Multinomial Logistic Regression with scikit-learn

Table of contents

Author: Nikos Papakostas
Created: November 2024
GitHub: https://github.com/papaknik
LinkedIn: https://www.linkedin.com/in/nikos-papakostas/

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import imblearn
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import warnings
import joblib
import watermark
from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings('ignore')
warnings.filterwarnings("always", category=ConvergenceWarning)
pd.set_option('max_colwidth', 38)
%load_ext watermark
```

In [2]:
```
%watermark
```

```
Last updated: 2025-02-13T13:30:56.894677+02:00

Python implementation: CPython
Python version       : 3.12.8
IPython version      : 8.30.0

Compiler    : MSC v.1929 64 bit (AMD64)
OS          : Windows
Release     : 11
Machine     : AMD64
Processor   : AMD64 Family 25 Model 80 Stepping 0, AuthenticAMD
CPU cores   : 16
Architecture: 64bit
```

In [3]:
```
%watermark -v
```

```
Python implementation: CPython
Python version       : 3.12.8
IPython version      : 8.30.0
```

In [ ]:
```
%watermark --iversions
```

```
IPython   : 8.30.0
joblib    : 1.4.2
imblearn  : 0.0
matplotlib: 3.10.0
numpy     : 1.26.4
seaborn   : 0.13.2
pandas    : 2.2.3
watermark : 2.5.0
sklearn   : 1.5.2
```

Introduction

This practice project utilizes data extracted from the Open-Meteo Historical Weather API to construct a Multinomial Classification problem. The dataset contains numerical weather attributes as independent variables, and we aim to predict a categorical target variable named 'weather_event', which can take the following values:

```
0: No precipitation
1: Rain event
2: Snowfall event
```

The major challenge of this project is the complete lack of Domain Knowledge which is a key asset in any machine learning project.
With limited time for subject research, I followed two simple steps:

1. Research similar machine learning tasks: I searched online for related ML projects to identify commonly used weather variables that could serve as predictors.

2. Logical reasoning: Before attempting to predict an event you first must be able to describe it. I considered how weather is usually described in forecasts, focusing on attributes like temperature, wind, humidity, and precipitation likelihood.

The dataset includes historical weather data from 2019 to 2023 for the center of Athens, Greece, and will serve as a practice set for multinomial classification. Our goal is to predict whether there will be no precipitation, rain, or snow at the specified location, utilizing Logistic Regression algorithm.
In subsequent notebooks, we will explore different algorithms to tackle this problem.

Important note
This project does not aim to replicate professional weather forecasting, where advanced numerical simulations and physical models are applied.
These models rely on complex simulations of atmospheric processes and can predict future values for key weather variables over time, while in this project our predictive modeling will be based purely on historical data.

Dataset Description

- temperature_2m: Represents the air temperature measured at a height of 2 meters above the ground. This is a standard height for temperature observations in weather forecasting.
- relative_humidity_2m: Measures the percentage of moisture in the air relative to the maximum amount of moisture the air can hold at the same temperature.
- dew_point_2m: Indicates the temperature at which air becomes saturated with moisture and condensation begins, leading to the formation of dew, fog, or clouds.
- rain: This variable measures the precipitation from rain, typically expressed in millimeters (mm). It represents the amount of rainfall over a certain period (e.g., hourly). Higher values indicate heavier rainfall.
- snowfall: This variable records the amount of precipitation in the form of snow, also generally measured in millimeters (mm). Like rain, it provides insight into the accumulation of snow over time.
- surface_pressure: Refers to the atmospheric pressure exerted at the Earth's surface, typically measured in hectopascals (hPa). It is crucial for determining weather conditions like high and low-pressure systems.

- cloud_cover_low & cloud_cover_mid: Represents the percentage of the sky covered by low-level and mid-level clouds. Low clouds form close to the ground, while mid clouds occur higher up in the atmosphere.
- evapotranspiration: Reflects the combined process of water evaporation from the surface and transpiration from plants. It is used to estimate water loss in ecosystems.
- vapour_pressure_deficit: Represents the difference between the amount of moisture in the air and how much moisture the air can hold when it is saturated. It's often used to assess plant water stress.
- wind_speed_10m: Measures the wind velocity at a height of 10 meters above the surface, which is the standard height for wind observations.
- wind_gusts_10m: Indicates the sudden, short-term increases in wind speed at 10 meters above the surface, which can impact weather severity.
- soil_temp_0_to_7cm: Measures the temperature of the soil in the top 7 cm layer, which is important for agriculture and assessing soil conditions.
- soil_moist_0_to_7cm: Reflects the water content in the top 7 cm of soil, important for understanding drought conditions and plant water availability.
- direct_radiation: Measures the amount of solar radiation received by the Earth's surface directly from the sun, without scattering by clouds or the atmosphere. This is important for energy balance calculations.

Detailed information about the selected variables, as well as all available options, can be found **here.**

Scraping with Open-Meteo API

Scraping data using the Open-Meteo Historical Forecast API is a simple process that can be completed in just a few steps. First, ensure that the required Python `openmeteo_requests`, `requests_cache` and `retry_requests` modules are installed in your environment.
If not, try running the following commands:
`pip install openmeteo_requests`

`pip install requests_cache` and
`pip install retry_requests`

Once the modules are installed, follow these steps:

- Navigate to the Open-Meteo Historical Weather API documentation, **here**
- Select the coordinates of the desired location (e.g., a city or a specific latitude and longitude).
- Choose the time interval by setting the Start Date and End Date.
- Pick the parameters (e.g., temperature, precipitation, wind speed) that you are interested in.

The API will generate a Python code snippet for you. Copy and paste this snippet into your notebook or script, and you are all set to start scraping data!

You can skip the following step and use directly the csv file that can be found in the `/data` folder of the repository in GitHub.

In [ ]:
```python
import openmeteo_requests
import requests_cache
from retry_requests import retry

# Setup the Open-Meteo API client with cache and retry on error
cache_session = requests_cache.CachedSession('.cache', expire_after = -1)
retry_session = retry(cache_session, retries = 5, backoff_factor = 0.2)
openmeteo = openmeteo_requests.Client(session = retry_session)

# Make sure all required weather variables are listed here
# The order of variables in hourly or daily is important to assign them correctly below
url = "https://archive-api.open-meteo.com/v1/archive"
params = {
        "latitude": 37.9838,
        "longitude": 23.7278,
        "start_date": "2019-01-01",
        "end_date": "2023-12-31",
        "hourly": ["temperature_2m", "relative_humidity_2m", "dew_point_2m", "rain", "snowfall",
               "surface_pressure", "cloud_cover_low", "cloud_cover_mid", "et0_fao_evapotranspiration",
               "vapour_pressure_deficit", "wind_speed_10m", "wind_gusts_10m", "soil_temperature_0_to_7cm",
               "soil_moisture_0_to_7cm", "direct_radiation"],
        "timezone": "auto"
}
```

```
responses = openmeteo.weather_api(url, params=params)

# Process first location. Add a for-loop for multiple locations or weather models
response = responses[0]
print(f"Coordinates {response.Latitude()}°N {response.Longitude()}°E")
print(f"Elevation {response.Elevation()} m asl")
print(f"Timezone {response.Timezone()} {response.TimezoneAbbreviation()}")
print(f"Timezone difference to GMT+0 {response.UtcOffsetSeconds()} s")

# Process hourly data. The order of variables needs to be the same as requested.
hourly = response.Hourly()
hourly_temperature_2m = hourly.Variables(0).ValuesAsNumpy()
hourly_relative_humidity_2m = hourly.Variables(1).ValuesAsNumpy()
hourly_dew_point_2m = hourly.Variables(2).ValuesAsNumpy()
hourly_rain = hourly.Variables(3).ValuesAsNumpy()
hourly_snowfall = hourly.Variables(4).ValuesAsNumpy()
hourly_surface_pressure = hourly.Variables(5).ValuesAsNumpy()
hourly_cloud_cover_low = hourly.Variables(6).ValuesAsNumpy()
hourly_cloud_cover_mid = hourly.Variables(7).ValuesAsNumpy()
hourly_et0_fao_evapotranspiration = hourly.Variables(8).ValuesAsNumpy()
hourly_vapour_pressure_deficit = hourly.Variables(9).ValuesAsNumpy()
hourly_wind_speed_10m = hourly.Variables(10).ValuesAsNumpy()
hourly_wind_gusts_10m = hourly.Variables(11).ValuesAsNumpy()
hourly_soil_temperature_0_to_7cm = hourly.Variables(12).ValuesAsNumpy()
hourly_soil_moisture_0_to_7cm = hourly.Variables(13).ValuesAsNumpy()
hourly_direct_radiation = hourly.Variables(14).ValuesAsNumpy()

hourly_data = {"date": pd.date_range(
        start = pd.to_datetime(hourly.Time(), unit = "s", utc = True),
        end = pd.to_datetime(hourly.TimeEnd(), unit = "s", utc = True),
        freq = pd.Timedelta(seconds = hourly.Interval()),
        inclusive = "left"
)}
hourly_data["temperature_2m"] = hourly_temperature_2m
hourly_data["relative_humidity_2m"] = hourly_relative_humidity_2m
hourly_data["dew_point_2m"] = hourly_dew_point_2m
hourly_data["rain"] = hourly_rain
hourly_data["snowfall"] = hourly_snowfall
hourly_data["surface_pressure"] = hourly_surface_pressure
hourly_data["cloud_cover_low"] = hourly_cloud_cover_low
hourly_data["cloud_cover_mid"] = hourly_cloud_cover_mid
```

```
hourly_data["evapotranspiration"] = hourly_et0_fao_evapotranspiration
hourly_data["vapour_pressure_deficit"] = hourly_vapour_pressure_deficit
hourly_data["wind_speed_10m"] = hourly_wind_speed_10m
hourly_data["wind_gusts_10m"] = hourly_wind_gusts_10m
hourly_data["soil_temp_0_to_7cm"] = hourly_soil_temperature_0_to_7cm
hourly_data["soil_moist_0_to_7cm"] = hourly_soil_moisture_0_to_7cm
hourly_data["direct_radiation"] = hourly_direct_radiation

df = pd.DataFrame(data = hourly_data)
df.info()
```

```
Coordinates 37.996482849121094°N 23.70967674255371°E
Elevation 92.0 m asl
Timezone b'Europe/Athens' b'EEST'
Timezone difference to GMT+0 10800 s
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 43824 entries, 0 to 43823
Data columns (total 16 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   date                     43824 non-null  datetime64[ns, UTC]
 1   temperature_2m           43824 non-null  float32
 2   relative_humidity_2m     43824 non-null  float32
 3   dew_point_2m             43824 non-null  float32
 4   rain                     43824 non-null  float32
 5   snowfall                 43824 non-null  float32
 6   surface_pressure         43824 non-null  float32
 7   cloud_cover_low          43824 non-null  float32
 8   cloud_cover_mid          43824 non-null  float32
 9   evapotranspiration       43824 non-null  float32
 10  vapour_pressure_deficit  43824 non-null  float32
 11  wind_speed_10m           43824 non-null  float32
 12  wind_gusts_10m           43824 non-null  float32
 13  soil_temp_0_to_7cm       43824 non-null  float32
 14  soil_moist_0_to_7cm      43824 non-null  float32
 15  direct_radiation         43824 non-null  float32
dtypes: datetime64[ns, UTC](1), float32(15)
memory usage: 2.8 MB
```

Creating the multiclass target variable

The first step is to create the target variable 'weather_event' for our multi-class classification problem.
We create a new column called 'weather_event' with the following rules:

```
0: When both the 'rain' and 'snowfall' variables are 0 (indicating no precipitation).
1: When the 'rain' variable is greater than 0 and the 'snowfall' variable is 0 (indicating a rain event).
2: When the 'snowfall' variable is greater than 0 and the 'rain' variable is 0 (indicating a snowfall
event).
```

Occasionally, there are some rare instances over the 5-year period where both rain and snowfall occurred within the same hour. In these cases, we assign to 'weather_event', the value based on which variable has the higher precipitation (rain or snowfall), ensuring that the most significant weather event is captured.

In [6]:
```python
# Define the conditions
conditions = [
    (df['rain'] > 0) & (df['snowfall'] > 0),   # Both rain and snowfall
    (df['rain'] > 0) & (df['snowfall'] == 0),  # Only rain
    (df['snowfall'] > 0) & (df['rain'] == 0)   # Only snowfall
]

# Define the values for each condition
values = [
    np.where(df['rain'] > df['snowfall'], 1, 2),  # Choose rain if higher, otherwise snowfall
    1,  # Only rain
    2   # Only snowfall
]

# Use np.select to create the 'weather_event' column
df['weather_event'] = np.select(conditions, values, default=0)  # Default is 0 for no precipitation
```

Removing the columns 'rain' and 'snowfall' since they are no longer needed after the creation of the target variable

In [7]:
```python
df = df.drop(columns=['rain', 'snowfall'])
```

In [ ]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 43824 entries, 0 to 43823
Data columns (total 15 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   date                     43824 non-null  datetime64[ns, UTC]
 1   temperature_2m           43824 non-null  float32
 2   relative_humidity_2m     43824 non-null  float32
 3   dew_point_2m             43824 non-null  float32
 4   surface_pressure         43824 non-null  float32
 5   cloud_cover_low          43824 non-null  float32
 6   cloud_cover_mid          43824 non-null  float32
 7   evapotranspiration       43824 non-null  float32
 8   vapour_pressure_deficit  43824 non-null  float32
 9   wind_speed_10m           43824 non-null  float32
 10  wind_gusts_10m           43824 non-null  float32
 11  soil_temp_0_to_7cm       43824 non-null  float32
 12  soil_moist_0_to_7cm      43824 non-null  float32
 13  direct_radiation         43824 non-null  float32
 14  weather_event            43824 non-null  int32
dtypes: datetime64[ns, UTC](1), float32(13), int32(1)
memory usage: 2.7 MB
```

Adding a time related variable in the dataset

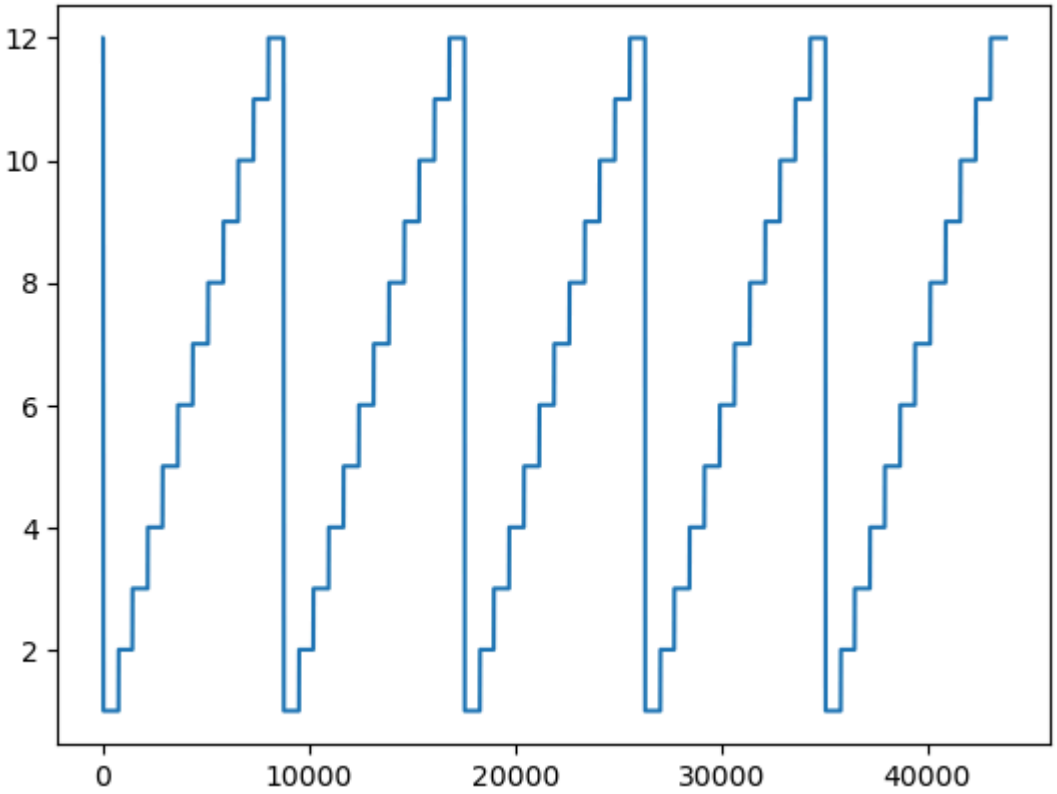
Everyone knows from life experience that weather is closely tied to seasons, meaning that the likelihood of a 'snowfall' event is much higher during winter months, or that the likelihood of 'rain' is expected to be lower during the months of July and August *(at least up to most recent years...)*.
Considering this, the addition of the month as a numerical variable (ranging from 1 to 12), can be a valuable addition to the dataset, and we expect it to have significant predictive power in the models we build.

In [9]:
```
df['month'] = df['date'].dt.month
```

But when plotting the month, we see something strange on the plot. While the months from 1 (January) to 12 (December) escalate sequentially by a step of 1, when reaching 12, the plot jumps back to 1 and repeats the same pattern. This means that encoding the month variable by a strict integer with a step of 1 exhibits a discontinuity when reaching December in every year of the dataframe. So, in the months of December and January while they should have a distance of 1 as all other consecutive months exhibit, their difference with the simple

integer encoding is 11 (12-1)! This is well known for all features that exhibit a cyclical behavior by nature and must be handled before modeling, in order to help the model interpret the influence of the 'month' variable on the result (predicting the target variable) more efficiently.

In [10]:
```
ax = df['month'].plot()
```



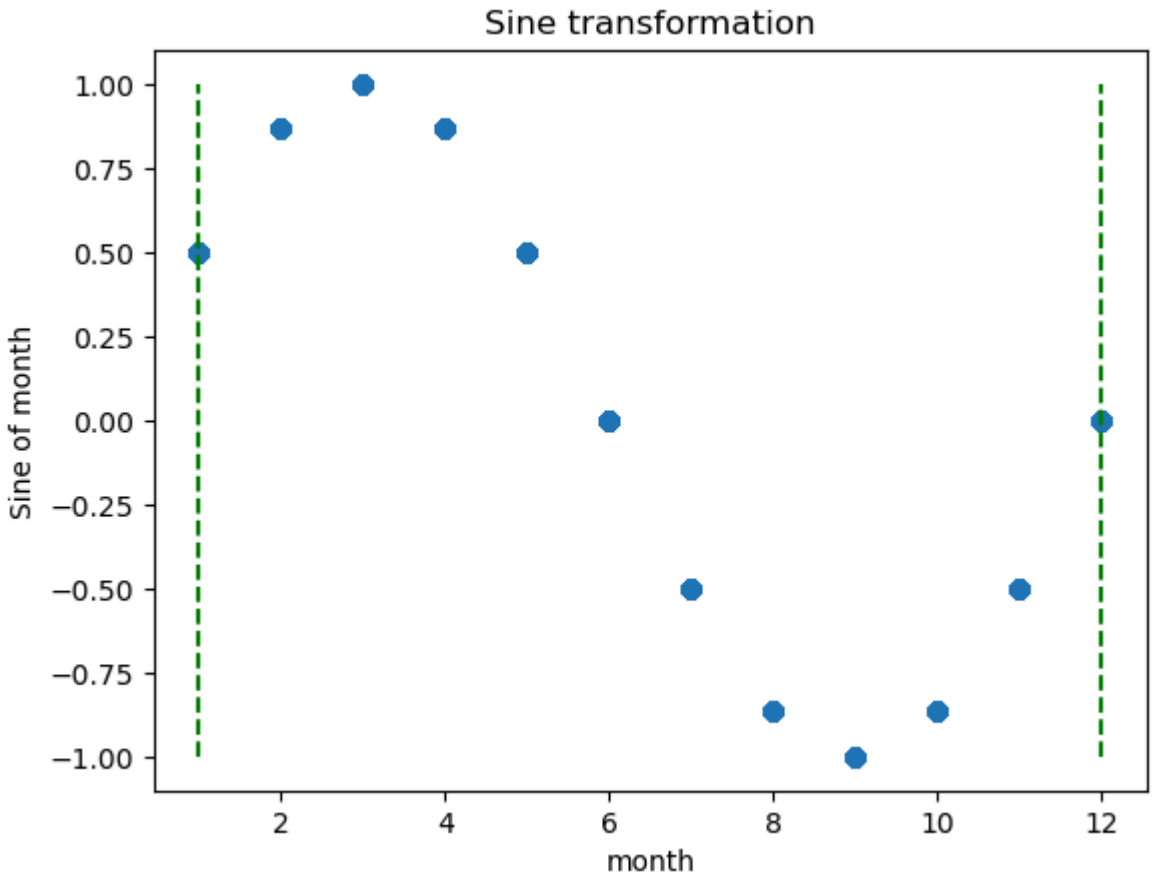
We can encode the `month` variable so as to capture its cyclical behavior and eliminate the discontinuity observed between months of December and January.
You can find more information on encoding cyclical features for ML modeling purposes in the link below:
https://feature-engine.trainindata.com/en/1.7.x/user_guide/creation/CyclicalFeatures.html

In [11]:
```
# Applying sin and cosine transformation on the  variable
df['month_sin'] = np.round(np.sin(2 * np.pi * df['month'] / 12), decimals=3)
```

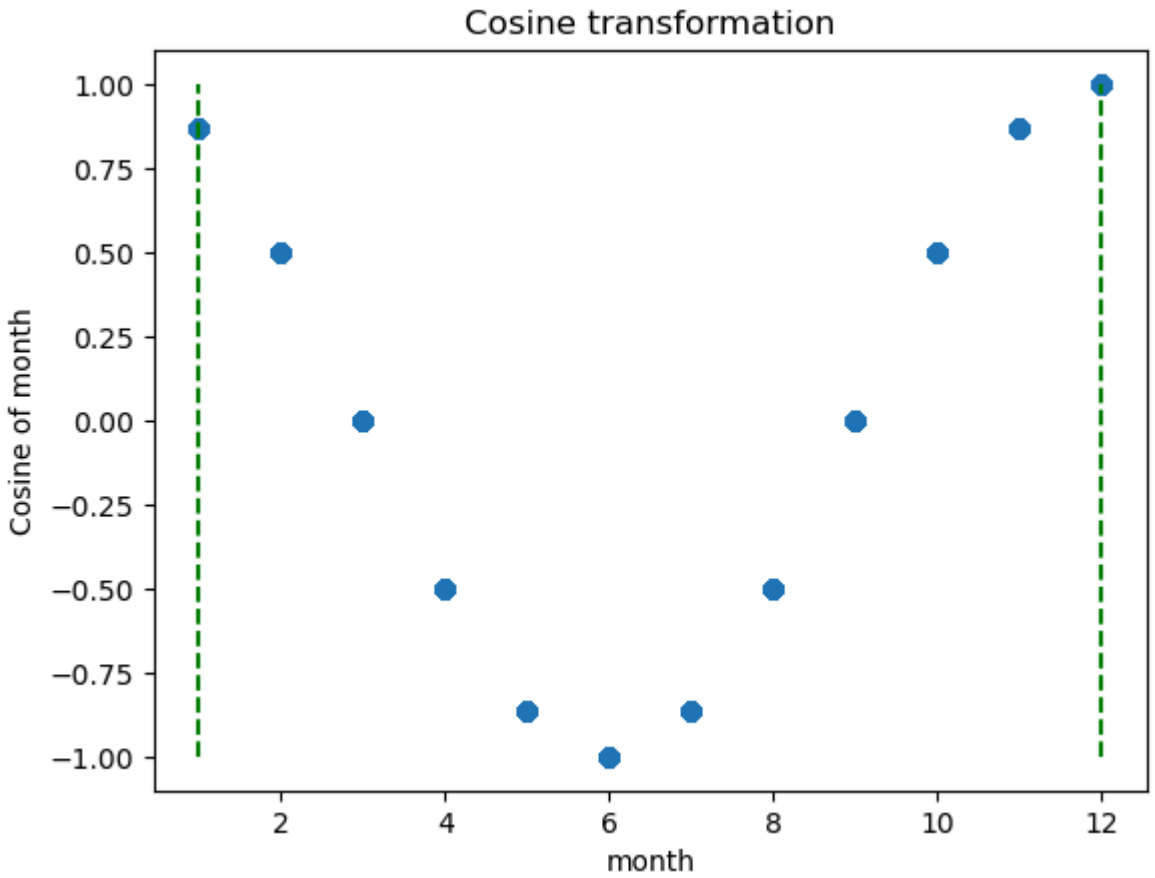
```
df['month_cos'] = np.round(np.cos(2 * np.pi * df['month'] / 12), decimals=3)
```

We can plot the sin and cosine transformations of the variable `month` and see that the discontinuity has been eliminated. The values on the y-axis for the transformed `month` variable in January (1) and December (12) are much closer than in the case of the absolute integer encoding.

In [12]:
```
plot = plt.scatter(df["month"], df["month_sin"])
vline1 = plt.vlines(x=1, ymin=-1, ymax=1, color='g', linestyles='dashed')
vline2 = plt.vlines(x=12, ymin=-1, ymax=1, color='g', linestyles='dashed')
# Axis labels
ylabel = plt.ylabel('Sine of month')
xlabel = plt.xlabel('month')
title = plt.title('Sine transformation')
```

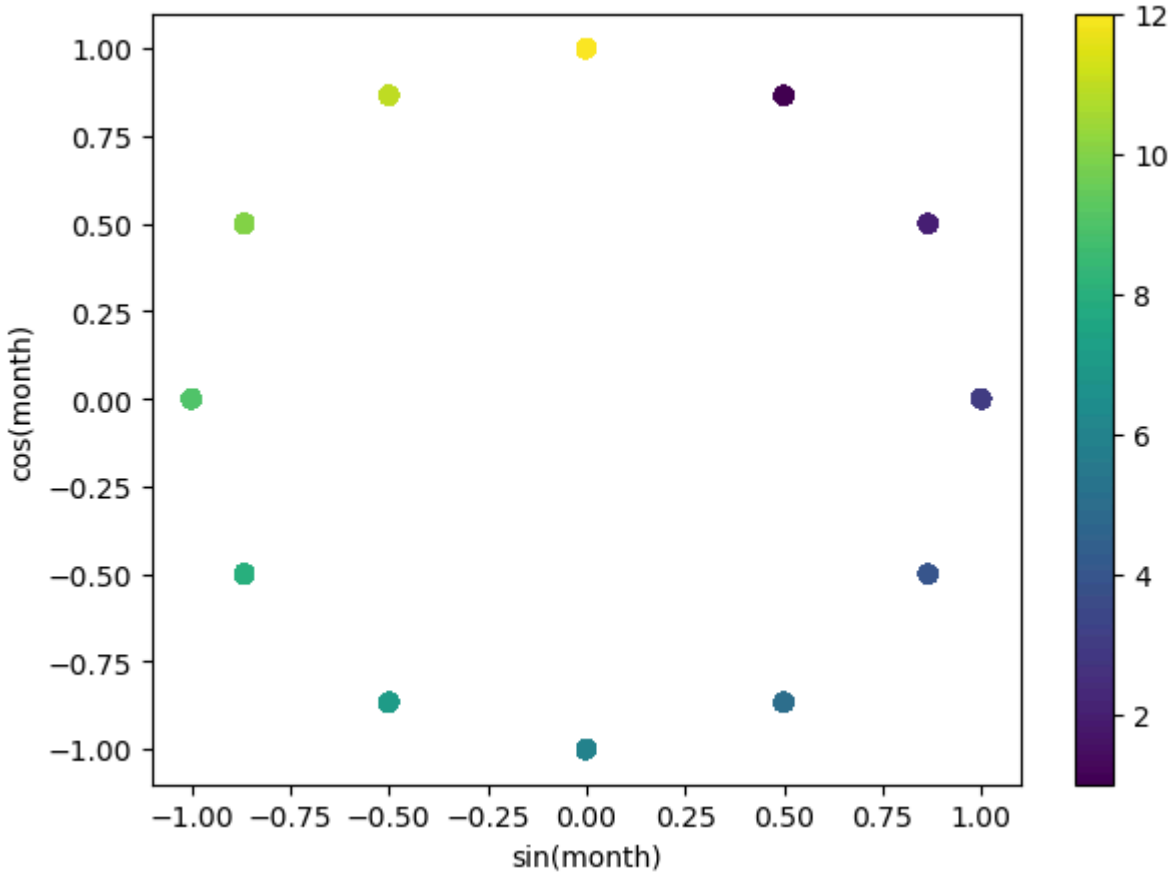


In [13]:
```
plot = plt.scatter(df["month"], df["month_cos"])
vline1 = plt.vlines(x=1, ymin=-1, ymax=1, color='g', linestyles='dashed')
vline2 = plt.vlines(x=12, ymin=-1, ymax=1, color='g', linestyles='dashed')
# Axis labels
ylabel = plt.ylabel('Cosine of month')
xlabel = plt.xlabel('month')
title = plt.title('Cosine transformation')
```



We can also scatter plot the 2 transformed variables and observe that the cyclical nature of the month variable has been successfully captured.

In [14]:
```
fig, ax = plt.subplots(figsize=(7, 5))
sp = ax.scatter(df['month_sin'], df['month_cos'], c=df["month"])
axis = ax.set(
    xlabel="sin(month)",
    ylabel="cos(month)",
)
colorbar = fig.colorbar(sp)
```



In the above plot we observe how the month of December (value 12-deep yellow color) neighbors with January (value 1-dark blue color), eliminating the discontinuity of the original `month` variable.

Since the `month` variable is no longer needed we drop it from the dataframe

In [ ]:
```
# Drop the original month column since it is no longer needed
df = df.drop('month', axis=1)
```

Creating Lagged Variables for Predictive Modeling

Predicting current weather using present data isn't true forecasting.
To make this project more realistic and challenging, we will introduce lagged variables.
Specifically, to predict whether it will rain, snow, or not at a given `time point t`, we will use weather parameters from one hour earlier (`time point t-1`). This approach enables us to predict future weather conditions based on past observations, simulating the use of historical data.
The addition of lagged variables makes the project more challenging and adds a touch of realism, while at the same time we can still practice with ML classification algorithms

Feel free to experiment and alter the data to your liking.
Instead of using the previous hour's weather attributes, you can use the values from the past 3 hours, or even compile a moving average value, for example an average of the past 3 hours where you average the values of each predictor for the time points t-3, t-2 and t-1, in order to predict the 'weather_event' attribute at time point t.

In [16]:
```python
# List of columns for which to create lag variables
lag_columns = ["temperature_2m",
    "relative_humidity_2m",
    "dew_point_2m",
    "surface_pressure",
    "cloud_cover_low",
    "cloud_cover_mid",
    "evapotranspiration",
    "vapour_pressure_deficit",
    "wind_speed_10m",
    "wind_gusts_10m",
    "soil_temp_0_to_7cm",
    "soil_moist_0_to_7cm",
    "direct_radiation",
]

# Create a new DataFrame to store the lagged features
lagged_1h_df = df.copy()  # Copy the original DataFrame to modify it

# Create 1-hour lagged features
for col in lag_columns:
    lagged_1h_df[f'{col}_lag_1h'] = df[col].shift(1)  # Shift by 1 step (1 hour back in the data)

# Drop rows with NaN values (if any), that result from the lagging process
```

```
lagged_1h_df = lagged_1h_df.dropna(subset=[f'{col}_lag_1h' for col in lag_columns])

# Drop the current-time columns to keep only the lagged versions
lagged_1h_df = lagged_1h_df.drop(columns=lag_columns)
```

A common practice in ML is to place the target/response variable as the final column in the dataframe, clearly separated from the predictor variables.

In [17]:
```
# Remove (pop) the 'weather_event' column from the DataFrame.
weather_event = lagged_1h_df.pop('weather_event')

# Reassign it as the last column.
lagged_1h_df['weather_event'] = weather_event

# Verify the new order:
lagged_1h_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 43823 entries, 1 to 43823
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   date                            43823 non-null  datetime64[ns, UTC]
 1   month_sin                       43823 non-null  float64
 2   month_cos                       43823 non-null  float64
 3   temperature_2m_lag_1h           43823 non-null  float32
 4   relative_humidity_2m_lag_1h     43823 non-null  float32
 5   dew_point_2m_lag_1h             43823 non-null  float32
 6   surface_pressure_lag_1h         43823 non-null  float32
 7   cloud_cover_low_lag_1h          43823 non-null  float32
 8   cloud_cover_mid_lag_1h          43823 non-null  float32
 9   evapotranspiration_lag_1h       43823 non-null  float32
 10  vapour_pressure_deficit_lag_1h  43823 non-null  float32
 11  wind_speed_10m_lag_1h           43823 non-null  float32
 12  wind_gusts_10m_lag_1h           43823 non-null  float32
 13  soil_temp_0_to_7cm_lag_1h       43823 non-null  float32
 14  soil_moist_0_to_7cm_lag_1h      43823 non-null  float32
 15  direct_radiation_lag_1h         43823 non-null  float32
 16  weather_event                   43823 non-null  int32
dtypes: datetime64[ns, UTC](1), float32(13), float64(2), int32(1)
memory usage: 3.7 MB
```

Save the dataframe to a `csv` file

In [ ]:
```
athens_weather_df_2019_2023 = lagged_1h_df.to_csv('data/athens_weather_df_2019_2023.csv', index=False)
```

Basic Exploratory Data Analysis

If you wish to skip all the above steps including downloading the data from the API, you can simply load the pre-saved dataset directly from the CSV file, available in the 'data' folder of this project's GitHub repository.

In [ ]:
```
lagged_1h_df = pd.read_csv('data/athens_weather_df_2019_2023.csv')
```

Descriptive statistics of the DataFrame

In [18]:
```
lagged_1h_df.describe().transpose().round(2)
```

Out[18]:

	count	mean	std	min	25%	50%	75%	max
month_sin	43823.0	-0.00	0.71	-1.00	-0.87	0.00	0.50	1.00
month_cos	43823.0	-0.00	0.71	-1.00	-0.87	0.00	0.87	1.00
temperature_2m_lag_1h	43823.0	18.24	7.93	-2.19	12.21	17.56	24.26	43.81
relative_humidity_2m_lag_1h	43823.0	62.36	18.49	8.38	48.58	64.57	77.69	100.00
dew_point_2m_lag_1h	43823.0	10.00	5.22	-11.04	6.76	10.41	13.76	23.36
surface_pressure_lag_1h	43823.0	1004.17	5.83	973.40	1000.32	1003.65	1007.60	1028.19
cloud_cover_low_lag_1h	43823.0	10.38	23.18	0.00	0.00	0.00	6.00	100.00
cloud_cover_mid_lag_1h	43823.0	14.50	27.04	0.00	0.00	0.00	15.00	100.00
evapotranspiration_lag_1h	43823.0	0.16	0.20	0.00	0.01	0.05	0.24	0.97
vapour_pressure_deficit_lag_1h	43823.0	1.04	0.99	0.00	0.32	0.66	1.45	8.26
wind_speed_10m_lag_1h	43823.0	9.26	5.24	0.00	5.45	8.40	12.14	42.49
wind_gusts_10m_lag_1h	43823.0	20.13	10.30	2.16	12.24	17.64	26.64	82.44
soil_temp_0_to_7cm_lag_1h	43823.0	20.60	10.76	-2.94	12.01	18.91	27.96	52.51
soil_moist_0_to_7cm_lag_1h	43823.0	0.15	0.14	0.00	0.02	0.08	0.29	0.43
direct_radiation_lag_1h	43823.0	144.76	221.38	0.00	0.00	1.00	250.00	873.00
weather_event	43823.0	0.09	0.29	0.00	0.00	0.00	0.00	2.00

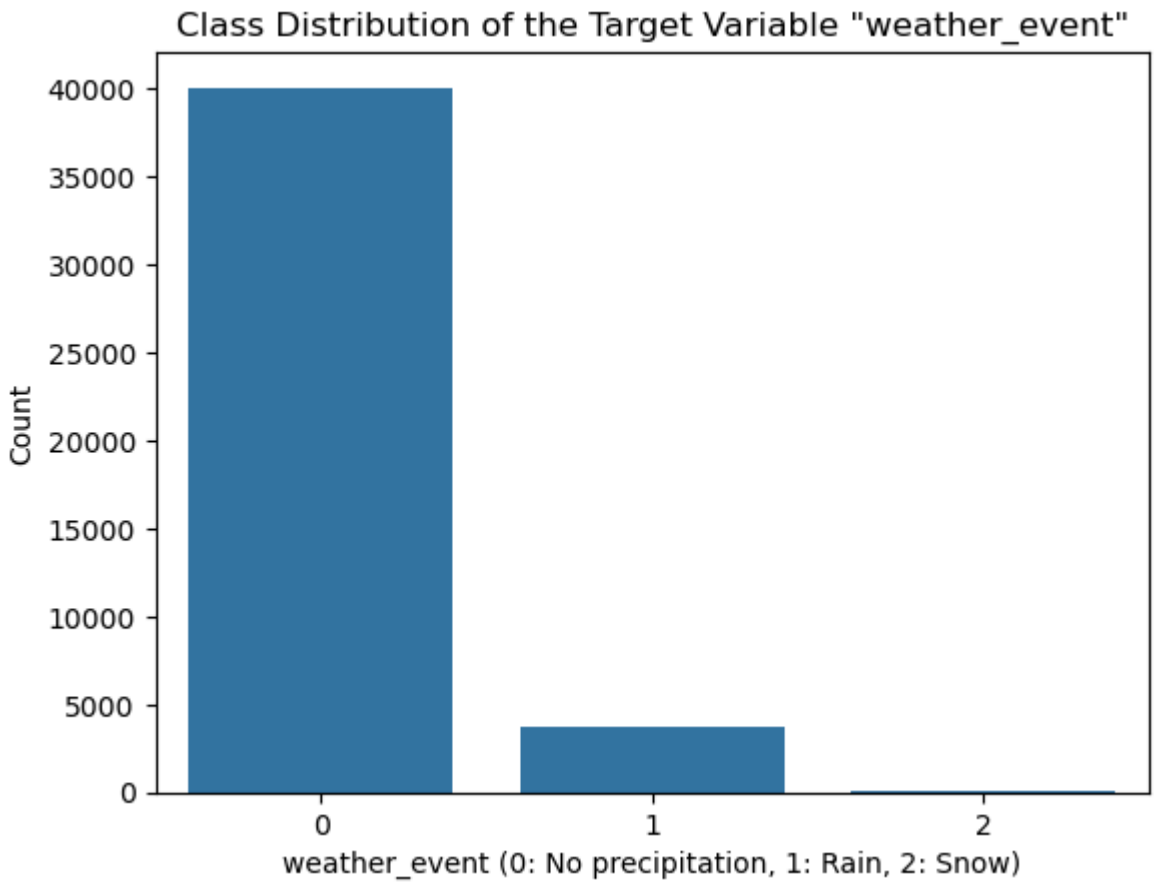
From the above descriptive statistics, it is clear that the dataset contains features with vastly different units and scales. Attributes such as temperature, pressure, and wind speed inherently have different ranges and variances.
Without scaling, features with larger values may disproportionately influence the model, leading to biased or incorrect results.

Scaling the data ensures that all features contribute equally to the learning process, allowing the model to consider each feature evenly. Therefore, it is crucial to scale the data before proceeding with any modeling.

Class imbalance of the target variable `weather_event`

In [22]:
```
# Checking the class distribution of the target variable 'rain'
class_counts = lagged_1h_df['weather_event'].value_counts()
print(class_counts)

# To see the percentage distribution:
class_percentages = lagged_1h_df['weather_event'].value_counts(normalize=True) * 100
print(class_percentages)

sns.countplot(x='weather_event', data=lagged_1h_df)
plt.title('Class Distribution of the Target Variable "weather_event"')
plt.xlabel('weather_event (0: No precipitation, 1: Rain, 2: Snow)')
plt.ylabel('Count')
plt.show();
```

```
weather_event
0    40083
1     3677
2       63
Name: count, dtype: int64
weather_event
0    91.465669
1     8.390571
2     0.143760
Name: proportion, dtype: float64
```



The dataset is **severely imbalanced**, with approximately 91.5% of the records representing “no precipitation" events, 8.4% for "rain" events, and only 0.1% for "snowfall" events. This class imbalance is likely to affect the model's performance, as it may become biased toward predicting the majority class (no precipitation) during training.
To address this issue, an oversampling technique such as SMOTE (Synthetic Minority Over-sampling Technique) should be applied to ensure that the model can learn from the minority classes effectively.
Additionally, when evaluating the model's performance, we need to focus on metrics like precision, recall and f1-score than relying on accuracy which can lead us to misleading conclusions. In our dataset for example, a 'dummy' model that always predicts 0, might achieve a high accuracy of 91.5%, but would completely fail in predicting rain or snowfall events, which was the purpose it was built for.

Duplicated records and missing values

In [26]:
```
lagged_1h_df.duplicated().sum()
```

Out[26]:
```
0
```

When preparing data for modeling with scikit-learn, it's crucial to ensure there are no missing (null) values, as most algorithms in scikit-learn cannot handle them. If missing values are present, the model will fail to execute. To handle this, you can use techniques like imputation (e.g., replacing missing values with the median or more advanced techniques).
If the dataset is large enough and the missing values are minimal, removing these records can also be an option, as long as the data's distributions are not distorted.
You can find useful information about missing data imputation in the **official documentation** of scikit-learn library.

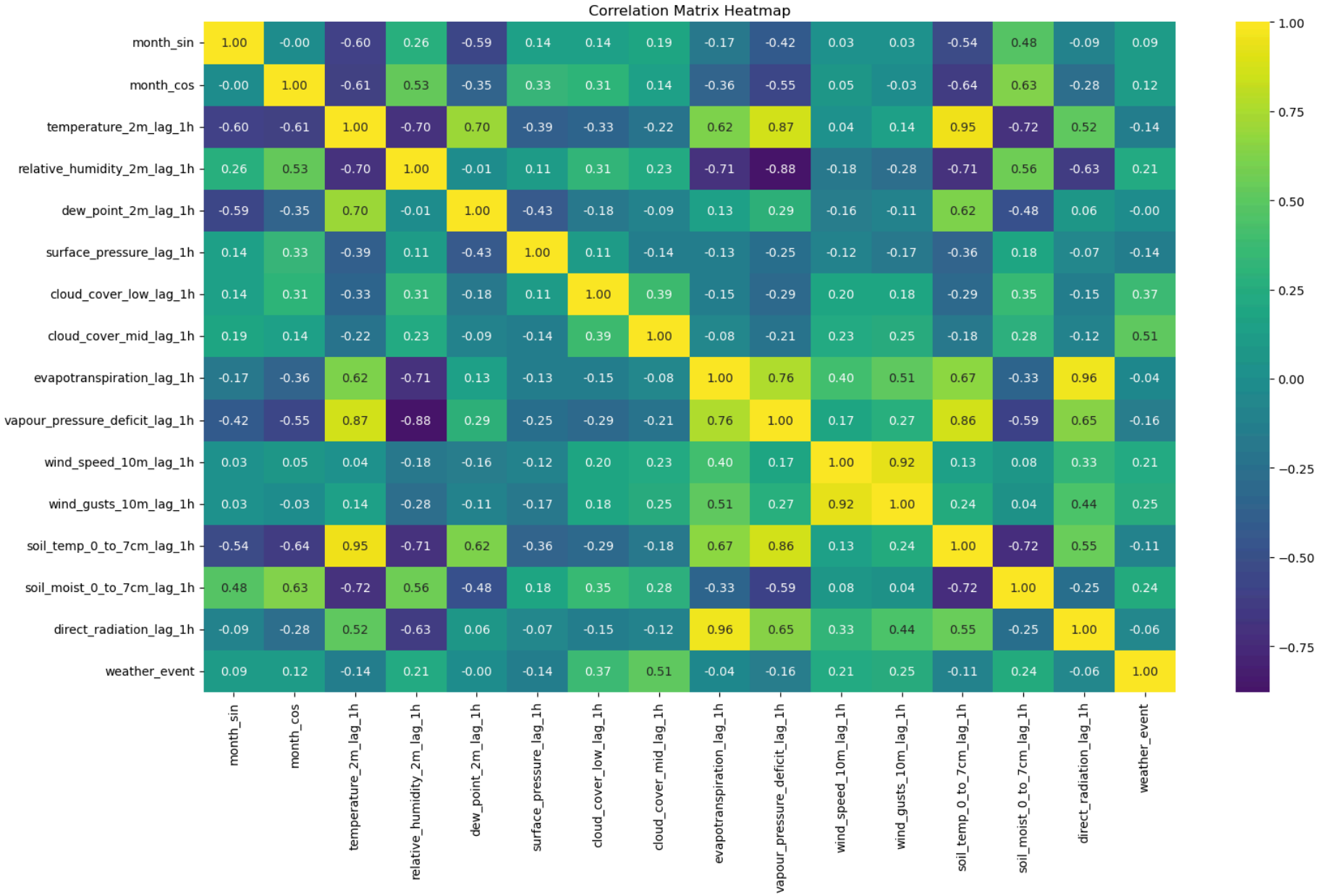
In [ ]:
```
lagged_1h_df.isna().sum()
```

Out[ ]:
```
date                             0
month_sin                        0
month_cos                        0
temperature_2m_lag_1h            0
relative_humidity_2m_lag_1h      0
dew_point_2m_lag_1h              0
surface_pressure_lag_1h          0
cloud_cover_low_lag_1h           0
cloud_cover_mid_lag_1h           0
evapotranspiration_lag_1h        0
vapour_pressure_deficit_lag_1h   0
wind_speed_10m_lag_1h            0
wind_gusts_10m_lag_1h            0
soil_temp_0_to_7cm_lag_1h        0
soil_moist_0_to_7cm_lag_1h       0
direct_radiation_lag_1h          0
weather_event                    0
dtype: int64
```

The correlation matrix

In [32]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
# Calculate the correlation matrix
corr_matrix = lagged_1h_df.corr(numeric_only = True, method = 'pearson')

# Create the heatmap
figure = plt.figure(figsize=(18, 10));
heatmap = sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='viridis', center=0)

# Add a title
title = plt.title('Correlation Matrix Heatmap')

# Display the heatmap
plt.show()
```



There are several strong correlations among the variables, with values well above 0.75. Some pairs of variables show correlations as high as -0.88, 0.92, or even 0.96!

The issue of multicollinearity

One common approach when dealing with strongly correlated variables is to remove one variable from each highly correlated pair—the one that proves to be less contributive to the model's outcome.
This can be based on the variables/features contribution to the model's performance, which can only be evaluated after modeling.
So, in this case rerunning the model is necessary to assess the impact on performance after each variable removal, making this an iterative process that requires time and patience.

An alternative approach is Principal Component Analysis (PCA), a technique used to reduce the dimensionality of the dataset and handle multicollinearity. PCA transforms correlated variables into uncorrelated principal components. However, the downside of this technique is that while it effectively addresses multicollinearity, it results in a model that is difficult to interpret. Since the transformed components do not correspond directly to the original features, and thus the concept of feature importance becomes less useful in cases, interpretability is important.

Feature engineering and Interaction Terms is another approach, but it definitely demands Domain knowledge and experience on this particular field, so this is not an option for us also.

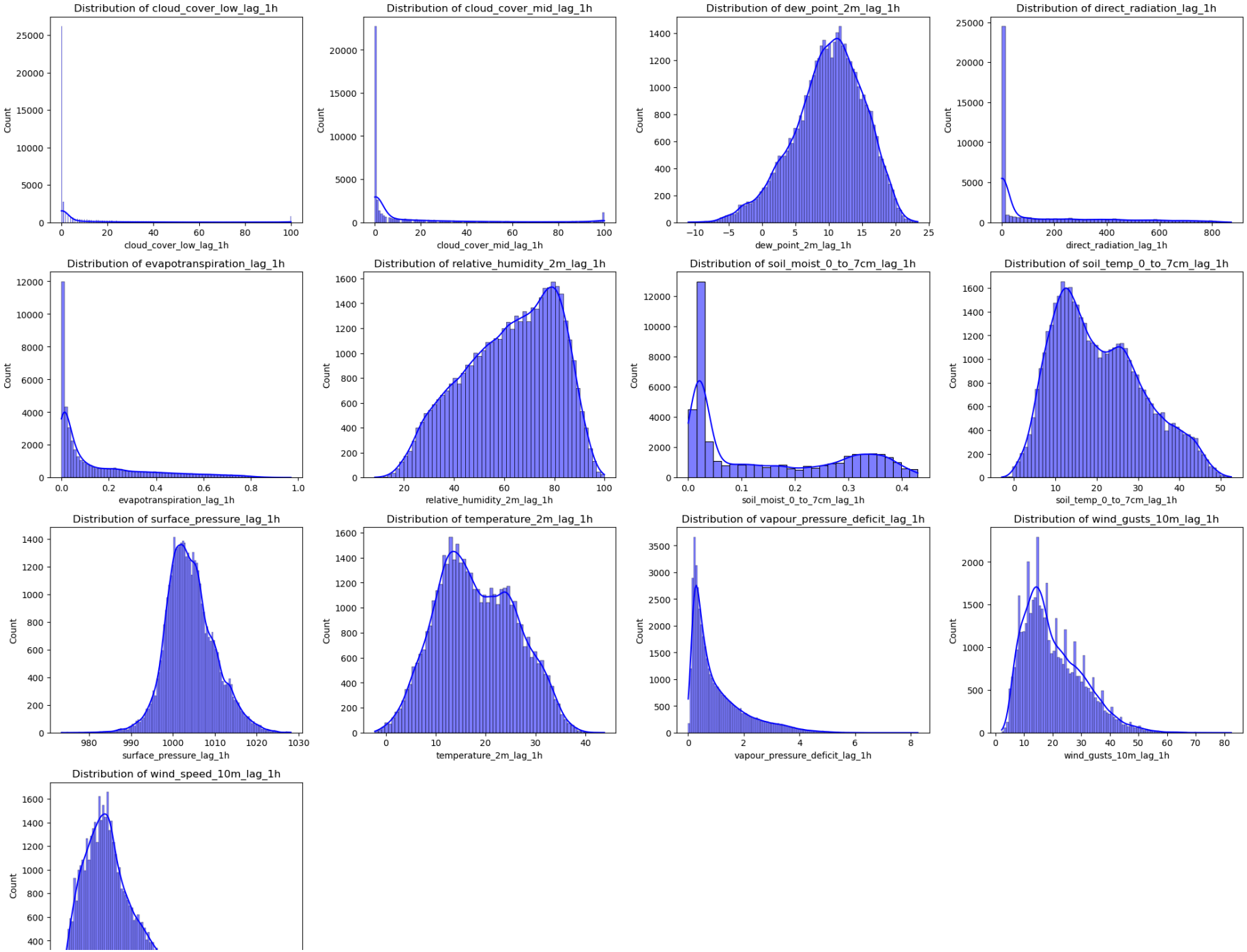
What remains as an option to handle the issue of collinearity and multicollinearity is Regularization.
Specifically, using L2 Regularization penalty within Logistic Regression class allows us to keep all variables, even if they are highly correlated, by shrinking their coefficients.
This addresses multicollinearity (*to an extent*), without losing interpretability while at the same time it helps control overfitting and provides a more balanced model without removing critical features or transforming them into less interpretable forms.
The good news is that the `LogisticRegression()` class of scikit-learn, uses by default for the penalty argument the l2 term and therefore we will leave multicollinearity to be handled (to an extent of course) internally by the algorithm.

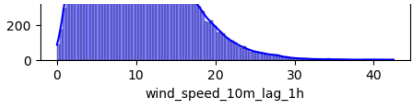
Therefore, we will retain all features in our dataset and let the algorithm mitigate the effects of multicollinearity during the modeling process.

Plotting the predictor distributions

In [39]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
# List of columns to plot
columns_to_plot = lagged_1h_df.columns.difference(['date', 'month_sin', 'month_cos', 'weather_event'])

figure = plt.figure(figsize=(20, 20))  # Adjusted for better spacing

# Loop through each column and create a distribution (histogram)
for i, col in enumerate(columns_to_plot, start=1):
    subplot = plt.subplot(len(columns_to_plot)//3 + 1, 4, i)
    histplot = sns.histplot(lagged_1h_df[col], kde=True, color='blue')

    title = plt.title(f'Distribution of {col}', fontsize=12)

# Adjust the layout for better visual spacing
layout = plt.tight_layout()
plt.show()
```


Distribution of cloud_cover_low_lag_1h
cloud_cover_low_lag_1h
Distribution of cloud_cover_mid_lag_1h
cloud_cover_mid_lag_1h
Distribution of dew_point_2m_lag_1h
dew_point_2m_lag_1h
Distribution of direct_radiation_lag_1h
direct_radiation_lag_1h
Distribution of evapotranspiration_lag_1h
evapotranspiration_lag_1h
Distribution of relative_humidity_2m_lag_1h
relative_humidity_2m_lag_1h
Distribution of soil_moist_0_to_7cm_lag_1h
soil_moist_0_to_7cm_lag_1h
Distribution of soil_temp_0_to_7cm_lag_1h
soil_temp_0_to_7cm_lag_1h
Distribution of surface_pressure_lag_1h
surface_pressure_lag_1h
Distribution of temperature_2m_lag_1h
temperature_2m_lag_1h
Distribution of vapour_pressure_deficit_lag_1h
vapour_pressure_deficit_lag_1h
Distribution of wind_gusts_10m_lag_1h
wind_gusts_10m_lag_1h
Distribution of wind_speed_10m_lag_1h
Count



From the distribution plots there are several conclusions we can derive, such as:

- 'surface_pressure' and 'dew_point_2m' display more symmetrical distributions, suggesting that the values for these variables are distributed more evenly around a central point. They can be well approximated from the normal distribution and therefore easier to predict
- 'evapotranspiration' and 'vapour_pressure' are highly right-skewed. This means that most of their values are concentrated close to 0, indicating that extreme values to the right tail are rare, and most likely will be indicated as outliers
- the values in features like 'relative_humidity' and 'wind_gusts' are spread across a wider range, which could imply that these features fluctuate significantly over time
- just like spotted in the descriptive statistics table, the varying scales of some distributions (some are tightly concentrated around a single value, while others are spread out over a large range) are evident, and therefore it is crucial to use scaling methods like `RobustScaler` or `StandardScaler` before applying machine learning algorithms, to prevent certain features from dominating due to their wider range.

Feature Transformation

In cases where variables deviate significantly from a Gaussian distribution, particularly those with high positive skewness or heavily tailed (high kurtosis), it may be necessary to apply a transformation to approximate normality.

Many machine learning models and statistical techniques assume normality, as variables with a Gaussian-like distribution can lead (but not guarantee) to better model performance and more reliable inferences. The Yeo-Johnson transformation is one such method, particularly useful for handling variables that include zero or negative values. Unlike the logarithmic and square root transformations, which are constrained to non-negative values, Yeo-Johnson can be applied to this range of values, making it a valuable transformation tool.

However, the effectiveness of this or any transformation is not guaranteed and may not improve the model's performance.
The process is iterative: you apply the transformation, evaluate the results, and reassess if needed.

Apart from observing visually the predictors distributions you can approach the task numerically and spot the variables that need a transformation and lead to a potential increase of the model’s performance by computing the skewness and kurtosis of each predictor variable.

If the values for these statistics exceed a certain threshold, then the predictor can be a serious candidate for transformation. A general rule of thumb is that a Kurtosis value above 4 captures heavy-tailed variables and a Skewness value above or below ±1 captures significantly asymmetric variables. These are the values we will also use as the threshold to spot the predictors in the data set that need further processing through Yeo-Johnson transformation.

Further information about feature transformation can be found in the links below:
https://www.analyticsvidhya.com/blog/2021/05/feature-transformations-in-data-science-a-detailed-walkthrough/
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PowerTransformer.html

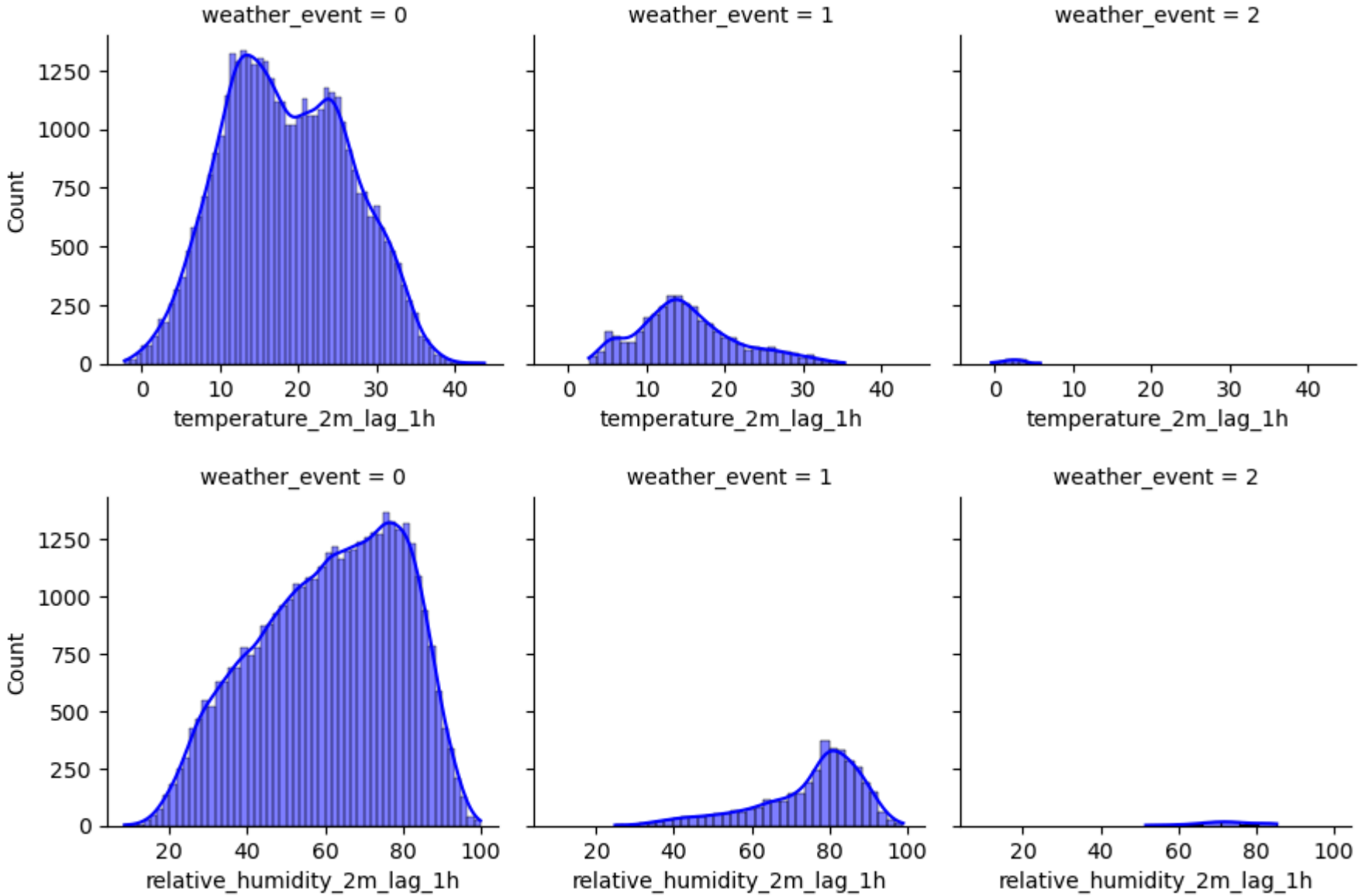
In [ ]:
```
from scipy.stats import skew, kurtosis

# Step 1: Filter numeric columns
numeric_cols = lagged_1h_df.select_dtypes(include=['float64'])

# Step 2: Set thresholds
skew_threshold = 1
kurt_threshold = 4

# Step 3: Calculate skewness and kurtosis for numeric columns
skewness = numeric_cols.apply(lambda col: skew(col.dropna()), axis=0)  # Drop NA to avoid errors
kurt = numeric_cols.apply(lambda col: kurtosis(col.dropna()), axis=0)

# Step 4: Identify variables exceeding thresholds
skewed_cols = skewness[abs(skewness) > skew_threshold].index.tolist()
high_kurt_cols = kurt[kurt > kurt_threshold].index.tolist()

# Step 5: Combine results
variables_to_transform = list(set(skewed_cols + high_kurt_cols))

# Display results
print("Skewness:\n", skewness)
print("\nKurtosis:\n", kurt)
print("\nVariables to consider for transformation:\n", variables_to_transform)
```

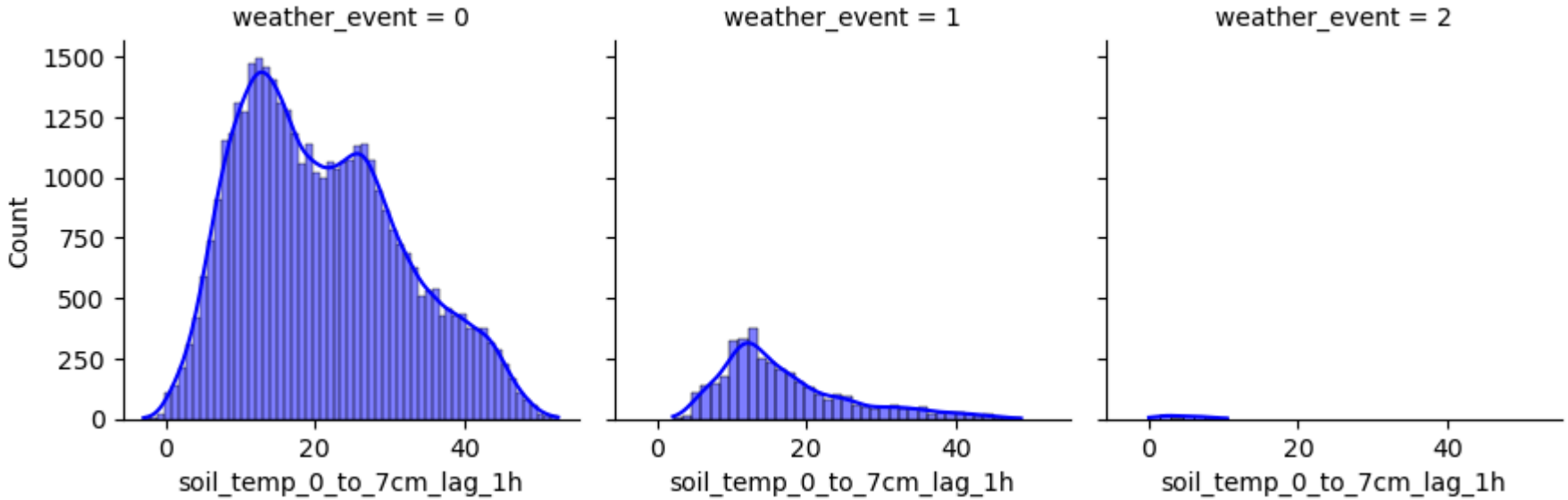
```
Skewness:
 month_sin                       0.010383
month_cos                       0.009882
temperature_2m_lag_1h           0.150348
relative_humidity_2m_lag_1h    -0.364587
dew_point_2m_lag_1h            -0.433332
surface_pressure_lag_1h         0.271628
cloud_cover_low_lag_1h          2.679562
cloud_cover_mid_lag_1h          2.087584
evapotranspiration_lag_1h       1.476544
vapour_pressure_deficit_lag_1h  1.603295
wind_speed_10m_lag_1h           0.933306
wind_gusts_10m_lag_1h           0.919183
soil_temp_0_to_7cm_lag_1h       0.491397
soil_moist_0_to_7cm_lag_1h      0.541617
direct_radiation_lag_1h         1.432146
dtype: float64

Kurtosis:
 month_sin                      -1.495498
month_cos                       -1.503180
temperature_2m_lag_1h           -0.675835
relative_humidity_2m_lag_1h     -0.785772
dew_point_2m_lag_1h             -0.028103
surface_pressure_lag_1h          0.730412
cloud_cover_low_lag_1h           6.387977
cloud_cover_mid_lag_1h           3.240495
evapotranspiration_lag_1h        1.235104
vapour_pressure_deficit_lag_1h   2.491644
wind_speed_10m_lag_1h            1.090578
wind_gusts_10m_lag_1h            0.755093
soil_temp_0_to_7cm_lag_1h       -0.539984
soil_moist_0_to_7cm_lag_1h      -1.327005
direct_radiation_lag_1h          0.823300
dtype: float64

Variables to consider for transformation:
 ['cloud_cover_low_lag_1h', 'evapotranspiration_lag_1h', 'direct_radiation_lag_1h', 'cloud_cover_mid_lag_1h', 'vapour_pressure_
deficit_lag_1h']
```

Exploring predictor variable distributions across target classes

Another essential and insightful step is to check how predictors behave across different classes of the target variable.
This step allows us to identify patterns and understand the relationship between predictors and the target classes (`no precipitation` , `rain` and `snowfall`).
Detecting any significant variation (shape and value counts across classes) in distributions might indicate that a variable could be an important predictor for the occurrence of a class, while similar distributions across classes could suggest a variable with less predictive power.

In [ ]:
```
g1 = sns.FacetGrid(lagged_1h_df, col="weather_event", col_wrap=3, height=3)
map1 = g1.map(sns.histplot, "temperature_2m_lag_1h", kde=True, color="blue")
subplot1 = g1.fig.subplots_adjust(top=0.9)

g2 = sns.FacetGrid(lagged_1h_df, col="weather_event", col_wrap=3, height=3)
map2 = g2.map(sns.histplot, "relative_humidity_2m_lag_1h", kde=True, color="blue")
subplot2 = g2.fig.subplots_adjust(top=0.9)

g3 = sns.FacetGrid(lagged_1h_df, col="weather_event", col_wrap=3, height=3)
map3 = g3.map(sns.histplot, "soil_temp_0_to_7cm_lag_1h", kde=True, color="blue")
subplot3 = g3.fig.subplots_adjust(top=0.9)

plt.show()
```

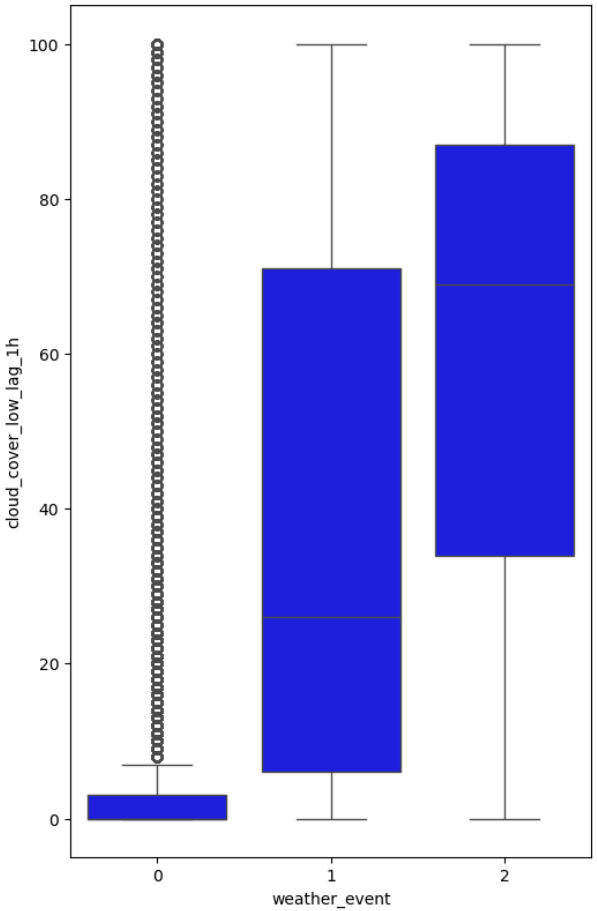

weather_event = 0
weather_event = 1
weather_event = 2
Count
0
250
500
750
1000
1250
0
10
20
30
40
temperature_2m_lag_1h
weather_event = 0
weather_event = 1
weather_event = 2
Count
0
250
500
750
1000
1250
20
40
60
80
100
relative_humidity_2m_lag_1h

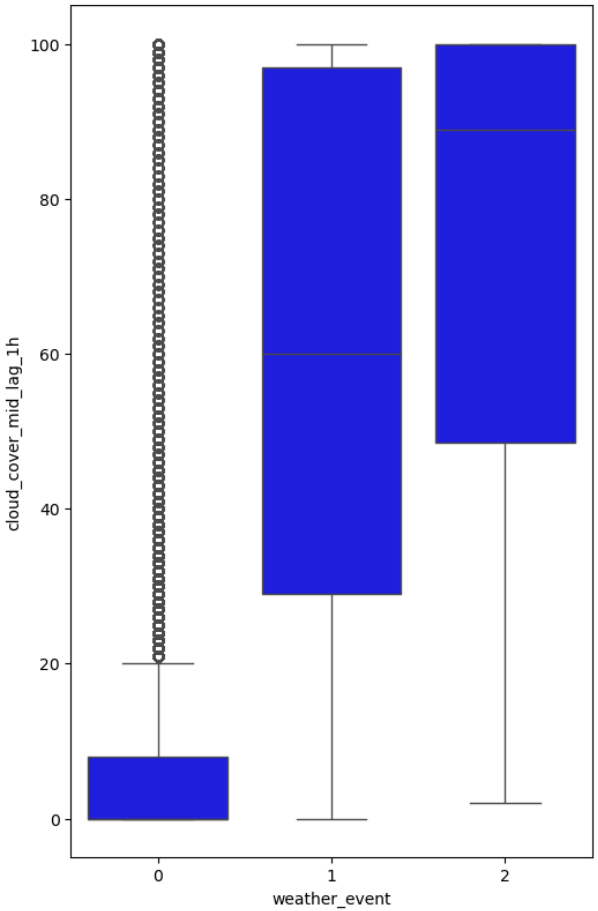


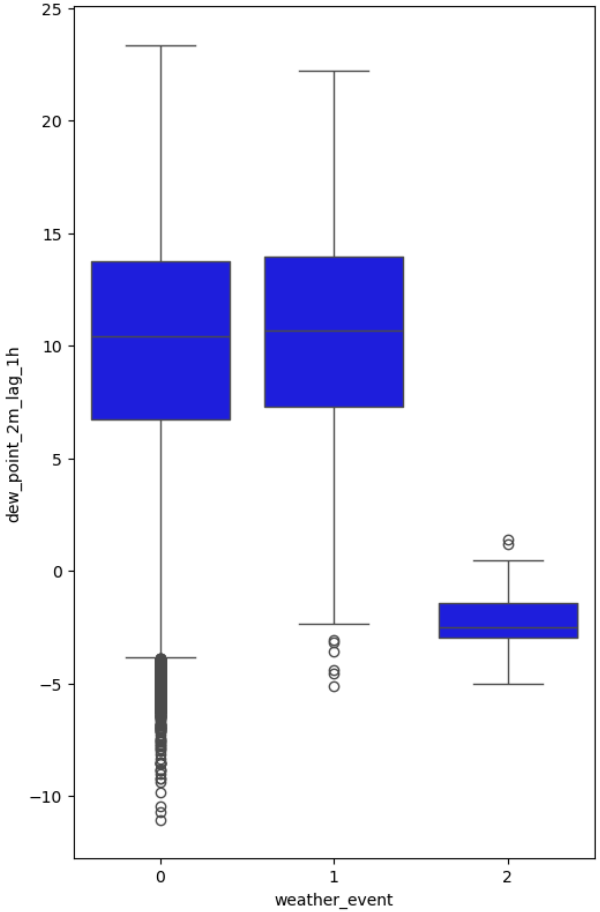
Detecting outliers

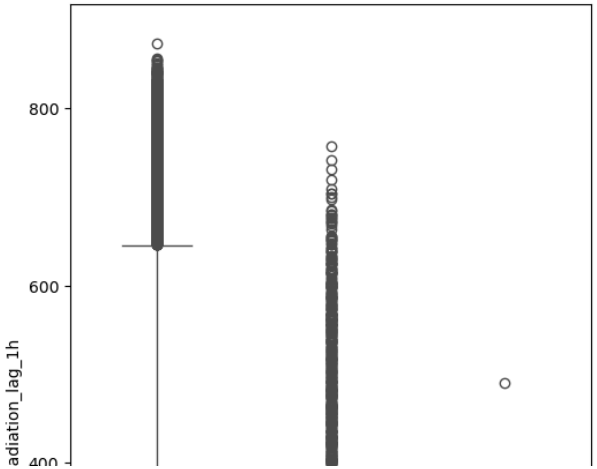
In [51]:

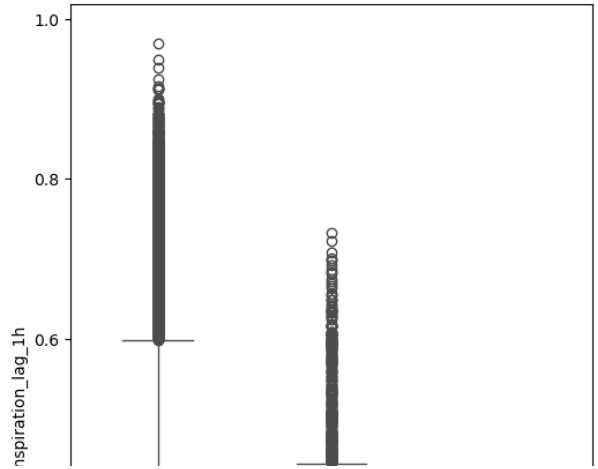
```
import seaborn as sns
import matplotlib.pyplot as plt

# List of columns to plot
columns_to_plot = lagged_1h_df.columns.difference(['date', 'month_sin', 'month_cos', 'weather_event'])

figure = plt.figure(figsize=(16, 40))

# Loop through each column and create a boxplot with weather_event grouping
for i, col in enumerate(columns_to_plot, start=1):
    subplot = plt.subplot(len(columns_to_plot)//3 + 1, 3, i)  # Define a 3-column layout
    boxplot = sns.boxplot(data=lagged_1h_df, x='weather_event', y=col, color='blue')  # Grouped by weather_event

# Adjust the layout
layout = plt.tight_layout()
plt.show()
```

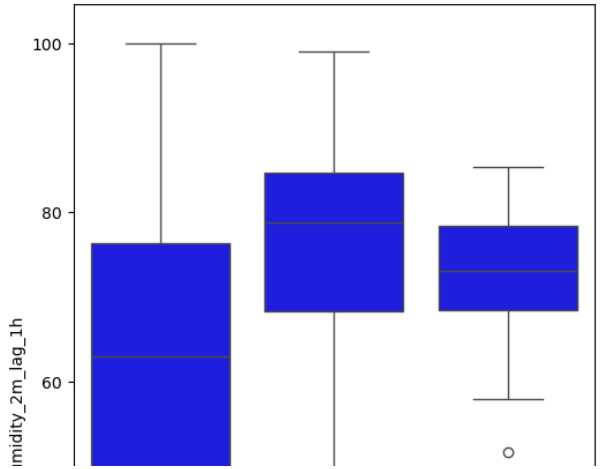

cloud_cover_low_lag_1h
weather_event

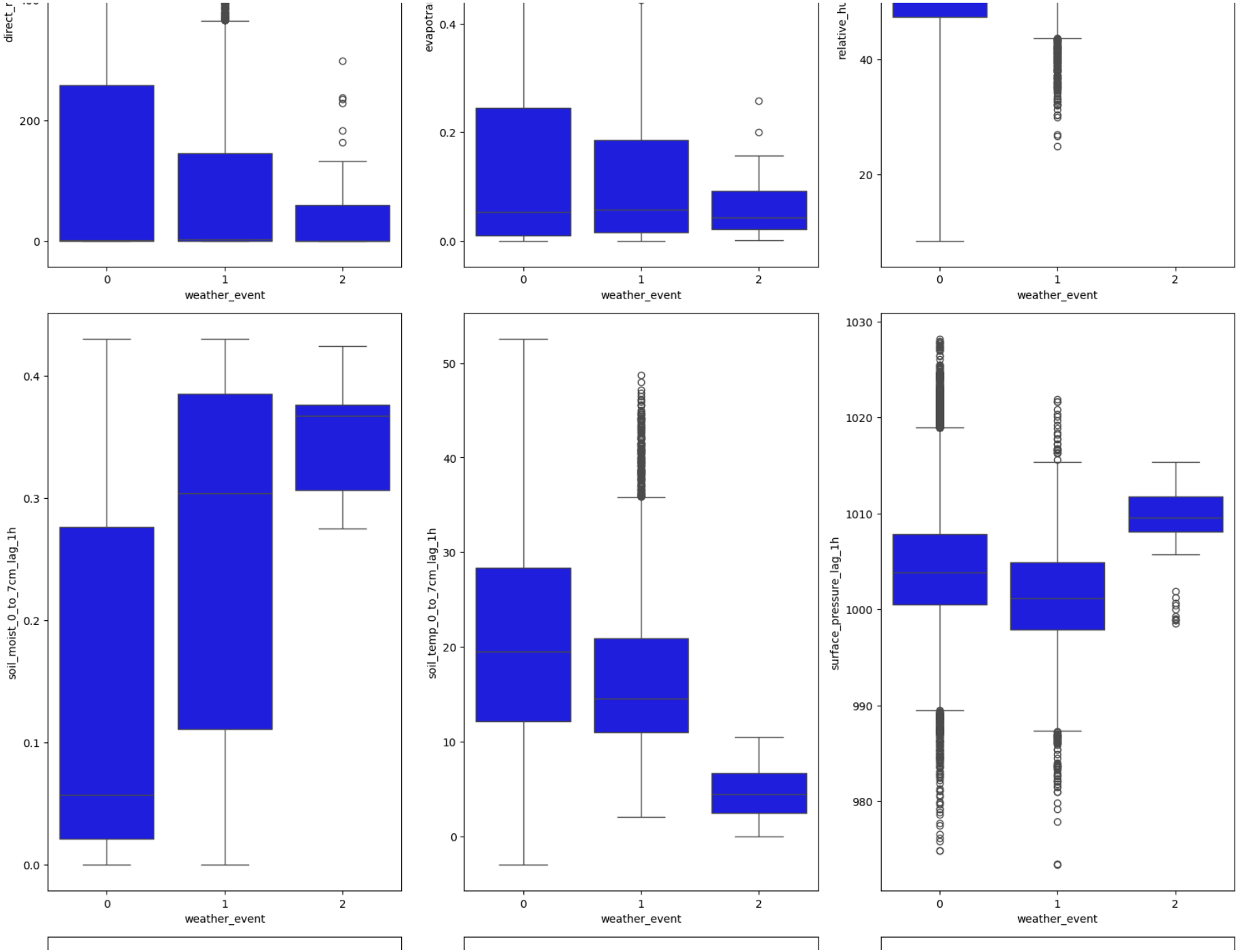

cloud_cover_mid_lag_1h
weather_event

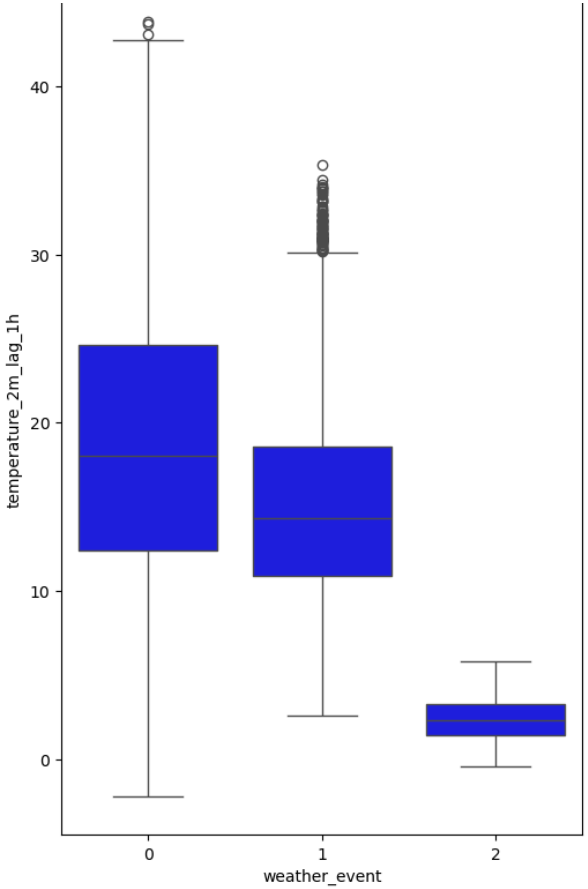

dew_point_2m_lag_1h
weather_event

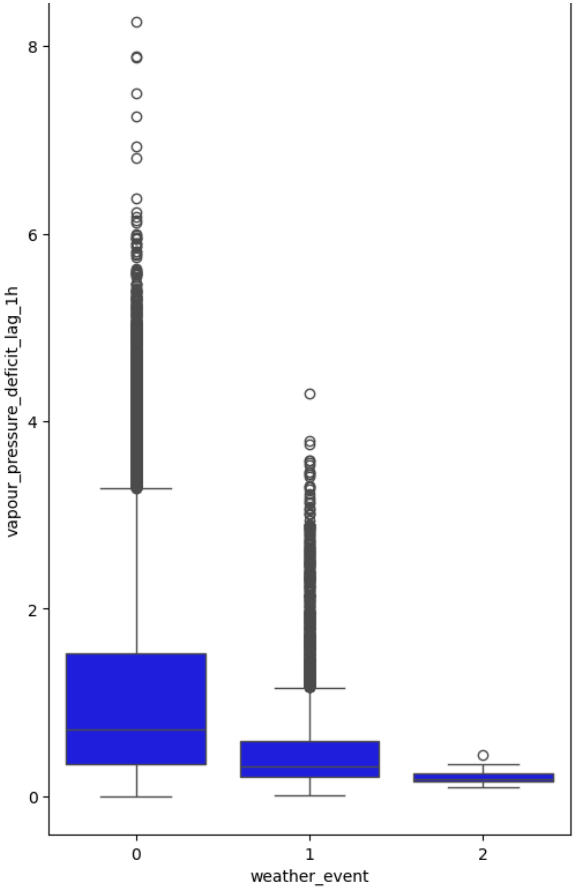

adiation_lag_1h

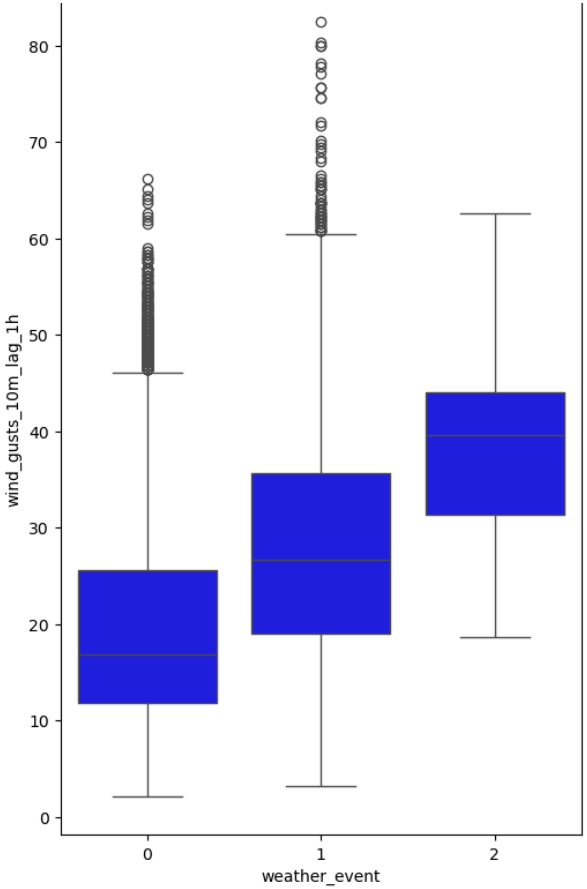

nspiration_lag_1h

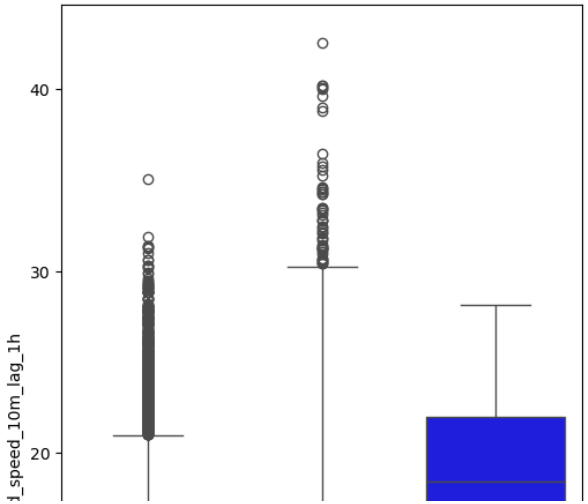

midity_2m_lag_1h

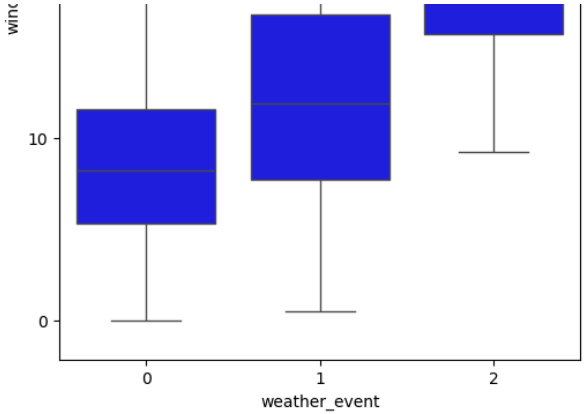

direct_r
200
0
0
1
2
weather_event
evapotra
0.4
0.2
0.0
0
1
2
weather_event
relative_hu
40
20
0
1
2
weather_event
soil_moist_0_to_7cm_lag_1h
0.4
0.3
0.2
0.1
0.0
0
1
2
weather_event
soil_temp_0_to_7cm_lag_1h
50
40
30
20
10
0
0
1
2
weather_event
surface_pressure_lag_1h
1030
1020
1010
1000
990
980
0
1
2
weather_event


temperature_2m_lag_1h
weather_event
0
1
2
0
10
20
30
40


vapour_pressure_deficit_lag_1h
weather_event
0
1
2
0
2
4
6
8


wind_gusts_10m_lag_1h
weather_event
0
1
2
0
10
20
30
40
50
60
70
80


d_speed_10m_lag_1h
20
30
40



Viewing the above box plots makes the presence of outliers more than obvious. Whether these values represent measurement errors, anomalies, or valid extreme observations is uncertain.

With the **lack of fundamental domain knowledge**, access to the instruments used for the measurements, or the procedures followed, makes any kind of inference on the outliers impossible and arbitrarily removing or modifying them inserts more uncertainty in the project.

For this reason, it is better to retain all records, since even a simple action like removing the outliers is very likely to distort the distributions, insert more uncertainty in the project, introducing bias to the modeling process and alter the model's performance to unknown directions.
We will try to mitigate their effect by using tools like robust scaling and Yeo-Johnson transformations.

Modeling process

During modeling phase the metric that will be used to evaluate the performance of the models as well as to facilitate the comparison between them is `balanced accuracy`.

Balanced accuracy is the average of the recall (sensitivity) for each class, treating each class equally regardless of its frequency.
Unlike standard accuracy, which in imbalanced datasets is biased towards the majority class and gives a false impression about the model's performance, balanced accuracy metric provides a more fair view of how well the model performs across all classes and illustrate the overall model performance.

For a heavily imbalanced dataset, balanced accuracy will prevent the majority class from dominating the metric, allowing us to see if the model truly performs well on the minority classes, and not only on the majority class.

You can find detailed information on evaluation metrics used for both classification and regression tasks in the **official documentation**

Logistic Regression Essentials

Multinomial Logistic Regression is a classification algorithm used when the target variable is categorical and has more than two possible classes.
Unlike binary logistic regression, which uses the sigmoid function, multinomial logistic regression uses the softmax function to compute the probability of each class.

In multinomial logistic regression, we model the probability of a data point belonging to one of multiple possible classes. For example, we might predict weather conditions such as "No Precipitation," "Rain," or "Snowfall."
In scikit-learn, multinomial logistic regression by default outputs a predicted class y for each data point, selecting the class with the highest probability.

How the Algorithm Works:

Input Features and Coefficients:
The model takes the input features (independent variables, X) and initializes a vector of coefficients (β) for each class.
These coefficients are initially set to small random values or zeros and will be adjusted during the training process by the solver (the optimization algorithm).

Understanding Logits (Log Odds):
In logistic regression, a core concept is the logit, also known as the log odds and represents a way of expressing probabilities.
For a given class k, the odds are defined as the ratio of the probability of that class occurring to the probability of it not occurring:

$$\text{odds}_k = \frac{P(y = y_k \mid X)}{1 - P(y = y_k \mid X)}$$

And consequently, for a given class k, the logit is defined as the natural logarithm of the odds of that class occurring.
This transformation maps probabilities (which range from 0 to 1) to logits (which range from $-\infty$ to $+\infty$), making it possible to create a linear relationship between the input features and the predicted outcome y.

Logit Calculation:
For each class k, the logit is computed as a linear combination of the input features and their corresponding coefficients:

$$\text{logit}_k = \beta_{0,k} + \beta_{1,k}x_1 + \beta_{2,k}x_2 + \cdots + \beta_{n,k}x_n$$

The model calculates a logit for each class, and this is the raw score before probabilities are computed. Applying this and you end up in our project with 3 logits for every record in your dataset. (1 logit for the 'no precipitation class/label, 1 logit for the 'rain' class/label and 1 logit for the 'snowfall' class/label)

Softmax Transformation:
The logits are then passed through the softmax function, which converts them into probabilities for each class.
The softmax function also ensures that all probabilities are between 0 and 1, and that the total probability across all classes equals 1:

$$P(y = k \mid X) = \frac{\exp(\text{logit}_k)}{\sum_{j=1}^{K} \exp(\text{logit}_j)}$$

So after the Softmax transformation every record in the dataset has been assigned with a probability for each class.
In our case the model predicts whether there will be 'No Precipitation', 'Rain', or 'Snowfall', and it calculates three logits (one for each class). These logits are then transformed into probabilities using the softmax function and the model assigns the class with the highest probability to the target variable y.
For example, a record in our dataset could end up having the following probabilities:

- No Precipitation: 0.60
- Rain: 0.30
- Snowfall: 0.10

Then the model would predict the class 'No Precipitation' for the target variable y, since it has the highest probability.

Loss/Error Function (Cross-Entropy):

The cross-entropy loss function measures the error between the predicted probabilities and the true class labels. It's calculated as:

$$L = -\sum_{k=1}^{K} y_k \log(P(y = k \mid X))$$

This loss is used as feedback to the model. The closer the predicted probability is to the true label, the smaller the loss.
To make it more concrete we will use the above example where the model predicted 'No precipitation' and suppose that the true label was 'rain'.
In this example, since the true label is Rain, we have $y_{\text{rain}} = 1$, and for the other classes, $y_{\text{no precipitation}} = 0$ and $y_{\text{snowfall}} = 0$.

Applying the formula of the loss function shown above, and considering only the probability for the correct class (Rain), as the other terms will be multiplied by 0, we have: $L = -(1 \times \log(0.30)) - (0 \times \log(0.60)) - (0 \times \log(0.10))$ which leads to:
$L = -\log(0.30) \approx -(-0.523) = 0.523$.
This is the loss for this particular record (0.523).

At this point the solver takes charge and adjusts the coefficients in the direction to minimize the loss function.

Solver and Coefficient Adjustment:
The solver is the optimization algorithm responsible for adjusting the β vectors (the coefficients). It iteratively updates the coefficients to reduce the loss, stopping when the change in the loss function between iterations falls below a certain threshold (tol) or after a maximum number of iterations (max_iter) is reached.

All the above come into place in the scikit-learn library where the model/algorithm is implemented. You can find all the details you may need in the official documentation of the library **here**

Standard Logistic Regression

We will begin the modeling phase by instantiating a Logistic Regression model with default settings, without any adjustments for class imbalance. This model will serve as our baseline, providing a reference point against which, we can compare other potentially more complex, adjusted models and offer a straightforward and intuitive way to compare different model implementations.

- separate the dataset into input space X (the predictor variables) and to output space y (the target/response) variable

In [61]:
```
X = lagged_1h_df.drop(columns=['date', 'weather_event'])
y = lagged_1h_df['weather_event']
```

- data is split into a training set (70%) and a test set (30%) using stratified sampling to maintain the proportions of the classes for the target variable y in both sets. This ensures that the model is trained and evaluated with the same label distribution. The random state is set for reproducibility.

In [64]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)
```

After splitting the data into train and test sets, we are ready to build the model's pipeline.
In this standard logistic regression approach, the pipeline will include two preprocessing steps:

- Scaling the Data:

We will use the RobustScaler() to scale the data, as it is resilient to outliers and ensures features are on a similar scale, which is important for logistic regression.

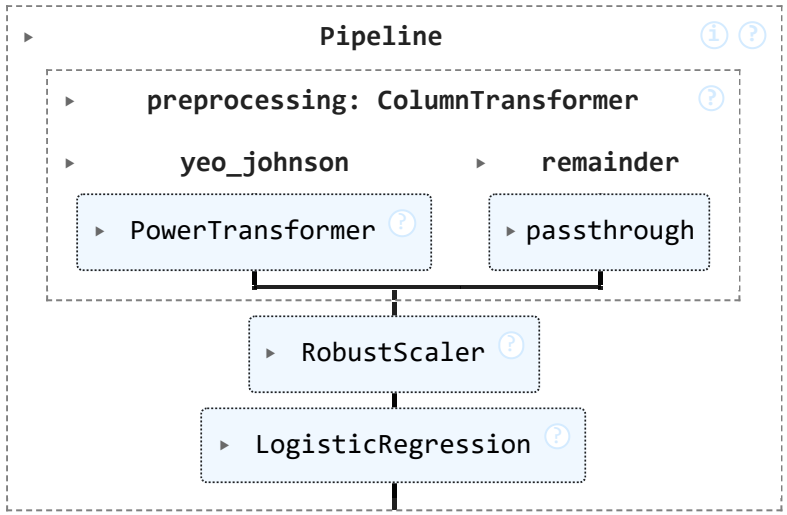
- Model Instantiation:

We will instantiate the LogisticRegression() class, and set the `multi_class` argument to 'multinomial'.

In [67]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer, RobustScaler
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# Define the preprocessing step for Yeo-Johnson transformation
preprocessor = ColumnTransformer([
    ('yeo_johnson', PowerTransformer(method='yeo-johnson'), variables_to_transform),
], remainder='passthrough')  # Leave other features untouched

# Define the pipeline
standard_pipeline = Pipeline([
    ('preprocessing', preprocessor),   # Step 1: Yeo-Johnson transformation for specific variables
    ('scaling', RobustScaler()),       # Step 2: Scale all features
    ('standard_model', LogisticRegression(
        max_iter=1500,
        random_state=42,
```

```
        solver='lbfgs',
        n_jobs=-1
    ))
])


import time
start = time.time()

# Step 3: Fit the pipeline to the training data
standard_pipeline.fit(X_train, y_train)

# Execution time and number of iterations for the solver to converge
execution_time = time.time() - start
n_iterations = standard_pipeline['standard_model'].n_iter_
print(f"Execution time: {execution_time:.2f} seconds")
print(f"Number of iterations to converge: {n_iterations}")
```

Out[67]:




```
Execution time: 1.87 seconds
Number of iterations to converge: [76]
```

- extracting the predicted values of y by passing the test set features (X_test) into the trained model

In [ ]:
```
y_pred_standard = standard_pipeline.predict(X_test)
```

Model Evaluation

Before evaluating the model on the test set (the truly "*unseen*" data), we will use a 10-fold cross-validation on the train dataset.
This step provides an estimate of the model's performance, consistency, and stability across different subsets of the train data, helping us understand what to expect from the model.

The scoring metric used for assessment is Balanced Accuracy, which is defined as the average recall across all classes. This is particularly useful in heavily imbalanced datasets when the primary concern is ensuring the model correctly identifies actual positive instances for each class, reducing the risk of missing Rain and Snowfall events.
In contrast, F1-macro and PR AUC-macro, which treat all classes equally, can understate overall performance if the model struggles with minority classes. Meanwhile, F1-weighted and PR AUC-weighted tend to be overly optimistic, as they are heavily influenced by the majority class. Balanced Accuracy avoids these pitfalls by focusing explicitly on recall and ensuring that all classes contribute equally to the final score, making it a more reliable metric in this context.

You can find more information about the cv evaluation of the train data as well as the most suitable metrics to use in the links below:
https://scikit-learn.org/stable/modules/cross_validation.html
https://scikit-learn.org/stable/modules/model_evaluation.html
https://machinelearningmastery.com/tour-of-evaluation-metrics-for-imbalanced-classification/

CV evaluation

- Setting up a 10-fold cross-validation with stratified sampling to ensure that each fold has the same proportion of classes as the original dataset. Setting the random state to ensure reproducibility.

In [76]:
```
from sklearn.model_selection import StratifiedKFold
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
```

- run the cv assessment on the train data using the 10-folds created in the previous step. Printing the mean value of the metric across all folds along with its standard deviation.

In [79]:
```
from sklearn.model_selection import cross_val_score
logreg_standard_cv = cross_val_score(standard_pipeline, X_train, y_train, cv=stkf, scoring='balanced_accuracy', n_jobs=-1)
print(logreg_standard_cv)
print(f'\nThe mean value of balanced_accuracy is {logreg_standard_cv.mean().round(3)}')
print(f'\nThe standard deviation of balanced_accuracy is {logreg_standard_cv.std().round(3)}')
```

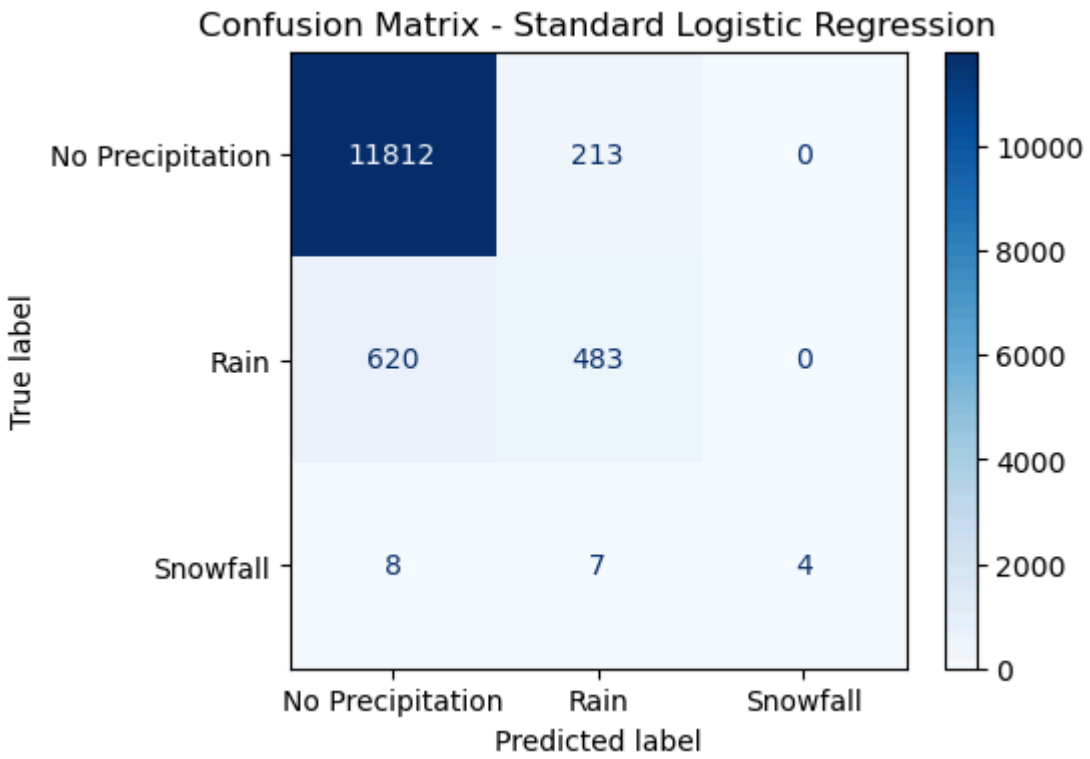
```
[0.5618898  0.63769405 0.55289037 0.50519408 0.48232295 0.5417021
 0.63919774 0.5672103  0.55636322 0.48151004]

The mean value of balanced_accuracy is 0.553

The standard deviation of balanced_accuracy is 0.052
```

The cross-validation results indicate that the model consistently performs poorly, likely due to the **unaddressed class imbalance**. We will further explore and justify this observation using the confusion matrix to assess the model's performance on the unseen test dataset across all target classes.

The Confusion Matrix

Generating the confusion matrix to visualize the performance of the model **on the test set** this time.
The confusion matrix shows how well the model predicted each class by displaying the true positive, false positive, true negative, and false negative counts for each class ('No Precipitation', 'Rain', 'Snowfall').

In [84]:
```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
conf_matrix_standard = confusion_matrix(y_test, y_pred_standard)
fig, ax = plt.subplots(figsize=(5, 4))
disp = ConfusionMatrixDisplay(conf_matrix_standard, display_labels=['No Precipitation', 'Rain', 'Snowfall'])
plot = disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
title = plt.title('Confusion Matrix - Standard Logistic Regression')
plt.show()
```



It is evident that the severe imbalance in the target classes significantly impacted the model's performance, as seen in the confusion matrix above.

The model performed very well in predicting instances of the majority class ('No Precipitation'), correctly identifying 11812 out of a total of 12025 instances. However, when it came to the minority classes, the model struggled. For 'Rain' events, it correctly predicted less than half of the instances (483 out of 1,103), and for the 'Snowfall' class, the model captured only 4 out of 19 events.

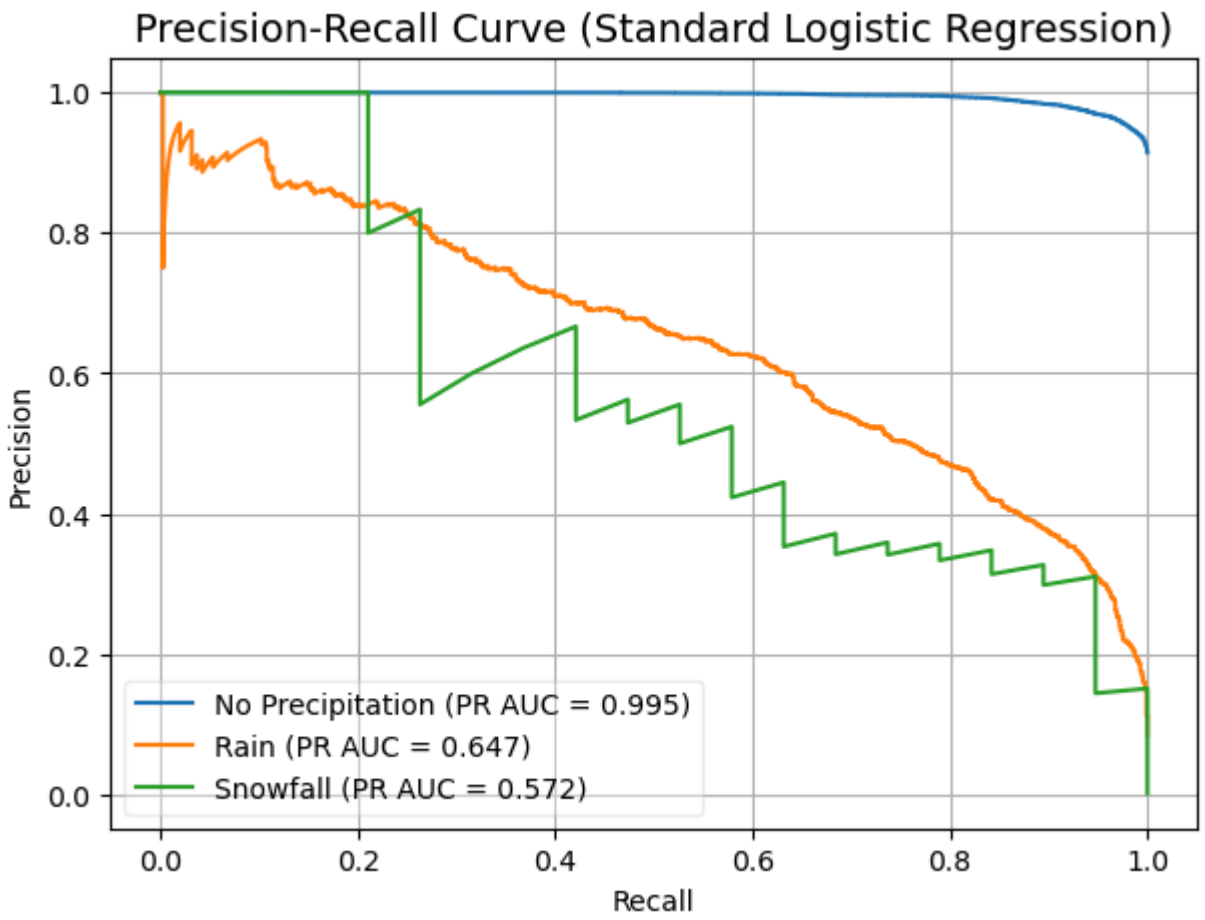
This is a common challenge in classification tasks when dealing with imbalanced datasets, where the model tends to favor the majority class at the expense of the minority classes.

The Precision-Recall Curve

The next step in evaluating the model's performance on the test set is to compute the Precision-Recall (PR) curve for each class: No Precipitation, Rain, and Snowfall. The PR curve visualizes the trade-off between precision and recall across various probability thresholds, making it a valuable tool for assessing models on imbalanced datasets.

This curve is particularly valuable for imbalanced datasets because it highlights how well the model performs on minority classes.
As recall (the proportion of true positive predictions out of all actual positives) increases, the model tends to capture more true positive instances. However, this often comes at the cost of precision (the proportion of true positive predictions out of all predicted positives), as more false positive predictions are introduced.

In [89]:
```
from sklearn.metrics import precision_recall_curve, auc
import matplotlib.pyplot as plt

# Predicted probabilities
y_pred_probs_standard = standard_pipeline.predict_proba(X_test)

# Define class labels
class_labels = ['No Precipitation', 'Rain', 'Snowfall']

# Plot Precision-Recall curves
figure = plt.figure(figsize=(7, 5))
for i, label in enumerate(class_labels):
    precision, recall, _ = precision_recall_curve(y_test == i, y_pred_probs_standard[:, i])
    pr_auc = auc(recall, precision)

    # Plot the precision-recall curve
    curve_plot = plt.plot(recall, precision, label=f'{label} (PR AUC = {pr_auc:.3f})')

# Add labels, title, and legend
xlabel = plt.xlabel('Recall')
ylabel = plt.ylabel('Precision')
title =plt.title('Precision-Recall Curve (Standard Logistic Regression)', fontsize=14)
legend = plt.legend(loc='lower left', framealpha=0.35)
grid = plt.grid(True)
plt.show()
```



The precision-recall curve highlights the poor performance of the model on the minority classes due to severe class imbalance.

While the model performs exceptionally well on 'No Precipitation' instances, this was expected. Even a dummy classifier that always predicts 'No Precipitation' would achieve a precision approximately equal to the majority class proportion (~91.5%). This is a well-known issue in severely imbalanced datasets, where the model becomes heavily biased toward the majority class.

However, when it comes to minority classes ('Rain' and 'Snowfall'), the model struggles to classify them correctly. As it attempts to improve recall, the false positives (FPs) increase, leading to a sharp decline in precision. When recall reaches 1 (capturing all minority instances), precision drops to zero, indicating that perfectly capturing the minority classes comes at the cost of excessive misclassifications.

This is a classic precision-recall trade-off inherent in imbalanced datasets, and its effects are further amplified as class imbalance becomes more severe. This behavior highlights the importance of balancing precision and recall, to achieve optimal performance for each class.

The Classification Report

One more step in evaluating the model is to create a classification report, summarizing key metrics for each class: precision, recall, F1-score, and PR AUC (Precision-Recall Area Under the Curve). The classification report provides a detailed breakdown of these metrics for each class: No Precipitation, Rain, and Snowfall.

Additionally, we calculate overall metrics:

- balanced_accuracy: more suitable for imbalanced datasets compared to simple accuracy.
- f1_macro: calculate the metric for each label and find their unweighted mean. This does not take label imbalance into account.
- pr_auc_macro: the unweighted average of the precision-recall AUC scores computed separately for each class, treating all classes equally.

We transformed the report to a single-row DataFrame so as to make future comparisons with other models easier.

In [ ]:
```
from sklearn.metrics import (
    classification_report, balanced_accuracy_score, f1_score,
    average_precision_score, precision_recall_curve, auc
)

report_dict_standard = classification_report(y_test, y_pred_standard, digits=3, output_dict=True)
y_pred_probs_standard = standard_pipeline.predict_proba(X_test)

# Calculate PR AUC for each class
precision_0, recall_0, _ = precision_recall_curve(y_test == 0, y_pred_probs_standard[:, 0])
pr_auc_0 = auc(recall_0, precision_0)

precision_rain, recall_rain, _ = precision_recall_curve(y_test == 1, y_pred_probs_standard[:, 1])
pr_auc_rain = auc(recall_rain, precision_rain)

precision_snow, recall_snow, _ = precision_recall_curve(y_test == 2, y_pred_probs_standard[:, 2])
pr_auc_snow = auc(recall_snow, precision_snow)

# Create a single row dataframe with the required metrics
report_df_standard = pd.DataFrame({
```

```
    'precision_0': [report_dict_standard['0']['precision']],
    'recall_0': [report_dict_standard['0']['recall']],
    'f1_0': [report_dict_standard['0']['f1-score']],
    'pr_auc_0': [pr_auc_0],

    'precision_rain': [report_dict_standard['1']['precision']],
    'recall_rain': [report_dict_standard['1']['recall']],
    'f1_rain': [report_dict_standard['1']['f1-score']],
    'pr_auc_rain': [pr_auc_rain],

    'precision_snow': [report_dict_standard['2']['precision']],
    'recall_snow': [report_dict_standard['2']['recall']],
    'f1_snow': [report_dict_standard['2']['f1-score']],
    'pr_auc_snow': [pr_auc_snow],
    'balanced_accuracy': [balanced_accuracy_score(y_test, y_pred_standard)],
    'f1_macro': [f1_score(y_test, y_pred_standard, average='macro')],
    'pr_auc_macro': [average_precision_score(y_test, y_pred_probs_standard, average='macro')]
})

print('Standard Logistic Regression')
report_df_standard
```

```
Standard Logistic Regression
```

Out[ ]:

	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_snow	pr_au
0	0.949518	0.982287	0.965624	0.994828	0.687055	0.437897	0.534884	0.647345	1.0	0.210526	0.347826	

The learning curve

Before interpreting a learning curve to gain insights into a model's performance, we must first understand what it represents.

A learning curve illustrates how a model's performance evolves as the dataset size increases, helping us diagnose underfitting, overfitting, and generalization ability. In our case, we use log-loss to track how the error in predictions changes as more training data is provided.

Each point on the curve represents the average performance across cross-validation folds for a specific training size, offering an estimate of how well the model performs with different data volumes. This allows us to observe whether model performance improves, deteriorates, or plateaus with additional data.

In the plot, we have two curves: one (yellow) represents the average metric score on the training dataset, and the other (purple) represents the average metric score on the validation/test dataset.

- If both curves are low relative to the y-axis and close to each other, the model may not be complex enough to capture data patterns and is likely underfitting.
- If the training curve is high but the validation curve is low, the model might be memorizing the training data without generalizing well, indicating overfitting.
- If both curves converge at a high level, the model is likely to perform well and generalizing effectively.

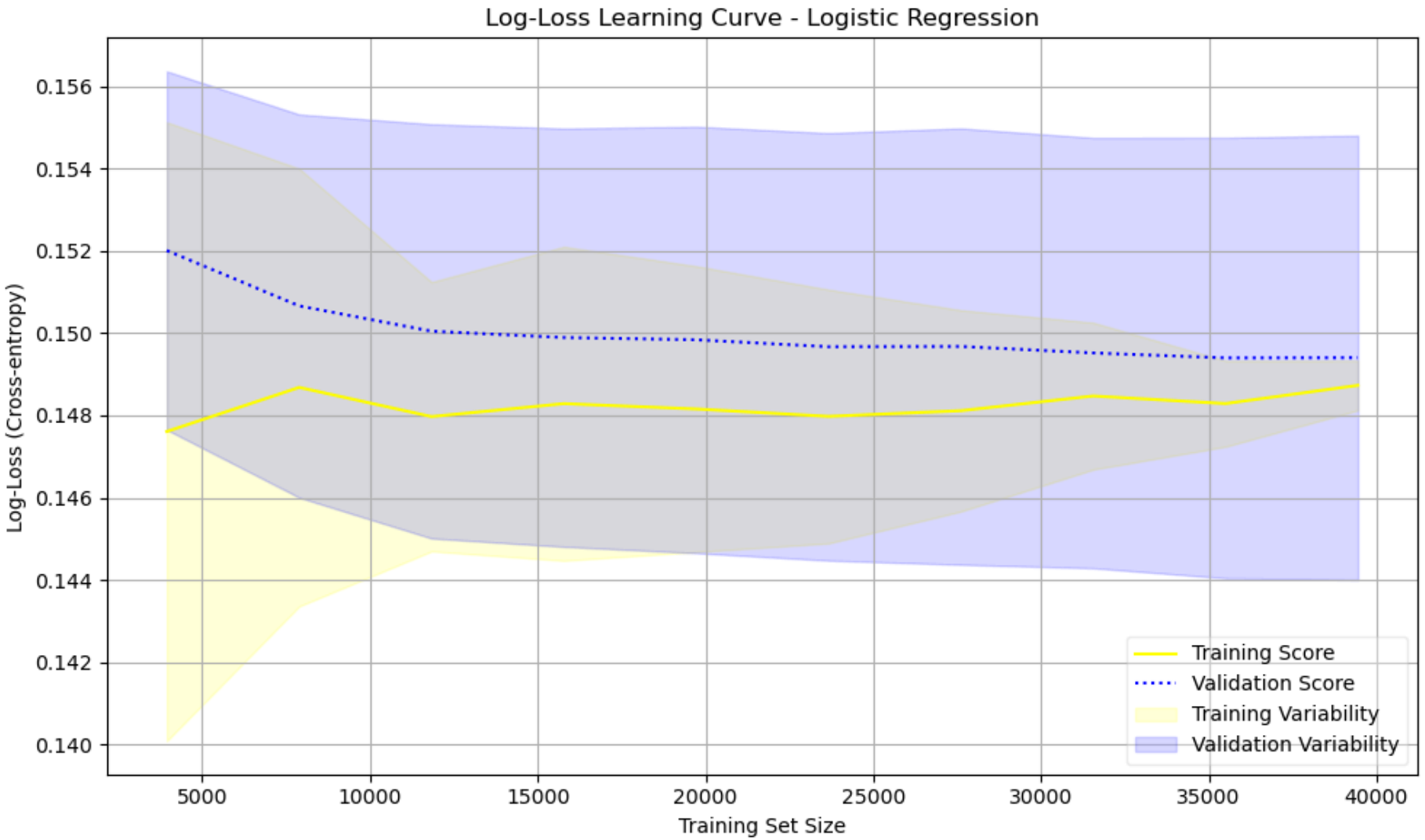
It is also important to observe whether the curves increase, plateau, or fluctuate as more data is fed to the model, as this indicates how additional data impacts performance.

Finally, you shouldn't expect the performance metric on the test set from stratified cross-validation on the full dataset to exactly match the final point on the learning curve. Due to data randomness and differences in train-validation splits, the training and validation sets in the final point of the learning curve may differ from those used in a cross-validation on the full dataset, leading to slight variations in metric results. The learning curve's final point represents the average of cross-validated scores across multiple folds, while the test set score is a single evaluation on a fixed data split.

However, if the difference between the two metrics is significant, it's worth investigating further. Potential issues could include model instability due to high complexity, overfitting to specific patterns in the data and more.

In [98]:
```
from sklearn.model_selection import StratifiedKFold
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

from sklearn.model_selection import learning_curve
train_sizes, train_scores, test_scores = learning_curve(
    standard_pipeline, X, y, cv=stkf, scoring='neg_log_loss', n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), shuffle=True, random_state=42
)

train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

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

```
plot1 = plt.plot(train_sizes, train_mean, label='Training Score', color='yellow')
plot2 = plt.plot(train_sizes, test_mean, label='Validation Score', color='blue', linestyle=':')
ce1_fill = plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color='yellow',
                            alpha=0.15, label='Training Variability')
ce2_fill = plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color='blue',
                            alpha=0.15, label='Validation Variability')

xlabel = plt.xlabel('Training Set Size')
ylabel = plt.ylabel('Log-Loss (Cross-entropy)')
title = plt.title('Log-Loss Learning Curve - Logistic Regression')
legend = plt.legend(loc='lower right', framealpha=0.35)
grid = plt.grid(True)
plt.tight_layout()
plt.show()
```



Inferences from the Learning Curve Plot

The log-loss of the training and validation sets converges around 0.149 when the full dataset is used to train the model. The small gap between the two curves indicates no signs of overfitting or underfitting, suggesting that the model is well-calibrated to the data.

However, the noticeable variance in validation loss, meaning that performance varies across different validation splits, suggests that the model struggles to generalize consistently on unseen data. This could be due to data complexity and severe class imbalance.

Weighted Logistic Regression

The issue of severe class imbalance in the target variable was evident during the EDA process and became even more pronounced when we fitted the Standard Logistic Regression model to the data. The unbalanced classes in the target variable significantly impacted the model's ability to predict minority class instances effectively.

To address this, we will fit a Logistic Regression model that attempts to balance the classes of the target variable.

This iteration handles class imbalance by automatically adjusting the model to account for the uneven distribution of classes. (this is done by setting the argument `class_weight='balanced'`).
In this *balanced mode*, the model automatically adjusts the weights of each class inversely proportional to their frequencies in the training data. This adjustment ensures that the minority classes receive more importance during training, without modifying the dataset itself.

By implementing this weighting strategy, the model is expected to improve its ability to predict **underrepresented classes**, potentially leading to better recall and precision for these categories and finally better overall performance.

- separate the dataset into input space X (the predictor variables) and output space y (the target/response) variable

In [104…
```
X = lagged_1h_df.drop(columns=['date', 'weather_event'])
y = lagged_1h_df['weather_event']
```

- data is split into a training set (70%) and a test set (30%) using stratified sampling to maintain the proportions of the target variable y in both sets. This ensures that the model is trained and evaluated with the same label distribution. The random state is set for reproducibility.

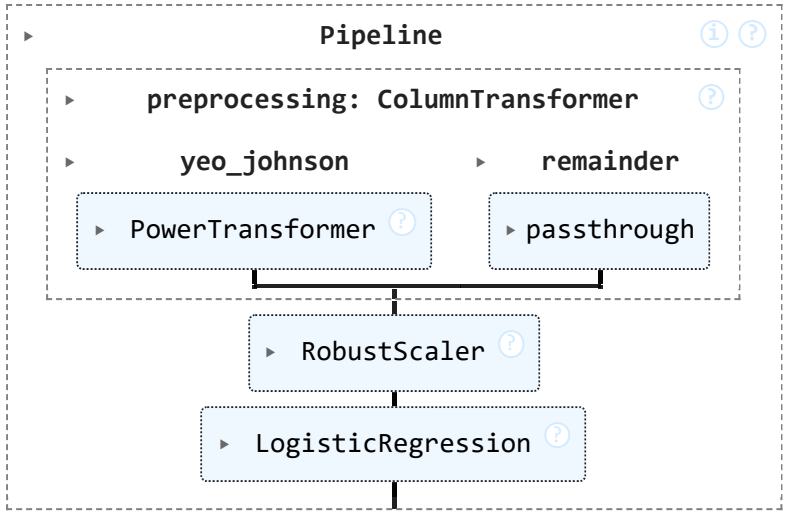
In [107…
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)
```

- Building and Fitting the Pipeline

In [110…

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer, RobustScaler
from sklearn.pipeline import Pipeline

# Define the preprocessing step for Yeo-Johnson transformation
preprocessor = ColumnTransformer([
    ('yeo_johnson', PowerTransformer(method='yeo-johnson'), variables_to_transform),
], remainder='passthrough')  # Leave other features untouched

# Define the pipeline
weighted_pipeline = Pipeline([
    ('preprocessing', preprocessor),   # Step 1: Yeo-Johnson transformation for specific variables
    ('scaling', RobustScaler()),       # Step 2: Scale all features
    ('weighted_model', LogisticRegression(
        class_weight='balanced',
        max_iter=1500,
        random_state=42,
        solver='lbfgs',
        n_jobs=-1
    ))
])


import time
start = time.time()

# Step 3: Fit the pipeline to the training data
weighted_pipeline.fit(X_train, y_train)

# Execution time and number of iterations for the solver to converge
execution_time = time.time() - start
print(f"Execution time: {execution_time:.2f} seconds")
weighted_model_fitted = weighted_pipeline['weighted_model']
print(f"Number of iterations for convergence: {weighted_model_fitted.n_iter_}")
```

Out[110…




```
Execution time: 1.62 seconds
Number of iterations for convergence: [141]
```

- extracting the predicted values of y by passing the test set features (X_test) into the trained model

In [ ]:
```
y_pred_weighted = weighted_pipeline.predict(X_test)
```

Model Evaluation

CV evaluation

In [117…
```
from sklearn.model_selection import StratifiedKFold, cross_val_score
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
```

In [ ]:
```
logreg_weighted_cv = cross_val_score(weighted_pipeline, X_train, y_train, cv=stkf, scoring='balanced_accuracy', n_jobs=-1)
print(logreg_weighted_cv)
print(f'\nThe mean value of balanced_accuracy on the cv evaluation is {logreg_weighted_cv.mean().round(3)}')
print(f'\nThe standard deviation of balanced_accuracy on the cv evaluation is {logreg_weighted_cv.std().round(3)}')
```

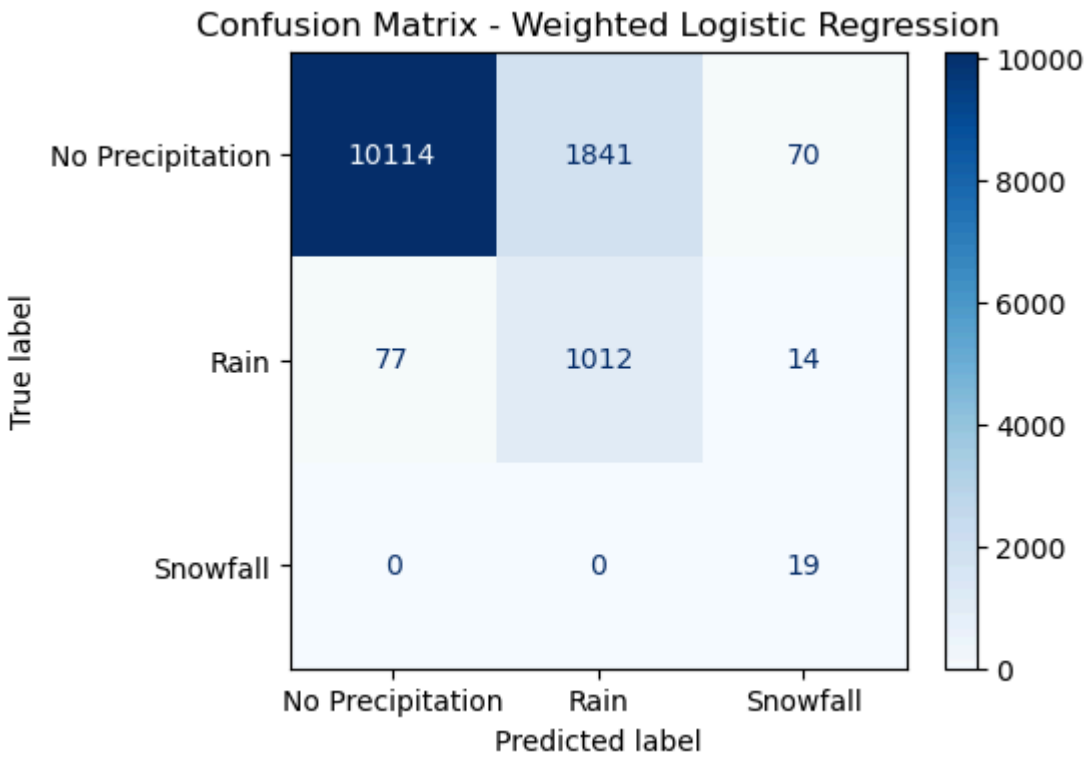
```
[0.83215783 0.91376821 0.91682082 0.90901931 0.92497871 0.91484987
 0.90303851 0.91671173 0.91410549 0.92139728]

The mean value of balanced_accuracy on the cv evaluation is 0.907

The standard deviation of balanced_accuracy on the cv evaluation is 0.025
```

The Confusion Matrix

- Generating the confusion matrix to visualize the performance of the model on the test set this time. The confusion matrix shows how well the model predicted each class by displaying the true positive, false positive, true negative, and false negative counts for each class ('No Precipitation', 'Rain', 'Snowfall').

In [123…

```python
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
conf_matrix_weighted = confusion_matrix(y_test, y_pred_weighted)
fig, ax = plt.subplots(figsize=(5, 4))
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_weighted, display_labels=['No Precipitation', 'Rain', 'Snowfall'])
plot = disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
title = plt.title('Confusion Matrix - Weighted Logistic Regression')
plt.show()
```



Handling class imbalance, we achieved significantly better results for the minority classes where the model correctly captured 1012 'Rain' events out of a total of 1103 instances, and also captured all 19 'Snowfall' instances. Additionally, misclassifications of 'Rain' as 'No Precipitation' or 'Snowfall' decreased substantially. For example, the model incorrectly predicted 'No Precipitation' instead of 'Rain' only 77 times, compared to 620 times in the unweighted standard approach.

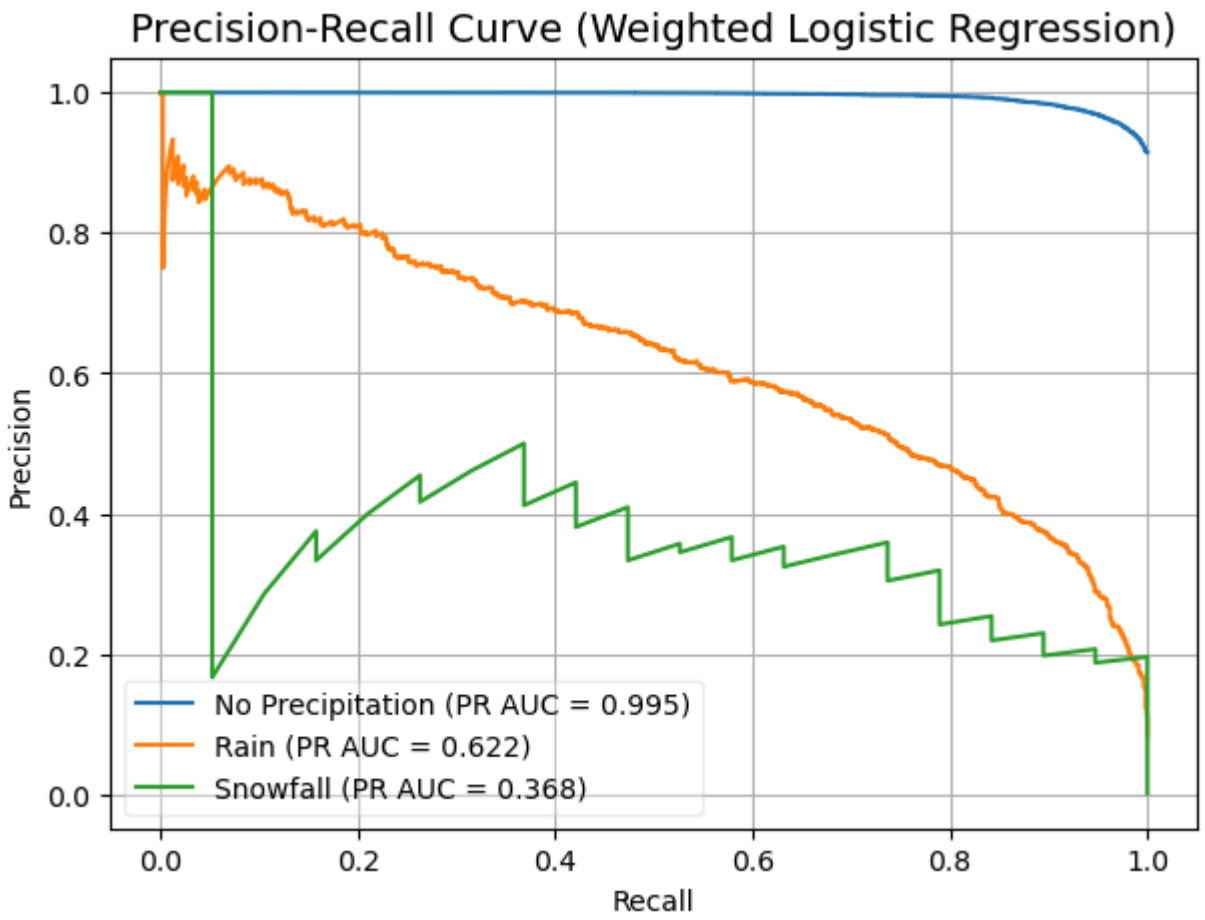
However, this improvement in recall for the minority classes came at the cost of precision for the majority class ('No Precipitation'). The true positives for 'No Precipitation' dropped from 11,812 to 10114, and the number of actual 'No Precipitation' instances misclassified as 'Rain' increased significantly—from 213 misclassifications in the standard model to 1841 in the weighted approach!

This also demonstrates the precision-recall trade-off mentioned earlier: by focusing on improving recall for the minority classes, the model sacrifices precision, particularly for the majority class.

The Precision-Recall Curve

In [ ]:

```
from sklearn.metrics import precision_recall_curve, auc
import matplotlib.pyplot as plt

# Predicted probabilities
y_pred_probs_weighted = weighted_pipeline.predict_proba(X_test)

# Define class labels
class_labels = ['No Precipitation', 'Rain', 'Snowfall']

# Plot Precision-Recall curves
figure = plt.figure(figsize=(7, 5))  # Slightly larger figure for clarity
for i, label in enumerate(class_labels):
    precision, recall, _ = precision_recall_curve(y_test == i, y_pred_probs_weighted[:, i])
    pr_auc = auc(recall, precision)

    # Plot the precision-recall curve
    curve_plot = plt.plot(recall, precision, label=f'{label} (PR AUC = {pr_auc:.3f})')

# Add labels, title, and legend
xlabel = plt.xlabel('Recall')
ylabel = plt.ylabel('Precision')
title =plt.title('Precision-Recall Curve (Weighted Logistic Regression)', fontsize=14)
legend = plt.legend(loc='lower left', framealpha=0.35)
grid = plt.grid(True)
plt.show()
```



The Classification Report

In [ ]:
```
from sklearn.metrics import (
    classification_report, balanced_accuracy_score, f1_score,
    average_precision_score, precision_recall_curve, auc
)
report_dict_weighted = classification_report(y_test, y_pred_weighted, digits=3, output_dict=True)

# Calculate PR AUC for each class
precision_0, recall_0, _ = precision_recall_curve(y_test == 0, y_pred_probs_weighted[:, 0])
pr_auc_0 = auc(recall_0, precision_0)
```

```
precision_rain, recall_rain, _ = precision_recall_curve(y_test == 1, y_pred_probs_weighted[:, 1])
pr_auc_rain = auc(recall_rain, precision_rain)

precision_snow, recall_snow, _ = precision_recall_curve(y_test == 2, y_pred_probs_weighted[:, 2])
pr_auc_snow = auc(recall_snow, precision_snow)

# Create a single row dataframe with the required metrics
report_df_weighted = pd.DataFrame({
    'precision_0': [report_dict_weighted['0']['precision']],
    'recall_0': [report_dict_weighted['0']['recall']],
    'f1_0': [report_dict_weighted['0']['f1-score']],
    'pr_auc_0': [pr_auc_0],

    'precision_rain': [report_dict_weighted['1']['precision']],
    'recall_rain': [report_dict_weighted['1']['recall']],
    'f1_rain': [report_dict_weighted['1']['f1-score']],
    'pr_auc_rain': [pr_auc_rain],

    'precision_snow': [report_dict_weighted['2']['precision']],
    'recall_snow': [report_dict_weighted['2']['recall']],
    'f1_snow': [report_dict_weighted['2']['f1-score']],
    'pr_auc_snow': [pr_auc_snow],
    'balanced_accuracy': [balanced_accuracy_score(y_test, y_pred_weighted)],
    'f1_macro': [f1_score(y_test, y_pred_weighted, average='macro')],
    'pr_auc_macro': [average_precision_score(y_test, y_pred_probs_weighted, average='macro')]
})

print('Weighted Logistic Regression')
report_df_weighted
```

```
Weighted Logistic Regression
```

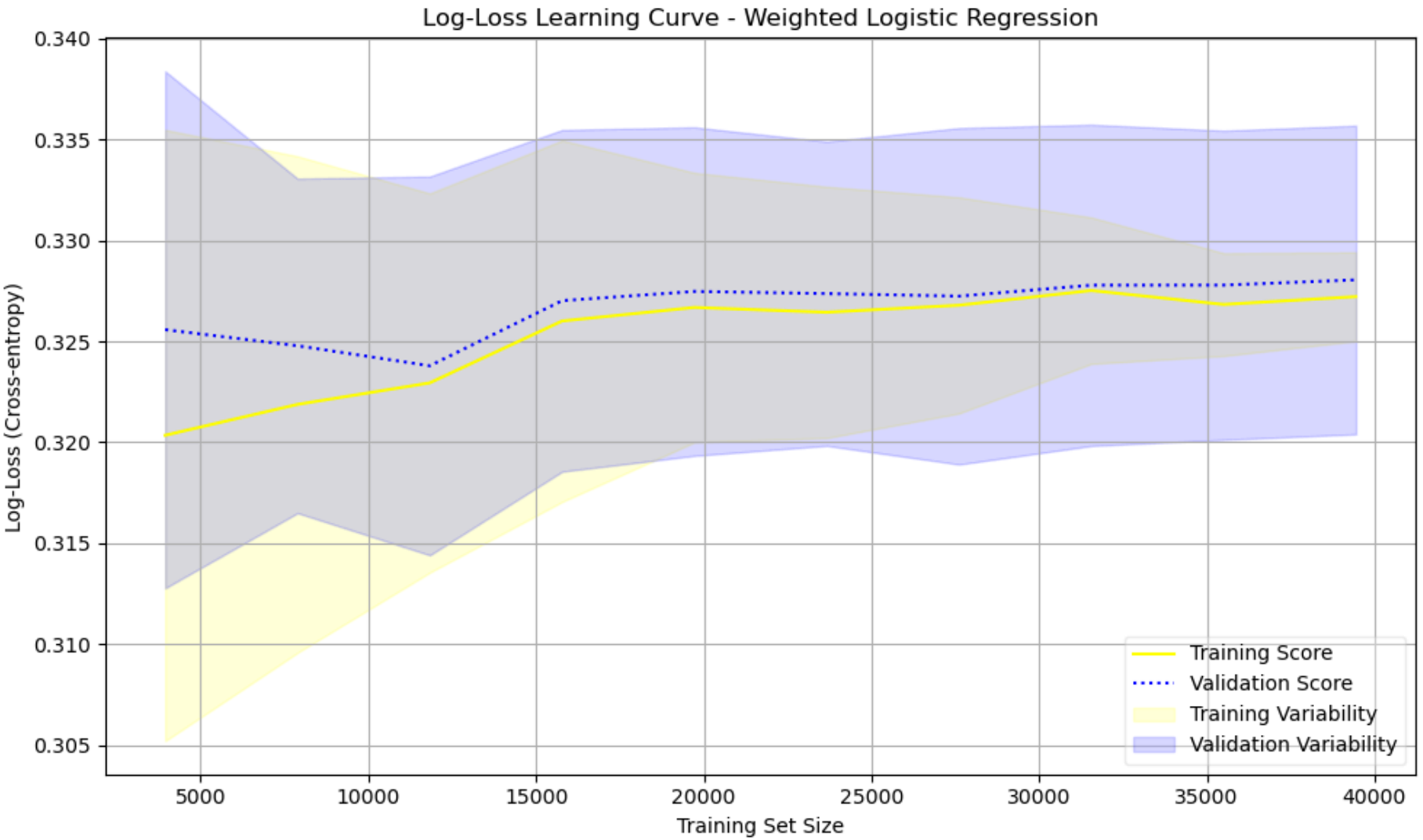
Out[ ]:

	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_snow	pr_au
0	0.992444	0.841081	0.910515	0.994658	0.354714	0.917498	0.511628	0.622406	0.184466	1.0	0.311475	

The learning curve

In [133…

```
from sklearn.model_selection import learning_curve
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

train_sizes, train_scores, test_scores = learning_curve(
    weighted_pipeline, X, y, cv=stkf, scoring='neg_log_loss', n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), shuffle=True, random_state=42
)

train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

figure = plt.figure(figsize=(10, 6))
plot1 = plt.plot(train_sizes, train_mean, label='Training Score', color='yellow')
plot2 = plt.plot(train_sizes, test_mean, label='Validation Score', color='blue', linestyle=':')
ce1_fill = plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color='yellow',
                            alpha=0.15, label='Training Variability')
ce2_fill = plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color='blue',
                            alpha=0.15, label='Validation Variability')

xlabel = plt.xlabel('Training Set Size')
ylabel = plt.ylabel('Log-Loss (Cross-entropy)')
title = plt.title('Log-Loss Learning Curve - Weighted Logistic Regression')
legend = plt.legend(loc='lower right', framealpha=0.35)
grid = plt.grid(True)
plt.tight_layout()
plt.show()
```



Inferences from the Learning Curve Plot

The learning curve for the weighted logistic regression model shows that the log-loss for both training and validation sets closely aligns, converging at ~0.328 when the full dataset is used. This suggests no signs of overfitting or underfitting.

However, the variance in validation prediction errors remains significant, indicating that the model still struggles to generalize well on unseen data. Compared to the standard (unweighted) model, log-loss increased significantly, which was expected. This reflects the trade-off introduced by balancing the target classes, which led to a higher number of misclassified instances overall.

Logistic Regression with Oversampling using SMOTE

Moving one step further we will try to address the challenges posed by the heavily imbalanced dataset, this time by employing an oversampling technique to balance the classes in the training data.

In this iteration, we will use `SMOTE` (Synthetic Minority Over-sampling Technique) to generate artificial samples for the minority classes. Oversampling the minority classes helps to improve the model's ability to make predictions, since by training the model on a balanced dataset, we aim to improve the recall for minority classes while maintaining the overall model accuracy.

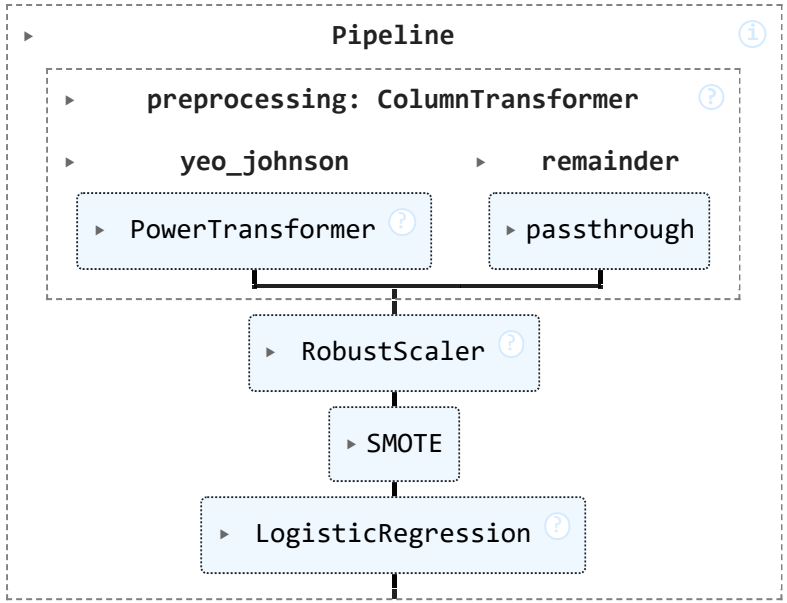
SMOTE creates synthetic records by interpolating between existing samples of the minority class and their nearest neighbors. This approach increases the representation of minority classes in the training set, ensuring that each class is equally represented.
The original test set is left untouched and retains its class imbalance so as to ensure an unbiased evaluation of the model's performance on *real-life* terms.

Further details on SMOTE can be found in the **official documentation**.

In [ ]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer, RobustScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline
import time

# Define features (X) and target (y)
X = lagged_1h_df.drop(columns=['date', 'weather_event'])
y = lagged_1h_df['weather_event']

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)

# Define the preprocessing step for Yeo-Johnson transformation
preprocessor = ColumnTransformer([
```

```
    ('yeo_johnson', PowerTransformer(method='yeo-johnson'), variables_to_transform),
], remainder='passthrough')  # Leave other features untouched

# Define the pipeline with SMOTE
oversampling_pipeline = Pipeline([
    ('preprocessing', preprocessor),   # Step 1: Yeo-Johnson transformation for specific variables
    ('scaling', RobustScaler()),       # Step 2: Scale all features
    ('smote', SMOTE(random_state=42)), # Step 3: Apply SMOTE to oversample the minority classes
    ('oversampling_model', LogisticRegression(
        max_iter=1500,
        random_state=42,
        solver='lbfgs',
        n_jobs=-1
    ))
])

# Fit the pipeline to the training data
start = time.time()
oversampling_pipeline.fit(X_train, y_train)
execution_time = time.time() - start

# Access fitted model and print details
oversampled_model_fitted = oversampling_pipeline['oversampling_model']
print(f"Execution time: {execution_time:.2f} seconds")
print(f"Number of iterations for convergence: {oversampled_model_fitted.n_iter_}")
```

Out[ ]:




```
Execution time: 2.73 seconds
Number of iterations for convergence: [145]
```

Model Evaluation

CV evaluation

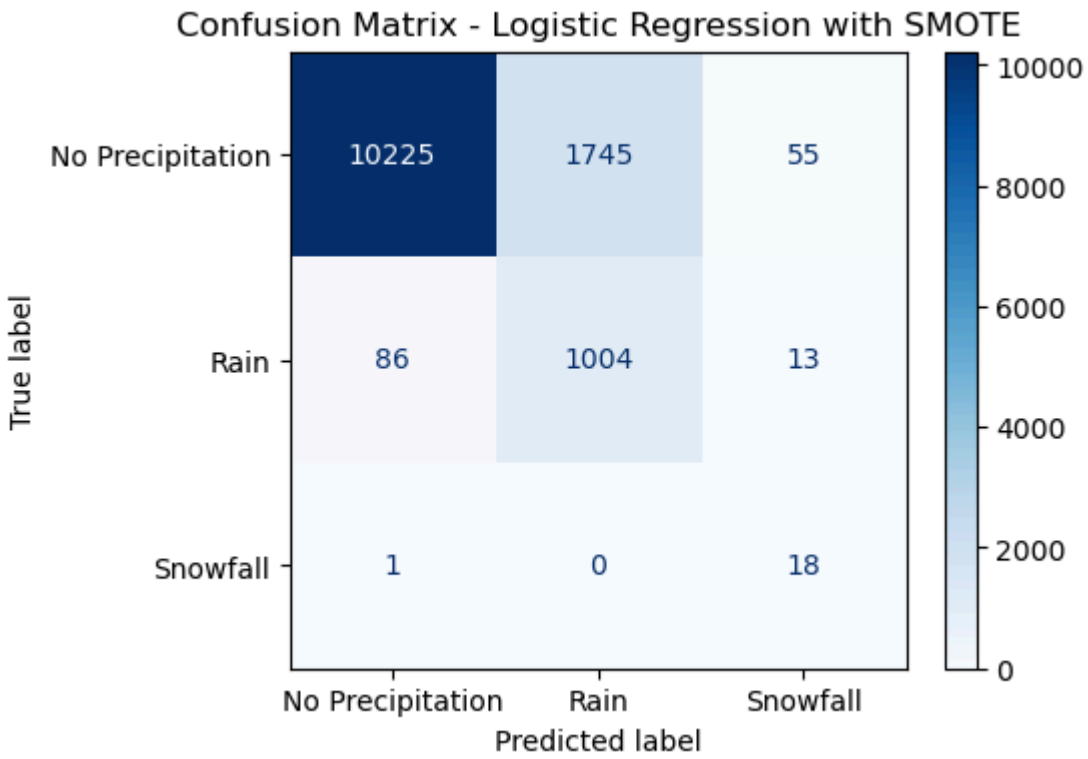
In [ ]:

```python
# Evaluate the model using cross-validation on the full data set
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
smote_cv = cross_val_score(oversampling_pipeline, X_train, y_train, cv=stkf, scoring="balanced_accuracy", n_jobs=-1)

print(smote_cv)
print(f'\nThe mean value of balanced_accuracy on the cv evaluation is {smote_cv.mean().round(3)}')
print(f'\nThe standard deviation of balanced_accuracy on the cv evaluation is {smote_cv.std().round(3)}')
print('\n')
```

```
[0.9172878  0.9161588  0.91931547 0.91211763 0.92677032 0.85092515
 0.90624593 0.92121616 0.91706117 0.84235722]

The mean value of balanced_accuracy on the cv evaluation is 0.903

The standard deviation of balanced_accuracy on the cv evaluation is 0.029
```

The Confusion Matrix

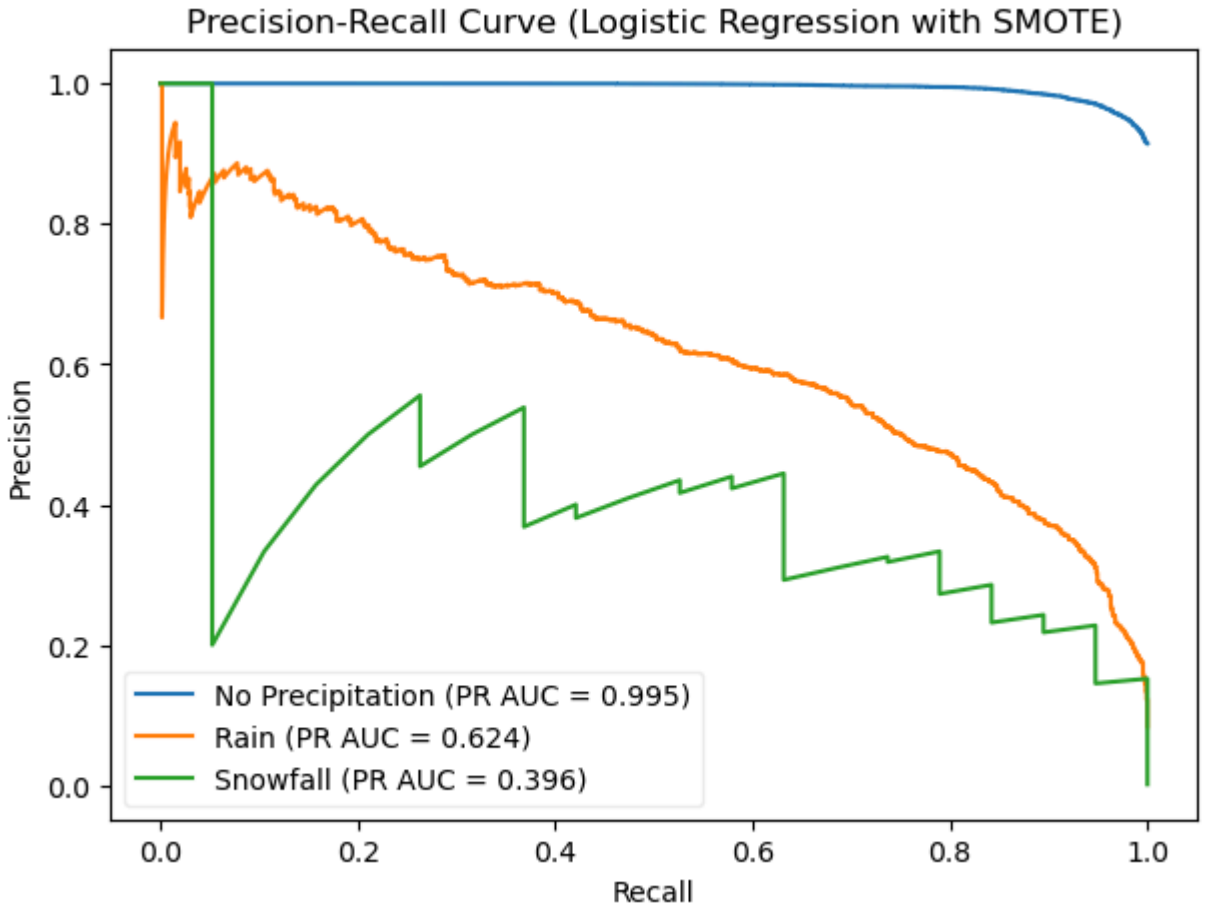
In [ ]:
```
y_pred_smote = oversampling_pipeline.predict(X_test)
# Confusion Matrix
conf_matrix_smote = confusion_matrix(y_test, y_pred_smote)
fig, ax = plt.subplots(figsize=(5, 4))
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_smote, display_labels=['No Precipitation', 'Rain', 'Snowfall'])
plot = disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
title = plt.title('Confusion Matrix - Logistic Regression with SMOTE')
plt.show()
```



The Precision-Recall Curve

In [ ]:
```
# Define the class labels
class_labels = ['No Precipitation', 'Rain', 'Snowfall']

# Producing the probabilities of each class for every data record in the test set
y_pred_probs_smote = oversampling_pipeline.predict_proba(X_test)

# Plot Precision-Recall curves
figure = plt.figure(figsize=(7, 5))
for i, label in enumerate(class_labels):
    precision, recall, _ = precision_recall_curve(y_test == i, y_pred_probs_smote[:, i])
    pr_auc = auc(recall, precision)

    # Plot the precision-recall curve
    plot = plt.plot(recall, precision, label=f'{label} (PR AUC = {pr_auc:.3f})')
```

```
xlabel = plt.xlabel('Recall')
ylabel = plt.ylabel('Precision')
title = plt.title('Precision-Recall Curve (Logistic Regression with SMOTE)')
legend = plt.legend(loc='lower left', framealpha=0.35)
plt.show()
```



The Classification Report

In [ ]:
```
# Classification report
report_dict_smote = classification_report(y_test, y_pred_smote, digits=3, output_dict=True)
```

```python
# Calculate PR AUC for each class
precision_0, recall_0, _ = precision_recall_curve(y_test == 0, y_pred_probs_smote[:, 0])
pr_auc_0 = auc(recall_0, precision_0)

precision_rain, recall_rain, _ = precision_recall_curve(y_test == 1, y_pred_probs_smote[:, 1])
pr_auc_rain = auc(recall_rain, precision_rain)

precision_snow, recall_snow, _ = precision_recall_curve(y_test == 2, y_pred_probs_smote[:, 2])
pr_auc_snow = auc(recall_snow, precision_snow)

# Create a single row dataframe with the required metrics
report_df_smote = pd.DataFrame({
    'precision_0': [report_dict_smote['0']['precision']],
    'recall_0': [report_dict_smote['0']['recall']],
    'f1_0': [report_dict_smote['0']['f1-score']],
    'pr_auc_0': [pr_auc_0],

    'precision_rain': [report_dict_smote['1']['precision']],
    'recall_rain': [report_dict_smote['1']['recall']],
    'f1_rain': [report_dict_smote['1']['f1-score']],
    'pr_auc_rain': [pr_auc_rain],

    'precision_snow': [report_dict_smote['2']['precision']],
    'recall_snow': [report_dict_smote['2']['recall']],
    'f1_snow': [report_dict_smote['2']['f1-score']],
    'pr_auc_snow': [pr_auc_snow],
    'balanced_accuracy': [balanced_accuracy_score(y_test, y_pred_smote)],
    'f1_macro': [f1_score(y_test, y_pred_smote, average='macro')],
    'pr_auc_macro': [average_precision_score(y_test, y_pred_probs_smote, average='macro')]

})

# Display the DataFrame
print('Logistic Regression with SMOTE')
report_df_smote
```

```
Logistic Regression with SMOTE
```

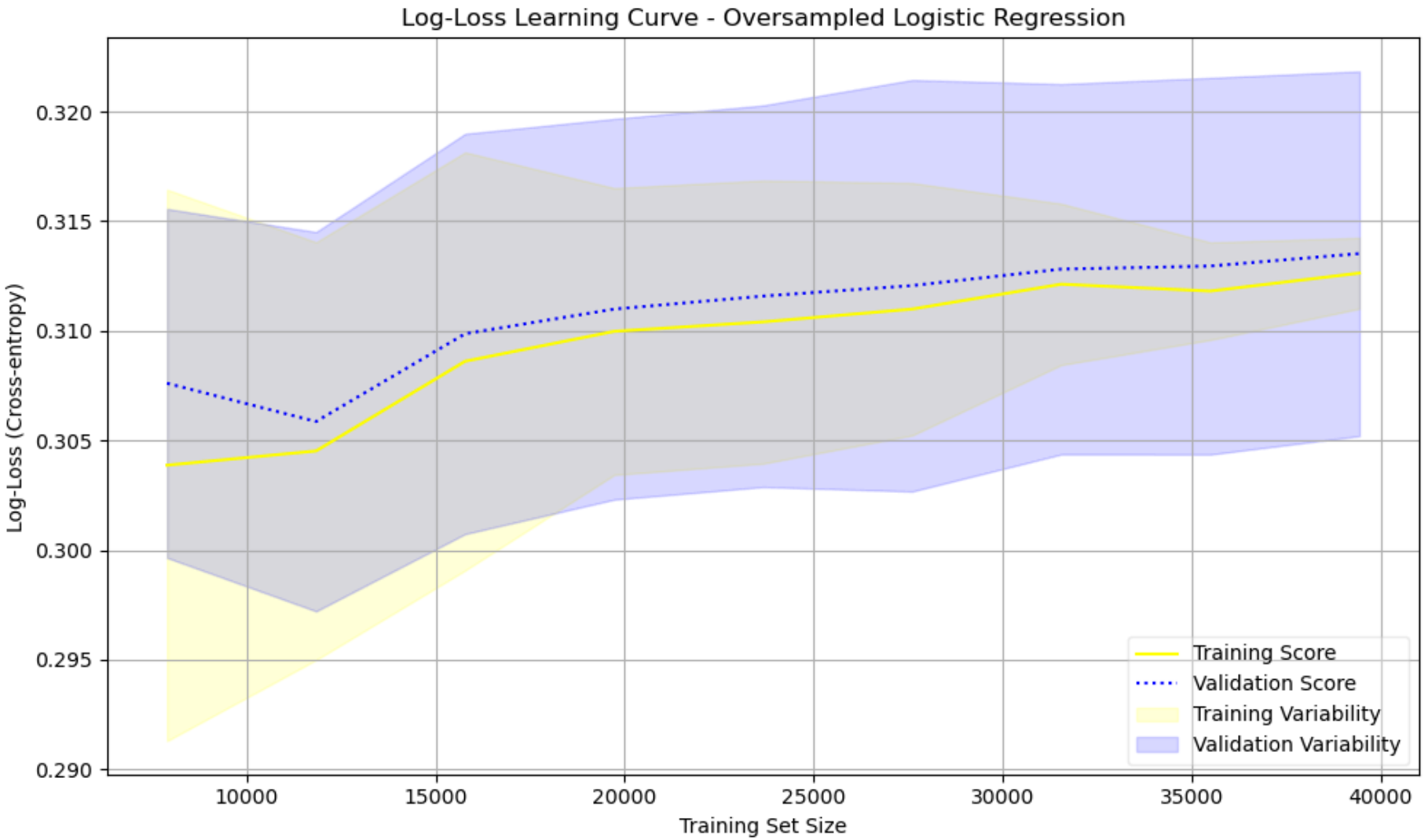
Out[ ]:

	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_snow	pr_au
0	0.991563	0.850312	0.915521	0.994716	0.365224	0.910245	0.521288	0.624033	0.209302	0.947368	0.342857	(

The learning curve

In [154…

```
from sklearn.model_selection import learning_curve
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

train_sizes, train_scores, test_scores = learning_curve(
    oversampling_pipeline, X, y, cv=stkf, scoring='neg_log_loss', n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), shuffle=True, random_state=42
)

train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

figure = plt.figure(figsize=(10, 6))
plot1 = plt.plot(train_sizes, train_mean, label='Training Score', color='yellow')
plot2 = plt.plot(train_sizes, test_mean, label='Validation Score', color='blue', linestyle=':')
ce1_fill = plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color='yellow',
                            alpha=0.15, label='Training Variability')
ce2_fill = plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color='blue',
                            alpha=0.15, label='Validation Variability')

xlabel = plt.xlabel('Training Set Size')
ylabel = plt.ylabel('Log-Loss (Cross-entropy)')
title = plt.title('Log-Loss Learning Curve - Oversampled Logistic Regression')
legend = plt.legend(loc='lower right', framealpha=0.35)
grid = plt.grid(True)
plt.tight_layout()
plt.show()
```



The log-loss learning curve for the oversampled logistic regression model (SMOTE) shows that the training and validation log-loss curves closely align, converging at approximately 0.312 when all data points are used. This indicates no signs of overfitting or underfitting, meaning the model generalizes well without excessively memorizing the training data.

However, the validation variance remains relatively high, suggesting that the model still struggles with generalization on unseen data, likely due to the synthetic samples introduced by SMOTE. Compared to the weighted logistic regression model (class_weight='balanced'), the log-

loss is slightly lower, indicating a slight improvement in handling class imbalance. This is expected, as oversampling increases the presence of minority class instances during training, reducing bias toward the majority class.

One-vs-Rest (OvR) Multinomial Logistic Regression

Following the previous section where we applied an upsampling technique (SMOTE) to handle class imbalance and enhance the logistic regression model's performance, this time in the search for a possible improvement we will explore a more complex model. This variant uses the One-vs-the-Rest (OvR) strategy.
Also known as one-vs-all, this strategy involves fitting one classifier per class, where each class is fitted against all the other classes, effectively transforming the multiclass problem into multiple binary classification problems, which in the end will be combined again to construct the final multiclass model.

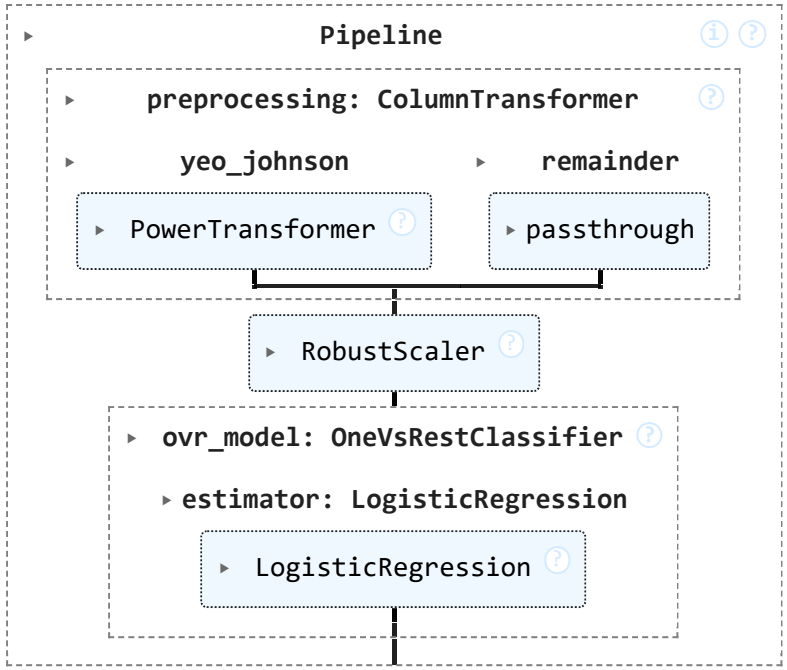
By wrapping the `LogisticRegression` class inside the OvR strategy, we can build separate binary classifiers for each class and combine their predictions to perform multiclass classification. *(each binary classifier outputs a score (a probability) for its respective class, and the final prediction is made by selecting the class with the highest score(probability)).*

While initially this seems unorthodox and naturally more computationally intensive, the goal of using this strategy is to reveal/identify patterns of the data at the binary level which might not be evident at the multi-class level and sometimes can enhance the model's ability to distinguish between classes.
For more information, you can refer to the **official documentation**

In [ ]:
```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer, RobustScaler
from sklearn.pipeline import Pipeline
from sklearn.multiclass import OneVsRestClassifier

# Define features (X) and target (y)
X = lagged_1h_df.drop(columns=['date', 'weather_event'])
y = lagged_1h_df['weather_event']

# Split data into training and test sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)

# Define the preprocessing step for Yeo-Johnson transformation
```

```
preprocessor = ColumnTransformer([
    ('yeo_johnson', PowerTransformer(method='yeo-johnson'), variables_to_transform),
], remainder='passthrough')  # Leave other features untouched

# Define the pipeline
ovr_pipeline = Pipeline([
    ('preprocessing', preprocessor),   # Step 1: Yeo-Johnson transformation for specific variables
    ('scaling', RobustScaler()),       # Step 2: Scale all features
    ('ovr_model', OneVsRestClassifier(LogisticRegression(
        class_weight='balanced',
        max_iter=1500,
        random_state=42,
        solver='lbfgs',
        n_jobs=-1
    )))
])

# Fit the pipeline
import time
start = time.time()

ovr_pipeline.fit(X_train, y_train)

execution_time = time.time() - start
print(f"Execution time: {execution_time:.2f} seconds")

# Check the number of iterations for each classifier in the OneVsRest setup
for i, estimator in enumerate(ovr_pipeline['ovr_model'].estimators_):
    print(f"Class {i}: Number of iterations = {estimator.n_iter_}")
```

Out[ ]:




```
Execution time: 3.39 seconds
Class 0: Number of iterations = [48]
Class 1: Number of iterations = [46]
Class 2: Number of iterations = [37]
```

Model Evaluation

CV evaluation

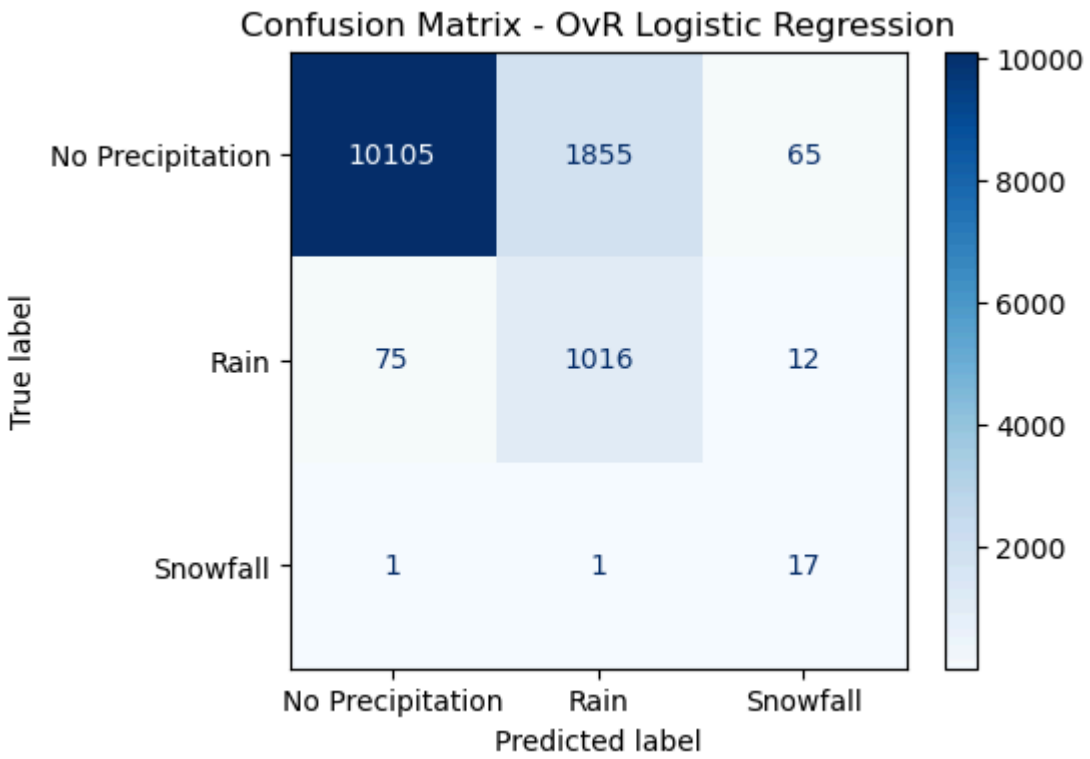
In [ ]:

```python
from sklearn.model_selection import StratifiedKFold
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

logreg_ovr_cv = cross_val_score(ovr_pipeline, X_train, y_train, cv=stkf, scoring="balanced_accuracy", n_jobs=-1)
print(logreg_ovr_cv)
print(f'\nThe mean value of balanced_accuracy is {logreg_ovr_cv.mean().round(3)}')
print(f'\nThe standard deviation of balanced_accuracy is {logreg_ovr_cv.std().round(3)}\n')
```

```
[0.82921748 0.91623339 0.91623656 0.91055391 0.92580056 0.91850334
 0.90339489 0.91493954 0.91609529 0.91986762]

The mean value of balanced_accuracy is 0.907

The standard deviation of balanced_accuracy is 0.027
```

The Confusion Matrix

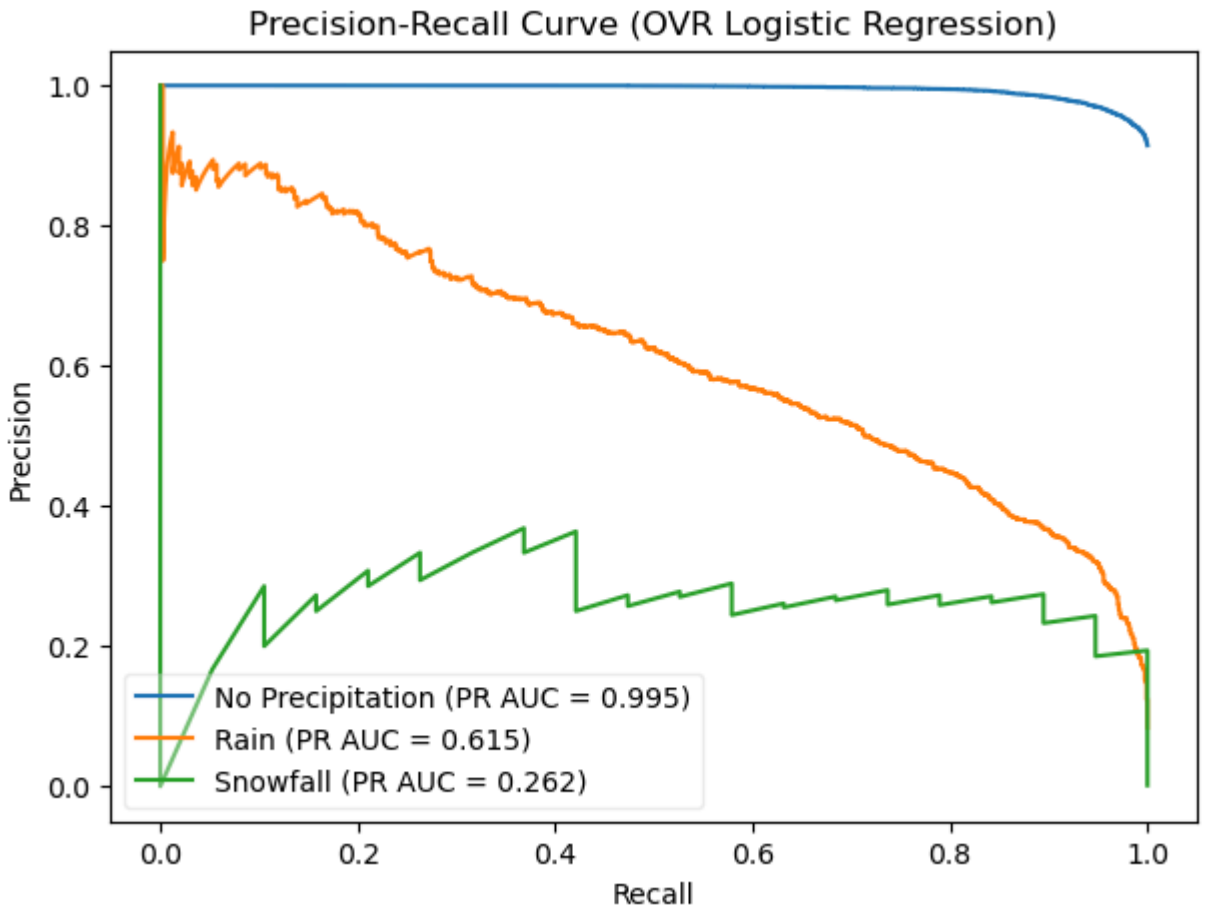
In [ ]:
```
# Predict on the test set
y_pred_ovr = ovr_pipeline.predict(X_test)
# Confusion Matrix
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
conf_matrix_ovr = confusion_matrix(y_test, y_pred_ovr)
fig, ax = plt.subplots(figsize=(5, 4))
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_ovr, display_labels=['No Precipitation', 'Rain', 'Snowfall'])
plot = disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
title = plt.title('Confusion Matrix - OvR Logistic Regression')
plt.show()
```



The Precision-Recall Curve

In [ ]:
```
# Class labels
class_labels = ['No Precipitation', 'Rain', 'Snowfall']

# Producing the probabilities of each class for every data record in the test set
y_pred_probs_ovr = ovr_pipeline.predict_proba(X_test)

# Initialize plot
figure = plt.figure(figsize=(7, 5))

# Loop through each class
for i, label in enumerate(class_labels):
    precision, recall, _ = precision_recall_curve(y_test == i, y_pred_probs_ovr[:, i])
    pr_auc = auc(recall, precision)
```

```
    # Plot the precision-recall curve for the class
    pr_curve = plt.plot(recall, precision, label=f'{label} (PR AUC = {pr_auc:.3f})')

xlabel = plt.xlabel('Recall')
ylabel = plt.ylabel('Precision')
title = plt.title('Precision-Recall Curve (OVR Logistic Regression)')
legend = plt.legend(loc='lower left', framealpha=0.35)
plt.show()
```



The Classification Report

In [ ]:

```
# Classification report
report_dict_ovr = classification_report(y_test, y_pred_ovr, digits=3, output_dict=True)

# Calculate PR AUC for each class
precision_0, recall_0, _ = precision_recall_curve(y_test == 0, y_pred_probs_ovr[:, 0])
pr_auc_0 = auc(recall_0, precision_0)

precision_rain, recall_rain, _ = precision_recall_curve(y_test == 1, y_pred_probs_ovr[:, 1])
pr_auc_rain = auc(recall_rain, precision_rain)

precision_snow, recall_snow, _ = precision_recall_curve(y_test == 2, y_pred_probs_ovr[:, 2])
pr_auc_snow = auc(recall_snow, precision_snow)

# Create a single row dataframe with the required metrics
report_df_ovr = pd.DataFrame({
    'precision_0': [report_dict_ovr['0']['precision']],
    'recall_0': [report_dict_ovr['0']['recall']],
    'f1_0': [report_dict_ovr['0']['f1-score']],
    'pr_auc_0': [pr_auc_0],

    'precision_rain': [report_dict_ovr['1']['precision']],
    'recall_rain': [report_dict_ovr['1']['recall']],
    'f1_rain': [report_dict_ovr['1']['f1-score']],
    'pr_auc_rain': [pr_auc_rain],

    'precision_snow': [report_dict_ovr['2']['precision']],
    'recall_snow': [report_dict_ovr['2']['recall']],
    'f1_snow': [report_dict_ovr['2']['f1-score']],
    'pr_auc_snow': [pr_auc_snow],
    'balanced_accuracy': [balanced_accuracy_score(y_test, y_pred_ovr)],
    'f1_macro': [f1_score(y_test, y_pred_ovr, average='macro')],
    'pr_auc_macro': [average_precision_score(y_test, y_pred_probs_ovr, average='macro')]

})

# Display the DataFrame
print('Logistic Regression with OVR')
report_df_ovr
```

```
Logistic Regression with OVR
```

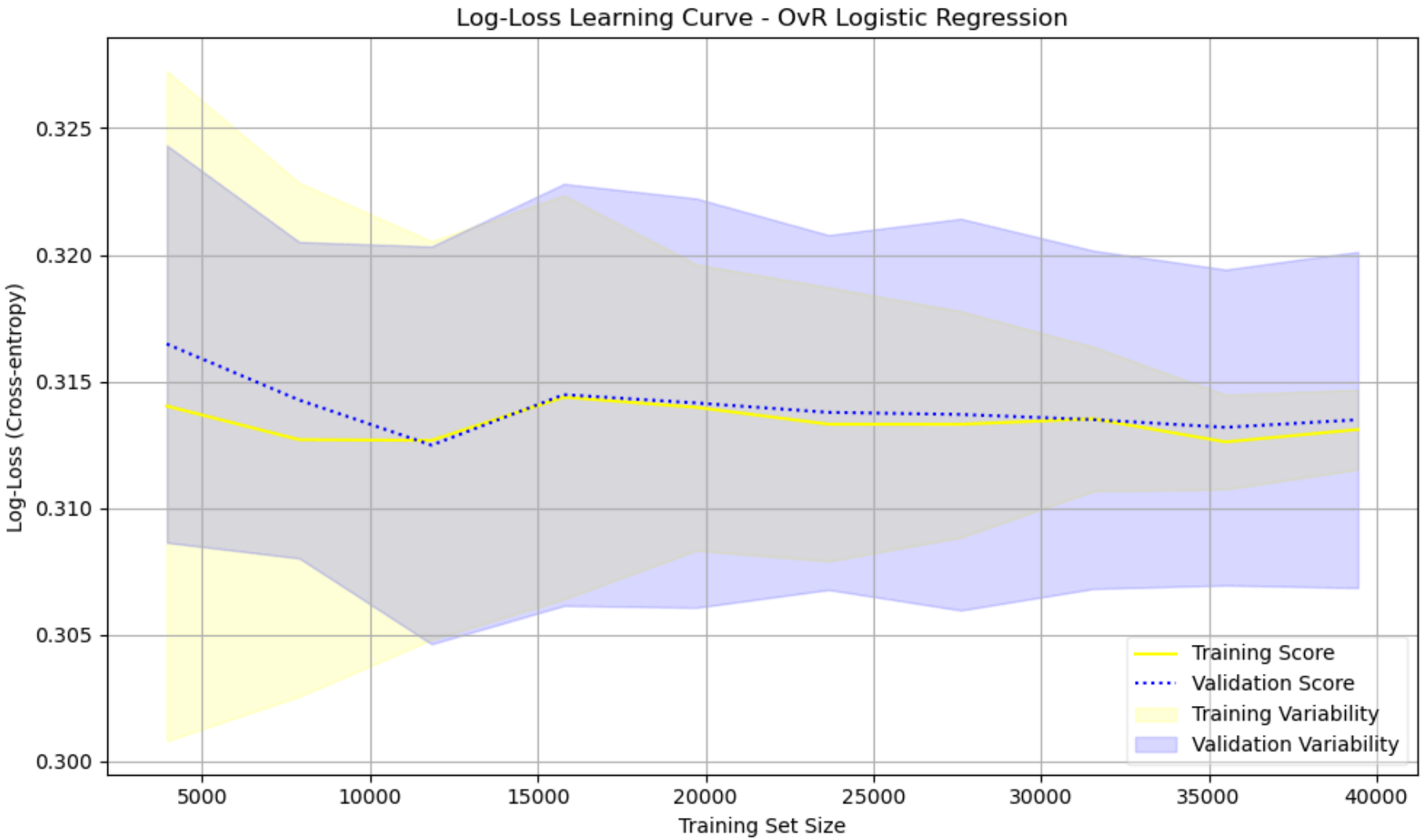
Out[ ]:

	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_snow	pr_au
0	0.992535	0.840333	0.910114	0.994825	0.35376	0.921124	0.511195	0.614794	0.180851	0.894737	0.300885	

The learning curve

In [175…

```
from sklearn.model_selection import learning_curve
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

train_sizes, train_scores, test_scores = learning_curve(
    ovr_pipeline, X, y, cv=stkf, scoring='neg_log_loss', n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), shuffle=True, random_state=42
)

train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

figure = plt.figure(figsize=(10, 6))
plot1 = plt.plot(train_sizes, train_mean, label='Training Score', color='yellow')
plot2 = plt.plot(train_sizes, test_mean, label='Validation Score', color='blue', linestyle=':')
ce1_fill = plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color='yellow',
                            alpha=0.15, label='Training Variability')
ce2_fill = plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color='blue',
                            alpha=0.15, label='Validation Variability')

xlabel = plt.xlabel('Training Set Size')
ylabel = plt.ylabel('Log-Loss (Cross-entropy)')
title = plt.title('Log-Loss Learning Curve - OvR Logistic Regression')
legend = plt.legend(loc='lower right', framealpha=0.35)
grid = plt.grid(True)
plt.tight_layout()
plt.show()
```



Inferences from the Learning Curve Plot

The log-loss learning curve for the One-vs-Rest (OvR) logistic regression model shows that the training and validation log-loss curves are well-aligned, converging at approximately 0.313 when the full dataset is used. This indicates that the model does not appear to overfit or underfit, meaning it generalizes well across the dataset.

Compared to the SMOTE-oversampled iteration, the log-loss remains in a similar range (- 0.312), showing a consistent performance. However, the validation variance is still relatively high, implying that the model's predictions remain somewhat unstable across different validation splits.

Notably, the log-loss remains more stable over increasing dataset sizes, suggesting that OvR Logistic Regression is benefiting from a more balanced binary classification of multiple classes rather than forcing a single decision boundary for all classes at once (as in standard multinomial logistic regression).

Hyperparameter Optimized Logistic Regression

So far, each iteration has improved the baseline model *(the initial unweighted logistic regression model)*. However, performance remains similar across iterations, with minor variations.

What if we try to fine tune the model?
In the arguments of `LogisticRegression` there is a hyperparameter C corresponding to the inverse of regularization strength, to control a possible overfit of the model to the data, and where smaller values specify stronger regularization. It can take any real value in an interval like from 1e-3 up to 100 for example. A fine tuning process consists in essence in finding the best value for this hyperparameter that yields the best performance with respect to the metric of our choice. We can check manually this range of numbers and review the results, but this can take several days...

There is a more effective way to search for all possible combinations using **GridSearchCV**, where you manually define the parameter space, and the method will take over, creating and exploring all possible combinations *(or values if it is for a single hyperparameter)* for you, evaluating the model based on a specified evaluation metric, and return the best model along with its set of optimal parameters.
It is important to note that GridSearchCV accepts only discrete values for each parameter defined by the user. If for example you need to tune 3 hyperparameters each taking 5 possible/distinct values, this will result in $5^3 = 125$ combinations and for a 10-fold cross-validation this will result in fitting 1250 models, making the method computationally expensive and often impractical for larger search spaces.

There is an alternative method called **RandomizedSearchCV**, which speeds up the process by randomly selecting values from the parameter space you have defined. You specify how many combinations should be evaluated by setting the `n_iter` argument. The method evaluates these combinations and returns the best model along with its corresponding parameters. This approach significantly reduces the computational time but does not guarantee that it will find the absolute best model, given the dataset and parameter space at hand.

In this section, we will use another method called **BayesSearchCV**, which exploits Bayesian optimization. This method intelligently searches the hyperparameter space using a probabilistic model to guide the search, making it more efficient than traditional methods like `GridSearchCV`

and `RandomizedSearchCV`.

The process begins by defining the hyperparameter search space. Initially, BayesSearchCV performs a small number of random evaluations by selecting different combinations *(or values in the case of a single hyperparameter)* of hyperparameters. These initial evaluations explore diverse regions of the parameter space to gather data. Based on this data, the algorithm builds a probabilistic model, which helps predict which areas of the hyperparameter space are most likely to contain optimal combinations.

This approach significantly reduces the number of hyperparameter combinations that need to be evaluated, making it much more efficient. The best model and its optimal hyperparameter settings are returned after evaluating all combinations. Even though this method is much more efficient, there's a degree of randomness in the cross-validation process and in the initial random samples selected during the BayesSearchCV process. This might cause optimization to focus on suboptimal areas of the hyperparameter space and it does not guarantee that you will indeed end up with the best possible model settings. An increase on the n_iter argument (the number of parameter combinations) can help in this direction but it will also increase the computational cost. Experience also will be valuable since you can restrict the hyperparameter space if you already know where the best possible combination might lie and this can be set by defining the prior distribution in the prior argument of each parameter you aim to search for and optimize.

Key points on BayesSearchCV

- The n_iter argument is specified by the user when calling BayesSearchCV(). After setting this parameter, the user no longer controls how the algorithm selects combinations within the search space. For example, if n_iter=30, the algorithm will evaluate 30 different hyperparameter combinations in total. Initially, a small number of combinations are randomly selected and evaluated. Based on these initial evaluations, the algorithm builds a probabilistic model, which it then uses to guide the remaining iterations, focusing on combinations that are predicted to yield better performance. The best model from all 30 evaluations is returned as the final model.
- Increasing the n_iter argument, increases your chances of finding the best possible model, but at the same time you are also increasing your computational costs (time and resources)
- Initially, BayesSearchCV performs a few random evaluations of hyperparameter combinations (the rest combinations will be used after the probabilistic model is built). These initial samples are influenced by the **`prior`** distribution that we define for each parameter (e.g., `log-uniform` for `C`). The prior distribution represents our **initial belief** about where the best hyperparameters might lie.
- In this phase, random hyperparameter values are sampled from the defined search space, but the **`prior`** (e.g., `log-uniform`) influences the frequency with which different values are selected. For example, if you're using 'log-uniform' for the hyperparameter'C', smaller values will be sampled more often than larger values, because this is your initial/prior belief of where the best parameter values might lie.

- If your initial guess/belief about the prior distribution for sampling values is incorrect, don't worry. This will be reflected in the probabilistic model built by the algorithm, based on the poor performance of the models built using that prior. These poor results will prompt the algorithm to adapt and guide the search towards more promising ranges. Therefore, it is the range of the parameter values that you should be careful not to restrict. An incorrect initial prior probability will not prevent the algorithm from finding good solutions, as it will adjust during the search.
- After evaluating the initial random samples, BayesSearchCV builds a **probabilistic model**. This model then uses the updated probabilities distribution (*posterior distribution*), which is the updated belief about where the best hyperparameters might lie after incorporating the results of the initial evaluations.
- From this point, the **posterior distribution** is used to guide the search. The algorithm no longer selects hyperparameters randomly; instead, it uses the posterior distribution to focus the search on areas of the hyperparameter space that are predicted to perform better.

Last important note
While BayesSearchCV is more effective than traditional methods at exploring and exploiting the hyperparameter space, it still cannot guarantee the optimal set of parameters—and thus the best possible model-, **due to its probabilistic nature**.

For more information on the BayesSearchCV, please refer to the following links:
https://scikit-optimize.github.io/stable/modules/generated/skopt.BayesSearchCV.html
and
https://deephyper.readthedocs.io/en/latest/_autosummary/deephyper.skopt.BayesSearchCV.html

Before running the script below, ensure that you have run the following:
`pip install scikit-optimize`

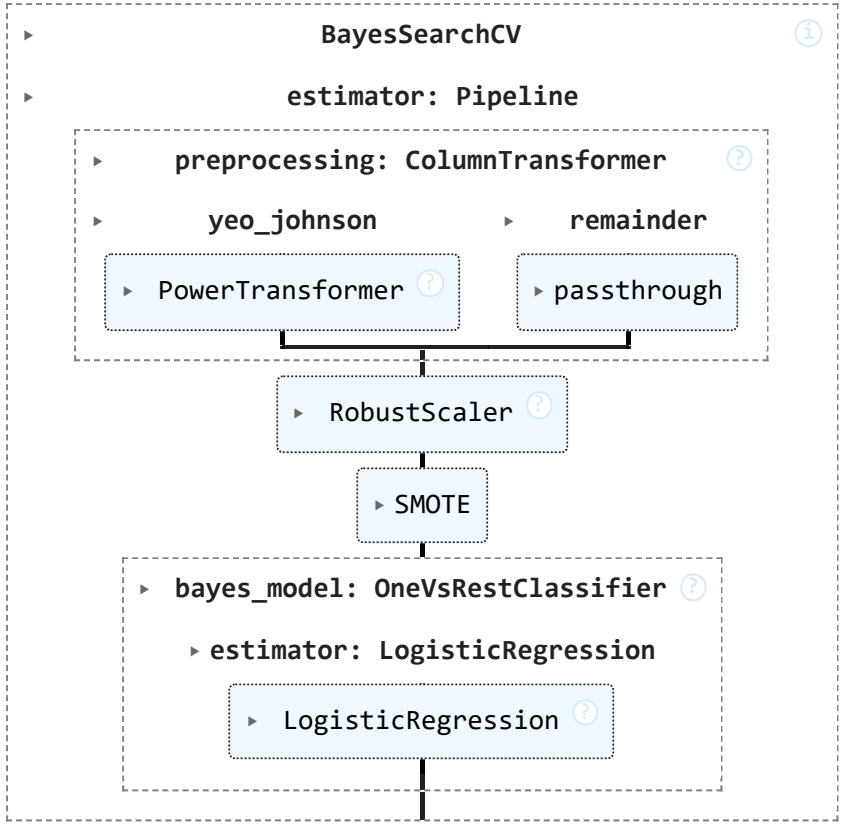
In [279…

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import PowerTransformer, RobustScaler
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline
from skopt import BayesSearchCV
from skopt.space import Real, Categorical
import time

# Define features (X) and target (y)
X = lagged_1h_df.drop(columns=['date', 'weather_event'])
y = lagged_1h_df['weather_event']
```

```python
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_state=42)

# Define the preprocessing step for Yeo-Johnson transformation
preprocessor = ColumnTransformer([
    ('yeo_johnson', PowerTransformer(method='yeo-johnson'), variables_to_transform),
], remainder='passthrough')  # Leave other features untouched

# Define the pipeline
bayes_pipeline = Pipeline([
    ('preprocessing', preprocessor),   # Step 1: Yeo-Johnson transformation for specific variables
    ('scaling', RobustScaler()),       # Step 2: Scale all features
    ('smote', SMOTE(random_state=42)),
    ('bayes_model', OneVsRestClassifier(LogisticRegression(
        max_iter=1500,
        random_state=42,
        solver='lbfgs',
        n_jobs=-1
    )))
])

param_space = {
    'bayes_model__estimator__C': Real(1e-1, 10, prior='log-uniform'),  # Regularization strength
    'bayes_model__estimator__solver': Categorical(['lbfgs', 'newton-cg'])  # Optimization solver
}

# Stratified k-fold cross-validation
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# BayesSearchCV for hyperparameter tuning
bayes_search = BayesSearchCV(
    estimator=bayes_pipeline,
    search_spaces=param_space,
    n_iter=50,  # Number of parameter combinations to evaluate
    scoring='balanced_accuracy',
    cv=stkf,
    random_state=42,
    n_jobs=-1
)
```

```python
# Fit BayesSearchCV to the training data
import time
start = time.time()
bayes_search.fit(X_train, y_train)
execution_time = time.time() - start

# Output results
print("Best parameters found")
bayes_search.best_params_
print("Best cross-validation score")
bayes_search.best_score_
print(f"Execution time: {execution_time:.2f} seconds")

# Extract and save detailed results
results_df = pd.DataFrame(bayes_search.cv_results_)
```

Out[279…



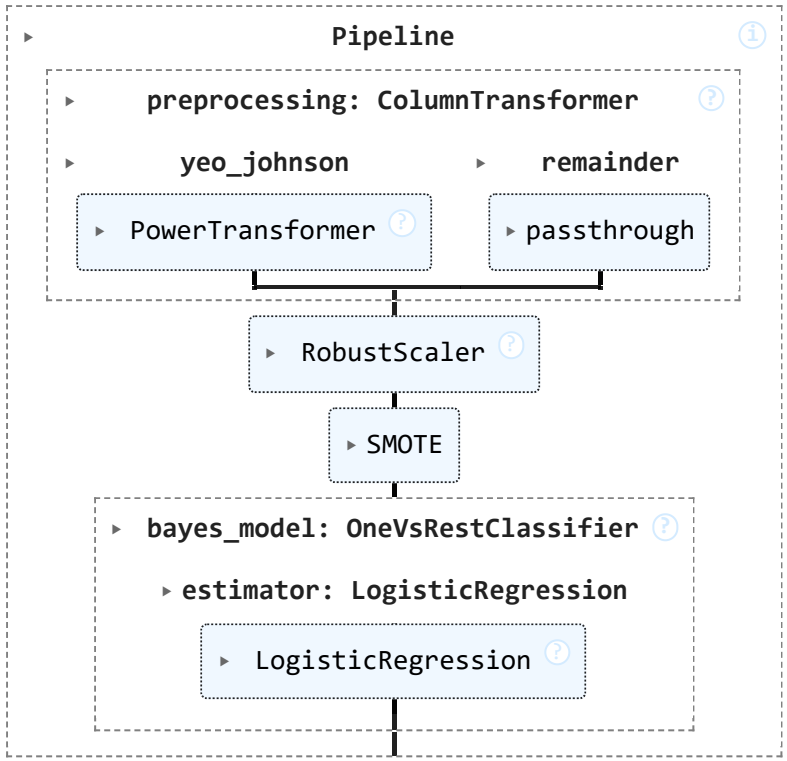
```
Best parameters found
```

Out[279…
```
OrderedDict([('bayes_model__estimator__C', 1.216091359905912),
             ('bayes_model__estimator__solver', 'lbfgs')])
Best cross-validation score
```

Out[279…
```
0.9165599969240079
Execution time: 138.99 seconds
```

In [281…
```
bayes_best_model = bayes_search.best_estimator_
bayes_best_model.fit(X_train, y_train)
```

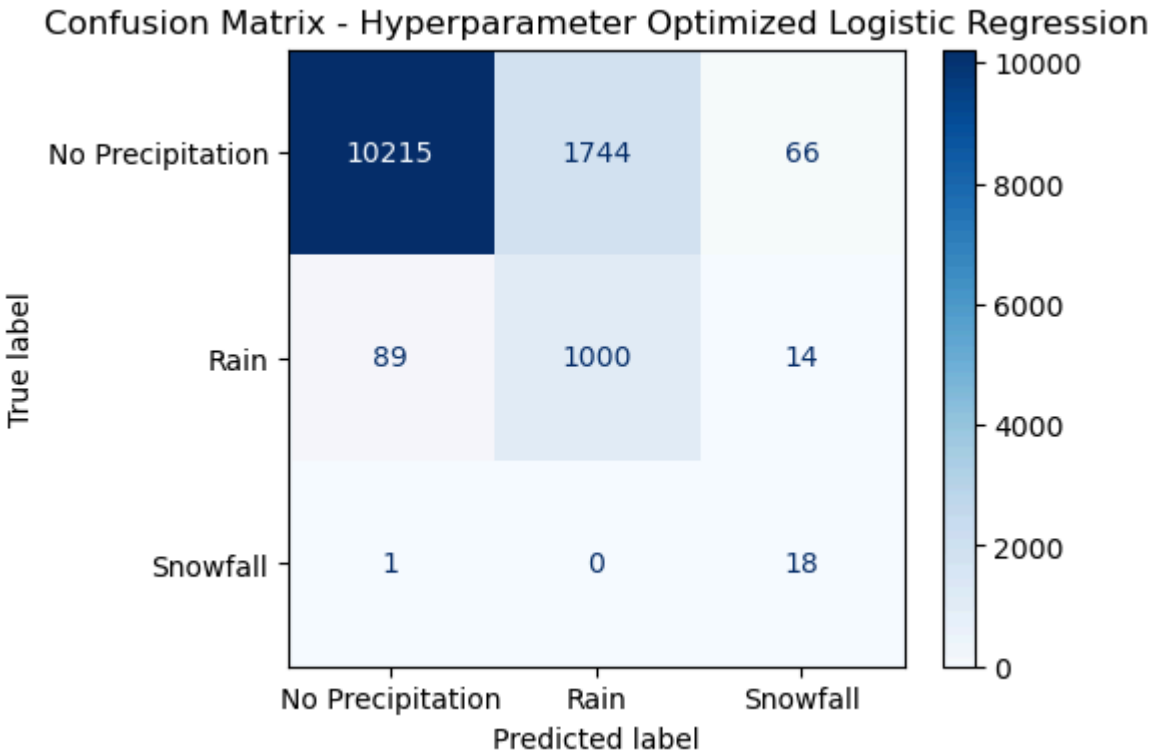
Out[281…



In [ ]:
```
y_pred_bayes_grid = bayes_best_model.predict(X_test)
```

The Confusion Matrix

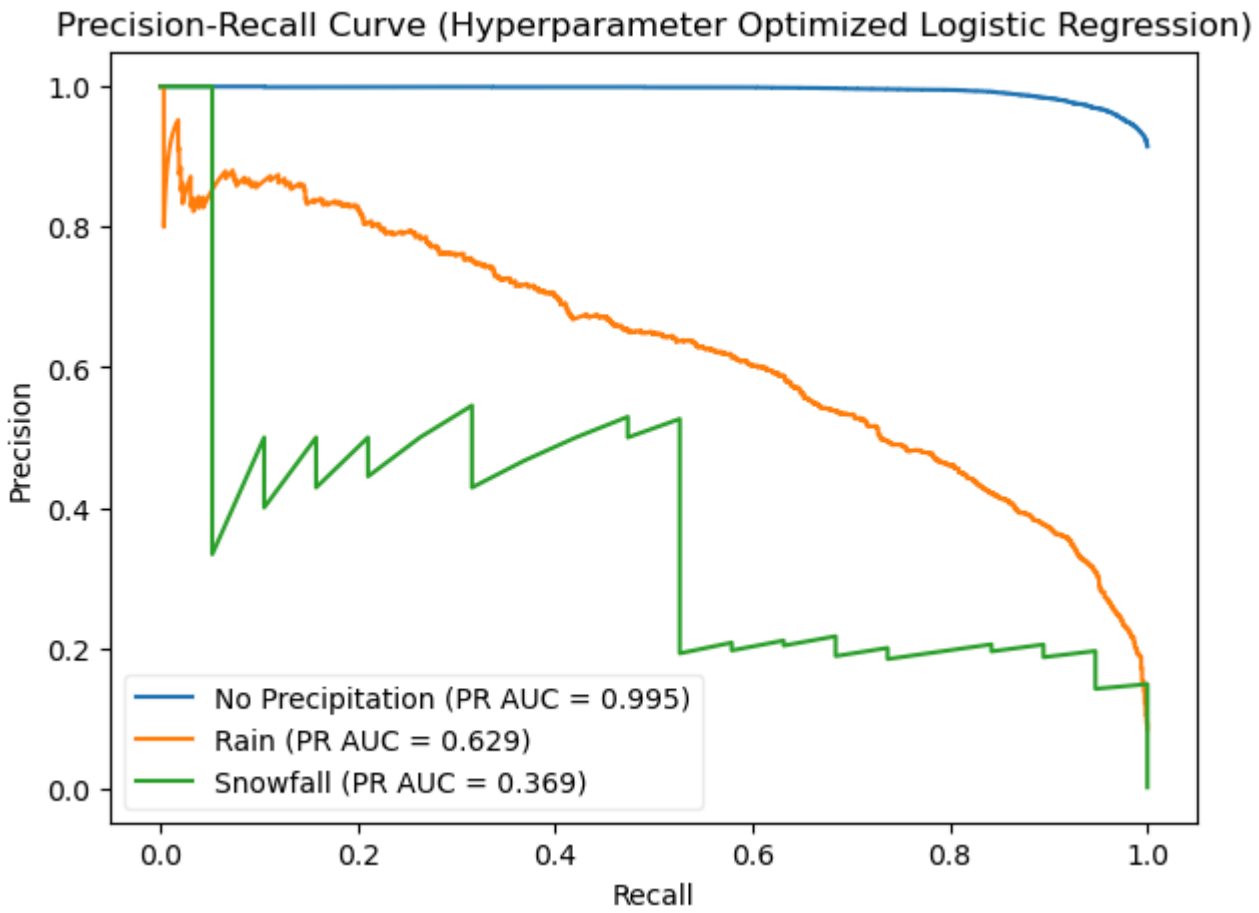
In [ ]:
```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix
fig, ax = plt.subplots(figsize=(5, 4))
conf_matrix_bayes_grid = confusion_matrix(y_test, y_pred_bayes_grid)
disp = ConfusionMatrixDisplay(conf_matrix_bayes_grid, display_labels=['No Precipitation', 'Rain', 'Snowfall'])
plot = disp.plot(ax=ax, cmap=plt.cm.Blues, values_format='d')
title = plt.title('Confusion Matrix - Hyperparameter Optimized Logistic Regression')
plt.show()
```



The Precision-Recall Curve

In [ ]:
```
# Generate precision-recall curve for each class
y_pred_probs_bayes_grid = bayes_best_model.predict_proba(X_test)

# Class labels
class_labels = ['No Precipitation', 'Rain', 'Snowfall']
```

```
# Initialize plot
figure = plt.figure(figsize=(7, 5))
for i, label in enumerate(class_labels):
    precision, recall, _ = precision_recall_curve(y_test == i, y_pred_probs_bayes_grid[:, i])
    pr_auc = auc(recall, precision)

    # Plot the precision-recall curve
    plot = plt.plot(recall, precision, label=f'{label} (PR AUC = {pr_auc:.3f})')

xlabel = plt.xlabel('Recall')
ylabel = plt.ylabel('Precision')
title = plt.title('Precision-Recall Curve (Hyperparameter Optimized Logistic Regression)')
legend = plt.legend(loc='lower left', framealpha=0.35)
plt.show()
```



The Classification Report

In [ ]:
```
# Classification report
report_dict_grid = classification_report(y_test, y_pred_bayes_grid, digits=3, output_dict=True)

# Calculate PR AUC for each class
precision_0, recall_0, _ = precision_recall_curve(y_test == 0, y_pred_probs_bayes_grid[:, 0])
pr_auc_0 = auc(recall_0, precision_0)

precision_rain, recall_rain, _ = precision_recall_curve(y_test == 1, y_pred_probs_bayes_grid[:, 1])
pr_auc_rain = auc(recall_rain, precision_rain)
```

```
precision_snow, recall_snow, _ = precision_recall_curve(y_test == 2, y_pred_probs_bayes_grid[:, 2])
pr_auc_snow = auc(recall_snow, precision_snow)
# Create a single row dataframe with the required metrics
report_df_bayes_grid = pd.DataFrame({
    'precision_0': [report_dict_grid['0']['precision']],
    'recall_0': [report_dict_grid['0']['recall']],
    'f1_0': [report_dict_grid['0']['f1-score']],
    'pr_auc_0': [pr_auc_0],

    'precision_rain': [report_dict_grid['1']['precision']],
    'recall_rain': [report_dict_grid['1']['recall']],
    'f1_rain': [report_dict_grid['1']['f1-score']],
    'pr_auc_rain': [pr_auc_rain],

    'precision_snow': [report_dict_grid['2']['precision']],
    'recall_snow': [report_dict_grid['2']['recall']],
    'f1_snow': [report_dict_grid['2']['f1-score']],
    'pr_auc_snow': [pr_auc_snow],
    'balanced_accuracy': [balanced_accuracy_score(y_test, y_pred_bayes_grid)],
    'f1_macro': [f1_score(y_test, y_pred_bayes_grid, average='macro')],
    'pr_auc_macro': [average_precision_score(y_test, y_pred_probs_bayes_grid, average='macro')]

})

# Display the DataFrame
print('Hyperparameter Optimized Logistic Regression')
report_df_bayes_grid
```

```
Hyperparameter Optimized Logistic Regression
```

Out[ ]:

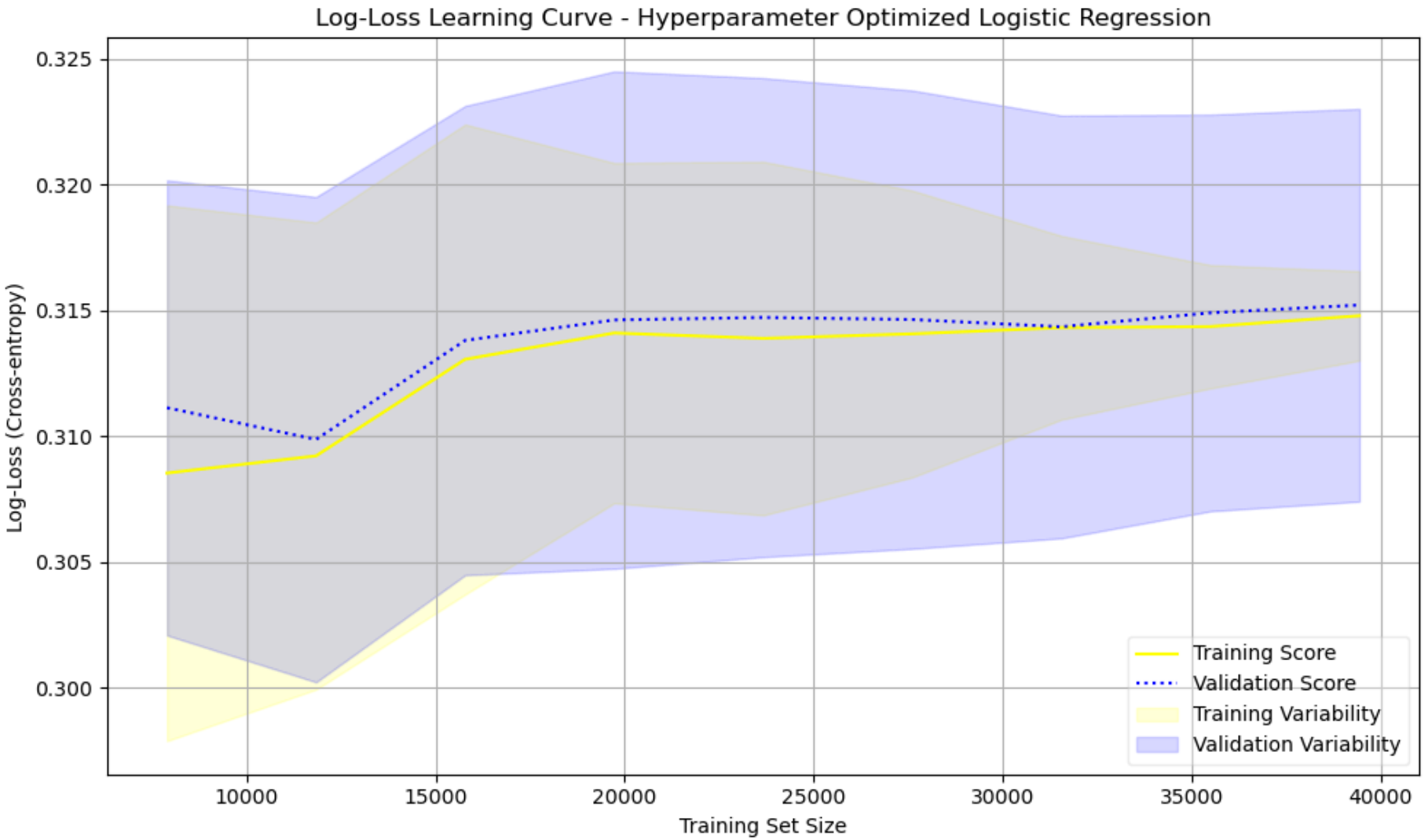
	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_snow	pr_au
0	0.991266	0.84948	0.914913	0.994558	0.364431	0.906618	0.519886	0.628755	0.183673	0.947368	0.307692	0.

The learning curve

In [294…

```
from sklearn.model_selection import learning_curve
stkf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
```

```
train_sizes, train_scores, test_scores = learning_curve(
    bayes_best_model, X, y, cv=stkf, scoring='neg_log_loss', n_jobs=-1,
    train_sizes=np.linspace(0.1, 1.0, 10), shuffle=True, random_state=42
)

train_mean = -np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = -np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

figure = plt.figure(figsize=(10, 6))
plot1 = plt.plot(train_sizes, train_mean, label='Training Score', color='yellow')
plot2 = plt.plot(train_sizes, test_mean, label='Validation Score', color='blue', linestyle=':')
ce1_fill = plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, color='yellow',
                            alpha=0.15, label='Training Variability')
ce2_fill = plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, color='blue',
                            alpha=0.15, label='Validation Variability')

xlabel = plt.xlabel('Training Set Size')
ylabel = plt.ylabel('Log-Loss (Cross-entropy)')
title = plt.title('Log-Loss Learning Curve - Hyperparameter Optimized Logistic Regression')
legend = plt.legend(loc='lower right', framealpha=0.35)
grid = plt.grid(True)
plt.tight_layout()
plt.show()
```



Inferences from the Learning Curve Plot

The log-loss learning curve for the Bayesian-optimized logistic regression model converges to approximately 0.315 for both the training and validation sets when the full dataset is used. The two curves remain closely aligned throughout the plot, indicating that the model does not overfit and maintains stable generalization performance across different training sizes.

Compared to the OvR and SMOTE-oversampled, the log-loss remains within a similar range (~0.3155), indicating that hyperparameter tuning alone did not lead to substantial improvements.

Despite the benefits of Bayesian optimization, the validation variability (blue shaded region) remains relatively high, suggesting that the model's performance on unseen data is still inconsistent.
The results strongly indicate that further improvements using the Logistic Regression algorithm alone are highly unlikely.

This could be due to several factors:

- **Complex, potentially non-linear relationships among the predictors that Logistic Regression fails to capture.**
- **Severe class imbalance, which continues to impact performance despite various balancing strategies.**
- **The algorithm may have reached its full potential given the constraints of this dataset.**

Save the trained model to reuse it later without re-running hyperparameter search

In [ ]:
```
import joblib
joblib.dump(bayes_best_model, "bayes_best_model.pkl")
```

Out[ ]:
```
['bayes_best_model.pkl']
```

Model comparison

A comparison DataFrame was generated for the records, including all important metrics for each logistic regression iteration.

In [337…
```
comparison_df = pd.concat(
    [report_df_standard, report_df_weighted, report_df_smote, report_df_ovr, report_df_bayes_grid],
    axis=0
)
comparison_df.index = ['Standard Logreg', 'Weighted', 'SMOTE', 'OvR', 'Grid']
```

In [339…
```
comparison_df
```

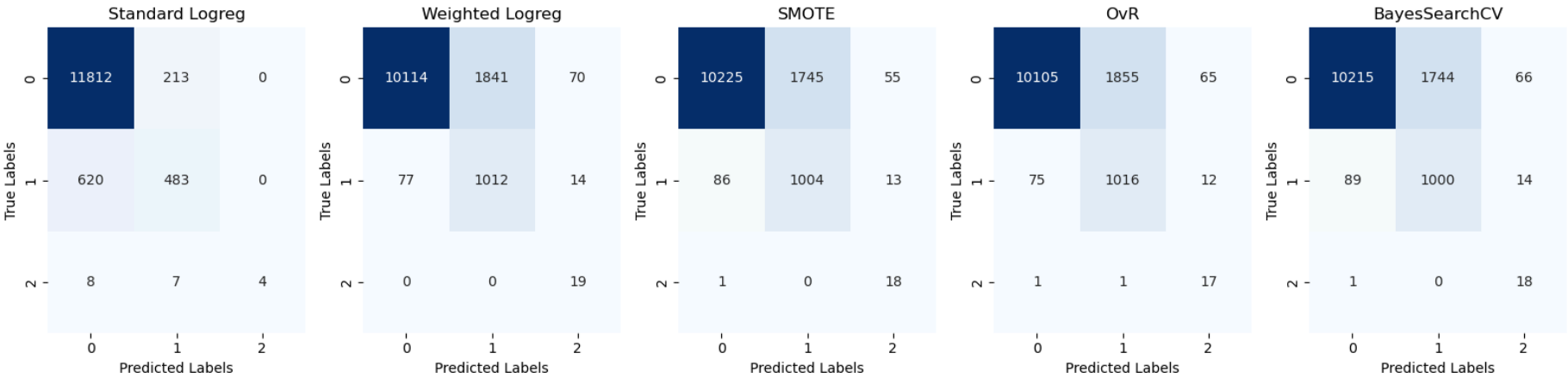
Out[339...

	precision_0	recall_0	f1_0	pr_auc_0	precision_rain	recall_rain	f1_rain	pr_auc_rain	precision_snow	recall_snow	f1_sno
Standard Logreg	0.949518	0.982287	0.965624	0.994828	0.687055	0.437897	0.534884	0.647345	1.000000	0.210526	0.34782
Weighted	0.992444	0.841081	0.910515	0.994658	0.354714	0.917498	0.511628	0.622406	0.184466	1.000000	0.31147
SMOTE	0.991563	0.850312	0.915521	0.994716	0.365224	0.910245	0.521288	0.624033	0.209302	0.947368	0.34285
OvR	0.992535	0.840333	0.910114	0.994825	0.353760	0.921124	0.511195	0.614794	0.180851	0.894737	0.30088
Grid	0.991266	0.849480	0.914913	0.994558	0.364431	0.906618	0.519886	0.628755	0.183673	0.947368	0.30769

A practical approach for evaluating the model's performance would be to plot all the confusion matrices side by side and compare the TPs, FPs and so on, model by model.

In [342...

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# List of confusion matrices and method names
confusion_matrices = [conf_matrix_standard, conf_matrix_weighted, conf_matrix_smote, conf_matrix_ovr, conf_matrix_bayes_grid]
method_names = ['Standard Logreg', 'Weighted Logreg', 'SMOTE', 'OvR', 'BayesSearchCV']

# Plot confusion matrices side by side
fig, axes = plt.subplots(1, len(confusion_matrices), figsize=(16, 4))

for i, ax in enumerate(axes.flat):
    map = sns.heatmap(confusion_matrices[i], annot=True, fmt="d", cmap="Blues", ax=ax, cbar=False)
    title =ax.set_title(method_names[i])
    xlabel = ax.set_xlabel('Predicted Labels')
    ylabel = ax.set_ylabel('True Labels')

plt.tight_layout()
plt.show()
```



The confusion matrices reveal a consistent trade-off between precision and recall across all logistic regression iterations, reinforcing the challenges posed by the severe class imbalance. Each approach—whether class weighting, SMOTE oversampling, OvR, or Bayesian optimization—introduced some improvements but ultimately failed to significantly enhance minority class detection without inflating false positives.

While class balancing techniques like SMOTE and class_weight='balanced' helped increase the model's sensitivity to minority classes, they did so at the cost of introducing more misclassifications in the majority class. Despite hyperparameter tuning (BayesSearchCV), the model's fundamental limitations remained, suggesting that logistic regression, as a linear model, lacks the flexibility to fully capture the complexities of this dataset.

Final Thoughts

In real-world machine learning projects, the goal is often to derive actionable insights, make data-driven recommendations, or deploy models for real-world applications.

However, the primary objective of this project was to explore and practice various classification techniques, **specifically Logistic Regression**, while examining its effectiveness in handling heavily imbalanced datasets. Class imbalance is a common challenge in real-world scenarios, making it a crucial aspect to study when developing classification models.

For this reason, feature importance was not extracted in any iteration, as it falls outside the scope of this exploration. Instead, the focus was on understanding model behavior, diagnosing performance limitations, and experimenting with techniques to enhance classification performance

under challenging conditions rather than deriving domain-specific insights.

This notebook serves as a valuable exercise in assessing the strengths and limitations of Logistic Regression for imbalanced classification problems, paving the way for more advanced modeling approaches in future projects.

I hope you find this notebook helpful.