In [ ]:
```
import os
import pandas as pd
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
import re
import logging
from datetime import datetime, date
from pandas import Series, DataFrame
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold, KFold
from sklearn.metrics import mean_squared_error
import xgboost as xgb
from xgboost import DMatrix
import lightgbm as lgb
from lightgbm import Dataset
import matplotlib.pyplot as plt
from astral import LocationInfo
from astral.sun import sunrise, sunset, dawn, noon, dusk
from lightgbm import Dataset, train as lgb_train
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
from sklearn.preprocessing import StandardScaler
import zipfile



import warnings
warnings.filterwarnings("ignore")
```

In [ ]:
```
# 定义文件夹路径
base_dir = './dataset.part1'

# 定义场站文件夹名称
stations = [f'JSFD{i:03d}' for i in range(1, 15)] + [f'JSGF{i:03d}' for i in ran

# 存储每个场站的数据
station_data = {}

# 遍历所有场站文件夹
for station in stations:
    station_dir = os.path.join(base_dir, station)

    # 初始化数据框
    output_df, wind_df, operation_df = None, None, None

    # 遍历该文件夹下所有的xlsx文件
    for file_name in os.listdir(station_dir):
        # 跳过以 ~ 开头的临时文件
        if file_name.startswith('~$'):
            continue

        file_path = os.path.join(station_dir, file_name)

        # 读取场站出力.xlsx
        if '场站出力' in file_name and file_name.endswith('.xlsx'):
            all_sheets = pd.read_excel(file_path, sheet_name=None)  # 读取所有工
```



```
            # 遍历所有工作表
            for sheet_name, df in all_sheets.items():
                if df.shape[0] > 5 and df.shape[1] >= 2:  # 检查是否有足够的行和列
                    df = df.iloc[5:, [0, 1]]  # 从第5行开始取第一列和第二列
                    df.columns = ['时间', '出力功率']  # 重命名列
                    # 清理时间列，去掉任何可能的无效字符
                    df['时间'] = df['时间'].astype(str).str.strip()  # 去除空格
                    df['时间'] = pd.to_datetime(df['时间'], errors='coerce')  # 
                    if output_df is None:
                        output_df = df
                    else:
                        output_df = pd.concat([output_df, df])
                else:
                    print(f"文件 {file_path} 的工作表 {sheet_name} 数据不足，跳过处
            print(f"读取文件: {file_path}")

        # 读取测风数据.xlsx和测风塔数据.xlsx（使用文件中的列名）
        elif ('测风数据' in file_name or '测风塔数据' in file_name or '测光数据' i
            all_sheets = pd.read_excel(file_path, sheet_name=None)  # 读取所有工作
            # 遍历所有工作表
            for sheet_name, df in all_sheets.items():
                if df.shape[0] > 5:  # 检查是否有足够的行
                    df = df.iloc[5:, :]  # 从第5行开始取数据
                    # 清理时间列，去掉任何可能的无效字符
                    df['时间'] = df['时间'].astype(str).str.strip()  # 去除空格
                    df['时间'] = pd.to_datetime(df['时间'], errors='coerce')  # 
                    if wind_df is None:
                        wind_df = df
                    else:
                        wind_df = pd.concat([wind_df, df])
                else:
                    print(f"文件 {file_path} 的工作表 {sheet_name} 数据不足，跳过处
            print(f"读取文件: {file_path}")

        # 读取运行记录.xlsx（如果存在）且排除JSGF006
        elif '运行记录' in file_name and file_name.endswith('.xlsx') and station 
            all_sheets = pd.read_excel(file_path, sheet_name=None)  # 读取所有工作
            # 遍历所有工作表
            for sheet_name, df in all_sheets.items():
                if df.shape[0] > 2 and df.shape[1] >= 3:  # 检查是否有足够的行和列
                    df = df.iloc[2:, 0:3]  # 从第三行开始取前三列
                    df.columns = ['起始时间', '终止时间', '最大出力功率']  # 自定义
                    df['起始时间'] = pd.to_datetime(df['起始时间'], errors='coerc
                    df['终止时间'] = pd.to_datetime(df['终止时间'], errors='coerc
                    if operation_df is None:
                        operation_df = df
                    else:
                        operation_df = pd.concat([operation_df, df])
                else:
                    print(f"文件 {file_path} 的工作表 {sheet_name} 数据不足，跳过处
            print(f"读取文件: {file_path}")

    # 如果有场站出力和测风数据，合并它们
    if output_df is not None and wind_df is not None:
        merged_df = pd.merge(output_df, wind_df, on='时间', how='left')

        # 如果存在运行记录数据，将最大出力功率加入到合并数据中
        if operation_df is not None:
            # 遍历运行记录中的每一行，匹配时间范围并插入最大出力功率
            for _, row in operation_df.iterrows():
```

```
                start_time, end_time, max_power = row['起始时间'], row['终止时间'
                # 将最大出力功率应用到时间范围内的所有行
                mask = (merged_df['时间'] >= start_time) & (merged_df['时间'] <=
                merged_df.loc[mask, '最大出力功率'] = max_power
        print(merged_df)
        # 存储合并后的数据到 station_data 字典
        station_data[station] = merged_df
    else:
        print(f"场站 {station} 数据不完整，无法合并")

print("数据处理完成")
```

In [ ]:
```
for station, df in station_data.items():
    print(f"站点: {station}, 数据行数: {len(df)}")
    print(df.head())  # 显示前几行数据
    print(df.isnull().sum())  # 显示每列的空值数量
```

In [ ]:
```
# 新字典用于保存处理后的数据
processed_station_data = {}

# 定义所需的特征列
required_columns = ['时间', '出力功率', '10米高度处风速（m/s）', '10米高度处风向（°）
                    '30米高度处风速（m/s）', '30米高度处风向（°）', '50米高度处风速
                    '50米高度处风向（°）', '70米高度处风速（m/s）', '70米高度处风向
                    '风机轮毂高度处风速（m/s）', '风机轮毂高度处风向（°）', '气温（°
                    '气压（hpa）', '相对湿度（%）', '最大出力功率']

# 读取每个站点的数据，检查列格式并进行转换
for station, df in station_data.items():
    if 'JSGF' in station:  # 光电数据
        processed_df = df.copy()  # 直接保留所有列

        # 检查是否存在 '最大出力功率' 列，若不存在则添加
        if '最大出力功率' not in processed_df.columns:
            processed_df['最大出力功率'] = 999999  # 全部值设为99999

    else:  # 风电数据
        # 只保留存在于 DataFrame 中的特征列
        existing_columns = [col for col in required_columns if col in df.columns
        processed_df = df[existing_columns].copy()

        # 检查是否存在 '最大出力功率' 列，若不存在则添加
        if '最大出力功率' not in processed_df.columns:
            processed_df['最大出力功率'] = 99999  # 全部值设为99999

    # 进行数据类型转换，允许最大出力功率和时间为空
    for column in processed_df.columns:
        if column != '最大出力功率' and column != '时间':
            try:
                processed_df[column] = processed_df[column].astype(float)
            except ValueError:
                print(f"无法将 {column} 转换为浮点数，数据类型可能为字符串")

    # 检查空值情况，除了最大出力功率允许为空
    null_counts = processed_df.isnull().sum()
    for column in null_counts.index:
        if column != '最大出力功率':
            print(f"{station} - {column} 空值数量: {null_counts[column]}")
```



```
    # 将处理后的数据保存在新字典中
    processed_station_data[station] = processed_df

print("处理后的数据:", processed_station_data)
```

In [ ]:
```
# 遍历每个站点的数据并处理
for station, df in processed_station_data.items():
    # 将 -99 \'--'替换为 NaN
    df.replace(-99, np.nan, inplace=True)
    df.replace('--', np.nan, inplace=True)
    df.replace('<NULL>', np.nan, inplace=True)
    # 进行线性插值，但不处理 '时间' 和 '最大出力功率' 列
    df.interpolate(method='linear', inplace=True, limit_direction='both')

    # 检查是否仍有NaN值，使用前向和后向填充处理剩余的空值
    df.fillna(method='ffill', inplace=True)  # 前向填充
    df.fillna(method='bfill', inplace=True)  # 后向填充

    # 将 '最大出力功率' 列的 NaN 值替换为 9999
    df['最大出力功率'].fillna(999999, inplace=True)

    # 规则一：删除全为 NaN 的列
    df.dropna(axis=1, how='all', inplace=True)

    # 规则二：删除连续超过20行、4列以上数据全为空值的行
    # 逐行检查超过4列为空值的情况
    count_nan_cols = df.isna().sum(axis=1)  # 统计每行 NaN 的列数
    mask = count_nan_cols >= 4  # 找出4列或以上为空值的行

    # 使用滚动窗口检测连续20行满足条件的情况
    rolling_window = mask.rolling(window=20, min_periods=20).sum() == 20
    df = df[~rolling_window]

    # 将处理后的数据替换回字典中
    processed_station_data[station] = df

# 检查处理后的结果
for station, df in processed_station_data.items():
    print(f"站点 {station} 的处理结果：")
    print(df.isna().sum())  # 查看每列中是否仍有 NaN 值
```

In [ ]:
```
# 检查 processed_station_data 的列名、数据类型、分布情况和空值情况
for station, df in processed_station_data.items():
    print(f"站点 {station} 的数据概况：")

    # 输出列名和数据类型
    print("列名和数据类型：")
    print(df.dtypes)

    # 输出数据的描述性统计信息
    print("\n数据分布情况：")
    print(df.describe(include='all'))  # 包括所有类型

    # 输出空值情况
    print("\n空值情况：")
    print(df.isna().sum())

    # 输出数据量情况
```

```
    print("\n数据量情况: ")
    print(len(df))

    print("\n" + "="*40 + "\n")  # 分隔线
```

In [ ]:
```
processed_station_data['JSGF006']
```

In [360…
```
# 读取每个站点的数据，添加时间特征
for station, df in processed_station_data.items():
    # 确保时间列为 datetime 类型
    df['时间'] = pd.to_datetime(df['时间'], errors='coerce')

    # 提取时间特征
    df['年'] = df['时间'].dt.year
    df['月'] = df['时间'].dt.month
    df['日'] = df['时间'].dt.day
    df['时'] = df['时间'].dt.hour
    df['分'] = df['时间'].dt.minute
    df['秒'] = df['时间'].dt.second

    # 将处理后的数据保存在新字典中
    processed_station_data[station] = df
```

In [329…
```
##历史值特征
# dfs = []
# for site, df_site in df.groupby("光伏用户编号"):
#     df_site = df_site.sort_values("时间")
#     df_site["辐照强度 (J/m2)  - 1"] = df_site["辐照强度 (J/m2) "].shift(1) - df_
#     df_site["辐照强度 (J/m2)  - 8"] = df_site["辐照强度 (J/m2) "].shift(8) - df_
# #     df_site["辐照强度 (J/m2)  - 2"] = df_site["辐照强度 (J/m2) "].shift(2) - d
#     dfs.append(df_site)
# df = pd.concat(dfs, axis=0)
```

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei']  # 黑体字体
plt.rcParams['axes.unicode_minus'] = False  # 使负号正常显示

# 选择需要绘图的站点
stations_to_plot = ['JSFD013', 'JSGF005']

# 遍历需要绘图的站点
for station_name in stations_to_plot:
    df = processed_station_data[station_name]

    # 打印当前站点的列名，方便调试
    print(f"{station_name} 的列名: {df.columns.tolist()}")

    # 确保“出力功率”列存在
    if '出力功率' in df.columns:
        numeric_columns = df.select_dtypes(include=[np.number]).columns.differen

        # 创建散点图
        cols = 3  # 每行的列数
        rows = (len(numeric_columns) + cols - 1) // cols  # 计算行数

        plt.figure(figsize=(15, 5 * rows))
```



```
        # 遍历每个数值列，绘制散点图
        for index, column in enumerate(numeric_columns):
            filtered_df = df[df[column] != -100]  # 排除 x = -100 的数据

            # 检查过滤后的数据是否为空
            if filtered_df.empty:
                print(f"站点 {station_name} 中列 {column} 的有效数据为空，无法绘制散
                continue

            # 检查当前列是否存在
            if column not in df.columns:
                print(f"站点 {station_name} 中没有列 {column}，跳过此列。")
                continue

            # 绘制散点图
            plt.subplot(rows, cols, index + 1)
            plt.scatter(filtered_df[column], filtered_df['出力功率'], alpha=0.5,

            # 拟合趋势线
            z = np.polyfit(filtered_df[column], filtered_df['出力功率'], 1)  # 线
            p = np.poly1d(z)
            plt.plot(filtered_df[column], p(filtered_df[column]), color='red', l

            plt.xlim(filtered_df[column].min(), filtered_df[column].max())
            plt.title(f"{station_name} - 出力功率与{column}的散点图")
            plt.xlabel(column)
            plt.ylabel('出力功率')
            plt.legend()
            plt.grid()

        # 调整布局并显示
        plt.tight_layout()
        plt.show()
    else:
        print(f"{station_name} 没有 '出力功率' 列")
```

In [331…
```
def score(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    return 1 / (1 + rmse)
```

In [ ]:
```
processed_station_data['JSFD005']
```

In [ ]:
```
# 设置文件夹路径和站点名称
folder_path = './dataset.part1/气象预测数据'  # 修改为你的文件夹路径

# 假设这些是你需要处理的站点
target_stations = [f'JSFD{i:03d}' for i in range(1, 15)] + [f'JSGF{i:03d}' for i

# 解压缩文件夹中的所有 ZIP 文件，解压到对应站点的文件夹下
for station in target_stations:
    zip_file_path = os.path.join(folder_path, f'cepri_historic_2019010112_202012
    station_folder_path = os.path.join(folder_path, station)  # 每个站点的文件夹
    os.makedirs(station_folder_path, exist_ok=True)  # 创建站点对应的文件夹
    print(f"解压缩文件: {zip_file_path} 到 {station_folder_path}")
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        zip_ref.extractall(station_folder_path)  # 解压缩到站点对应的文件夹
```

In [ ]:
```
# 存储处理后的数据
processed_weather_data = {}

# 遍历每个站点对应的文件夹中的 CSV 文件
for station in target_stations:
    station_folder_path = os.path.join(folder_path, station)
    station_data = []  # 用于存储该站点的所有CSV数据
    print(station_folder_path)
    for filename in os.listdir(station_folder_path):

        if filename.endswith('.csv'):


            csv_file_path = os.path.join(station_folder_path, filename)

            # 读取 CSV 文件
            df = pd.read_csv(csv_file_path, sep='\s+', header=0,
                             names=['sitename', 'date1', 'time1', 'date2', 'time
                                    'ws30', 'ws31', 'ws32', 'ws10', 'direction30
                                    'mslp', 'clc', 'senf', 'latf', 'swr', 'lwr',
                                    'T2m', 'RH2m'],
                             engine='python')

            # 提取前 96 行数据
            if df.shape[0] >= 96:
                df = df.iloc[:96].copy()

                # 创建新的数据时间列
                # 确保 date2 和 time2 列为字符串
                df['date2'] = df['date2'].astype(str)
                df['time2'] = df['time2'].astype(str)

                # 创建新的数据时间列，并将其格式化为 "YYYY-MM-DD HH:MM:SS"
                df['新数据时间'] = pd.to_datetime(df['date2'] + ' ' + df['time2']

                # 转换为所需的时间格式
                df['新数据时间'] = df['新数据时间'].dt.strftime('%Y-%m-%d %H:%M:%S



                # 提取需要的列
                relevant_columns = ['sitename', '新数据时间', 'T', 'momf', 'ws30'
                                    'direction30', 'direction31', 'dir10', 'mslp
                df = df[relevant_columns]

                # 将处理后的数据添加到站点列表
                station_data.append(df)

    if station_data:  # 如果该站点有处理好的数据
        processed_weather_data[station] = pd.concat(station_data, ignore_index=T

# 打印处理结果
for station, data in processed_weather_data.items():
    print(f"站点 {station} 的处理结果：")
    print(data.head())  # 打印前几行
```

In [ ]:
```
import pandas as pd

# 假设 processed_station_data 和 processed_weather_data 字典已存在
```



```
# processed_station_data = {station: df}  # 每个站点的数据
# processed_weather_data = {station: df}  # 每个站点的天气数据
processed_train_data = {}
# 遍历每个站点，将天气数据合并到风电站点数据中
for station in processed_station_data.keys():
    if station in processed_weather_data:
        # 提取对应的 DataFrame
        station_data = processed_station_data[station]
        weather_data = processed_weather_data[station]

        # 确保有时间列，转换为日期时间格式
        station_data['时间'] = pd.to_datetime(station_data['时间'])  # 如果时间列
        weather_data['新数据时间'] = pd.to_datetime(weather_data['新数据时间'])

        # 根据时间合并数据，使用左连接，保留风电站点数据
        merged_data = pd.merge(station_data, weather_data, left_on='时间', right_

        # 更新 processed_station_data 中的相应站点数据
        processed_train_data[station] = merged_data

# 查看更新后的 processed_station_data
for station, data in processed_train_data.items():
    print(f"{station} 更新后的数据：")
    print(data)  # 打印前几行以便检查合并效果
```

In [ ]:
```
# 遍历每个站点的数据并处理
for station, df in processed_train_data.items():
    # 将 -99 \'--'替换为 NaN
    df.replace(-99, np.nan, inplace=True)
    df.replace('--', np.nan, inplace=True)
    df.replace('<NULL>', np.nan, inplace=True)
    # 进行线性插值，但不处理 '时间' 和 '最大出力功率' 列
    df.interpolate(method='linear', inplace=True, limit_direction='both')

    # 检查是否仍有NaN值，使用前向和后向填充处理剩余的空值
    df.fillna(method='ffill', inplace=True)  # 前向填充
    df.fillna(method='bfill', inplace=True)  # 后向填充

    # 将 '最大出力功率' 列的 NaN 值替换为 9999
    df['最大出力功率'].fillna(999999, inplace=True)

    # 规则一：删除全为 NaN 的列
    df.dropna(axis=1, how='all', inplace=True)

    # 规则二：删除连续超过20行、4列以上数据全为空值的行
    # 逐行检查超过4列为空值的情况
    count_nan_cols = df.isna().sum(axis=1)  # 统计每行 NaN 的列数
    mask = count_nan_cols >= 4  # 找出4列或以上为空值的行

    # 使用滚动窗口检测连续20行满足条件的情况
    rolling_window = mask.rolling(window=12, min_periods=12).sum() == 12
    df = df[~rolling_window]

    # 将处理后的数据替换回字典中
    processed_train_data[station] = df

# 检查处理后的结果
for station, df in processed_train_data.items():
```

```
    print(f"站点 {station} 的处理结果: ")
    print(df.isna().sum())  # 查看每列中是否仍有 NaN 值
```

In [ ]:
```
def create_lag_features(processed_train_data, target_column, lag=4):
    processed_train_data_with_lags = {}

    for station, df in processed_train_data.items():
        print(f"Processing station: {station}")

        # 确保时间列为 datetime 类型
        df['时间'] = pd.to_datetime(df['时间'])

        # 筛选出数值型特征列，排除目标列
        numeric_columns = df.select_dtypes(include=[np.number]).columns.tolist()
        numeric_columns.remove(target_column)  # 确保不对目标列创建滞后特征

        # 创建历史特征
        for column in numeric_columns:
            df[f'{column}_lag{lag}'] = df[column].shift(lag)

        # 删除产生的空值
        df.dropna(inplace=True)

        # 保存处理后的 DataFrame
        processed_train_data_with_lags[station] = df

    return processed_train_data_with_lags

# 假设 processed_train_data 是您之前定义的字典格式的数据
# 设置目标列名
target_column = "出力功率"  # 根据需要替换为目标列名

# 调用函数
processed_train_data = create_lag_features(processed_train_data, target_column,
```

In [ ]:
```
processed_train_data['JSFD005'].columns
```

In [ ]:
```
# 目标列
target_column = "出力功率"  # 根据需要替换为目标列名

# 准备模型参数
params_lgb = {
    "num_boost_round": 10000,
    'learning_rate': 0.0125,
    'boosting_type': 'gbdt',
    'objective': 'mse',
    'metric': 'rmse',
    'num_leaves': 100,
    'seed': 42,
    'n_jobs': -1,
    'feature_fraction': 0.9,
    'bagging_fraction': 0.8,
    'bagging_freq': 4,
    "early_stopping_round": 50
}

model_lgb = []

# 确保输出目录存在
```



```
output_dir = "./models/"
os.makedirs(output_dir, exist_ok=True)

# 指定要训练的站点范围
target_stations = [f'JSFD{i:03d}' for i in range(1, 15)] + [f'JSGF{i:03d}' for i

# 定义创建滞后特征的函数
def create_lag_features(df, numeric_columns, lag=4):
    # 确保时间列为 datetime 类型
    df['时间'] = pd.to_datetime(df['时间'])

    # 创建历史特征
    for column in numeric_columns:
        df[f'{column}_lag{lag}'] = df[column].shift(lag)

    # 删除产生的空值
    df.dropna(inplace=True)

    return df

# 遍历每个站点的数据
for station, df in processed_train_data.items():
    if station not in target_stations:  # 过滤掉不在目标站点范围内的站点
        continue

    # 计算最后一个月的开始日期
    last_month_start = df['时间'].max() - pd.DateOffset(months=1)

    train_data = df[df['时间'] < last_month_start]
    val_data = df[df['时间'] >= last_month_start]

    # 确保有目标列
    if target_column not in train_data.columns or target_column not in val_data.
        print(f"{station} 缺少目标列 {target_column}，跳过该站点")
        continue




    # 删除包含 '_天气' 的特征列
    train_data = train_data.loc[:, ~train_data.columns.str.contains('_天气')]
    val_data = val_data.loc[:, ~val_data.columns.str.contains('_天气')]
    # 筛选数值型特征
    numeric_columns = train_data.select_dtypes(include=[np.number]).columns.toli
    if target_column in numeric_columns:
        numeric_columns.remove(target_column)  # 移除目标列

    # 确保目标列在数值型特征中
    if target_column not in train_data.columns or target_column not in val_data.
        print(f"{station} 的目标列 {target_column} 不是数值型，跳过该站点")
        continue
    print(numeric_columns)
    # 准备数据
    x_train = train_data[numeric_columns].fillna(0).astype(np.float32)
    y_train = train_data[target_column].astype(np.float32)

    x_val = val_data[numeric_columns].fillna(0).astype(np.float32)
    y_val = val_data[target_column].astype(np.float32)

    # 1折交叉验证
```

```
    kfold = KFold(n_splits=2, random_state=42, shuffle=True)
    mse = 0
    mape_total = 0

    for fold, (train_index, val_index) in enumerate(kfold.split(x_train, y_train
        logging.info(f'############ {station} - fold: {fold} ############')

        x_fold_train, x_fold_val = x_train.iloc[train_index], x_train.iloc[val_i
        y_fold_train, y_fold_val = y_train.iloc[train_index], y_train.iloc[val_i

        # LightGBM训练
        trainset = lgb.Dataset(x_fold_train, y_fold_train)
        valset = lgb.Dataset(x_fold_val, y_fold_val)
        model = lgb.train(params_lgb, trainset, valid_sets=[trainset, valset],
                          callbacks=[lgb.log_evaluation(1000)])

        model.save_model(os.path.join(output_dir, f"lgb_{station}_{fold}.txt"))
        model_lgb.append(model)

        # 计算特征重要性
        feature_importance = model.feature_importance()
        importance_df = pd.DataFrame({
            'feature': x_fold_train.columns,
            'importance': feature_importance
        }).sort_values(by='importance', ascending=False)

        # 筛选前10个重要特征
        top_features = importance_df['feature'].head(10).tolist()

        # 使用重要特征重新训练专属模型
        x_train_top = x_train[top_features]
        x_val_top = x_val[top_features]

        trainset_top = lgb.Dataset(x_train_top, y_train)
        valset_top = lgb.Dataset(x_val_top, y_val)

        # 训练专属模型
        exclusive_model = lgb.train(params_lgb, trainset_top, valid_sets=[trains
                                    callbacks=[lgb.log_evaluation(1000)])
        exclusive_model.save_model(os.path.join(output_dir, f"exclusive_lgb_{sta

        # 计算预测值和评估指标
        predictions = exclusive_model.predict(x_val_top)
        mse += mean_squared_error(y_val, predictions)
        mape_total += mean_absolute_percentage_error(y_val, predictions)

    rmse = np.sqrt(mse / kfold.n_splits)
    mape_avg = (mape_total / kfold.n_splits) * 100  # 计算平均MAPE

    # 打印 RMSE 和 MAPE
    logging.info(f"--------------{station} RMSE: {rmse}, MAPE: {mape_avg}% -----
    print(f"--------------{station} RMSE: {rmse}, MAPE: {mape_avg}% ------------

    # 保存实际值和预测值
    output_file_path = os.path.join(output_dir, f"{station}_predictions.txt")
    with open(output_file_path, 'w', encoding='utf-8') as f:
        f.write(f"{station} y_val:\n{y_val}\n")
        f.write(f"{station} model.predict(x_val):\n{predictions}\n")

    # 绘制实际值和预测值的对比图
```



```
    plt.figure(figsize=(14, 7))
    plt.plot(df['时间'][df['时间'] >= last_month_start], y_val, label='实际值 (y_
    plt.plot(df['时间'][df['时间'] >= last_month_start], predictions, label='预测
    plt.title(f'{station} - 实际值与预测值对比')
    plt.xlabel('时间')
    plt.ylabel('出力功率')
    plt.legend()
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, f"{station}_predictions_plot.png"))
    plt.close()
```

In [ ]: