**成绩**



**数据仓库与数据挖掘**

**大作业报告**

**题　　目： Weather Dataset数据挖掘**

**专 业： 软件工程**

**姓 名： 俞乐楠 徐文彬 傅裕翔**

**学 号： 1120221303 1120221397**

**1120221267**

目录

1. [前言 2](#_Toc183684753)

[1.1 数据集概括性介绍 2](#_Toc183684754)

1. [数据可视化探索分析 3](#_Toc183684755)

[2.1 箱线图 3](#_Toc183684756)

[2.2 数据集可视化 3](#_Toc183684757)

[2.2.1 单个属性的可视化 3](#_Toc183684758)

[2.2.2 热力图 5](#_Toc183684759)

1. [数据清洗 7](#_Toc183684760)

[3.1 缺失值处理 7](#_Toc183684761)

[3.2 去重 7](#_Toc183684762)

[3.3 噪声识别和处理 8](#_Toc183684763)

[**3.4 特征构建** 9](#_Toc183684764)

1. [数据挖掘 10](#_Toc183684765)

[4.1 聚类 10](#_Toc183684766)

[4.2 分类 12](#_Toc183684767)

[4.3 特征重要性分析 24](#_Toc183684768)

[4.4 关联规则挖掘 25](#_Toc183684769)

1. [异常检测 27](#_Toc183684770)

[5.1 DBSCAN 27](#_Toc183684771)

[**5.2 LOF** 28](#_Toc183684772)

[**5.3 随机森林** 29](#_Toc183684773)

1. [附录 30](#_Toc183684774)

[6.1 未来挑战 30](#_Toc183684775)

[6.2 人员分工 30](#_Toc183684776)

[6.3 代码 30](#_Toc183684777)

摘 要:

在经济发展发展的今日，人们对天气预报的精确度要求越来越高，本小组以 Weather Dataset欧洲十年天气数据集为研究对象，使用了箱线图绘制、直方图，属性相关性分析、热力图等方法对数据集进行了探索分析并可视化探索结果，对数据集进行了去重、缺失值处理、异常值检测，进行了聚类、分类、关联规则挖掘等数据挖掘任务，并使用DBSCAN、LOF、随机森林算法对其进行了异常检测。较好地实现了对天气预报的预测效果。

关键词: 可视化；异常检测；聚类；分类，预测性任务

# 前言

## 1.1 数据集概括性介绍

Weather Dataset数据集网址： <https://www.kaggle.com/datasets/muthuj7/weather-dataset/data>

Weather Dataset数据集是记录了欧洲某一地区从2006年开始的，长达十年的每小时天气情况记录，该数据集大小：共96453条数据记录，11个维度。

属性名称：

- Formatted Date：时间戳

- Summary：天气概述

- Precip Type：降水类型

- Temperature (C)：温度（摄氏度）

- Apparent Temperature (C)：体感温度（摄氏度）

- Humidity：湿度

- Wind Speed (km/h)：风速（公里/小时）

- Wind Bearing (degrees)：风向（度）

- Visibility (km)：能见度（公里）

- Pressure (millibars)：气压（毫巴）

- Daily Summary：每日总结数据类型：均为浮点数类型。

数据质量：

数据集真实可靠；

按时间顺序排列，数据集是完整的，经处理后无缺省值，不存在重复数据记录；

所有数据在具体进行数据挖掘操作时均已经过正则化处理。

图1 Weather Dataset数据集示例（此为暂未正则化的原始数据）

# 数据可视化探索分析

## 2.1 箱线图

而在本项目的数据处理过程中，我们使用分箱这种将连续变量划分为离散类别的方法。分箱有助于将连续数据转换为分类数据，以便于进一步分析和处理。在天气数据处理中，分箱可以帮助我们将时间、风速、风向等连续变量转换为有意义的分类，以便于后续的分析和建模

对每个数值类型属性绘制箱线图如下：

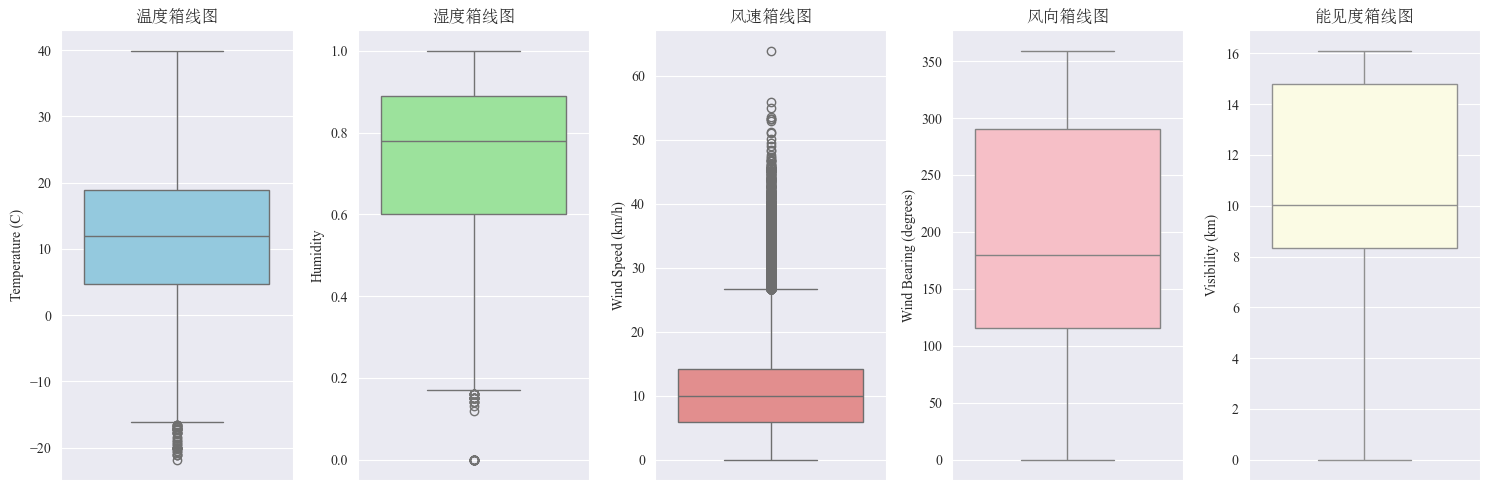


图2 温度，湿度，风速，风向，能见度属性箱线图

通过阅读分析箱线图，可以对五个属性做如下分析：

温度 (Temperature)：数据略显分散，具有一定的对称性，接近正态分布，而异常值分布均衡，既有低温也有高温的异常值。

湿度 (Humidity)：下聚集十分明显，不具有对称性，中位数相较其他属性较大，异常值集中在湿度较低的一侧。

风速 (WindSpeed)：数据分散，范围较大，不具对称性，中位数较低，异常值集中在数据值较大的一侧，说明存在一些极端的高风速值。

风向角度 (Wind Bearing)：数据略显分散，具有一定的对称性，中位数适中，接近正态分布，异常值很少且分布均匀。

能见度 (Visibility)：数据略显分散，不具有对称性，呈现一定的下聚集趋势，中位数相较其他属性较大，异常值主要分布在数据值较小的一侧。

## 2.2 数据集可视化

### 2.2.1 单个属性的可视化

使用Seaborn 库中的sns.barplot()函数绘制每天的全天整体天气状况的数量分布柱状图，根据这张图，我们可以看出在西欧地区天气以Partly Cloudly,Mostly Cloudly,Overcast,Foggy,Clear这几类天气状况居多

图表

低可信度描述已自动生成

图3 全天整体天气状况的数量分布柱状图

而使用Matplotlib 库中的plt.pie()函数绘制每天的天气类型分布饼状图，可以直观的看出欧洲在不同的时间段的天气状况，同样的，可以看出Cloudy，Foggy，Overcast相关的天气类型依然是欧洲各时间阶段中天气的主流。

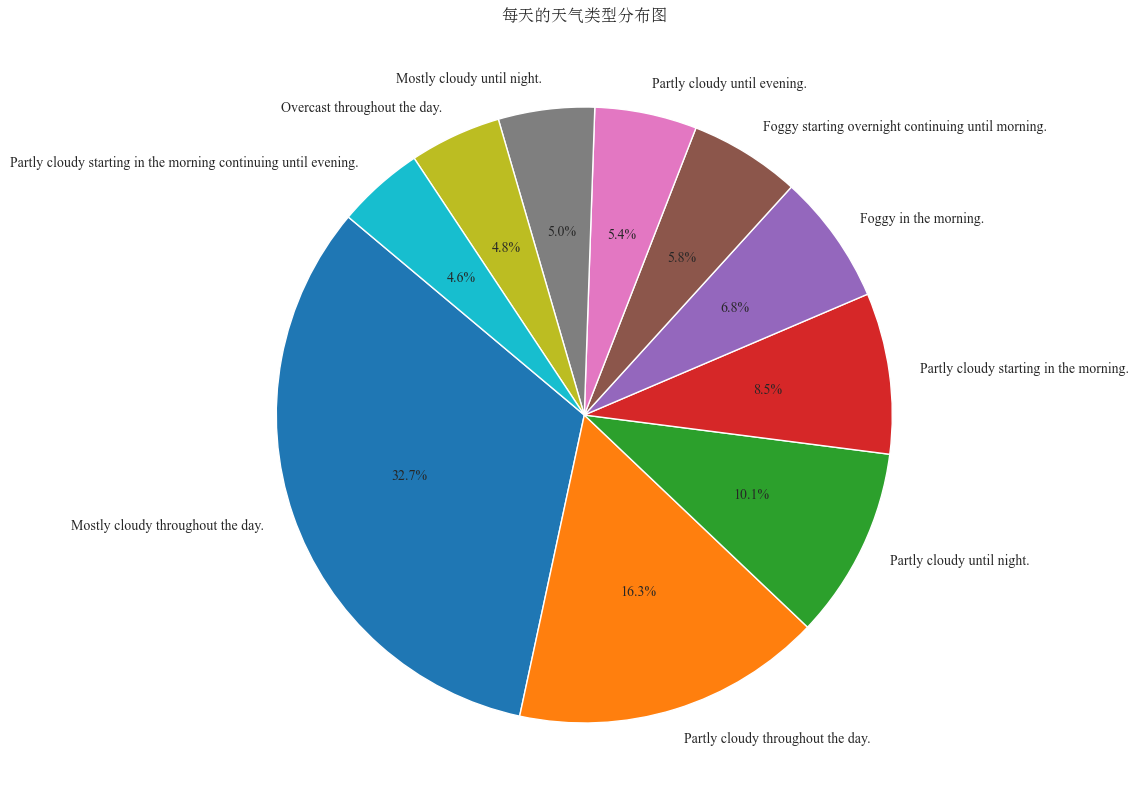


图4 每天的天气类型分布饼状图

使用Seaborn 库中sns.histplot()绘制温度直方图，温度、湿度的核密度图，形象地描述了温度和湿度数据的集中趋势、离散程度和分布形态。

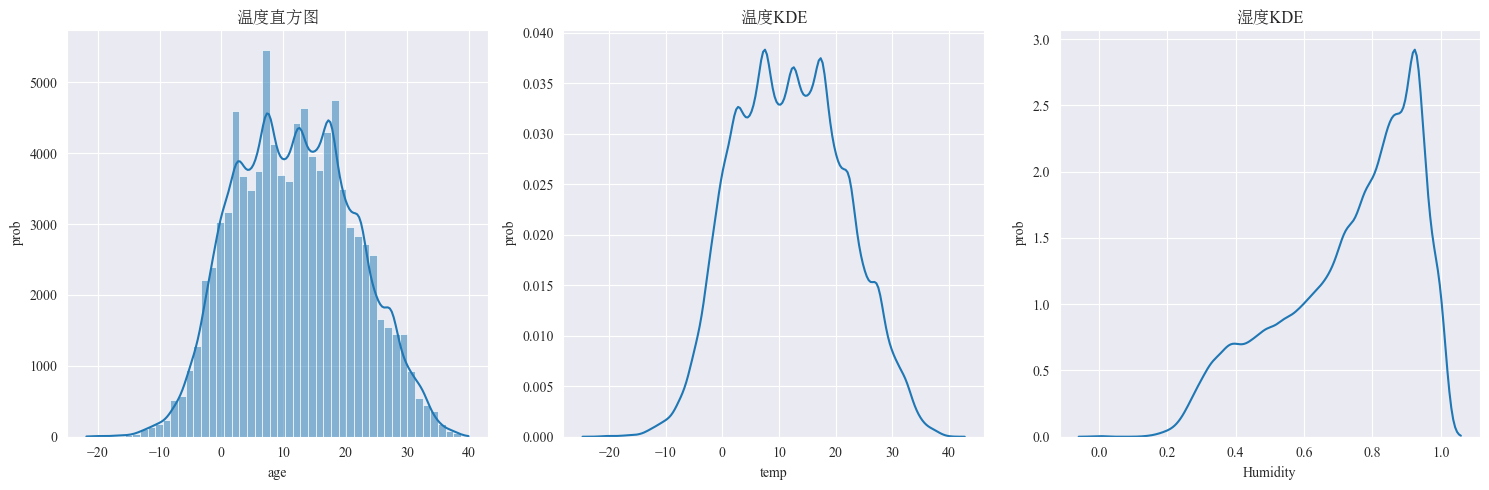
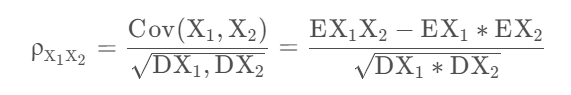


图5 温度直方图，温度、湿度的核密度图

### 2.2.2 热力图

热力图可以表示各属性的相关性关系，其中不同方块颜色对应的相关系数的大小，该值越大，色块颜色越深，两属性的线性相关性越高，两个属性变量之间相关系数（皮尔森相关系数）的计算公式如下：



使用基于matplotlib的Python可视化库Seaborn中的heatmap()函数进行图像的绘 制，所得图像如下：

图形用户界面, 应用程序, Teams

描述已自动生成

图6 热力图

我们分析这张热力图可知：Temperature (C)与Apparent Temperature (C)这两个属性之间的线性相关性极高，相关系数为0.99。两个变量之间有很强的线性关系；Temperature (C)与Humidity这两个属性之间呈现出负相关关系，相关系数为-0.63。这表明温度越高，湿度通常越低；Humidity与Apparent Temperature (C)之间也有显著的负相关关系，相关系数为-0.60。这意味着在感知温度高时，湿度通常较低； Wind Speed (km/h)与其他大多数属性的相关性都较低，最高相关系数为0.10（与Wind Bearing (degrees)），这表明风速相对独立于其他变量；Visibility (km)与Temperature (C)和Apparent Temperature (C)这两个属性与能见度呈现中度正相关，相关系数分别为0.39和0.38，在温度高时，能见度通常较高；Pressure (millibars)与其他所有属性之间的相关性都非常低，相关系数在-0.01到0.05之间。所以压力基本上独立于其他变量。

由此我们可以得出这些结论**：**Humidity与Temperature (C)和Apparent Temperature (C)均呈负相关，其变化对温度的影响显著；Wind Speed (km/h)和Pressure (millibars)二者与其他变量的相关性较低，可以在需要独立变量时首先考虑。Visibility (km)在选择涉及温度的变量时，可以考虑其正相关性。

# 数据清洗

## 3.1 缺失值处理

我们使用 Python 的 pandas 库来检测 CSV 文件中的缺失值，读取 weatherHistory.csv 文件并输出每一列中的缺失值数量。可以根据输出结果查看数据集中是否存在缺失值以及它们分布在哪些列。

缺失值检测结果：

Formatted Date 0

Summary 0

Precip Type 517

Temperature (C) 0

Apparent Temperature (C) 0

Humidity 0

Wind Speed (km/h) 0

Wind Bearing (degrees) 0

Visibility (km) 0

Loud Cover 0

Pressure (millibars) 0

Daily Summary 0

dtype: int64

可以看到Precip Type有517个数据为null，经检查都是rain，使用 pandas 库中的 fillna() 函数将 Precip Type 列中的缺失值替换为 'rain'



图7 缺失值填补程序运行结果

## 3.2 去重

因为现实中Formatted Date（日期时间）不可能重复。需要检查其是否存在重复。

表格

描述已自动生成 运行程序得到以下48行重复的时间记录：

图8 重复的时间记录

可得出Formatted Date（日期时间）确实是有重复数据，但是在保留了第一次出现的数据之后，还需要把数据按照时间顺序排序。我们先去除 Formatted Date 列中的重复数据（保留首次出现的记录），然后根据 Formatted Date 进行排序。确保原先是时间顺序不变的前提下清除了重复的记录。

值得注意的是，在欧洲一些地区，冬季和夏季时间会有所不同，例如：一些使用+01:00的国家在夏季时可能会调整到+02:00（例如，欧洲的夏时制）。这种情况下，在夏季某些地区的时间可能会被认为相同（例如，冬季的+01:00变为+02:00)，但在标准时间上它们是不同的。

例如，欧洲中部时间（Central European Time, CET）是一个典型的例子。我们将把时间数据统一转换为标准时区 UTC+01:00，即欧洲中部标准时间（CET），不再考虑夏令时的变动。这样做有助于确保数据分析时的时区一致性。

## 3.3 噪声识别和处理

基于箱线图和统计特征,原先数据集中Loud Cover(云覆盖度)次列数据全部为0，推测是数据存在残缺或者问题，将此列删除。

使用 pandas 库来删除 Loud Cover 列。

使用 df.drop(columns=['Loud Cover'], inplace=True) 删除该列并在原数据上应用更改。：

Loud Cover 列中的值全部为 0，推测为异常数据，将该列删除。

更新后的数据列：

Index(['Formatted Date', 'Summary', 'Precip Type', 'Temperature (C)',

'Apparent Temperature (C)', 'Humidity', 'Wind Speed (km/h)',

'Wind Bearing (degrees)', 'Visibility (km)', 'Pressure (millibars)',

'Daily Summary'],

dtype='object')

**3.4 特征构建**

为使得后续数据处理更方便，我们对部分特征进行构建。

首先我们使用特征分箱，在数据处理过程中，分箱是一种将连续变量划分为离散类别的方法。分箱有助于将连续数据转换为分类数据，以便于进一步分析和处理。在天气数据处理中，分箱可以帮助我们将时间、风速、风向等连续变量转换为有意义的分类，以便于后续的分析和建模。

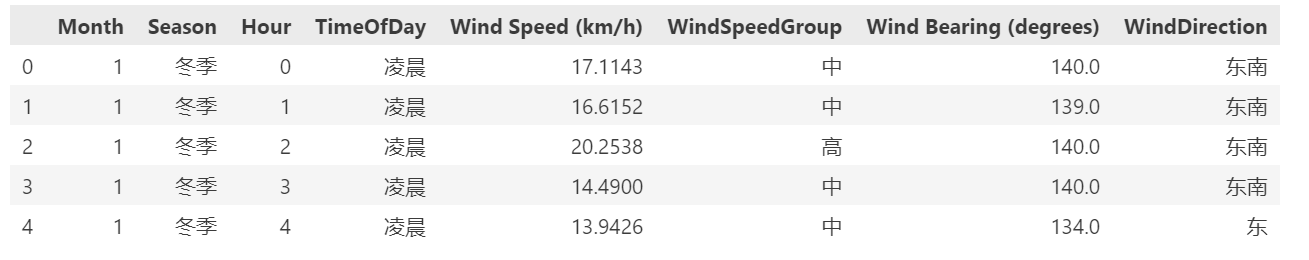
接着我们进行特征交互，特征交互是指通过组合现有特征来创建新的特征，以捕捉更复杂的关系。适合特征交互的特征通常是数值特征，以下特征可以进行特征交互：

Temperature (C) 与 Apparent Temperature (C)：温度和体感温度的差值可以反映人体对气温的感知差异

Humidity 与 Temperature (C)：湿度和温度的组合可以用来计算热指数。

Wind Speed (km/h) 与 Wind Bearing (degrees)：风速和风向的组合可以用来计算风的分量（东向分量和北向分量）。

效果如下：



表格

描述已自动生成

# 数据挖掘

## 4.1 聚类

K-means算法是一种常见的聚类算法，用于将数据集分成K个簇。其工作原理是：首先随机选择K个初始中心点，然后重复以下步骤：1) 将每个数据点分配到距离最近的中心点所在的簇；2) 重新计算每个簇的中心点，即簇内所有点的平均值。重复上述过程，直到中心点的位置不再发生显著变化或达到预定的迭代次数。K-means算法的目标是最小化簇内数据点到中心点的总距离，从而使得同一簇内的数据点尽可能相似，而不同簇之间的数据点尽可能不同。

我们对数据集中的Temperature (C)、 Humidity、Wind Speed (km/h)这几个数据进行聚类分析，使用k-means算法。

我们首先使用肘部法，观察到k=3时出现拐点，我们选取该值作为k-means算法的k值。

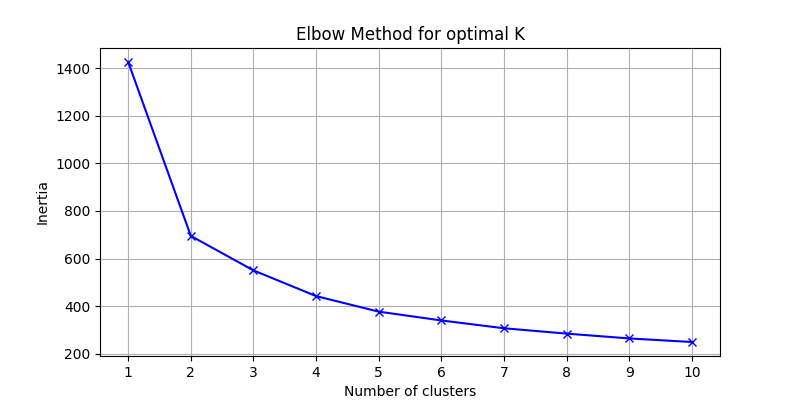


图9 肘部法求k值

由于原数据集训练量较大，聚类图中点过多影响观察效果，对原数据绘制聚类图时采用了采样的数据约减方法。

我们使用了Matplotlib库中的scatter()函数来绘制 3D 聚类图

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

图10 温度，湿度，风速的3D聚类图

由此图，我们不难得出以下聚类结果：

（1）紫色（Cluster 0）：这一类数据的点主要集中在湿度较高、温度和风速适中的区域。

（2）青绿色（Cluster 1）：这一类数据的点主要分布在温度适中、湿度较低的的区域。

（3）黄色（Cluster 2）：这一类数据的点主要分布在温度较高、湿度较低的区域。

而在图像中可以看到，Cluster 0 和 Cluster 1 有一定的重叠，说明它们在某些特征上比较接近。Cluster 2 的数据点相对集中在一个特定的区域，与其他两个聚类有明显的区别。这和现实中欧洲自然气候条件较为温和。多数时间气候处于温暖潮湿（Cluster 0），干燥温暖（Cluster 1），干燥炎热（Cluster 2）的情况。

## 4.2 分类

**·4.2.1 已知多变量对连续单变量的预测**

**·4.2.1.1 线性回归模型**

该分类任务的目标变量为Temperature (C)（温度），输入特征为所有其他特征，包括日期、分类变量和数值变量。

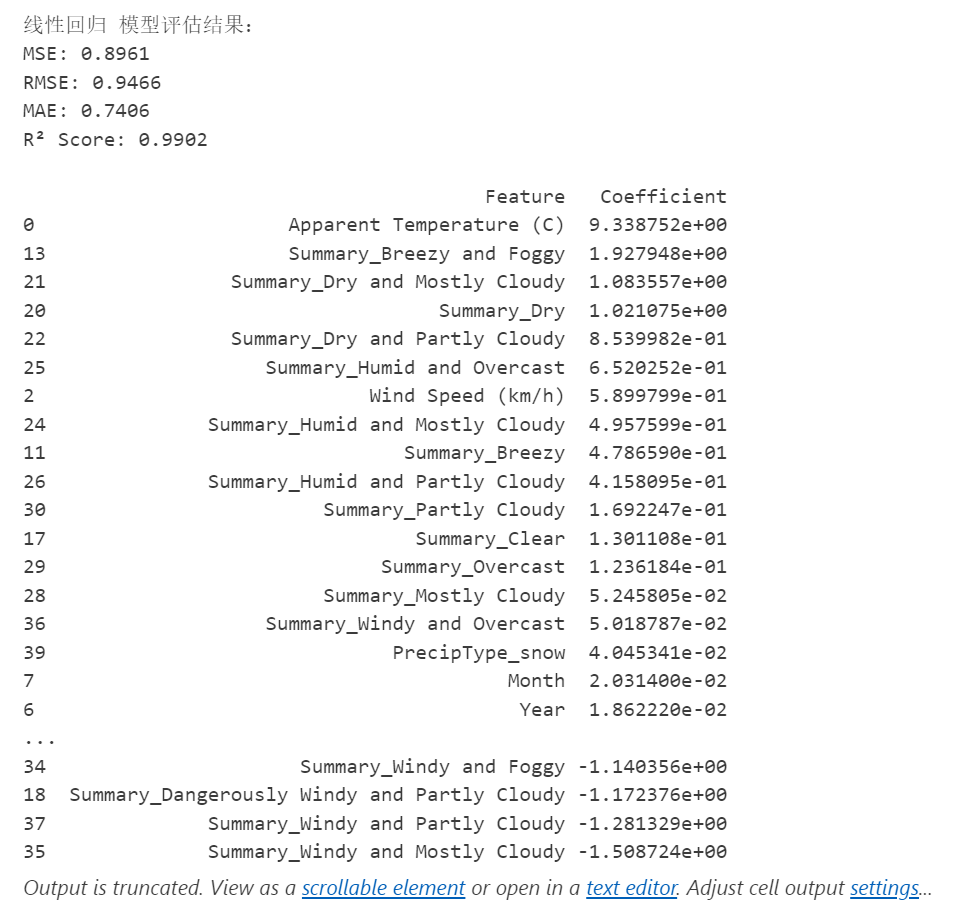
我们使用sklearn机器学习库，现将数据集进行处理，将对 'Summary' 和 'Precip Type' 进行独热编码，并将原先的的数据列删除以免干扰预测；我们采用线性回归模型，以80%：20%的训练集：测试集比例对模型进行训练，训练结果如图：  


图11 线性回归模型评估结果

由该图可得模型性能指标：

·均方误差 (MSE): 0.8961

·均方根误差 (RMSE): 0.9466

·平均绝对误差 (MAE): 0.7406

·R² Score: 0.9902

R² Score (0.9902)：表示模型解释了温度变化的99.02%。这是一个非常高的值，通常表明模型拟合得非常好。然而，过高的R²可能暗示着模型过拟合，尤其是在特征数量较多的情况下。

MSE, RMSE, MAE：这些误差指标显示了预测值与真实值之间的差异。较低的值通常表示模型性能良好。

再对模型系数分析，以下是部分特征及其对应的回归系数：

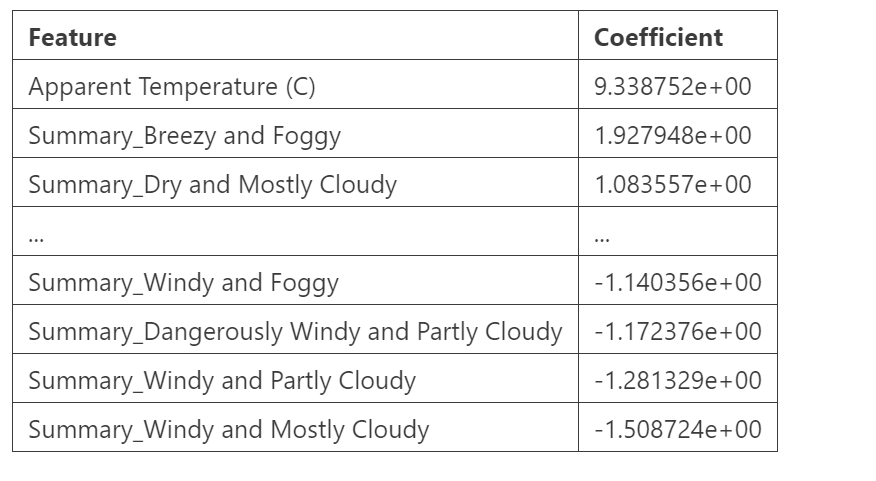


图12 部分特征对于的回归系数

正向影响较大的特征：

Apparent Temperature (C): 回归系数为9.34，表明体感温度对实际温度有显著的正向影响。这是合理的，因为体感温度通常与实际温度密切相关。

Summary\_Breezy and Foggy、Summary\_Dry and Mostly Cloudy等天气描述特征也对温度有正向影响。

负向影响较大的特征：

Summary\_Windy and Foggy、Summary\_Dangerously Windy and Partly Cloudy 等天气描述特征对温度有显著的负向影响。这可能表明在这些特定天气条件下，温度较低。

**·4.2.1.2随机森林回归模型**

为了全面评估模型性能，可以继续训练和评估随机森林回归模型。随机森林能够捕捉特征之间的非线性关系，并提供特征重要性分析[5]。我们导入随机森林相关的库sklearn.ensemble，训练结果如下

随机森林回归模型评估结果:

MSE:0.0024

RMSE:0.0494

MAE:0.0136

R2 Score:1.0000

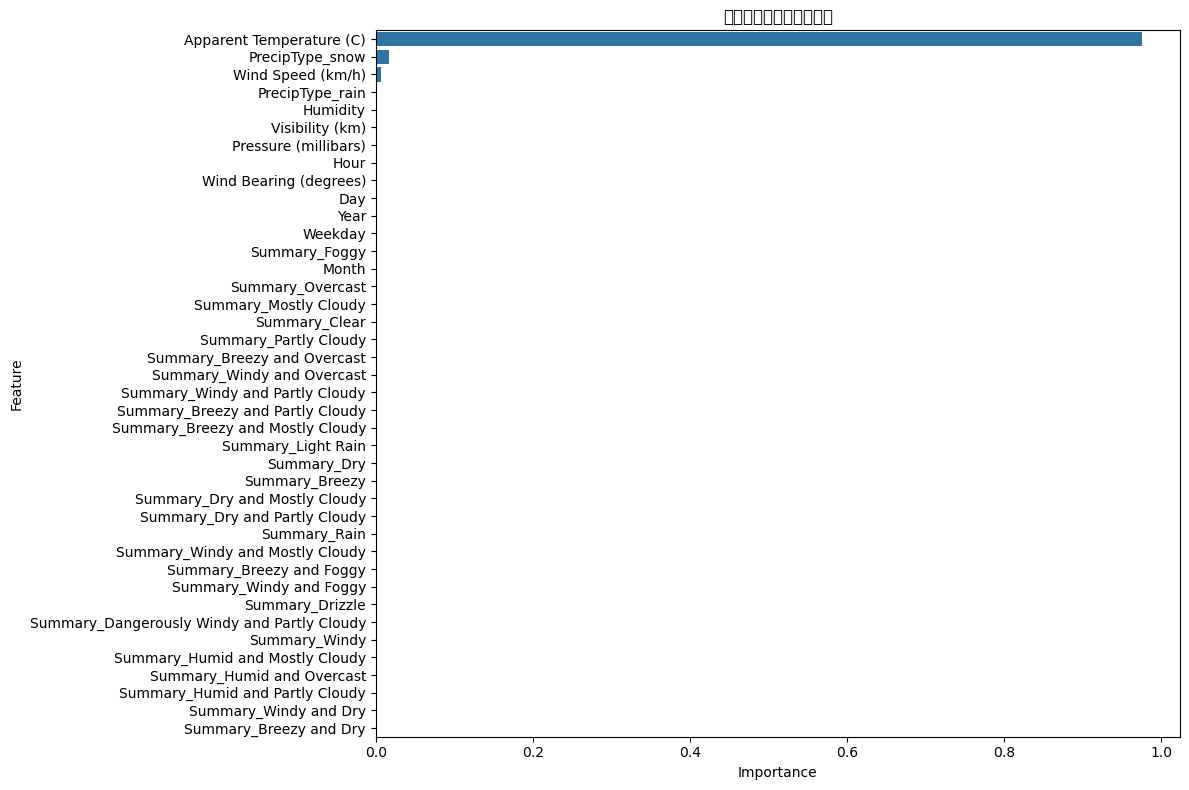


图13 随机森林模型训练结果1

结果中R²得分为1.0，非常不寻常，通常情况下，这可能是由于数据泄露或模型过拟合等问题引起的。需要注意的是，Apparent Temperature (C)（体感温度）与Temperature (C)通常高度相关，可能导致模型几乎完美地预测。

于是我们需要检查特征与目标变量的相关性，计算Apparent Temperature (C)与Temperature (C)之间的相关系数近似于1，说明这两个变量几乎是线性相关的。

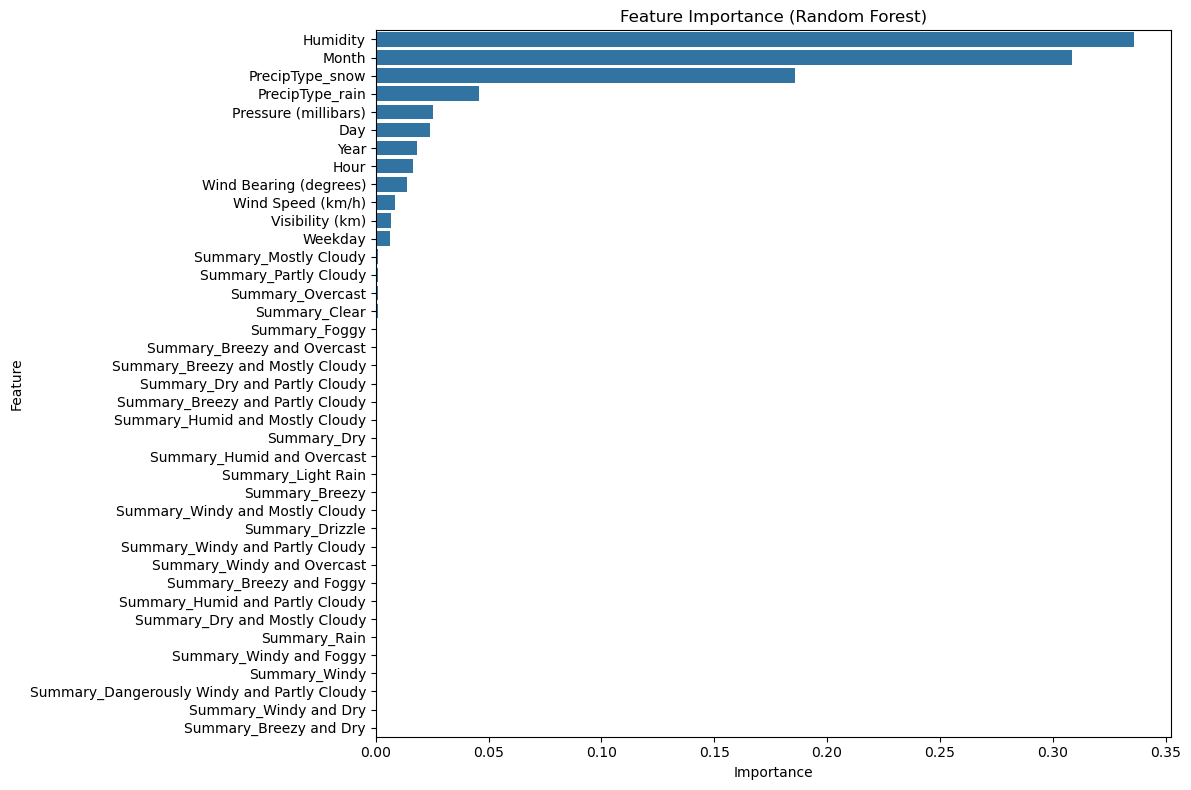
故在这种情况下，模型可以轻松地利用Apparent Temperature (C)来预测Temperature (C)，导致过拟合。由于Apparent Temperature (C)与Temperature (C)高度相关，我们从特征矩阵X中移除该特征。新的训练结果为：  


图14 随机森林模型训练结果2

随机森林回归模型在移除高度相关特征（Apparent Temperature (C)）后，仍然显示出非常高的性能（R² Score: 0.9707）。后续还需要通过交叉验证的方式来增强模型的泛化能力。

**·4.2.1.3 交叉验证**

在移除高度相关特征（Apparent Temperature (C)）后，再使用之前的线性回归模型进行对比。

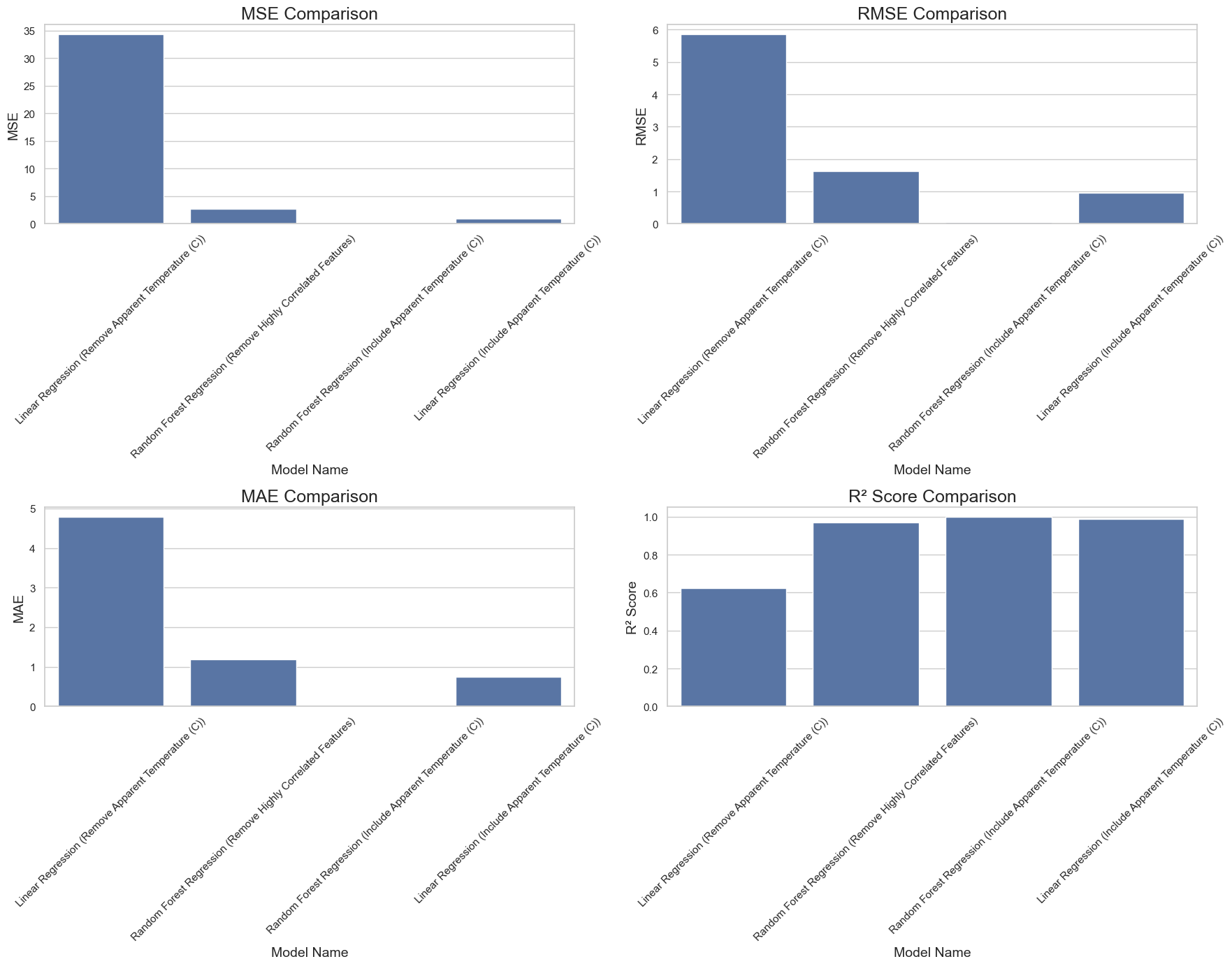


图15 MSE, RMSE, MAE, R² Score模型对比

**·4.2.2 已知多变量对离散单变量的预测**

**·4.2.2.1. 准确度（Accuracy）公式：**

准确度衡量模型预测正确的比例，计算公式为：

·TP（True Positive，真正类）：预测为正类且实际为正类的数量。

·TN（True Negative，真负类）：预测为负类且实际为负类的数量。

·FP（False Positive，假正类）：预测为正类但实际为负类的数量。

·FN（False Negative，假负类）：预测为负类但实际为正类的数量。

准确度并不能总是反映模型的整体表现，特别是在类别不平衡的数据集中。

**·4.2.2.2. ROC-AUC 曲线：**

ROC 曲线（Receiver Operating Characteristic Curve）表示在不同的分类阈值下，模型的 假正率（FPR）和 真正率（TPR）之间的关系。

**TPR（True Positive Rate，真正率）**，即 召回率（Recall），计算公式为：

**FPR（False Positive Rate，假正率）**，计算公式为：

AUC（Area Under Curve）\*\*：即 ROC 曲线下的面积，表示模型区分正负类能力的一个综合指标。AUC 值越接近 1，模型的区分能力越强；若 AUC=0.5，则模型的表现与随机猜测没有区别。

**·4.2.2.3. 多分类问题下 TPR 和 FPR 的计算：**

对于多分类问题，通常使用 "一对多"（One-vs-Rest, OvR）的方法来计算 TPR 和 FPR。

**真正率 (TPR) 公式：**

每个类别的真正率（Recall）是在将该类别作为正类时的召回率，公式为：

其中：

TP 是第i类的真正类（预测为第i类且实际为第i类的样本数）。

FN 是第i类的假负类（实际为第i类但预测为其他类的样本数）。

**假正率 (FPR) 公式：**

假正率是将其他所有类别作为负类时的错误率，公式为：

其中：

FP 是第i类的假正类（实际为其他类别但预测为第i类的样本数）。

TN是第 i 类的真负类（实际为其他类别且预测也为其他类别的样本数）。

**·4.2.2.4. 多分类问题的 ROC-AUC 计算：**

通常有两种方法来计算多分类问题的 AUC 值：

**(1) 宏平均（Macro-Averaging）：**

对每个类别的 AUC 值进行简单平均：

**(2) 微平均（Micro-Averaging）：**

将每个类别的 TP、FP、TN 和 FN 累加起来，计算整体的 TPR 和 FPR，绘制整体的 ROC 曲线并计算 AUC 值：

曲线下的面积 (基于所有类别的预测结果)

同样，ROC-AUC曲线也有一个严重的问题，精确率为 0 对 AUC 没有很大影响

**4.2.2.5 预测性任务**

目标变量: Summary(天气摘要)Summary 是一个描述天气状况的分类变量，因此我们可以将其视为多分类问题。

 数据预处理：

* 转换日期格式并提取年份、月份、日期、小时和星期几等时间特征。
* 删除原始日期列和摘要列，以减少不必要的信息。
* 将目标变量 'Summary' 编码为数字形式。
* 处理 'Precip Type' 列中的缺失值，并进行独热编码。
* 删除高度相关的特征以避免数据泄露。

然后我们使用随机森林模型，对特征重要性分析：评估各特征的重要性并可视化展示：

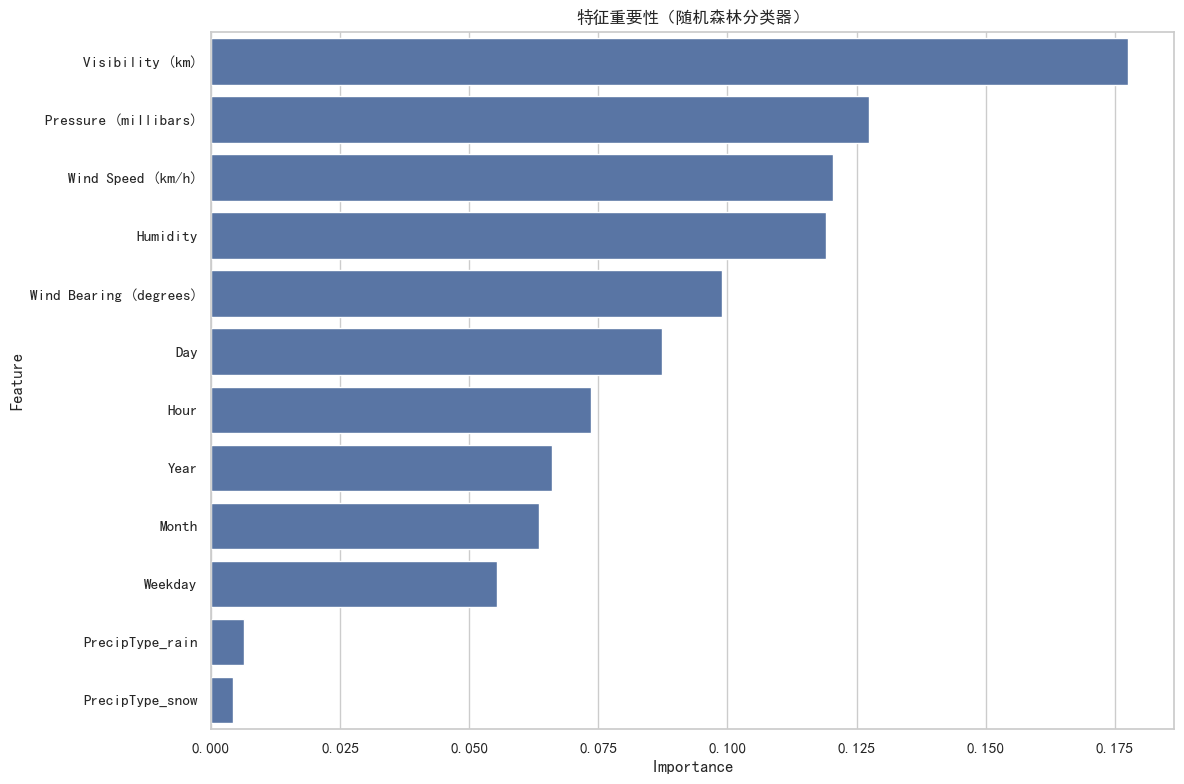


图16 特征重要性

此时模型准确率只有0.6936。因而需要通过**超参数**优化区进行调优，超参数则是在算法运行之前手动设置的参数，用于控制模型的行为和性能。这些超参数的选择会影响到模型的训练速度、收敛性、容量和泛化能力等方面。例如，学习率、迭代次数、正则化参数、隐藏层的神经元数量等都是常见的超参数。使用超参数之后：

初始模型准确率：0.6936

优化后模型准确率：0.7138

模型的整体准确率从 69.36% 提升到了 71.38%，这表明经过超参数调优后，模型的性能有所提升。

由此我们也发现了少数类别的性能问题：对于一些少数类别，如 'Breezy and Dry'、'Dry and Mostly Cloudy'、'Humid and Mostly Cloudy' 等，precision 和 recall 都为 0，这意味着模型无法正确预测这些类别。于是我们继续优化：

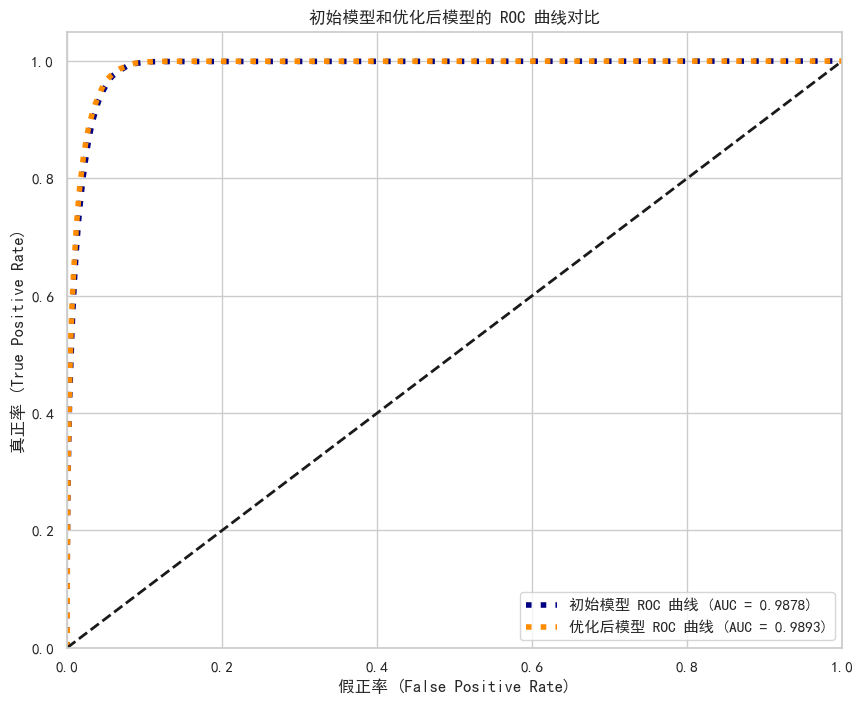


图17 优化后ROC曲线

在进行超参数优化之后，可以注意到模型的能力有所提升，从优化前的0.9878提升到了0.9893。

**·4.2.3 时间序列上的预测**

我们使用ARIMA模型进行单变量时间序列数据建模，选择Temperature (C)列为建模对象。

利用训练集的温度数据（train\_temperature），使用ARIMA模型（参数设置为 (5, 1, 0)）进行建模。ARIMA模型的参数分别表示自回归阶数（p=5）、差分次数（d=1）和滑动平均阶数（q=0）。该模型被训练后用于预测未来24小时的温度。然后使用ARIMA模型的 forecast 方法预测接下来的24小时温度，并与测试集中的实际温度值进行比较[1]。通过计算均方误差（MSE）和均方根误差（RMSE），评估模型的预测性能。最后，绘制出测试集实际温度与模型预测温度的曲线图，可视化模型的预测效果如下：

Length of test\_temperature: 19287

Shape of y\_true: (24,)

Shape of y\_pred: (24,)

ARIMA model RMSE: 4.642457203508018

图表, 折线图

描述已自动生成

图18 ARIMA可视化模型

如图，我们成功生成了ARIMA模型的预测结果和相应的图表，且预测的均方根误差（RMSE）为 4.64。从图中可以看到，模型预测的温度（红色线）相对平稳，而实际温度（蓝色线）有较大的波动。

接下来我们建立SARIMA模型，效果图如下：

SARIMA model RMSE: 3.5695041571623998

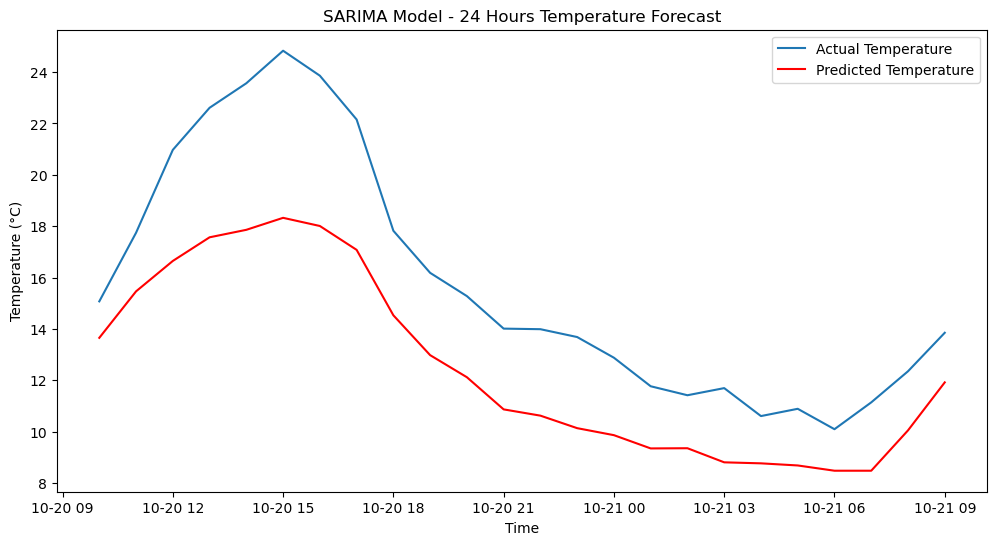


图19 SARIMA可视化模型

为了实现更好的预测效果，我们构建 LSTM 模型：使用 Keras 构建一个包含两个 LSTM 层和一个全连接层的模型。第一层 LSTM 具有100个神经元，并设置 return\_sequences=True 以便向下一层传递序列数据。第二层 LSTM 输出最终的特征向量。最后，输出层为一个神经元，用来预测下一个时间步的温度。编译模型并使用训练数据进行训练，优化器为 adam，损失函数为均方误差（MSE）。训练时使用批量大小64，并训练3个周期（epochs=3）。效果图如下：

图形用户界面, 图表, 直方图

描述已自动生成

图20 LSTM预测

之前的预测效果较好但是预测为短时间的预测，为进行长时间序列(一天、一周、一个月)的预测，我们采用了滚动预测：

图形用户界面, 图表

描述已自动生成

图21 滚动预测（长时间序列）

可以看出长时间序列下的预测效果不佳。我们优化了滚动序列[2]的算法，并采用多进程预测函数，得出的最佳预测效果如下：

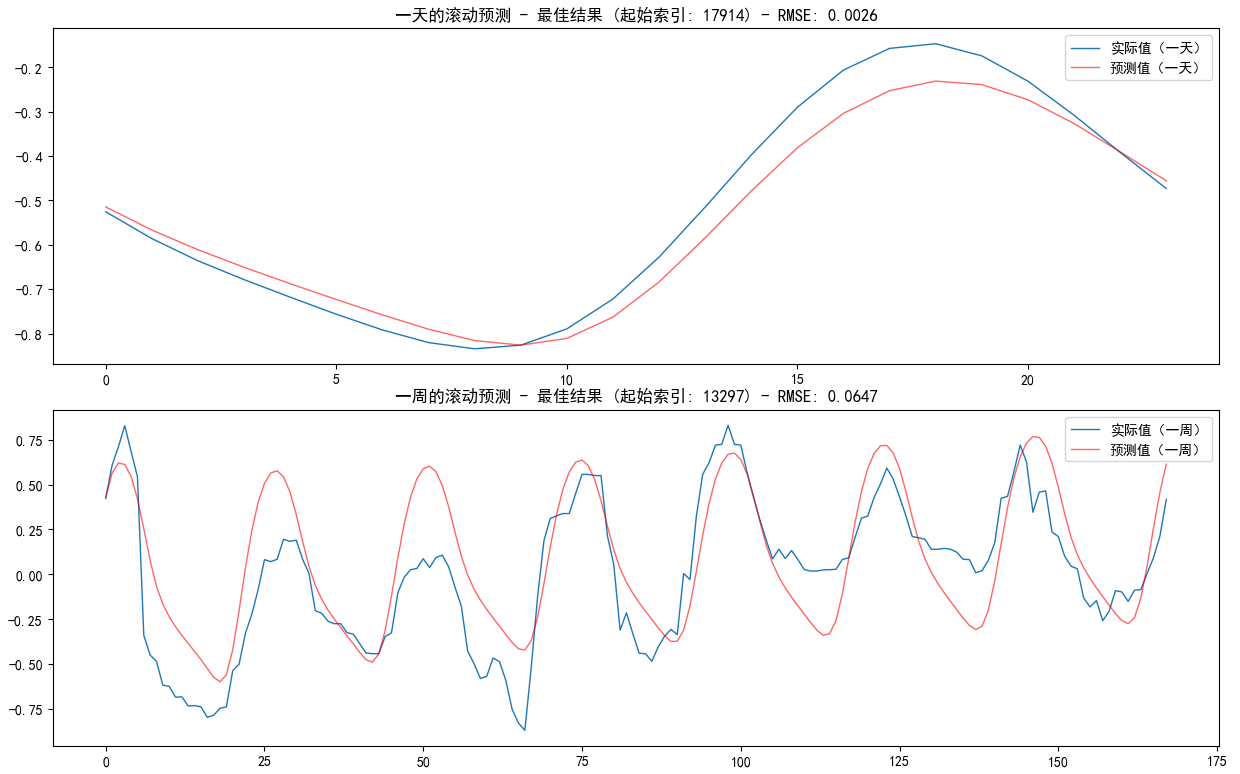


图22 滚动预测（长时间序列优化）

为了展现模拟能力与预测时间长度的关系，我们还计算了不同预测时间长度下的RSME的关系。

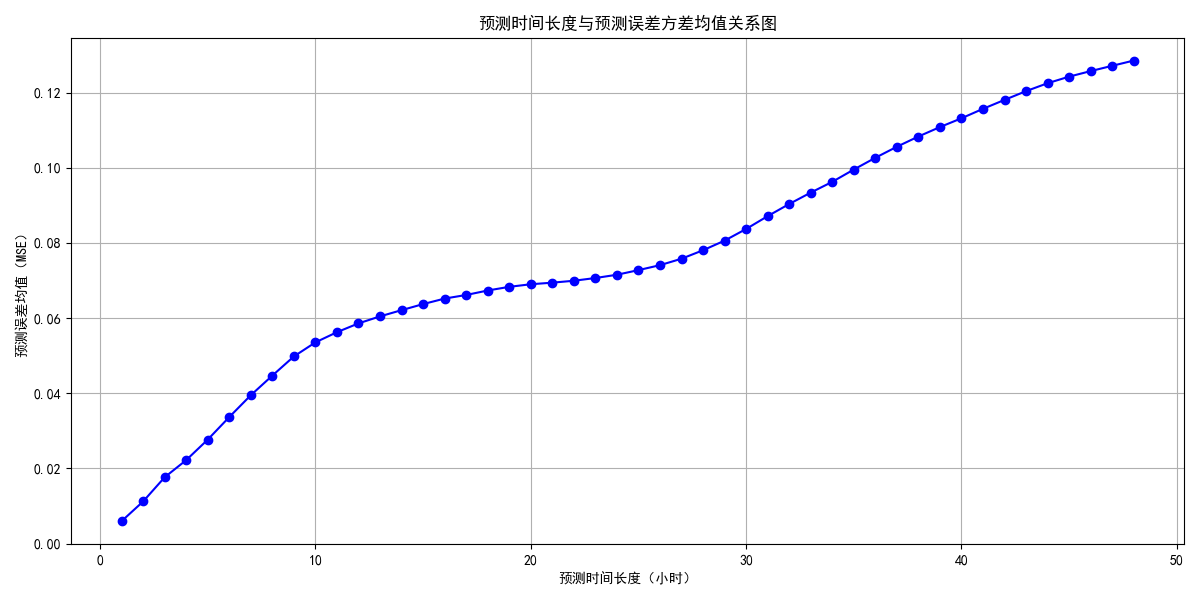


图23 预测时间长度与预测误差方差均值关系图

可以看到模型的能力会随着预测时间的增长而降低，但是当时间到1天/2天左右的时候能力又会有所回升。可能的原因是相差两天相同时间的温度接近。

**未来的优化方向**：

**1. 贝叶斯优化** 使用贝叶斯优化方法优化超参数，提高模型性能。

**2. 使用双向LSTM** 双向LSTM能够同时捕捉序列的前向和后向信息。

**3. 引入注意力机制** 注意力机制可以帮助模型关注序列中的重要部分。

## 4.3 特征重要性分析

为计算可视化SHAP值，我们创建 SHAP 解释器，使用TensorFlow 中的 KernelExplainer() 进行模型解释。定义了一个 model\_predict 函数，用于将数据从二维转换为 LSTM 模型需要的三维格式，并进行了可视化 SHAP 结果：

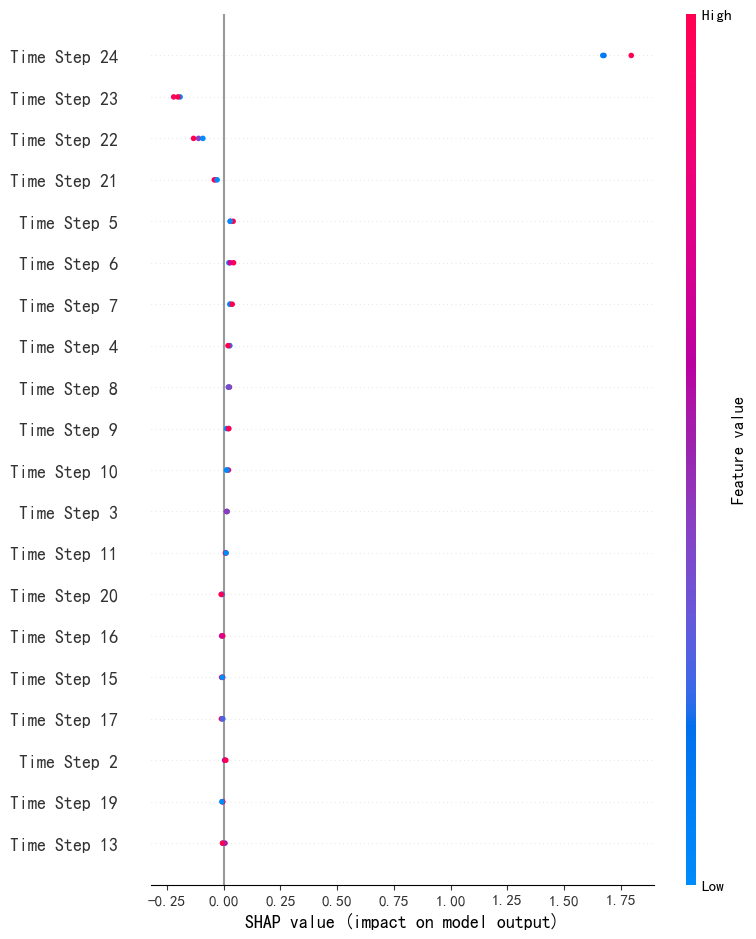
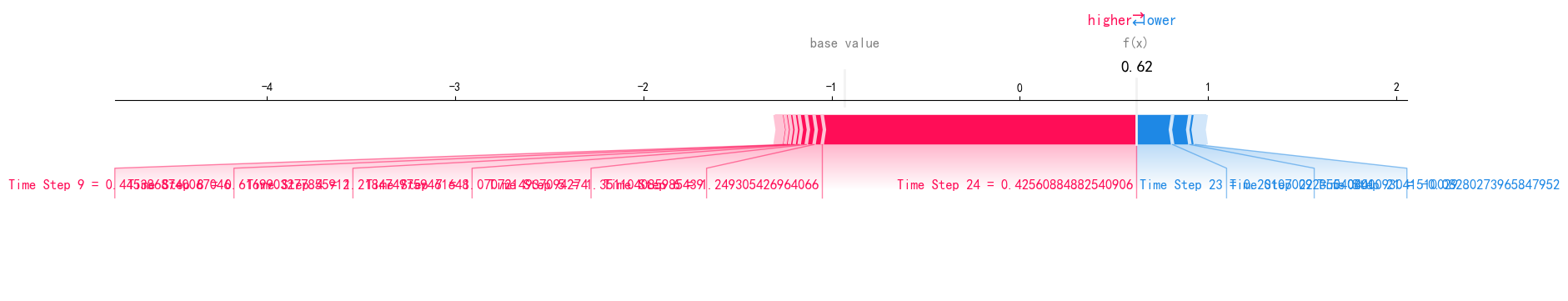


图24 可视化总体 SHAP 概述



图表

描述已自动生成

图25 特定时间步的 SHAP 值分布

## 4.4 关联规则挖掘

我们先进行数据离散化，接着将每条记录转换为事务，包含离散化后的特征值。再者，

我们使用使用Apriori算法进行关联规则挖掘，使用mlxtend库中的Apriori算法挖掘频繁项集：

各列的解释[3]

antecedents（前件）：规则的左侧部分，表示条件。

consequents（后件）：规则的右侧部分，表示结果。

antecedent support（前件支持度）：在数据集中，前件出现的频率。

consequent support（后件支持度）：后件出现的频率。

support（支持度）：同时满足前件和后件的记录频率。

confidence（置信度）：给定前件的情况下，后件发生的概率。计算公式为 support(antecedents ∩ consequents) / support(antecedents)。

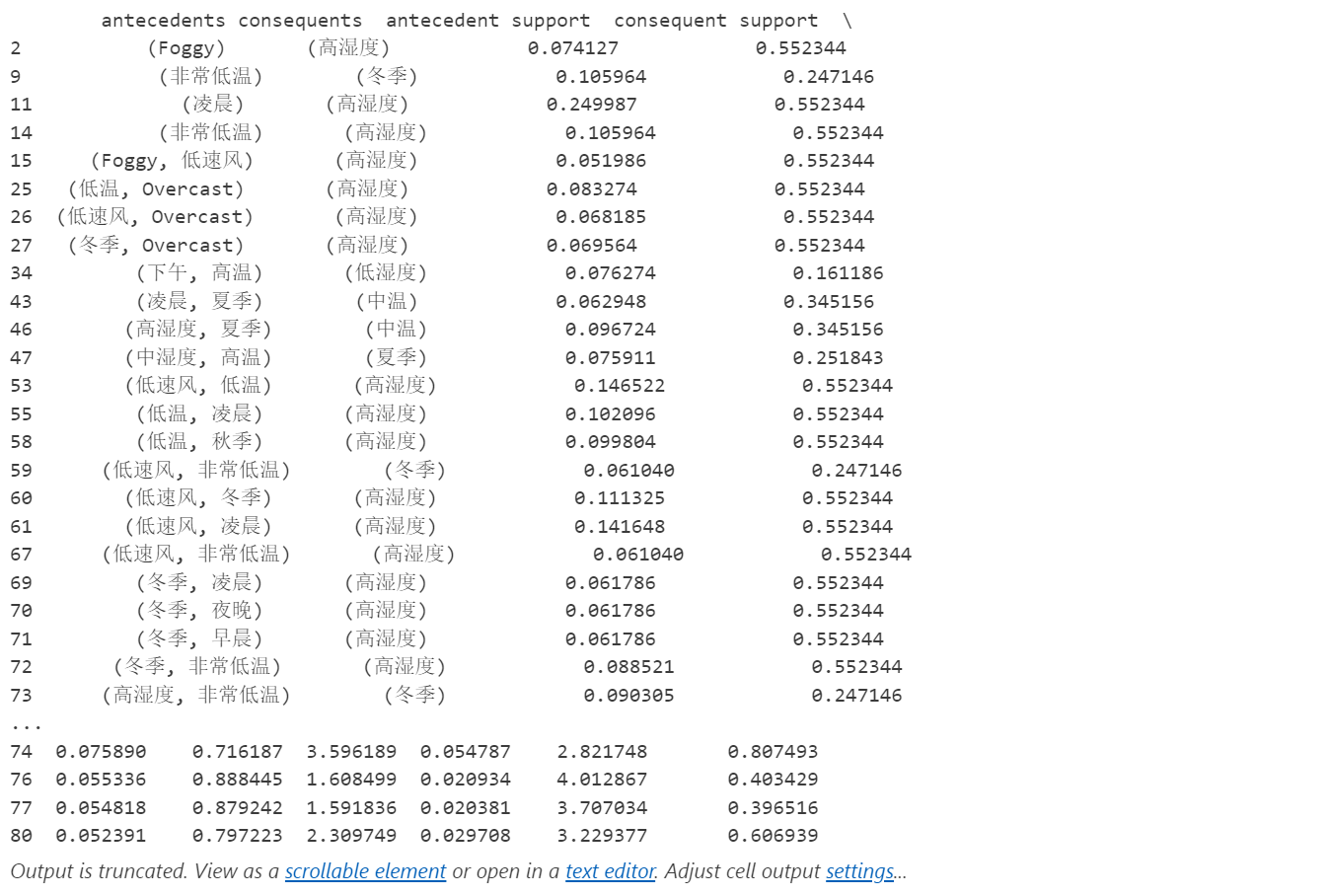
lift（提升度）：后件在给定前件情况下发生的可能性相对于其独立发生的可能性。计算公式为 confidence / support(consequents)。值大于1表示前件对后件有正向影响。

leverage（杠杆值）：反映前件和后件共同发生的超出随机发生的程度。计算公式为 support(antecedents ∩ consequents) - (support(antecedents) \* support(consequents))。值越大表示相关性越强。

conviction：衡量前件发生时后件不发生的可能性。计算公式为 (1 - support(consequents)) / (1 - confidence)。值越大表示前件对后件的影响越强。

zhangs\_metric：Zhang的指标，用于评估规则的重要性，通常在某些特定的研究中使用。

输出效果如图：



在上述数据中，我们看到一系列的关联规则，这些规则描述了不同气象条件（如温度、湿度、风速等）之间的关系。通过分析这些规则的度量值，我们可以评估它们的有用性和可靠性。于是，我们进行了规则分析。

**规则分析：**

1.**规则："Foggy" -> "高湿度"**

**Confidence (置信度)**: 0.9959，非常高，意味着几乎可以确定当天气“雾”时，湿度会很高。

**Lift (提升度)**: 1.8031，表明“雾”和“高湿度”之间存在较强的正相关关系。

**有用性**: 对于农业、交通等行业来说，这个规则非常有用，可以帮助预测恶劣天气，采取预防措施。

2. **规则："非常低温" -> "冬季"**

**Confidence (置信度)**: 0.8354，较高，说明当气温非常低时，很可能是冬季。

**Lift (提升度)**: 3.3801，表示“非常低温”与“冬季”之间有很强的联系。

**有用性**: 对于季节性活动规划或能源需求预测等领域，此规则具有实际应用价值。

3. **规则："凌晨" -> "高湿度"**

**Confidence (置信度)**: 0.8405，高，表示凌晨时分湿度较高的可能性很大。

**Lift (提升度)**: 1.5216，显示“凌晨”与“高湿度”之间有一定的相关性。

**有用性**: 对于气象预报或户外活动安排，这条规则可以提供参考信息。

4. **规则："非常低温" & "高湿度" -> "冬季"**

**Confidence (置信度)**: 0.8404，较高，意味着当同时出现“非常低温”和“高湿度”时，几乎可以断定是冬季。

**Lift (提升度)**: 3.4003，非常高的提升度反映了这两者组合对冬季的强烈指示作用。

**有用性**: 这个规则对于季节性气候变化的研究特别有价值。

从以上分析可以看出，这些关联规则提供了关于特定气象条件下其他气象条件发生的概率的信息。对于需要根据天气状况做出决策的领域，比如农业灌溉、交通管理、旅游规划等，这些规则都是非常有用的工具。然而，需要注意的是，虽然这些规则在统计上有效，但在实际应用时还需要结合其他因素进行综合判断。例如，地理位置、季节变化等因素都可能影响规则的应用效果。因此，在利用这些规则时，应该考虑到这些外部因素的影响。

# 异常检测

## 5.1 DBSCAN

DBSCAN是一种基于密度的聚类算法，它通过定义核心点、边界点和异常点，以及直接密度可达、密度可达和密度相连的概念，能够在包含噪声的空间数据库中识别任意形状的聚类，并将高密度区域划分为簇[4]。

我们采用eps=0.5, MinPts =5的参数对数据集进行聚类异常检测。

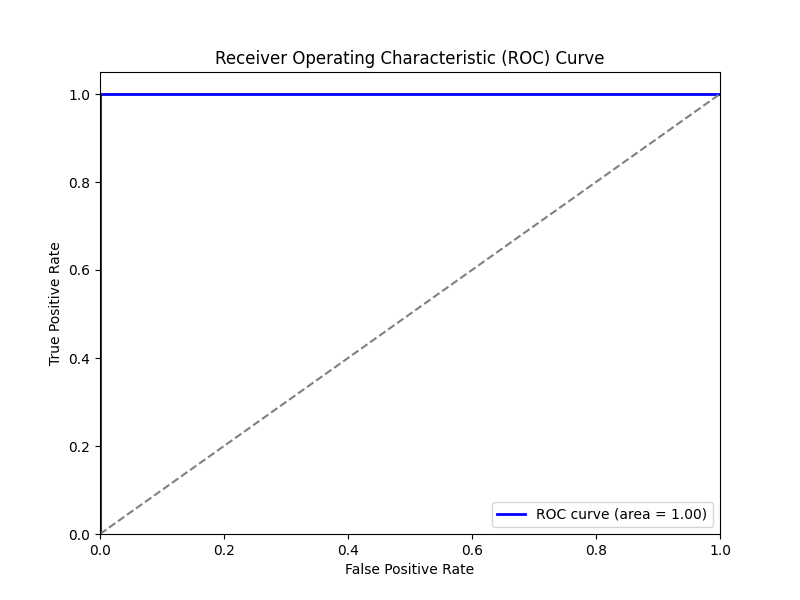


图26 DBSCAN异常检测

噪声数量: 1845

噪声比: 0.019133248296674236

混淆矩阵:

[[94584 0]

[ 0 1845]]

F1 分数: 1.0000

可以得到数据集噪声点较少，ROC召回率非常高。

**5.2 LOF**

局部离群因子（Local Outlier Factor, LOF）是一种基于密度的异常检测算法。它通过比较一个点的局部密度与其邻居的局部密度，来判断该点是否为异常点。如果一个点的局部密度显著低于其邻居的密度，则该点可能是一个异常点。LOF算法的核心思想是利用局部密度偏差来检测异常，适用于数据分布不均匀的情况。

我们采用n\_neighbors=20， contamination=0.1的参数对数据集进行聚类异常检测。

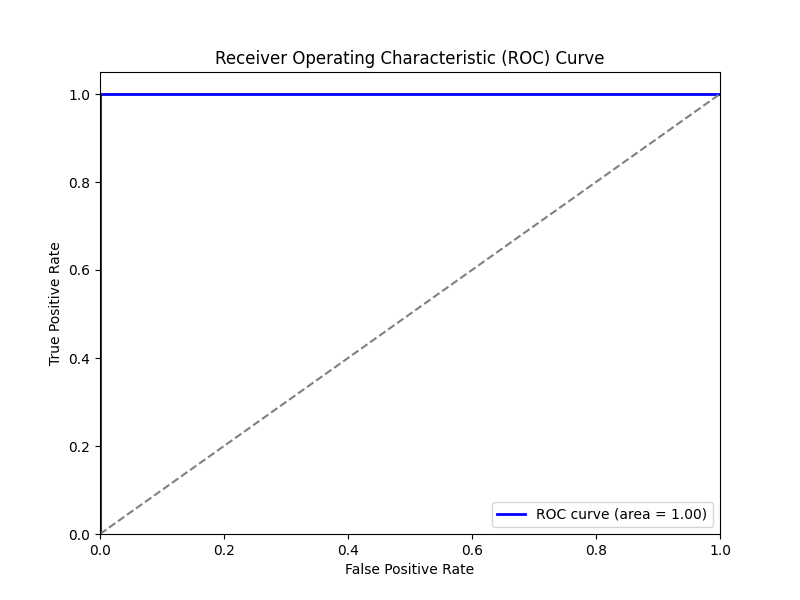


图27 LOF异常检测

噪声数量: 9643

噪声比: 0.100001037032428

混淆矩阵:

[[86786 0]

[ 0 9643]]

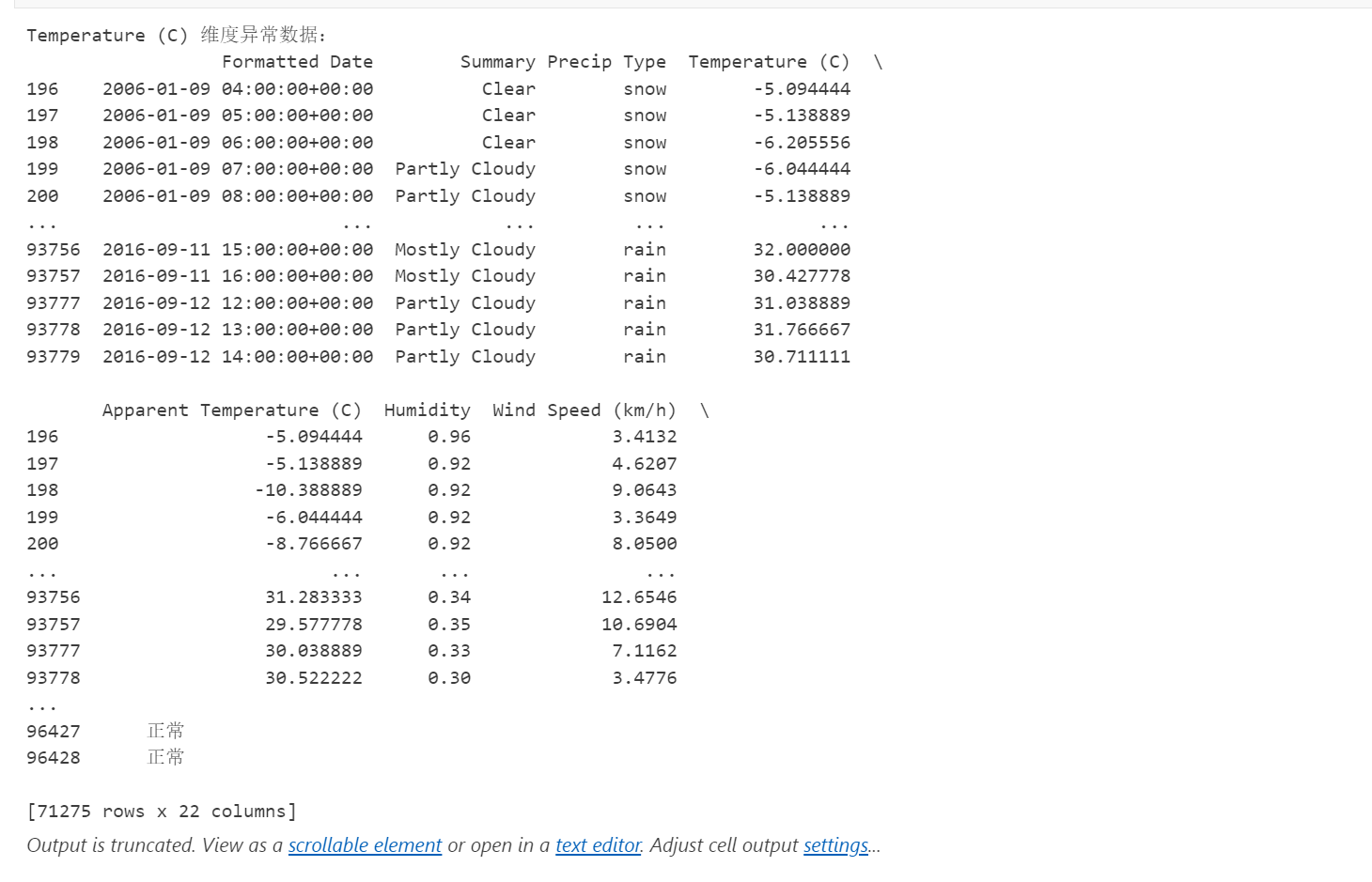
F1 分数: 1.0000

可以看出，噪声数量变多，可能是因为天气数据分布较为均匀导致其检测噪声数多于DBSCAN。

**5.3 随机森林**

我们对数据中的每个特征进行异常值检测。它逐个特征地训练孤立森林模型，识别并标记异常数据点，并将异常数据保存为单独的 CSV 文件。异常值会被删除，最终生成一个净化后的数据集。每个特征的异常数据和处理后的结果都被保存，确保数据集仅包含正常数据。

维度异常输出如图：



参考文献：

1. Box, G. E. P., Jenkins, G. M., Reinsel, G. C. Time Series Analysis: Forecasting and Control. Wiley Series in Probability and Statistics, 5th edition, 2015.
2. Krauss, C. Data Sampling and Forecasting with Rolling Windows: A Comparative Study. Computational Statistics & Data Analysis, 2017, 113, 132-148. DOI: 10.1016/j.csda.2017.02.015
3. Al Shehabi, S., Baba, A. MARC: Mining Association Rules from datasets by using Clustering models. International Journal of Data Science and Analytics, 2019, 8(3), 215-230.
4. Fayyad, U. M., et al. A Survey of Clustering Algorithms for Big Data: Taxonomy & Empirical Analysis. Journal of Big Data, 2020, 5(1), 1-22.
5. Breiman, L. Random Forests. Machine Learning, 2001, 45(1), 5-32.

# 附录

## 6.1 未来挑战

1. 进一步通过更新算法和调整参数来优化异常检测模型。

2. 继续优化数据挖掘任务中的预测和描述性任务。

## 6.2 人员分工

俞乐楠：分类、线性回归分析、数据清洗、数据可视化、报告撰写、ppt制作

徐文彬：数据可视化探索分析、关联规则挖掘、异常检测、报告撰写、ppt制作

傅裕翔：数据清洗、异常检测、聚类分析、报告撰写与整合、ppt制作与整合

## 6.3 代码

**数据集可视化分析：**

# 导入必要的库

import os

import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

import warnings

warnings.filterwarnings("ignore")

# 检查数据集是否存在

def check\_data(path: str):

    if not os.path.exists(path):

        print("路径不存在")

    else:

        print("路径存在")

data\_path="weatherHistory.csv"

check\_data(data\_path)

# 加载数据集

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

df['Formatted Date'] = df['Formatted Date'].str[:-6]#  去掉后五个字符

df['Formatted Date'] = pd.to\_datetime(df['Formatted Date'], format='%Y-%m-%d %H:%M:%S.%f', errors='coerce')# 将 'Formatted Date' 列转换为日期时间类型

df["Summary"]=df["Summary"].astype("category")

df["Precip Type"]=df["Precip Type"].astype("category")

df["Daily Summary"]=df["Daily Summary"].astype("category")

print(df.dtypes)

df.head()

"""画图"""

plt.rcParams['font.family'] = ["Times New Roman", 'SimSun']

fig, axs = plt.subplots(1, 5, figsize=(15, 5))

# 箱线图

# 绘制第一个箱线图并添加统计值标签

sns.boxplot(y=df['Temperature (C)'], ax=axs[0],color='skyblue')

axs[0].set\_title('温度箱线图')

# 绘制第二个箱线图并添加统计值标签

sns.boxplot(y=df['Humidity'], ax=axs[1],color='lightgreen')

axs[1].set\_title('湿度箱线图')

# 绘制第三个箱线图并添加统计值标签

sns.boxplot(y=df['Wind Speed (km/h)'], ax=axs[2],color='lightcoral')

axs[2].set\_title('风速箱线图')

# 绘制第四个箱线图并添加统计值标签

sns.boxplot(y=df['Wind Bearing (degrees)'], ax=axs[3],color='lightpink')

axs[3].set\_title('风向箱线图')

# 绘制第五个箱线图并添加统计值标签

sns.boxplot(y=df['Visibility (km)'], ax=axs[4],color='lightyellow')

axs[4].set\_title('能见度箱线图')

# 调整布局

plt.tight\_layout()

plt.show()

# 天气数量分布图-柱状图

df\_weather\_row\_count, df\_weather\_column\_count=df.shape

print('Total number of rows:', df\_weather\_row\_count) # 行数

print('Total number of columns:', df\_weather\_column\_count) # 列数

print ("Unique values are:\n",df.nunique()) # 查看每列的唯一值

Summary\_Weather=df["Summary"].value\_counts().reset\_index() # 查看天气类型的分布

Summary\_Weather.columns=["Weather Type","Count"] # 重命名列名

# Summary\_Weather # 显示天气类型的分布

# 绘制柱状图，从高到低排序，标记数字

plt.figure(figsize=(30,10))

sns.barplot(x="Weather Type",y="Count",data=Summary\_Weather,order=Summary\_Weather["Weather Type"],palette="viridis") # order=Summary\_Weather["Weather Type"]按照天气类型排序

plt.xticks(rotation=60)

plt.title('天气数量分布图')

plt.show()

# 每天的天气类型分布图-饼图

Daily\_Summary\_Weather=df["Daily Summary"].value\_counts().reset\_index() # 查看每天的天气类型分布

Daily\_Summary\_Weather.columns=["Daily Weather Type","Count"] # 重命名列名

# Daily\_Summary\_Weather # 显示每天的天气类型分布

# 绘制饼图,只取前10个，别的划分为其余，带箭头

plt.figure(figsize=(20,10))

plt.pie(Daily\_Summary\_Weather["Count"][:10],labels=Daily\_Summary\_Weather["Daily Weather Type"][:10],autopct='%1.1f%%',startangle=140)

plt.title('每天的天气类型分布图')

plt.show()

# 直方图

print(df.columns)

# 创建子图

fig,axes = plt.subplots(1, 3, figsize=(15, 5))

# 绘制直方图和 KDE,kde是核密度估计

sns.histplot(data=df, x='Temperature (C)', kde=True, bins=50, ax=axes[0])  # bins

axes[0].set\_title('温度直方图')

axes[0].set\_xlabel('age')

axes[0].set\_ylabel('prob')

# 按地区绘制 KDE

sns.kdeplot(data=df, x='Temperature (C)', common\_norm=False, ax=axes[1])

axes[1].set\_title('温度KDE')

axes[1].set\_xlabel('temp')

axes[1].set\_ylabel('prob')

# 按作物类型绘制 KDE

sns.kdeplot(data=df, x='Humidity',  common\_norm=False, ax=axes[2])

axes[2].set\_title('湿度KDE')

axes[2].set\_xlabel('Humidity')

axes[2].set\_ylabel('prob')

# 显示图形

plt.tight\_layout()

plt.show()

# Correlation heatmap

numeric\_df = df.select\_dtypes(include=[np.number])

plt.figure(figsize=(10, 8))

sns.heatmap(numeric\_df.corr(), annot=True, fmt='.2f', cmap='coolwarm')

# 设置横坐标倾斜

plt.xticks(rotation=45)

plt.title('热力图')

plt.show()

# 查看缺失值

wt\_missing =df.isna().sum()

wt\_missing

# 计算缺失值的百分比

t\_cells = np.product(df.shape)

t\_missing = wt\_missing.sum()

percent\_missing = (t\_missing/t\_cells) \* 100

print(percent\_missing)

# 删除缺失值

df['Precip Type'].fillna(df['Precip Type'].value\_counts().index[0],inplace=True)

df.isna().sum()

df['day\_of\_year']=df['Formatted Date'].dt.dayofyear

df['year']=df['Formatted Date'].dt.year

numeric\_cols=['Temperature (C)','Apparent Temperature (C)','Humidity','Wind Speed (km/h)','Wind Bearing (degrees)','Visibility (km)','Loud Cover','Pressure (millibars)']

df\_daily = df.groupby(df['Formatted Date'].dt.date)[numeric\_cols].mean().reset\_index()

df\_daily['day\_of\_year'] = pd.to\_datetime(df\_daily['Formatted Date']).dt.dayofyear

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 8))

# 温度折线图，一年一条折线，一条折线上有365个点，每个点代表一天的平均温度

sns.lineplot(data=df,x='day\_of\_year',y='Temperature (C)',hue='year',ax=axes[0])

axes[0].set\_title('每年的气温变化情况')

axes[0].set\_xlabel('day')

axes[0].set\_ylabel('℃')

axes[0].legend(title='年份')

# 湿度折线图，一年一条折线，一条折线上有365个点，每个点代表一天的平均温度

sns.lineplot(data=df,x='day\_of\_year',y='Hi (C)',hue='year',ax=axes[0])

axes[0].set\_title('每年的气温变化情况')

axes[0].set\_xlabel('day')

axes[0].set\_ylabel('℃')

axes[0].legend(title='年份')

df.head()

**数据清洗：**

import pandas as pd

# 读取 CSV 文件

df = pd.read\_csv('weatherHistory.csv')

# 检查每列的缺失值

missing\_values = df.isnull().sum()

# 输出每列缺失值的数量

print("缺失值检测结果：")

print(missing\_values)

import pandas as pd

# 读取 CSV 文件

print("正在读取文件 weatherHistory.csv...")

df = pd.read\_csv('weatherHistory.csv')

print("文件读取成功！")

# 检查 Precip Type 列中的缺失值数量

missing\_before = df['Precip Type'].isnull().sum()

print(f"替换之前，Precip Type 列中有 {missing\_before} 个缺失值。")

# 将 Precip Type 列中的缺失值替换为 'rain'

df['Precip Type'] = df['Precip Type'].fillna('rain')

# 检查替换结果

missing\_after = df['Precip Type'].isnull().sum()

print(f"替换之后，Precip Type 列中有 {missing\_after} 个缺失值。")

# 直接覆盖保存到原始 CSV 文件

print("正在保存更改到文件 weatherHistory.csv...")

df.to\_csv('weatherHistory.csv', index=False)

print("保存成功！")

# 输出保存后的信息

print("文件已更新，缺失值已处理完毕。")

import pandas as pd

# 读取 CSV 文件

df = pd.read\_csv('weatherHistory.csv')

# 检查 Loud Cover 列是否全部为0

if (df['Loud Cover'] == 0).all():

    print("Loud Cover 列中的值全部为 0，推测为异常数据，将该列删除。")

    # 删除 Loud Cover 列

    df.drop(columns=['Loud Cover'], inplace=True)

# 保存删除后的数据到原始文件

df.to\_csv('weatherHistory.csv', index=False)

# 输出确认删除后的数据列名

print("更新后的数据列：")

print(df.columns)

import pandas as pd

# 读取 CSV 文件

df = pd.read\_csv('weatherHistory.csv')

# 查找重复的 Formatted Date

duplicate\_rows = df[df['Formatted Date'].duplicated(keep=False)]

# 检查是否存在重复值并打印出来

if not duplicate\_rows.empty:

    print(f"发现 {len(duplicate\_rows)} 行重复的日期时间记录：")

    print(duplicate\_rows)

else:

    print("没有发现重复的日期时间记录。")

import pandas as pd

# 读取 CSV 文件

df = pd.read\_csv('weatherHistory.csv')

# 将 'Formatted Date' 列转换为 UTC datetime 类型

df['Formatted Date'] = pd.to\_datetime(df['Formatted Date'], utc=True)

# 将时间转换为 UTC+1

df['Formatted Date'] = df['Formatted Date'] + pd.Timedelta(hours=1)

# 将时间转换回带时区的字符串格式

df['Formatted Date'] = df['Formatted Date'].dt.strftime('%Y-%m-%d %H:%M:%S') + '+01:00'

# 去重 'Formatted Date' 列

df = df.drop\_duplicates(subset=['Formatted Date'])

# 保存修改后的数据到 CSV 文件

df.to\_csv('weatherHistory.csv', index=False)

print("处理完成，已保存为 'weatherHistory.csv'")

**//特征构建**

import pandas as pd

import numpy as np

# 读取 CSV 文件

df = pd.read\_csv("weatherHistory.csv")

# 将 Formatted Date 列转换为 datetime 类型

df['Formatted Date'] = pd.to\_datetime(df['Formatted Date'], utc=True)

# 提取小时信息

df['Hour'] = df['Formatted Date'].dt.hour

# 对小时进行分箱处理

hour\_bins = [0, 5, 11, 17, 23]  # 分箱区间

hour\_labels = ["凌晨", "早晨", "下午", "夜晚"]  # 分箱标签

df['TimeOfDay'] = pd.cut(df['Hour'], bins=hour\_bins, labels=hour\_labels, right=True, include\_lowest=True)

# 提取月份信息

df['Month'] = df['Formatted Date'].dt.month

# 定义季节分箱

def get\_season(month):

    if month in [12, 1, 2]:

        return '冬季'

    elif month in [3, 4, 5]:

        return '春季'

    elif month in [6, 7, 8]:

        return '夏季'

    elif month in [9, 10, 11]:

        return '秋季'

# 应用季节分箱

df['Season'] = df['Month'].apply(get\_season)

# 对风速进行分箱处理（假设分为：低、中、高）

wind\_speed\_bins = [0, 10, 20, df['Wind Speed (km/h)'].max()]  # 分箱区间

wind\_speed\_labels = ["低", "中", "高"]  # 分箱标签

df['WindSpeedGroup'] = pd.cut(df['Wind Speed (km/h)'], bins=wind\_speed\_bins, labels=wind\_speed\_labels, right=False)

# 对风向进行分箱处理（分为8个方向）

wind\_bearing\_bins = [0, 45, 90, 135, 180, 225, 270, 315, 360]  # 分箱区间

wind\_bearing\_labels = ["北", "东北", "东", "东南", "南", "西南", "西", "西北"]  # 分箱标签

df['WindDirection'] = pd.cut(df['Wind Bearing (degrees)'], bins=wind\_bearing\_bins, labels=wind\_bearing\_labels, right=False, include\_lowest=True)

# 查看分箱后的结果

df[['Month', 'Season', 'Hour', 'TimeOfDay', 'Wind Speed (km/h)', 'WindSpeedGroup', 'Wind Bearing (degrees)', 'WindDirection']].head()

# 进行特征交互

df\_interaction = df\_normalized.copy()

# 新增温度和体感温度的交互特征

df\_interaction['Temp\_AppTemp'] = df\_interaction['Temperature (C)'] \* df\_interaction['Apparent Temperature (C)']

# 新增温度和湿度的交互特征

df\_interaction['Temp\_Humidity'] = df\_interaction['Temperature (C)'] \* df\_interaction['Humidity']

# 新增体感温度和湿度的交互特征

df\_interaction['AppTemp\_Humidity'] = df\_interaction['Apparent Temperature (C)'] \* df\_interaction['Humidity']

# 查看特征交互后的数据

df\_interaction[['Temp\_AppTemp', 'Temp\_Humidity', 'AppTemp\_Humidity']].head()

**数据挖掘：**

**聚类**

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

import seaborn as sns

# 读取CSV文件

df = pd.read\_csv("weatherHistory.csv", encoding='ISO-8859-1')

# 选择聚类和分类的特征

features = df[['Temperature (C)', 'Humidity', 'Wind Speed (km/h)']]

# 对数据进行采样

sampled\_df = features.sample(frac=0.2, random\_state=42)  # 采样10%的数据

# 使用肘部法确定K值

inertia = []

K\_range = range(1, 11)

for k in K\_range:

    kmeans = KMeans(n\_clusters=k, random\_state=42, n\_init=10, max\_iter=300)

    kmeans.fit(sampled\_df)

    inertia.append(kmeans.inertia\_)

# 绘制肘部法图表

plt.figure(figsize=(8, 4))

plt.plot(K\_range, inertia, 'bx-')

plt.xticks(K\_range)

plt.xlabel('Number of clusters')

plt.ylabel('Inertia')

plt.title('Elbow Method for optimal K')

plt.grid(True)

plt.show()

# 应用K均值聚类

optimal\_k = 3  # 通过肘部法或其他方法确定K值

kmeans = KMeans(n\_clusters=optimal\_k, random\_state=42, n\_init=10, max\_iter=300)

clusters = kmeans.fit\_predict(sampled\_df)

# 添加聚类结果到采样数据框

sampled\_df['Cluster'] = clusters

# 可视化聚类结果

fig = plt.figure(figsize=(10, 6))

ax = fig.add\_subplot(111, projection='3d')

scatter = ax.scatter(sampled\_df['Temperature (C)'], sampled\_df['Humidity'], sampled\_df['Wind Speed (km/h)'], c=sampled\_df['Cluster'], cmap='viridis', alpha=0.5)

ax.set\_xlabel('Temperature (C)')

ax.set\_ylabel('Humidity')

ax.set\_zlabel('Wind Speed (km/h)')

ax.set\_title('3D Clusters of Weather Data')

legend1 = ax.legend(\*scatter.legend\_elements(), title="Clusters")

ax.add\_artist(legend1)

plt.show()

**分类：  
线性回归**import pandas as pd

import numpy as np

# 可视化库

import matplotlib.pyplot as plt

import seaborn as sns

# 机器学习库

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

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

# 假设数据保存在 'weatherHistory.csv' 文件中

data = pd.read\_csv('weatherHistory.csv')

# 将 'Formatted Date' 转换为日期时间格式

data['Formatted Date'] = pd.to\_datetime(data['Formatted Date'], utc=True)

# 提取时间特征

data['Year'] = data['Formatted Date'].dt.year

data['Month'] = data['Formatted Date'].dt.month

data['Day'] = data['Formatted Date'].dt.day

data['Hour'] = data['Formatted Date'].dt.hour

data['Weekday'] = data['Formatted Date'].dt.weekday

# 对 'Summary' 和 'Precip Type' 进行独热编码

data = pd.get\_dummies(data, columns=['Summary', 'Precip Type'], prefix=['Summary', 'PrecipType'])

# 删除原始的 'Formatted Date' 和 'Daily Summary' 列

data = data.drop(['Formatted Date', 'Daily Summary'], axis=1)

# 再次查看数据的信息

print(data.info())

# 定义目标变量 y

y = data['Temperature (C)']

# 删除目标变量，剩下的都是特征

X = data.drop('Temperature (C)', axis=1)

# 将数据集划分为训练集和测试集，比例为 80%:20%

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42)

# 初始化标准化器

scaler = StandardScaler()

# 识别需要缩放的数值型特征（排除独热编码的特征）

numeric\_features = ['Apparent Temperature (C)', 'Humidity', 'Wind Speed (km/h)',

                    'Wind Bearing (degrees)', 'Visibility (km)', 'Pressure (millibars)',

                    'Year', 'Month', 'Day', 'Hour', 'Weekday']

# 对训练集进行拟合和转换

X\_train[numeric\_features] = scaler.fit\_transform(X\_train[numeric\_features])

# 对测试集进行转换

X\_test[numeric\_features] = scaler.transform(X\_test[numeric\_features])

# 初始化线性回归模型

lr\_model = LinearRegression()

# 训练模型

lr\_model.fit(X\_train, y\_train)

# 对测试集进行预测

y\_pred\_lr = lr\_model.predict(X\_test)

# 定义评估函数

def evaluate\_model(y\_true, y\_pred, model\_name):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    r2 = r2\_score(y\_true, y\_pred)

    print(f'{model\_name} 模型评估结果：')

    print(f'MSE: {mse:.4f}')

    print(f'RMSE: {rmse:.4f}')

    print(f'MAE: {mae:.4f}')

    print(f'R² Score: {r2:.4f}\n')

# 评估线性回归模型

evaluate\_model(y\_test, y\_pred\_lr, '线性回归')

# 获取特征名称

feature\_names = X\_train.columns

# 获取模型系数

coefficients = lr\_model.coef\_

# 创建系数数据框

coeff\_df = pd.DataFrame({'Feature': feature\_names, 'Coefficient': coefficients})

# 按系数大小排序

coeff\_df = coeff\_df.sort\_values(by='Coefficient', ascending=False)

# 输出系数数据框

print(coeff\_df)

**随机森林回归模型**

# 导入必要的库

from sklearn.ensemble import RandomForestRegressor

# 初始化随机森林回归模型

rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42, n\_jobs=-1)

# 训练模型

rf\_model.fit(X\_train, y\_train)

# 对测试集进行预测

y\_pred\_rf = rf\_model.predict(X\_test)

# 评估函数

def evaluate\_model(y\_true, y\_pred, model\_name):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    r2 = r2\_score(y\_true, y\_pred)

    print(f'{model\_name} 模型评估结果：')

    print(f'MSE: {mse:.4f}')

    print(f'RMSE: {rmse:.4f}')

    print(f'MAE: {mae:.4f}')

    print(f'R² Score: {r2:.4f}\n')

# 评估随机森林模型

evaluate\_model(y\_test, y\_pred\_rf, '随机森林回归')

import matplotlib.pyplot as plt

import seaborn as sns

# 获取特征重要性

importances = rf\_model.feature\_importances\_

# 创建特征重要性数据框

feature\_importances = pd.DataFrame({

    'Feature': X\_train.columns,

    'Importance': importances

})

# 按重要性排序

feature\_importances = feature\_importances.sort\_values(by='Importance', ascending=False)

# 绘制特征重要性图

plt.figure(figsize=(12, 8))

sns.barplot(x='Importance', y='Feature', data=feature\_importances)

plt.title('特征重要性（随机森林）')

plt.tight\_layout()

plt.show()

# 计算相关系数

correlation = data['Temperature (C)'].corr(data['Apparent Temperature (C)'])

print(f"Temperature 与 Apparent Temperature 的相关系数: {correlation}")

# 6. 定义特征矩阵 X，移除 'Temperature (C)' 和 'Apparent Temperature (C)'

X = data.drop(['Temperature (C)', 'Apparent Temperature (C)'], axis=1)

# 确认特征移除是否成功

print("\n特征列表（移除 'Temperature (C)' 和 'Apparent Temperature (C)' 后）：")

print(X.columns.tolist())

import pandas as pd

import numpy as np

# 可视化库

import matplotlib.pyplot as plt

import seaborn as sns

# 机器学习库

from sklearn.model\_selection import train\_test\_split, GridSearchCV, cross\_val\_score

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LinearRegression

from sklearn.ensemble import RandomForestRegressor

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

# 5. 定义目标变量 y

y = data['Temperature (C)']

# 7. 数据集划分

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42)

# 8. 特征缩放

scaler = StandardScaler()

numeric\_features = ['Humidity', 'Wind Speed (km/h)',

                    'Wind Bearing (degrees)', 'Visibility (km)', 'Pressure (millibars)',

                    'Year', 'Month', 'Day', 'Hour', 'Weekday']

# 仅对数值型特征进行缩放

X\_train[numeric\_features] = scaler.fit\_transform(X\_train[numeric\_features])

X\_test[numeric\_features] = scaler.transform(X\_test[numeric\_features])

# 9. 训练随机森林回归模型

rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42, n\_jobs=-1)

rf\_model.fit(X\_train, y\_train)

# 10. 对测试集进行预测

y\_pred\_rf = rf\_model.predict(X\_test)

# 11. 评估函数

def evaluate\_model(y\_true, y\_pred, model\_name):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    r2 = r2\_score(y\_true, y\_pred)

    print(f'{model\_name} 模型评估结果：')

    print(f'MSE: {mse:.4f}')

    print(f'RMSE: {rmse:.4f}')

    print(f'MAE: {mae:.4f}')

    print(f'R² Score: {r2:.4f}\n')

# 12. 评估随机森林模型

evaluate\_model(y\_test, y\_pred\_rf, '随机森林回归（移除高相关特征后）')

# 13. 特征重要性分析

importances = rf\_model.feature\_importances\_

feature\_importances = pd.DataFrame({

    'Feature': X\_train.columns,

    'Importance': importances

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

# 绘制特征重要性图

plt.figure(figsize=(12, 8))

sns.barplot(x='Importance', y='Feature', data=feature\_importances)

plt.title('Feature Importance (Random Forest)')

plt.tight\_layout()

plt.show()

**交叉验证**

from sklearn.model\_selection import cross\_val\_score

scores = cross\_val\_score(rf\_model, X, y, cv=5, scoring="r2")

print(f"交叉验证 R² scores: {scores}")

print(f"平均 R²: {scores.mean():.4f}")

# 对训练集进行拟合和转换

X\_train[numeric\_features] = scaler.fit\_transform(X\_train[numeric\_features])

# 对测试集进行转换

X\_test[numeric\_features] = scaler.transform(X\_test[numeric\_features])

# 初始化线性回归模型

lr\_model = LinearRegression()

# 训练模型

lr\_model.fit(X\_train, y\_train)

# 对测试集进行预测

y\_pred\_lr = lr\_model.predict(X\_test)

# 定义评估函数

def evaluate\_model(y\_true, y\_pred, model\_name):

    mse = mean\_squared\_error(y\_true, y\_pred)

    rmse = np.sqrt(mse)

    mae = mean\_absolute\_error(y\_true, y\_pred)

    r2 = r2\_score(y\_true, y\_pred)

    print(f'{model\_name} （移除Apparent Temperature (C)之后）模型评估结果：')

    print(f'MSE: {mse:.4f}')

    print(f'RMSE: {rmse:.4f}')

    print(f'MAE: {mae:.4f}')

    print(f'R² Score: {r2:.4f}\n')

# 评估线性回归模型

evaluate\_model(y\_test, y\_pred\_lr, '线性回归')

# 获取特征名称

feature\_names = X\_train.columns

# 获取模型系数

coefficients = lr\_model.coef\_

# 创建系数数据框

coeff\_df = pd.DataFrame({'Feature': feature\_names, 'Coefficient': coefficients})

# 按系数大小排序

coeff\_df = coeff\_df.sort\_values(by='Coefficient', ascending=False)

# 输出系数数据框

print(coeff\_df)

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import matplotlib

from sklearn.model\_selection import train\_test\_split, GridSearchCV, cross\_val\_score

from sklearn.preprocessing import StandardScaler

from sklearn.ensemble import RandomForestRegressor

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

from sklearn.feature\_selection import SelectFromModel

evaluation\_data = {

    'Model Name': [

        'Linear Regression (Remove Apparent Temperature (C))',

        'Random Forest Regression (Remove Highly Correlated Features)',

        'Random Forest Regression (Include Apparent Temperature (C))',

        'Linear Regression (Include Apparent Temperature (C))'

    ],

    'MSE': [34.3205, 2.6677, 0.0024, 0.8961],

    'RMSE': [5.8584, 1.6333, 0.0494, 0.9466],

    'MAE': [4.7881, 1.1897, 0.0136, 0.7406],

    'R² Score': [0.6231, 0.9707, 1.0000, 0.9902]

}

# Create DataFrame

evaluation\_df = pd.DataFrame(evaluation\_data)

# Set Model Name as index

evaluation\_df.set\_index('Model Name', inplace=True)

# Display the table

print(evaluation\_df)

# Set Seaborn plot style

sns.set(style="whitegrid")

# Create four subplots

fig, axes = plt.subplots(2, 2, figsize=(18, 14))  # Increased figure size for better readability

axes = axes.flatten()

# Define metrics to plot

metrics = ['MSE', 'RMSE', 'MAE', 'R² Score']

for i, metric in enumerate(metrics):

    sns.barplot(

        x=evaluation\_df.index,

        y=evaluation\_df[metric],

        ax=axes[i]

        # palette='Set2'  # Removed to prevent FutureWarning

    )

    axes[i].set\_title(f'{metric} Comparison', fontsize=18)

    axes[i].set\_ylabel(metric, fontsize=14)

    axes[i].set\_xlabel('Model Name', fontsize=14)

    for label in axes[i].get\_xticklabels():

        label.set\_rotation(45)  # Rotate x-axis labels to prevent overlap

plt.tight\_layout()

plt.show()

**已知多变量对离散单变量预测**

import pandas as pd

import numpy as np

# 可视化库

import matplotlib.pyplot as plt

import seaborn as sns

# 机器学习库

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.preprocessing import StandardScaler, LabelEncoder

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score, classification\_report

# 设置中文字体（如果需要显示中文）

plt.rcParams['font.sans-serif'] = ['SimHei']

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

# 1. 加载数据

data = pd.read\_csv('weatherHistory.csv')

# 2. 数据预处理

# 转换日期时间格式

data['Formatted Date'] = pd.to\_datetime(data['Formatted Date'], utc=True)

# 提取时间特征

data['Year'] = data['Formatted Date'].dt.year

data['Month'] = data['Formatted Date'].dt.month

data['Day'] = data['Formatted Date'].dt.day

data['Hour'] = data['Formatted Date'].dt.hour

data['Weekday'] = data['Formatted Date'].dt.weekday

# 删除原始日期列和 'Daily Summary'

data = data.drop(['Formatted Date', 'Daily Summary'], axis=1)

# 处理目标变量 'Summary'

label\_encoder = LabelEncoder()

data['Summary\_encoded'] = label\_encoder.fit\_transform(data['Summary'])

# 处理 'Precip Type' 的缺失值

data['Precip Type'] = data['Precip Type'].fillna('unknown')

# 独热编码 'Precip Type'

data = pd.get\_dummies(data, columns=['Precip Type'], prefix='PrecipType')

# 删除高度相关的特征，避免数据泄露

data = data.drop(['Temperature (C)', 'Apparent Temperature (C)'], axis=1)

# 删除原始的 'Summary' 列

data = data.drop(['Summary'], axis=1)

# 3. 数据集划分

X = data.drop(['Summary\_encoded'], axis=1)

y = data['Summary\_encoded']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=42)

# 4. 特征缩放

numeric\_features = ['Humidity', 'Wind Speed (km/h)', 'Wind Bearing (degrees)',

                    'Visibility (km)', 'Pressure (millibars)', 'Year', 'Month',

                    'Day', 'Hour', 'Weekday']

scaler = StandardScaler()

X\_train[numeric\_features] = scaler.fit\_transform(X\_train[numeric\_features])

X\_test[numeric\_features] = scaler.transform(X\_test[numeric\_features])

# 5. 模型训练

rf\_classifier = RandomForestClassifier(n\_estimators=100, random\_state=42, n\_jobs=-1)

rf\_classifier.fit(X\_train, y\_train)

# 6. 模型评估

y\_pred = rf\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'模型准确率：{accuracy:.4f}')

print("\n分类报告：")

# 修改部分：获取实际存在的标签和对应的类别名称

import numpy as np

# 获取预测和真实值中的所有标签

labels = np.unique(np.concatenate((y\_test, y\_pred)))

target\_names = label\_encoder.inverse\_transform(labels)

# 生成分类报告

print(classification\_report(y\_test, y\_pred, labels=labels, target\_names=target\_names))

# 7. 特征重要性分析

importances = rf\_classifier.feature\_importances\_

feature\_names = X\_train.columns

feature\_importances = pd.DataFrame({'Feature': feature\_names, 'Importance': importances})

feature\_importances = feature\_importances.sort\_values(by='Importance', ascending=False)

plt.figure(figsize=(12, 8))

sns.barplot(x='Importance', y='Feature', data=feature\_importances)

plt.title('特征重要性（随机森林分类器）')

plt.tight\_layout()

plt.show()

**优化后**

from sklearn.metrics import roc\_curve, auc

from sklearn.preprocessing import label\_binarize

import matplotlib.pyplot as plt

# 初始模型

classes = rf\_classifier.classes\_

n\_classes = len(classes)

y\_test\_binarized = label\_binarize(y\_test, classes=classes)

y\_score = rf\_classifier.predict\_proba(X\_test)

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(n\_classes):

    fpr[i], tpr[i], \_ = roc\_curve(y\_test\_binarized[:, i], y\_score[:, i])

    roc\_auc[i] = auc(fpr[i], tpr[i])

fpr["micro"], tpr["micro"], \_ = roc\_curve(y\_test\_binarized.ravel(), y\_score.ravel())

roc\_auc["micro"] = auc(fpr["micro"], tpr["micro"])

# 优化后模型

classes\_best = best\_rf\_classifier.classes\_

n\_classes\_best = len(classes\_best)

y\_test\_binarized\_best = label\_binarize(y\_test, classes=classes\_best)

y\_score\_best = best\_rf\_classifier.predict\_proba(X\_test)

fpr\_best = dict()

tpr\_best = dict()

roc\_auc\_best = dict()

for i in range(n\_classes\_best):

    fpr\_best[i], tpr\_best[i], \_ = roc\_curve(y\_test\_binarized\_best[:, i], y\_score\_best[:, i])

    roc\_auc\_best[i] = auc(fpr\_best[i], tpr\_best[i])

fpr\_best["micro"], tpr\_best["micro"], \_ = roc\_curve(y\_test\_binarized\_best.ravel(), y\_score\_best.ravel())

roc\_auc\_best["micro"] = auc(fpr\_best["micro"], tpr\_best["micro"])

# 绘制 ROC 曲线

plt.figure(figsize=(10, 8))

plt.plot(fpr["micro"], tpr["micro"],

         label='初始模型 ROC 曲线 (AUC = {0:0.4f})'.format(roc\_auc["micro"]),

         color='navy', linestyle=':', linewidth=4)

plt.plot(fpr\_best["micro"], tpr\_best["micro"],

         label='优化后模型 ROC 曲线 (AUC = {0:0.4f})'.format(roc\_auc\_best["micro"]),

         color='darkorange', linestyle=':', linewidth=4)

plt.plot([0, 1], [0, 1], 'k--', linewidth=2)

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('假正率 (False Positive Rate)')

plt.ylabel('真正率 (True Positive Rate)')

plt.title('初始模型和优化后模型的 ROC 曲线对比')

plt.legend(loc="lower right")

plt.show()

**时间序列上的预测**

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

# 数据预处理

data = pd.read\_csv("weatherHistory.csv")

data["Formatted Date"] = pd.to\_datetime(data["Formatted Date"])

data.set\_index("Formatted Date", inplace=True)

data["Year"] = data.index.year

data["Month"] = data.index.month

data["Day"] = data.index.day

data["Hour"] = data.index.hour

data = pd.get\_dummies(data, columns=["Summary", "Precip Type"])

# \*\*删除非数值型列\*\*

data.drop(columns=['Daily Summary'], inplace=True)

# 数据集划分

train = data.loc[:"2016-12-31"]

test = data.loc["2017-01-01":]

# 特征缩放

scaler = StandardScaler()

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

test\_scaled = scaler.transform(test)

**ARIMA模型**

import pandas as pd

import numpy as np

from statsmodels.tsa.arima.model import ARIMA

from sklearn.metrics import mean\_squared\_error

import matplotlib.pyplot as plt

data = pd.read\_csv("weatherHistory.csv")

data["Formatted Date"] = pd.to\_datetime(data["Formatted Date"])

data.set\_index("Formatted Date", inplace=True)

# 设置时间索引的频率

data = data.asfreq('H')

data["Year"] = data.index.year

data["Month"] = data.index.month

data["Day"] = data.index.day

data["Hour"] = data.index.hour

data = pd.get\_dummies(data, columns=["Summary", "Precip Type"])

data.drop(columns=['Daily Summary'], inplace=True)

# 数据集划分（调整划分方式以确保测试集有足够的数据）

split\_point = int(len(data) \* 0.8)

train = data.iloc[:split\_point]

test = data.iloc[split\_point:]

# 提取温度数据

train\_temperature = train['Temperature (C)']

test\_temperature = test['Temperature (C)']

# 检查测试集长度

print(f"Length of test\_temperature: {len(test\_temperature)}")

# 建立ARIMA模型

model = ARIMA(train\_temperature, order=(5,1,0))

model\_fit = model.fit()

# 预测未来24小时

forecast = model\_fit.forecast(steps=24)

# 获取真实值和预测值

y\_true = test\_temperature.iloc[:24].values

y\_pred = forecast.values

# 检查形状

print(f"Shape of y\_true: {y\_true.shape}")

print(f"Shape of y\_pred: {y\_pred.shape}")

# 计算均方误差

mse = mean\_squared\_error(y\_true, y\_pred)

rmse = np.sqrt(mse)

print(f'ARIMA model RMSE: {rmse}')

# 绘制预测结果

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

plt.plot(test\_temperature.index[:24], y\_true, label='Actual Temperature')

plt.plot(test\_temperature.index[:24], y\_pred, label='Predicted Temperature', color='red')

plt.xlabel('Time')

plt.ylabel('Temperature (°C)')

plt.title('ARIMA Model - 24 Hours Temperature Forecast')

plt.legend()

plt.show()

**SARIMA模型**

from statsmodels.tsa.statespace.sarimax import SARIMAX

# 定义季节性周期长度，比如一天24小时

season\_length = 24

# 建立SARIMA模型

model = SARIMAX(train\_temperature, order=(5, 1, 0), seasonal\_order=(1, 1, 1, season\_length))

model\_fit = model.fit()

# 预测未来24小时

forecast = model\_fit.forecast(steps=24)

# 获取真实值和预测值

y\_true = test\_temperature.iloc[:24].values

y\_pred = forecast.values

# 计算均方误差

mse = mean\_squared\_error(y\_true, y\_pred)

rmse = np.sqrt(mse)

print(f'SARIMA model RMSE: {rmse}')

# 绘制预测结果

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

plt.plot(test\_temperature.index[:24], y\_true, label='Actual Temperature')

plt.plot(test\_temperature.index[:24], y\_pred, label='Predicted Temperature', color='red')

plt.xlabel('Time')

plt.ylabel('Temperature (°C)')

plt.title('SARIMA Model - 24 Hours Temperature Forecast')

plt.legend()

plt.show()

**Prophet模型**

# 导入必要的库

import pandas as pd

from prophet import Prophet

import matplotlib.pyplot as plt

from sklearn.metrics import mean\_squared\_error

import numpy as np

# 然后在导入 Prophet 之前，设置 Prophet 使用 cmdstanpy 后端：

from prophet import Prophet

import cmdstanpy

# 安装 CmdStan（默认会进行编译）

cmdstanpy.install\_cmdstan()

# 使用 Prophet 模型

model = Prophet(stan\_backend="CMDSTANPY")

# 数据预处理

data = pd.read\_csv("weatherHistory.csv", encoding='utf-8')

# 格式化日期

data["Formatted Date"] = pd.to\_datetime(data["Formatted Date"])

# Prophet 要求数据有特定格式，列名必须是 'ds' 和 'y'

df\_prophet = data[['Formatted Date', 'Temperature (C)']].rename(columns={'Formatted Date': 'ds', 'Temperature (C)': 'y'})

# 将训练集划分为训练数据和测试数据

train = df\_prophet.loc[df\_prophet['ds'] <= '2016-12-31']

test = df\_prophet.loc[df\_prophet['ds'] > '2016-12-31']

# 创建并拟合 Prophet 模型

model = Prophet()

model.fit(train)

# 进行未来24小时的预测

future = model.make\_future\_dataframe(periods=24, freq='H')

forecast = model.predict(future)

# 获取测试集的实际温度值

y\_true = test['y'].iloc[:24].values

# 获取预测的温度值

y\_pred = forecast['yhat'].iloc[-24:].values

# 计算均方误差 (MSE) 和根均方误差 (RMSE)

mse = mean\_squared\_error(y\_true, y\_pred)

rmse = np.sqrt(mse)

print(f'Prophet模型的RMSE: {rmse}')

# 可视化预测结果

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

plt.plot(test['ds'].iloc[:24], y\_true, label='实际温度')

plt.plot(test['ds'].iloc[:24], y\_pred, label='预测温度', color='red')

plt.xlabel('时间')

plt.ylabel('温度 (°C)')

plt.title('Prophet 模型 - 24 小时温度预测')

plt.legend()

plt.show()

**LSTM**

# 导入必要的库

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense

from sklearn.metrics import mean\_squared\_error

# 1. 数据预处理

data = pd.read\_csv("weatherHistory.csv")

data["Formatted Date"] = pd.to\_datetime(data["Formatted Date"])

data.set\_index("Formatted Date", inplace=True)

# 选择需要的特征和目标变量，假设温度为预测目标

data = data[['Temperature (C)']]

# 缺失值处理

data.dropna(inplace=True)

# 2. 标准化数据

scaler = StandardScaler()

scaled\_data = scaler.fit\_transform(data)

# 3. 创建监督学习格式的数据集

def create\_dataset(data, time\_step=24):

    X, y = [], []

    for i in range(len(data)-time\_step-1):

        X.append(data[i:(i+time\_step), 0])

        y.append(data[i + time\_step, 0])

    return np.array(X), np.array(y)

# 定义时间步长（比如 24小时作为一个时间步长）

time\_step = 24

# 划分训练集和测试集

train\_size = int(len(scaled\_data) \* 0.8)

train\_data = scaled\_data[:train\_size]

test\_data = scaled\_data[train\_size:]

# 构造 LSTM 所需的数据集格式

X\_train, y\_train = create\_dataset(train\_data, time\_step)

X\_test, y\_test = create\_dataset(test\_data, time\_step)

# 调整数据为 LSTM 需要的三维格式 [样本数, 时间步数, 特征数]

X\_train = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1)

X\_test = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)

# 4. 构建 LSTM 模型，增加层数和神经元数量

model = Sequential()

model.add(LSTM(100, return\_sequences=True, input\_shape=(time\_step, 1)))  # 第一层 LSTM

model.add(LSTM(100, return\_sequences=False))  # 第二层 LSTM

model.add(Dense(50))  # 全连接层

model.add(Dense(1))   # 输出层

# 编译模型

model.compile(optimizer='adam', loss='mean\_squared\_error')

# 5. 增加训练轮数，重新训练模型

model.fit(X\_train, y\_train, batch\_size=64, epochs=3)  # 增加训练轮数

# 6. 模型预测

train\_predict = model.predict(X\_train)

test\_predict = model.predict(X\_test)

# 反标准化预测结果和真实数据

train\_predict = scaler.inverse\_transform(train\_predict)

y\_train\_actual = scaler.inverse\_transform([y\_train])

test\_predict = scaler.inverse\_transform(test\_predict)

y\_test\_actual = scaler.inverse\_transform([y\_test])

# 计算 RMSE

train\_rmse = np.sqrt(mean\_squared\_error(y\_train\_actual[0], train\_predict[:, 0]))

test\_rmse = np.sqrt(mean\_squared\_error(y\_test\_actual[0], test\_predict[:, 0]))

print(f"训练集 RMSE: {train\_rmse}")

print(f"测试集 RMSE: {test\_rmse}")

# 7. 可视化结果 - 不同时间段的展示

# 选择测试集中的某一部分数据进行可视化

# 假设一天有24个时间点，一个月大约是720个时间点（24 \* 30），一周是168个时间点（24 \* 7）

# 定义各个时间段的长度

day\_length = 24

week\_length = 24 \* 7

month\_length = 24 \* 30

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

# 可视化一个月的数据

plt.subplot(3, 1, 1)

plt.plot(y\_test\_actual[0][:month\_length], label='实际值（一个月）', linewidth=1)

plt.plot(test\_predict[:, 0][:month\_length], label='预测值（一个月）', color='red', linewidth=1, alpha=0.6)

plt.title('一个月的预测 - LSTM 预测', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

# 可视化一周的数据

plt.subplot(3, 1, 2)

plt.plot(y\_test\_actual[0][:week\_length], label='实际值（一周）', linewidth=1)

plt.plot(test\_predict[:, 0][:week\_length], label='预测值（一周）', color='red', linewidth=1, alpha=0.6)

plt.title('一周的预测 - LSTM 预测', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

# 可视化一天的数据

plt.subplot(3, 1, 3)

plt.plot(y\_test\_actual[0][:day\_length], label='实际值（一天）', linewidth=1)

plt.plot(test\_predict[:, 0][:day\_length], label='预测值（一天）', color='red', linewidth=1, alpha=0.6)

plt.title('一天的预测 - LSTM 预测', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

plt.tight\_layout()

plt.show()

**滚动预测**

# 定义滚动预测函数

def recursive\_forecast(model, input\_data, steps):

    predictions = []

    current\_input = input\_data

    for \_ in range(steps):

        # 预测下一个时间步

        pred = model.predict(current\_input, verbose=0)

        # 将预测值保存

        predictions.append(pred[0, 0])

        # 使用当前预测值更新输入

        current\_input = np.roll(current\_input, -1)

        current\_input[0, -1, 0] = pred  # 将最后一个时间步更新为预测值

    return np.array(predictions)

# 定义各个时间段的长度

day\_length = 24  # 一天的步数

week\_length = 24 \* 7  # 一周的步数

month\_length = 24 \* 30  # 一个月的步数

# 初始输入为测试集中某一段

start\_index = time\_step  # 确保从第一个有效输入开始

initial\_input = X\_test[start\_index].reshape(1, time\_step, 1)

# 真实值应该从 start\_index 对应的点开始

y\_test\_actual\_segment = y\_test[start\_index:]

# 预测并展示一个月的数据

month\_predict = recursive\_forecast(model, initial\_input, month\_length)

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

plt.subplot(3, 1, 1)

plt.plot(y\_test\_actual\_segment[:month\_length], label='实际值（一个月）', linewidth=1)

plt.plot(month\_predict, label='预测值（一个月）', color='red', linewidth=1, alpha=0.6)

plt.title('一个月的滚动预测 - LSTM', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

# 预测并展示一周的数据

week\_predict = recursive\_forecast(model, initial\_input, week\_length)

plt.subplot(3, 1, 2)

plt.plot(y\_test\_actual\_segment[:week\_length], label='实际值（一周）', linewidth=1)

plt.plot(week\_predict, label='预测值（一周）', color='red', linewidth=1, alpha=0.6)

plt.title('一周的滚动预测 - LSTM', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

# 预测并展示一天的数据

day\_predict = recursive\_forecast(model, initial\_input, day\_length)

plt.subplot(3, 1, 3)

plt.plot(y\_test\_actual\_segment[:day\_length], label='实际值（一天）', linewidth=1)

plt.plot(day\_predict, label='预测值（一天）', color='red', linewidth=1, alpha=0.6)

plt.title('一天的滚动预测 - LSTM', fontproperties='SimHei')

plt.legend(prop={'family': 'SimHei'})

plt.tight\_layout()

plt.show()

**优化后**

import numpy as np

from sklearn.metrics import mean\_squared\_error

from tensorflow.keras.models import load\_model

from concurrent.futures import ProcessPoolExecutor

import multiprocessing

import matplotlib.pyplot as plt

import sys

# 配置 Matplotlib 以支持中文字体

plt.rcParams["font.sans-serif"] = ["SimHei"]  # 使用 SimHei 字体显示中文

plt.rcParams["axes.unicode\_minus"] = False  # 正确显示负号

# 定义滚动预测函数

def recursive\_forecast(model\_path, input\_data, steps):

    """

    使用保存的模型进行滚动预测。

    """

    # 加载模型

    model = load\_model(model\_path)

    predictions = []

    current\_input = input\_data.copy()

    for \_ in range(steps):

        # 预测下一个时间步

        pred = model.predict(current\_input, verbose=0)

        # 将预测值保存

        predictions.append(pred[0, 0])

        # 使用当前预测值更新输入

        current\_input = np.roll(current\_input, -1)

        current\_input[0, -1, 0] = pred

    return np.array(predictions)

# 定义处理每个起始点的函数

def process\_starting\_point(args):

    """

    处理单个起始点的预测任务。

    """

    (model\_path, start\_index, X\_test, y\_test, time\_step, day\_length, week\_length) = args

    initial\_input = X\_test[start\_index].reshape(1, time\_step, 1)

    y\_test\_actual\_segment = y\_test[start\_index:]

    # 预测一天的数据

    day\_predict = recursive\_forecast(model\_path, initial\_input, day\_length)

    day\_actual = y\_test\_actual\_segment[:day\_length]

    day\_error = mean\_squared\_error(day\_actual, day\_predict)

    # 预测一周的数据

    week\_predict = recursive\_forecast(model\_path, initial\_input, week\_length)

    week\_actual = y\_test\_actual\_segment[:week\_length]

    week\_error = mean\_squared\_error(week\_actual, week\_predict)

    return {

        "start\_index": start\_index,

        "day\_error": day\_error,

        "day\_predict": day\_predict,

        "day\_actual": day\_actual,

        "week\_error": week\_error,

        "week\_predict": week\_predict,

        "week\_actual": week\_actual,

    }

# 定义选择多个起始索引的函数

def select\_start\_indices(total\_length, N, time\_step, max\_steps):

    """

    选择多个起始索引，确保每个起始点后有足够的步数进行预测。

    """

    available\_length = total\_length - time\_step - max\_steps

    if available\_length <= 0:

        raise ValueError("测试集长度不足以进行所需的预测步数。")

    start\_indices = np.linspace(time\_step, available\_length, N).astype(int)

    return start\_indices

# 定义多进程预测函数

def run\_multiprocessing(

    model\_path, start\_indices, X\_test, y\_test, time\_step, day\_length, week\_length

):

    """

    使用多进程进行预测，并返回所有结果。

    """

    # 准备参数列表

    args\_list = [

        (model\_path, idx, X\_test, y\_test, time\_step, day\_length, week\_length)

        for idx in start\_indices

    ]

    # 获取 CPU 核心数量

    cpu\_count = multiprocessing.cpu\_count()

    print(f"使用 {cpu\_count} 个进程进行预测...")

    # 使用 ProcessPoolExecutor 进行多进程预测

    with ProcessPoolExecutor(max\_workers=cpu\_count) as executor:

        results = list(executor.map(process\_starting\_point, args\_list))

    return results

# 主函数

def main():

    import pandas as pd

    from sklearn.preprocessing import StandardScaler

    from tensorflow.keras.models import Sequential

    from tensorflow.keras.layers import LSTM, Dense

    # 数据预处理和模型训练部分（假设之前已经完成并保存了模型）

    # 如果需要重新训练模型，可以在这里添加代码

    # 加载测试数据（假设在运行该脚本前已经处理好）

    # 这里假设 X\_test 和 y\_test 已经保存为 .npy 文件

    # 你需要根据实际情况调整数据加载方式

    # 示例：加载已处理好的测试数据

    X\_test = np.load("X\_test.npy")

    y\_test = np.load("y\_test.npy")

    time\_step = 24

    day\_length = 24

    week\_length = 24 \* 7

    N = 100  # 选择100个起始点

    # 选择多个起始索引

    start\_indices = select\_start\_indices(len(X\_test), N, time\_step, week\_length)

    print(f"选择的起始索引: {start\_indices}")

    # 模型保存路径

    model\_path = "lstm\_model.h5"

    # 运行多进程预测

    results = run\_multiprocessing(

        model\_path, start\_indices, X\_test, y\_test, time\_step, day\_length, week\_length

    )

    print("多进程预测完成！")

    # 从结果中选取一天和一周预测误差最小的

    best\_day\_result = min(results, key=lambda x: x["day\_error"])

    best\_week\_result = min(results, key=lambda x: x["week\_error"])

    print(

        f"最佳一天预测起始索引: {best\_day\_result['start\_index']} - RMSE: {best\_day\_result['day\_error']:.4f}"

    )

    print(

        f"最佳一周预测起始索引: {best\_week\_result['start\_index']} - RMSE: {best\_week\_result['week\_error']:.4f}"

    )

    # 可视化最佳预测结果

    plt.figure(figsize=(14, 10))

    # 可视化最佳一天的预测结果

    plt.subplot(2, 1, 1)

    plt.plot(best\_day\_result["day\_actual"], label="实际值（一天）", linewidth=1)

    plt.plot(

        best\_day\_result["day\_predict"],

        label="预测值（一天）",

        color="red",

        linewidth=1,

        alpha=0.6,

    )

    plt.title(

        f'一天的滚动预测 - 最佳结果 (起始索引: {best\_day\_result["start\_index"]}) - RMSE: {best\_day\_result["day\_error"]:.4f}'

    )

    plt.legend(prop={"family": "SimHei"})

    # 可视化最佳一周的预测结果

    plt.subplot(2, 1, 2)

    plt.plot(best\_week\_result["week\_actual"], label="实际值（一周）", linewidth=1)

    plt.plot(

        best\_week\_result["week\_predict"],

        label="预测值（一周）",

        color="red",

        linewidth=1,

        alpha=0.6,

    )

    plt.title(

        f'一周的滚动预测 - 最佳结果 (起始索引: {best\_week\_result["start\_index"]}) - RMSE: {best\_week\_result["week\_error"]:.4f}'

    )

    plt.legend(prop={"family": "SimHei"})

    plt.tight\_layout()

    plt.show()

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

    main()

**时间长度与预测误差方差**

# predict\_with\_variance\_plot.py

import numpy as np

from sklearn.metrics import mean\_squared\_error

from tensorflow.keras.models import load\_model

from concurrent.futures import ProcessPoolExecutor

import multiprocessing

import matplotlib.pyplot as plt

import sys

import random

# 配置 Matplotlib 以支持中文字体

plt.rcParams["font.sans-serif"] = ["SimHei"]  # 使用 SimHei 字体显示中文

plt.rcParams["axes.unicode\_minus"] = False  # 正确显示负号

# 定义滚动预测函数

def recursive\_forecast(model\_path, input\_data, steps):

    """

    使用保存的模型进行滚动预测。

    """

    # 加载模型

    model = load\_model(model\_path)

    predictions = []

    current\_input = input\_data.copy()

    for \_ in range(steps):

        # 预测下一个时间步

        pred = model.predict(current\_input, verbose=0)

        # 将预测值保存

        predictions.append(pred[0, 0])

        # 使用当前预测值更新输入

        current\_input = np.roll(current\_input, -1)

        current\_input[0, -1, 0] = pred

    return np.array(predictions)

# 定义处理每个起始点和预测长度的函数

def process\_forecast(args):

    """

    处理单个起始点和预测长度的预测任务。

    """

    (model\_path, start\_index, X\_test, y\_test, time\_step, pred\_length) = args

    initial\_input = X\_test[start\_index].reshape(1, time\_step, 1)

    y\_test\_actual\_segment = y\_test[start\_index : start\_index + pred\_length]

    if len(y\_test\_actual\_segment) < pred\_length:

        # 如果实际数据不足，返回高误差

        return {"pred\_length": pred\_length, "mse": float("inf")}

    # 进行递归预测

    pred = recursive\_forecast(model\_path, initial\_input, pred\_length)

    # 计算均方误差

    mse = mean\_squared\_error(y\_test\_actual\_segment, pred)

    return {"pred\_length": pred\_length, "mse": mse}

# 定义选择多个随机起始索引的函数

def select\_random\_start\_indices(total\_length, N, time\_step, max\_steps):

    """

    随机选择多个起始索引，确保每个起始点后有足够的步数进行预测。

    """

    available\_length = total\_length - time\_step - max\_steps

    if available\_length <= 0:

        raise ValueError("测试集长度不足以进行所需的预测步数。")

    # 随机选择 N 个起始点

    start\_indices = random.sample(range(time\_step, available\_length), N)

    return start\_indices

# 定义多进程预测函数

def run\_multiprocessing\_forecast(

    model\_path, start\_indices, X\_test, y\_test, time\_step, pred\_length

):

    """

    使用多进程进行预测，并返回所有结果。

    """

    # 准备参数列表

    args\_list = [

        (model\_path, idx, X\_test, y\_test, time\_step, pred\_length)

        for idx in start\_indices

    ]

    # 获取 CPU 核心数量

    cpu\_count = multiprocessing.cpu\_count()

    print(f"使用 {cpu\_count} 个进程进行预测...")

    # 使用 ProcessPoolExecutor 进行多进程预测

    with ProcessPoolExecutor(max\_workers=cpu\_count) as executor:

        results = list(executor.map(process\_forecast, args\_list))

    return results

# 定义计算不同预测长度下误差的函数

def compute\_variance\_over\_time(

    model\_path, X\_test, y\_test, time\_step, N, max\_pred\_length

):

    """

    计算不同预测长度下预测误差的均值。

    """

    # 随机选择 N 个起始点

    start\_indices = select\_random\_start\_indices(

        len(X\_test), N, time\_step, max\_pred\_length

    )

    print(f"随机选择的起始索引: {start\_indices}")

    # 获取所有预测长度

    pred\_lengths = list(range(1, max\_pred\_length + 1))  # 1小时到72小时

    variance\_means = []

    for pred\_length in pred\_lengths:

        print(f"计算预测长度: {pred\_length} 小时")

        results = run\_multiprocessing\_forecast(

            model\_path, start\_indices, X\_test, y\_test, time\_step, pred\_length

        )

        # 过滤掉因数据不足返回的无穷大误差

        valid\_results = [res["mse"] for res in results if res["mse"] != float("inf")]

        if valid\_results:

            mean\_mse = np.mean(valid\_results)

        else:

            mean\_mse = None  # 或者设为 np.nan

        variance\_means.append(mean\_mse)

        print(f"预测长度 {pred\_length} 小时的平均MSE: {mean\_mse}")

    return pred\_lengths, variance\_means

# 主函数

def main():

    import pandas as pd

    from sklearn.preprocessing import StandardScaler

    from tensorflow.keras.models import Sequential

    from tensorflow.keras.layers import LSTM, Dense

    # 加载测试数据

    X\_test = np.load("X\_test.npy")

    y\_test = np.load("y\_test.npy")

    time\_step = 24  # 假设时间步为24小时

    N = 20  # 随机选择20个起始点

    max\_pred\_length = 48  # 最大预测长度为48小时（2天）

    # 模型保存路径

    model\_path = "lstm\_model.h5"

    # 计算不同预测长度下的平均MSE

    pred\_lengths, variance\_means = compute\_variance\_over\_time(

        model\_path, X\_test, y\_test, time\_step, N, max\_pred\_length

    )

    print("不同预测长度下的平均MSE计算完成！")

    # 绘制预测长度与平均MSE的关系图

    plt.figure(figsize=(12, 6))

    plt.plot(pred\_lengths, variance\_means, marker="o", linestyle="-", color="b")

    plt.title("预测时间长度与预测误差方差均值关系图")

    plt.xlabel("预测时间长度（小时）")

    plt.ylabel("预测误差均值（MSE）")

    plt.grid(True)

    plt.tight\_layout()

    plt.show()

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

    main()

**特征重要性**

# SHAP 分析脚本

# 1. 导入必要的库

import pandas as pd

import numpy as np

from sklearn.preprocessing import StandardScaler

import shap

from tensorflow.keras.models import load\_model

import matplotlib.pyplot as plt

import tensorflow as tf

# 2. 数据预处理

# 读取数据

data = pd.read\_csv("weatherHistory.csv")

# 解析日期并设置为索引

data["Formatted Date"] = pd.to\_datetime(data["Formatted Date"])

data.set\_index("Formatted Date", inplace=True)

# 选择温度作为目标变量

data = data[['Temperature (C)']]

# 处理缺失值

data.dropna(inplace=True)

# 标准化数据

scaler = StandardScaler()

scaled\_data = scaler.fit\_transform(data)

# 创建监督学习格式的数据集

def create\_dataset(data, time\_step=24):

    X, y = [], []

    for i in range(len(data) - time\_step - 1):

        X.append(data[i:(i + time\_step), 0])

        y.append(data[i + time\_step, 0])

    return np.array(X), np.array(y)

# 定义时间步长（24小时）

time\_step = 24

# 划分训练集和测试集

train\_size = int(len(scaled\_data) \* 0.8)

train\_data = scaled\_data[:train\_size]

test\_data = scaled\_data[train\_size:]

# 构造 LSTM 所需的数据集格式

X\_train, y\_train = create\_dataset(train\_data, time\_step)

X\_test, y\_test = create\_dataset(test\_data, time\_step)

# 调整数据为 LSTM 需要的三维格式 [样本数, 时间步数, 特征数]

X\_train = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], 1)

X\_test = X\_test.reshape(X\_test.shape[0], X\_test.shape[1], 1)

# 3. 加载已训练的 LSTM 模型

model\_path = 'lstm\_model.h5'

model = load\_model(model\_path)

print(f"已加载模型：{model\_path}")

# 4. SHAP 分析

# 4.1. 检查 TensorFlow 版本

print(f"TensorFlow 版本: {tf.\_\_version\_\_}")

# 4.2. 准备背景数据

# 选择训练集的一部分作为背景数据

background = X\_train[:100]  # 根据内存和计算资源调整样本数量

# 4.3. 创建 SHAP 解释器

# 由于 DeepExplainer 在 TensorFlow 2.17.0 下存在兼容性问题，使用 KernelExplainer

def model\_predict(X):

    """

    定义一个预测函数，将 KernelExplainer 的输入（二维）转换为模型需要的三维格式

    """

    X = X.reshape(X.shape[0], time\_step, 1)

    return model.predict(X).flatten()

# 使用 KernelExplainer 进行解释

explainer = shap.KernelExplainer(model\_predict, background.reshape(background.shape[0], time\_step))

print("已创建 SHAP KernelExplainer 解释器")

# 4.4. 选择要解释的样本

X\_sample = X\_test[:3]  # 根据需要调整样本数量

print(f"选择了 {X\_sample.shape[0]} 个样本进行 SHAP 分析")

# 4.5. 计算 SHAP 值

print("正在计算 SHAP 值，请耐心等待（KernelExplainer 计算较慢）...")

shap\_values = explainer.shap\_values(X\_sample.reshape(X\_sample.shape[0], time\_step))

print("已计算 SHAP 值")

# 4.6. 可视化 SHAP 结果

# 将数据重新塑形为 [样本数, 时间步数]

X\_sample\_reshaped = X\_sample.reshape(X\_sample.shape[0], time\_step)

# 4.6.1. 可视化总体 SHAP 概述

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

shap.summary\_plot(shap\_values, X\_sample\_reshaped, feature\_names=[f'Time Step {i+1}' for i in range(time\_step)])

plt.title('SHAP Summary Plot')

plt.show()

# 4.6.2. 可视化单个样本的 SHAP 值（力导向图）

sample\_index = 0  # 选择第一个样本

plt.figure(figsize=(10, 5))

shap.force\_plot(

    explainer.expected\_value,

    shap\_values[sample\_index],

    X\_sample\_reshaped[sample\_index],

    feature\_names=[f'Time Step {i+1}' for i in range(time\_step)],

    matplotlib=True

)

plt.title(f'SHAP Force Plot for Sample {sample\_index + 1}')

plt.show()

# 4.6.3. 可视化特定时间步的 SHAP 值分布

time\_step\_index = 11  # 选择第 12 个时间步（0-based index）

shap\_values\_time\_step = shap\_values[:, time\_step\_index]

plt.figure(figsize=(8, 4))

plt.hist(shap\_values\_time\_step, bins=30, edgecolor='k', alpha=0.7)

plt.title(f'SHAP Values Distribution for Time Step {time\_step\_index + 1}')

plt.xlabel('SHAP Value')

plt.ylabel('Frequency')

plt.show()

**关联规则挖掘**

import pandas as pd

# 读取数据

data = pd.read\_csv('fyx.csv')

# 温度离散化

data['temperature\_discrete'] = pd.cut(data['Temperature (C)'], bins=[-10, 0, 10, 20, 30], labels=["非常低温", "低温", "中温", "高温"])

# 湿度离散化

data['humidity\_discrete'] = pd.cut(data['Humidity'], bins=[0, 0.5, 0.75, 1], labels=["低湿度", "中湿度", "高湿度"])

# 风速离散化

data['windspeed\_discrete'] = pd.cut(data['Wind Speed (km/h)'], bins=[0, 10, 20, 30], labels=["低速风", "中速风", "高速风"])

# 查看离散化结果

print(data[['Temperature (C)', 'temperature\_discrete', 'Humidity', 'humidity\_discrete', 'Wind Speed (km/h)', 'windspeed\_discrete']].head())

data.to\_csv('离散化数据.csv', index=False)

print(data.dtypes)

# 选择相关的离散化特征

transactions = data[['temperature\_discrete', 'humidity\_discrete', 'windspeed\_discrete', 'Summary', 'WindDirection','Season','TimeOfDay','Daily Summary']].astype(str)

# 将每行记录转换为事务

transactions\_list = transactions.values.tolist()

# 打印前5条记录

for i in range(5):

    print(transactions\_list[i])

from mlxtend.preprocessing import TransactionEncoder

from mlxtend.frequent\_patterns import apriori, association\_rules

# 转换为适合Apriori算法的格式

te = TransactionEncoder() # 定义模型

transactions\_encoded = te.fit(transactions\_list).transform(transactions\_list) # 转换数据

df\_transactions = pd.DataFrame(transactions\_encoded, columns=te.columns\_) # 转换为DataFrame

# 设定最小支持度和置信度阈值并挖掘频繁项集

frequent\_itemsets = apriori(df\_transactions, min\_support=0.05, use\_colnames=True)

# 生成关联规则

rules = association\_rules(frequent\_itemsets, metric="confidence", min\_threshold=0.6)

**异常检测**

**DBSCAN**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import DBSCAN

from sklearn.metrics import confusion\_matrix, f1\_score, roc\_curve, auc

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report

# 读取CSV文件

df = pd.read\_csv("weatherHistory.csv", encoding='ISO-8859-1')

# 选择聚类和分类的特征

features = df[['Temperature (C)', 'Apparent Temperature (C)', 'Humidity',

               'Wind Speed (km/h)', 'Wind Bearing (degrees)',

               'Visibility (km)', 'Pressure (millibars)']]

# 标准化数据

scaler = StandardScaler()

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

# 使用DBSCAN进行聚类

dbscan = DBSCAN(eps=0.5, min\_samples=5)

labels = dbscan.fit\_predict(features\_scaled)

# 将 DBSCAN 的标签映射为 0 和 1，-1 为噪声，标记为 1（异常），其他标记为 0（正常）

y\_true = np.where(labels == -1, 1, 0)  # 1: 异常, 0: 正常

y\_pred = np.where(labels == -1, 1, 0)  # DBSCAN 输出的标签

# 计算噪声比（DBSCAN 标签为-1的比例）

noise\_count = np.sum(labels == -1)

noise\_ratio = noise\_count / len(labels)

# 输出噪声比

print(f"噪声数量: {noise\_count}")

print(f"噪声比: {noise\_ratio}")

# 混淆矩阵

cm = confusion\_matrix(y\_true, y\_pred)

print("混淆矩阵:\n", cm)

# 计算F1分数

f1 = f1\_score(y\_true, y\_pred)

print(f"F1 分数: {f1:.4f}")

# 计算 ROC 曲线

fpr, tpr, thresholds = roc\_curve(y\_true, y\_pred)

roc\_auc = auc(fpr, tpr)

# 绘制 ROC 曲线

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='gray', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.show()

# 返回其他评估指标（详细的分类报告）

print("\n分类报告:\n", classification\_report(y\_true, y\_pred))

**LOF**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import LocalOutlierFactor

from sklearn.metrics import confusion\_matrix, f1\_score, roc\_curve, auc, classification\_report

# 读取CSV文件

df = pd.read\_csv("weatherHistory.csv", encoding='ISO-8859-1')

# 选择特征

features = df[['Temperature (C)', 'Apparent Temperature (C)', 'Humidity',

               'Wind Speed (km/h)', 'Wind Bearing (degrees)',

               'Visibility (km)', 'Pressure (millibars)']]

# 数据标准化

scaler = StandardScaler()

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

# 使用LOF进行异常检测

lof = LocalOutlierFactor(n\_neighbors=20, contamination=0.1)

y\_pred = lof.fit\_predict(features\_scaled)

# LOF输出的预测值：1 表示正常点，-1 表示异常点（噪声）

# 将预测结果转换为 0 和 1 的标签，1 表示异常，0 表示正常

y\_pred = np.where(y\_pred == 1, 0, 1)

# 真实标签，在没有真实标签的情况下，假设LOF的噪声标签为真实标签

y\_true = y\_pred

# 计算噪声数量和噪声比

noise\_count = np.sum(y\_pred == 1)

noise\_ratio = noise\_count / len(y\_pred)

# 输出噪声数量和噪声比

print(f"噪声数量: {noise\_count}")

print(f"噪声比: {noise\_ratio}")

# 计算混淆矩阵

conf\_matrix = confusion\_matrix(y\_true, y\_pred)

print("混淆矩阵:\n", conf\_matrix)

# 计算F1分数

f1 = f1\_score(y\_true, y\_pred)

print(f"F1 分数: {f1:.4f}")

# 计算ROC曲线及AUC

fpr, tpr, thresholds = roc\_curve(y\_true, y\_pred)

roc\_auc = auc(fpr, tpr)

# 绘制ROC曲线

plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area = {roc\_auc:.2f})')

plt.plot([0, 1], [0, 1], color='gray', linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver Operating Characteristic (ROC) Curve')

plt.legend(loc='lower right')

plt.show()

# 返回详细的分类报告

print("\n分类报告:\n", classification\_report(y\_true, y\_pred))

0

**随机森林**

from sklearn.preprocessing import StandardScaler

# 选择用于检测异常的特征

features = data[['Temperature (C)', 'Humidity', 'Wind Speed (km/h)','Apparent Temperature (C)','Visibility (km)','Pressure (millibars)']]

# 标准化处理

scaler = StandardScaler()

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

from sklearn.ensemble import IsolationForest

# 初始化数据

remaining\_data = data.copy()

# 逐个维度进行异常检测

for feature in features.columns:

    # 训练孤立森林模型，仅使用当前特征

    feature\_values = remaining\_data[feature].values.reshape(-1, 1)

    isf\_feature = IsolationForest(contamination=0.05, random\_state=42)

    # 拟合模型并预测异常

    feature\_anomaly\_score = isf\_feature.fit\_predict(feature\_values)

    # 标记异常

    remaining\_data[f'{feature}\_anomaly'] = feature\_anomaly\_score < 0  # 小于0表示异常

    # 打印异常原因并删除异常数据,只打印异常数据

    anomalies = remaining\_data[remaining\_data[f'{feature}\_anomaly']]

    anomalies.to\_csv(f'{feature}\_异常数据.csv'.replace(' ','\_').replace('/','\_'), index=False)

    if not anomalies.empty:

        print(f"{feature} 维度异常数据：")

        print(anomalies)

        remaining\_data = remaining\_data[~remaining\_data[f'{feature}\_anomaly']]  # 删除异常数据

    # 删除标记列

    remaining\_data.drop(columns=[f'{feature}\_anomaly'], inplace=True)

# 最终更新的数据

print("处理后的数据：")

print(remaining\_data)

remaining\_data.to\_csv('异常数据处理后.csv', index=False)