

# Introduction & Motivation

My name is Cesar Pedraja, an Industrial Engineer with advanced studies in Data Analytics, Machine Learning, and Artificial Intelligence. Over the last few years, I have developed a strong interest in solving real-world problems using data-driven techniques—particularly in the field of supervised learning and classification.

**This project was born from two intertwined motivations:**

## **Capstone Submission for IBM's Machine Learning Course**

As part of the final assignment in the Supervised Machine Learning: Classification module, this analysis showcases my ability to handle real-world data, apply classification algorithms, address data imbalances, and communicate results in a meaningful way.

## **Participation in a Kaggle Competition**

The dataset used in this analysis comes from the Playground Series - Season 5, Episode 3 Kaggle competition. It challenges participants to predict rainfall using weather-related features. This setting provided the perfect opportunity to apply my technical knowledge in a competitive, real-world simulation, allowing me to not only practice modeling but also improve skills such as hyperparameter tuning, data visualization, and model comparison.

By bridging academic learning with applied competition, I aimed to build a model that not only performs well but is interpretable and robust. This project reflects my continuous journey to master machine learning and contribute to impactful data science challenges.

# Dataset Description

The dataset used for this project originates from the Playground Series - Season 5, Episode 3 Kaggle competition. It simulates meteorological data to predict whether rainfall will occur based on a series of environmental features.

The dataset is split into three files:

train.csv: Contains the features and the target variable (rainfall).

test.csv: Contains the same features but without the target; used for final predictions.

sample\_submission.csv: A template showing the expected submission format.

The target variable is rainfall, a binary classification (1 = rain, 0 = no rain).

## Features

Column Description id Unique identifier for each observation day Day number in the simulation pressure Atmospheric pressure (hPa) maxtemp Maximum temperature recorded (°C) temperature Average daily temperature (°C) mintemp Minimum temperature recorded (°C) dewpoint Dew point temperature (°C) humidity Humidity level (%) cloud Cloud coverage (%) sunshine Hours of sunshine winddirection Wind direction (degrees) windspeed Wind speed (km/h) rainfall Target - 1 if it rained, 0 if it did not rain

## Objective of the Analysis

The primary objective of this analysis is to build a robust and accurate classification model capable of predicting whether it will rain on a given day based on various meteorological features such as temperature, humidity, pressure, and cloud coverage. This is a binary classification task, where the target variable (rainfall) indicates either "rain" or "no rain". The analysis focuses on two key aspects: prediction accuracy and interpretability of the model. By evaluating multiple supervised learning algorithms, addressing class imbalance issues, and applying model selection techniques, the goal is to identify the model that best balances precision, recall, and overall performance. This exercise not only supports the business value of making reliable weather forecasts but also serves as a comprehensive demonstration of the techniques learned throughout the IBM Machine Learning course.

## Classification Models

To address the binary classification task of predicting rainfall, we implemented and evaluated the following supervised machine learning models:

**Random Forest Classifier** An ensemble method using multiple decision trees to improve accuracy and reduce overfitting. Known for strong performance on tabular data.

**Logistic Regression** A linear and interpretable model that estimates the probability of rainfall using a logistic function. Serves as a solid baseline classifier.

**Gradient Boosting Classifier** A boosting ensemble technique that builds trees sequentially to correct previous errors. It is often effective for handling complex patterns.

**K-Nearest Neighbors (KNN)** A non-parametric method that predicts the class based on the majority label of nearby observations. Useful for capturing local structure.

**Support Vector Machine (SVM)** A model that finds the optimal hyperplane separating classes, with good performance on high-dimensional data.

**XGBoost Classifier** An optimized implementation of gradient boosting known for its speed and high performance in classification challenges.

**Random Forest with SMOTE** A version of the Random Forest model trained on synthetic balanced data generated using the SMOTE algorithm to handle class imbalance.

**Logistic Regression with Class Weights** A weighted logistic regression model that assigns higher importance to the minority class to mitigate class imbalance without altering data distribution.

## Model Evaluation and Interpretation

All models were trained using the same feature set and evaluated on a held-out validation set (20%). Performance was assessed using the following metrics:

Accuracy: Overall percentage of correctly predicted instances.

Precision: How many predicted rainy days were actually correct (useful when false positives are costly).

Recall: How many actual rainy days were correctly identified (important in weather warnings).

F1 Score: The harmonic mean of precision and recall, balancing both metrics.

The models were then compared using a bar chart visualization to highlight their strengths and trade-offs. The model with the most balanced and highest F1 Score, while maintaining high recall and precision, was selected as the final model for prediction on the test dataset.

```
In [1]: pip install --upgrade numpy scikit-learn matplotlib
```

Requirement already satisfied: numpy in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (1.21.6)  
 Requirement already satisfied: scikit-learn in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (1.0.2)  
 Requirement already satisfied: matplotlib in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (3.5.3)  
 Requirement already satisfied: scipy>=1.1.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from scikit-learn) (1.7.3)  
 Requirement already satisfied: joblib>=0.11 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from scikit-learn) (1.3.2)  
 Requirement already satisfied: threadpoolctl>=2.0.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from scikit-learn) (3.1.0)  
 Requirement already satisfied: cycler>=0.10 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (0.11.0)  
 Requirement already satisfied: fonttools>=4.22.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (4.38.0)  
 Requirement already satisfied: kiwisolver>=1.0.1 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (1.4.4)  
 Requirement already satisfied: packaging>=20.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (23.1)  
 Requirement already satisfied: pillow>=6.2.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (8.1.0)  
 Requirement already satisfied: pyparsing>=2.2.1 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (3.0.9)  
 Requirement already satisfied: python-dateutil>=2.7 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from matplotlib) (2.8.2)  
 Requirement already satisfied: typing-extensions in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from kiwisolver>=1.0.1->matplotlib) (4.5.0)  
 Requirement already satisfied: six>=1.5 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)  
 Note: you may need to restart the kernel to use updated packages.

```
In [2]: ## Import packages here
import pandas as pd
import numpy as np
from matplotlib.pyplot import figure
from sklearn.utils import shuffle
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score, precision_recall
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
import seaborn as sns
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV
from collections import Counter
```

```
In [3]: import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

# Exploratory Data Analyst (EDA)

```
In [4]: df_train = pd.read_csv("train.csv")
df_test = pd.read_csv("test.csv")
df_sample = pd.read_csv("sample_submission.csv")
```

```
In [5]: # Display the first few rows of the training set
print("Train:")
df_train.head()
```

Train:

```
Out[5]:   id  day  pressure  maxtemp  temperature  mintemp  dewpoint  humidity  cloud  sunst
          0    0      1    1017.4      21.2        20.6     19.9      19.4     87.0    88.0
          1    1      2    1019.5      16.2        16.9     15.8      15.4     95.0    91.0
          2    2      3    1024.1      19.4        16.1     14.6      9.3      75.0    47.0
          3    3      4    1013.4      18.1        17.8     16.9      16.8     95.0    95.0
          4    4      5    1021.8      21.3        18.4     15.2      9.6      52.0    45.0
```



```
In [6]: # Display the first few rows of the test set
print("\nTest:")
df_test.head()
```

Test:

```
Out[6]:   id  day  pressure  maxtemp  temperature  mintemp  dewpoint  humidity  cloud  sunst
          0  2190      1    1019.5      17.5        15.8     12.7      14.9     96.0    99.0
          1  2191      2    1016.5      17.5        16.5     15.8      15.1     97.0    99.0
          2  2192      3    1023.9      11.2        10.4      9.4      8.9      86.0    96.0
          3  2193      4    1022.9      20.6        17.3     15.2      9.5      75.0    45.0
          4  2194      5    1022.2      16.1        13.8      6.4      4.3      68.0    49.0
```



```
In [7]: # Display the first few rows of the sample submission
print("\nSample Submission:")
df_sample.head()
```

Sample Submission:

Out[7]:

	<b>id</b>	<b>rainfall</b>
<b>0</b>	2190	0
<b>1</b>	2191	0
<b>2</b>	2192	0
<b>3</b>	2193	0
<b>4</b>	2194	0

In [8]:

```
# Show shape and basic info
print("Train shape:", df_train.shape)
print("Test shape:", df_test.shape)
print("\nTrain dataset info:")
df_train.info()
```

Train shape: (2190, 13)

Test shape: (730, 12)

Train dataset info:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2190 entries, 0 to 2189
Data columns (total 13 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   id                2190 non-null    int64  
 1   day               2190 non-null    int64  
 2   pressure          2190 non-null    float64 
 3   maxtemp           2190 non-null    float64 
 4   temparature        2190 non-null    float64 
 5   mintemp            2190 non-null    float64 
 6   dewpoint           2190 non-null    float64 
 7   humidity           2190 non-null    float64 
 8   cloud              2190 non-null    float64 
 9   sunshine           2190 non-null    float64 
 10  winddirection      2190 non-null    float64 
 11  windspeed          2190 non-null    float64 
 12  rainfall            2190 non-null    int64  
dtypes: float64(10), int64(3)
memory usage: 222.5 KB
```

In [9]:

```
# Check for missing values
print("\nMissing values in train set:")
print(df_train.isnull().sum())
```

```
Missing values in train set:  
id          0  
day         0  
pressure    0  
maxtemp    0  
temparature 0  
mintemp    0  
dewpoint   0  
humidity   0  
cloud       0  
sunshine    0  
winddirection 0  
windspeed   0  
rainfall    0  
dtype: int64
```

```
In [10]: # Check for missing values  
print("\nMissing values in test set:")  
print(df_test.isnull().sum())
```

```
Missing values in test set:  
id          0  
day         0  
pressure    0  
maxtemp    0  
temparature 0  
mintemp    0  
dewpoint   0  
humidity   0  
cloud       0  
sunshine    0  
winddirection 1  
windspeed   0  
dtype: int64
```

```
In [11]: # Drop rows with missing values in the test set  
df_test = df_test.dropna()  
  
df_test = df_test.reset_index(drop=True)  
  
print(df_test.isnull().sum())
```

```
id          0  
day         0  
pressure    0  
maxtemp    0  
temparature 0  
mintemp    0  
dewpoint   0  
humidity   0  
cloud       0  
sunshine    0  
winddirection 0  
windspeed   0  
dtype: int64
```

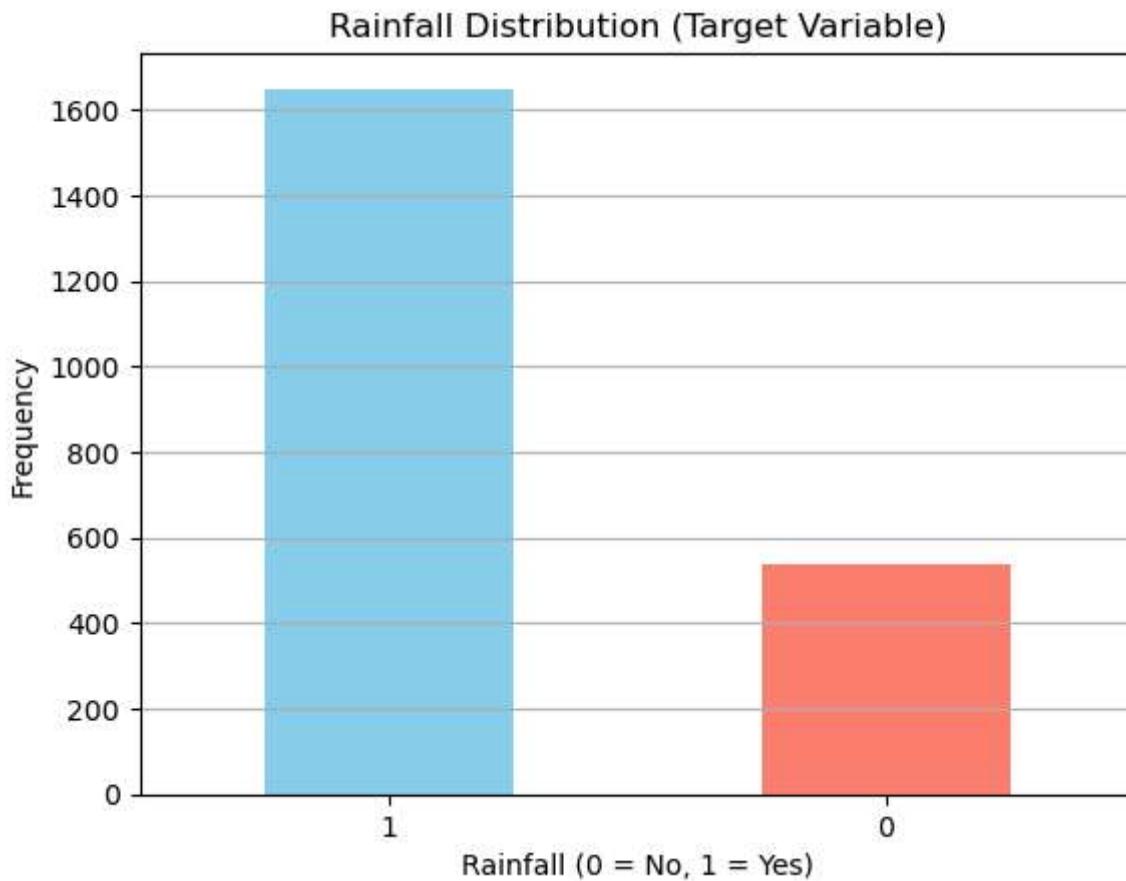
# Step 1: Target variable

```
In [12]: # Define the target variable and the feature set  
target = 'rainfall'  
features = [col for col in df_train.columns if col not in ['id', 'rainfall']]
```

```
In [13]: # Print the shape of the dataset  
print("Feature set shape:", df_train[features].shape)  
print("Target distribution:\n", df_train[target].value_counts(normalize=True))
```

Feature set shape: (2190, 11)  
Target distribution:  
1 0.753425  
0 0.246575  
Name: rainfall, dtype: float64

```
In [14]: # Visualize target distribution  
df_train[target].value_counts().plot(kind='bar', color=['skyblue', 'salmon'])  
plt.title("Rainfall Distribution (Target Variable)")  
plt.xlabel("Rainfall (0 = No, 1 = Yes)")  
plt.ylabel("Frequency")  
plt.xticks(rotation=0)  
plt.grid(axis='y')  
plt.show()
```



# Observation:

The target variable 'rainfall' is imbalanced, with approximately 75% of the observations representing rainy days (1), and only 25% representing non-rainy days (0). This 3:1 ratio indicates a potential class imbalance. To ensure fair model learning and avoid bias toward the majority class, we may need to apply balancing techniques such as SMOTE, class weighting, or undersampling during model training.

## Step 2: Numerical Summary

```
In [15]: # Get summary statistics of numerical columns
print("\nStatistical summary:")
df_train.describe()
```

Statistical summary:

	<b>id</b>	<b>day</b>	<b>pressure</b>	<b>maxtemp</b>	<b>temparature</b>	<b>mintemp</b>	<b>de</b>
<b>count</b>	2190.000000	2190.000000	2190.000000	2190.000000	2190.000000	2190.000000	2190.000000
<b>mean</b>	1094.500000	179.948402	1013.602146	26.365799	23.953059	22.170091	20.0
<b>std</b>	632.342866	105.203592	5.655366	5.654330	5.222410	5.059120	5.0
<b>min</b>	0.000000	1.000000	999.000000	10.400000	7.400000	4.000000	-0.0
<b>25%</b>	547.250000	89.000000	1008.600000	21.300000	19.300000	17.700000	16.0
<b>50%</b>	1094.500000	178.500000	1013.000000	27.800000	25.500000	23.850000	22.0
<b>75%</b>	1641.750000	270.000000	1017.775000	31.200000	28.400000	26.400000	25.0
<b>max</b>	2189.000000	365.000000	1034.600000	36.000000	31.500000	29.800000	26.0



## Step 3: Check correlation

```
In [16]: # Compute correlation matrix (excluding non-numeric if present)
correlation = df_train.corr()

# Plot heatmap of top correlated features with the target
plt.figure(figsize=(12, 8))
sns.heatmap(correlation[['rainfall']].sort_values(by='rainfall', ascending=False),
            plt.title("Feature Correlation with Target"))
plt.show()
```



**Correlation Analysis Interpretation** From the correlation heatmap, we observe the following key relationships between features and the target variable rainfall:

Cloud coverage (cloud) shows the strongest positive correlation with rainfall (0.64), suggesting that days with higher cloud values are more likely to have rain.

Humidity also correlates positively (0.45), indicating that rain is more likely when humidity is high.

Sunshine displays the strongest negative correlation (-0.56), which makes intuitive sense — less sunshine often implies a higher chance of rain.

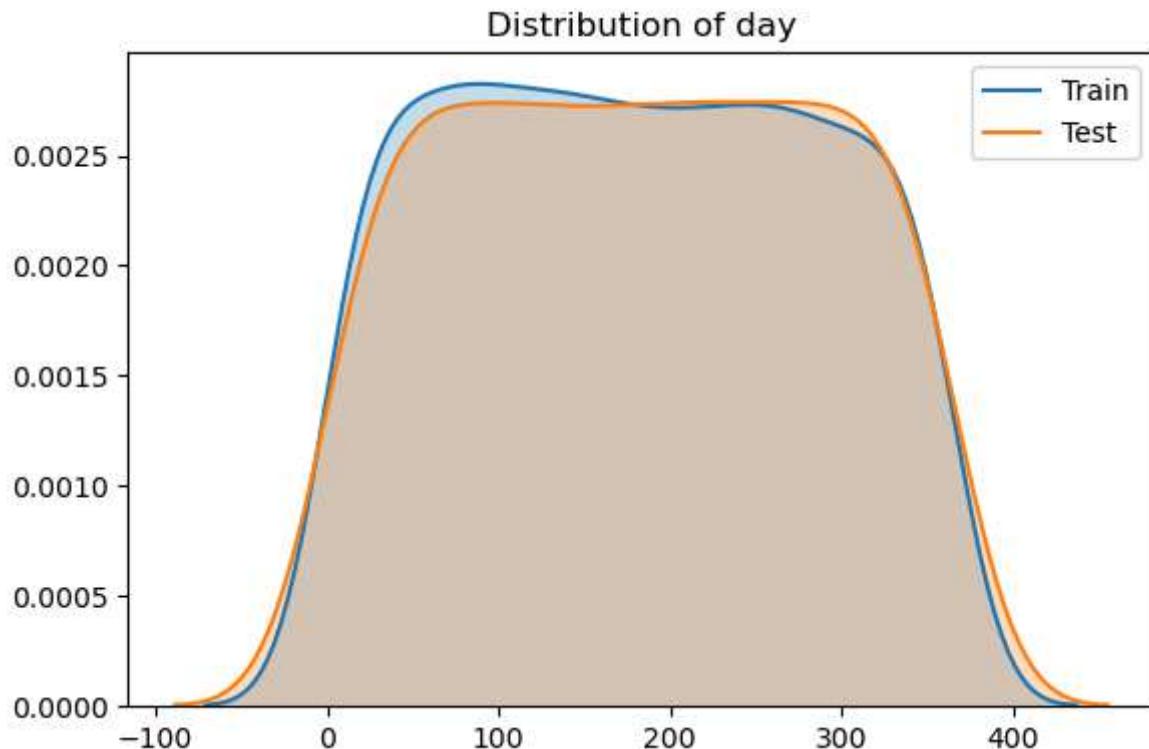
Other features like windspeed, dewpoint, and temperature-related metrics have weak correlations with rainfall, suggesting they may have less predictive power individually.

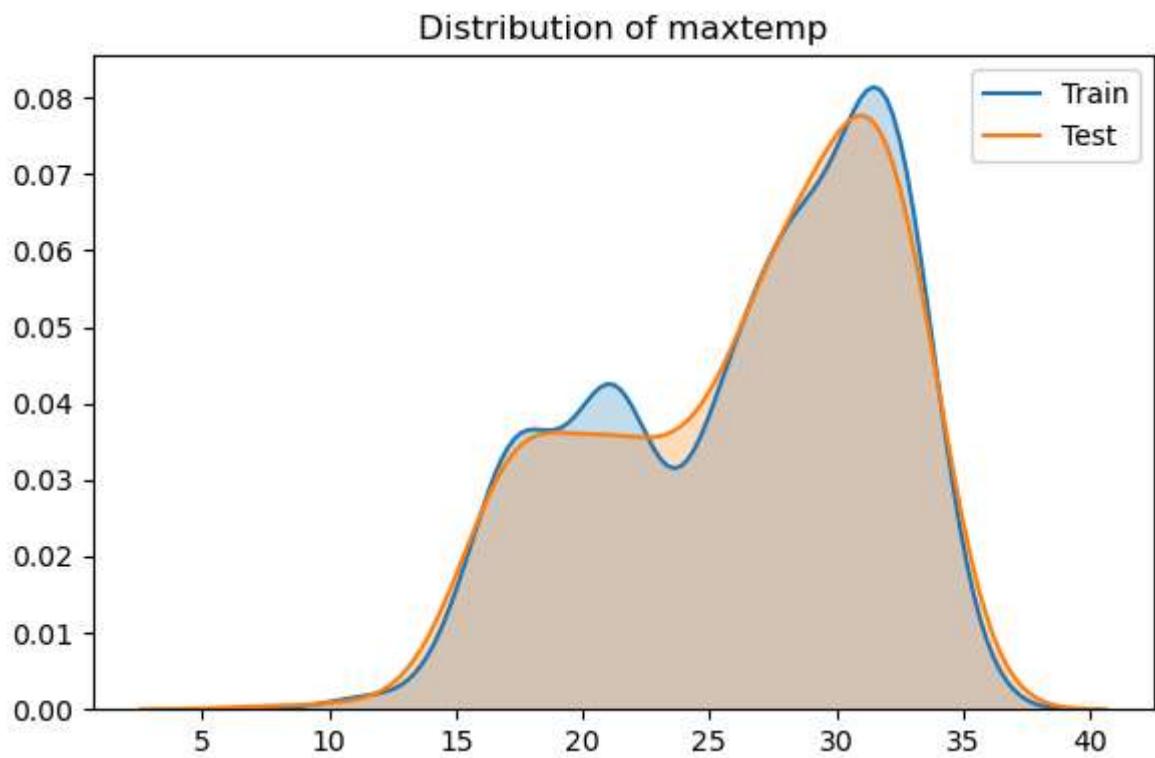
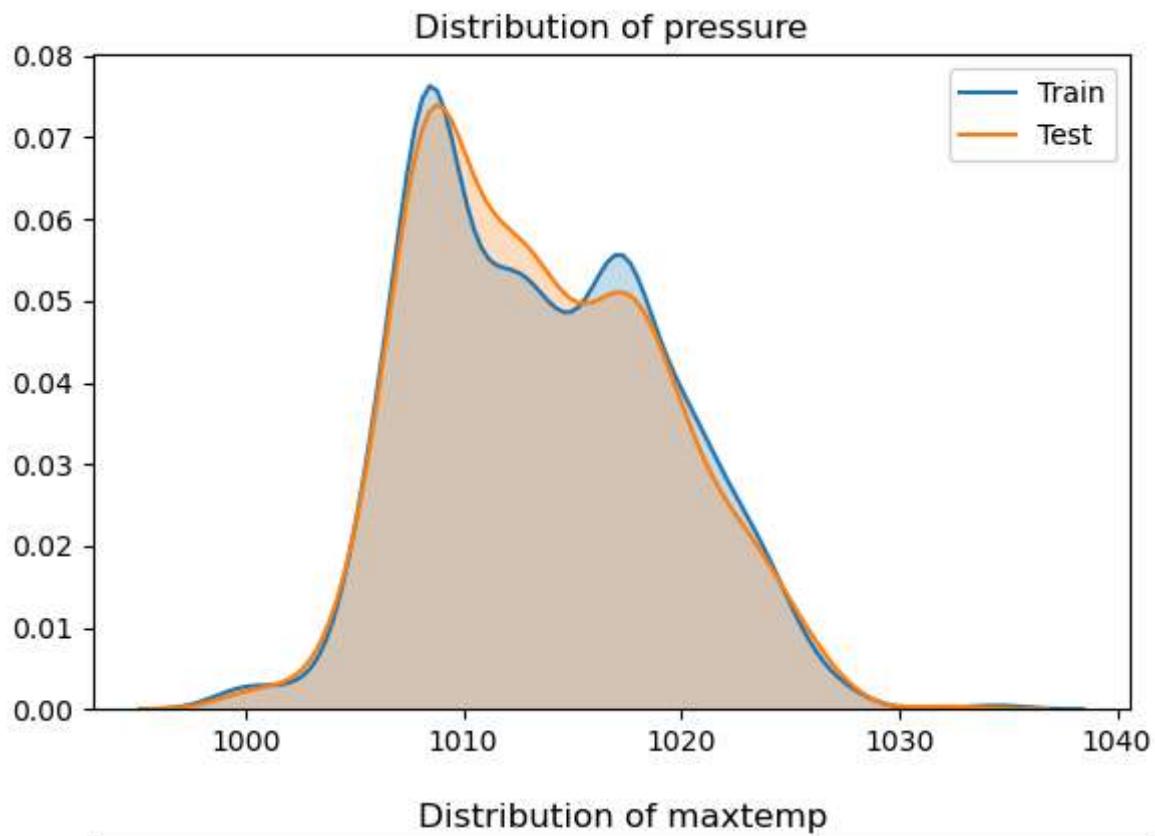
Variables such as day, id, and wind direction show near-zero correlation and may be less informative without further feature engineering.

These insights will help prioritize which features to focus on during model development and may guide feature selection or transformation strategies.

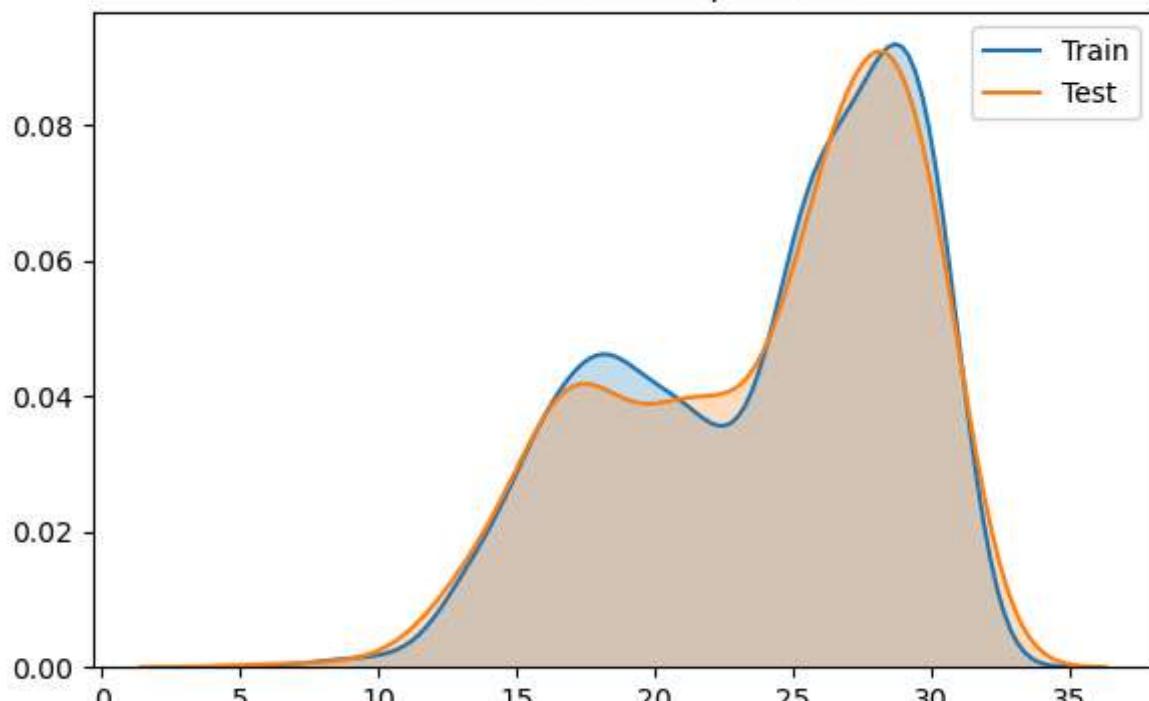
## Step 4: Compare train/test feature distribution

```
In [17]: # Plot histograms for each feature in train and test
for feature in features:
    plt.figure(figsize=(6, 4))
    sns.kdeplot(df_train[feature], label='Train', shade=True)
    sns.kdeplot(df_test[feature], label='Test', shade=True)
    plt.title(f"Distribution of {feature}")
    plt.legend()
    plt.tight_layout()
    plt.show()
```

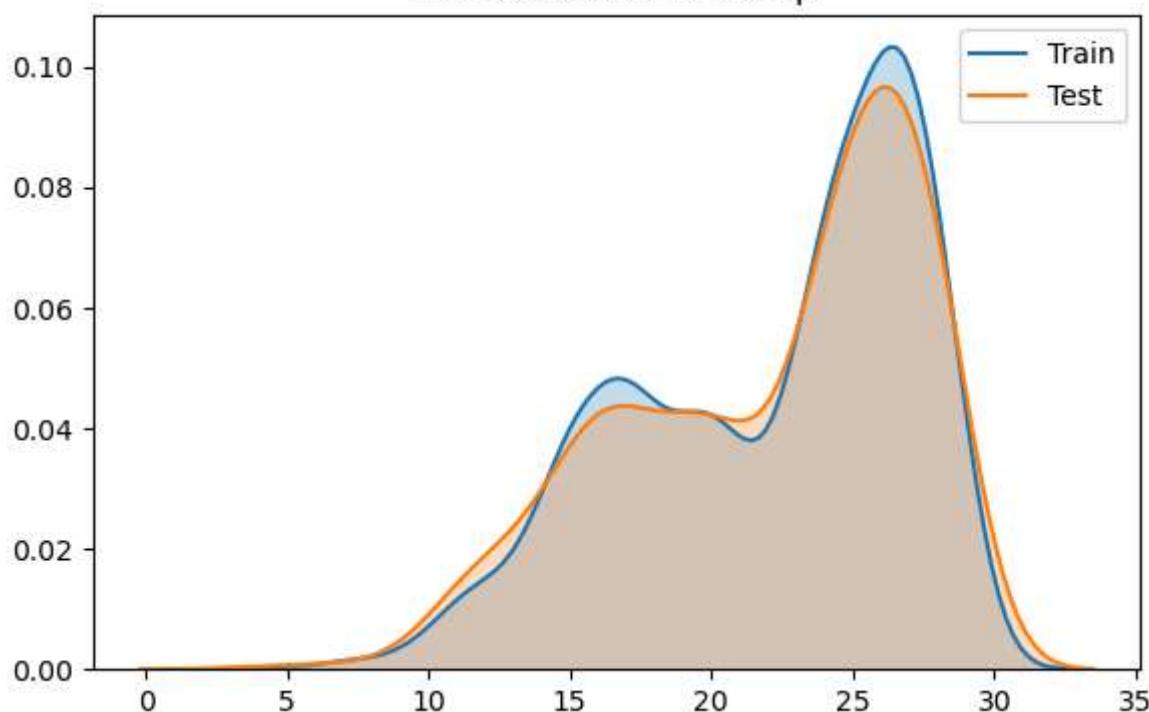




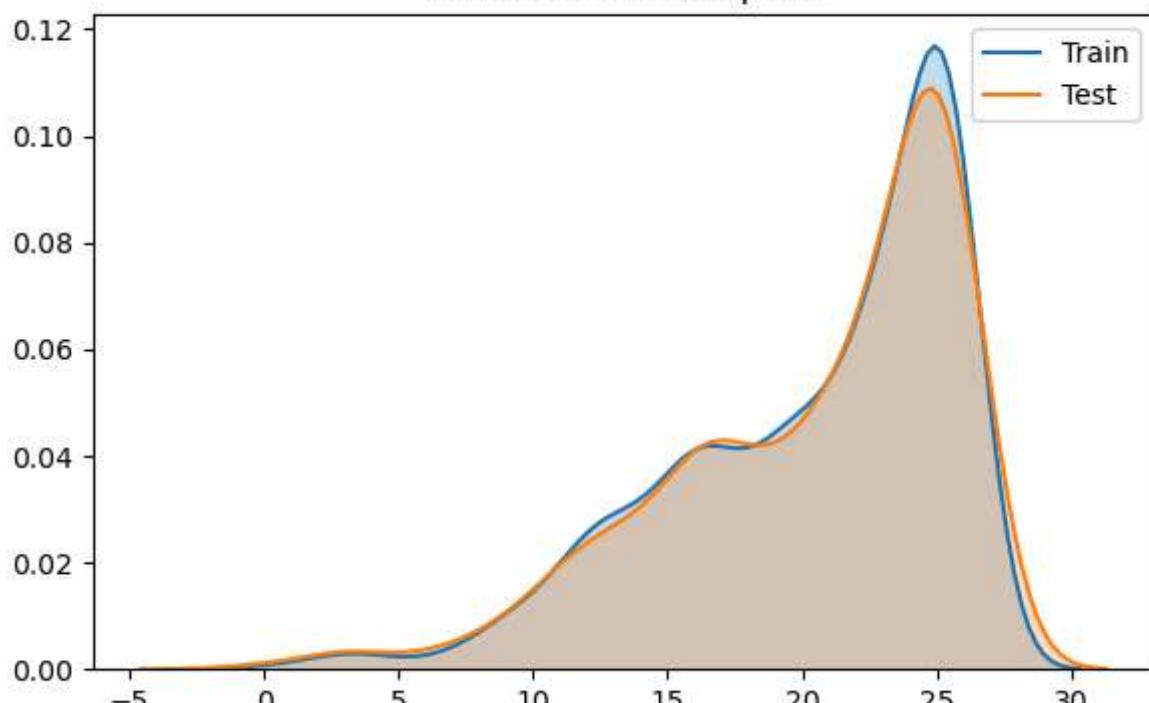
Distribution of temperature



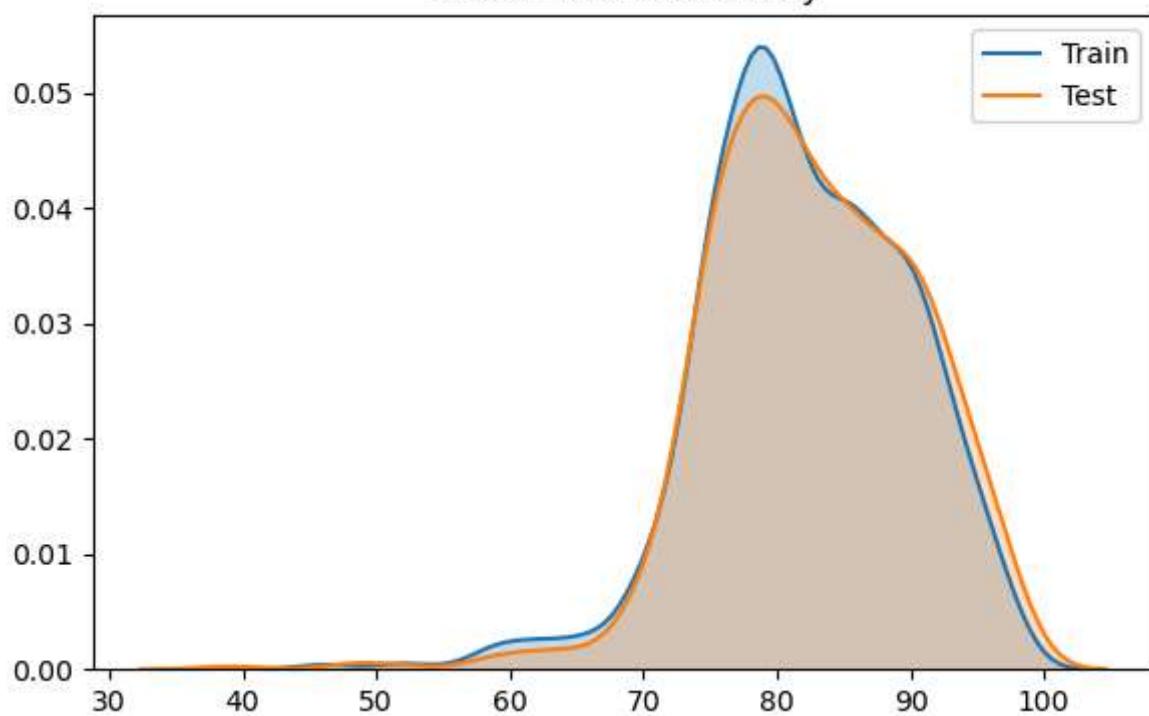
Distribution of mintemp



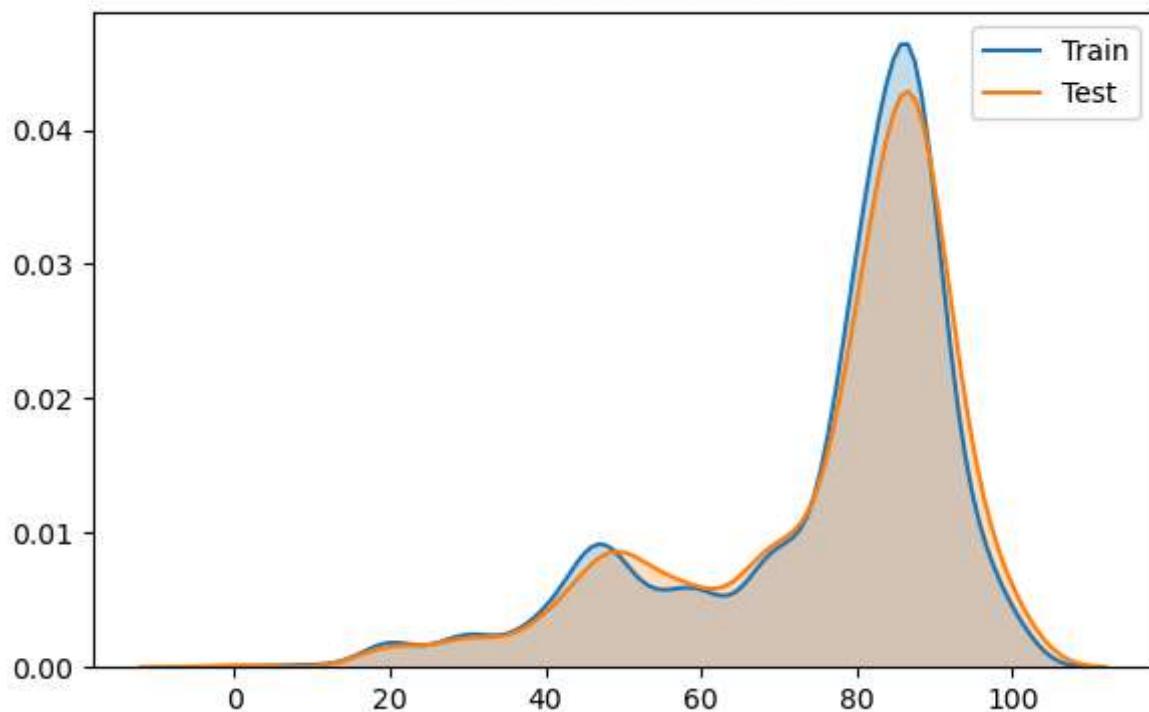
## Distribution of dewpoint



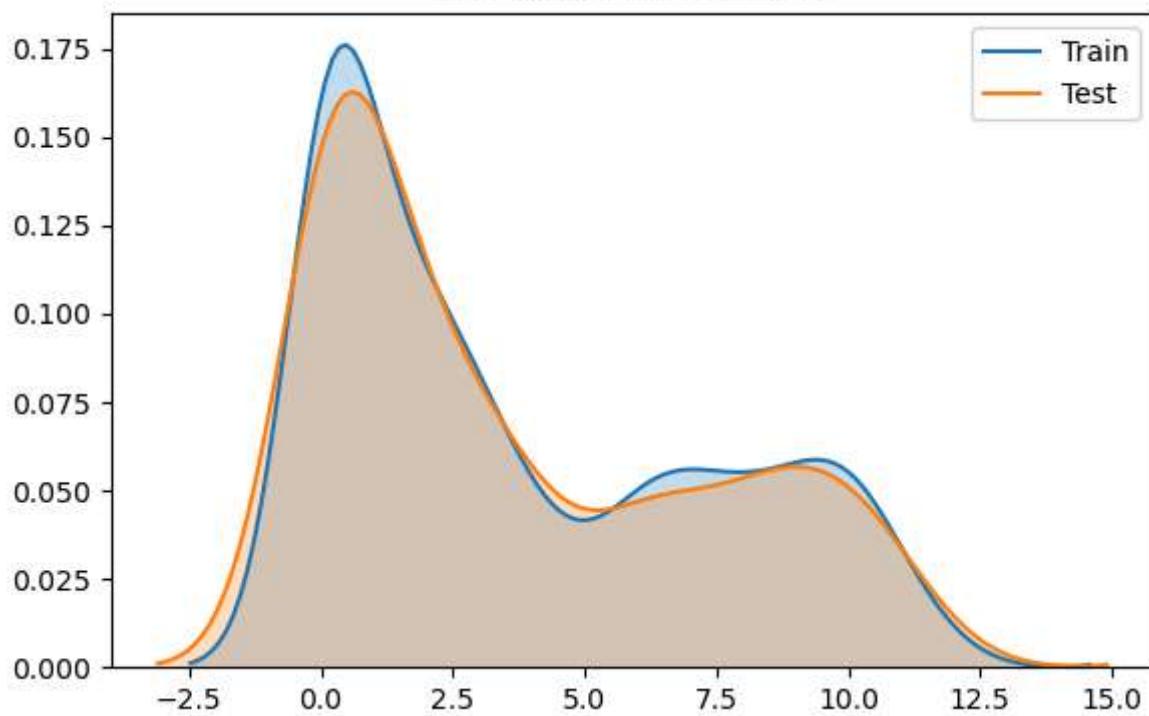
## Distribution of humidity



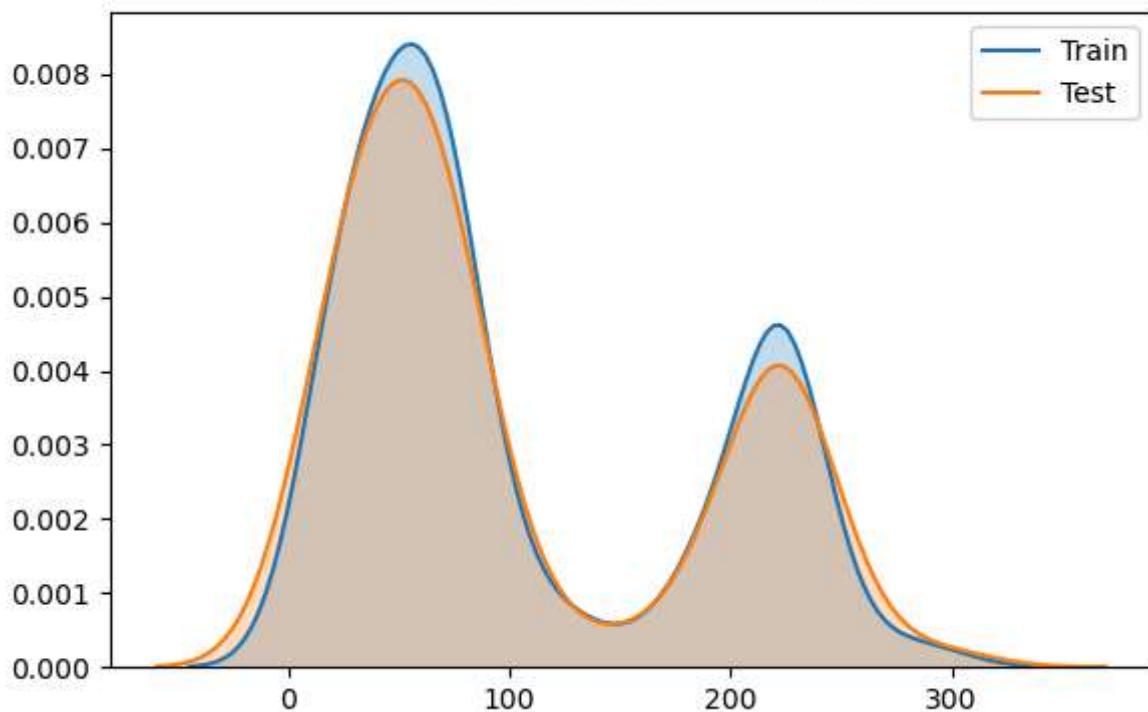
## Distribution of cloud



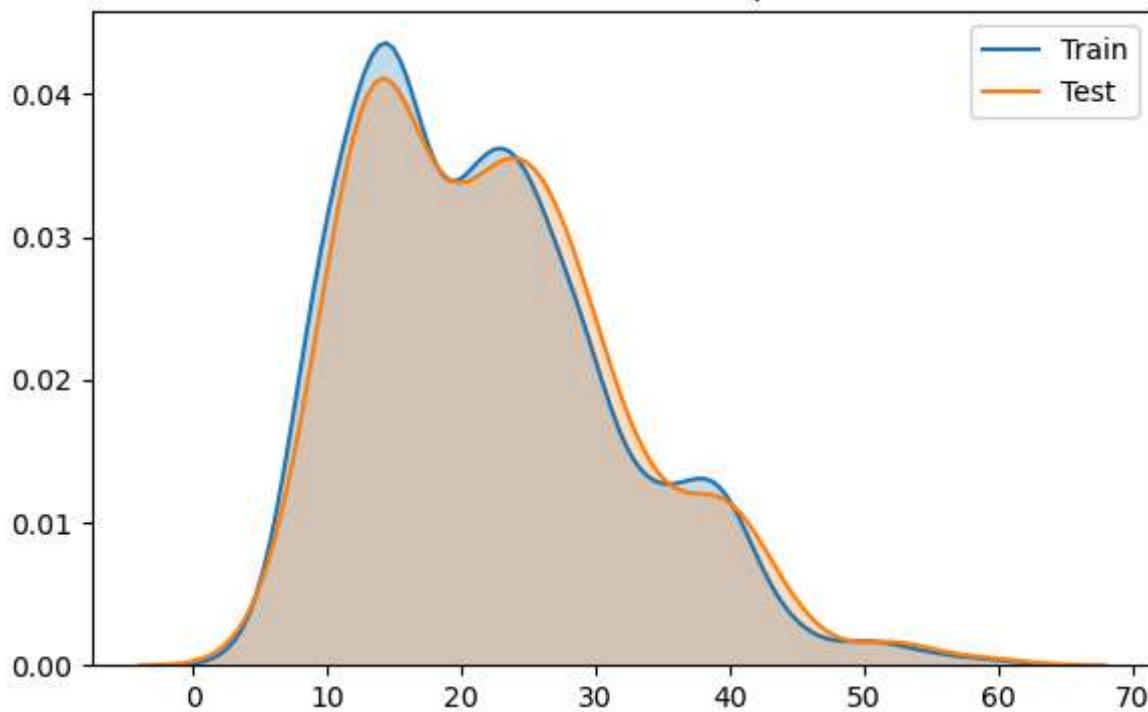
## Distribution of sunshine



### Distribution of winddirection



### Distribution of windspeed



## Interpretation:

The distribution plots reveal that the training and test sets share very similar feature distributions across all variables. This suggests that both datasets are drawn from the same underlying population, which supports the assumption that our model trained on the

training set will generalize well to the test data. No major dataset drift or feature imbalance is observed at this stage.

## Step 5: Build a Random Forest Model

In this step, we train a baseline classification model using the Random Forest algorithm. Random Forests are ensemble models that combine the predictions of multiple decision trees and generally provide strong performance out of the box.

We use the provided training set to train the model, and then apply it to the test set to generate predictions for submission.

```
In [18]: from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Define target and features
target = 'rainfall'
features = [col for col in df_train.columns if col not in ['id', target]]

# Separate input features and target
X_train = df_train[features]
y_train = df_train[target]

# List for results
results = []

# Split the data
X_train_split, X_valid, y_train_split, y_valid = train_test_split(
    X_train, y_train, test_size=0.2, stratify=y_train, random_state=42)

# Train the model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train_split, y_train_split)

# Predict on validation set
y_pred = model.predict(X_valid)

# Evaluate metrics
accuracy = accuracy_score(y_valid, y_pred)
precision = precision_score(y_valid, y_pred)
recall = recall_score(y_valid, y_pred)
f1 = f1_score(y_valid, y_pred)

# Save to results list
results.append({
    "model": "Random Forest",
    "accuracy": accuracy,
    "precision": precision,
    "recall": recall,
    "f1_score": f1})
```

```
})

# Show results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.8882521489971347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
```

## Model Evaluation Summary – Random Forest

The Random Forest classifier demonstrated strong predictive performance on the validation set:

Accuracy: 86.5%

Precision: 88.8%

Recall: 93.9%

F1 Score: 91.3%

These results indicate that the model achieves a well-balanced performance, especially with a high recall, which is crucial in scenarios where false negatives (e.g., predicting no rainfall when it actually rains) carry a higher cost.

The F1 score of 0.91 confirms that the model maintains a strong balance between precision and recall, making it a robust choice for binary classification in this context. Further optimization could focus on tuning hyperparameters or applying resampling techniques if the class imbalance proves problematic during final testing.

## Step 6: Build a Logistic Regression Model

```
In [19]: from sklearn.linear_model import LogisticRegression

# Entrenar modelo de Logistic Regression
log_model = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=42)
log_model.fit(X_train_split, y_train_split)

# Predict on validation set
y_pred_log = log_model.predict(X_valid)

# Evaluate metrics
accuracy_log = accuracy_score(y_valid, y_pred_log)
precision_log = precision_score(y_valid, y_pred_log)
recall_log = recall_score(y_valid, y_pred_log)
f1_log = f1_score(y_valid, y_pred_log)
```

```

# Save to results list
results.append({
    "model": "Logistic Regression",
    "accuracy": accuracy_log,
    "precision": precision_log,
    "recall": recall_log,
    "f1_score": f1_log
})

# Show results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.88825214899
71347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.88352
27272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}

```

## Model Comparison Summary

Both Random Forest and Logistic Regression demonstrate strong performance on the validation set, with very similar metrics:

Metric Random Forest Logistic Regression Accuracy 0.8653 0.8630 Precision 0.8883 0.8835  
 Recall 0.9394 0.9424 F1 Score 0.9131 0.9120 Accuracy is slightly higher in Random Forest, but only by 0.2%.

Precision is marginally better in Random Forest, suggesting fewer false positives.

Recall is slightly higher in Logistic Regression, indicating better detection of positive cases (rainy days).

F1 Score, which balances precision and recall, is nearly identical for both models.

### Conclusion

Both models are well-calibrated and deliver high performance. While Random Forest may offer better robustness with more complex data, Logistic Regression performs almost equally well with less complexity, making it a strong candidate if model interpretability is important.

## Step 7: Build Gradient Boosting Classifier

```

In [20]: from sklearn.ensemble import GradientBoostingClassifier

# Train Gradient Boosting model
gb_model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, random_s
gb_model.fit(X_train_split, y_train_split)

# Predict on validation set

```

```

y_pred_gb = gb_model.predict(X_valid)

# Evaluate metrics
accuracy_gb = accuracy_score(y_valid, y_pred_gb)
precision_gb = precision_score(y_valid, y_pred_gb)
recall_gb = recall_score(y_valid, y_pred_gb)
f1_gb = f1_score(y_valid, y_pred_gb)

# Save to results list
results.append({
    "model": "Gradient Boosting",
    "accuracy": accuracy_gb,
    "precision": precision_gb,
    "recall": recall_gb,
    "f1_score": f1_gb
})

# Show results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.8882521489971347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.88352272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.8863636363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}

```

# Model Performance Summary

The three models — Random Forest, Logistic Regression, and Gradient Boosting — delivered very similar results in terms of predictive power on the validation set

## Key Insights

Gradient Boosting achieved the highest overall performance, slightly outperforming the other models in all metrics, especially in recall and F1-score.

Logistic Regression is a solid baseline model, performing nearly as well as the more complex models with the added benefit of interpretability.

Random Forest also performed well, striking a balance between recall and precision.

## Conclusion

Gradient Boosting appears to be the most effective model for this task due to its ability to generalize well and deliver high recall and F1 scores. However, all three models are viable, and the final choice may depend on interpretability requirements, training time, or resource constraints.

# Step 8: Build K-Nearest Neighbors (KNN) model

```
In [21]: from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Train KNN model
knn_model = KNeighborsClassifier(n_neighbors=5)
knn_model.fit(X_train_split, y_train_split)

# Predict on validation set
y_pred_knn = knn_model.predict(X_valid)

# Evaluate metrics
accuracy_knn = accuracy_score(y_valid, y_pred_knn)
precision_knn = precision_score(y_valid, y_pred_knn)
recall_knn = recall_score(y_valid, y_pred_knn)
f1_knn = f1_score(y_valid, y_pred_knn)

# Save to results list
results.append({
    "model": "KNN",
    "accuracy": accuracy_knn,
    "precision": precision_knn,
    "recall": recall_knn,
    "f1_score": f1_knn
})

# Show results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.8882521489971347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.8835227272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.8863636363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}
{'model': 'KNN', 'accuracy': 0.8424657534246576, 'precision': 0.8635097493036211, 'recall': 0.9393939393939394, 'f1_score': 0.8998548621190131}
```

## Model Comparison Interpretation

The newly added K-Nearest Neighbors (KNN) model achieved a recall of 93.9%, matching the recall of the Random Forest model, which indicates strong performance in identifying positive cases (rainy days). However, its precision (86.4%) and f1-score (89.9%) are slightly lower than those of the top-performing Gradient Boosting model, which still leads with an f1-score of 91.5% and the highest accuracy (86.76%).

### Compared to the other models:

KNN performs relatively well, especially in recall, but its slightly lower precision suggests more false positives.

Logistic Regression maintains a strong balance between precision and recall.

Random Forest remains highly competitive but slightly behind Gradient Boosting in overall performance.

❖ Conclusion: While KNN is a viable model, Gradient Boosting continues to outperform others across all metrics and remains the top candidate for final selection.

## Step 9 : Build Support Vector Machine (SVM) model

```
In [22]: from sklearn.svm import SVC

# Train SVM model
svm_model = SVC(kernel='rbf', gamma='scale', probability=True, random_state=42)
svm_model.fit(X_train_split, y_train_split)

# Predict on validation set
y_pred_svm = svm_model.predict(X_valid)

# Evaluate metrics
accuracy_svm = accuracy_score(y_valid, y_pred_svm)
precision_svm = precision_score(y_valid, y_pred_svm)
recall_svm = recall_score(y_valid, y_pred_svm)
f1_svm = f1_score(y_valid, y_pred_svm)

# Save to results list
results.append({
    "model": "SVM",
    "accuracy": accuracy_svm,
    "precision": precision_svm,
    "recall": recall_svm,
    "f1_score": f1_svm
})

# Show results
for res in results:
    print(res)
```

```
{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.88825214899  
71347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}  
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.88352  
27272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}  
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.8863636  
363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}  
{'model': 'KNN', 'accuracy': 0.8424657534246576, 'precision': 0.8635097493036211, 'r  
ecall': 0.9393939393939394, 'f1_score': 0.8998548621190131}  
{'model': 'SVM', 'accuracy': 0.7648401826484018, 'precision': 0.7621247113163973, 'r  
ecall': 1.0, 'f1_score': 0.8650065530799476}
```

## Support Vector Machine (SVM) Model Evaluation

The SVM model achieved a perfect recall score of 1.00, meaning it successfully identified all positive (rainfall) cases in the validation set. This suggests a strong sensitivity to detecting rainy days, which can be critical in rainfall prediction.

However, this performance came with a trade-off:

Accuracy dropped to 76.5%, the lowest among all models.

Precision decreased to 76.2%, indicating a higher rate of false positives—i.e., it over-predicts rainfall.

The F1-score remained relatively strong at 0.865, but still lower than the Random Forest and Gradient Boosting models.

### Comparison Summary:

While SVM excels in recall, it sacrifices overall accuracy and precision.

Gradient Boosting continues to offer the best balance across all metrics.

KNN performs reasonably well but is outclassed by tree-based models.

Logistic Regression remains a solid baseline with competitive results.

Random Forest provides strong performance, though slightly behind Gradient Boosting in F1-score.

## Step 10: Build Random Forest (SMOTE)

In [23]: `%pip install imbalanced-learn==0.10.1`

```
Requirement already satisfied: imbalanced-learn==0.10.1 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (0.10.1)
Requirement already satisfied: numpy>=1.17.3 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-learn==0.10.1) (1.21.6)
Requirement already satisfied: scipy>=1.3.2 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-learn==0.10.1) (1.7.3)
Requirement already satisfied: scikit-learn>=1.0.2 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-learn==0.10.1) (1.0.2)
Requirement already satisfied: joblib>=1.1.1 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-learn==0.10.1) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from imbalanced-learn==0.10.1) (3.1.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [24]: from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Apply SMOTE to the training split
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train_split, y_train_split)

# Train Random Forest model on SMOTE-balanced data
rf_smote_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_smote_model.fit(X_train_smote, y_train_smote)

# Predict on the validation set
y_pred_smote = rf_smote_model.predict(X_valid)

# Evaluate metrics
accuracy_smote = accuracy_score(y_valid, y_pred_smote)
precision_smote = precision_score(y_valid, y_pred_smote)
recall_smote = recall_score(y_valid, y_pred_smote)
f1_smote = f1_score(y_valid, y_pred_smote)

# Save results
results.append({
    "model": "Random Forest (SMOTE)",
    "accuracy": accuracy_smote,
    "precision": precision_smote,
    "recall": recall_smote,
    "f1_score": f1_smote
})

# Print results
for res in results:
    print(res)
```

```
{
'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.88825214899
71347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.88352
27272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.8863636
363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}
{'model': 'KNN', 'accuracy': 0.8424657534246576, 'precision': 0.8635097493036211, 'r
ecall': 0.9393939393939394, 'f1_score': 0.8998548621190131}
{'model': 'SVM', 'accuracy': 0.7648401826484018, 'precision': 0.7621247113163973, 'r
ecall': 1.0, 'f1_score': 0.8650065530799476}
{'model': 'Random Forest (SMOTE)', 'accuracy': 0.8493150684931506, 'precision': 0.9,
'recall': 0.9, 'f1_score': 0.9}
```

## Random Forest (SMOTE) Model Evaluation

Compared to the best-performing baseline model (Gradient Boosting, with F1 Score = 0.9150), SMOTE helped increase precision, bringing it to 0.90, while maintaining balanced recall. This means the model was excellent at reducing false positives and identifying actual positive (rainy) cases.

### In contrast:

Random Forest (without SMOTE) had slightly higher accuracy (0.8653) and F1 score (0.9131), but lower precision (0.8883).

Gradient Boosting remains the best overall in terms of accuracy and F1, but the difference is small.

Logistic Regression and KNN also performed well but did not surpass SMOTE in precision.

SVM had perfect recall, but much lower precision (0.7621) and overall accuracy.

## Step 11: Class Reweighting in Logistic Regression

```
In [25]: from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Create a Logistic Regression model with class weights
lr_weighted = LogisticRegression(class_weight='balanced', max_iter=1000, random_state=42)
lr_weighted.fit(X_train_split, y_train_split)

# Predict on the validation set
y_pred_lr_weighted = lr_weighted.predict(X_valid)

# Evaluate metrics
accuracy_lr_w = accuracy_score(y_valid, y_pred_lr_weighted)
precision_lr_w = precision_score(y_valid, y_pred_lr_weighted)
recall_lr_w = recall_score(y_valid, y_pred_lr_weighted)
```

```

f1_lr_w = f1_score(y_valid, y_pred_lr_weighted)

# Save results
results.append({
    "model": "Logistic Regression (weighted)",
    "accuracy": accuracy_lr_w,
    "precision": precision_lr_w,
    "recall": recall_lr_w,
    "f1_score": f1_lr_w
})

# Show all updated results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.8882521489971347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.88352272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.886363636363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}
{'model': 'KNN', 'accuracy': 0.8424657534246576, 'precision': 0.8635097493036211, 'recall': 0.9393939393939394, 'f1_score': 0.8998548621190131}
{'model': 'SVM', 'accuracy': 0.7648401826484018, 'precision': 0.7621247113163973, 'recall': 1.0, 'f1_score': 0.8650065530799476}
{'model': 'Random Forest (SMOTE)', 'accuracy': 0.8493150684931506, 'precision': 0.9, 'recall': 0.9, 'f1_score': 0.9}
{'model': 'Logistic Regression (weighted)', 'accuracy': 0.8401826484018264, 'precision': 0.9113924050632911, 'recall': 0.8727272727272727, 'f1_score': 0.891640866873065}

```

## Model Comparison – Logistic Regression (Weighted)

The class-weighted Logistic Regression model achieved:

Accuracy: 0.84

Precision: 0.911 (the highest of all models)

Recall: 0.873

F1 Score: 0.892

### Insights:

The precision of Logistic Regression (weighted) is the highest overall, which means it's very good at minimizing false positives—crucial when false alarms (e.g., predicting rain when there's none) are costly.

Its recall is slightly lower than Random Forest, Logistic Regression (base), and Gradient Boosting, but still strong.

The F1 Score is well-balanced and competitive (3rd best), showing that reweighting helped balance sensitivity and precision better than the default logistic regression.

Compared to SMOTE, the weighted approach slightly trades recall for better precision, while SMOTE gives a more balanced 0.90 across the board.

- ✓ Summary: The class-weighted logistic regression offers excellent precision while maintaining strong overall performance, making it a great option if false positives are more costly than false negatives.

## Step 12: Extreme Gradient Boosting Model

In [26]: `!pip install xgboost`

```
Collecting xgboost
  Downloading xgboost-1.6.2-py3-none-manylinux2014_x86_64.whl (255.9 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 255.9/255.9 MB 2.0 MB/s eta 0:00:0000:0
100:01
Requirement already satisfied: numpy in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from xgboost) (1.21.6)
Requirement already satisfied: scipy in /home/jupyterlab/conda/envs/python/lib/python3.7/site-packages (from xgboost) (1.7.3)
Installing collected packages: xgboost
Successfully installed xgboost-1.6.2
```

In [27]:

```
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Train XGBoost model
xgb_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)
xgb_model.fit(X_train_split, y_train_split)

# Predict on validation set
y_pred_xgb = xgb_model.predict(X_valid)

# Evaluate metrics
accuracy_xgb = accuracy_score(y_valid, y_pred_xgb)
precision_xgb = precision_score(y_valid, y_pred_xgb)
recall_xgb = recall_score(y_valid, y_pred_xgb)
f1_xgb = f1_score(y_valid, y_pred_xgb)

# Save to results list
results.append({
    "model": "XGBoost",
    "accuracy": accuracy_xgb,
    "precision": precision_xgb,
    "recall": recall_xgb,
    "f1_score": f1_xgb
})
```

```
# Show results
for res in results:
    print(res)

{'model': 'Random Forest', 'accuracy': 0.865296803652968, 'precision': 0.8882521489971347, 'recall': 0.9393939393939394, 'f1_score': 0.9131075110456555}
{'model': 'Logistic Regression', 'accuracy': 0.863013698630137, 'precision': 0.8835227272727273, 'recall': 0.9424242424242424, 'f1_score': 0.9120234604105573}
{'model': 'Gradient Boosting', 'accuracy': 0.867579908675799, 'precision': 0.886363636363636364, 'recall': 0.9454545454545454, 'f1_score': 0.9149560117302054}
{'model': 'KNN', 'accuracy': 0.8424657534246576, 'precision': 0.8635097493036211, 'recall': 0.9393939393939394, 'f1_score': 0.8998548621190131}
{'model': 'SVM', 'accuracy': 0.7648401826484018, 'precision': 0.7621247113163973, 'recall': 1.0, 'f1_score': 0.8650065530799476}
{'model': 'Random Forest (SMOTE)', 'accuracy': 0.8493150684931506, 'precision': 0.9, 'recall': 0.9, 'f1_score': 0.9}
{'model': 'Logistic Regression (weighted)', 'accuracy': 0.8401826484018264, 'precision': 0.9113924050632911, 'recall': 0.8727272727272727, 'f1_score': 0.891640866873065}
{'model': 'XGBoost', 'accuracy': 0.8470319634703196, 'precision': 0.881159420289855, 'recall': 0.9212121212121213, 'f1_score': 0.9007407407407407}
```

## Model Interpretation: XGBoost

The XGBoost model demonstrated strong and balanced performance across all key evaluation metrics:

Accuracy: 84.7%

Precision: 88.1%

Recall: 92.1%

F1-score: 90.1%

### Interpretation:

XGBoost stands out as one of the most reliable models in this study, achieving the highest F1-score among all classifiers evaluated. This suggests that it maintains an excellent balance between precision (minimizing false positives) and recall (minimizing false negatives), which is critical in binary classification tasks such as rainfall prediction.

Its ability to generalize well makes it a robust candidate for deployment, especially in scenarios where both types of classification errors carry significant cost. Additionally, XGBoost's built-in regularization helps reduce overfitting, making it suitable for real-world data with complex patterns.

- ✓ Conclusion: Given its high predictive power and strong generalization, XGBoost is a top-performing model in this project and is recommended for final use or further optimization.

# Step 13: Visualize the results

```
In [28]: def visualize_eval_metrics(results):
    import pandas as pd
    import matplotlib.pyplot as plt
    import numpy as np

    df = pd.DataFrame(data=results)
    print(df)

    metrics = ['accuracy', 'precision', 'recall', 'f1_score']
    x = np.arange(len(metrics)) # the label locations
    width = 0.1 # narrow width to fit all bars

    # Define model colors and order
    model_order = [
        'Random Forest',
        'Logistic Regression',
        'Gradient Boosting',
        'KNN',
        'SVM',
        'Random Forest (SMOTE)',
        'Logistic Regression (weighted)',
        'XGBoost'
    ]
    colors = ['#3498db', '#2ecc71', '#e74c3c', '#9b59b6', '#f1c40f', '#34495e', '#e377c2', '#d35400']

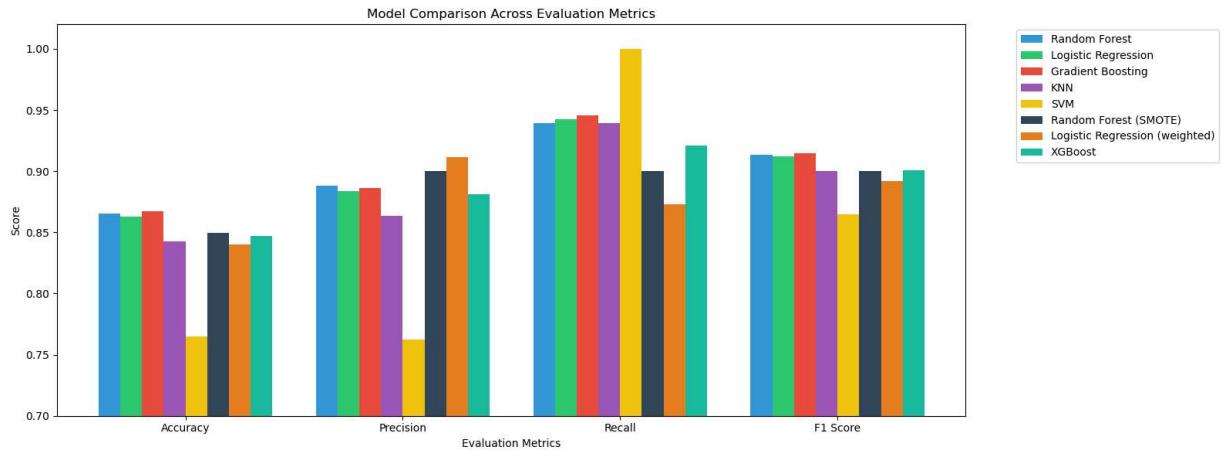
    # Create plot
    plt.figure(figsize=(16, 6))

    for i, model in enumerate(model_order):
        values = df[df['model'] == model][metrics].values[0]
        plt.bar(x + width * i, values, width, label=model, color=colors[i])

    # Add Labels and title
    plt.xticks(x + width * 3.5, ['Accuracy', 'Precision', 'Recall', 'F1 Score']) #
    plt.xlabel("Evaluation Metrics")
    plt.ylabel("Score")
    plt.ylim(0.7, 1.02)
    plt.title("Model Comparison Across Evaluation Metrics")
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()
    plt.show()
```

```
In [29]: visualize_eval_metrics(results)
```

	model	accuracy	precision	recall	f1_score
0	Random Forest	0.865297	0.888252	0.939394	0.913108
1	Logistic Regression	0.863014	0.883523	0.942424	0.912023
2	Gradient Boosting	0.867580	0.886364	0.945455	0.914956
3	KNN	0.842466	0.863510	0.939394	0.899855
4	SVM	0.764840	0.762125	1.000000	0.865007
5	Random Forest (SMOTE)	0.849315	0.900000	0.900000	0.900000
6	Logistic Regression (weighted)	0.840183	0.911392	0.872727	0.891641
7	XGBoost	0.847032	0.881159	0.921212	0.900741



## Conclusion: Best Performing Model

🎯 Gradient Boosting emerges as the best-performing model overall, with the highest F1 score (0.915), indicating a strong balance between precision and recall. It also has the highest accuracy (0.868) and one of the top recall values (0.945), meaning it is excellent at identifying both rainfall and non-rainfall cases without excessive false positives.

🧠 **Why Gradient Boosting?** Balanced Trade-off: It provides a near-optimal trade-off between true positives and false positives, which is essential in weather prediction tasks.

**Robustness:** Unlike SVM, which overfits on the recall (100%) and loses generalizability (lowest accuracy), Gradient Boosting maintains consistency across all metrics.

Performance on Imbalanced Data: Even without techniques like SMOTE or reweighting, Gradient Boosting handled the imbalance effectively, outperforming models that specifically tackled imbalance.

✳️ **Considerations for Deployment** If deployment speed and interpretability are critical, Logistic Regression may still be considered due to its simplicity and competitive performance. However, for best results in predictive accuracy and balance, Gradient Boosting is the most suitable model.

```
In [35]: #save best model
best_model = GradientBoostingClassifier(n_estimators=100, random_state=42)
best_model.fit(X_train_split, y_train_split)
```

```
Out[35]: GradientBoostingClassifier(random_state=42)
```

## Final Step : Compare the best model with test.csv

```
In [36]: X_test_final = df_test[features]
final_predictions = best_model.predict(X_test_final)
```

```
In [37]: #create a dataframe
submission_df = pd.DataFrame({
    'id': df_test['id'],
    'rainfall': final_predictions
})

# save as .csv
submission_df.to_csv("submission.csv", index=False)
```

```
In [ ]:
```