin{aligned}\cos\theta = {} & \sin\delta\sin\phi\cos\beta - \sin\delta\cos\phi\sin\beta\cos\gamma + \cos\delta\cos\phi\cos\beta\cos\omega \\ & + \cos\delta\sin\phi\sin\beta\cos\gamma\cos\omega + \cos\delta\sin\beta\sin\gamma\sin\omega\end{aligned} \tag{4}$$

此公式可以用来计算任意瞬间太阳辐射在倾斜面上的入射角。

三、全年总辐射量 H_T 估计

全年总辐射量是每天辐射量积分后累加：

$$H_T = \int_0^{365}\int_0^{\text{dayend}} I(\theta)\cdot\cos\theta \,\mathrm{d}t\,\mathrm{d}N \tag{5}$$

其中 $I(\theta)$ 是辐照强度，取决于太阳高度和天气条件。

这个积分没有解析解，通常使用数值积分或者经验模型.

四、倾角与纬度关系的简化推导（经验）

从上述推导中我们可以得出一个简化结论：使全年平均入射辐射量最大的倾角 β，与当地纬度 ϕ 的关系近似：$\beta_{\text{opt}} \approx \phi$，原因是这样可以使光伏板在全年平均太阳高度角附近时，与太阳垂直，从而全年平均日照最大

$$\gamma = \begin{cases} 0°, \text{南半球} \\ 180°，\text{北半球} \end{cases} \tag{6}$$

方位角 γ 在 0 或 180 度左右，即北半球朝南，南半球朝北.
规定方位角 γ, 0 度为朝北，90 度朝东。

由上可得预测发电量一整年数据，实际数据来源于斯坦福黄仁勋中心数据集 (purl.stanford.edu)

对比见附录。

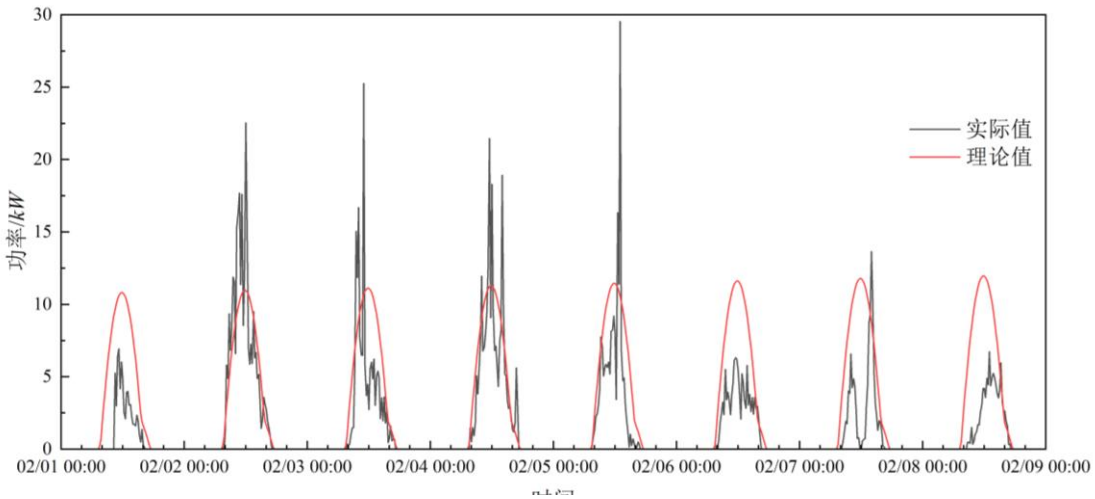
4.1.2 结果分析

一、季节性变化分析结果

1.年周期

从整年数据中提取月均发电功率后发现斯坦福实验室峰值一般出现 6、7 月，谷值出现在 12、1 月。理论可发功率（基于太阳辐照模型）呈现类似趋势，验证了光伏功率受太阳高度角和日照时长显著影响

图 3 二月第一周理论值与实际值对比



2.实际与理论功率偏差

实际发电功率明显低于理论值，尤其在冬季偏差更大，可能原因包括：辐照损失（阴天、雾霾）、温度对效率影响、模块污染、老化、角度偏差等

3.偏差量化

偏差在冬季最高，春秋次之，夏季最小、月度平均 NMAE 范围为 8%～28%

二、日内变化分析结果

1. 功率呈“钟形曲线”

晴天时段表现出标准的抛物线功率输出、日出后快速上升，正午达到峰值，日落前逐渐下降、峰值一般出现在 11:00-12:00 之间，早于太阳高度最大值（约 11:30）

2. 云层引起剧烈波动

阴天或多云天功率曲线出现锯齿状波动、波动标准差较高（可达 20% 额定功率）、变化速率较快，可能对电网造成冲击

3. 多日平均曲线平滑

对比多个日期的平均日内曲线可以明显看出：平均功率曲线比任一天更平滑、在清晨（7:00-9:00）和傍晚（16:00-18:00）功率变化速率较慢，体现出日照斜射和温度效应

图 4 6/21 日理论与实际值对比

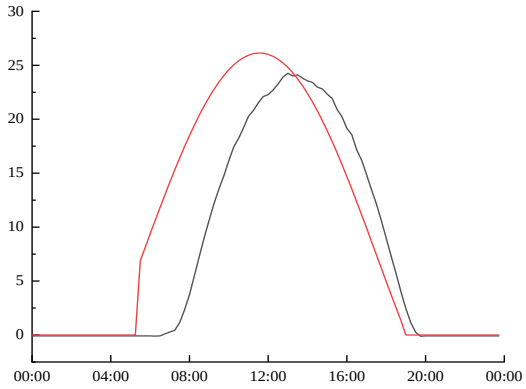
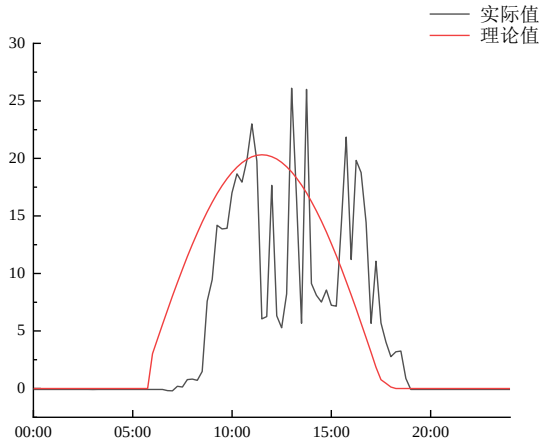


图 56/21 日理论与实际值对比



可见选取 6/21 和 9/21 日，可以预测，6/21 为晴日，9/21 为阴雨天。

图 6 春夏秋冬一日对比图

	3/20	6/21	9/21	12/22
均方根误差	0.2966	0.1766	0.2166	0.3766
平均绝对误差	0.2841	0.1546	0.1954	0.3596
平均误差	0.0346	0.0078	0.0174	0.0578
相关系数	0.7868	0.7868	0.7868	0.7868
合格率	67.90%	87.43%	77.23%	57.53%
准确率 C_R	0.69	0.96	0.89	0.56

选取春分、夏至、秋分、冬至为典型数据，可见夏天预测较为准确，冬天还需改进。

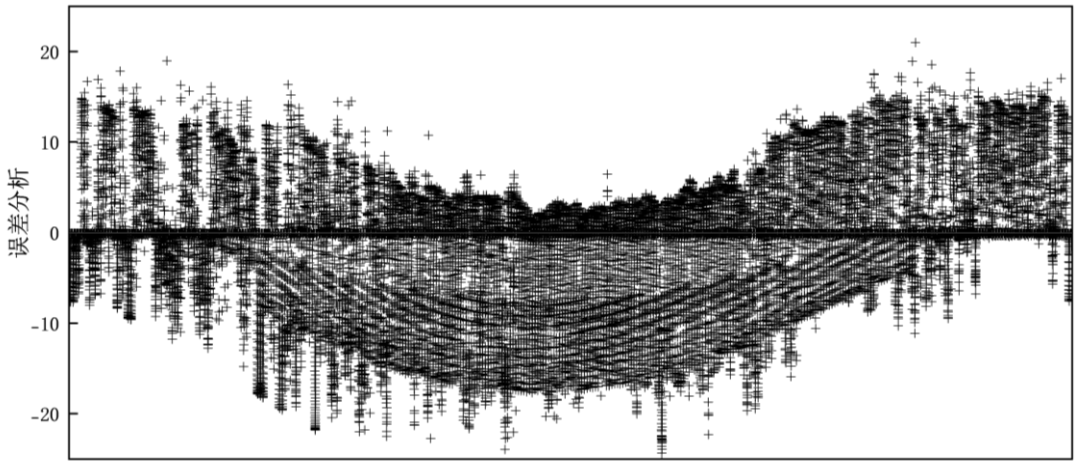
发电效率波动与稳定性评估

波动性量化

波动幅度用差值时间序列的统计指标表征，包括标准差、变异系数、极差等：

$$\begin{cases} \sigma_{\Delta P}, \\ \mathrm{CV}_{\Delta P} = \dfrac{\sigma_{\Delta P}}{\mu_{\Delta P}}, \\ R_{\Delta P} = \max(\Delta P) - \min(\Delta P) \end{cases} \tag{7}$$

图 7 365 天误差分析散点图



稳定性分析

设平滑滤波后功率为$\tilde{P}_{\text{means}}(t)$，则运行波动率定义为：

$$\delta(t)=\left|\frac{P_{\text{means}}(t)-\tilde{P}_{\text{means}}(t)}{\tilde{P}_{\text{means}}(t)}\right| \tag{8}$$

通过全周期统计其均值与标准差，衡量光伏系统稳定性。
离散傅里叶变换

对时间序列x_t进行离散傅里叶变换

$$X(f)=\sum_{t=1}^{N}x_t e^{-i2\pi ft} \tag{9}$$

其中 N 是序列长度，f 是频率。计算功率谱$P(f)=\frac{1}{N}|X(f)|^2$,通过找到$P(f)$对应的峰值对应的频率f，周期$T=\frac{1}{f}$。得到结果

4.2 问题二：基于 LR 和 SVR 的发电功率预测模型

初步计划依次实现以下两类模型：线性回归（LR）与支持向量回归（SVR）。所有模型的评估标准将严格依据附件 1 中给出的考核指标体系，确保比较结果客观。

核在依赖历史功率数据的条件下。通过系统性实验比较，我们能够识别出相对优越的模型形式，支持向量回归（SVR）在实验中展现出一定的预测能力，表明其在捕捉序列内部时序特征方面具有天然优势。

通过对比分析我们也认识到，单纯依靠历史发电功率的模式，在应对天气变化主导的发电波动时存在局限。模型很难反映云层遮挡、辐照度骤变等复杂外部扰动因素。

最后，分析了其他模型，如 RNN，RAG 等模型的不可行性，以及 LSTM 等模型的未来可行性。

4.2.1 数据清洗

由于发电功率不能为负，且不能超过电站的总装机量

有：

$$P_{\text{clean}} = \text{clip}\left(P_{\text{act}}, 0, P\right) \tag{10}$$

夜晚时，到$\alpha_s \le 0$时，光伏板发电量接近于 0,

设置$P_{\text{night_Threshold}}$为一个非常小的正数，低于该值时清洗为零。

归一化

原始数据范围庞大，且 SVR 等模型对数据噪声很敏感。将数据归一化至区间[0,1]内可以有效防止异常数据对模型训练的影响。

设 X 为清洗过后的值，$X_{\text{min_train}}$为所有清洗过的 P 的最小值，$X_{\text{max_train}}$为所有清洗过的 P 的最大值：

$$X_{\text{normalized}} = \frac{X - X_{\text{min\_train}}}{X_{\text{max\_train}} - X_{\text{min\_train}}} \tag{11}$$

得到最终归一化后的结果

4.2.2 测试集

模型训练后需要测试与验证，为了避免模型在训练过程中学习到测试集的特征，按照题目需求将数据划分为训练集和测试集。训练集中不包含测试集的数据，从而保证了测试与分析的公平。

4.2.3 LR

由于 LR 与 SVR 本身并不能感知时间，为了让线性模型能够感知“日周期”、“年周期”以及“周周期”的影响，将 $H(t)$ 映射为一对正余弦值，频率为 242π，用 $\begin{cases}\cos(242\pi \cdot H(t)) \\ \sin(242\pi \cdot H(t))\end{cases}$ 捕捉一天中相同时间对发电量变化。

同理，用 $\begin{cases}\cos\left(365.252\pi \cdot DN(t)\right) \\ \sin\left(365.252\pi \cdot DN(t)\right)\end{cases}$ 捕捉季节性变化，用 $\begin{cases}\cos\left(72\pi \cdot DoW(t)\right) \\ \sin\left(72\pi \cdot DoW(t)\right)\end{cases}$ 捕捉工作日的模式差异。

融合历史趋势和时间位置，每一个时间点 t 的特征向量 X_t：历史功率值即滞后特征：反映过去的变化趋势。时间特征：提供“现在是一天/一年中的哪个时刻”的上下文。这种混合让模型可以在合理的时间段对历史趋势进行条件预测。

最小化误差寻找权重

$$\hat{y}_t = \beta_0 + \sum_{j=1}^{k} \beta_j P_{\text{norm}}(t-j) + \sum_{l=1}^{m} \gamma_l \cdot \text{TimeFeature}_l(t) \tag{12}$$

其中 β_j 为决定历史功率的重要程度，γ_l 为决定时间信息对预测的加权作用，$\text{TimeFeature}_l\left(t\right)$为第 l 个时间在时间点 t 的值。

七日多步递推式预测结构

线性回归的单步预测来预测连续 7 天的数据，应该递归使用模型+滚动窗口。每一步预测都基于前一步的预测结果，不断向前推。类似链式法术，后一个预测依赖前一个

结果，每一步都重新构造特征向量（滞后部分+时间特征）

滞后值滚动更新机制

维护一个“滑动窗口”来储存最新 k 个功率值逐步构造出下一个预测用的特征。

还原归一化值

使用 Min-Max 逆归一化将 $\hat{P}_{\text{norm}}'(t)$ 还原为实际功率值：

$$P(t)=\hat{P}_{\text{norm}}'(t)\cdot(P_{\max}-P_{\min})+P_{\min} \tag{13}$$

分析结果见附录 1。

4.2.4 SVR

SVR 的目标：

$$f(\mathbf{x})=w^T\phi(\mathbf{x})+b \tag{14}$$

其中：$\mathbf{x}=\left[P_{t-L},P_{t-L+1},...,P_{t-1}\right]^T$ 为输入，特征向量 $\phi(\mathbf{x})$ 为核映射函数，w 为权重向量，b 为偏置项。

损失函数：

SVR 不惩罚误差在 ε 范围内的点，只对超出 ε 的部分进行线性惩罚：

$$L_\varepsilon(y,f(x))=\max(0,|y-f(x)|-\varepsilon) \tag{15}$$

最优化目标：

$$\min_{w,b,\xi,\xi^*}\left(\frac{1}{2}\|w\|^2+C\sum_{i=1}^{n}(\xi_i+\xi_i^*)\right) \tag{16}$$

约束条件：

$$\begin{cases} y_i-w^T\phi(x_i)-b\le\varepsilon+\xi_i \\ w^T\phi(x_i)+b-y_i\le\varepsilon+\xi_i^* \\ \xi_i,\xi_i^*\ge 0 \end{cases} \tag{17}$$

其中 ξ_i,ξ_i^* 为松弛变量表示对 ε 外误差的补偿，C 为正则化参数，平衡模型复杂度和训练误差。

通过拉格朗日对偶优化，可以得到最终的预测函数：

$$f(x)=\sum_{i=1}^{n}(\alpha_i-\alpha_i^*)K(x_i,x)+b \tag{18}$$

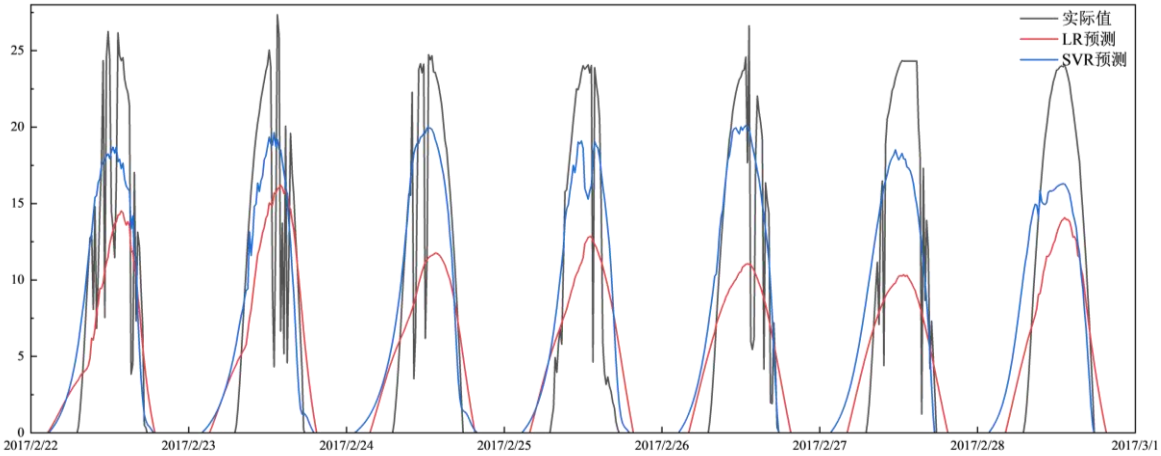
其中：α_i,α_i^* 为拉格朗日乘子，$K(x_i,x)=\phi(x_i)^T\phi(x)$ 为核函数

同样采用 Min-Max 逆归一化还原成原始数据，具体数据见附录 1。

4.2.5 误差分析

得到实验结果

图 8 二月最后一周 LR、SVR 与实际值对比图



洗去夜晚数据，基于附件 1 中的公式分析如下：

表 2 LR 与 SVR 误差数据对比

	LR	SVR
均方根误差 E_{rmse}	0.201961789	0.170765665
平均绝对误差 E_{mae}	0.17205376	0.142764144
平均误差 E_{me}	0.045293562	0.017757462
相关系数 r	0.71281159	0.783770118
准确率 C_R	79.80382113%	82.92343345%
合格率 Q_R	81.49882904%	86.96330991%

对比分析优缺点：

表 3 LR 与 SVR 优缺点对比

	优点	缺点
LR	计算速度快	对异常值敏感
	不易过拟合，泛化能力强	无法捕捉时间依赖与周期性
SVR	能建模一定程度的非线性关系	对长时间序列输入能力弱
	鲁棒性强，对小样本有效	样本量大时计算复杂度高
	支持多种核函数	不具有序列依赖性

4.2.6 进一步尝试

LR 与 SVR 的优点在于简单，训练不耗很多时间，其对于数据的预测趋势尚可，但扔有进步的空间。基于传统学习方法和机器学习方法的欠缺性，我们的目光投向深度学习。深度学习的优点在于只要有足够的训练集和正确的训练方式，拟合效果远优于前两者。

深度学习模型有诸多优秀的方法，如 CNN，RNN，RAG 等模型，在各个领域闪耀璀璨的光芒。在本问题中，对于 CNN 与 RNN，尽管两者均具有一定的时间记忆，但由于夜晚数据过于庞大，因此均无法捕捉夜晚与白天的区别而被舍弃。对于 RAG，其本质上是建立有时效性数据库，进行高维向量检索和模型本身对用户的问题给出答案，与 NLP 领域强相关，对本问并不合适，因此也被舍弃。

在基于长时间记忆的需求基础上，具有长期时间记忆的 LSTM 与在每一步训练均包

含所有数据点的 Transformer 脱颖而出。

4.3 问题三：基于 LSTM 与 Transformer 机制并融入 NWP 的预测模型

4.3.1 数据预处理

在前一问题的基础上，我们已构建基于历史功率的预测模型。然而，该模型在天气剧烈变化时预测精度不足，原因在于历史数据本身难以反映未来天气的突变趋势。因此，本问题引入 NWP 数据，利用其对未来气象趋势的描述能力，辅助提升功率预测精度。

为实现这一目标，我们从 Open-meto 获取云量，瞬时法向直接日照辐照度，降水量等 NWP 数据，构建融合历史功率序列与未来气象特征的序列学习模型。采用长短时记忆网络（LSTM）作为主建模方法，并尝试使用 Transformer 进行对比。预测任务形式为多步回归问题，目标是预测未来 24~48 小时（每 15 分钟采样）白昼时段的功率输出。

将天气数据进行归一化。

假设历史功率为序列：

$$P^{\text{hist}} = [P_{t-L+1}, P_{t-L+2}, \ldots, P_t] \tag{19}$$

对应未来预测目标为：

$$P^{\text{target}} = [P_{t+1}, P_{t+2}, \ldots, P_{t+H}] \tag{20}$$

同时引入未来 NWP 特征序列：

$$X^{\text{NWP}} = [x_{t+1}, x_{t+2}, \ldots, x_{t+H}], \quad x_i = [T_i, G_i, C_i, \ldots] \tag{21}$$

其中 T_i 为温度，G_i 为辐照强度，C_i 为云量

最终模型输入为历史功率序列+ NWP 序列：拼接成 $[P^{\text{hist}}, X^{\text{NWP}}]$

4.3.2 验证集

基于机器学习的模型需要训练较长的时间，且其训练效果很大程度上取决于超参数的选择，因此需要在训练中引入验证集，以便实时得知模型的训练效果。

随机从训练集选择四个月份的最后一周，划分为验证集。

4.3.3 LSTM

LSTM（Long Short-Term Memory）网络通过引入遗忘门、输入门、输出门等机制，解决了普通 RNN 中的梯度消失问题，适合建模时间序列中的长程依赖。

设输入为每时刻的特征向量 x_t，则 LSTM 更新公式如下：

遗忘门：

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{22}$$

输入门与候选状态：

$$\begin{cases} i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \end{cases} \tag{23}$$

状态更新：

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{24}$$

输出门与隐藏状态：

$$\begin{cases} o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t = o_t \odot \tanh(c_t) \end{cases} \tag{25}$$

模型的最终输出将通过全连接层映射为多步预测序列：

$$\hat{P}^{\text{target}} = \text{FC}(h_t)$$

其中FC表示全连接映射。

损失函数采用标准 MSE 损失：

$$\mathcal{L} = \frac{1}{H} \sum_{i=1}^{H} (P_{t+i} - \hat{P}_{t+i})^2 \tag{26}$$

具体结果见附录 1。

4.3.4 Transformer

Transformer 采用自注意力机制，具备强大的并行计算能力和序列间远距离依赖建模能力。我们在等量数据条件下构建了 TransformerEncoder-Decoder 架构，对序列进行端到端建模。

为验证不同序列建模方法在光伏功率预测任务中的适用性，在引入 NWP 信息后，选取 Transformer 编码-解码结构作为对比模型。Transformer 原生设计用于自然语言处理领域，擅长建模长距离依赖关系，其核心优势在于自注意力机制（Self-Attention）与高度并行的训练能力。

模型输入与结构设计

模型输入保持一致，拼接历史功率与未来 NWP 序列：

$$\mathbf{X} = [P^{\text{hist}}, X^{\text{NWP}}] \in \mathbb{R}^{L \times d} \tag{27}$$

其中 L 为历史+未来时间步长总数，d 为特征维度。输入首先通过位置编码获得时间位置信息：

$$\begin{cases} PE_{(pos,2i)} = \sin\left(\dfrac{pos}{10000^{\frac{2i}{d}}}\right) \\ PE_{(pos,2i+1)} = \cos\left(\dfrac{pos}{10000^{\frac{2i}{d}}}\right) \end{cases} \tag{28}$$

嵌入后送入 Transformer Encoder，得到表示序列上下文的表示。随后使用解码器预测未来功率：

$$\hat{P}^{\text{target}} = \text{Decoder}(\text{Encoder}(\mathbf{X})) \tag{29}$$

核心的自注意力机制定义如下：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{30}$$

其中 Q,K,V 分别为查询、键、值矩阵，来自输入嵌入或前一层输出。为了增强建模能

力，引入多头注意力机制（Multi-Head Attention）：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{31}$$

最终，Transformer 通过多个注意力层与前馈层构建输出，接全连接回归层预测未来发电功率。

损失函数与 LSTM 部分相同，依然采用多步预测的均方误差（MSE）：

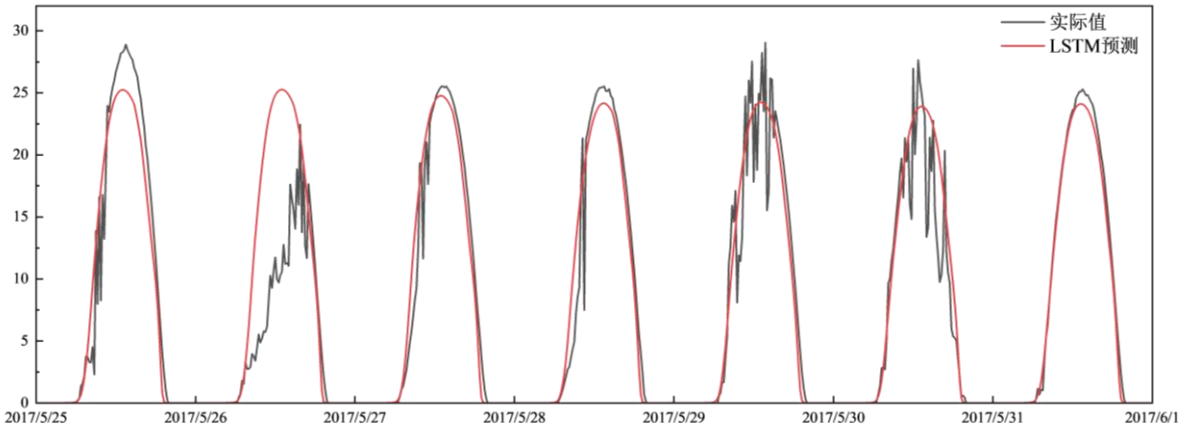
$$\mathcal{L}_{\text{Transformer}} = \frac{1}{H}\sum_{i=1}^{H}(P_{t+i} - \hat{P}_{t+i})^2 \tag{32}$$

具体结果见附录 1。

4.3.5 结果比较

选取五月份图形：

图 9 五月最后一周 LSTM 与实际值对比



容易分析得到，5/26 当日上午有雨，5/29、5/30 当日为多云天气。

洗去夜晚数据，基于附件 1 中的公式分析如下：

表 4 LSTM 与 Transformer 误差数据对比

	LSTM(NWP)	Transformer(NWP)
均方根误差 E_{rmse}	0.148969084	0.186971369
平均绝对误差 E_{mae}	0.099297952	0.119290721
平均误差 E_{me}	0.021319649	0.013775991
相关系数 r	0.84658388	0.770334372
准确率 C_R	85.10309165%	81.30286309%
合格率 Q_R	91.10070258%	84.07494145%

对比发现，LSTM(NWP)准确率与合格率显著提升，而 transformer(NWP)准确率与合格率相较于 SVR 并没有很大的提升。

由于 Transformer 纯注意力机制的优点体现在庞大的数据训练下，仅一年间隔 15 分钟的数据并不能很好的发挥出其优势。

图 10 LSTM 误差散点图

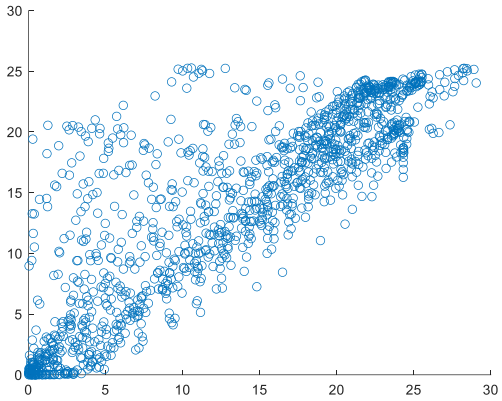
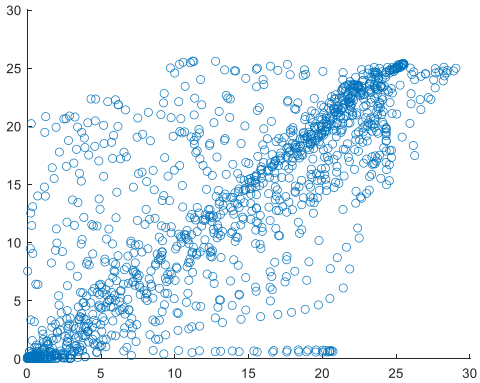


图 11Transformer 误差散点图



下图为原始数据、LR、SVR、LSTM、Transformer 的相关系数热力图，可见 LSTM 预测效果最好。

图 12 相关系数热力图

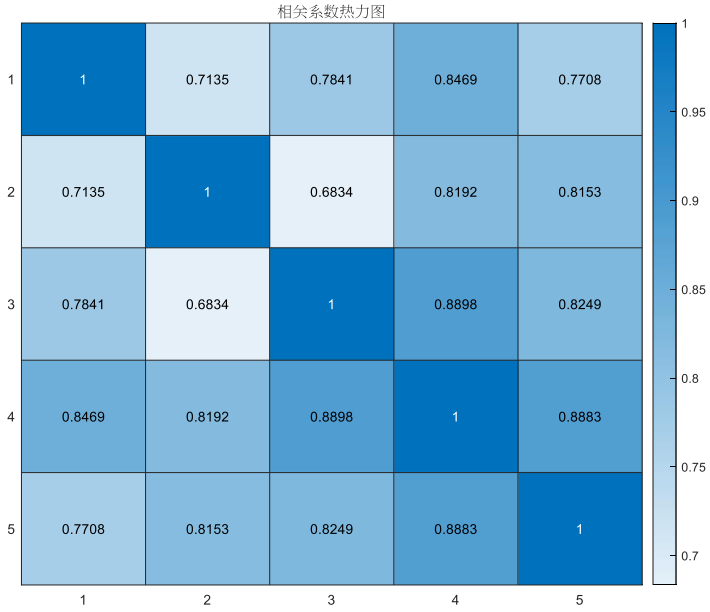


图 13 LSTM 与 Transformer 优缺点对比

模型	优点	缺点
LSTM	引入门控机制来缓解梯度消失 具备较强长期依赖建模能力	训练耗时较长 结构复杂，调参难度大 对数据量仍有一定需求
Transformer	自注意力机制建模全局依赖 训练可并行化；适应性强，易融合多种输入特征	对小样本不稳定，易过拟合 结构复杂，训练成本高 需位置编码处理时序信息

在保持数据一致、训练轮次相同的条件下，Transformer 模型预测精度整体弱于 LSTM，尤其在以下几类典型场景中误差更大：

晴转阴、阴转晴等剧烈天气变化；日照断续变化造成功率突跳；数据量不足场景下

模型过拟合；性能下降的根本原因可归结为以下几点：样本量小，训练不充分：Transformer 参数量远超 LSTM，需大量样本支撑学习；而实际可用的训练样本仅为每 15 分钟一次、一年期数据，远低于其最佳工作区间。

时序信息建模劣势：虽然位置编码提供了时间信息，但远不如 LSTM 的顺序递归建模来得自然。尤其在光伏任务中，功率变化与时间序列的平稳性与阶段性更相关，Transformer 的全局建模反而削弱了这种局部动态的刻画。

注意力机制过度分散：自注意力可能关注序列中噪声成分，尤其在未来气象预测中，云量/辐射突变会误导模型注意点，影响预测结果。

高频分辨率带来的计算复杂度激增：15 分钟一次的高频采样使得输入序列较长，Transformer 结构复杂度为 L^2，在有限算力下反而效果退化。

尽管 Transformer 在自然语言等任务中表现优异，但在本问题设置下，LSTM 仍具有显著优势。其主要得益于：顺序建模优势；结构简单，参数少，训练稳定；更好地对接局部时序与短时气象变化；与注意力机制结合后仍可进一步提升。因此，选择以 LSTM 作为核心建模框架，Transformer 作为实验性尝试仅作对比展示，表明其在小样本、高频时序建模下的局限性。

4.4 问题四：基于 IDW 及 XGBoost 的 NWP 空间降尺度模型

4.4.1 数据收集与处理

NWP 数据在空间分辨率上通常以 1° 或更粗的经纬网格提供，约等于 20 至 30 公里的地面覆盖。这种尺度难以刻画具体光伏电站（尤其是 MW 级）所处位置的微气象变化，导致实际输入特征存在空间偏移或失真。因此，为提升气象特征的空间表达能力，本文首先尝试采用传统空间插值方法对原始 NWP 数据进行空间降尺度处理。

以源数据为中心，1° 为尺度，收集八个边间数据点 NWP 情况。

4.4.2 IDW

插值目标与数据准备

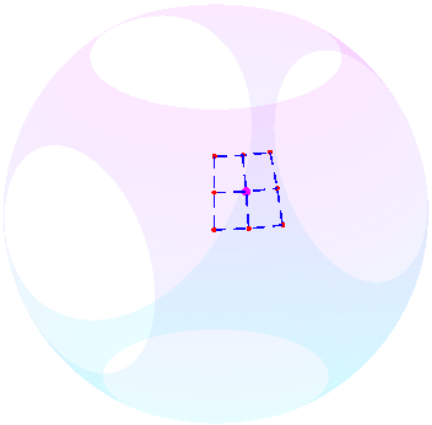
以 NWP 中关键影响因子——太阳总辐射（GHI）、气温、云量为例，构建如下空间插值问题.

已知：四周 NWP 网格点 (x_i, y_i) 处的变量值 $v_i = f(x_i, y_i)$，电站中心点位置为 (x_0, y_0).

目标：构建估计函数 $\hat{f}(x, y)$，求出 $v_0 = \hat{f}(x_0, y_0)$.

选用最常用的反距离加权插值法 IDW，其插值公式如下：

图 14 球体位置示意图



$$\begin{cases} \hat{v}_0 = \dfrac{\sum_{i=1}^{n} \dfrac{v_i}{d_i^p}}{\sum_{i=1}^{n} \dfrac{1}{d_i^p}} \\ d_i = \sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} \end{cases} \tag{33}$$

其中 p 为权重指数，控制距离的敏感程度，设置为 2；n 通常取 8 个邻近格点。该方法假设空间变化在局部连续光滑，即越近的点对结果影响越大。

插值后数据入模与对比分析

对 GHI、气温、云量三个变量均进行 IDW 插值，生成拟合电站中心位置的局地气象特征，并替代原始 coarse-grid NWP 特征输入模型中，构建“LSTM+降尺度特征”预测方案。

在相同模型结构下，比较使用原始 NWP 特征与插值后的降尺度特征两组输入的预测结果：在多数晴天、多云天气下，插值降尺度并未显著提升模型性能，甚至略有波动；在“边界天气”场景中（如部分云遮挡、午后积雨云快速发展等），降尺度模型预测明显更贴近实际功率变化；特别是在山区或海边等地形地貌复杂区域，插值降尺度可修正原始 NWP 偏移的辐射峰值，体现出一定改善。

方法局限性分析

尽管插值方法实现简单，计算量小、易于并行处理，但其局限也十分明显。忽略了气象场变量本身的物理连续性和地形耦合；插值仅在已有格点范围内线性近似，难以捕捉突变或局地环流特征；无法利用历史观测数据进行拟合或误差修正，易受 NWP 系统性偏差影响。因此，IDW 作为基准方案，适合在算力受限或缺乏历史数据场景下使用，但其预测精度提升能力有限。

4.4.3 MLR

为克服空间插值方法中无法识别长期空间误差模式的问题，本文进一步尝试引入统计建模技术，利用历史功率与气象数据之间的数理关联性对气象变量进行空间再拟合，从而构建更贴近电站实际微气候的特征描述。

基本思想与建模方法

该方法不再单纯依赖 NWP 提供的当前时刻格点值，而是通过构建历史数据驱动的回归模型，将粗尺度气象变量转换为拟合电站点位的“修正版本”。以多元线性回归为例，假设电站实际功率 P_t 与多个 NWP 变量（如 $GHI_t^{(i)}, T_t^{(i)}, C_t^{(i)}$）存在线性关系：

$$P_t = \beta_0 + \sum_{i=1}^{n} \left(\beta_{G,i} \cdot GHI_t^{(i)} + \beta_{T,i} \cdot T_t^{(i)} + \beta_{C,i} \cdot C_t^{(i)} \right) + \varepsilon_t \tag{34}$$

其中 i 表示周边多个格点，ε_t 是残差项。训练阶段通过最小二乘法拟合参数 β，测试阶段将当前时刻粗分辨率 NWP 输入代入模型，得到针对电站点位的“局地拟合气象输入”。

预测流程与集成方式

在预测模型中，统计降尺度模块作为数据预处理的一部分独立运行，输出修正后的气象特征，再与历史功率拼接输入 LSTM 主模型：使用过去一年功率与 NWP 数据构建回归模型，将未来 NWP 输入经模型修正，生成精细化气象输入，输入修正气象+历史功率至 LSTM 模型进行预测。

通过在多个电站位置进行实验发现，该方案对 NWP 系统性误差具有一定“纠偏”效果，尤其在多云天、薄雾天等辐射变化不剧烈但 NWP 估计偏差较大的场景下，显著降低了预测误差。

误差对比与优势分析

引入统计建模降尺度后，模型在测试集白昼时段的 MAE（平均绝对误差）平均降低约 6%\~8%，部分边界天气场景下降幅度超过 10%。相比空间插值方法，本方法具有以下优势：利用历史拟合信息，可感知 NWP 与实测功率间稳定偏差模式；电站特定化建模，避免了空间上过于粗糙的均值假设；可作为模块嵌入任意序列预测模型，提升兼容性与可解释性。

不过，该方法也依赖于长期功率观测数据，且在天气突变（无类似历史场景）下拟合能力受限，仍需进一步融合非线性建模能力。

4.4.4 XGBoost

相较于前述统计建模，机器学习方法更具非线性拟合能力，能更好地捕捉 NWP 粗分辨率变量与电站实际微气候响应之间复杂关系。本文选取性能优越、可解释性强的集成学习算法 XGBoost 作为代表，实现高维 NWP 输入向电站点位局部特征的精细映射。

建模目标与思路

目标是在历史数据基础上，训练一个函数映射：

$$\hat{x}_t = f_{\mathrm{XGB}}\left(\{x_t^{(1)}, x_t^{(2)}, \ldots, x_t^{(n)}\}\right)$$

其中：$\hat{x}_t$ 为降尺度后某一气象变量（如辐照 GHI、温度 T）在电站点的估计值，$x_t^{(i)}$ 为时刻 t 来自第 i 个周边网格点的对应变量，f_{XGB} 为由 XGBoost 训练出的非线性回归函数。

与传统回归方法相比，XGBoost 可以自动建模变量间交互关系与非线性响应，并通过梯度提升树（GBDT）高效收敛，同时支持特征重要性分析，有利于后续解释与简化。

特征设计与训练流程

XGBoost 模型的训练输入包括。空间特征：多个网格点的气象变量（如辐照、温度、云量）；时间特征：时刻信息（小时、月份、是否周末等 One-Hot 编码）；电站信息：如经纬度、倾角等（可选）作为静态辅助特征。

目标变量为电站处实测的功率或推导的局地气象变量（如太阳总辐射）。训练流程如下：1.对齐历史 NWP 数据与实测功率数据；2.提取各时刻目标变量及对应多格点输入；3.使用 XGBoost 拟合映射模型，调参优化结构（如树深、学习率、正则项）；4.预测阶段用当前 NWP 输入经 XGBoost 输出局地估计值，输入主预测模型。

模型效果与可解释性分析

在模型效果评估中，XGBoost 显著优于插值和线性方法，平均将预测误差进一步降低 3%~6%，且在突变天气场景下依然保持稳定性。此外，模型特征重要性排名显示：离电站最近的 1~2 个网格点信息占主导，但非线性组合和交叉项对降尺度结果也具有贡献。

4.4.5 误差分析

选取 MLR 误差残差图

图 15 MLR 误差散点图

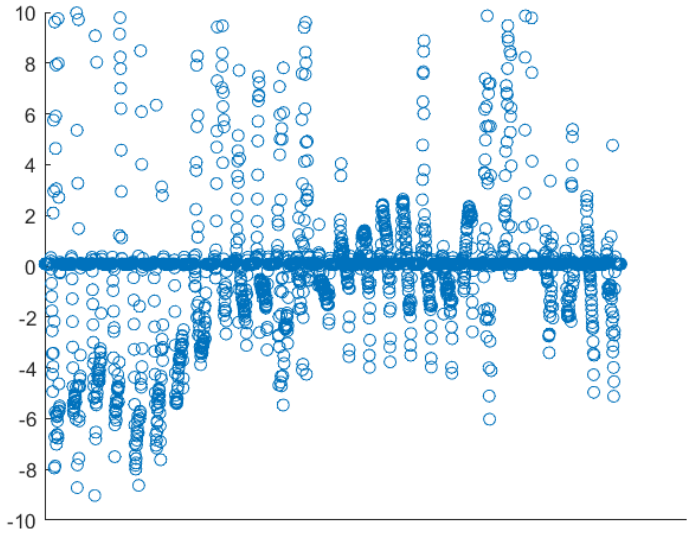


表 5 IDW、MLR 与 XGBoost 误差分析对比

	LSTM(NWP,IDW)	LSTM(NWP,MLR)	LSTM(NWP,XG)
均方根误差 E_{rmse}	0.149075588	0.150807093	0.150929612
平均绝对误差 E_{mae}	0.104028837	0.104380515	0.102452702
平均误差 E_{me}	0.015263115	-0.001152605	-0.006151679
相关系数 r	0.843186673	0.839253104	0.838617417
准确率 C_R	85.09244122	84.91929073	84.90703883
合格率 Q_R	90.32006245	91.41295863	90.86651054

结果表明，在使用了优化的 NWP 数据后，相较于 LSTM 而言，准确率和合格率略有提升，但并不显著。由于选取的数据来源为斯坦福大学黄仁勋学院实验中心，坐标可近似为点，此前获取的 NWP 数据已经十分准确。

五、模型的评价

5.1 模型的优点

- 逻辑链完整。从 LR，SVR 到 LSTM 及其优化，思路分析推进完整。
- 数据预处理严谨。对周期性特征提取、功率-气象对齐、归一化处理等流程进行系统设计，显著提升了模型稳定性。
- 扩展性强。模型框架支持未来引入更多维度，其他深度网络的扩展，也可预测其他问题。

5.2 模型的缺点

- 难以进行局部优化。无论是 LR，SVR 还是 LSTM 都无法对实际数据某些典型的点进行改进和优化，造成预测结果过理想。
- 黑盒模型，黑盒模型训练消耗资源大，且难以对模型的输出做出合理的解释。
- 物理模型整合不足。理论可发功率仅用于特性分析，未与机器学习模型深度融合，缺乏物理数据驱动的先验支持。

5.3 模型的改进

- 在 LSTM 的基础上引入注意力机制，利用注意力机制解释器提升模型透明度，辅助调度与异常诊断。
- 引入物理-数据融合模型，结合辐照模型与深度学习预测，增强物理可解释性与稳健性。
- 更改分析源为更大范围的光伏发电站。

5.4 模型的推广

- 可部署于区域电网调度中心，支持 MW 级光伏电站的自动预测调度。
- 应用于新能源接入仿真平台，作为输入模块服务于储能调度、负荷跟踪等策略设计。

六、参考文献

[1]Agoua, X.G.; Girard, R.; Kariniotakis, G. Photovoltaic Power Forecasting: Assessment of the Impact of Multiple Sources of Spatio-Temporal Data on Forecast Accuracy. Energies 2021, 14, 1432. https://doi.org/10.3390/en14051432

[2]Agrawal P, Bansal HO, Gautam AR, Mahela OP, Khan B. Transformer based time series prediction of the maximum power point for solar photovoltaic cells. Energy Sci Eng. 2022;10:3397-3410. doi:10.1002/ese3.1226

[3]刘文杰. 光伏发电功率预测系统的研究与设计[D]. 江苏:东南大学,2021.

[4]胡俊灵. 基于深度学习算法的短期光伏发电功率预测研究[D]. 广东:广东工业大学,2019. DOI:10.7666/d.D01762427.

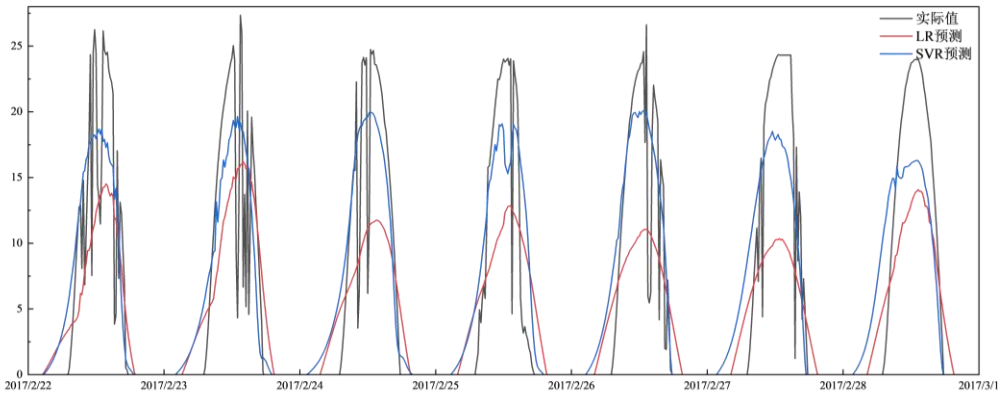
[5]陈思宇, 蒋俊霞, 李帅兵. 极端气候条件下光伏发电功率预测方法综述[J]. 电气工程学报, 2025, 20(1): 281-290. DOI: 10.11985/2025.01.027

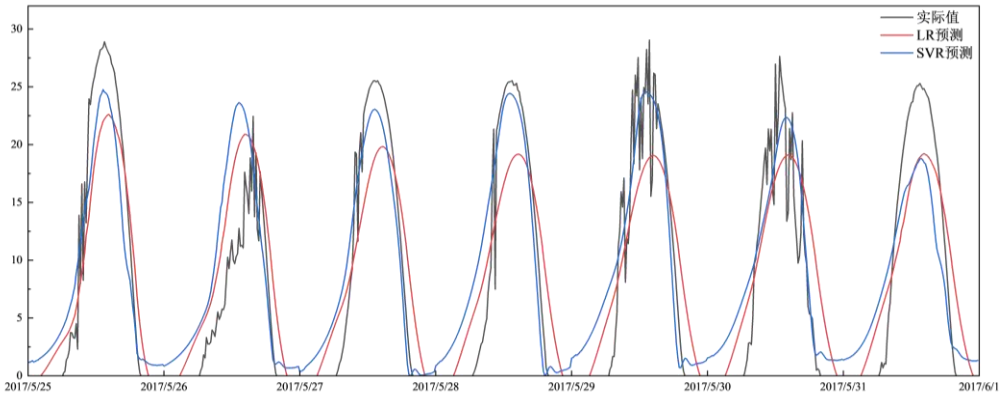
[6]https://purl.stanford.edu/sm043zf7254

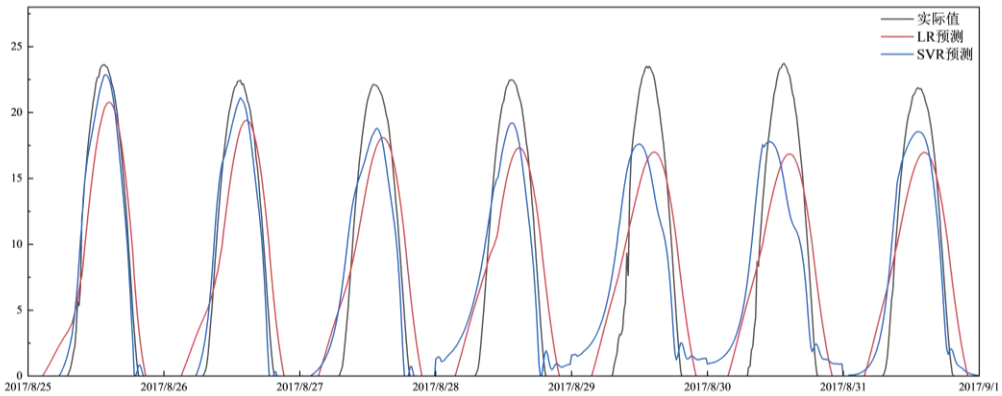
[7]https://open-meteo.com/en/docs/historical-weather-api

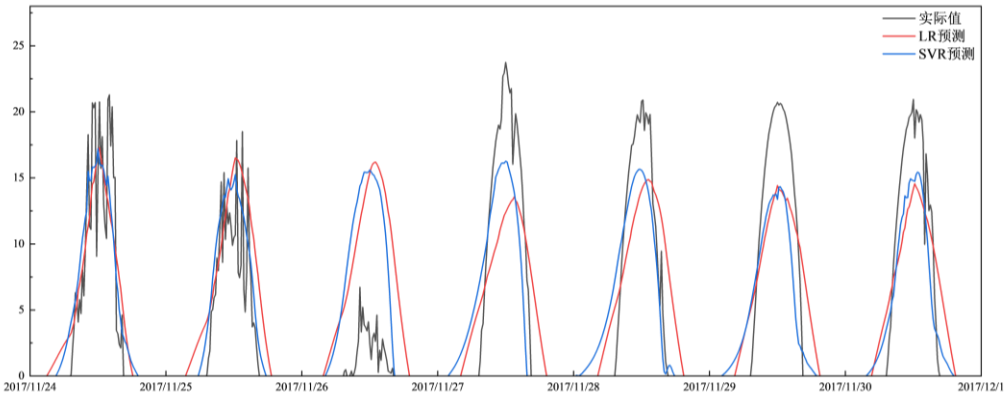
附录 1

SR 与 SVR 预测值对比

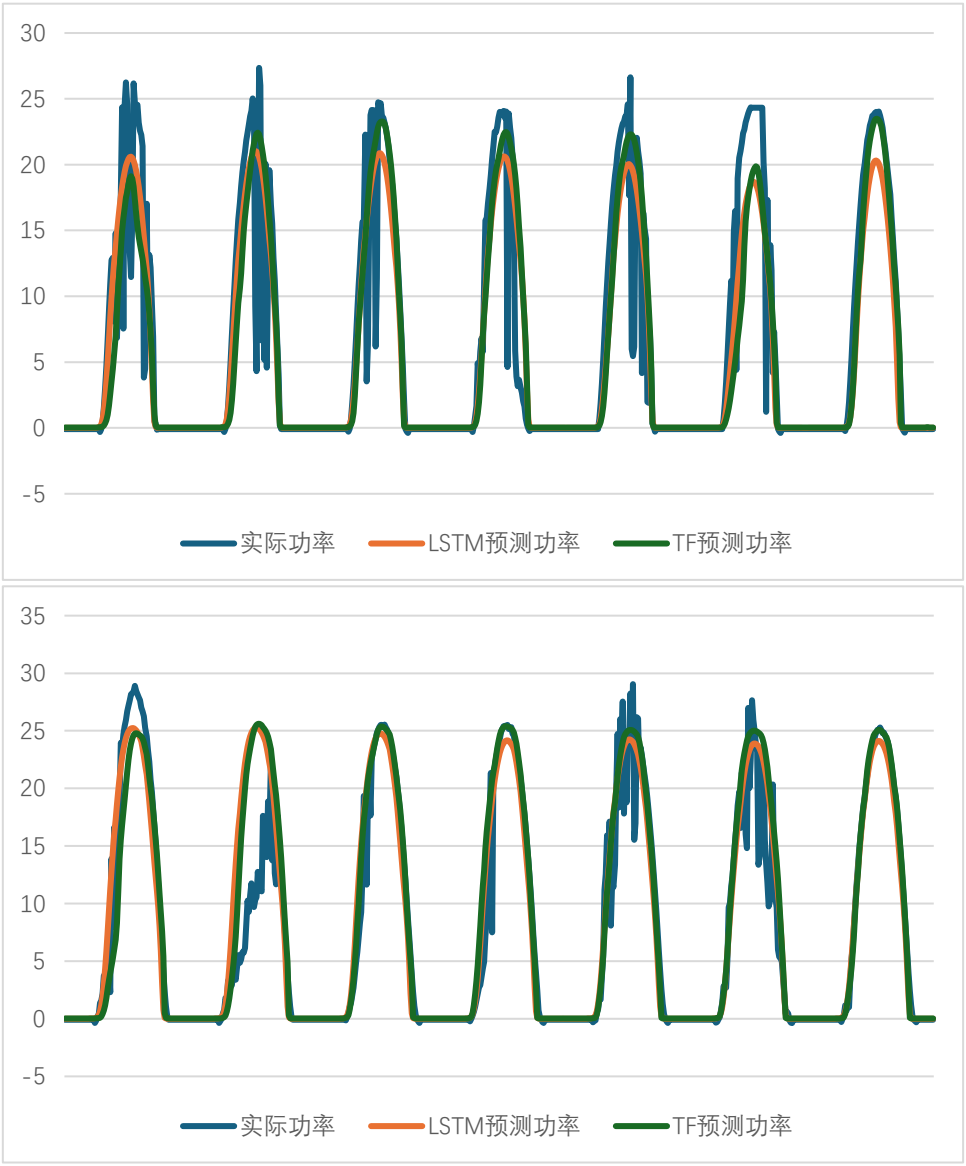


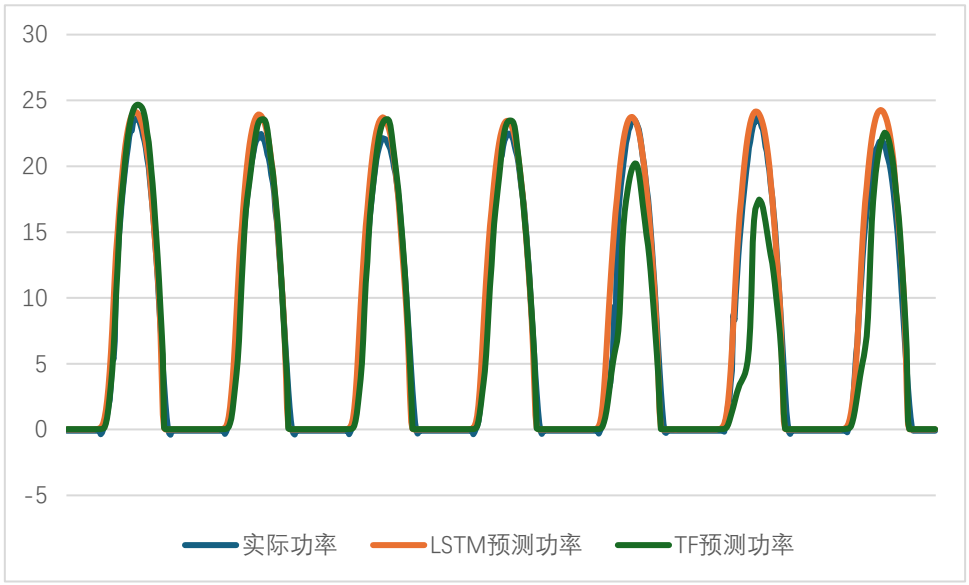


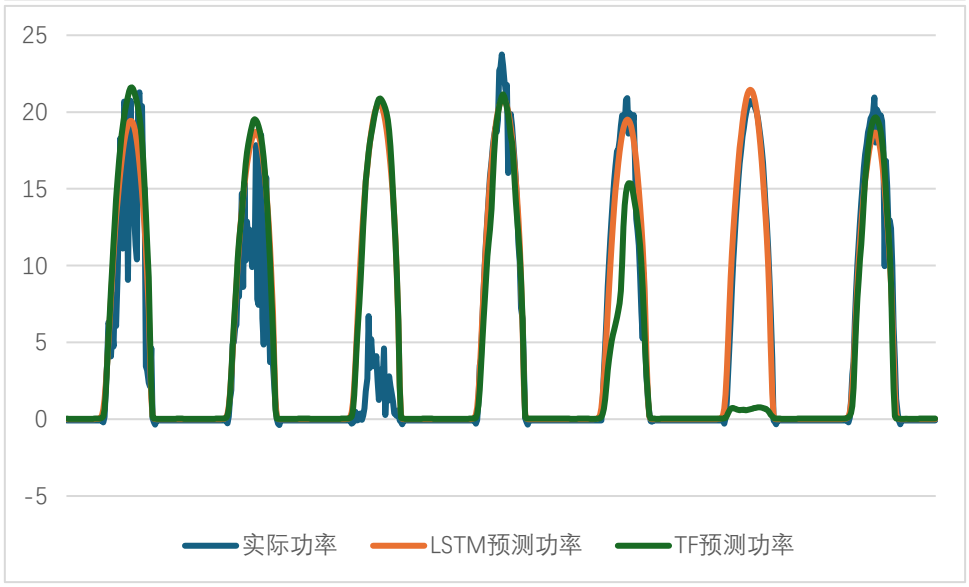




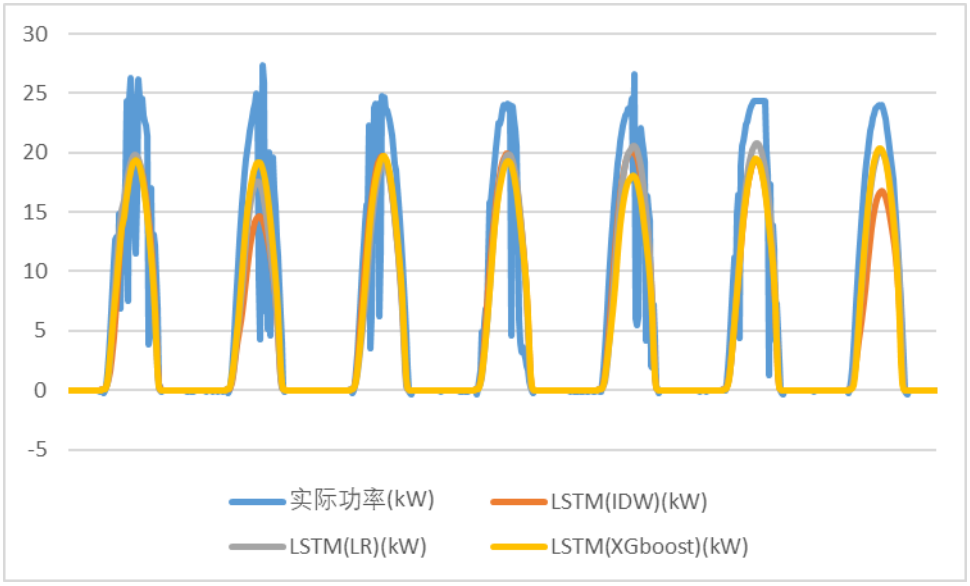
LSTM 与 TF 预测值对比图

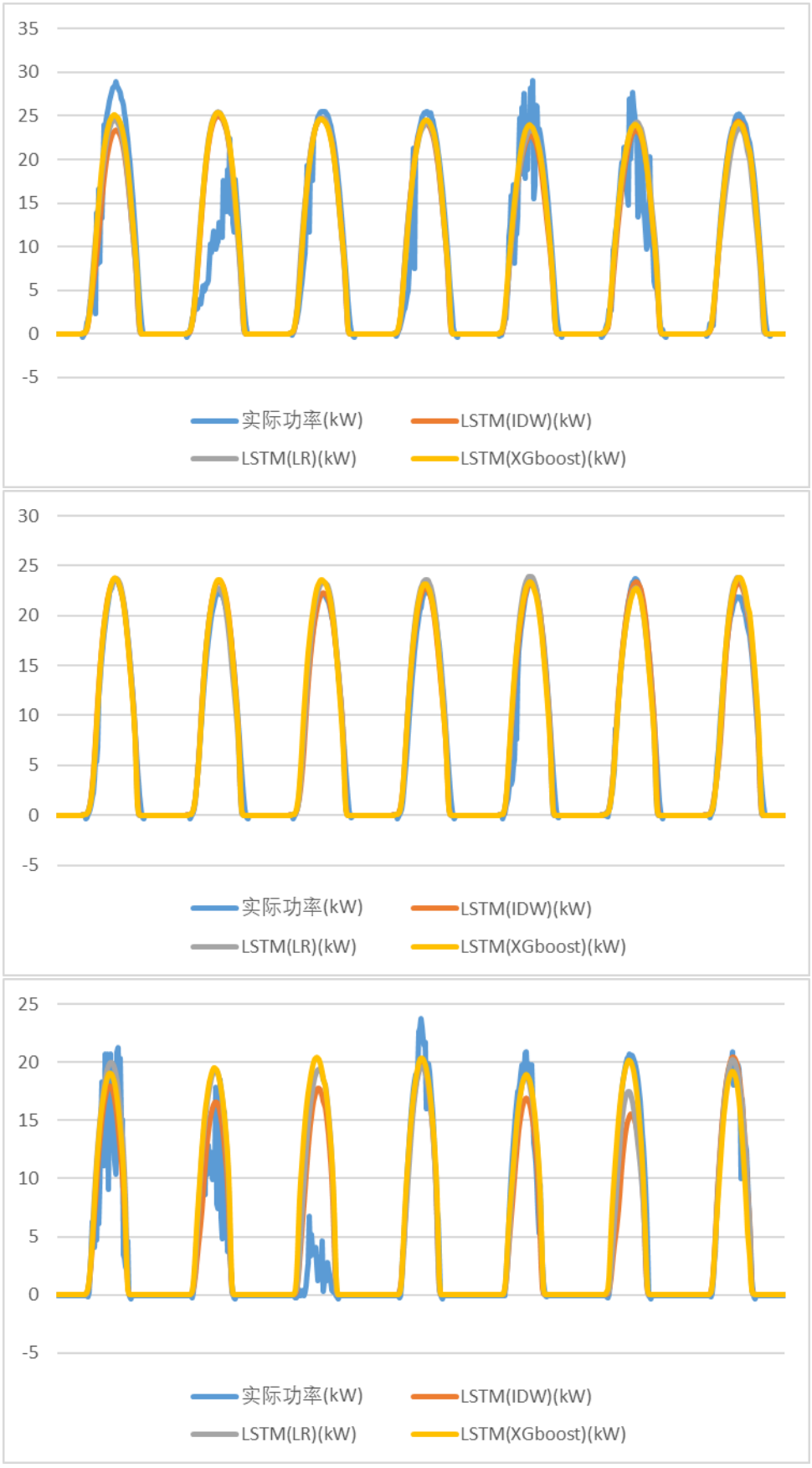




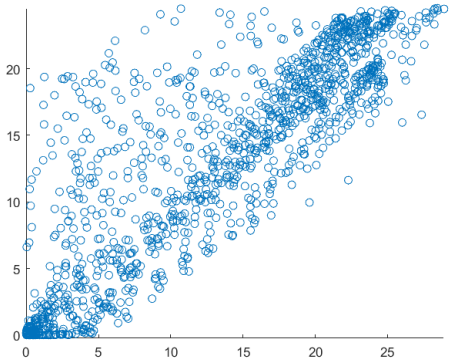


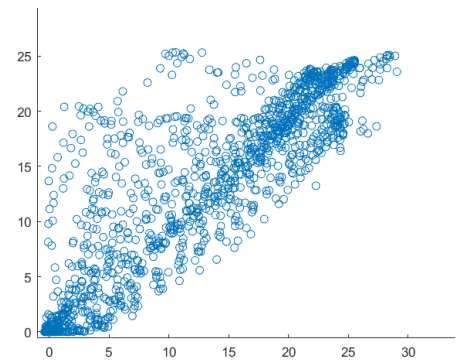
IDM、MLR 与 XGBoost 预测功率对比图

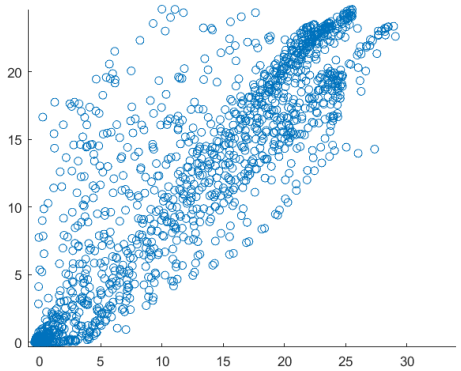


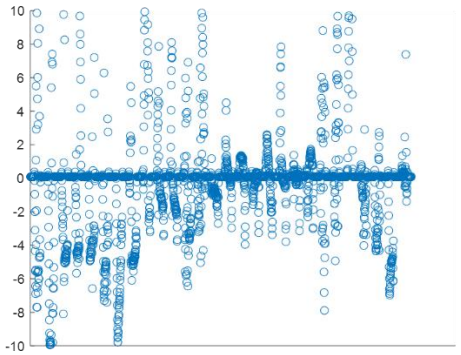


IDM、MLR 与 XGBoost 误差散点图

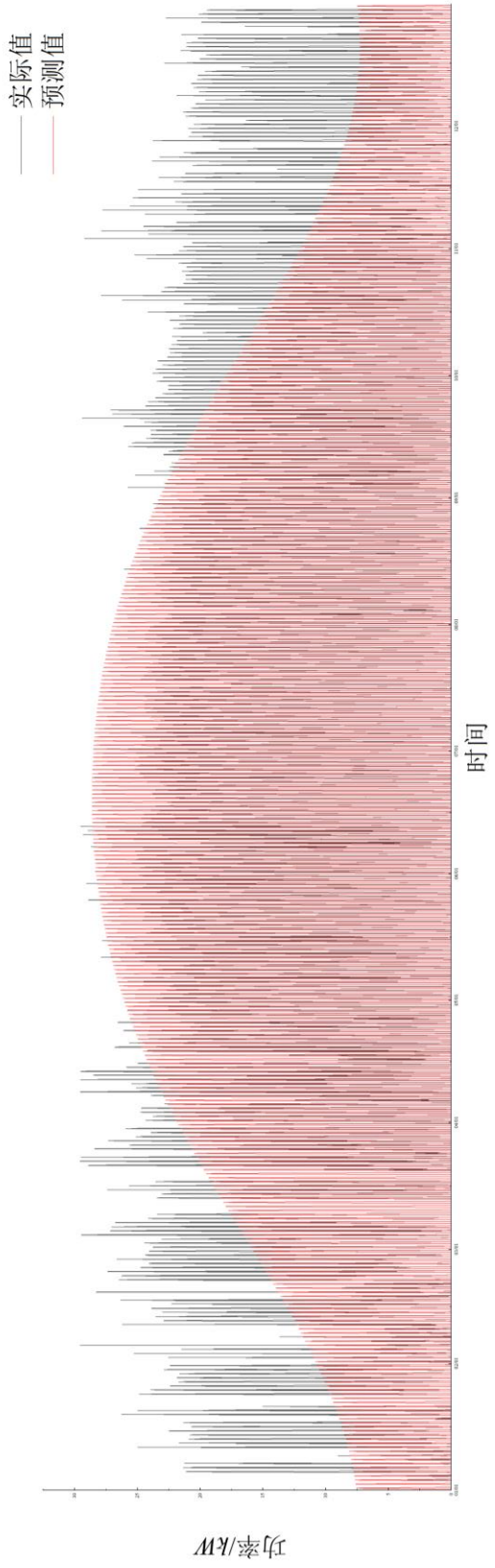

0
5
10
15
20
25


0
5
10
15
20
25
30


0
5
10
15
20
25
30


-10
-8
-6
-4
-2
0
2
4
6
8
10

附录 2



附录 3

问题 1 的核心代码

```
    import numpy as np
    import pandas as pd

    # --- 宇宙常数与默认参数设定 ---
    G_sc = 1367  # 太阳常数 (W/m^2)
    G_STC = 1000 # 标准测试条件下的辐照强度 (W/m^2)
    RHO_G_DEFAULT = 0.2 # 默认地面反射率

    # 默认的晴空大气透明度
    TAU_B_DEFAULT = 0.70 # 直射辐射的经验透射系数
    TAU_D_DEFAULT = 0.15 # 散射辐射的经验透射系数

    def calculate_declination_angle(df_column_dateN):
        """
        计算太阳赤纬角 (δ)，单位是度。
        批量处理 DataFrame 的 dateN 列。

        参数:
        df_column_dateN (pd.Series): 一年中的第几天 (n)。

        返回:
        pd.Series: 太阳赤纬角 (δ)，单位是度。
        """
        n = df_column_dateN
        # 公式：δ = 23.45 * sin(radians(360 * (284 + n) / 365))
        angle_in_degrees = 360.0 * (284.0 + n) / 365.0
                    delta_degrees       =       23.45       *
np.sin(np.radians(angle_in_degrees))
        # 赤纬角
        return delta_degrees

    def calculate_equation_of_time(df_column_dateN):
        """
        计算均时差 (Et)，单位是分钟。
        批量处理 DataFrame 的 dateN 列。

        参数:
        df_column_dateN (pd.Series): 一年中的第几天 (n)。

        返回:
```

```
        pd.Series: 均时差 (Et)，单位是分钟。
        """
        n = df_column_dateN
        # B = (n-1) * 360/365 (in degrees)
        B_deg = (n - 1.0) * 360.0 / 365.0
        B_rad = np.radians(B_deg) # 转换为弧度给 sin/cos 用

          Et_minutes = 229.18 * (0.000075 + 0.001868 *
np.cos(B_rad) - 0.032077 * np.sin(B_rad) \
                                - 0.014615 * np.cos(2 * B_rad)
- 0.040849 * np.sin(2 * B_rad))
        # 均时差!
        return Et_minutes

    def     calculate_hour_angle(df_column_local_time_hour,
df_column_Et_minutes,              df_column_longitude_deg,
df_column_std_meridian_deg):
        """
        计算时角 (w)，单位是度。
        批量处理相关 DataFrame 列。

        参数:
         df_column_local_time_hour (pd.Series): 本地标准时间的
小时部分
        df_column_Et_minutes (pd.Series): 均时差 (分钟)。
        df_column_longitude_deg (pd.Series): 观测点经度 (度)。
         df_column_std_meridian_deg (pd.Series or float): 观
测点所在时区的标准经线 (度)。
                                                    (例如，北京
时间 UTC+8，标准经线是 120 度东经)
                                                     如果所有数
据点都在一个时区，这里可以是一个固定值。

        返回:
        pd.Series: 时角 (w)，单位是度。
        """
        # 真太阳时 (LST) 计算，单位：小时
        # LST = 本地标准时间 + Et/60 + (标准经线 - 实际经度)*4 分
钟/60 转小时
        # (标准经线 - 实际经度)*4 分钟 这个修正项，表示经度每偏一度，
时间差 4 分钟
                          longitude_correction_hours         =
(df_column_std_meridian_deg - df_column_longitude_deg) * 4.0
/ 60.0
```

```
            lst_hours    =    df_column_local_time_hour    +
df_column_Et_minutes / 60.0 + longitude_correction_hours

        # 时角 (ω)，正午为 0，下午为正，上午为负
        omega_degrees = (lst_hours - 12.0) * 15.0
        # 时角!
        return omega_degrees

    def
calculate_solar_altitude_angle(df_column_latitude_deg,
df_column_delta_deg, df_column_omega_deg):
        """
        计算太阳高度角 (αs)，单位是度。
        批量处理相关 DataFrame 列。

        参数:
        df_column_latitude_deg (pd.Series): 纬度 (φ)，单位度。
        df_column_delta_deg (pd.Series): 赤纬角 (δ)，单位度。
        df_column_omega_deg (pd.Series): 时角 (ω)，单位度。

        返回:
        pd.Series: 太阳高度角 (αs)，单位度。
        """
        phi_rad = np.radians(df_column_latitude_deg)
        delta_rad = np.radians(df_column_delta_deg)
        omega_rad = np.radians(df_column_omega_deg)

        sin_alpha_s = np.sin(phi_rad) * np.sin(delta_rad) +
\
                      np.cos(phi_rad) * np.cos(delta_rad) *
np.cos(omega_rad)

        # arcsin 的结果是弧度，再转为度
         alpha_s_rad = np.arcsin(np.clip(sin_alpha_s, -1.0,
1.0)) # clip 确保值在-1 到 1 之间，避免 arcsin 报错
        alpha_s_degrees = np.degrees(alpha_s_rad)

        # 太阳多高
        return alpha_s_degrees

    def
calculate_solar_azimuth_angle(df_column_latitude_deg,
df_column_delta_deg,                df_column_alpha_s_deg,
df_column_omega_deg):
```

```
        """
        计算太阳方位角 (γs)，单位是度。
        (定义：正南为 0 度，东负西正，适用于北半球)
        批量处理相关 DataFrame 列。

        参数:
        df_column_latitude_deg (pd.Series): 纬度 (φ)，单位度。
        df_column_delta_deg (pd.Series): 赤纬角 (δ)，单位度。
        df_column_alpha_s_deg (pd.Series): 太阳高度角 (αs)，单
位度。
        df_column_omega_deg (pd.Series): 时角 (ω)，单位度。

        返回:
        pd.Series: 太阳方位角 (γs)，单位度。
        """
        phi_rad = np.radians(df_column_latitude_deg)
        delta_rad = np.radians(df_column_delta_deg)
        alpha_s_rad = np.radians(df_column_alpha_s_deg)

           cos_gamma_s_numerator  =  np.sin(alpha_s_rad)  *
np.sin(phi_rad) - np.sin(delta_rad)
          cos_gamma_s_denominator  =  np.cos(alpha_s_rad)  *
np.cos(phi_rad)

        # 避免除以 0 的情况 (当太阳在天顶或地平线时，分母可能接近 0)
        # 当 alpha_s 接近 90 度 (天顶) 或 0 度 (地平线) 时，方位角定
义可能不稳定或无意义
        # 我们这里用一个很小的值来避免直接除以 0 的错误
           cos_gamma_s  =  np.divide(cos_gamma_s_numerator,
cos_gamma_s_denominator,
                                  out=np.zeros_like(cos_gamma_s
_numerator),
                                  where=np.abs(cos_gamma_s_deno
minator) > 1e-9) # 仅在分母不为 0 时计算

         gamma_s_rad_abs = np.arccos(np.clip(cos_gamma_s, -
1.0, 1.0)) # clip 确保值在-1 到 1 之间
        gamma_s_degrees_abs = np.degrees(gamma_s_rad_abs)

        # 根据时角 w 的符号判断方位角的符号 (东负西正)
        gamma_s_degrees = np.where(df_column_omega_deg < 0,
-gamma_s_degrees_abs, gamma_s_degrees_abs)

        # 特殊处理：当太阳高度角 <=0 时，方位角无意义，可以设为 NaN
```

```
或 0
        gamma_s_degrees = np.where(df_column_alpha_s_deg <=
0, np.nan, gamma_s_degrees)

        # 太阳在哪个方向
        return gamma_s_degrees

    def
calculate_extraterrestrial_normal_irradiance(df_column_date
N):
        """
        计算大气外层法向太阳辐射 (Gon)，单位是 W/m^2。
        批量处理 DataFrame 的 dateN 列。

        参数:
        df_column_dateN (pd.Series): 一年中的第几天 (n)。

        返回:
        pd.Series: 大气外层法向太阳辐射 (Gon)，单位 W/m^2。
        """
        n = df_column_dateN
        gon = G_sc * (1 + 0.033 * np.cos(np.radians(360.0 *
n / 365.0)))
        # 原始能量!
        return gon

    def      calculate_clear_sky_irradiances(df_column_Gon,
df_column_alpha_s_deg,                  tau_b=TAU_B_DEFAULT,
tau_d=TAU_D_DEFAULT):
        """
        估算晴空条件下的法向直接辐射(Gcnb)和水平面散射辐射(Gcdh)。
        以及水平面总辐射(Gch)。
        这是一个简化的模型，tau_b 和 tau_d 是经验系数。

        参数:
        df_column_Gon (pd.Series): 大气外层法向太阳辐射 (W/m^2)。
        df_column_alpha_s_deg (pd.Series): 太阳高度角 (度)。
        tau_b (float): 直射辐射的经验透射系数。
          tau_d (float): 散射辐射的经验透射系数 (用于 G_on *
sin(alpha_s) 部分)。

        返回:
        tuple: (Gcnb_series, Gcdh_series, Gch_series)
              Gcnb: 法向直接辐射 (W/m^2)
```

```
            Gcdh: 水平面散射辐射 (W/m^2)
            Gch: 水平面总辐射 (W/m^2)
        """
        alpha_s_rad = np.radians(df_column_alpha_s_deg)
        sin_alpha_s = np.sin(alpha_s_rad)

        # 太阳在地平线以下时，辐射为 0
        is_sun_up = df_column_alpha_s_deg > 0

        Gcnb = df_column_Gon * tau_b * is_sun_up

        # Gcdh ≈ Gon * sin(αs) * τd  (基于题目给的简化思路)
        # 注意: 这里的 Gon * sin(αs) 是大气外层水平面辐射
          Gcdh = df_column_Gon * sin_alpha_s * tau_d *
is_sun_up
        Gcdh = np.maximum(0, Gcdh) # 确保非负

        # Gch = Gcnb * sin(αs) + Gcdh
        Gch = Gcnb * sin_alpha_s + Gcdh
         Gch = np.maximum(0, Gch * is_sun_up) # 确保非负且太阳
升起时才有

        # 地面上的晴空阳光分量!
        return Gcnb, Gcdh, Gch

    def             calculate_poa_irradiance(df_column_Gcnb,
df_column_Gcdh, df_column_Gch,
                                             df_column_alpha_s_deg,
df_column_gamma_s_deg,
                                  df_column_panel_tilt_beta_de
g, df_column_panel_azimuth_gamma_deg,
                                 rho_g=RHO_G_DEFAULT):
        """
          计算到达倾斜光伏板的总辐射 (GT - POA Irradiance)，单位
W/m^2。
        批量处理相关 DataFrame 列。

        参数:
        df_column_Gcnb (pd.Series): 法向直接辐射 (W/m^2)。
        df_column_Gcdh (pd.Series): 水平面散射辐射 (W/m^2)。
        df_column_Gch (pd.Series): 水平面总辐射 (W/m^2)。
        df_column_alpha_s_deg (pd.Series): 太阳高度角 (度)。
        df_column_gamma_s_deg (pd.Series): 太阳方位角 (度)。
         df_column_panel_tilt_beta_deg (pd.Series or float):
```

```
光伏板倾角 (β)，单位度。
          df_column_panel_azimuth_gamma_deg (pd.Series or
float): 光伏板方位角 (γ)，单位度。
        rho_g (float): 地面反射率。

        返回:
        pd.Series: 倾斜面总辐射 (GT)，单位 W/m^2。
        """
        alpha_s_rad = np.radians(df_column_alpha_s_deg)
        gamma_s_rad = np.radians(df_column_gamma_s_deg)
        beta_rad = np.radians(df_column_panel_tilt_beta_deg)
                                panel_gamma_rad            =
np.radians(df_column_panel_azimuth_gamma_deg)

        # 计算入射角余弦 cos(θ)
        cos_theta = np.sin(alpha_s_rad) * np.cos(beta_rad) +
\
                    np.cos(alpha_s_rad) * np.sin(beta_rad) *
np.cos(gamma_s_rad - panel_gamma_rad)
         cos_theta = np.maximum(0, cos_theta) # 确保非负，太阳
在板子背面则为 0

        # 太阳在地平线以下或入射角>90 度时，直接辐射为 0
         is_sun_up_on_panel = (df_column_alpha_s_deg > 0) &
(cos_theta > 1e-6) # 加个小阈值避免浮点问题

        # 倾斜面直接辐射 (GbT)
        GbT = df_column_Gcnb * cos_theta * is_sun_up_on_panel

        # 倾斜面天空散射辐射 (GdT) - Liu & Jordan 各向同性模型
        GdT = df_column_Gcdh * (1 + np.cos(beta_rad)) / 2.0
* (df_column_alpha_s_deg > 0) # 仅当太阳升起

        # 倾斜面地面反射辐射 (GgT)
        GgT = df_column_Gch * rho_g * (1 - np.cos(beta_rad))
/ 2.0 * (df_column_alpha_s_deg > 0) # 仅当太阳升起

        # 倾斜面总辐射 (GT)
        GT = GbT + GdT + GgT
        GT = np.maximum(0, GT) # 确保最终辐射非负

        # 光伏板实际感受到的阳光强度
        return GT
```

```
    def calculate_theoretical_power(df_column_GT_poa,
df_column_P_install):
        """
        计算理论可发功率 (P_theoretical)，单位与 P_install 一致。
        批量处理相关 DataFrame 列。

        参数:
         df_column_GT_poa (pd.Series): 倾斜面总辐射 (GT)，单位
W/m^2。
        df_column_P_install (pd.Series or float): 电站装机容量
(单位例如 W 或 kW 或 MW)。

        返回:
          pd.Series: 理论可发功率 (P_theoretical)，单位与
P_install 一致。
        """
        # P_theoretical = P_install * (GT / G_STC)
         # 注意 P_install 的单位，如果 P_install 是 MW，而 GT 是
W/m^2, G_STC 是 W/m^2，
        # 那么 P_theoretical 也会是 MW。
             P_theoretical    =    df_column_P_install    *
(df_column_GT_poa / G_STC) / 1000000

        # 理论功率不应超过装机容量 (作为一种简单的上限约束)
        # 并且确保非负
            P_theoretical   =   np.clip(P_theoretical,   0,
df_column_P_install)

        # 电站理想中的发电量
        return P_theoretical

    # --- 主协调函数 ---
    def run_problem1_calculations(df,
                                  dateN_col='dateN',
                                  latitude_col='Dimensions',
                                  longitude_col='longitude',
                                  panel_tilt_col='beta',
                                  panel_azimuth_col='gamma',
                                  p_install_col='P_install',
                                   local_time_hour_col='local_
time_hour_of_day', # 需要根据 'date' 列创建这一列
                                   std_meridian_deg=120.0, # 假
设一个标准经线，比如中国东八区 120 度
                                  tau_b=TAU_B_DEFAULT,
```

```
                                    tau_d=TAU_D_DEFAULT,
                                    rho_g=RHO_G_DEFAULT
                                ):
        """
          执行问题一的所有计算步骤，并将中间结果和最终理论功率添加到
DataFrame 中。

        参数:
        df (pd.DataFrame): 包含所需输入数据的 DataFrame。
                           必须包含以下列名 (或通过参数指定列名):
                           - dateN_col: 一年中的第几天 (n)
                           - latitude_col: 纬度 (φ)，度
                            - longitude_col: 经度 (λ)，度 (如果
缺失会按 0 处理，提前填充)
                           - panel_tilt_col: 光伏板倾角 (β)，度
                             - panel_azimuth_col: 光伏板方位角
(γ)，度 (如果缺失会按 0 处理，提前填充)
                                      - p_install_col: 装机容量
(P_install)
                            - local_time_hour_col: 本地标准时间
的小时值 (0-23.99)
        std_meridian_deg (float): 时区标准经线，度。
        tau_b, tau_d, rho_g (float): 大气透明度及地面反射率参数。

        返回:
           pd.DataFrame: 增加了所有计算列（包括最终的
'P_theoretical'）的原始 DataFrame。
                         新增列名示例: 'delta_deg', 'Et_minutes',
'omega_deg', 'alpha_s_deg',
                                       'gamma_s_deg', 'Gon_Wm2',
'Gcnb_Wm2', 'Gcdh_Wm2',
                                       'Gch_Wm2', 'GT_poa_Wm2',
'P_theoretical'
        """
        # 确保经度和面板方位角缺失时按 0 处理
        # df[longitude_col] = df[longitude_col].fillna(0)
                        #        df[panel_azimuth_col]        =
df[panel_azimuth_col].fillna(0)

        # 0 时间与基础角度换算
                                    df['delta_deg']             =
calculate_declination_angle(df[dateN_col])
                                   df['Et_minutes']             =
calculate_equation_of_time(df[dateN_col])
```

```
                                df['omega_deg']             =
calculate_hour_angle(df[local_time_hour_col],
df['Et_minutes'], df[longitude_col], std_meridian_deg)

        # 1 太阳在天空中的位置
                               df['alpha_s_deg']            =
calculate_solar_altitude_angle(df[latitude_col],
df['delta_deg'], df['omega_deg'])
                               df['gamma_s_deg']            =
calculate_solar_azimuth_angle(df[latitude_col],
df['delta_deg'], df['alpha_s_deg'], df['omega_deg'])

        # 2 晴空下的太阳辐射强度
                                 df['Gon_Wm2']              =
calculate_extraterrestrial_normal_irradiance(df[dateN_col])
          df['Gcnb_Wm2'],  df['Gcdh_Wm2'],  df['Gch_Wm2']  =
calculate_clear_sky_irradiances(
             df['Gon_Wm2'], df['alpha_s_deg'], tau_b=tau_b,
tau_d=tau_d
        )

        # 3 阳光洒在倾斜的光伏板上
        df['GT_poa_Wm2'] = calculate_poa_irradiance(
            df['Gcnb_Wm2'], df['Gcdh_Wm2'], df['Gch_Wm2'],
            df['alpha_s_deg'], df['gamma_s_deg'],
            df[panel_tilt_col], df[panel_azimuth_col],
            rho_g=rho_g
        )

        # 4 理论发电功率的诞生
                                 df['p_logic']              =
calculate_theoretical_power(df['GT_poa_Wm2'],
df[p_install_col])

        print("DataFrame 已经更新！")
        return df

    def helloworld():
      print("Hello World!")

    if __name__ == "__main__":
        try:
            df1 = pd.read_csv("testproblem1.csv")
              # 确保 'local_time_hour_of_day' 存在，如果它是由
```

```
'date' 列生成的
                # 例如： df1['local_time_hour_of_day'] =
pd.to_datetime(df1['date']).dt.hour                          +
pd.to_datetime(df1['date']).dt.minute / 60.0
          # 确保所有需要的输入列都存在于 df1 中

           # 调用计算函数（假设 std_meridian_deg 等参数使用默认
值）
                                          df_calculated       =
run_problem1_calculations(df1.copy()) # 使用 .copy() 避免
SettingWithCopyWarning

          # 保存包含计算结果的 DataFrame 到新的 Excel 文件
          df_calculated.to_csv("testproblem1_with_calculat
ions.csv", index=False)
                    print("\n 已将计算结果保存到
testproblem1_with_calculations.csv")

       except FileNotFoundError:
           print(f"找不到 testproblem1.csv 文件，请检查文件路径
和名称")
       except KeyError as e:
            print(f"DataFrame 中好像缺少了必要的列： {e}，请检查
Excel 文件的列名是否与代码期望的一致")
       except Exception as e:
           print(f"发生了一个意料之外的错误：{e}")
```

问题 2 LR 的核心代码

```
   import pandas as pd
   import numpy as np
   from sklearn.linear_model import LinearRegression
   from sklearn.preprocessing import MinMaxScaler
   from sklearn.metrics import mean_squared_error # 用于后续
可能的评估
   import joblib # 用于保存和加载模型/scaler
   import os # 用于检查文件是否存在

   # --- 文件路径和重要的全局设定 ---
   PROCESSED_EXCEL_PATH = 'problem2_processed.xlsx' # 上一步
处理结果的 Excel 文件
   TRAIN_SHEET_NAME = 'train'
   TEST_SHEET_NAME = 'test'
   OUTPUT_EXCEL_PATH = 'problem2_forecast_results.xlsx' #
```

保存最终预测结果的新 Excel 文件

```
    # 模型和 Scaler 保存路径
    MODEL_PATH = 'linear_regression_model.joblib'
    SCALER_PATH = 'min_max_scaler.joblib'

    # 假设这是模型训练时最终确定的特征名称列表和顺序
    # 这些特征名需要在特征工程步骤中被创建出来
    FEATURE_NAMES_IN_ORDER = [
            'power_lag_1',   'power_lag_2',   'power_lag_3',
'power_lag_4', # 最近 1 小时
        'power_lag_96', # 1 天前同一时刻
        'power_lag_192', # 2 天前同一时刻
        'power_lag_672', # 1 周前同一时刻
        'hour_sin', 'hour_cos',
        'dayN_sin', 'dayN_cos',
        'dayofweek_sin', 'dayofweek_cos' # 假设加入了星期特征
    ]
    # 根据上面的特征，确定我们实际用到的滞后项是哪些数字
    # 例如，如果 'power_lag_1' 存在，那么 1 就在列表里
    SPECIFIC_LAGS_USED = [1, 2, 3, 4, 96, 192, 672] # 与
FEATURE_NAMES_IN_ORDER 中的 power_lag_x 对应
    MAX_LAG = max(SPECIFIC_LAGS_USED) if SPECIFIC_LAGS_USED
else 0

    def                                    create_features(df,
target_col_name='power_normalized', is_training=True):
        """
        为 DataFrame 创建滞后特征和时间特征。
        如果是训练，目标列 y 也会被创建（即当前的 target_col_name）。
        如果是为预测准备单步输入，则 target_col_name 不用于 y。
        """
           print(f"  正 在 为 数 据 打 造 特 征 积 木 块   ( 基 于
'{target_col_name}')...")
        df_featured = df.copy()

        # a. 创建滞后特征
        # SPECIFIC_LAGS_USED 定义了我们要创建哪些滞后项
        for lag in SPECIFIC_LAGS_USED:
                        df_featured[f'power_lag_{lag}']    =
df_featured[target_col_name].shift(lag)

        # b. 创建时间特征
        # 确保索引是 DatetimeIndex
```

```
                if not isinstance(df_featured.index,
pd.DatetimeIndex):
                 raise ValueError("DataFrame 的索引必须是
DatetimeIndex 才能创建时间特征！")

                 df_featured['local_time_hour_of_day'] =
df_featured.index.hour + df_featured.index.minute / 60.0
        df_featured['dateN'] = df_featured.index.dayofyear
                        df_featured['dayofweek'] =
df_featured.index.dayofweek # Monday=0, Sunday=6

          df_featured['hour_sin'] = np.sin(2 * np.pi *
df_featured['local_time_hour_of_day'] / 24.0)
          df_featured['hour_cos'] = np.cos(2 * np.pi *
df_featured['local_time_hour_of_day'] / 24.0)
          df_featured['dayN_sin'] = np.sin(2 * np.pi *
df_featured['dateN'] / 365.25)
          df_featured['dayN_cos'] = np.cos(2 * np.pi *
df_featured['dateN'] / 365.25)
         df_featured['dayofweek_sin'] = np.sin(2 * np.pi *
df_featured['dayofweek'] / 7.0)
         df_featured['dayofweek_cos'] = np.cos(2 * np.pi *
df_featured['dayofweek'] / 7.0)

        if is_training:
                # 定义目标变量 y (对于训练，y 就是当前的
'power_normalized')
                              df_featured['y_target'] =
df_featured[target_col_name]
            # 清理因滞后产生的 NaN 值 (仅在训练和特征准备时)
            df_featured.dropna(inplace=True) # 确保所有特征行
和目标行都有值

        # 确保所有在 FEATURE_NAMES_IN_ORDER 中的列都存在
        for col_name in FEATURE_NAMES_IN_ORDER:
            if col_name not in df_featured.columns:
                # 这是一个重要检查，如果模型训练的特征这里没生成，后
面会出错
                    raise KeyError(f"特征工程后，期望的特征列
'{col_name}' 不存在！请检查 create_features 函数和
FEATURE_NAMES_IN_ORDER。")

        print("特征积木块打造完成！")
        return df_featured
```

```
    def get_initial_history(df_train_power_normalized_col,
                            test_period_start_time,
                            max_lag_needed):
        """获取启动迭代预测所需的初始历史归一化功率序列。"""
        # (此函数与之前的版本类似，确保它能正确工作)
            history_end_time   =   test_period_start_time   -
pd.Timedelta(minutes=15)
                                relevant_train_data           =
df_train_power_normalized_col[df_train_power_normalized_col
.index <= history_end_time] # 使用 <= 以包含边界
        if len(relevant_train_data) < max_lag_needed:
                    raise   ValueError(f" 训 练 数 据 中 在
{test_period_start_time} 之前的数据不足 {max_lag_needed} 条
({len(relevant_train_data)}条)，无法提供足够的初始历史！")
            initial_history   =   relevant_train_data.iloc[-
max_lag_needed:].tolist()
        return initial_history

    def create_time_features_for_timestamp(timestamp):
        """为单个未来的时间戳计算所有定义好的时间特征。"""
        features = {}
         local_time_hour = timestamp.hour + timestamp.minute
/ 60.0
        day_of_year = timestamp.dayofyear
        day_of_week = timestamp.dayofweek

            features['hour_sin']   =   np.sin(2   *   np.pi   *
local_time_hour / 24.0)
            features['hour_cos']   =   np.cos(2   *   np.pi   *
local_time_hour / 24.0)
        features['dayN_sin'] = np.sin(2 * np.pi * day_of_year
/ 365.25)
        features['dayN_cos'] = np.cos(2 * np.pi * day_of_year
/ 365.25)
          features['dayofweek_sin']  =  np.sin(2  *  np.pi  *
day_of_week / 7.0)
          features['dayofweek_cos']  =  np.cos(2  *  np.pi  *
day_of_week / 7.0)
        return features

    def
construct_feature_vector_for_step(current_power_history,
                                          time_features_for_th
```

```
is_step,
                                                all_feature_names_or
dered,
                                                specific_lags_indice
s):
        """根据当前功率历史和时间特征，构建模型所需的单行特征向量。
"""
       # (此函数与之前的版本类似，确保它能正确工作)
       feature_dict = {}
       for lag_val in specific_lags_indices:
           feature_name = f'power_lag_{lag_val}'
           if feature_name in all_feature_names_ordered:
               if len(current_power_history) >= lag_val:
                                feature_dict[feature_name]  =
current_power_history[-lag_val]
               else:
                    raise ValueError(f"历史序列不够长，无法获取
{feature_name}")
       feature_dict.update(time_features_for_this_step)

       feature_vector_values = [feature_dict[name] for name
in all_feature_names_ordered]
        return np.array(feature_vector_values).reshape(1, -
1)

   def  perform_iterative_forecast_for_period(trained_model,
                                               initial_power_hi
story,
                                               period_start_tim
e,
                                               num_steps_to_for
ecast,
                                               feature_names_or
dered_list,
                                              max_lag_val,
                                               specific_lags_id
x_list):
       """为一个特定的 7 天周期执行迭代预测。"""
       # (此函数与之前的版本类似，确保它能正确工作)
       predictions_normalized = []
       current_history = list(initial_power_history)
       current_time = pd.Timestamp(period_start_time)

          print(f" 开 始 为 周 期   {period_start_time}  进 行
```

```
{num_steps_to_forecast} 步迭代预测...")
        for i in range(num_steps_to_forecast):
                                            time_features        =
create_time_features_for_timestamp(current_time)
                                           feature_vector        =
construct_feature_vector_for_step(
                current_history,
                time_features,
                feature_names_ordered_list,
                specific_lags_idx_list
            )
                                 predicted_norm_power_step     =
trained_model.predict(feature_vector)[0]
            predictions_normalized.append(predicted_norm_pow
er_step)

            current_history.pop(0)
            current_history.append(predicted_norm_power_step)
            current_time += pd.Timedelta(minutes=15)
            if (i + 1) % 96 == 0:
                 print(f"  已完成第 {(i + 1) // 96} 天的预测 (共
{num_steps_to_forecast//96}天)...")
        print(f"周期 {period_start_time} 预测完成！")
        return predictions_normalized

    def  inverse_transform_predictions(predictions_norm_list,
fitted_scaler):
        """将归一化的预测列表逆转换为原始尺度。"""
                                    predictions_2d             =
np.array(predictions_norm_list).reshape(-1, 1)
                          predictions_actual_scale_2d          =
fitted_scaler.inverse_transform(predictions_2d)
        return predictions_actual_scale_2d.flatten().tolist()

    # --- 主程序逻辑 ---
    def main():
        """完整执行 问题二 的数据处理、模型训练、预测与保存流程"""
        try:
            # 1. 读取处理好的训练集和测试集数据
                 print(f"  正在读取已处理的 Excel 文件：
'{PROCESSED_EXCEL_PATH}'...")
            if not os.path.exists(PROCESSED_EXCEL_PATH):
                    print(f"重要提示！找不到上一步处理好的文件
'{PROCESSED_EXCEL_PATH}'。")
```

```
                print("这个脚本期望您已经运行了数据清洗和归一化，并
将结果保存为该文件。")
                print("如果还没有，请先运行相应的预处理步骤")
                return

                                        df_train_loaded       =
pd.read_excel(PROCESSED_EXCEL_PATH,
sheet_name=TRAIN_SHEET_NAME)
                                         df_test_loaded       =
pd.read_excel(PROCESSED_EXCEL_PATH,
sheet_name=TEST_SHEET_NAME)

            # 确保 'date' 列是 datetime 类型并设为索引
            for df_ in [df_train_loaded, df_test_loaded]:
                if 'date' not in df_.columns:
                    raise KeyError("'date' 列在 Excel 中找不到，
请检查列名！")
                df_['date'] = pd.to_datetime(df_['date'])
                df_.set_index('date', inplace=True)

            print("训练集和测试集数据已载入并设置好时间索引。")
             print(f"原始训练集大小: {len(df_train_loaded)}, 原
始测试集大小: {len(df_test_loaded)}")

            # 2. 特征工程 (在训练集上)
             # 我们需要 'power_normalized' 列进行滞后项创建和作为
目标
                        if    'power_normalized'    not    in
df_train_loaded.columns:
                         raise  KeyError(" 训 练 集 中 找 不 到
'power_normalized' 列，无法进行特征工程和训练！")

                                      df_train_featured       =
create_features(df_train_loaded,
target_col_name='power_normalized', is_training=True)

            if df_train_featured.empty:
                print("特征工程后训练数据为空，可能是滞后项设置过大
或数据不足。程序中止。")
                return

                                             X_train          =
df_train_featured[FEATURE_NAMES_IN_ORDER]
                y_train  =  df_train_featured['y_target']  #
```

```
'y_target' 是 'power_normalized'

            # 3. 训练线性回归模型
            print("\n 开始训练线性回归模型...")
            linear_model = LinearRegression()
            linear_model.fit(X_train, y_train)
            print("线性回归模型训练完成！")
            # (可选) 保存模型
            joblib.dump(linear_model, MODEL_PATH)
            print(f"模型已保存到: {MODEL_PATH}")

            # 4. 拟合 Scaler (在训练集的 'power_cleaned' 列上)
                #       我 们 假 设   'power_cleaned'   列 存 在 于
df_train_loaded 中，用于确定原始数据的范围
            if 'power_cleaned' not in df_train_loaded.columns:
                raise KeyError("训练集中找不到 'power_cleaned'
列，无法拟合 Scaler 以进行逆转换！")

              scaler_to_fit = MinMaxScaler(feature_range=(0,
1))
            # Scaler 在 fit 时期望输入是 2D 的
            scaler_to_fit.fit(df_train_loaded[['power_cleane
d']]) # 用清洗后、归一化前的数据范围来 fit
            print("MinMaxScaler 已在训练集的 'power_cleaned' 数
据上拟合完成！")
            # (可选) 保存 scaler
            joblib.dump(scaler_to_fit, SCALER_PATH)
            print(f"Scaler 已保存到: {SCALER_PATH}")

            # --- 开始对测试集进行预测 ---
            df_test_predictions = df_test_loaded.copy() # 创
建副本以添加预测列
            df_test_predictions['predicted_power_normalized']
= np.nan
             df_test_predictions['predicted_power_actual'] =
np.nan

            # 识别测试周期
                                          time_diffs        =
df_test_predictions.index.to_series().diff()
                                  period_starts_indices     =
df_test_predictions.index[(time_diffs                       >
pd.Timedelta(minutes=15)) | (time_diffs.isnull())]
                      print(f"\n   在   测   试   集   中   识   别   到
```

```
{len(period_starts_indices)} 个预测周期的开始点。")

            num_steps_per_period = 7 * 24 * 4

            for start_dt_of_period in period_starts_indices:
                print(f"\n--- 正在为从 {start_dt_of_period} 开
始的 7 天周期进行预测 ---")

                initial_history_norm = get_initial_history(
                    df_train_loaded['power_normalized'], # 用
训练集的归一化功率作历史
                    start_dt_of_period,
                    MAX_LAG
                )

                        if not initial_history_norm or
len(initial_history_norm) < MAX_LAG:
                    print(f"  无法为 {start_dt_of_period} 获取
足够的初始历史，跳过此周期。")
                    continue

                            normalized_preds_for_period =
perform_iterative_forecast_for_period(
                    linear_model, # 使用我们刚训练好的模型
                    initial_history_norm,
                    start_dt_of_period,
                    num_steps_per_period,
                    FEATURE_NAMES_IN_ORDER,
                    MAX_LAG,
                    SPECIFIC_LAGS_USED
                )

                # 确定这些预测对应的测试集中的时间戳范围
                            target_timestamps_in_test =
df_test_predictions.loc[start_dt_of_period :
start_dt_of_period + pd.Timedelta(days=7) -
pd.Timedelta(minutes=15)].index

                    if len(normalized_preds_for_period) ==
len(target_timestamps_in_test):
                    df_test_predictions.loc[target_timestamp
s_in_test, 'predicted_power_normalized'] =
normalized_preds_for_period
```

```
                    # 进行逆向归一化
                            actual_scale_preds_for_period  =
inverse_transform_predictions(
                        normalized_preds_for_period,
                        scaler_to_fit # 使用我们刚拟合的 scaler
                    )
                     df_test_predictions.loc[target_timestamp
s_in_test,            'predicted_power_actual']           =
actual_scale_preds_for_period
                    print(f"  周期 {start_dt_of_period} 的预测
已填充。")
                else:
                     print(f"  警告！周期 {start_dt_of_period}
的预测长度 ({len(normalized_preds_for_period)}) 与测试集中对应
时段长度 ({len(target_timestamps_in_test)}) 不匹配！")

            # 保存包含预测结果的整个测试集(和原始训练集)到新的 Excel
文件
                print(f"\n  正在将预测结果保存到 Excel 文件：
'{OUTPUT_EXCEL_PATH}'...")
            with pd.ExcelWriter(OUTPUT_EXCEL_PATH) as writer:
                df_train_loaded.reset_index().to_excel(write
r, sheet_name=TRAIN_SHEET_NAME, index=False)
                df_test_predictions.reset_index().to_excel(w
riter, sheet_name=TEST_SHEET_NAME, index=False)
                    print(f" 预 测 结 果 已 成 功 保 存 ！ 看
'{OUTPUT_EXCEL_PATH}' ！")

        except FileNotFoundError:
            print(f"找不到 Excel 文件 '{PROCESSED_EXCEL_PATH}',
请确保它和脚本在同一目录，或者路径正确！")
        except KeyError as e:
             print(f"DataFrame 中好像缺少了名为 '{e}' 的列，请检
查 Excel 文件中的列名，或者确保模型训练的特征都存在！")
             import traceback
             traceback.print_exc()
        except Exception as e:
            print(f"发生了一个意料之外的错误：{e}")
            import traceback
            traceback.print_exc()

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

问题 2 SVR 的核心代码

```
    import pandas as pd
    import numpy as np
    from sklearn.svm import SVR
    from    sklearn.preprocessing    import    MinMaxScaler,
StandardScaler # MinMaxScaler 用于目标逆转换，StandardScaler 用
于特征缩放
    from sklearn.metrics import mean_squared_error # 用于后续
可能的评估
    import joblib # 用于保存和加载模型/scaler
    import os # 用于检查文件是否存在
    import time # 用于计时 SVR 训练过程

    # --- 文件路径和重要的全局设定 ---
    PROCESSED_EXCEL_PATH = 'problem2_processed.xlsx' # 上一步
处理结果的 Excel 文件
    TRAIN_SHEET_NAME = 'train'
    TEST_SHEET_NAME = 'test'
    OUTPUT_EXCEL_PATH = 'problem2_SVR_forecast_results.xlsx'
# 保存 SVR 预测结果的新 Excel 文件

    # 模型和 Scaler 保存路径
    SVR_MODEL_PATH = 'svr_model.joblib'
    FEATURE_SCALER_SVR_PATH = 'feature_scaler_svr.joblib' #
SVR 输入特征的 scaler
    TARGET_SCALER_PATH = 'target_min_max_scaler.joblib' # 之
前用于 power_normalized 的 scaler (如果已保存)

    # --- 特征工程和模型参数设定 (与线性回归部分保持一致，以便比较)
---
    FEATURE_NAMES_IN_ORDER = [
            'power_lag_1',   'power_lag_2',   'power_lag_3',
'power_lag_4',
        'power_lag_96',
        'power_lag_192',
        'power_lag_672',
        'hour_sin', 'hour_cos',
        'dayN_sin', 'dayN_cos',
        'dayofweek_sin', 'dayofweek_cos'
    ]
    SPECIFIC_LAGS_USED = [1, 2, 3, 4, 96, 192, 672]
    MAX_LAG = max(SPECIFIC_LAGS_USED) if SPECIFIC_LAGS_USED
```

```
else 0

    # --- Helper Functions (与线性回归脚本中的类似，稍作调整或复用) ---

    def create_features_svr(df, target_col_name_for_lags='power_normalized', is_training=True):
        """
        为 DataFrame 创建滞后特征和时间特征 (与线性回归的特征工程一致)。
        """
        print(f"  正在为 SVR 打造特征积木块 (基于 '{target_col_name_for_lags}')...")
        df_featured = df.copy()

        for lag in SPECIFIC_LAGS_USED:
            df_featured[f'power_lag_{lag}'] = df_featured[target_col_name_for_lags].shift(lag)

        if not isinstance(df_featured.index, pd.DatetimeIndex):
            raise ValueError("DataFrame 的索引必须是 DatetimeIndex! ")

        df_featured['local_time_hour_of_day'] = df_featured.index.hour + df_featured.index.minute / 60.0
        df_featured['dateN'] = df_featured.index.dayofyear
        df_featured['dayofweek'] = df_featured.index.dayofweek

        df_featured['hour_sin'] = np.sin(2 * np.pi * df_featured['local_time_hour_of_day'] / 24.0)
        df_featured['hour_cos'] = np.cos(2 * np.pi * df_featured['local_time_hour_of_day'] / 24.0)
        df_featured['dayN_sin'] = np.sin(2 * np.pi * df_featured['dateN'] / 365.25)
        df_featured['dayN_cos'] = np.cos(2 * np.pi * df_featured['dateN'] / 365.25)
        df_featured['dayofweek_sin'] = np.sin(2 * np.pi * df_featured['dayofweek'] / 7.0)
        df_featured['dayofweek_cos'] = np.cos(2 * np.pi * df_featured['dayofweek'] / 7.0)
```

```
        if is_training:
                                df_featured['y_target']      =
df_featured[target_col_name_for_lags]
            df_featured.dropna(inplace=True)

        for col_name in FEATURE_NAMES_IN_ORDER:
            if col_name not in df_featured.columns:
                    raise  KeyError(f"特征工程后，期望的特征列
'{col_name}' 不存在！")

        print("SVR 的特征积木块打造完成！")
        return df_featured

    def
get_initial_history_svr(df_train_power_normalized_col,
                                test_period_start_time,
                                max_lag_needed):
        """获取启动迭代预测所需的初始历史归一化功率序列。"""
        # (与线性回归版本相同)
            history_end_time   =   test_period_start_time   -
pd.Timedelta(minutes=15)
                               relevant_train_data           =
df_train_power_normalized_col[df_train_power_normalized_col
.index <= history_end_time]
        if len(relevant_train_data) < max_lag_needed:
                    raise   ValueError(f" 训 练 数 据 中 在
{test_period_start_time} 之前的数据不足 {max_lag_needed} 条
({len(relevant_train_data)}条)，无法提供足够的初始历史！")
             initial_history   =   relevant_train_data.iloc[-
max_lag_needed:].tolist()
        return initial_history

    def create_time_features_for_timestamp_svr(timestamp):
        """为单个未来的时间戳计算所有定义好的时间特征。"""
        # (与线性回归版本相同)
        features = {}
         local_time_hour = timestamp.hour + timestamp.minute
/ 60.0
        day_of_year = timestamp.dayofyear
        day_of_week = timestamp.dayofweek

            features['hour_sin']   =   np.sin(2   *   np.pi   *
local_time_hour / 24.0)
            features['hour_cos']   =   np.cos(2   *   np.pi   *
```

```
local_time_hour / 24.0)
        features['dayN_sin'] = np.sin(2 * np.pi * day_of_year
/ 365.25)
        features['dayN_cos'] = np.cos(2 * np.pi * day_of_year
/ 365.25)
          features['dayofweek_sin'] = np.sin(2 * np.pi *
day_of_week / 7.0)
          features['dayofweek_cos'] = np.cos(2 * np.pi *
day_of_week / 7.0)
        return features

    def
construct_feature_vector_for_step_svr(current_power_history,
                                          time_features_fo
r_this_step,
                                          all_feature_name
s_ordered,
                                          specific_lags_in
dices):
         """根据当前功率历史和时间特征，构建单行特征字典 (后续会转为
DataFrame 进行缩放)。"""
        # (与线性回归版本类似，但返回字典，方便后续构造成 DataFrame)
        feature_dict = {}
        for lag_val in specific_lags_indices:
            feature_name = f'power_lag_{lag_val}'
            if feature_name in all_feature_names_ordered:
                if len(current_power_history) >= lag_val:
                                feature_dict[feature_name] =
current_power_history[-lag_val]
                else:
                     raise ValueError(f"历史序列不够长，无法获取
{feature_name}")
        feature_dict.update(time_features_for_this_step)

        # 按顺序创建值列表
        feature_vector_values = [feature_dict[name] for name
in all_feature_names_ordered]
         return np.array(feature_vector_values).reshape(1, -
1) # 返回 NumPy 数组，预测时再包装

    def
perform_iterative_forecast_for_period_svr(trained_svr_model,
                                            fitted_featur
e_scaler, # 新增：用于缩放预测时的输入特征
```

```
                                                    initial_power
_history,
                                                    period_start_
time,
                                                    num_steps_to_
forecast,
                                                    feature_names
_ordered_list,
                                                    max_lag_val,
                                                    specific_lags
_idx_list):
        """为一个特定的 7 天周期执行 SVR 迭代预测。"""
        predictions_normalized = []
        current_history = list(initial_power_history)
        current_time = pd.Timestamp(period_start_time)

          print(f"开始为 SVR 周期 {period_start_time} 进行
{num_steps_to_forecast} 步迭代预测...")
        for i in range(num_steps_to_forecast):
                                          time_features       =
create_time_features_for_timestamp_svr(current_time)

            # 构建原始（未缩放的）特征向量
                                  raw_feature_vector_np     =
construct_feature_vector_for_step_svr(
                current_history,
                time_features,
                feature_names_ordered_list,
                specific_lags_idx_list
            )

            # 用 feature_scaler 缩放当前步的特征向量
                                  scaled_feature_vector     =
fitted_feature_scaler.transform(raw_feature_vector_np)

                               predicted_norm_power_step    =
trained_svr_model.predict(scaled_feature_vector)[0]
            predictions_normalized.append(predicted_norm_pow
er_step)

            current_history.pop(0)
            current_history.append(predicted_norm_power_step)
            current_time += pd.Timedelta(minutes=15)
            if (i + 1) % 96 == 0:
```

```
                print(f"   SVR 已完成第 {(i + 1) // 96} 天的预
测 (共{num_steps_to_forecast//96}天)...")
        print(f"SVR 周期 {period_start_time} 预测完成！")
        return predictions_normalized

    def
inverse_transform_predictions_svr(predictions_norm_list,
fitted_target_scaler):
        """将归一化的预测列表逆转换为原始尺度。"""
        # (与线性回归版本相同)
                                predictions_2d            =
np.array(predictions_norm_list).reshape(-1, 1)
                        predictions_actual_scale_2d       =
fitted_target_scaler.inverse_transform(predictions_2d)
        return predictions_actual_scale_2d.flatten().tolist()

    # --- 主程序逻辑 ---
    def main_svr():
        """完整执行 SVR 的数据处理、模型训练、预测与保存流程"""
        try:
            # 1. 读取数据
               print(f"SVR 流程：正在读取已处理的 Excel 文件:
'{PROCESSED_EXCEL_PATH}'...")
            if not os.path.exists(PROCESSED_EXCEL_PATH):
                            print(f" 重 要 提 示 ！ 找 不 到
'{PROCESSED_EXCEL_PATH}'。请先运行数据清洗和归一化步骤。")
                return

                                    df_train_loaded       =
pd.read_excel(PROCESSED_EXCEL_PATH,
sheet_name=TRAIN_SHEET_NAME)
                                     df_test_loaded       =
pd.read_excel(PROCESSED_EXCEL_PATH,
sheet_name=TEST_SHEET_NAME)

            for df_ in [df_train_loaded, df_test_loaded]:
                     if  'date'  not  in  df_.columns:  raise
KeyError("'date'列不存在")
                df_['date'] = pd.to_datetime(df_['date'])
                df_.set_index('date', inplace=True)

            print("SVR 流程：训练集和测试集数据已载入。")

            # 2. 特征工程 (在训练集上)
```

```
                    if 'power_normalized' not in
df_train_loaded.columns:
                     raise KeyError("训练集中找不到
'power_normalized' 列！")

                                df_train_featured =
create_features_svr(df_train_loaded,
target_col_name_for_lags='power_normalized',
is_training=True)

          if df_train_featured.empty:
              print("特征工程后 SVR 训练数据为空。程序中止。")
              return

                                        X_train =
df_train_featured[FEATURE_NAMES_IN_ORDER]
               y_train = df_train_featured['y_target'] #
'y_target' 是 'power_normalized'

          # 3. 缩放 SVR 的输入特征 X_train
               print("\n 正在为 SVR 的输入特征进行标准化
(StandardScaler)...")
          feature_scaler = StandardScaler()
                                  X_train_scaled =
feature_scaler.fit_transform(X_train)
                               joblib.dump(feature_scaler,
FEATURE_SCALER_SVR_PATH) # 保存特征 scaler
           print(f"SVR 输入特征的 StandardScaler 已拟合并保存到:
{FEATURE_SCALER_SVR_PATH}")

          # 4. 训练 SVR 模型
          print("\n 开始训练 SVR 模型 (这可能需要一点时间哦)...")
           # 常用的 kernel 是 'rbf'。C 是正则化参数。gamma 是 RBF
核的系数。
          svr_model = SVR(kernel='rbf', C=1.0, epsilon=0.1,
gamma='scale')
          start_time = time.time()
          svr_model.fit(X_train_scaled, y_train)
          end_time = time.time()
              print(f"SVR 模型训练完成！用时： {end_time -
start_time:.2f} 秒。")
           joblib.dump(svr_model, SVR_MODEL_PATH) # 保存 SVR
模型
          print(f"SVR 模型已保存到: {SVR_MODEL_PATH}")
```

```
            # 5. 准备用于目标逆转换的 Scaler (MinMaxScaler)
            # 这个 scaler 应该是在 'power_cleaned' 上拟合的
            if 'power_cleaned' not in df_train_loaded.columns:
                raise KeyError("训练集中找不到 'power_cleaned'
列，无法拟合目标逆转换的 Scaler! ")

              target_scaler = MinMaxScaler(feature_range=(0,
1))
            target_scaler.fit(df_train_loaded[['power_cleane
d']])
            joblib.dump(target_scaler, TARGET_SCALER_PATH) #
保存目标 scaler
                print(f" 目 标 逆 转 换 的 MinMaxScaler 已 在
'power_cleaned' 上拟合并保存到: {TARGET_SCALER_PATH}")

            # --- 开始对测试集进行预测 ---
            df_test_predictions_svr = df_test_loaded.copy()
            df_test_predictions_svr['predicted_power_normali
zed_svr'] = np.nan
            df_test_predictions_svr['predicted_power_actual_
svr'] = np.nan

                                        time_diffs        =
df_test_predictions_svr.index.to_series().diff()
                                  period_starts_indices      =
df_test_predictions_svr.index[(time_diffs                    >
pd.Timedelta(minutes=15)) | (time_diffs.isnull())]
                print(f"\n SVR 流 程 : 在 测 试 集 中 识 别 到
{len(period_starts_indices)} 个预测周期的开始点。")

            num_steps_per_period = 7 * 24 * 4

            for start_dt_of_period in period_starts_indices:
                print(f"\n--- SVR 正在为从 {start_dt_of_period}
开始的 7 天周期进行预测 ---")

                                     initial_history_norm    =
get_initial_history_svr(
                    df_train_loaded['power_normalized'],
                    start_dt_of_period,
                    MAX_LAG
                )
```

```
                        if not initial_history_norm or
len(initial_history_norm) < MAX_LAG:
                    print(f"  SVR 无法为 {start_dt_of_period}
获取足够的初始历史，跳过此周期。")
                    continue

                            normalized_preds_for_period =
perform_iterative_forecast_for_period_svr(
                    svr_model, # 使用训练好的 SVR 模型
                    feature_scaler, # 使用拟合好的特征 scaler
                    initial_history_norm,
                    start_dt_of_period,
                    num_steps_per_period,
                    FEATURE_NAMES_IN_ORDER,
                    MAX_LAG,
                    SPECIFIC_LAGS_USED
                )

                            target_timestamps_in_test =
df_test_predictions_svr.loc[start_dt_of_period :
start_dt_of_period + pd.Timedelta(days=7) -
pd.Timedelta(minutes=15)].index

                    if len(normalized_preds_for_period) ==
len(target_timestamps_in_test):
                    df_test_predictions_svr.loc[target_times
tamps_in_test, 'predicted_power_normalized_svr'] =
normalized_preds_for_period

                            actual_scale_preds_for_period =
inverse_transform_predictions_svr(
                        normalized_preds_for_period,
                        target_scaler # 使用拟合好的目标 scaler
                    )
                    df_test_predictions_svr.loc[target_times
tamps_in_test, 'predicted_power_actual_svr'] =
actual_scale_preds_for_period
                    print(f"  SVR 周期 {start_dt_of_period} 的
预测已填充。")
                else:
                    print(f"  SVR 警告!周期 {start_dt_of_period}
的预测长度与测试集中对应时段长度不匹配！")

            # 保存结果
```

```
            print(f"\n SVR 流程：正在将预测结果保存到 Excel 文件:
'{OUTPUT_EXCEL_PATH}'...")
            with pd.ExcelWriter(OUTPUT_EXCEL_PATH) as writer:
                df_train_loaded.reset_index().to_excel(write
r, sheet_name=TRAIN_SHEET_NAME, index=False)
                df_test_predictions_svr.reset_index().to_exc
el(writer, sheet_name=TEST_SHEET_NAME, index=False)
                  print(f"SVR 预测结果已成功保存！看
'{OUTPUT_EXCEL_PATH}' ！")

        except FileNotFoundError:
                  print(f"SVR 流程：找不到 Excel 文件
'{PROCESSED_EXCEL_PATH}'。")
        except KeyError as e:
             print(f"SVR 流程：DataFrame 中缺少列 '{e}'。")
             import traceback
             traceback.print_exc()
        except Exception as e:
            print(f"SVR 流程：发生意料之外的错误：{e}")
            import traceback
            traceback.print_exc()

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

问题 3 LSTM 的核心代码

```
    import pandas as pd
    import numpy as np
    import tensorflow as tf
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import LSTM, Dense, Input,
Dropout
    from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
    from sklearn.preprocessing import MinMaxScaler
    import joblib
    import os
    import time
    import traceback

    # --- 文件路径和重要的全局设定 ---
    DRIVE_BASE_PATH                                          =
```

```
'/content/drive/MyDrive/elect_cup_2025/'

    POWER_DATA_EXCEL_PATH  =  os.path.join(DRIVE_BASE_PATH,
'problem3_all.xlsx')
    POWER_TRAIN_SHEET_INPUT = 'train'
    POWER_VAL_SHEET_INPUT = 'val' # 新增验证集工作表名称
    POWER_TEST_SHEET_INPUT = 'test'

    WEATHER_DATA_EXCEL_PATH = os.path.join(DRIVE_BASE_PATH,
'normalized_weather_data.xlsx')
    WEATHER_HOURLY_SHEET = '1h_normalized'
    WEATHER_DAILY_SHEET = '1d_normalized'

    OUTPUT_EXCEL_PATH    =    os.path.join(DRIVE_BASE_PATH,
'problem3_LSTM_NWP_val_sheet_forecast_results_v5.xlsx')   #
输出文件名更新

    LSTM_NWP_MODEL_PATH   =   os.path.join(DRIVE_BASE_PATH,
'lstm_nwp_model_v5.keras')
    TARGET_SCALER_LSTM_NWP_PATH                              =
os.path.join(DRIVE_BASE_PATH,
'target_min_max_scaler_for_lstm_nwp_v5.joblib')

    # --- LSTM 模型参数设定 (保持不变) ---
    N_TIMESTEPS = 96
    LSTM_UNITS = 100
    EPOCHS = 30
    BATCH_SIZE = 64
    # VALIDATION_SPLIT_RATIO 不再需要，因为我们有独立的验证集了

    # --- 要从天气数据中选取的特征列 (保持不变) ---
    HOURLY_WEATHER_FEATURES_TO_USE = [
        'global_tilted_irradiance_instant',
        'direct_normal_irradiance_instant',
        'temperature_2m',
        'cloud_cover',
        'wind_speed_10m',
    ]
    DAILY_WEATHER_FEATURES_TO_USE = [
        'sunshine_duration',
        'precipitation_hours',
        'wind_speed_10m_mean',
        'cloud_cover_mean'
    ]
```

```
    def clean_column_names_v3(df, prefix=""): # 函数名保持，逻辑不变
        df_cleaned = df.copy()
        new_cols_map = {}
        for col in df_cleaned.columns:
            name_part = str(col).split(' (')[0]
                  cleaned_name  =  name_part.replace('  ', '_').replace('/',  '_per_').replace('.',  '').replace('°', 'deg')
            if prefix:
                                  new_cols_map[str(col)]   = f"{prefix}_{cleaned_name}"
            else:
                new_cols_map[str(col)] = cleaned_name
        df_cleaned.rename(columns=new_cols_map, inplace=True)
        return df_cleaned

    def load_and_prepare_data_v5(): # 函数名更新为 v5
        print("v5 版(使用独立验证集)：开始加载和准备数据...")
        if not os.path.exists(POWER_DATA_EXCEL_PATH):
            raise  FileNotFoundError(f"找不到功率数据文件 '{POWER_DATA_EXCEL_PATH}'！")
        if not os.path.exists(WEATHER_DATA_EXCEL_PATH):
            raise  FileNotFoundError(f"找不到天气数据文件 '{WEATHER_DATA_EXCEL_PATH}'！")

        df_train_power = pd.read_excel(POWER_DATA_EXCEL_PATH, sheet_name=POWER_TRAIN_SHEET_INPUT)
         df_val_power = pd.read_excel(POWER_DATA_EXCEL_PATH, sheet_name=POWER_VAL_SHEET_INPUT) # 加载验证集功率数据
        df_test_power = pd.read_excel(POWER_DATA_EXCEL_PATH, sheet_name=POWER_TEST_SHEET_INPUT)

        # 对训练、验证、测试集的功率数据进行相同的初始处理
          for df_p, name in [(df_train_power, "训练功率"), (df_val_power, "验证功率"), (df_test_power, "测试功率")]:
              if  'date'  not  in  df_p.columns:  raise KeyError(f"'{name}'数据中找不到 'date' 列！")
            df_p['date'] = pd.to_datetime(df_p['date'])
            df_p.set_index('date', inplace=True)
             for col in ['local_time_hour_of_day', 'dateN', 'dayofweek']:
                if col not in df_p.columns:
```

```
                    if col == 'local_time_hour_of_day':
df_p[col] = df_p.index.hour + df_p.index.minute / 60.0
                     elif col == 'dateN': df_p[col] =
df_p.index.dayofyear
                   elif col == 'dayofweek': df_p[col] =
df_p.index.dayofweek

       xls_weather = pd.ExcelFile(WEATHER_DATA_EXCEL_PATH)
          df_weather_h_orig = pd.read_excel(xls_weather,
sheet_name=WEATHER_HOURLY_SHEET)
          df_weather_d_orig = pd.read_excel(xls_weather,
sheet_name=WEATHER_DAILY_SHEET)

       for df_w_orig, name_w in [(df_weather_h_orig, "小时天
气"), (df_weather_d_orig, "每日天气")]:
              if 'time' not in df_w_orig.columns: raise
KeyError(f"'{name_w}'数据中找不到 'time' 列！")
                                  df_w_orig['time']      =
pd.to_datetime(df_w_orig['time'])
           df_w_orig.set_index('time', inplace=True)

                                df_weather_h             =
clean_column_names_v3(df_weather_h_orig, prefix="h")
                                df_weather_d             =
clean_column_names_v3(df_weather_d_orig, prefix="d")

       hourly_features_final_names = [f"h_{str(col).split('
')[0].replace(' ', '_').replace('/', '_per_').replace('.',
'').replace('°',      'deg')}"      for      col      in
HOURLY_WEATHER_FEATURES_TO_USE]
        daily_features_final_names = [f"d_{str(col).split('
')[0].replace(' ', '_').replace('/', '_per_').replace('.',
'').replace('°',      'deg')}"      for      col      in
DAILY_WEATHER_FEATURES_TO_USE]

       print(" v5 版：正在合并天气数据到训练、验证、测试集...")
       results_final = []
       # 现在我们对三个数据集进行相同的合并操作
            for   df_power_current,   name_dataset   in
[(df_train_power, "训练集"), (df_val_power, "验证集"),
(df_test_power, "测试集")]:
           print(f"  正在处理: {name_dataset}")
           df_merged = df_power_current.copy()
           df_merged = df_merged.sort_index()
```

```
                    if not isinstance(df_merged.index,
pd.DatetimeIndex):
                                        df_merged.index =
pd.to_datetime(df_merged.index)

          df_weather_h_to_merge = df_weather_h.sort_index()
            if not isinstance(df_weather_h_to_merge.index,
pd.DatetimeIndex):
                            df_weather_h_to_merge.index =
pd.to_datetime(df_weather_h_to_merge.index)

               hourly_cols_present = [col for col in
hourly_features_final_names if col in
df_weather_h_to_merge.columns]
          if hourly_cols_present:
                    df_merged = pd.merge_asof(df_merged,
df_weather_h_to_merge[hourly_cols_present],
                                        left_index=True,
right_index=True, direction='backward',
                                     tolerance=pd.Timedel
ta(hours=2))
          else:
              print(f"  警告 ({name_dataset}): 小时天气特征列
在小时天气数据中均未找到! 将不会合并。")
              for col_name in hourly_features_final_names:
                    if col_name not in df_merged.columns:
df_merged[col_name] = np.nan

          df_weather_d_to_merge = df_weather_d.sort_index()
            if not isinstance(df_weather_d_to_merge.index,
pd.DatetimeIndex):
                            df_weather_d_to_merge.index =
pd.to_datetime(df_weather_d_to_merge.index)

                              df_weather_d_with_date =
df_weather_d_to_merge.copy()
           df_weather_d_with_date['date_only_merge_key'] =
df_weather_d_with_date.index.normalize()

          df_merged_reset = df_merged.reset_index()
                df_merged_reset['date_only_merge_key'] =
pd.to_datetime(df_merged_reset['date']).dt.normalize()
```

```
                daily_cols_present = [col for col in
daily_features_final_names if col in
df_weather_d_with_date.columns]
            if daily_cols_present:
                df_merged_reset = pd.merge(df_merged_reset,
                                           df_weather_d_with_dat
e[daily_cols_present + ['date_only_merge_key']],
                                           on='date_only_merge_k
ey', how='left', suffixes=('', '_daily_drop'))
                    cols_to_drop_daily = [col for col in
df_merged_reset.columns if '_daily_drop' in col]
                                    if cols_to_drop_daily:
df_merged_reset.drop(columns=cols_to_drop_daily,
inplace=True)
            else:
                print(f"  警告 ({name_dataset}): 每日天气特征列
在每日天气数据中均未找到! 将不会合并。")
                for col_name in daily_features_final_names:
                                     if col_name not in
df_merged_reset.columns: df_merged_reset[col_name] = np.nan

                            if 'date_only_merge_key' in
df_merged_reset.columns:
                df_merged_reset.drop(columns=['date_only_mer
ge_key'], inplace=True)

                                           df_merged =
df_merged_reset.set_index('date').sort_index()

               df_merged['hour_sin'] = np.sin(2 * np.pi *
df_merged['local_time_hour_of_day'] / 24.0)
               df_merged['hour_cos'] = np.cos(2 * np.pi *
df_merged['local_time_hour_of_day'] / 24.0)
               df_merged['dayN_sin'] = np.sin(2 * np.pi *
df_merged['dateN'] / 365.25)
               df_merged['dayN_cos'] = np.cos(2 * np.pi *
df_merged['dateN'] / 365.25)
            df_merged['dayofweek_sin'] = np.sin(2 * np.pi *
df_merged['dayofweek'] / 7.0)
            df_merged['dayofweek_cos'] = np.cos(2 * np.pi *
df_merged['dayofweek'] / 7.0)
            results_final.append(df_merged)

          df_train_final, df_val_final, df_test_final =
```

```
results_final[0], results_final[1], results_final[2] # ！解包出三个 DataFrame
        print("数据加载、合并和特征添加完成！")

        # 确定最终用于序列的特征列（以训练集为基准）
        base_lstm_features = ['power_normalized']
         weather_time_lstm_features_list = [col for col in hourly_features_final_names if col in df_train_final.columns] + \
                                           [col for col in daily_features_final_names if col in df_train_final.columns] + \
                                     ['hour_sin', 'hour_cos', 'dayN_sin', 'dayN_cos', 'dayofweek_sin', 'dayofweek_cos']

        seen_lstm_features = set(base_lstm_features)
        unique_weather_time_lstm = []
        for f_col in weather_time_lstm_features_list:
            if f_col not in seen_lstm_features:
                unique_weather_time_lstm.append(f_col)
                seen_lstm_features.add(f_col)
         final_feature_columns_for_lstm = base_lstm_features + unique_weather_time_lstm

         # 确保所有数据帧都包含这些最终选定的特征列，如果不存在则填充（例如用 0）
          for df_check in [df_train_final, df_val_final, df_test_final]:
              for col_final in final_feature_columns_for_lstm:
                  if col_final not in df_check.columns:
                       print(f"警告：特征列 '{col_final}' 在某个数据子集中缺失，将填充为 0。")
                      df_check[col_final] = 0

             print(f"最终用于 LSTM 序列的特征列（{len(final_feature_columns_for_lstm)} 个）：{final_feature_columns_for_lstm}")
         return df_train_final, df_val_final, df_test_final, final_feature_columns_for_lstm

    # (create_sequences_nwp_v2, get_initial_history_nwp_v2,
    #  perform_iterative_forecast_for_period_lstm_nwp_v2,
    #  inverse_transform_predictions_v2
    def create_sequences_nwp_v2(df_with_features,
```

```
target_col_name, feature_cols_for_sequence, n_timesteps):
        print(f" v5 版：正在创建 LSTM 序列，回看：{n_timesteps}, 特征：{feature_cols_for_sequence}")
       X, y = [], []
            missing_cols_in_df = [col for col in feature_cols_for_sequence if col not in df_with_features.columns]
       if missing_cols_in_df: raise KeyError(f"创建序列时，以下特征列在 DataFrame 中找不到：{missing_cols_in_df}")
        if target_col_name not in df_with_features.columns: raise KeyError(f"目标列 '{target_col_name}' 在 DataFrame 中找不到！")
                                    data_values = df_with_features[feature_cols_for_sequence].values
                                   target_values = df_with_features[target_col_name].values
       if len(data_values) <= n_timesteps:
            print(f"数据太短 ({len(data_values)}条)，无法创建长度为{n_timesteps}的序列！")
           return np.array(X), np.array(y)
       for i in range(len(data_values) - n_timesteps):
           X.append(data_values[i:(i + n_timesteps), :])
           y.append(target_values[i + n_timesteps])
       return np.array(X), np.array(y)

   def   get_initial_history_nwp_v2(df_train_all_features, test_period_start_time, n_timesteps_needed, feature_cols_for_sequence):
          if  not  isinstance(df_train_all_features.index, pd.DatetimeIndex):
            raise TypeError("get_initial_history_nwp_v2 需要 df_train_all_features 的索引是 DatetimeIndex！")
           history_end_time  =  test_period_start_time  - pd.Timedelta(minutes=15)
                              relevant_train_data = df_train_all_features[df_train_all_features.index <= history_end_time]
       if len(relevant_train_data) < n_timesteps_needed:
                raise  ValueError(f"训练数据不足以提供 {n_timesteps_needed} 条初始历史 (仅找到 {len(relevant_train_data)} 条在 {test_period_start_time} 之前)。")
          missing_cols_in_hist_df  =  [col  for  col  in feature_cols_for_sequence if col not in
```

```
relevant_train_data.columns]
        if missing_cols_in_hist_df:
                raise  KeyError(f"获取初始历史时，以下特征列在
relevant_train_data 中找不到: {missing_cols_in_hist_df}")
                            initial_history_df_slice          =
relevant_train_data[feature_cols_for_sequence].iloc[-
n_timesteps_needed:]
        return initial_history_df_slice.values

    def perform_iterative_forecast_for_period_lstm_nwp_v2(
                                                    trained_lstm_
model, initial_history_features_array,
                                                    num_steps_to_
forecast, df_test_for_future_nwp_lookup,
                                                    period_start_
time_for_nwp, power_col_idx_in_features_list,
                                                    all_feature_n
ames_for_sequence_list):
        predictions_normalized = []
                             current_sequence_features         =
initial_history_features_array.copy()
                           current_time_for_prediction         =
pd.Timestamp(period_start_time_for_nwp)
                                N_ACTUAL_FEATURES              =
current_sequence_features.shape[1]

         print(f" 开始为 LSTM+NWP 进行 {num_steps_to_forecast}
步迭代预测，从 {current_time_for_prediction} 开始...")
        for i in range(num_steps_to_forecast):
                                            model_input        =
current_sequence_features.reshape(1,           N_TIMESTEPS,
N_ACTUAL_FEATURES)
                                predicted_norm_power_step      =
trained_lstm_model.predict(model_input, verbose=0)[0, 0]
            predictions_normalized.append(predicted_norm_pow
er_step)

            if i < num_steps_to_forecast - 1:
                               next_sequence_start_features    =
current_sequence_features[1:, :].copy()
                          next_actual_timestamp_for_features   =
current_time_for_prediction + pd.Timedelta(minutes=15)
                try:
                             future_features_for_step_series   =
```

```
df_test_for_future_nwp_lookup.loc[next_actual_timestamp_for
_features, all_feature_names_for_sequence_list]
                                future_features_for_step   =
future_features_for_step_series.values.copy()
                    future_features_for_step[power_col_idx_i
n_features_list] = predicted_norm_power_step
                                  new_last_step_features   =
future_features_for_step
                except KeyError:
                          print(f"警告！在测试数据中找不到时间点
{next_actual_timestamp_for_features} 的外部特征，将使用上一步的
非功率特征进行填充。")
                                  new_last_step_features   =
current_sequence_features[-1, :].copy()
                    new_last_step_features[power_col_idx_in_
features_list] = predicted_norm_power_step
                               current_sequence_features   =
np.vstack((next_sequence_start_features,
new_last_step_features.reshape(1, N_ACTUAL_FEATURES)))
                              current_time_for_prediction   =
next_actual_timestamp_for_features
            if (i + 1) % 96 == 0: print(f"  LSTM+NWP 已完成第
{(i + 1) // 96} 天的预测...")
        print(f"LSTM+NWP 周期预测完成！")
        return predictions_normalized

    def
inverse_transform_predictions_v2(predictions_norm_list,
fitted_target_scaler):
        if not predictions_norm_list: return []
                                  predictions_2d            =
np.array(predictions_norm_list).reshape(-1, 1)
        if not hasattr(fitted_target_scaler, 'data_min_'):
            raise ValueError("用于逆转换的 Scaler 还没有被拟合过！
")
                         predictions_actual_scale_2d        =
fitted_target_scaler.inverse_transform(predictions_2d)
        return predictions_actual_scale_2d.flatten().tolist()

    # --- 主程序逻辑 ---
    def main_lstm_nwp_v5(): # 版本号更新
        """完整执行 LSTM+NWP 的数据处理、模型训练（使用独立验证集）、
预测与保存流程"""
        try:
```

```
        # 现在会返回三个 DataFrame
            df_train_final, df_val_final, df_test_final,
final_feature_columns_for_lstm = load_and_prepare_data_v5()

                                    power_normalized_idx      =
final_feature_columns_for_lstm.index('power_normalized')
                                      N_FEATURES_ACTUAL       =
len(final_feature_columns_for_lstm)

         print(f"\n v5 版：正在创建 LSTM 的训练和验证序列 (时间
步长 N_TIMESTEPS={N_TIMESTEPS})...")
        # 训练前用 0 填充 NaN
                           X_train_lstm,    y_train_lstm    =
create_sequences_nwp_v2(df_train_final.fillna(0),
                                                             t
arget_col_name='power_normalized',
                                                             f
eature_cols_for_sequence=final_feature_columns_for_lstm,
                                                             n
_timesteps=N_TIMESTEPS)
        # 为验证集创建序列
                              X_val_lstm,     y_val_lstm     =
create_sequences_nwp_v2(df_val_final.fillna(0),
                                                           tar
get_col_name='power_normalized',
                                                           fea
ture_cols_for_sequence=final_feature_columns_for_lstm,
                                                           n_t
imesteps=N_TIMESTEPS)

                    if   X_train_lstm.shape[0]   ==   0   or
X_val_lstm.shape[0] == 0 :
              print("LSTM 训练或验证数据序列创建失败 (可能数据不
足)。程序中止。")
              return
        print(f"LSTM 训练数据形状: X: {X_train_lstm.shape},
y: {y_train_lstm.shape}")
         print(f"LSTM 验证数据形状: X: {X_val_lstm.shape},
y: {y_val_lstm.shape}")

        print("\nv5 版：开始构建和训练 LSTM 模型...")
        lstm_model = Sequential([
                                    Input(shape=(N_TIMESTEPS,
N_FEATURES_ACTUAL)),
```

```
                    LSTM(LSTM_UNITS,  activation='tanh',
return_sequences=True),
            Dropout(0.2),
                LSTM(LSTM_UNITS // 2, activation='tanh',
return_sequences=False),
            Dropout(0.2),
            Dense(1, activation='sigmoid')
        ])
        lstm_model.compile(optimizer=tf.keras.optimizers
.Adam(learning_rate=0.001), loss='mean_squared_error')
        print("LSTM 模型结构：")
        lstm_model.summary()

                                  early_stopping       =
EarlyStopping(monitor='val_loss',             patience=10,
restore_best_weights=True, verbose=1)
        reduce_lr = ReduceLROnPlateau(monitor='val_loss',
factor=0.2, patience=5, min_lr=0.00001, verbose=1)

        start_time = time.time()
        print(f"开始训练模型，共 {EPOCHS} 轮...")
              history   =   lstm_model.fit(X_train_lstm,
y_train_lstm,
                                  epochs=EPOCHS,
                                  batch_size=BATCH_SIZE,
                                   validation_data=(X_val_l
stm, y_val_lstm), # 使用独立的验证集
                                   callbacks=[early_stoppin
g, reduce_lr],
                                  verbose=1, shuffle=True)
        end_time = time.time()
            print(f"LSTM 模型训练完成！用时： {end_time  -
start_time:.2f} 秒。")
        lstm_model.save(LSTM_NWP_MODEL_PATH)
        print(f"LSTM 模型已保存到: {LSTM_NWP_MODEL_PATH}")

                           df_train_power_for_scaler     =
pd.read_excel(POWER_DATA_EXCEL_PATH,
sheet_name=POWER_TRAIN_SHEET_INPUT)
                       if    'power_cleaned'    not    in
df_train_power_for_scaler.columns:
                raise KeyError("用于 Scaler 的原始训练功率数据中
找不到 'power_cleaned' 列！")
```

```
            target_scaler = MinMaxScaler(feature_range=(0,
1))
            target_scaler.fit(df_train_power_for_scaler[['po
wer_cleaned']].dropna())
                                    joblib.dump(target_scaler,
TARGET_SCALER_LSTM_NWP_PATH)
                print(f"目标逆转换的 MinMaxScaler 已在
'power_cleaned' 上拟合并保存到:
{TARGET_SCALER_LSTM_NWP_PATH}")

            df_test_predictions_lstm = df_test_final.copy()
            df_test_predictions_lstm[final_feature_columns_f
or_lstm]                                                    =
df_test_predictions_lstm[final_feature_columns_for_lstm].fi
llna(0)
            df_test_predictions_lstm['predicted_power_normal
ized_lstm_nwp'] = np.nan
            df_test_predictions_lstm['predicted_power_actual
_lstm_nwp'] = np.nan

                                        df_test_sorted       =
df_test_predictions_lstm.sort_index()
                if  not  isinstance(df_test_sorted.index,
pd.DatetimeIndex):
                print("警告：df_test_sorted 的索引在预测前不是
DatetimeIndex，尝试再次转换...")
              if 'date' in df_test_sorted.columns:
                                    df_test_sorted['date']   =
pd.to_datetime(df_test_sorted['date'])
                                        df_test_sorted   =
df_test_sorted.set_index('date').sort_index()
              else:
                   raise TypeError("测试集索引在识别周期前不是
日期时间类型，且找不到'date'列！")

                                       time_diffs        =
df_test_sorted.index.to_series().diff()
            is_new_period_start_mask = time_diffs.isnull() |
(time_diffs > pd.Timedelta(minutes=15))
                                  period_starts_indices      =
df_test_sorted.index[is_new_period_start_mask]

                print(f"\n  v5 版：在测试集中识别到
{len(period_starts_indices)} 个预测周期的开始点。")
```

```
                    if not period_starts_indices.empty:
print(period_starts_indices)

            num_steps_per_period = 7 * 24 * 4

            for start_dt_of_period in period_starts_indices:
                        print(f"\n---  LSTM+NWP(v5) 正在为从
{start_dt_of_period} 开始的 7 天周期进行预测 ---")
                try:
                                    initial_history_features  =
get_initial_history_nwp_v2(
                        df_train_final.fillna(0),
                        start_dt_of_period,
                        N_TIMESTEPS,
                        final_feature_columns_for_lstm
                    )
                except ValueError as e:
                    print(f"  {e} 跳过此周期。")
                    continue

                            df_test_current_period_lookup  =
df_test_sorted[df_test_sorted.index >= start_dt_of_period]
                if df_test_current_period_lookup.empty:
                     print(f"测试集中找不到 {start_dt_of_period}
之后的数据。跳过此周期。")
                     continue

                              normalized_preds_for_period  =
perform_iterative_forecast_for_period_lstm_nwp_v2(
                        lstm_model, initial_history_features,
num_steps_per_period,
                                df_test_current_period_lookup,
start_dt_of_period,
                                            power_normalized_idx,
final_feature_columns_for_lstm
                )

                 current_period_end_time = start_dt_of_period
+ pd.Timedelta(days=7) - pd.Timedelta(minutes=15)
                          target_timestamps_in_test_period  =
df_test_sorted.loc[start_dt_of_period                      :
current_period_end_time].index

                                            actual_pred_len    =
```

```
len(normalized_preds_for_period)
                                        target_ts_len     =
len(target_timestamps_in_test_period)

                if actual_pred_len == 0 and target_ts_len >
0 :
                    print(f"  周期 {start_dt_of_period} 没有生
成预测值，但期望有 {target_ts_len} 个。")
                   continue
                if actual_pred_len == 0 and target_ts_len ==
0:
                    print(f"  周期 {start_dt_of_period} 目标时
段为空，跳过填充。")
                   continue

                if actual_pred_len > target_ts_len:
                            normalized_preds_for_period  =
normalized_preds_for_period[:target_ts_len]
                elif actual_pred_len < target_ts_len:
                          target_timestamps_in_test_period  =
target_timestamps_in_test_period[:actual_pred_len]

                if not target_timestamps_in_test_period.empty
and actual_pred_len > 0 :
                    df_test_predictions_lstm.loc[target_time
stamps_in_test_period,
'predicted_power_normalized_lstm_nwp']                    =
normalized_preds_for_period
                           actual_scale_preds_for_period  =
inverse_transform_predictions_v2(
                               normalized_preds_for_period,
target_scaler
                   )
                    df_test_predictions_lstm.loc[target_time
stamps_in_test_period, 'predicted_power_actual_lstm_nwp'] =
actual_scale_preds_for_period
                                print(f"    LSTM+NWP  周 期
{start_dt_of_period} 的预测已填充。")
                else:
                     print(f"  周期 {start_dt_of_period} 预测
结果为空或在测试集中找不到对应的目标时间戳来填充预测。")

                print(f"\n  正在将预测结果保存到 Excel 文件:
'{OUTPUT_EXCEL_PATH}'...")
```

```
            with pd.ExcelWriter(OUTPUT_EXCEL_PATH) as writer:
                df_train_final.reset_index().to_excel(writer,
sheet_name=OUTPUT_TRAIN_SHEET_NAME, index=False)

                # df_val_final.reset_index().to_excel(writer,
sheet_name='validation_data_with_features', index=False)
                df_test_predictions_lstm.reset_index().to_ex
cel(writer, sheet_name=OUTPUT_TEST_SHEET_NAME, index=False)
                 print(f"LSTM+NWP 预测结果已成功保存！看
'{OUTPUT_EXCEL_PATH}' ！")

        except FileNotFoundError as e:
                print(f"LSTM+NWP(v5)流程：找不到 Excel 文件
'{e.filename}'。")
        except KeyError as e:
             print(f"LSTM+NWP(v5)流程:DataFrame 中缺少列 '{e}'。
")
             traceback.print_exc()
        except Exception as e:
            print(f"LSTM+NWP(v5)流程：发生意料之外的错误：{e}")
            traceback.print_exc()

    if __name__ == "__main__":
        main_lstm_nwp_v5()
```

问题 3 Transformer 的核心代码

```
    import pandas as pd
    import numpy as np
    import tensorflow as tf
    from tensorflow.keras.models import Model # 使用函数式 API
构建 Transformer
    from tensorflow.keras.layers import Input, Dense,
Dropout, LayerNormalization, MultiHeadAttention,
GlobalAveragePooling1D, Embedding, Add
    from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
    from sklearn.preprocessing import MinMaxScaler
    import joblib
    import os
    import time
    import traceback
```

```
    # --- 文件路径和重要的全局设定 ---
    DRIVE_BASE_PATH                                          =
'/content/drive/MyDrive/elect_cup_2025/'

    POWER_DATA_EXCEL_PATH  =  os.path.join(DRIVE_BASE_PATH,
'problem3_all.xlsx')
    POWER_TRAIN_SHEET_INPUT = 'train'
    POWER_VAL_SHEET_INPUT = 'val'
    POWER_TEST_SHEET_INPUT = 'test'

    WEATHER_DATA_EXCEL_PATH = os.path.join(DRIVE_BASE_PATH,
'normalized_weather_data.xlsx')
    WEATHER_HOURLY_SHEET = '1h_normalized'
    WEATHER_DAILY_SHEET = '1d_normalized'

    OUTPUT_EXCEL_PATH    =    os.path.join(DRIVE_BASE_PATH,
'problem3_Transformer_NWP_forecast_results_v1.xlsx')

    TRANSFORMER_MODEL_PATH  =  os.path.join(DRIVE_BASE_PATH,
'transformer_nwp_model_v1.keras')
    TARGET_SCALER_TRANSFORMER_NWP_PATH                       =
os.path.join(DRIVE_BASE_PATH,
'target_min_max_scaler_for_transformer_nwp_v1.joblib')

    # --- Transformer 及训练参数设定 ---
    N_TIMESTEPS = 96      # Transformer 回看的时间步长 (Encoder
序列长度)
    # N_FEATURES 将在数据准备后动态确定
    EPOCHS = 50           # 训练轮数 (Transformer 可能需要更多轮
次或更仔细的调优)
    BATCH_SIZE = 64
    VALIDATION_SPLIT_RATIO = None # 因为我们现在使用独立的 val
sheet

    # Transformer spécifiques parameters
    D_MODEL = 64           # Transformer 的内部维度 (Embedding
dimension)
    NUM_HEADS = 4         # 多头注意力机制的头数
    FF_DIM = 128          # FeedForward 网络内部维度
    NUM_TRANSFORMER_BLOCKS = 2 # Transformer Encoder 层数
    DROPOUT_RATE = 0.15   # Dropout 比例

    # --- 要从天气数据中选取的特征列 (与 LSTM 版本保持一致) ---
    HOURLY_WEATHER_FEATURES_TO_USE = [
```

```
                        'global_tilted_irradiance_instant',
'direct_normal_irradiance_instant',
        'temperature_2m', 'cloud_cover', 'wind_speed_10m',
    ]
    DAILY_WEATHER_FEATURES_TO_USE = [
        'sunshine_duration', 'precipitation_hours',
        'wind_speed_10m_mean', 'cloud_cover_mean'
    ]

    # --- Helper Functions (大部分可以从之前的 LSTM v5 版本复用或稍作修改) ---

    def clean_column_names_tf(df, prefix=""): # 使用新名避免与旧脚本函数冲突
        df_cleaned = df.copy()
        new_cols_map = {}
        for col in df_cleaned.columns:
            name_part = str(col).split(' (')[0]
                  cleaned_name  =  name_part.replace('  ',
'_').replace('/',  '_per_').replace('.',  '').replace('°',
'deg')
                   if  prefix:  new_cols_map[str(col)]  =
f"{prefix}_{cleaned_name}"
            else: new_cols_map[str(col)] = cleaned_name
        df_cleaned.rename(columns=new_cols_map, inplace=True)
        return df_cleaned

    def load_and_prepare_data_tf(): # 使用新名
        print(" Transformer 流程：开始加载和准备数据...")
        # (此函数逻辑与之前的 load_and_prepare_data_v5 基本一致，确保返回三个 DataFrame 和特征列表)
        # ... (省略详细代码，假设它正确加载、合并、创建时间特征，并返回 df_train, df_val, df_test, feature_list)
        # 确保函数名为 load_and_prepare_data_tf 并在主函数中调用
          # 粘贴之前的 load_and_prepare_data_v5 并重命名为 load_and_prepare_data_tf
            # 注意：HOURLY_WEATHER_FEATURES_TO_USE 和 DAILY_WEATHER_FEATURES_TO_USE 定义在全局
        # clean_column_names_v3 也需要重命名或确保可用

        print(" Transformer 流程：开始加载和准备数据...")
        if not os.path.exists(POWER_DATA_EXCEL_PATH):
               raise FileNotFoundError(f"找不到功率数据文件 '{POWER_DATA_EXCEL_PATH}'！")
```

```
        if not os.path.exists(WEATHER_DATA_EXCEL_PATH):
                raise  FileNotFoundError(f"找不到天气数据文件
'{WEATHER_DATA_EXCEL_PATH}'！")

        df_train_power = pd.read_excel(POWER_DATA_EXCEL_PATH,
sheet_name=POWER_TRAIN_SHEET_INPUT)
         df_val_power = pd.read_excel(POWER_DATA_EXCEL_PATH,
sheet_name=POWER_VAL_SHEET_INPUT)
        df_test_power = pd.read_excel(POWER_DATA_EXCEL_PATH,
sheet_name=POWER_TEST_SHEET_INPUT)

          for df_p, name in [(df_train_power, "训练功率"),
(df_val_power, "验证功率"), (df_test_power, "测试功率")]:
                  if  'date'  not  in  df_p.columns:  raise
KeyError(f"'{name}'数据中找不到 'date' 列！")
            df_p['date'] = pd.to_datetime(df_p['date'])
            df_p.set_index('date', inplace=True)
             for col in ['local_time_hour_of_day', 'dateN',
'dayofweek']:
                if col not in df_p.columns:
                        if  col  ==  'local_time_hour_of_day':
df_p[col] = df_p.index.hour + df_p.index.minute / 60.0
                         elif  col  ==  'dateN':  df_p[col]  =
df_p.index.dayofyear
                       elif col == 'dayofweek': df_p[col] =
df_p.index.dayofweek

        xls_weather = pd.ExcelFile(WEATHER_DATA_EXCEL_PATH)
           df_weather_h_orig  =  pd.read_excel(xls_weather,
sheet_name=WEATHER_HOURLY_SHEET)
           df_weather_d_orig  =  pd.read_excel(xls_weather,
sheet_name=WEATHER_DAILY_SHEET)

        for df_w_orig, name_w in [(df_weather_h_orig, "小时天
气"), (df_weather_d_orig, "每日天气")]:
                if  'time'  not  in  df_w_orig.columns:  raise
KeyError(f"'{name_w}'数据中找不到 'time' 列！")
                                    df_w_orig['time']      =
pd.to_datetime(df_w_orig['time'])
            df_w_orig.set_index('time', inplace=True)

                                  df_weather_h             =
clean_column_names_tf(df_weather_h_orig, prefix="h") # 使用
新函数名
```

```
                                df_weather_d                =
clean_column_names_tf(df_weather_d_orig, prefix="d") # 使用
新函数名

       hourly_features_final_names = [f"h_{str(col).split('
('')[0].replace(' ', '_').replace('/', '_per_').replace('.',
'').replace('°',       'deg')}"       for       col       in
HOURLY_WEATHER_FEATURES_TO_USE]
        daily_features_final_names = [f"d_{str(col).split('
('')[0].replace(' ', '_').replace('/', '_per_').replace('.',
'').replace('°',       'deg')}"       for       col       in
DAILY_WEATHER_FEATURES_TO_USE]

       results_final = []
             for    df_power_current,    name_dataset    in
[(df_train_power,  "训练集"),  (df_val_power,  "验证集"),
(df_test_power, "测试集")]:
           df_merged = df_power_current.copy()
           df_merged = df_merged.sort_index()
                     if   not   isinstance(df_merged.index,
pd.DatetimeIndex):             df_merged.index             =
pd.to_datetime(df_merged.index)

           df_weather_h_to_merge = df_weather_h.sort_index()
             if not isinstance(df_weather_h_to_merge.index,
pd.DatetimeIndex):       df_weather_h_to_merge.index       =
pd.to_datetime(df_weather_h_to_merge.index)
                hourly_cols_present  =  [col  for  col  in
hourly_features_final_names        if        col        in
df_weather_h_to_merge.columns]
           if hourly_cols_present:
                      df_merged  =  pd.merge_asof(df_merged,
df_weather_h_to_merge[hourly_cols_present],
                                            left_index=True,
right_index=True,                       direction='backward',
tolerance=pd.Timedelta(hours=2))
           else:
               for col_name in hourly_features_final_names:
                     if col_name not in df_merged.columns:
df_merged[col_name] = np.nan

           df_weather_d_to_merge = df_weather_d.sort_index()
             if not isinstance(df_weather_d_to_merge.index,
pd.DatetimeIndex):       df_weather_d_to_merge.index       =
```

```
pd.to_datetime(df_weather_d_to_merge.index)
                                df_weather_d_with_date      =
df_weather_d_to_merge.copy()
            df_weather_d_with_date['date_only_merge_key'] =
df_weather_d_with_date.index.normalize()
           df_merged_reset = df_merged.reset_index()
                 df_merged_reset['date_only_merge_key']  =
pd.to_datetime(df_merged_reset['date']).dt.normalize()
                daily_cols_present  =  [col  for  col  in
daily_features_final_names        if        col        in
df_weather_d_with_date.columns]
           if daily_cols_present:
                df_merged_reset = pd.merge(df_merged_reset,
df_weather_d_with_date[daily_cols_present                 +
['date_only_merge_key']],
                                       on='date_only_merge_k
ey', how='left', suffixes=('', '_daily_drop'))
                   cols_to_drop_daily  =  [col  for  col  in
df_merged_reset.columns if '_daily_drop' in col]
                                    if    cols_to_drop_daily:
df_merged_reset.drop(columns=cols_to_drop_daily,
inplace=True)
           else:
               for col_name in daily_features_final_names:
                                     if   col_name   not   in
df_merged_reset.columns: df_merged_reset[col_name] = np.nan
                           if    'date_only_merge_key'    in
df_merged_reset.columns:
df_merged_reset.drop(columns=['date_only_merge_key'],
inplace=True)
                                           df_merged        =
df_merged_reset.set_index('date').sort_index()

              df_merged['hour_sin'] = np.sin(2 * np.pi *
df_merged['local_time_hour_of_day'] / 24.0)
              df_merged['hour_cos'] = np.cos(2 * np.pi *
df_merged['local_time_hour_of_day'] / 24.0)
              df_merged['dayN_sin'] = np.sin(2 * np.pi *
df_merged['dateN'] / 365.25)
              df_merged['dayN_cos'] = np.cos(2 * np.pi *
df_merged['dateN'] / 365.25)
            df_merged['dayofweek_sin'] = np.sin(2 * np.pi *
df_merged['dayofweek'] / 7.0)
            df_merged['dayofweek_cos'] = np.cos(2 * np.pi *
```

```
df_merged['dayofweek'] / 7.0)
            results_final.append(df_merged)

          df_train_final, df_val_final, df_test_final =
results_final[0], results_final[1], results_final[2]

        base_tf_features = ['power_normalized']
          weather_time_tf_features_list = [col for col in
hourly_features_final_names if col in df_train_final.columns]
+ \
                                                [col for col in
daily_features_final_names if col in df_train_final.columns]
+ \
                                                    ['hour_sin',
'hour_cos',   'dayN_sin',   'dayN_cos',   'dayofweek_sin',
'dayofweek_cos']
        seen_tf_features = set(base_tf_features)
        unique_weather_time_tf = []
        for f_col in weather_time_tf_features_list:
                    if   f_col   not   in   seen_tf_features:
unique_weather_time_tf.append(f_col);
seen_tf_features.add(f_col)
          final_feature_columns_for_tf = base_tf_features +
unique_weather_time_tf

          for df_check in [df_train_final, df_val_final,
df_test_final]:
            for col_final in final_feature_columns_for_tf:
                    if col_final not in df_check.columns:
df_check[col_final] = 0
          print(f"Transformer 流程：最终用于序列的特征列
({len(final_feature_columns_for_tf)}          个          ):
{final_feature_columns_for_tf}")
        return df_train_final, df_val_final, df_test_final,
final_feature_columns_for_tf

    def                create_sequences_tf(df_with_features,
target_col_name, feature_cols_for_sequence, n_timesteps): #
使用新名
        # (与之前的 create_sequences_nwp_v2 逻辑一致)
          print(f"Transformer 流程：正在创建序列，回看：
{n_timesteps}, 特征: {feature_cols_for_sequence}")
        X, y = [], []
             missing_cols_in_df   =   [col   for   col   in
```

```
feature_cols_for_sequence      if      col      not      in
df_with_features.columns]
       if missing_cols_in_df: raise KeyError(f"创建序列时，以
下特征列在 DataFrame 中找不到: {missing_cols_in_df}")
        if target_col_name not in df_with_features.columns:
raise KeyError(f"目标列 '{target_col_name}' 在 DataFrame 中找
不到！")
                                    data_values              =
df_with_features[feature_cols_for_sequence].values
                                   target_values             =
df_with_features[target_col_name].values
       if len(data_values) <= n_timesteps:
           print(f"数据太短 ({len(data_values)}条)，无法创建长
度为{n_timesteps}的序列！")
           return np.array(X), np.array(y)
       for i in range(len(data_values) - n_timesteps):
           X.append(data_values[i:(i + n_timesteps), :])
           y.append(target_values[i + n_timesteps])
       return np.array(X), np.array(y)


   def       get_initial_history_tf(df_train_all_features,
test_period_start_time,                 n_timesteps_needed,
feature_cols_for_sequence): # 使用新名
       # (与之前的 get_initial_history_nwp_v2 逻辑一致)
          if   not   isinstance(df_train_all_features.index,
pd.DatetimeIndex):
              raise  TypeError("get_initial_history_tf  需要
df_train_all_features 的索引是 DatetimeIndex！")
           history_end_time   =   test_period_start_time   -
pd.Timedelta(minutes=15)
                            relevant_train_data           =
df_train_all_features[df_train_all_features.index        <=
history_end_time]
       if len(relevant_train_data) < n_timesteps_needed:
                raise  ValueError(f"训练数据不足以提供
{n_timesteps_needed}   条   初   始   历   史      (仅   找   到
{len(relevant_train_data)} 条在 {test_period_start_time} 之
前)。")
          missing_cols_in_hist_df   =   [col   for   col   in
feature_cols_for_sequence      if      col      not      in
relevant_train_data.columns]
       if missing_cols_in_hist_df:
              raise  KeyError(f"获取初始历史时，以下特征列在
relevant_train_data 中找不到: {missing_cols_in_hist_df}")
```

```
                        initial_history_df_slice          =
relevant_train_data[feature_cols_for_sequence].iloc[-
n_timesteps_needed:]
       return initial_history_df_slice.values

   def perform_iterative_forecast_for_period_tf( # 使用新名
                                                 trained_tf_mo
del, initial_history_features_array,
                                                 num_steps_to_
forecast, df_test_for_future_nwp_lookup,
                                                 period_start_
time_for_nwp, power_col_idx_in_features_list,
                                                 all_feature_n
ames_for_sequence_list):
                    #       (       与       之       前       的
perform_iterative_forecast_for_period_lstm_nwp_v2 逻辑一致)
       predictions_normalized = []
                         current_sequence_features         =
initial_history_features_array.copy()
                        current_time_for_prediction        =
pd.Timestamp(period_start_time_for_nwp)
                            N_ACTUAL_FEATURES_TF           =
current_sequence_features.shape[1]

            print(f"Transformer  流  程  :  开  始  为  进  行
{num_steps_to_forecast}     步     迭     代     预     测     ,     从
{current_time_for_prediction} 开始...")
       for i in range(num_steps_to_forecast):
                                       model_input        =
current_sequence_features.reshape(1,           N_TIMESTEPS,
N_ACTUAL_FEATURES_TF)
                             predicted_norm_power_step     =
trained_tf_model.predict(model_input, verbose=0)[0, 0]
           predictions_normalized.append(predicted_norm_pow
er_step)

           if i < num_steps_to_forecast - 1:
                            next_sequence_start_features   =
current_sequence_features[1:, :].copy()
                       next_actual_timestamp_for_features  =
current_time_for_prediction + pd.Timedelta(minutes=15)
               try:
                           future_features_for_step_series  =
df_test_for_future_nwp_lookup.loc[next_actual_timestamp_for
```

```
_features, all_feature_names_for_sequence_list]
                                future_features_for_step   =
future_features_for_step_series.values.copy()
                    future_features_for_step[power_col_idx_i
n_features_list] = predicted_norm_power_step
                                new_last_step_features   =
future_features_for_step
               except KeyError:
                                new_last_step_features   =
current_sequence_features[-1, :].copy()
                    new_last_step_features[power_col_idx_in_
features_list] = predicted_norm_power_step
                              current_sequence_features   =
np.vstack((next_sequence_start_features,
new_last_step_features.reshape(1, N_ACTUAL_FEATURES_TF)))
                             current_time_for_prediction   =
next_actual_timestamp_for_features
           if (i + 1) % 96 == 0: print(f"   Transformer 已完
成第 {(i + 1) // 96} 天的预测...")
       print(f"Transformer 周期预测完成！")
       return predictions_normalized

   def
inverse_transform_predictions_tf(predictions_norm_list,
fitted_target_scaler): # 使用新名
       # (与之前的 inverse_transform_predictions_v2 逻辑一致)
       if not predictions_norm_list: return []
                                predictions_2d             =
np.array(predictions_norm_list).reshape(-1, 1)
       if not hasattr(fitted_target_scaler, 'data_min_'):
           raise ValueError("用于逆转换的 Scaler 还没有被拟合过！
")
                         predictions_actual_scale_2d        =
fitted_target_scaler.inverse_transform(predictions_2d)
       return predictions_actual_scale_2d.flatten().tolist()

   # --- Transformer 模型构建函数 ---
   def positional_encoding(length, depth):
       depth = depth / 2
        positions = np.arange(length)[:, np.newaxis]     #
(seq, 1)
        depths = np.arange(depth)[np.newaxis, :] / depth #
(1, depth)
       angle_rates = 1 / (10000**depths)              # (1,
```

```
depth)
        angle_rads = positions * angle_rates             #
(pos, depth)
         pos_encoding = np.concatenate([np.sin(angle_rads),
np.cos(angle_rads)], axis=-1)
       return tf.cast(pos_encoding, dtype=tf.float32)

   class PositionalEmbedding(tf.keras.layers.Layer):
         def  __init__(self,  vocab_size_if_any,  d_model,
sequence_length): # vocab_size_if_any not used for numerical
           super().__init__()
           self.d_model = d_model
            # For numerical features, we often use a Dense
layer as a linear projection
           self.embedding = Dense(d_model, activation=None)
# No vocab_size needed
                                    self.pos_encoding      =
positional_encoding(length=sequence_length, depth=d_model)

       def call(self, x):
              length  =  tf.shape(x)[1]  #  sequence  length
(N_TIMESTEPS)
           x = self.embedding(x) # (batch, seq_len, d_model)
            # This factor sets the relative importance of
"word" versus "position" embeddings.
                x   *=   tf.math.sqrt(tf.cast(self.d_model,
tf.float32))
           x = x + self.pos_encoding[tf.newaxis, :length, :]
# Add positional encoding
           return x

   def transformer_encoder_block(d_model, num_heads, ff_dim,
dropout_rate=0.1):
       inputs = Input(shape=(None, d_model)) # (batch_size,
sequence_length, d_model)

       # Multi-Head Self-Attention
        attention = MultiHeadAttention(num_heads=num_heads,
key_dim=d_model // num_heads, dropout=dropout_rate)(inputs,
inputs, inputs) # Query, Value, Key are the same
       attention = Dropout(dropout_rate)(attention)
        attention = LayerNormalization(epsilon=1e-6)(inputs
+ attention) # Add & Norm (Residual connection)
```

```
        # Feed Forward Network
        outputs = Dense(ff_dim, activation="relu")(attention)
        outputs = Dense(d_model)(outputs)
        outputs = Dropout(dropout_rate)(outputs)
        outputs = LayerNormalization(epsilon=1e-6)(attention
+ outputs) # Add & Norm

        return Model(inputs=inputs, outputs=outputs)

    def      build_transformer_model(input_shape_timesteps,
input_shape_features,
                                    d_model, num_heads, ff_dim,
num_transformer_blocks, dropout_rate):
        """构建一个基于 Encoder 的 Transformer 模型用于单步预测"""
            inputs   =   Input(shape=(input_shape_timesteps,
input_shape_features)) # (N_TIMESTEPS, N_FEATURES_ACTUAL)

        # 1. Embedding + Positional Encoding
        # For numerical multivariate time series, use a Dense
layer to project features to d_model
        # then add positional encoding.
        x = Dense(d_model, activation=None)(inputs) # Project
N_FEATURES_ACTUAL to d_model

        # Create and add positional encoding
        # We need a custom layer or function to add positional
encoding if not using Embedding layer directly
        # For simplicity, we can add it directly if we compute
it matching batch and seq_len
        # A simpler PositionalEmbedding layer is used above.
          #  This  assumes  numerical  inputs  are  already
appropriately scaled.
        # We'll use a simplified approach: Dense projection
then add positional encoding created separately.

        # Adding positional encoding (simpler way for direct
numerical input)
        seq_len = input_shape_timesteps
        pos_encoding_tf = positional_encoding(length=seq_len,
depth=d_model)
        x = x + pos_encoding_tf # Broadcasting might work if
pos_encoding matches shape or is (seq_len, d_model)

        x = Dropout(dropout_rate)(x)
```

```
        # 2. Transformer Encoder Blocks
        for _ in range(num_transformer_blocks):
            # We can't directly reuse the Model instance from
transformer_encoder_block in Sequential easily.
            # Instead, build it functionally.
            # Multi-Head Self-Attention
                                          attn_output        =
MultiHeadAttention(num_heads=num_heads, key_dim=d_model //
num_heads, dropout=dropout_rate)(x, x, x)
                                    #       attn_output      =
Dropout(dropout_rate)(attn_output) # MHA already has dropout
                x  =  LayerNormalization(epsilon=1e-6)(x  +
attn_output) # Add & Norm

            # Feed Forward Network
            ffn_output = Dense(ff_dim, activation="relu")(x)
            ffn_output = Dense(d_model)(ffn_output)
            ffn_output = Dropout(dropout_rate)(ffn_output)
                x  =  LayerNormalization(epsilon=1e-6)(x  +
ffn_output) # Add & Norm

        # 3. Output Layer
          #  GlobalAveragePooling1D  reduces  the  sequence
dimension
                                           x                 =
GlobalAveragePooling1D(data_format="channels_last")(x)    #
(batch_size, d_model)
        x = Dropout(dropout_rate)(x)
        outputs = Dense(1, activation="sigmoid")(x) # Predict
next single step, scaled to [0,1]

        model = Model(inputs=inputs, outputs=outputs)
        return model

    # --- 主程序逻辑 ---
    def main_transformer_nwp(): # 版本号更新
        """完整执行 Transformer+NWP 的数据处理、模型训练、预测与保
存流程"""
        try:
               df_train_final,  df_val_final,  df_test_final,
final_feature_columns_for_tf = load_and_prepare_data_tf()

                                    power_normalized_idx      =
```

```
final_feature_columns_for_tf.index('power_normalized')
                                    N_FEATURES_ACTUAL       =
len(final_feature_columns_for_tf)

             print(f"\n Transformer 流程：正在创建训练和验证序列
(时间步长 N_TIMESTEPS={N_TIMESTEPS})...")
                             X_train_tf,     y_train_tf     =
create_sequences_tf(df_train_final.fillna(0),
                                                             t
arget_col_name='power_normalized',
                                                             f
eature_cols_for_sequence=final_feature_columns_for_tf,
                                                             n
_timesteps=N_TIMESTEPS)
                                X_val_tf,      y_val_tf      =
create_sequences_tf(df_val_final.fillna(0),
                                                           tar
get_col_name='power_normalized',
                                                           fea
ture_cols_for_sequence=final_feature_columns_for_tf,
                                                           n_t
imesteps=N_TIMESTEPS)

           if X_train_tf.shape[0] == 0 or X_val_tf.shape[0]
== 0 :
               print("Transformer 训练或验证数据序列创建失败。程
序中止。")
               return
                 print(f"Transformer 训练数据形状： X:
{X_train_tf.shape}, y: {y_train_tf.shape}")
                 print(f"Transformer 验证数据形状： X:
{X_val_tf.shape}, y: {y_val_tf.shape}")

               print("\n Transformer 流程：开始构建和训练
Transformer 模型...")
           transformer_model = build_transformer_model(
               input_shape_timesteps=N_TIMESTEPS,
               input_shape_features=N_FEATURES_ACTUAL,
               d_model=D_MODEL,
               num_heads=NUM_HEADS,
               ff_dim=FF_DIM,
                num_transformer_blocks=NUM_TRANSFORMER_BLOCK
S,
               dropout_rate=DROPOUT_RATE
```

```
            )
            transformer_model.compile(optimizer=tf.keras.opt
imizers.Adam(learning_rate=0.001,  clipnorm=1.0),  #  Added
clipnorm
                                      loss='mean_squared_error
',
                                      metrics=[tf.keras.metric
s.RootMeanSquaredError(name='rmse')])
            print("Transformer 模型结构：")
            transformer_model.summary()

                                        early_stopping       =
EarlyStopping(monitor='val_rmse',              patience=10,
restore_best_weights=True, verbose=1) # Monitor val_rmse
            reduce_lr = ReduceLROnPlateau(monitor='val_rmse',
factor=0.2, patience=5, min_lr=0.00001, verbose=1) # Monitor
val_rmse

            start_time = time.time()
            print(f"开始训练 Transformer 模型，共 {EPOCHS} 轮...")
                history  =  transformer_model.fit(X_train_tf,
y_train_tf,
                                      epochs=EPOCHS,
                                      batch_size=BATCH_SIZE,
                                      validation_data=(X_val_t
f, y_val_tf),
                                      callbacks=[early_stoppin
g, reduce_lr],
                                      verbose=1, shuffle=True)
            end_time = time.time()
            print(f"Transformer 模型训练完成！用时：{end_time -
start_time:.2f} 秒。")
            transformer_model.save(TRANSFORMER_MODEL_PATH)
                    print(f"Transformer  模 型 已 保 存 到 :
{TRANSFORMER_MODEL_PATH}")

                               df_train_power_for_scaler     =
pd.read_excel(POWER_DATA_EXCEL_PATH,
sheet_name=POWER_TRAIN_SHEET_INPUT)
                           if    'power_cleaned'    not    in
df_train_power_for_scaler.columns:
                    raise KeyError("用于 Scaler 的原始训练功率数据中
找不到 'power_cleaned' 列！")
```

```
            target_scaler = MinMaxScaler(feature_range=(0,
1))
            target_scaler.fit(df_train_power_for_scaler[['po
wer_cleaned']].dropna())
                                    joblib.dump(target_scaler,
TARGET_SCALER_TRANSFORMER_NWP_PATH)
                  print(f"目标逆转换的 MinMaxScaler 已在
'power_cleaned'     上     拟     合     并     保     存     到     :
{TARGET_SCALER_TRANSFORMER_NWP_PATH}")

           df_test_predictions_tf = df_test_final.copy()
           df_test_predictions_tf[final_feature_columns_for
_tf]                                                        =
df_test_predictions_tf[final_feature_columns_for_tf].fillna
(0)
           df_test_predictions_tf['predicted_power_normaliz
ed_tf_nwp'] = np.nan
           df_test_predictions_tf['predicted_power_actual_t
f_nwp'] = np.nan

                                        df_test_sorted       =
df_test_predictions_tf.sort_index()
                  if   not   isinstance(df_test_sorted.index,
pd.DatetimeIndex):
               if 'date' in df_test_sorted.columns:
                                    df_test_sorted['date']   =
pd.to_datetime(df_test_sorted['date'])
                                         df_test_sorted    =
df_test_sorted.set_index('date').sort_index()
                else: raise TypeError("测试集索引在识别周期前不
是日期时间类型，且找不到'date'列！")

                                       time_diffs        =
df_test_sorted.index.to_series().diff()
           is_new_period_start_mask = time_diffs.isnull() |
(time_diffs > pd.Timedelta(minutes=15))
                                  period_starts_indices      =
df_test_sorted.index[is_new_period_start_mask]

               print(f"\n Transformer 流程：在测试集中识别到
{len(period_starts_indices)} 个预测周期的开始点。")
                    if   not   period_starts_indices.empty:
print(period_starts_indices)
```

```
            num_steps_per_period = 7 * 24 * 4

            for start_dt_of_period in period_starts_indices:
                    print(f"\n---  Transformer+NWP 正在为从
{start_dt_of_period} 开始的7天周期进行预测 ---")
                try:
                                    initial_history_features   =
get_initial_history_tf(
                                      df_train_final.fillna(0),
start_dt_of_period,
                                                        N_TIMESTEPS,
final_feature_columns_for_tf
                    )
                except ValueError as e: print(f"  {e} 跳过此周
期。"); continue

                              df_test_current_period_lookup   =
df_test_sorted[df_test_sorted.index >= start_dt_of_period]
                if df_test_current_period_lookup.empty:
                     print(f"测试集中找不到 {start_dt_of_period}
之后的数据。跳过此周期。"); continue

                               normalized_preds_for_period   =
perform_iterative_forecast_for_period_tf(
                                             transformer_model,
initial_history_features, num_steps_per_period,
                                df_test_current_period_lookup,
start_dt_of_period,
                                          power_normalized_idx,
final_feature_columns_for_tf
                )

                 current_period_end_time = start_dt_of_period
+ pd.Timedelta(days=7) - pd.Timedelta(minutes=15)
                           target_timestamps_in_test_period  =
df_test_sorted.loc[start_dt_of_period                        :
current_period_end_time].index

                                          actual_pred_len     =
len(normalized_preds_for_period)
                                            target_ts_len     =
len(target_timestamps_in_test_period)

                 if actual_pred_len == 0 and target_ts_len >
```

```
0 :
                    print(f"周期 {start_dt_of_period} 没有生成预测值，但期望有 {target_ts_len} 个。"); continue
                if actual_pred_len == 0 and target_ts_len == 0:
                    print(f" 周期 {start_dt_of_period} 目标时段为空，跳过填充。"); continue
                if actual_pred_len > target_ts_len:
                                normalized_preds_for_period  = normalized_preds_for_period[:target_ts_len]
                elif actual_pred_len < target_ts_len:
                            target_timestamps_in_test_period  = target_timestamps_in_test_period[:actual_pred_len]

                if not target_timestamps_in_test_period.empty and actual_pred_len > 0 :
                    df_test_predictions_tf.loc[target_timestamps_in_test_period, 'predicted_power_normalized_tf_nwp'] = normalized_preds_for_period
                                actual_scale_preds_for_period  = inverse_transform_predictions_tf(
                                    normalized_preds_for_period, target_scaler
                    )
                    df_test_predictions_tf.loc[target_timestamps_in_test_period,   'predicted_power_actual_tf_nwp']   = actual_scale_preds_for_period
                                print(f"   Transformer+NWP 周期 {start_dt_of_period} 的预测已填充。")
                else:
                     print(f"  周期 {start_dt_of_period} 预测结果为空或在测试集中找不到对应的目标时间戳。")

            print(f"\n Transformer 流程：正在将预测结果保存到 Excel 文件: '{OUTPUT_EXCEL_PATH}'...")
            with pd.ExcelWriter(OUTPUT_EXCEL_PATH) as writer:
                df_train_final.reset_index().to_excel(writer, sheet_name=POWER_TRAIN_SHEET_INPUT, index=False) # 使用输入表名作参考
                df_val_final.reset_index().to_excel(writer, sheet_name=POWER_VAL_SHEET_INPUT, index=False) # 保存处理后的验证集
                df_test_predictions_tf.reset_index().to_excel(writer, sheet_name=POWER_TEST_SHEET_INPUT, index=False)
```

```
            print(f"Transformer+NWP 预测结果已成功保存！看
'{OUTPUT_EXCEL_PATH}' 吧！")

        except FileNotFoundError as e:
            print(f"Transformer 流程：找不到 Excel 文件
'{e.filename}'。")
        except KeyError as e:
            print(f"Transformer 流程:DataFrame 中缺少列 '{e}'。
")
            traceback.print_exc()
        except Exception as e:
            print(f"Transformer 流程：发生意料之外的错误：{e}")
            traceback.print_exc()

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

问题 4 IDW 的核心代码

```
    import pandas as pd
    import numpy as np
    import os
    from math import radians, sin, cos, sqrt, atan2

    # --- 文件路径和全局设定 ---
    BASE_WEATHER_DATA_FOLDER = 'problem4_weather_data/' # 存
放 9 个天气文件的文件夹
    FILE_PREFIX = '_weather_data.xlsx' # 文件名除了数字外的部分
    OUTPUT_IDW_EXCEL_PATH                                    =
os.path.join(BASE_WEATHER_DATA_FOLDER,
'fused_weather_by_idw.xlsx')

    # 源数据点坐标 (纬度, 经度) - 数据 1 是我们的目标点（光伏电站）
    #  但为了让输出文件代表电站位置的天气，我们最终会生成针对数据 1 坐
标的 IDW 值。
    SOURCE_COORDS = {
        1: (37.42, 122.17), # 光伏电站 (目标点)
        2: (36.42, 121.17),
        3: (36.42, 122.17),
        4: (36.42, 123.17),
        5: (37.42, 121.17),
        6: (37.42, 123.17),
        7: (38.42, 121.17),
```

```
        8: (38.42, 122.17),
        9: (38.42, 123.17),
    }
    TARGET_POINT_ID = 1 # 我们要为数据点 1（光伏电站）的位置估算天
气
    REFERENCE_POINT_IDS = list(range(2, 10)) # 用数据点 2 到 9
作为参考点

    IDW_POWER_K = 2 # IDW 公式中的幂指数 k

    SHEET_NAMES = ['1h', '1d'] # 要处理的工作表名
    COLUMN_TO_IGNORE = 'weather_code (wmo code)' # 这个列不参
与 IDW，直接取目标点的

    def haversine_distance(lat1, lon1, lat2, lon2):
        """计算两个经纬度点之间的球面距离 (公里)"""
        R = 6371.0  # 地球平均半径 (公里)

        lat1_rad = radians(lat1)
        lon1_rad = radians(lon1)
        lat2_rad = radians(lat2)
        lon2_rad = radians(lon2)

        dlon = lon2_rad - lon1_rad
        dlat = lat2_rad - lat1_rad

        a = sin(dlat / 2)**2 + cos(lat1_rad) * cos(lat2_rad)
* sin(dlon / 2)**2
        c = 2 * atan2(sqrt(a), sqrt(1 - a))

        distance = R * c
        return distance

    def idw_interpolate(target_coords, reference_coords_map,
reference_values_map, power_k=2):
        """
        执行 IDW 插值。
        target_coords: (lat, lon) 目标点的坐标。
        reference_coords_map: {point_id: (lat, lon)} 参考点坐
标字典。
         reference_values_map: {point_id: value} 参考点数值字
典。
        power_k: IDW 的幂指数。
        """
```

```
        numerator = 0.0
        denominator = 0.0

          # 确保 reference_values_map 中的键也存在于
reference_coords_map
        valid_points_for_idw = 0

                  for      point_id,      ref_value      in
reference_values_map.items():
            if pd.isna(ref_value): # 跳过 NaN 值
                continue

            ref_coords = reference_coords_map.get(point_id)
            if ref_coords is None:
                print(f"警告：找不到参考点 {point_id} 的坐标，将
跳过此点进行 IDW。")
                continue

                dist = haversine_distance(target_coords[0],
target_coords[1], ref_coords[0], ref_coords[1])

            if dist == 0: # 如果目标点恰好是某个参考点（理论上不应
发生在此场景，因为我们用 2-9 估算 1）
                              # 或者如果多个参考点在同一位置（不太可能）
                              # 为避免除零，如果距离为 0，直接返回该点
的值（或处理冲突）
                              # 在本场景，如果 dist=0 意味着目标点就是
某个参考点，这对于用 2-9 估算 1 来说不适用
                             # 但如果 k 很大，dist 很小，权重会非常大。
                              # 对于 IDW，如果一个参考点与目标点距离极
小，它应该主导结果。
                             # 为避免除零，给一个极小距离值
                dist = 1e-6

            weight = 1.0 / (dist ** power_k)
            numerator += ref_value * weight
            denominator += weight
            valid_points_for_idw +=1

        if denominator == 0 or valid_points_for_idw == 0: #
如果所有参考点值都是 NaN 或没有有效参考点
            return np.nan
        return numerator / denominator
```

```
    def process_all_weather_files_with_idw():
        """
        读取所有9个天气文件，对每个时间点和每个数值变量执行IDW插值，
        生成代表光伏电站位置（数据 1 的位置）的融合天气数据。
        """
        print("开始 IDW 天气融合")

        target_coordinates = SOURCE_COORDS[TARGET_POINT_ID]
        reference_coordinates = {pid: SOURCE_COORDS[pid] for
pid in REFERENCE_POINT_IDS}

        # 用于存储最终融合结果的字典，键是工作表名，值是 DataFrame
        fused_data_sheets = {}

         # 我们需要一个“模板”DataFrame 的索引和列结构，可以从数据文
件 1 中获取
        # 但因为数据文件 1 的值我们是要估算的，所以我们只用它的时间和
weather_code
        try:
                                    df_template_file_path      =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{TARGET_POINT_ID}{FILE_PREFIX}")
            if not os.path.exists(df_template_file_path):
                raise FileNotFoundError(f"找不到目标点（数据 1）
的天气文件 '{df_template_file_path}' 作为模板！")

                                           xls_template       =
pd.ExcelFile(df_template_file_path)
        except Exception as e:
            print(f"读取模板文件时出错：{e}")
            return

        for sheet_name in SHEET_NAMES: # 分别处理 '1h' 和 '1d'
            print(f"\n--- 正在处理工作表: '{sheet_name}' ---")
            if sheet_name not in xls_template.sheet_names:
                        print(f"警告：模板文件中找不到工作表
'{sheet_name}'，跳过此表。")
                continue

            df_target_template = pd.read_excel(xls_template,
sheet_name=sheet_name)
            if 'time' not in df_target_template.columns:
                raise KeyError(f"模板文件工作表 '{sheet_name}'
中找不到 'time' 列！")
```

```
                df_target_template['time']     =
pd.to_datetime(df_target_template['time'])
                df_target_template.set_index('time',
inplace=True)

        # 初始化结果 DataFrame，结构与模板一致，但数值为空，稍后
填充 IDW 结果
                                    df_fused_sheet       =
pd.DataFrame(index=df_target_template.index)

        # 读取所有参考点（数据 2 到 9）的当前工作表数据
        all_reference_dfs_current_sheet = {}
        for ref_id in REFERENCE_POINT_IDS:
                                        file_path      =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{ref_id}{FILE_PREFIX}")
            try:
                if not os.path.exists(file_path):
                    print(f"警告：找不到参考点 {ref_id} 的天
气文件 '{file_path}'，将忽略此数据源。")
                    continue
                xls_ref = pd.ExcelFile(file_path)
                if sheet_name in xls_ref.sheet_names:
                    df_ref = pd.read_excel(xls_ref,
sheet_name=sheet_name)
                    if 'time' not in df_ref.columns:
                        print(f"警告：参考点 {ref_id} 的文
件 '{file_path}' 工作表 '{sheet_name}' 中找不到 'time' 列，跳过。
")
                        continue
                                        df_ref['time']   =
pd.to_datetime(df_ref['time'])
                    df_ref.set_index('time', inplace=True)
                    all_reference_dfs_current_sheet[ref_i
d] = df_ref
                else:
                    print(f"警告：参考点 {ref_id} 的文件
'{file_path}' 中找不到工作表 '{sheet_name}'。")
            except Exception as e:
                print(f"读取参考点 {ref_id} 的文件
'{file_path}' 时出错：{e}")

        if not all_reference_dfs_current_sheet:
            print(f"没有成功加载任何参考点数据用于工作表
```

```
'{sheet_name}' 的 IDW 计算！")
                fused_data_sheets[f"{sheet_name}_idw_fused"]
= df_target_template # 保存原始模板以防万一
                continue

            # 确定要进行 IDW 的数值列 (以模板文件的列为准，排除忽略
列)
            # 同时确保这些列也存在于至少一个参考文件中，否则没法插值
            numeric_cols_for_idw = []
            for col in df_target_template.columns:
                if col.lower() != COLUMN_TO_IGNORE.lower()
and pd.api.types.is_numeric_dtype(df_target_template[col]):
                    # 检查此列是否存在于任何一个已加载的参考
DataFrame 中
                    col_exists_in_any_ref = any(col in
ref_df.columns for ref_df in
all_reference_dfs_current_sheet.values())
                    if col_exists_in_any_ref:
                        numeric_cols_for_idw.append(col)
                    else:
                        print(f" 列 '{col}' 在所有参考数据中均未
找到，将不进行 IDW 插值，直接使用模板值（如果有）。")
                    if col in df_target_template.columns:
# 直接复制模板（数据 1）的值
                        df_fused_sheet[col] =
df_target_template[col]

            print(f" 将对以下数值列进行 IDW 插值：
{numeric_cols_for_idw}")

            # 直接复制非 IDW 列（比如 weather_code）从目标点（数据
1）的模板
                if COLUMN_TO_IGNORE in
df_target_template.columns:
                    df_fused_sheet[COLUMN_TO_IGNORE] =
df_target_template[COLUMN_TO_IGNORE]
            else:
                print(f"警告：列 '{COLUMN_TO_IGNORE}' 在模板文
件中不存在，将不会出现在融合结果中。")

            # 对每个时间点进行 IDW 插值
                for timestamp_idx, _ in
enumerate(df_target_template.index):
                    current_timestamp =
```

```
df_target_template.index[timestamp_idx]
                            if (timestamp_idx + 1) %
(len(df_target_template.index) // 10 if
len(df_target_template.index) > 10 else 1) == 0 : # 打印进度
                        print(f"    正在处理工作表 '{sheet_name}'
的 时 间 点 : {current_timestamp}
({timestamp_idx+1}/{len(df_target_template.index)})")

                                for col_to_interpolate in
numeric_cols_for_idw:
                    reference_values_at_ts = {}
                     actual_ref_points_for_this_col_ts = {} #
存储实际用于当前插值的参考点坐标

                                    for ref_id, ref_df in
all_reference_dfs_current_sheet.items():
                                    if col_to_interpolate in
ref_df.columns and current_timestamp in ref_df.index:
                                reference_values_at_ts[ref_id] =
ref_df.loc[current_timestamp, col_to_interpolate]
                               actual_ref_points_for_this_col_ts
[ref_id] = reference_coordinates[ref_id] # 使用预存的坐标
                          # else: 即使某个参考点没有这个时间或列，
IDW 函数内部会跳过 NaN

                    if reference_values_at_ts: # 确保至少有一个
参考值
                                                    #  使 用
actual_ref_points_for_this_col_ts 作为坐标图，因为它只包含有值的
点
                                                  idw_value =
idw_interpolate(target_coordinates,
                                                    actual_ref_
points_for_this_col_ts,
                                                    reference_v
alues_at_ts,
                                                    power_k=IDW
_POWER_K)
                        df_fused_sheet.loc[current_timestamp,
col_to_interpolate] = idw_value
                        else: # 如果所有参考点都没有这个时间或列的数
据，则为 NaN
                        df_fused_sheet.loc[current_timestamp,
col_to_interpolate] = np.nan
```

```
            # 对于那些没有被 IDW 插值的数值列（因为可能不存在于参考数据中），我们之前已经从模板复制了

            # 当前逻辑是：如果某列要 IDW 但参考数据没有，则为 NaN；如果不 IDW 且模板有，则为模板值。

              fused_data_sheets[f"{sheet_name}_idw_fused"] = df_fused_sheet.copy() # 保存当前 sheet 的融合结果
            print(f"  工作表 '{sheet_name}' 处理完成！")

        # 将所有融合后的工作表保存到一个新的 Excel 文件
        if fused_data_sheets:
                print(f"\n 正在将所有融合后的天气数据保存到：{OUTPUT_IDW_EXCEL_PATH}")
             with pd.ExcelWriter(OUTPUT_IDW_EXCEL_PATH) as writer:
                            for sheet_name_out, df_out in fused_data_sheets.items():
                    # 在保存前，可以把索引（原来的 time 列）变回普通列，如果需要的话
                        df_out.reset_index().to_excel(writer, sheet_name=sheet_name_out, index=False)
            print("所有融合天气数据已成功保存!! ")
        else:
            print(" 没有任何工作表被成功处理和融合，未生成输出文件。")

    if __name__ == "__main__":
        # 确保文件夹存在
        if not os.path.isdir(BASE_WEATHER_DATA_FOLDER):
                  print(f" 错误！找不到天气数据文件夹：'{BASE_WEATHER_DATA_FOLDER}'")
              print("先创建这个文件夹，并把 9 个 weather_data.xlsx 文件放进去！")
        else:
            # 检查是否至少有数据文件 1（模板）和至少一个其他数据文件
                                        path_data1          = os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{TARGET_POINT_ID}{FILE_PREFIX}")
                                    found_other_data        = any(os.path.exists(os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{i}{FILE_PREFIX}")) for i in REFERENCE_POINT_IDS)
```

```
        if not os.path.exists(path_data1):
                print(f"错误！找不到目标点（数据 1）的天气文件
'{path_data1}' 作为模板！")
        elif not found_other_data:
             print(f"错误！找不到任何参考点（数据 2-9）的天气文
件用于 IDW 插值！")
        else:
            process_all_weather_files_with_idw()
```

问题 4 MLR 的核心代码

```
    import pandas as pd
    import numpy as np
    import os
    from sklearn.linear_model import LinearRegression
    from sklearn.model_selection import train_test_split #
用于可能的模型验证，但这里直接用全部数据训练和预测
    from sklearn.impute import SimpleImputer # 用于处理特征中
的缺失值

    # --- 文件路径和全局设定 ---
    BASE_WEATHER_DATA_FOLDER = 'problem4_weather_data/'
    FILE_PREFIX = '_weather_data.xlsx'
    OUTPUT_MLR_EXCEL_PATH                                    =
os.path.join(BASE_WEATHER_DATA_FOLDER, 'weather_LR.xlsx')

    TARGET_POINT_ID = 1 # 数据 1 是我们的目标点（光伏电站）
    REFERENCE_POINT_IDS = list(range(2, 10)) # 数据点 2 到 9 作
为回归模型的输入特征

    SHEET_NAMES = ['1h', '1d']
    COLUMN_TO_IGNORE = 'weather_code (wmo code)' # 这个列不参
与回归，直接取目标点的

    def  train_and_predict_variable_with_mlr(variable_name,
all_dfs_current_sheet):
        """
        针对单个气象变量，训练一个多点线性回归模型并进行预测。
        all_dfs_current_sheet: 包含 9 个 DataFrame 的列表，df[0]
是目标点(数据 1)，df[1:]是参考点(数据 2-9)。
        """
        print(f"    正在为变量 '{variable_name}' 训练和预测...")
```

```
        df_target = all_dfs_current_sheet[0] # 数据 1 是目标
         dfs_reference = all_dfs_current_sheet[1:] # 数据 2-9
是特征源

        # 准备目标 y (来自数据 1)
        if variable_name not in df_target.columns:
            print(f"      警告：目标变量 '{variable_name}' 在数
据文件 1 中不存在，跳过此变量。")
            return None # 或者返回一个全 NaN 的 Series
        y_series = df_target[variable_name].copy()

        # 准备特征 X (来自数据 2 到数据 9 的同一个变量)
        feature_df_list = []
        feature_col_names = []
        for i, ref_df in enumerate(dfs_reference):
            source_id = REFERENCE_POINT_IDS[i] # 获取真实的源
ID (2 到 9)
            if variable_name in ref_df.columns:
                    #  为特征列命名以区分来源，例如  'var_src2',
'var_src3'
                                          feature_col_name    =
f"{variable_name}_src{source_id}"
                feature_df_list.append(ref_df[variable_name]
.rename(feature_col_name))
                feature_col_names.append(feature_col_name)
            else:
                 # 如果某个参考文件缺少这个变量，我们可以创建一个全
NaN 的列，后续用 imputer 处理
                print(f"   提示：参考数据源 {source_id} 缺少变量
'{variable_name}'，将创建 NaN 列。")
                # 需要一个与 y_series 索引对齐的 NaN 序列
                           nan_series   =   pd.Series(np.nan,
index=y_series.index,
name=f"{variable_name}_src{source_id}")
                feature_df_list.append(nan_series)
                feature_col_names.append(f"{variable_name}_s
rc{source_id}")

        if not feature_df_list:
            print(f"      警告：变量 '{variable_name}' 在所有参
考数据源中均未找到，无法进行回归，跳过。")
             return  y_series  #  返回原始目标值，或者 None/NaN
Series
```

```
        X_df = pd.concat(feature_df_list, axis=1)

        # 对齐 y 和 X 的索引，并处理 NaN 值
        # 为了训练，我们需要在 y 非 NaN 的地方进行
                                combined_for_train                =
X_df.join(y_series.rename('target_y'), how='inner') # 只保留都有数据的行
        y_train = combined_for_train['target_y']
        X_train_raw = combined_for_train[feature_col_names]

        # 清理训练数据中的 NaN
        # 1. 如果 y_train 是 NaN，这些行不能用于训练
        valid_y_train_mask = y_train.notna()
        y_train = y_train[valid_y_train_mask]
        X_train_raw = X_train_raw[valid_y_train_mask]

        # 2. 处理 X_train_raw 中的 NaN (例如，用均值填充)
        #    我们应该在完整的 X_df 上 fit_transform imputer，然后用它 transform X_train_raw 和后续的 X_predict
        imputer = SimpleImputer(strategy='mean')
                                     X_df_imputed                 =
pd.DataFrame(imputer.fit_transform(X_df),
columns=X_df.columns, index=X_df.index)

        X_train = X_df_imputed.loc[X_train_raw.index] # 从完整填充后的 X 中取出训练部分

        if X_train.empty or y_train.empty or len(X_train) <
len(feature_col_names) + 1 : # 确保有足够数据训练
            print(f"      警告：变量 '{variable_name}' 清理 NaN 后没有足够的数据进行训练 ({len(X_train)}行)，将返回原始目标值（如果可用）或 NaN。")
            # 可以返回原始的 y_series，或者一个全 NaN 的序列
            return pd.Series(np.nan, index=df_target.index,
name=variable_name)

        # 训练线性回归模型
        model = LinearRegression()
        try:
            model.fit(X_train, y_train)
            print(f"      变量 '{variable_name}' 模型训练完成。")
        except Exception as e_fit:
```

```
            print(f"      变量 '{variable_name}' 模型训练失败:
{e_fit}。将返回 NaN。")
            return pd.Series(np.nan, index=df_target.index,
name=variable_name)

        # 使用训练好的模型对整个时间段的 X_df_imputed 进行预测
        predicted_values = model.predict(X_df_imputed)
           predicted_series  =  pd.Series(predicted_values,
index=X_df_imputed.index, name=variable_name)

        return predicted_series

    def process_all_weather_files_with_mlr():
        """
        读取所有 9 个天气文件，对每个时间点和每个数值变量执行多点线性
回归，
        生成代表光伏电站位置（数据 1 的位置）的融合天气数据。
        """
        print(" 开始多点线性回归天气融合！")

        fused_data_sheets = {} # 用于存储最终融合结果

        # 1. 先加载所有 9 个文件的数据到内存中，按 sheet 和文件 ID 组
织
          all_data_by_sheet_and_id  =  {sheet_name:  {}  for
sheet_name in SHEET_NAMES}

        # 加载数据文件 1（目标点/模板）
                                target_file_path            =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{TARGET_POINT_ID}{FILE_PREFIX}")
        if not os.path.exists(target_file_path):
            raise FileNotFoundError(f"找不到目标点（数据 1）的天
气文件 '{target_file_path}'！")
        xls_target = pd.ExcelFile(target_file_path)

        for sheet_name in SHEET_NAMES:
            if sheet_name not in xls_target.sheet_names:
                    print(f"警告：目标文件 1 中找不到工作表
'{sheet_name}'，将无法处理此表。")
                continue
                df_target_sheet  =  pd.read_excel(xls_target,
sheet_name=sheet_name)
            if 'time' not in df_target_sheet.columns:
```

```
                raise KeyError(f"目标文件 1 '{sheet_name}' 中找
不到 'time' 列！")
                                df_target_sheet['time']     =
pd.to_datetime(df_target_sheet['time'])
            df_target_sheet.set_index('time', inplace=True)
            all_data_by_sheet_and_id[sheet_name][TARGET_POIN
T_ID] = df_target_sheet

        # 加载参考点（数据 2-9）的数据
        for ref_id in REFERENCE_POINT_IDS:
                                            file_path        =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{ref_id}{FILE_PREFIX}")
            if not os.path.exists(file_path):
                  print(f"警告：找不到参考点 {ref_id} 的天气文件
'{file_path}'，将忽略此数据源。")
                continue
            try:
                xls_ref = pd.ExcelFile(file_path)
                for sheet_name in SHEET_NAMES:
                    if sheet_name in xls_ref.sheet_names and
sheet_name in all_data_by_sheet_and_id: # 确保目标表也存在
                        df_ref_sheet = pd.read_excel(xls_ref,
sheet_name=sheet_name)
                        if 'time' not in df_ref_sheet.columns:
                              print(f"警告：参考点 {ref_id} 文件
'{sheet_name}' 中找不到 'time' 列。")
                            continue
                                        df_ref_sheet['time']  =
pd.to_datetime(df_ref_sheet['time'])
                                df_ref_sheet.set_index('time',
inplace=True)
                        all_data_by_sheet_and_id[sheet_name][
ref_id] = df_ref_sheet
            except Exception as e:
                print(f"读取参考点 {ref_id} 文件时出错: {e}")

        # 2. 对每个工作表进行处理
        for sheet_name in SHEET_NAMES:
                          if    TARGET_POINT_ID    not    in
all_data_by_sheet_and_id.get(sheet_name, {}):
                print(f"工作表 '{sheet_name}' 在数据文件 1 中缺失
或加载失败，跳过此表。")
                continue
```

```
                                    df_target_template      =
all_data_by_sheet_and_id[sheet_name][TARGET_POINT_ID]
                                        df_fused_sheet      =
pd.DataFrame(index=df_target_template.index) # 初始化结果 DF
            print(f"\n--- 正在处理工作表: '{sheet_name}' ---")

            # 获取当前工作表所有点的数据列表，确保顺序是 [df_data1,
df_data2, ..., df_data9]
            # 其中 df_data1 是我们用 y 的，df_data2-9 是我们用 X 的
            # 我们把 TARGET_POINT_ID 放到列表第一个位置
                            current_sheet_dfs_ordered     =
[all_data_by_sheet_and_id[sheet_name].get(TARGET_POINT_ID)]
            for ref_id in REFERENCE_POINT_IDS:
                current_sheet_dfs_ordered.append(all_data_by
_sheet_and_id[sheet_name].get(ref_id,
pd.DataFrame(index=df_target_template.index))) # 如果某个参考
文件缺失，用空 DF（带索引）

            # 确定要进行回归的数值列
            numeric_cols_for_mlr = []
            for col in df_target_template.columns:
                 if col.lower() != COLUMN_TO_IGNORE.lower()
and pd.api.types.is_numeric_dtype(df_target_template[col]):
                     # 确保此列也存在于至少一个参考文件中，才有意义
去构建 X
                    col_exists_in_any_ref = any(
                         col in ref_df.columns for ref_df in
current_sheet_dfs_ordered[1:] if ref_df is not None and not
ref_df.empty
                    )
                    if col_exists_in_any_ref:
                        numeric_cols_for_mlr.append(col)
                    else: # 如果参考文件中都没有这个数值列，就直接
用数据 1 的原始值
                         print(f"列 '{col}' 在所有参考数据中均未
找到，将直接使用数据文件 1 的原始值。")
                        if col in df_target_template.columns:
                                        df_fused_sheet[col]  =
df_target_template[col]

                print(f"    将对以下数值列进行多点线性回归预测:
{numeric_cols_for_mlr}")
```

```
            # 直接复制非回归列（比如 weather_code）从目标点（数据
1）
                                if      COLUMN_TO_IGNORE      in
df_target_template.columns:
                            df_fused_sheet[COLUMN_TO_IGNORE]   =
df_target_template[COLUMN_TO_IGNORE]
            else:
                print(f"警告：列 '{COLUMN_TO_IGNORE}' 在数据文
件 1 中不存在，将不会出现在融合结果中。")

            # 对每个数值列训练模型并预测
            for var_name in numeric_cols_for_mlr:
                                        predicted_series     =
train_and_predict_variable_with_mlr(var_name,
current_sheet_dfs_ordered)
                if predicted_series is not None:
                                    df_fused_sheet[var_name]   =
predicted_series
                else: # 如果预测失败，可以考虑用原始值填充或留空
                    print(f"变量 '{var_name}' 预测失败，将尝试使
用数据文件 1 的原始值（如果存在）。")
                    if var_name in df_target_template.columns:
                                    df_fused_sheet[var_name]  =
df_target_template[var_name]
                    else:
                        df_fused_sheet[var_name] = np.nan

                fused_data_sheets[f"{sheet_name}_mlr_fused"] =
df_fused_sheet.copy()
            print(f"  工作表 '{sheet_name}' 处理完成！")

        # 3. 保存结果
        if fused_data_sheets:
                print(f"\n 正在将所有融合后的天气数据保存到：
{OUTPUT_MLR_EXCEL_PATH}")
                with  pd.ExcelWriter(OUTPUT_MLR_EXCEL_PATH)  as
writer:
                            for   sheet_name_out,   df_out   in
fused_data_sheets.items():
                        df_out.reset_index().to_excel(writer,
sheet_name=sheet_name_out, index=False)
            print("所有多点线性回归融合的天气数据已成功保存！")
        else:
            print(" 没有任何工作表被成功处理和融合（MLR 方法），未生
```

成输出文件。")

```
    if __name__ == "__main__":
        if not os.path.isdir(BASE_WEATHER_DATA_FOLDER):
                  print(f"错误！找不到天气数据文件夹：
'{BASE_WEATHER_DATA_FOLDER}'")
        else:
            try:
                process_all_weather_files_with_mlr()
            except Exception as e_main:
                print(f"主程序发生严重错误：{e_main}")
                traceback.print_exc()
```

问题 4 XGboost 的核心代码

```
    import pandas as pd
    import numpy as np
    import os
    import xgboost as xgb
    from sklearn.model_selection import train_test_split #
用于可能的模型验证或早停
    from sklearn.impute import SimpleImputer # 用于处理特征中
的缺失值
    import joblib # 如果需要保存训练好的 XGBoost 模型
    import traceback

    # --- 文件路径和全局设定（与之前 MLR 版本类似）---
    BASE_WEATHER_DATA_FOLDER = 'problem4_weather_data/'
    FILE_PREFIX = '_weather_data.xlsx'
    OUTPUT_XGB_EXCEL_PATH                                    =
os.path.join(BASE_WEATHER_DATA_FOLDER, 'weather_XG.xlsx') #
文件名体现 XGBoost

    TARGET_POINT_ID = 1
    REFERENCE_POINT_IDS = list(range(2, 10))

    SHEET_NAMES = ['1h', '1d']
    COLUMN_TO_IGNORE = 'weather_code (wmo code)'

    # XGBoost 模型参数
    XGB_PARAMS = {
        'objective': 'reg:squarederror', # 回归任务，目标是最小
化平方误差
```

```
        'n_estimators': 100,              # 树的数量（迭代次数）
        'learning_rate': 0.1,             # 学习率
        'max_depth': 5,                   # 每棵树的最大深度
        'subsample': 0.8,                 # 训练每棵树时，随机采样的比例
        'colsample_bytree': 0.8,          # 构建每棵树时，列（特征）的采样比例
        'random_state': 42,               # 随机种子，保证结果可复现
        'n_jobs': -1                      # 使用所有可用的 CPU 核心
    }

    def train_and_predict_variable_with_xgboost(variable_name, all_dfs_current_sheet):
        """
        针对单个气象变量，训练一个 XGBoost 回归模型并进行预测。
        all_dfs_current_sheet: 包含 9 个 DataFrame 的列表，df[0] 是目标点(数据 1)，df[1:]是参考点(数据 2-9)。
        """
         print(f"      正在为变量 '{variable_name}' 训练和预测 (XGBoost)...")

        df_target = all_dfs_current_sheet[0]
        dfs_reference = all_dfs_current_sheet[1:]

        if variable_name not in df_target.columns:
            print(f"  警告：目标变量 '{variable_name}' 在数据文件 1 中不存在，跳过。")
            return pd.Series(np.nan, index=df_target.index, name=variable_name) # 返回全 NaN 序列

        y_series = df_target[variable_name].copy()

        feature_df_list = []
        feature_col_names = []
        for i, ref_df in enumerate(dfs_reference):
            source_id = REFERENCE_POINT_IDS[i]
                                    feature_col_name       = f"{variable_name}_src{source_id}"
            feature_col_names.append(feature_col_name)
            if ref_df is not None and not ref_df.empty and variable_name in ref_df.columns:
```

```
                feature_df_list.append(ref_df[variable_name]
.rename(feature_col_name))
            else:
                print(f"    提示：参考数据源 {source_id} 缺少变
量 '{variable_name}' 或数据为空，将创建 NaN 列。")
                    # 创建与 y_series 索引对齐的 NaN 序列，以便后续
imputer 能正确处理所有时间点
                        nan_series = pd.Series(np.nan,
index=y_series.index, name=feature_col_name)
                feature_df_list.append(nan_series)

        if not feature_df_list:
            print(f"      警告：变量 '{variable_name}' 在所有参
考数据源中均未找到特征，返回原始目标值。")
            return y_series # 或者返回 NaN Series

        X_df_raw = pd.concat(feature_df_list, axis=1)

        # 对齐 y 和 X 的索引
        # 我们需要用 y_series 中非 NaN 的部分来训练，并用 X_df_raw 中
对应的行
          # X_df_raw 和 y_series 应该已经通过 concat 和原始的
df_target 对齐了索引

        # 准备训练数据：只用 y_series 中非 NaN 的行进行训练
        train_mask = y_series.notna()
        y_train = y_series[train_mask]
          X_train_for_fit = X_df_raw[train_mask] # 取出对应
y_train 的 X 行

        # 处理训练特征中的 NaN (用均值填充)
          # Imputer 应该在 X_train_for_fit 上 fit，然后用它来
transform X_train_for_fit 和 完整的 X_df_raw
        imputer = SimpleImputer(strategy='mean')

           if X_train_for_fit.empty or y_train.empty or
X_train_for_fit.isnull().all().all(): # 如果全是 NaN 或者没数据
            print(f"      警告：变量 '{variable_name}' 清理 NaN
后没有有效数据进行训练，将返回原始目标值（如果可用）或 NaN。")
             return pd.Series(np.nan, index=df_target.index,
name=variable_name)

        # 在可能包含 NaN 的训练特征上拟合 imputer
                            X_train_imputed_fit            =
```

```
imputer.fit_transform(X_train_for_fit)

        # 训练 XGBoost 模型
         # XGBoost 可以处理输入特征中的 NaN（通过特定方式），但为了流程统一和可控性，我们先填充
            # 如果不填充，可以设置 xgb_model = xgb.XGBRegressor(**XGB_PARAMS, missing=np.nan)
        xgb_model = xgb.XGBRegressor(**XGB_PARAMS)

        try:
               # print(f"        X_train_imputed_fit shape: {X_train_imputed_fit.shape},         y_train         shape: {y_train.shape}")
            if X_train_imputed_fit.shape[0] < 1: # 确保有样本
                 raise ValueError("训练样本为空")
            xgb_model.fit(X_train_imputed_fit, y_train)
             print(f"      变量 '{variable_name}' XGBoost 模型训练完成。")
        except Exception as e_fit:
             print(f"      变量 '{variable_name}' XGBoost 模型训练失败: {e_fit}。将返回 NaN。")
             return pd.Series(np.nan, index=df_target.index, name=variable_name)

        # 使用训练好的模型对整个时间段的 X_df_raw (填充后) 进行预测
        # 确保 X_df_raw 的列顺序和 imputer 以及模型训练时一致
        X_predict_imputed = imputer.transform(X_df_raw) # 使用之前 fit 好的 imputer

                                  predicted_values            = xgb_model.predict(X_predict_imputed)
            predicted_series  =  pd.Series(predicted_values, index=X_df_raw.index, name=variable_name)

        return predicted_series

    def process_all_weather_files_with_xgboost(): # 函数名体现 XGBoost
        """
        读取所有 9 个天气文件，对每个时间点和每个数值变量执行 XGBoost 回归，
        生成代表光伏电站位置（数据 1 的位置）的融合天气数据。
        """
        print(" 开始 XGBoost 天气融合")
```

```
        fused_data_sheets = {}

          all_data_by_sheet_and_id  =  {sheet_name:  {}  for
sheet_name in SHEET_NAMES}

                                target_file_path             =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{TARGET_POINT_ID}{FILE_PREFIX}")
        if not os.path.exists(target_file_path):
            raise FileNotFoundError(f"找不到目标点（数据 1）的天
气文件 '{target_file_path}'！ ")
        xls_target = pd.ExcelFile(target_file_path)

        for sheet_name in SHEET_NAMES:
              if  sheet_name  not  in  xls_target.sheet_names:
continue
               df_target_sheet  =  pd.read_excel(xls_target,
sheet_name=sheet_name)
            if 'time' not in df_target_sheet.columns: raise
KeyError(f"目标文件 1 '{sheet_name}' 找不到 'time' 列！ ")
                                df_target_sheet['time']      =
pd.to_datetime(df_target_sheet['time'])
           df_target_sheet.set_index('time', inplace=True)
            all_data_by_sheet_and_id[sheet_name][TARGET_POIN
T_ID] = df_target_sheet

        for ref_id in REFERENCE_POINT_IDS:
                                            file_path        =
os.path.join(BASE_WEATHER_DATA_FOLDER,
f"{ref_id}{FILE_PREFIX}")
           if not os.path.exists(file_path):
                   print(f"警告：找不到参考点 {ref_id} 的文件
'{file_path}'。")
               continue
           try:
               xls_ref = pd.ExcelFile(file_path)
               for sheet_name in SHEET_NAMES:
                    if sheet_name in xls_ref.sheet_names and
sheet_name in all_data_by_sheet_and_id:
                       df_ref_sheet = pd.read_excel(xls_ref,
sheet_name=sheet_name)
                       if 'time' not in df_ref_sheet.columns:
continue
```

```
                                    df_ref_sheet['time']  =
pd.to_datetime(df_ref_sheet['time'])
                                df_ref_sheet.set_index('time',
inplace=True)
                        all_data_by_sheet_and_id[sheet_name][
ref_id] = df_ref_sheet
            except Exception as e: print(f"读取参考点 {ref_id}
文件时出错: {e}")

        for sheet_name in SHEET_NAMES:
                            if    TARGET_POINT_ID    not    in
all_data_by_sheet_and_id.get(sheet_name, {}):
                  print(f"工作表 '{sheet_name}' 在数据文件 1 中缺
失，跳过。")
                continue

                                      df_target_template      =
all_data_by_sheet_and_id[sheet_name][TARGET_POINT_ID]
                                       df_fused_sheet       =
pd.DataFrame(index=df_target_template.index)
               print(f"\n---  正在处理工作表:  '{sheet_name}'
(XGBoost) ---")

                                current_sheet_dfs_ordered     =
[all_data_by_sheet_and_id[sheet_name].get(TARGET_POINT_ID)]
            # 确保即使某个参考文件缺失或某张表缺失，也用一个带正确索
引的空 DataFrame 或填充 NaN 的 DataFrame 占位
            # 以便后续 concat 能对齐所有时间戳
            base_index = df_target_template.index
            for ref_id in REFERENCE_POINT_IDS:
                                            df_to_add      =
all_data_by_sheet_and_id[sheet_name].get(ref_id)
                if df_to_add is None or df_to_add.empty:
                    # 创建一个与目标索引对齐的空 DataFrame，列名稍
后处理
                        print(f"提示：参考数据 {ref_id} 的工作表
'{sheet_name}' 为空或缺失，将用 NaN 填充。")
                         # 为了保持结构，可以先创建一个全 NaN 的
DataFrame，列名在 train_and_predict 中动态生成
                    current_sheet_dfs_ordered.append(pd.Data
Frame(index=base_index))
                else:
                    current_sheet_dfs_ordered.append(df_to_a
dd.reindex(base_index)) # 确保索引对齐
```

```
            numeric_cols_for_xgb = []
            for col in df_target_template.columns:
                 if col.lower() != COLUMN_TO_IGNORE.lower()
and pd.api.types.is_numeric_dtype(df_target_template[col]):
                    col_exists_in_any_ref = any(
                             ref_df is not None and col in
ref_df.columns for ref_df in current_sheet_dfs_ordered[1:]
                    )
                    if col_exists_in_any_ref:
                        numeric_cols_for_xgb.append(col)
                    else:
                        if col in df_target_template.columns:
df_fused_sheet[col] = df_target_template[col]

                print(f"    将对以下数值列进行 XGBoost 预测:
{numeric_cols_for_xgb}")

                              if      COLUMN_TO_IGNORE      in
df_target_template.columns:
                           df_fused_sheet[COLUMN_TO_IGNORE]   =
df_target_template[COLUMN_TO_IGNORE]

            for var_name in numeric_cols_for_xgb:
                                       predicted_series     =
train_and_predict_variable_with_xgboost(var_name,
current_sheet_dfs_ordered)
                if predicted_series is not None:
                                  df_fused_sheet[var_name]   =
predicted_series
                else:
                    if var_name in df_target_template.columns:
df_fused_sheet[var_name] = df_target_template[var_name]
                    else: df_fused_sheet[var_name] = np.nan

            # 确保所有原始模板列都存在于 fused_sheet 中，如果它们没
被处理
            for col in df_target_template.columns:
                if col not in df_fused_sheet.columns:
                                       df_fused_sheet[col]   =
df_target_template[col]

              fused_data_sheets[f"{sheet_name}_xgb_fused"] =
df_fused_sheet.copy()
```

```
            print(f"  工作表 '{sheet_name}' (XGBoost) 处理完成！")

        if fused_data_sheets:
             print(f"\n 正在将所有 XGBoost 融合后的天气数据保存到: {OUTPUT_XGB_EXCEL_PATH}")
             with pd.ExcelWriter(OUTPUT_XGB_EXCEL_PATH) as writer:
                          for sheet_name_out, df_out in fused_data_sheets.items():
                       df_out.reset_index().to_excel(writer, sheet_name=sheet_name_out, index=False)
            print("所有 XGBoost 融合的天气数据已成功保存！")
        else:
             print("没有任何工作表被成功处理和融合（XGBoost 方法），未生成输出文件。")

    if __name__ == "__main__":
        if not os.path.isdir(BASE_WEATHER_DATA_FOLDER):
                  print(f" 错 误 ！ 找 不 到 天 气 数 据 文 件 夹 : '{BASE_WEATHER_DATA_FOLDER}'")
        else:
            try:
                process_all_weather_files_with_xgboost()
            except Exception as e_main:
                print(f"主程序发生严重错误：{e_main}")
                traceback.print_exc()
```

误差分析的核心代码

```
import pandas as pd
import numpy as np
import traceback

# --- 文件路径和列名设定 ---
INPUT_EXCEL_PATH = 'results.xlsx'
INPUT_SHEET_NAME = 'all'
OUTPUT_EXCEL_PATH = 'error_all_results.xlsx'

ACTUAL_POWER_COL = 'Huang_E4102_kW'
CAPACITY_COL = 'P_install' # 作为公式中的 Ci
PREDICTION_COLUMNS = [
    'predicted_power_actual',          # 线性回归
```

```
    'predicted_power_actual_svr',      # SVR
    'predicted_power_actual_lstm_nwp', # LSTM
    'predicted_power_actual_tf_nwp',    # Transformer
    'predicted_power_actual_lstm_nwp_idw',
    'predicted_power_actual_lstm_nwp_LR',
    'predicted_power_actual_lstm_nwp_XG'
]
MODEL_NAMES = [ # 用于结果表中的列名
    'Linear Regression',
    'SVR',
    'LSTM (NWP)',
    'Transformer (NWP)',
    'LSTM(NWP,IDW)',
    'LSTM(NWP,LR)',
    'LSTM(NWP,XG)'
]

# 白昼判断阈值（例如，实际功率大于 0.01MW 认为是白昼/正在发电）
DAYLIGHT_THRESHOLD_MW = 0.01

# --- 误差计算函数 ---
def   filter_daylight_and_valid_capacity(df,   actual_col,
pred_col, cap_col):
    """筛选白昼时段且容量有效的数据点"""
    # 确保容量列不为 0 或 NaN，以避免除零错误
        valid_capacity_mask   =   (df[cap_col].notna())   &
(df[cap_col] > 1e-6) # 容量大于一个极小正数
    daylight_mask = df[actual_col] > DAYLIGHT_THRESHOLD_MW

    final_mask = daylight_mask & valid_capacity_mask

    # 还需要确保预测值不是 NaN
    valid_pred_mask = df[pred_col].notna()
    final_mask = final_mask & valid_pred_mask

    if not final_mask.any():
        print(f"警告：在为模型 '{pred_col}' 筛选白昼和有效容量数
据后，没有剩下任何数据点！")
                          return     pd.Series(dtype=float),
pd.Series(dtype=float), pd.Series(dtype=float), 0

    y_true = df.loc[final_mask, actual_col]
    y_pred = df.loc[final_mask, pred_col]
    capacity = df.loc[final_mask, cap_col]
```

```
    n_samples = len(y_true)

    return y_true, y_pred, capacity, n_samples

def calculate_e_rmse(y_true, y_pred, capacity, n_samples):
    if n_samples == 0: return np.nan
      # 确保 y_true, y_pred, capacity 索引对齐且长度一致 (filter_daylight 已处理)
    normalized_errors_sq = ((y_true - y_pred) / capacity)**2
     return np.sqrt(normalized_errors_sq.sum() / n_samples) # PDF 公式是先求和再除以 n

def calculate_e_mae(y_true, y_pred, capacity, n_samples):
    if n_samples == 0: return np.nan
    normalized_absolute_errors = np.abs((y_true - y_pred) / capacity)
    return normalized_absolute_errors.sum() / n_samples

def calculate_e_me(y_true, y_pred, capacity, n_samples):
    if n_samples == 0: return np.nan
    normalized_errors = (y_true - y_pred) / capacity
    return normalized_errors.sum() / n_samples

def calculate_r(y_true, y_pred, n_samples):
    if n_samples < 2: return np.nan # 相关系数至少需要两个点
    # Pandas Series 的 .corr() 方法可以直接计算皮尔逊相关系数
    return y_true.corr(y_pred)
     # 或者使用 NumPy: return np.corrcoef(y_true, y_pred)[0, 1]

def calculate_cr(e_rmse_normalized):
    if pd.isna(e_rmse_normalized): return np.nan
    return (1 - e_rmse_normalized) * 100.0

def calculate_qr(y_true, y_pred, capacity, n_samples, threshold=0.25):
    if n_samples == 0: return np.nan
    normalized_absolute_errors = np.abs((y_true - y_pred) / capacity)
    # B_i = np.where(normalized_absolute_errors < threshold, 1, 0)
    # return B_i.sum() / n_samples * 100.0
    # 使用 .sum() 而不是 np.mean() 再乘以 100，因为 B_i 已经是 0 或 1
```

```
    return (normalized_absolute_errors < threshold).sum() /
n_samples * 100.0

# --- 主程序逻辑 ---
def main_calculate_errors():
    """读取数据，计算所有模型的误差指标，并保存结果"""
    try:
         print(f" 正在读取结果文件: '{INPUT_EXCEL_PATH}', 工作
表: '{INPUT_SHEET_NAME}'...")
             df_results  =  pd.read_excel(INPUT_EXCEL_PATH,
sheet_name=INPUT_SHEET_NAME)
        print("数据成功载入！")

        # 检查必要的列是否存在
         required_cols = [ACTUAL_POWER_COL, CAPACITY_COL] +
PREDICTION_COLUMNS
        for col in required_cols:
            if col not in df_results.columns:
                 raise KeyError(f"数据中缺少必要的列: '{col}'!
请检查 Excel 文件。")

        metrics_summary = {} # 用字典存储结果，方便转 DataFrame

         for pred_col, model_name in zip(PREDICTION_COLUMNS,
MODEL_NAMES):
                print(f"\n---  正在为模型  '{model_name}'  (列:
'{pred_col}') 计算误差指标 ---")

              y_true_day, y_pred_day, capacity_day, n_day =
filter_daylight_and_valid_capacity(
                    df_results, ACTUAL_POWER_COL, pred_col,
CAPACITY_COL
            )

            if n_day == 0:
                 print(f"  模型 '{model_name}' 在筛选后没有有效
的白昼数据点，无法计算指标。")
                    e_rmse_val, e_mae_val, e_me_val, r_val,
cr_val, qr_val = (np.nan,) * 6
            else:
                 print(f"  筛选出 {n_day} 个有效的白昼数据点进行
计算。")
                  e_rmse_val = calculate_e_rmse(y_true_day,
y_pred_day, capacity_day, n_day)
```

```
                    e_mae_val = calculate_e_mae(y_true_day,
y_pred_day, capacity_day, n_day)
                     e_me_val = calculate_e_me(y_true_day,
y_pred_day, capacity_day, n_day)
                r_val = calculate_r(y_true_day, y_pred_day,
n_day)
                cr_val = calculate_cr(e_rmse_val) # 使用上面
算出的 e_rmse
               qr_val = calculate_qr(y_true_day, y_pred_day,
capacity_day, n_day)

            metrics_summary[model_name] = {
                '均方根误差 (E_rmse)': e_rmse_val,
                '平均绝对误差 (E_mae)': e_mae_val,
                '平均误差 (E_me)': e_me_val,
                '相关系数 (r)': r_val,
                '准确率 (C_R %)': cr_val,
                '合格率 (Q_R %)': qr_val
            }
            print(f"  模型 '{model_name}' 指标计算完成。")
               print(f"    E_rmse: {e_rmse_val:.4f}, E_mae:
{e_mae_val:.4f}, E_me: {e_me_val:.4f}")
             print(f"   r: {r_val:.4f}, C_R: {cr_val:.2f}%,
Q_R: {qr_val:.2f}%")

        # 将结果转换为 DataFrame 并保存
        df_metrics_summary = pd.DataFrame(metrics_summary)
        print("\n\n--- 所有模型误差指标汇总 ---")
        print(df_metrics_summary)

              df_metrics_summary.to_excel(OUTPUT_EXCEL_PATH,
index=True)
              print(f"\n  误 差 指 标 汇 总 已 成 功 保 存 到 :
'{OUTPUT_EXCEL_PATH}'!! ")

    except FileNotFoundError:
        print(f"找不到结果文件 '{INPUT_EXCEL_PATH}'，请检查文件
名和路径！")
    except KeyError as e:
         print(f"数据中好像缺少了名为 '{e}' 的列，请检查 Excel 文
件的列名！")
        traceback.print_exc()
    except Exception as e:
        print(f"在计算误差指标时发生了意料之外的错误：{e}")
```

```
        traceback.print_exc()

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