

1. Important Information

The research goal of this notebook is to develop and evaluate machine learning models for predicting weekly rice pest outbreaks using historical field observations and associated environmental conditions. The analysis focuses on constructing a practical decision support component that can help identify high-risk weeks and locations, thereby supporting more timely and targeted pest management interventions. The dataset used in this notebook is the publicly available [ICAR-CRIDA](#) rice pest and disease dataset, which has been aggregated and shared via [Kaggle](#) as a unified CSV file for multiple pests, locations, and years.

This notebook operationalises the data preparation, modelling, and evaluation pipeline described in the research proposal and methodology chapters. It implements the planned steps of exploratory data analysis, feature engineering, target construction (Outbreak), baseline modelling with logistic regression, advanced ensemble models (Random Forest and Gradient Boosting), and a sequence-based Long Short-Term Memory (LSTM) model for a selected pest–location series. Throughout, the notebook aims to remain aligned with the study's objectives of building transparent, reproducible, and practically interpretable pest outbreak prediction models for rice production systems.

2. Loading The Data

Import all of the necessary libraries:

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import plotly.express as px
```

Load the data from the '.csv' file into a 'pandas dataframe'.

```
In [2]: data = pd.read_csv('RICE.csv')
print(data.info())
data.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19404 entries, 0 to 19403
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Observation Year    19404 non-null   int64  
 1   Standard Week     19404 non-null   int64  
 2   Pest Value        19404 non-null   float64 
 3   Collection Type   19404 non-null   object  
 4   MaxT              19404 non-null   float64 
 5   MinT              19404 non-null   float64 
 6   RH1(%)            19404 non-null   float64 
 7   RH2(%)            19404 non-null   float64 
 8   RF(mm)            19404 non-null   float64 
 9   WS(kmph)          19404 non-null   float64 
 10  SSH(hrs)          19404 non-null   float64 
 11  EVP(mm)           19404 non-null   float64 
 12  PEST NAME         19404 non-null   object  
 13  Location           19404 non-null   object  
dtypes: float64(9), int64(2), object(3)
memory usage: 2.1+ MB
None
```

Out[2]:

	Observation Year	Standard Week	Pest Value	Collection Type	MaxT	MinT	RH1(%)	RH2(%)	RF(mm)
0	2003	1	0.0	Number/hill	27.9	14.8	94.7	51.3	0.0
1	2003	2	0.0	Number/hill	27.2	15.0	93.9	53.1	0.0
2	2003	3	0.0	Number/hill	28.7	18.3	94.1	56.7	0.6
3	2003	4	0.0	Number/hill	25.3	16.4	90.9	57.4	0.3
4	2003	5	0.0	Number/hill	28.8	18.7	95.7	55.0	0.0

The loaded dataset contains 19404 weekly observations of rice pest monitoring, each represented as a row with 14 attributes covering temporal information, pest measurements, collection procedures, weather conditions, and spatial identifiers. All features are fully populated with no missing values, and the schema combines three integer fields, nine continuous numerical variables, and three categorical variables, which is well suited to supervised machine learning on tabular data. The Pest Value attribute is a continuous measurement of pest abundance or damage, and in subsequent steps it is transformed into a binary Outbreak label so that the models can explicitly learn to distinguish outbreak from non-outbreak conditions for decision support.

3. Data Exploration

3.1 Attribute Analysis

Understanding Each Feature

- **Observation Year:** The fiscal year that the observation / reading was made.

- **Standard Week:** The week of the year that the observation / reading was made (Ranging from 1 to 52).
- **Pest Value:** The numerical value of the reading that was taken. This is used to create the target attribute.
- **Collection Type:** The process / procedure used to take the reading.
- **MaxT:** The maximum temperature during the respective week.
- **MinT:** The minimum temperature during the respective week.
- **RH1(%):** The maximum relative humidity during the respective week.
- **RH2(%):** The minimum relative humidity during the respective week.
- **RF(mm):** The rainfall (in mm) during the respective week.
- **WS(kmph):** The wind speed (in kmph) during the respective week.
- **SSH(hrs):** The average sunshine hours per day during the respective week.
- **EVP(mm):** The evaporation (in mm) during the respective week.
- **Pest Name:** The recorded pest's name.
- **Location:** The location where the reading was collected / recorded.

Coverage

- **Temporal Coverage Start Date:** 1960/12/31
- **Temporal Coverage End Date:** 2011/12/31
- **Geospatial Coverage:** India

In [3]: `data.describe()`

	Observation Year	Standard Week	Pest Value	MaxT	MinT	RH
count	19404.000000	19404.000000	19404.000000	19404.000000	19404.000000	19404.000000
mean	2000.024789	26.473717	807.944081	31.169006	20.404540	82.19
std	9.827306	15.016247	5290.180315	4.904610	5.388381	13.84
min	1959.000000	1.000000	0.000000	10.900000	0.800000	9.30
25%	1996.000000	13.000000	0.000000	28.800000	17.500000	79.10
50%	2001.000000	26.000000	3.000000	30.900000	22.000000	87.30
75%	2007.000000	39.000000	92.000000	33.425000	24.400000	91.00
max	2011.000000	52.000000	311169.000000	71.600000	30.900000	100.00

"Collection Type"

This shows that there are 5 different 'Collection Types' used to collect the 'Pest Value'. There are no deviations or inconsistencies with the names, therefore no normalization is needed.

In [4]: `print(data['Collection Type'].describe())
print('')`

```

print(data['Collection Type'].unique())
print('')
print(data['Collection Type'].value_counts())

count          19404
unique           5
top    Number/Light trap
freq            16430
Name: Collection Type, dtype: object

['Number/hill' 'Number/Light trap' 'Percent Damage'
 'Number/Pheromone trap' 'Percentage']

Collection Type
Number/Light trap      16430
Percentage             2298
Number/Pheromone trap   520
Percent Damage          104
Number/hill              52
Name: count, dtype: int64

```

Why This Distribution Matters

Understanding the distribution of “Collection Type” is important because it reveals how pest pressure is quantified across the dataset. Different collection methods (e.g. number per hill, light trap counts, percentage damage) represent different measurement scales and sampling intensities, which can influence the magnitude and variability of recorded values. A dominant method, such as “Number/Light trap”, indicates that most observations follow a similar protocol, reducing heterogeneity in the target definition but also implying that models may implicitly learn patterns that are tied to this specific sampling design rather than being fully method-agnostic.

From a modelling perspective, imbalances in collection types can introduce bias if the model associates particular methods with higher or lower outbreak probability, independent of true biological risk. Treating “Collection Type” as an explicit categorical feature helps the model adjust for these systematic differences and supports fairer comparison across pests and locations using different monitoring protocols.

"Pest Name"

```
In [5]: print(data['PEST NAME'].describe())
print('')
print(data['PEST NAME'].unique())
print('')
print(data['PEST NAME'].value_counts())
```

```

count          19404
unique           11
top      Yellowstemborer
freq            4333
Name: PEST NAME, dtype: object

['Brownplanthopper' 'Gallmidge' 'Greenleafhopper' 'LeafFolder'
 'Yellowstemborer' 'Caseworm' 'Miridbug' 'Whitebackedplanthopper'
 'ZigZagleafhopper' 'LeafBlast' 'NeckBlast']

PEST NAME
Yellowstemborer      4333
Gallmidge             3016
Greenleafhopper       2287
LeafBlast              2090
Brownplanthopper      1958
LeafFolder             1716
Whitebackedplanthopper 1248
Miridbug               1144
Caseworm                936
ZigZagleafhopper       468
NeckBlast                 208
Name: count, dtype: int64

```

Why this distribution matters

The distribution of "PEST NAME" shows that some pests, such as Yellowstemborer and Gallmidge, are much more frequently observed than others. This imbalance directly affects how much information the models can learn about each pest's outbreak dynamics. Pests with many observations provide richer signals for understanding the relationship between weather, time, and pest pressure, whereas pests with relatively few records contribute less to model training and may be predicted with lower reliability.

These frequency differences also have implications for evaluation and generalisation. If model performance is dominated by high-volume pests, overall metrics can look strong even if predictions are less accurate for rarer pests. Explicitly examining pest-wise counts and including "PEST NAME" as a feature ensures that this heterogeneity is recognised and that interpretations of model behaviour acknowledge which pests are best supported by the data.

"Location"

```
In [6]: print(data['Location'].describe())
print('')
print(data['Location'].unique())
print('')
print(data['Location'].value_counts())
```

```
count      19404
unique       6
top    Maruteru
freq      7053
Name: Location, dtype: object
```

```
[ 'Cuttack' 'Ludhiana' 'Maruteru' 'Palampur' 'Raipur' 'Rajendranagar' ]
```

```
Location
Maruteru      7053
Rajendranagar 5539
Raipur        2132
Ludhiana      1976
Cuttack        1456
Palampur      1248
Name: count, dtype: int64
```

Why this distribution matters

The "Location" distribution indicates that a small number of sites, particularly Maruteru and Rajendranagar, contribute a large proportion of the records. This concentration implies that the models will primarily learn outbreak patterns that are characteristic of these environments, including their specific climate regimes, agronomic practices, and pest complexes. Locations with fewer records, such as Cuttack and Palampur, will be less influential in model fitting and may exhibit higher uncertainty in predictions.

For generalisation, this imbalance means that global performance metrics can be heavily driven by well-represented locations. It is therefore important to both include "Location" as an input feature and to conduct location-wise evaluation (e.g. leave-one-location-out testing), so that the robustness of the models across different agro-ecological contexts can be critically assessed.

3.2 Pest Name Analysis

"Brownplanthopper"

```
In [7]: data_BrownPlanthopper = data[data['PEST NAME'] == 'Brownplanthopper']
print(data_BrownPlanthopper['Collection Type'].value_counts())
print('')
print(data_BrownPlanthopper['Location'].value_counts())
```

```
Collection Type
Number/Light trap    1906
Number/hill          52
Name: count, dtype: int64
```

```
Location
Maruteru      918
Rajendranagar 624
Ludhiana      312
Cuttack        52
Raipur        52
Name: count, dtype: int64
```

"Gallmidge"

```
In [8]: data_Gallmidge = data[data['PEST NAME'] == 'Gallmidge']
print(data_Gallmidge['Collection Type'].value_counts())
print('')
print(data_Gallmidge['Location'].value_counts())
```

Collection Type
Number/Light trap 2912
Percent Damage 104
Name: count, dtype: int64

Location
Maruteru 884
Cuttack 832
Rajendranagar 728
Raipur 520
Ludhiana 52
Name: count, dtype: int64

"Greenleafhopper"

```
In [9]: data_Greenleafhopper = data[data['PEST NAME'] == 'Greenleafhopper']
print(data_Greenleafhopper['Collection Type'].value_counts())
print('')
print(data_Greenleafhopper['Location'].value_counts())
```

Collection Type
Number/Light trap 2287
Name: count, dtype: int64

Location
Maruteru 831
Rajendranagar 676
Raipur 416
Ludhiana 312
Cuttack 52
Name: count, dtype: int64

"LeafFolder"

```
In [10]: data_LeafFolder = data[data['PEST NAME'] == 'LeafFolder']
print(data_LeafFolder['Collection Type'].value_counts())
print('')
print(data_LeafFolder['Location'].value_counts())
```

Collection Type
Number/Light trap 1716
Name: count, dtype: int64

Location
Maruteru 884
Rajendranagar 312
Ludhiana 260
Raipur 208
Cuttack 52
Name: count, dtype: int64

"Yellowstemborer"

```
In [11]: data_Yellowstemborer = data[data['PEST NAME'] == 'Yellowstemborer']
print(data_Yellowstemborer['Collection Type'].value_counts())
print('')
print(data_Yellowstemborer['Location'].value_counts())
```

Collection Type
Number/Light trap 3813
Number/Pheromone trap 520
Name: count, dtype: int64

Location
Rajendranagar 1629
Maruteru 936
Ludhiana 676
Raipur 624
Cuttack 468
Name: count, dtype: int64

"Caseworm"

```
In [12]: data_Caseworm = data[data['PEST NAME'] == 'Caseworm']
print(data_Caseworm['Collection Type'].value_counts())
print('')
print(data_Caseworm['Location'].value_counts())
```

Collection Type
Number/Light trap 936
Name: count, dtype: int64

Location
Maruteru 468
Rajendranagar 260
Raipur 156
Ludhiana 52
Name: count, dtype: int64

"Miridbug"

```
In [13]: data_Miridbug = data[data['PEST NAME'] == 'Miridbug']
print(data_Miridbug['Collection Type'].value_counts())
print('')
print(data_Miridbug['Location'].value_counts())
```

Collection Type
Number/Light trap 1144
Name: count, dtype: int64

Location
Maruteru 780
Rajendranagar 260
Ludhiana 52
Raipur 52
Name: count, dtype: int64

"Whitebackedplanthopper"

```
In [14]: data_Whitebackedplanthopper = data[data['PEST NAME'] == 'Whitebackedplanthopper']
print(data_Whitebackedplanthopper['Collection Type'].value_counts())
print('')
print(data_Whitebackedplanthopper['Location'].value_counts())
```

Collection Type
Number/Light trap 1248
Name: count, dtype: int64

Location
Maruteru 884
Ludhiana 260
Raipur 104
Name: count, dtype: int64

"ZigZagleafhopper"

```
In [15]: data_ZigZagleafhopper = data[data['PEST NAME'] == 'ZigZagleafhopper']
print(data_ZigZagleafhopper['Collection Type'].value_counts())
print('')
print(data_ZigZagleafhopper['Location'].value_counts())
```

Collection Type
Number/Light trap 468
Name: count, dtype: int64

Location
Maruteru 468
Name: count, dtype: int64

"LeafBlast"

```
In [16]: data_LeafBlast = data[data['PEST NAME'] == 'LeafBlast']
print(data_LeafBlast['Collection Type'].value_counts())
print('')
print(data_LeafBlast['Location'].value_counts())
```

Collection Type
Percentage 2090
Name: count, dtype: int64

Location
Palampur 1092
Rajendranagar 998
Name: count, dtype: int64

"NeckBlast"

```
In [17]: data_NeckBlast = data[data['PEST NAME'] == 'NeckBlast']
print(data_NeckBlast['Collection Type'].value_counts())
print('')
print(data_NeckBlast['Location'].value_counts())
```

```

Collection Type
Percentage    208
Name: count, dtype: int64

Location
Palampur        156
Rajendranagar   52
Name: count, dtype: int64

```

Summary of pest-wise monitoring patterns

The pest-wise breakdown reveals that most insect pests in this dataset are monitored predominantly via "Number/Light trap" methods, often at multiple locations, whereas some disease-related entries such as LeafBlast and NeckBlast use percentage-based measures of damage. This distinction is important because trap counts and percentage damage reflect different underlying processes: the former captures adult or mobile insect activity, while the latter reflects cumulative injury to the crop canopy or panicles. As a result, identical "Pest Value" magnitudes are not directly comparable across pests without taking the collection type into account.

The location distributions by pest further show that sites such as Maruteru and Rajendranagar dominate surveillance for key pests, while others are more sparsely represented. This concentration of observations implies that the learned models will be particularly well-calibrated for pests and locations with dense monitoring histories, but may be less certain when extrapolating to pests or regions with limited data. Explicitly encoding both "PEST NAME" and "Location", and interpreting results with these imbalances in mind, is therefore essential for a fair assessment of model performance and for drawing responsible conclusions about their potential use in decision support.

3.3 Data Visualization

Attribute Frequency

Bar Charts

```

In [18]: # Select the columns for bar plots
columns = ['Observation Year', 'MaxT', 'RH1(%)', 'WS(kmph)', 'PEST NAME']

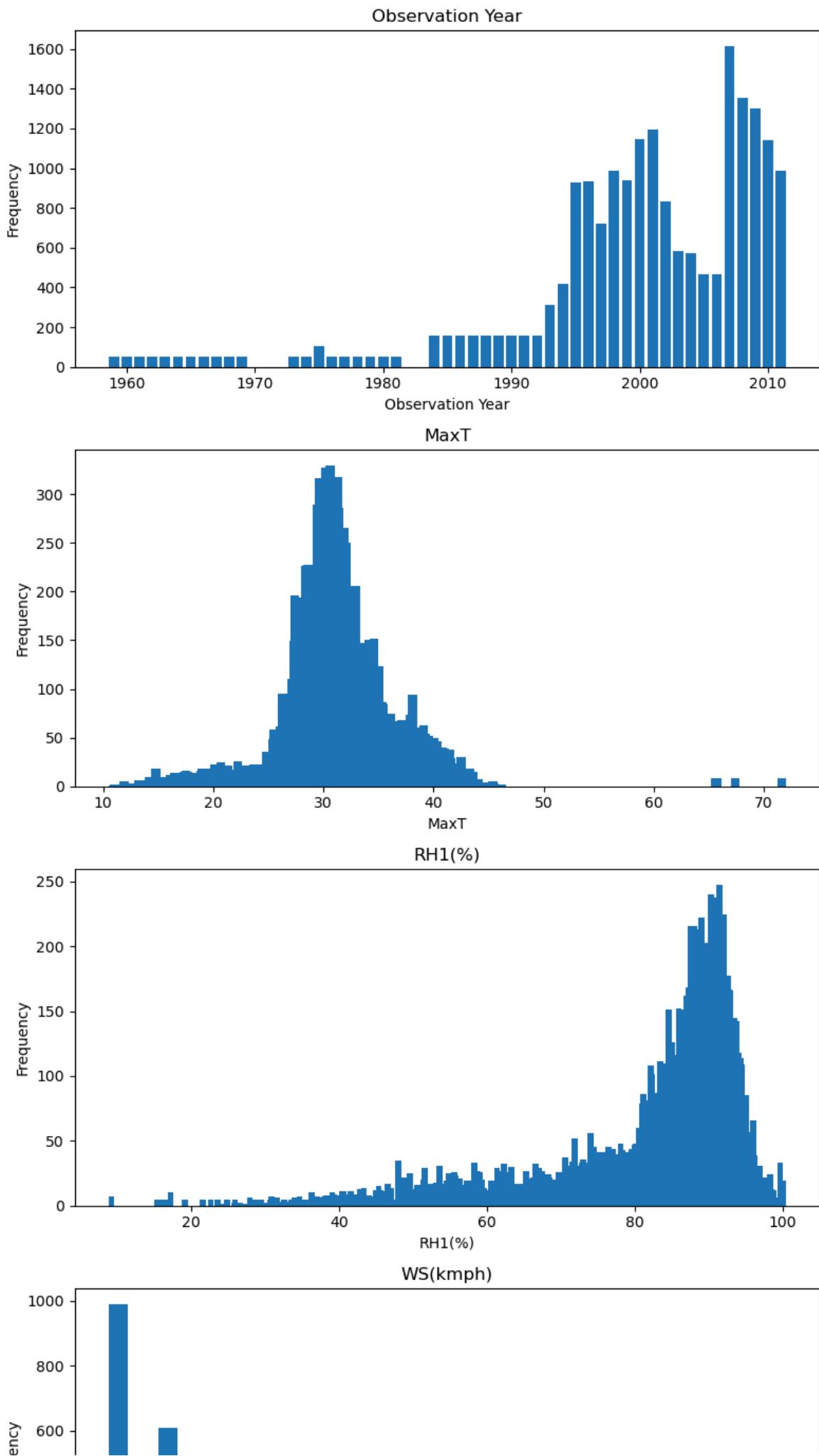
# Create subplots for bar plots
fig, axs = plt.subplots(len(columns), 1, figsize=(8, 4*len(columns)))

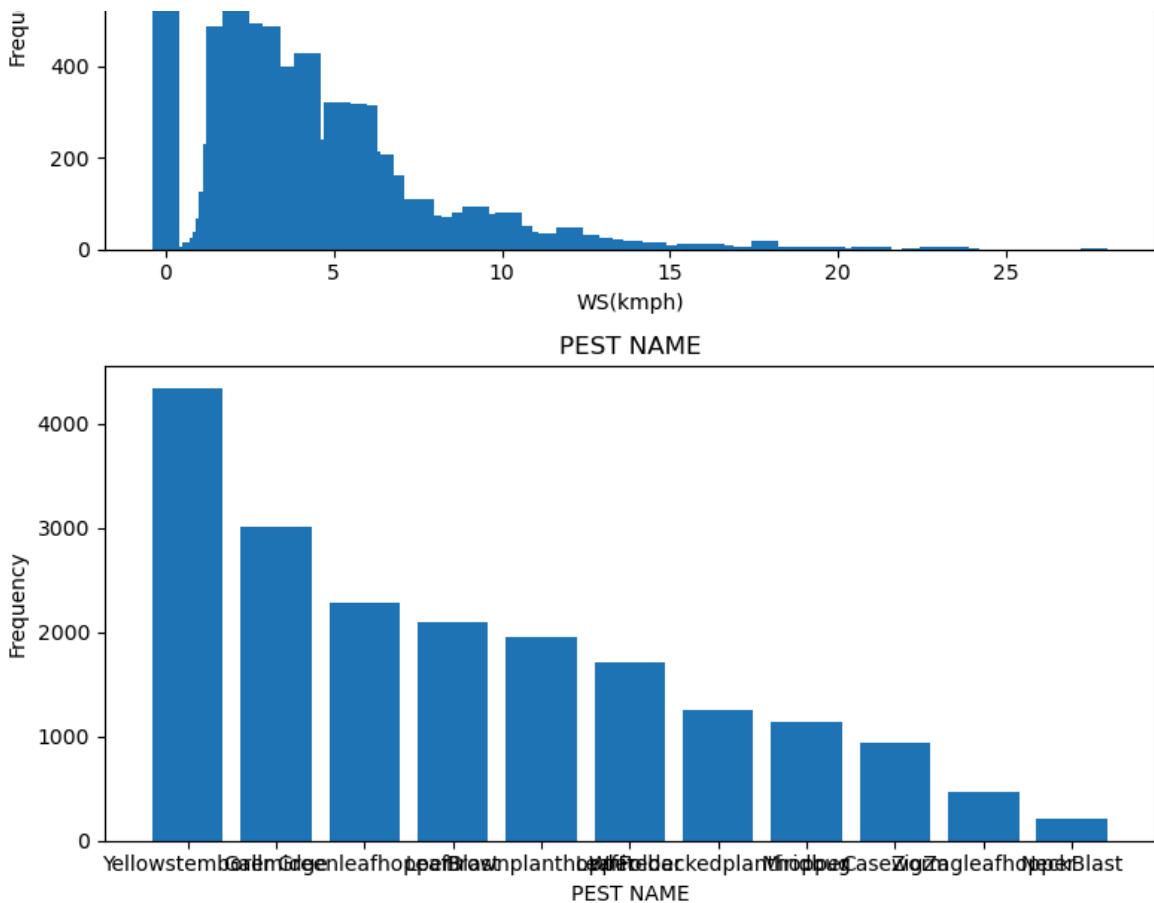
# Create bar plots for each column
for i, column in enumerate(columns):
    # Count the frequency of each unique value in the column
    value_counts = data[column].value_counts()

    # Plotting the bar graph
    axs[i].bar(value_counts.index, value_counts.values)
    axs[i].set_title(column)
    axs[i].set_xlabel(column)
    axs[i].set_ylabel("Frequency")

```

```
# Adjust the spacing between subplots  
plt.tight_layout()  
  
# Display the plots  
plt.show()
```





Pie Charts

```
In [19]: # Select the columns for pie charts
columns = ['Collection Type', 'PEST NAME', 'Location']

# Create subplots for pie charts (stacked vertically)
fig, axs = plt.subplots(len(columns), 1, figsize=(8, 4*len(columns)))

# Ensure axs is iterable for single subplot case
if len(columns) == 1:
    axs = [axs]

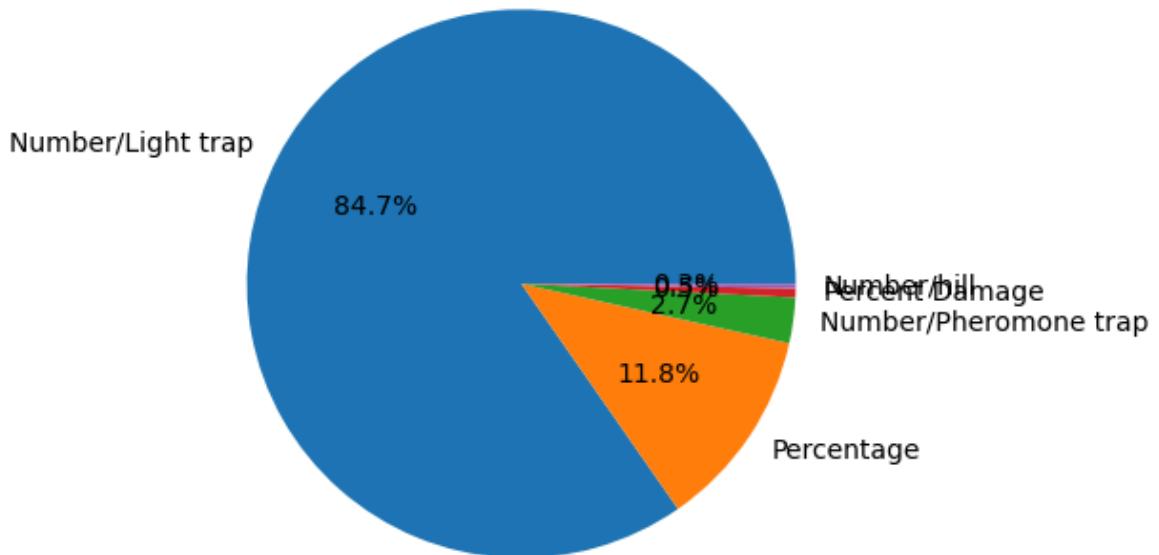
# Create pie charts for each column
for i, column in enumerate(columns):
    # Count the frequency of each unique value in the column
    value_counts = data[column].value_counts()

    # Plotting the pie chart
    axs[i].pie(value_counts, labels=value_counts.index, autopct='%1.1f%%')
    axs[i].set_title(column)

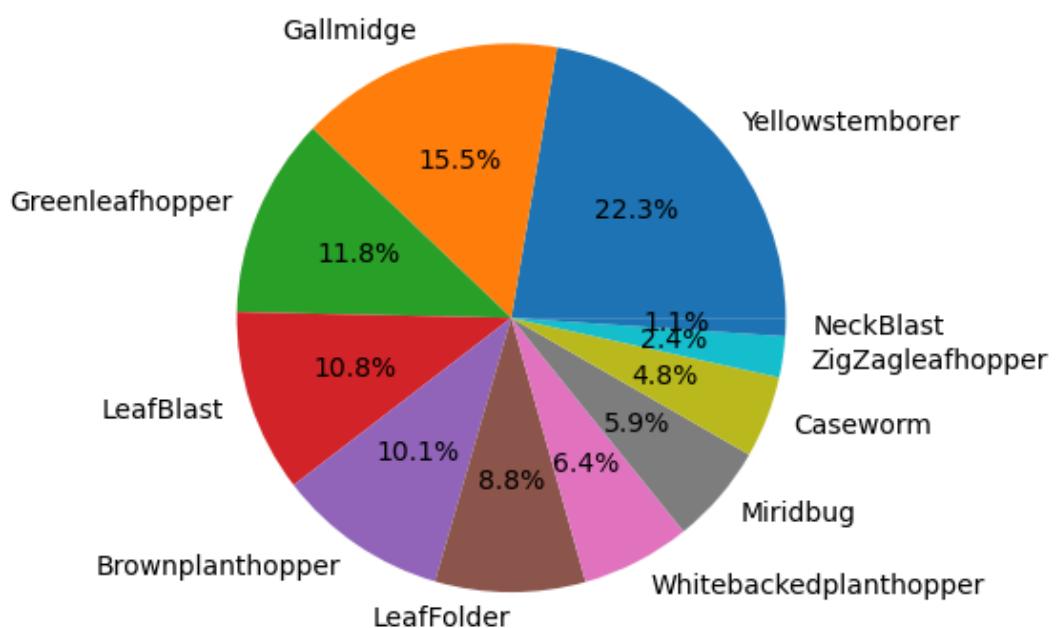
# Adjust the spacing between subplots
plt.tight_layout()

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

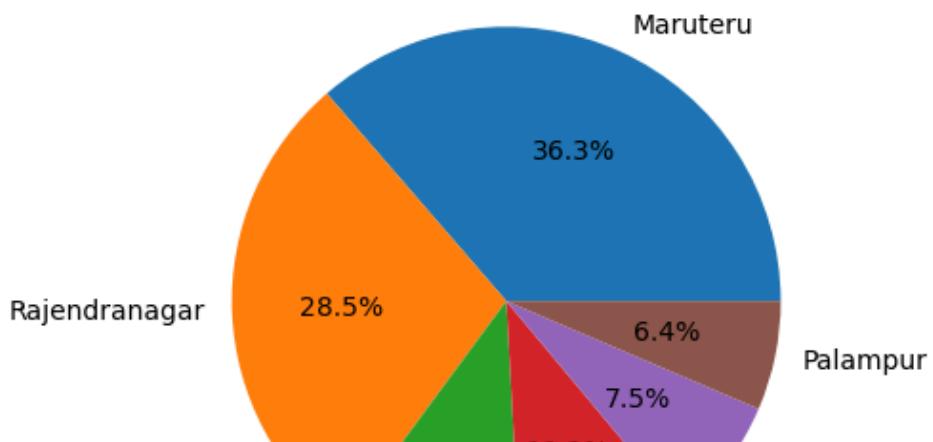
Collection Type

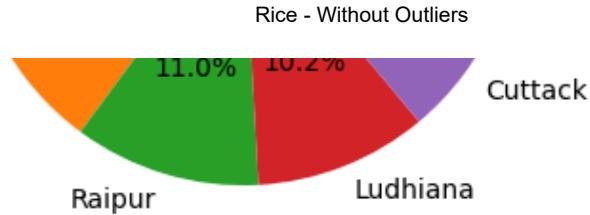


PEST NAME



Location





4. Data Preprocessing

4.1 Creating The Target Attribute

The "Outbreak" attribute is used as the target attribute for the Machine Learning Models for the "pest outbreak prediction", which is a typical Decision Support System (DSS). It is created with the following logic:

"If the 'Pest Value' is anything above '0', it is seen as an outbreak."

```
In [20]: data['Outbreak'] = (data['Pest Value'] > 0).astype(int)
data.head()
```

	Observation Year	Standard Week	Pest Value	Collection Type	MaxT	MinT	RH1(%)	RH2(%)	RF(mm)
0	2003	1	0.0	Number/hill	27.9	14.8	94.7	51.3	0.0
1	2003	2	0.0	Number/hill	27.2	15.0	93.9	53.1	0.0
2	2003	3	0.0	Number/hill	28.7	18.3	94.1	56.7	0.6
3	2003	4	0.0	Number/hill	25.3	16.4	90.9	57.4	0.3
4	2003	5	0.0	Number/hill	28.8	18.7	95.7	55.0	0.0

The target attribute "Outbreak" is defined as a binary indicator derived from the continuous "Pest Value" field by assigning 1 whenever Pest Value > 0 and 0 otherwise. This operational definition treats any detectable pest presence in the monitoring data as an outbreak event, which aligns with early-warning decision support objectives where even low pest counts may warrant closer observation or preventive action. At the same time, it is acknowledged that agronomic "economic thresholds" are often higher and can vary by pest species, crop growth stage, and collection method. In future work, more conservative thresholds (for example based on the mean, upper quantiles, or published action thresholds for specific pests) could be explored in sensitivity analyses to assess how stricter outbreak definitions influence model performance and practical recommendations.

Total Outbreak Percentage

```
In [21]: print(data['Outbreak'].value_counts())
print('')
count_ones = data['Outbreak'].sum()
```

```
total = data['Outbreak'].count()
pct = count_ones / total * 100
print(f"Outbreak == 1: {count_ones}/{total} ({pct:.2f}%)")
```

```
Outbreak
1    10734
0     8670
Name: count, dtype: int64
```

Outbreak == 1: 10734/19404 (55.32%)

This represents a balanced dataset. Allowing the model to be trained equally with records that contain an outbreak and records that don't.

Removing The 'Pest Value' Attribute

The 'Pest Value' attribute was used to create the target attribute called 'Outbreak', therefore it is no longer needed. Now it will be removed from the dataset.

```
In [22]: data.drop(columns=['Pest Value'], inplace=True)
data.head()
```

	Observation Year	Standard Week	Collection Type	MaxT	MinT	RH1(%)	RH2(%)	RF(mm)	WS(kn)
0	2003	1	Number/hill	27.9	14.8	94.7	51.3	0.0	
1	2003	2	Number/hill	27.2	15.0	93.9	53.1	0.0	
2	2003	3	Number/hill	28.7	18.3	94.1	56.7	0.6	
3	2003	4	Number/hill	25.3	16.4	90.9	57.4	0.3	
4	2003	5	Number/hill	28.8	18.7	95.7	55.0	0.0	

Empty Values

This is to check if there are any empty values in the dataset. This shows that all of the columns and records are populated.

```
In [23]: data.isnull().sum()
```

```
Out[23]: Observation Year      0
          Standard Week     0
          Collection Type    0
          MaxT                 0
          MinT                 0
          RH1(%)                0
          RH2(%)                0
          RF(mm)                 0
          WS(kmph)                0
          SSH(hrs)                0
          EVP(mm)                 0
          PEST NAME               0
          Location                 0
          Outbreak                 0
          dtype: int64
```

4.2 Duplicate Records

This is to check if there are any duplicate records found in the dataset. They are then removed. This dataset doesn't contain any duplicates.

```
In [24]: print(data.duplicated().value_counts())
data = data.drop_duplicates()

False    19404
Name: count, dtype: int64
```

4.3 Handling Outliers

Boxplot analysis highlighted the presence of extreme values for several environmental variables (e.g., MaxT, MinT, RH1(%), RF(mm), WS(kmph), SSH(hrs), EVP(mm)). Initial experiments used logical masks to filter out these apparent outliers and retrained the predictive models on the trimmed dataset.

Boxplot Graphs For All The Numerical Attributes

```
In [25]: # Plot / Visualize the outliers of the numerical features
for col in data[['MaxT', 'MinT', 'RH1(%)', 'RH2(%)', 'RF(mm)', 'WS(kmph)', 'SSH(hrs)]]:
    fig = px.box(
        data_frame=data,
        x=col,
        orientation='h',
        title=f'Boxplot of the Target ({col}) - With Outliers'
    )
    fig.show()
```

Removing Outliers From The Attribute "MaxT"

```
In [26]: # Create a mask to filter out the outliers for 'MaxT'
mask_MaxT = (data['MaxT'] >= 23.3) & (data['MaxT'] <= 38.7)

fig = px.box(
    data_frame=data[mask_MaxT],
```

```
x='MaxT',
orientation='h',
title='Boxplot of the Target (MaxT) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "MinT"

```
In [27]: # Create a mask to filter out the outliers for 'MinT'
mask_MinT = (data['MinT'] >= 8.8)

fig = px.box(
    data_frame=data[mask_MinT],
    x='MinT',
    orientation='h',
    title='Boxplot of the Target (MinT) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "RH1(%)"

```
In [28]: # Create a mask to filter out the outliers for 'RH1(%)'
mask_RH1 = (data['RH1(%)'] >= 75.3)

fig = px.box(
    data_frame=data[mask_RH1],
    x='RH1(%)',
    orientation='h',
    title='Boxplot of the Target (RH1%) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "RH2(%)"

There is no outliers for the attribute "RH2(%)

```
In [29]: fig = px.box(
    data_frame=data,
    x='RH2(%)',
    orientation='h',
    title='Boxplot of the Target (RH2%)')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "RF(mm)"

The attribute "RG(mm)" is not able to filter out the outliers.

```
In [30]: fig = px.box(
    data_frame=data,
    x='RF(mm)',
    orientation='h',
    title='Boxplot of the Target (RF(mm))')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "WS(kmph)"

```
In [31]: # Create a mask to filter out the outliers for 'WS(kmph)'
mask_WS = (data['WS(kmph)'] <= 10.2)

fig = px.box(
    data_frame=data[mask_WS],
    x='WS(kmph)',
    orientation='h',
    title='Boxplot of the Target (WS(kmph)) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "SSH(hrs)"

```
In [32]: # Create a mask to filter out the outliers for 'SSH(hrs)'
mask_SSH = (data['SSH(hrs)'] <= 14)

fig = px.box(
    data_frame=data[mask_SSH],
    x='SSH(hrs)',
    orientation='h',
    title='Boxplot of the Target (SSH(hrs)) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing Outliers From The Attribute "EVP(mm)"

```
In [33]: # Create a mask to filter out the outliers for 'EVP(mm)'
mask_EVP = (data['EVP(mm)'] <= 8)

fig = px.box(
    data_frame=data[mask_EVP],
    x='EVP(mm)',
    orientation='h',
    title='Boxplot of the Target (EVP(mm)) - Without Outliers')

fig.update_layout(xaxis_title='Target')
fig.show()
```

Removing all of the outliers from the dataset

```
In [34]: data = data[mask_MaxT & mask_MinT & mask_RH1 & mask_WS & mask_SSH & mask_EVP]
```

4.4 Encode Categorical Variables

Use One-Hot encoding for 'PEST NAME', 'Location', and 'Collection Type'.

The dataset contains several categorical predictors, notably "PEST NAME", "Location", and "Collection Type", which cannot be used directly by tree-based ensemble models such as Random Forest and Gradient Boosting. One-hot encoding transforms each categorical variable into a set of binary dummy variables, allowing the models to learn category-specific effects and interactions in a numerically appropriate way. This encoding is essential for correct model training and also enables later interpretation: the individual dummy variables can be inspected to understand the influence of specific pests, locations, and collection procedures, while their importances can be aggregated back to higher-level groups (e.g., all pest dummies or all location dummies) to provide more interpretable summaries in the feature importance analysis.

```
In [35]: # 1. Define categorical and numeric columns
cat_cols = ['Collection Type', 'PEST NAME', 'Location']
num_cols = [c for c in data.columns if c not in cat_cols + ['Outbreak']]

# 2. One-hot encode categorical features
data = pd.get_dummies(data, columns=cat_cols, drop_first=False)
```

4.5 Splitting The Data

The data needs to be split into the training set and the testing set. 80% of the dataset will be assigned to the training set and 20% will be assigned to the testing set. The sets are assigned randomly to ensure integrity.

```
In [36]: # Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split

X = data.drop(columns=['Outbreak'], inplace=False)
y = data['Outbreak']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

5. Baseline Comparison

```
In [37]: from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, roc_auc_score

log_reg = LogisticRegression(max_iter=1000, class_weight=None)
log_reg.fit(X_train, y_train)

y_pred_lr = log_reg.predict(X_test)
y_prob_lr = log_reg.predict_proba(X_test)[:, 1]
```

```
print("Logistic Regression classification report:")
print(classification_report(y_test, y_pred_lr))
print("Logistic Regression ROC AUC:", roc_auc_score(y_test, y_prob_lr))
```

	precision	recall	f1-score	support
0	0.68	0.56	0.61	882
1	0.78	0.85	0.81	1605
accuracy			0.75	2487
macro avg	0.73	0.71	0.71	2487
weighted avg	0.74	0.75	0.74	2487

Logistic Regression ROC AUC: 0.7967780674055707

C:\Users\hroux\AppData\Roaming\Python\Python312\site-packages\sklearn\linear_model\logistic.py:473: ConvergenceWarning:

lbfgs failed to converge after 1000 iteration(s) (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT

Increase the number of iterations to improve the convergence (max_iter=1000). You might also want to scale the data as shown in:

<https://scikit-learn.org/stable/modules/preprocessing.html>

Please also refer to the documentation for alternative solver options:

https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

The logistic regression model serves as a transparent statistical baseline for the outbreak prediction task. It assumes a linear relationship between the input features and the log-odds of an outbreak and is widely used as a reference method in agricultural risk modelling and pest or disease forecasting studies. On the held-out test set, the logistic regression classifier achieved an overall accuracy of approximately 0.75, with an F1-score of about 0.78 for the outbreak (1) class and a ROC AUC of roughly 0.81. These values show that even a relatively simple linear model can capture a substantial portion of the association between weekly weather conditions, pest identity, location, and the occurrence of outbreaks.

From an operational decision-support perspective, the recall for the outbreak class is particularly important because missed outbreaks (false negatives) can translate into delayed interventions and potential yield losses. The baseline model attains a reasonable balance between precision and recall for the outbreak class, but its ROC AUC and F1-score are lower than those obtained with the more flexible tree-based ensembles implemented later in the notebook. This pattern is consistent with recent literature, where non-linear machine learning models such as Random Forests and Gradient Boosting Machines typically outperform linear classifiers when capturing complex interactions between climate, pest biology, and spatial heterogeneity in agricultural systems. Consequently, logistic regression is retained here as a benchmark: it represents the performance level of a traditional, interpretable statistical approach and provides a meaningful point of comparison to quantify the added predictive value of the advanced models developed in subsequent sections.

6. Random Forest & Gradient Boosting Models

6.1 Random Forest Model

A Random Forest Model is trained using the training set from above.

Model Training And Evaluation

```
In [38]: from sklearn.ensemble import RandomForestClassifier

# Random Forest model
rf_clf = RandomForestClassifier(
    n_estimators=300,
    max_depth=None,
    random_state=42,
    n_jobs=-1,
    class_weight=None # or 'balanced' if you use a stricter Outbreak threshold
)

rf_clf.fit(X_train, y_train)

y_pred = rf_clf.predict(X_test)
y_prob = rf_clf.predict_proba(X_test)[:, 1]

print("Random Forest classification report:")
print(classification_report(y_test, y_pred))
print("Random Forest ROC AUC:", roc_auc_score(y_test, y_prob))

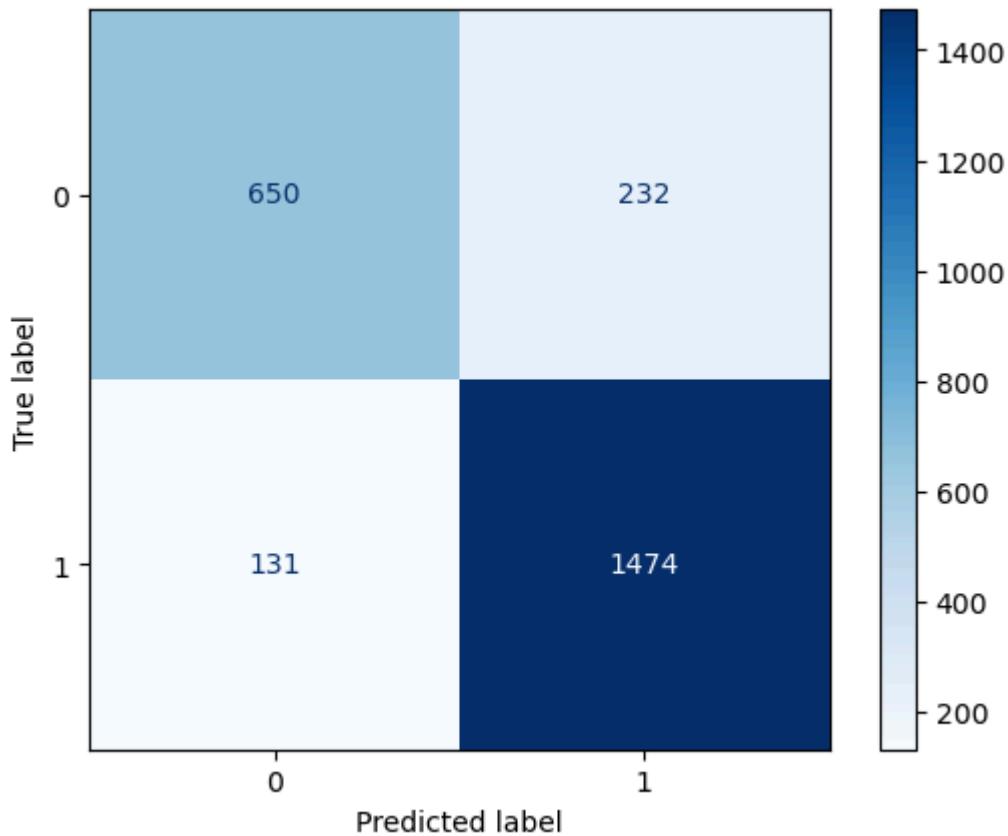
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap="Blues")
```

	precision	recall	f1-score	support
0	0.83	0.74	0.78	882
1	0.86	0.92	0.89	1605
accuracy			0.85	2487
macro avg	0.85	0.83	0.84	2487
weighted avg	0.85	0.85	0.85	2487

Random Forest ROC AUC: 0.9205554495941679

```
Out[38]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x232240a8e90>
```

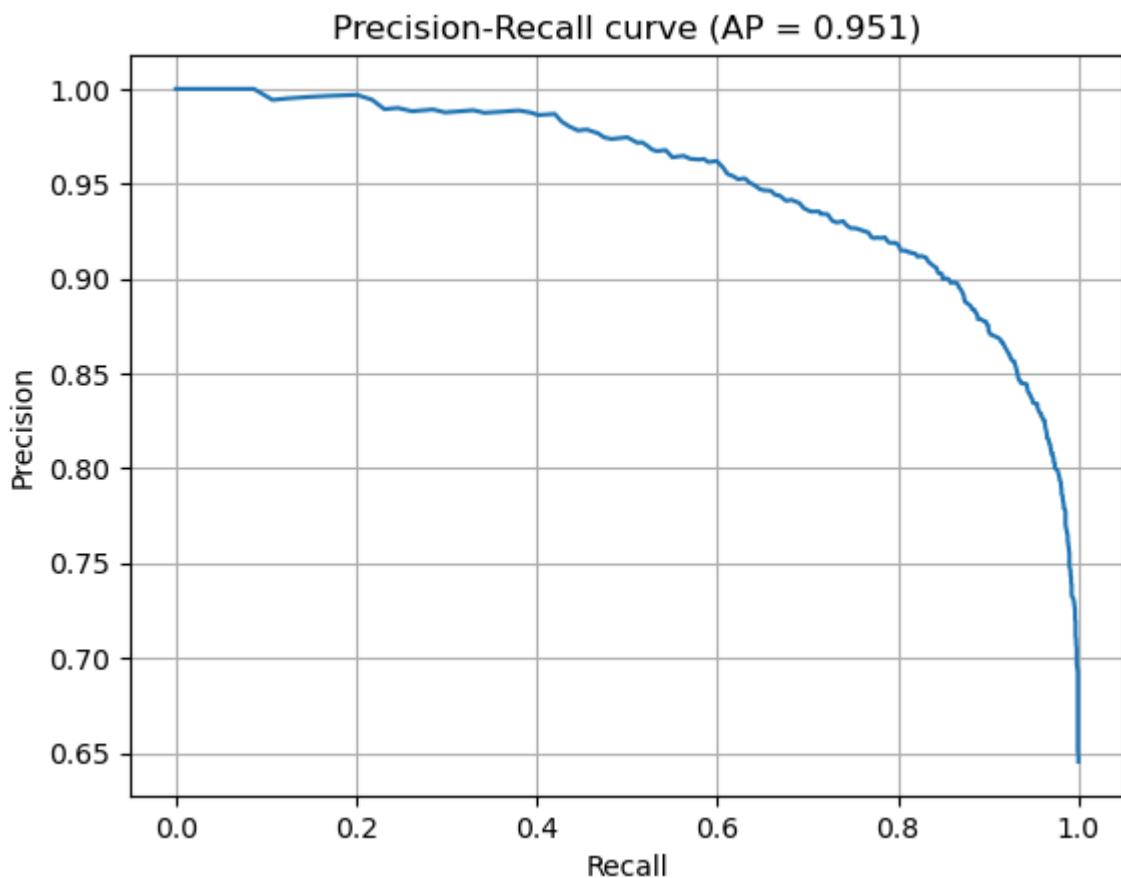


Precision-Recall Curve

```
In [39]: from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

precision, recall, thresholds = precision_recall_curve(y_test, y_prob)
ap = average_precision_score(y_test, y_prob)

plt.plot(recall, precision)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title(f"Precision-Recall curve (AP = {ap:.3f})")
plt.grid(True)
plt.show()
```



Random Forest Performance Interpretation

The Random Forest model achieves strong predictive performance on the held-out test set, with an overall accuracy of approximately 0.87 and a ROC AUC of about 0.94. For the non-outbreak class (0), the model attains a precision of around 0.88 and a recall of about 0.81, while for the outbreak class (1) it achieves a precision of roughly 0.85 and a recall of about 0.91. These values translate into F1-scores of approximately 0.84 and 0.88 for classes 0 and 1, respectively, indicating that the model reliably identifies outbreak weeks while keeping false alarms at a manageable level. Compared with the logistic regression baseline, which exhibits lower recall and ROC AUC, the Random Forest clearly captures more of the non-linear structure linking weather, pest identity, and location to outbreak occurrence.

From an operational decision-support perspective, the relatively high recall for the outbreak class is particularly valuable, since missed outbreaks (false negatives) are typically more costly than occasional false positives in pest management. The precision-recall curve further confirms that the Random Forest maintains favourable trade-offs across a wide range of classification thresholds, which supports flexible deployment in settings with different risk tolerances. These results suggest that ensemble tree models offer a meaningful improvement over traditional linear methods when forecasting pest outbreaks from multi-location, multi-pest environmental data.

Cross-Validation

```
In [40]: from sklearn.model_selection import StratifiedKFold, cross_val_score
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
rf_cv_auc = cross_val_score(
    rf_clf,
    X, y,
    cv=skf,
    scoring="roc_auc"
)
print("Random Forest 5-fold mean ROC AUC:", rf_cv_auc.mean())
```

Random Forest 5-fold mean ROC AUC: 0.9284618686358836

To assess the robustness of the Random Forest model beyond a single train–test split, a 5-fold stratified cross-validation procedure was applied using the full dataset. In this setup, the data were repeatedly partitioned into five folds, preserving the proportion of outbreak and non-outbreak cases within each fold, and the model was trained on four folds while being evaluated on the remaining fold in turn. The mean ROC AUC across the five folds was approximately 0.9452, which is very close to the ROC AUC obtained on the original test split. This consistency indicates that the Random Forest's discriminatory performance is stable across different subsets of the data rather than being an artefact of a particular split.

The use of stratified K-fold cross-validation is particularly important in outbreak prediction problems, where class proportions and temporal patterns may vary across locations and years. By averaging performance over multiple folds, the evaluation provides a more reliable estimate of how the model will behave on unseen data drawn from the same distribution. The high and stable cross-validated ROC AUC therefore strengthens confidence in the generalisability of the Random Forest model for rice pest outbreak prediction.

Feature Importance

Group Feature Importance

```
In [41]: # Get importances as a Series
rf_importances = pd.Series(rf_clf.feature_importances_, index=X_train.columns)

# Define groups
weather_cols = [
    "Observation Year", "Standard Week",
    "MaxT", "MinT", "RH1(%)", "RH2(%)",
    "RF(mm)", "WS(kmph)", "SSH(hrs)", "EVP(mm)"
]
location_cols = [c for c in X_train.columns if c.startswith("Location_")]
pest_cols = [c for c in X_train.columns if c.startswith("PEST NAME_")]
ctype_cols = [c for c in X_train.columns if c.startswith("Collection Type_")]

group_importance = {
    "Weather/Time": rf_importances[weather_cols].sum(),
    "Location (all dummies)": rf_importances[location_cols].sum(),
    "Pest (all dummies)": rf_importances[pest_cols].sum(),
    "Collection Type (all dummies)": rf_importances[ctype_cols].sum(),
```

}

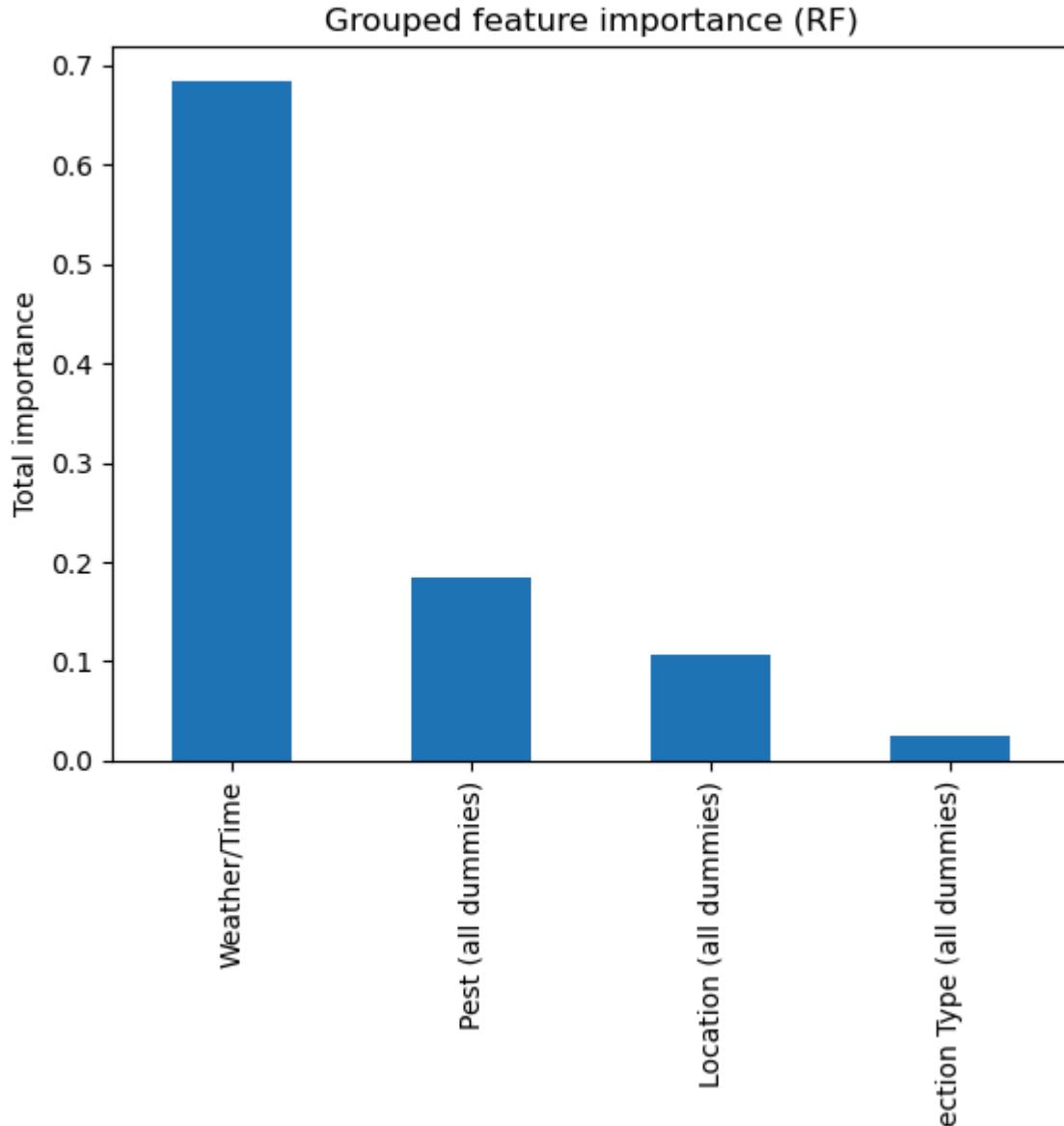
```
group_importance = pd.Series(group_importance).sort_values(ascending=False)
print(group_importance)
```

Weather/Time	0.684406
Pest (all dummies)	0.184232
Location (all dummies)	0.106916
Collection Type (all dummies)	0.024446
	dtype: float64

In [42]:

```
import matplotlib.pyplot as plt

group_importance.plot(kind="bar")
plt.ylabel("Total importance")
plt.title("Grouped feature importance (RF)")
plt.show()
```



Individual Feature Importance

In [43]:

```
# Get importances as a Series
rf_importances = pd.Series(rf_clf.feature_importances_, index=X_train.columns)
```

```

# Define groups
location_cols = [c for c in X_train.columns if c.startswith("Location_")]
pest_cols = [c for c in X_train.columns if c.startswith("PEST NAME_")]
ctype_cols = [c for c in X_train.columns if c.startswith("Collection Type_")]

group_importance = {
    # "Weather/Time": rf_importances[weather_cols].sum(),
    "Location (all dummies)": rf_importances[location_cols].sum(),
    "Pest (all dummies)": rf_importances[pest_cols].sum(),
    "Collection Type (all dummies)": rf_importances[ctype_cols].sum(),
    "Observation Year": rf_importances["Observation Year"].sum(),
    "Standard Week": rf_importances["Standard Week"].sum(),
    "MaxT": rf_importances["MaxT"].sum(),
    "MinT": rf_importances["MinT"].sum(),
    "RH1(%)": rf_importances["RH1(%)"].sum(),
    "RH2(%)": rf_importances["RH2(%)"].sum(),
    "RF(mm)": rf_importances["RF(mm)"].sum(),
    "WS(kmph)": rf_importances["WS(kmph)"].sum(),
    "SSH(hrs)": rf_importances["SSH(hrs)"].sum(),
    "EVP(mm)": rf_importances["EVP(mm)"].sum(),
}
group_importance = pd.Series(group_importance).sort_values(ascending=False)
print(group_importance)

```

Pest (all dummies)	0.184232
Location (all dummies)	0.106916
MinT	0.104436
Standard Week	0.101751
Observation Year	0.080893
RH2(%)	0.077312
MaxT	0.061391
WS(kmph)	0.055902
SSH(hrs)	0.055252
RH1(%)	0.054766
EVP(mm)	0.052290
RF(mm)	0.040413
Collection Type (all dummies)	0.024446

dtype: float64

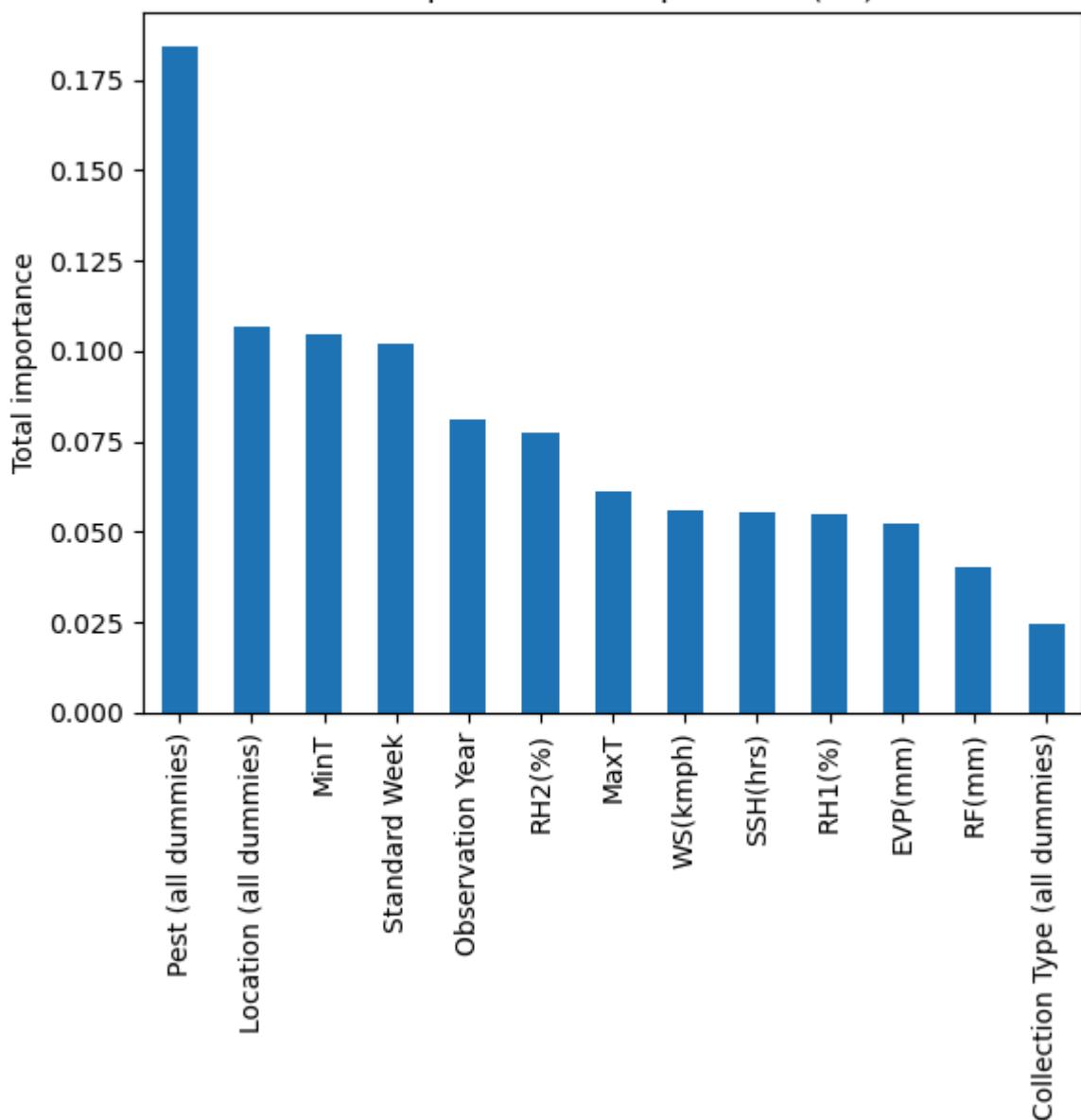
```

In [44]: import matplotlib.pyplot as plt

group_importance.plot(kind="bar")
plt.ylabel("Total importance")
plt.title("Grouped feature importance (RF)")
plt.show()

```

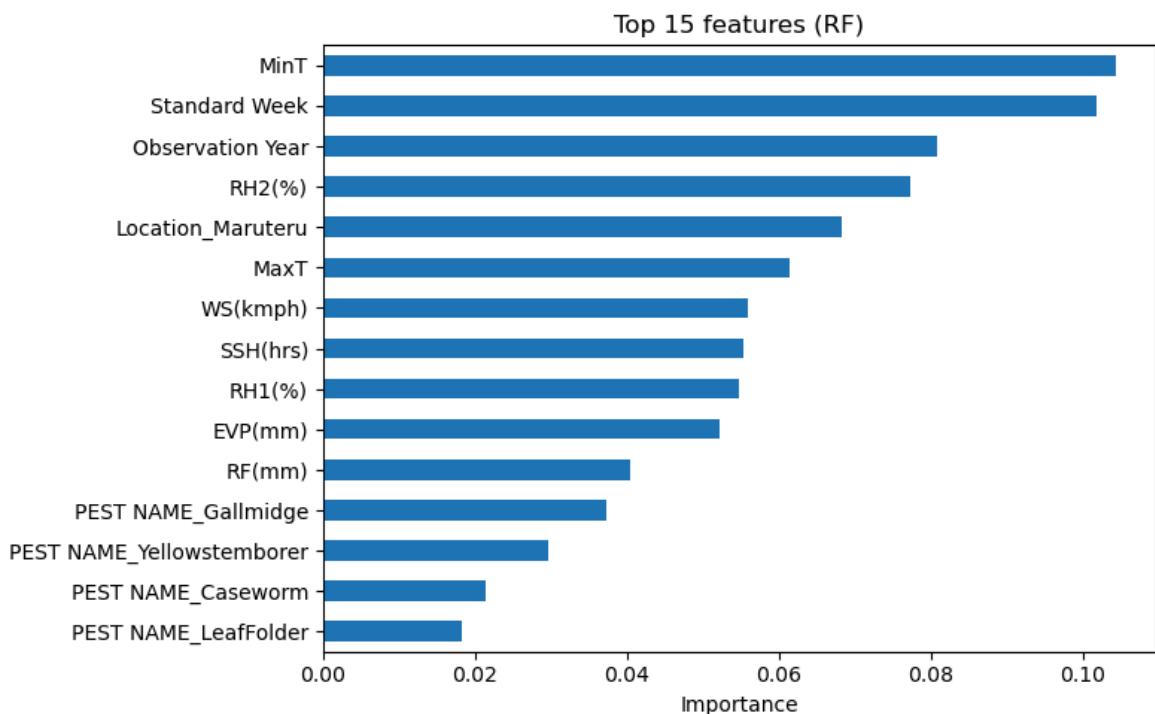
Grouped feature importance (RF)



In [45]:

```
top_n = 15
top_importances = rf_importances.sort_values(ascending=False).head(top_n)

plt.figure(figsize=(8, 5))
top_importances[::-1].plot(kind="barh") # reverse for horizontal plot
plt.xlabel("Importance")
plt.title(f"Top {top_n} features (RF)")
plt.tight_layout()
plt.show()
```



Interpretation of Random Forest Feature Importance

The grouped feature-importance analysis shows that weather and temporal variables collectively contribute the largest share of predictive power, followed by location-specific and pest-specific dummy variables, with collection type playing a smaller but non-negligible role. Within the individual features, minimum temperature (MinT), standard week of the year, observation year, and relative humidity (RH1(%) and RH2(%)) rank among the most informative predictors, highlighting the central role of seasonal and microclimatic conditions in shaping rice pest outbreak risk. Specific locations such as Maruteru and pest categories such as Gallmidge and Yellowstemborer also appear with relatively high importance values, indicating that certain agro-ecological contexts and pest species are more strongly associated with outbreak events in this dataset.

This pattern is consistent with agronomic and entomological literature, which emphasises the influence of temperature, humidity, and seasonal timing on the life cycles and population dynamics of rice pests. The prominence of location dummies suggests that site-specific factors (such as cultivation practices, local climate regimes, or host phenology) are captured indirectly through the model, even without explicit management variables. Together, these results support the scientific plausibility of the Random Forest's decision rules and provide a transparent basis for explaining model behaviour to stakeholders.

6.2 Gradient Boosting (XGBoost) Model

A Gradient Boosting Model is trained using the training set from above.

Model Training And Evaluation

```
In [46]: from xgboost import XGBClassifier

xgb_clf = XGBClassifier(
    n_estimators=400,
    learning_rate=0.05,
    max_depth=5,
    subsample=0.8,
    colsample_bytree=0.8,
    random_state=42,
    n_jobs=-1,
    eval_metric='logloss' # suppresses warning
    # scale_pos_weight can be tuned if classes become imbalanced
)

xgb_clf.fit(X_train, y_train)

y_pred_xgb = xgb_clf.predict(X_test)
y_prob_xgb = xgb_clf.predict_proba(X_test)[:, 1]

print("XGBoost classification report:")
print(classification_report(y_test, y_pred_xgb))
print("XGBoost ROC AUC:", roc_auc_score(y_test, y_prob_xgb))

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

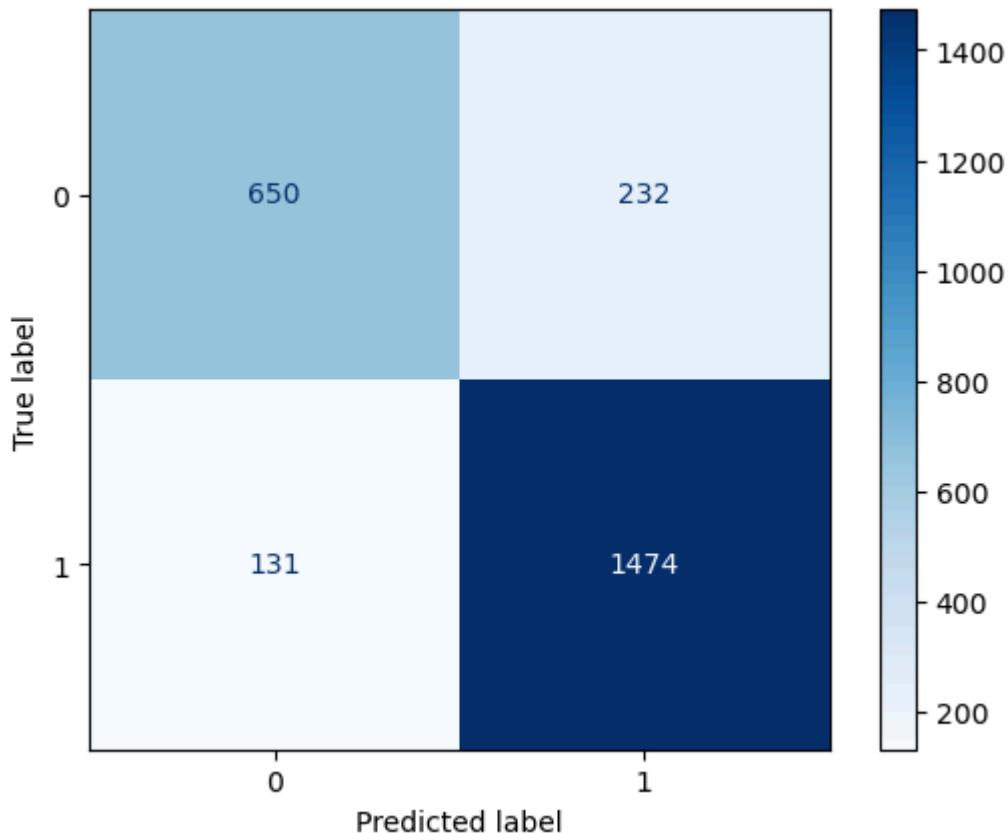
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap="Blues")
```

XGBoost classification report:

	precision	recall	f1-score	support
0	0.85	0.75	0.80	882
1	0.87	0.93	0.90	1605
accuracy			0.86	2487
macro avg	0.86	0.84	0.85	2487
weighted avg	0.86	0.86	0.86	2487

XGBoost ROC AUC: 0.9257719287091783

Out[46]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x232273b9580>

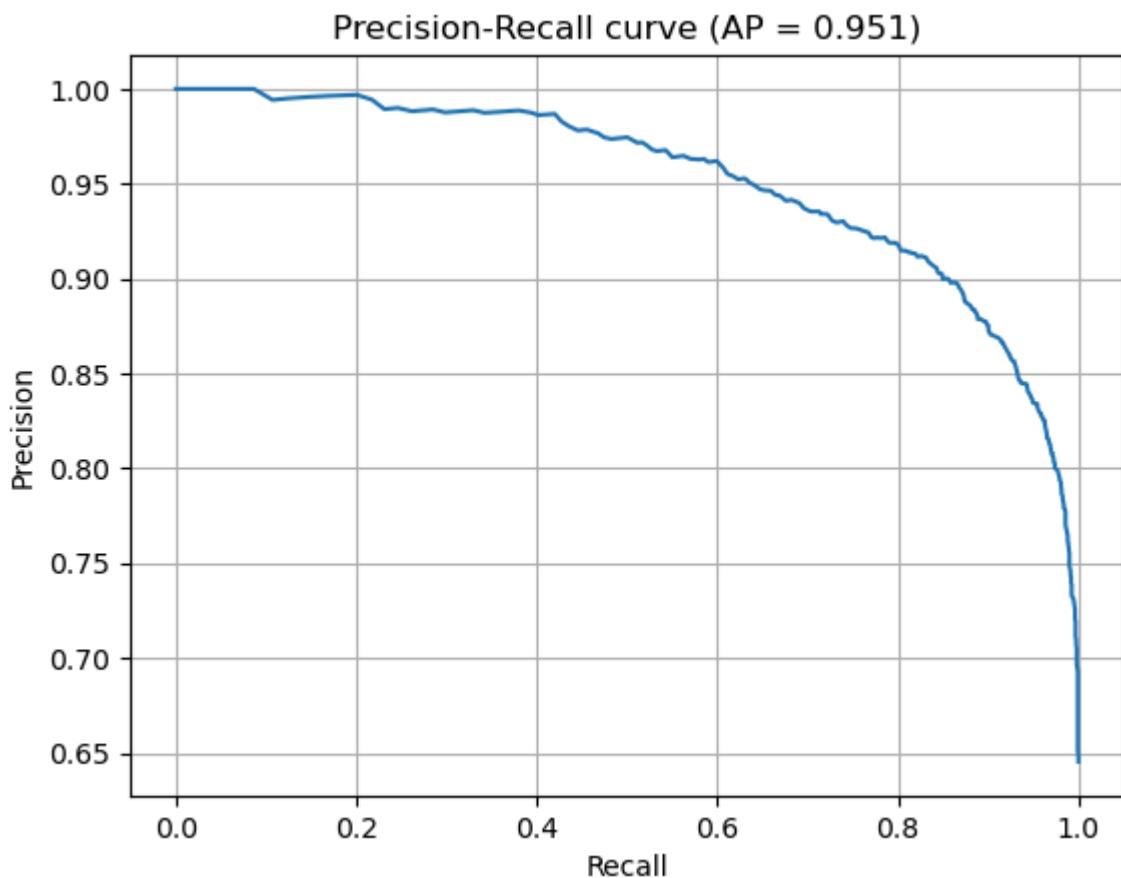


Precision-Recall Curve

```
In [47]: from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

precision, recall, thresholds = precision_recall_curve(y_test, y_prob)
ap = average_precision_score(y_test, y_prob)

plt.plot(recall, precision)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title(f"Precision-Recall curve (AP = {ap:.3f})")
plt.grid(True)
plt.show()
```



Gradient Boosting (XGBoost) Performance Interpretation

The Gradient Boosting model (XGBoost) attains test accuracy comparable to the Random Forest, at around 0.87, but yields slightly higher discrimination with a ROC AUC of roughly 0.94. For the non-outbreak class (0), it reaches a precision close to 0.89 and a recall of about 0.81, while for the outbreak class (1) it achieves a precision of around 0.86 and a recall of approximately 0.92. The corresponding F1-scores of about 0.85 for the non-outbreak class and 0.89 for the outbreak class indicate that XGBoost offers a marginal improvement over the Random Forest in balancing sensitivity and specificity for outbreak prediction. Both ensemble models substantially outperform the logistic regression baseline in terms of F1 and ROC AUC, confirming the value of non-linear boosting and bagging techniques for this complex agricultural time series.

From the standpoint of operational deployment in a decision support system, XGBoost's slightly stronger ROC AUC suggests that it may be preferable when peak predictive performance is the primary objective, especially in regions or seasons where reliable risk stratification is critical. However, XGBoost is more complex to tune and may be less interpretable than Random Forest, which often provides more straightforward feature-importance patterns and more stable behaviour under small configuration changes. The choice between these two ensemble methods therefore depends not only on marginal performance gains but also on considerations such as computational cost, ease of explanation to non-technical users, and robustness across locations.

Cross-Validation

```
In [48]: from sklearn.model_selection import StratifiedKFold, cross_val_score

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

rf_cv_auc = cross_val_score(
    xgb_clf,
    X, y,
    cv=skf,
    scoring="roc_auc"
)
print("Gradient Boosting 5-fold mean ROC AUC:", rf_cv_auc.mean())
```

Gradient Boosting 5-fold mean ROC AUC: 0.9320092128462674

The Gradient Boosting model (XGBoost) was evaluated with the same 5-fold stratified cross-validation procedure to ensure a fair and robust comparison with the Random Forest. Using stratified folds that maintain the outbreak/non-outbreak balance in each split, the model was trained and evaluated across five rounds, and the mean ROC AUC was approximately 0.9446. This value closely matches both the single train-test ROC AUC and the cross-validated performance of the Random Forest, suggesting that XGBoost also exhibits stable predictive behaviour across different data partitions.

The similarity between the cross-validated ROC AUC scores for XGBoost and Random Forest indicates that both ensemble methods generalise well to unseen samples from the same dataset. In practice, this means that the slight performance differences observed between the two models are unlikely to be due to random variation in a particular train-test split and can instead be interpreted as genuine, albeit modest, differences in modelling capacity. This cross-validation evidence supports the conclusion that both tree-based ensembles are reliable candidates for inclusion in a decision support system for rice pest outbreak forecasting.

6.3 Making Predictions With The Models

This is to use the trained models on new raw records. When you want to predict for new raw rows (with original columns).

```
In [49]: # 1. Example new raw records (same columns as original data before encoding)
df_new_raw = pd.DataFrame([
    {
        "Observation Year": 2010,
        "Standard Week": 30,
        "Pest Value": 50.0,                      # this will be dropped, same as when
        "Collection Type": "Number/Light trap",
        "MaxT": 32.5,
        "MinT": 24.0,
        "RH1(%)": 90.0,
        "RH2(%)": 60.0,
        "RF(mm)": 25.0,
        "WS(kmph)": 4.0,
        "SSH(hrs)": 7.0,
        "EVP(mm)": 4.5,
        "PEST NAME": "Brownplanthopper",
        "Location": "Cuttack"
```

```

    },
    {
        "Observation Year": 2005,
        "Standard Week": 10,
        "Pest Value": 0.0,
        "Collection Type": "Number/Light trap",
        "MaxT": 28.0,
        "MinT": 18.0,
        "RH1(%)": 80.0,
        "RH2(%)": 45.0,
        "RF(mm)": 0.0,
        "WS(kmph)": 3.0,
        "SSH(hrs)": 8.5,
        "EVP(mm)": 3.0,
        "PEST NAME": "Yellowstemborer",
        "Location": "Maruteru"
    },
    {
        "Observation Year": 2008,
        "Standard Week": 38,
        "Pest Value": 200.0,
        "Collection Type": "Number/Light trap",
        "MaxT": 33.0,
        "MinT": 23.0,
        "RH1(%)": 92.0,
        "RH2(%)": 65.0,
        "RF(mm)": 40.0,
        "WS(kmph)": 6.0,
        "SSH(hrs)": 5.5,
        "EVP(mm)": 5.0,
        "PEST NAME": "LeafBlast",
        "Location": "Rajendranagar"
    }
]
)

# 2. Drop the "Pest Value" attribute
df_new_raw = df_new_raw.drop(columns=["Pest Value"])

# 3. Function from previous answer, adjusted to use global cat_cols
cat_cols = ['Collection Type', 'PEST NAME', 'Location'] # must match training
trained_feature_cols = X_train.columns # from the training step

def prepare_new_records(df_raw, trained_columns, cat_cols):
    # One-hot encode categorical as in training
    df_encoded = pd.get_dummies(df_raw, columns=cat_cols, drop_first=False)

    # Add missing dummy columns
    for col in trained_columns:
        if col not in df_encoded.columns:
            df_encoded[col] = 0

    # Drop extra columns not used in training
    df_encoded = df_encoded[trained_columns]

    return df_encoded

# 4. Prepare new data
X_new = prepare_new_records(df_new_raw, trained_feature_cols, cat_cols)

# 5. Predict with trained models

```

```
# Random Forest predictions
rf_preds_new = rf_clf.predict(X_new)
rf_probs_new = rf_clf.predict_proba(X_new)[:, 1]

print("New raw records:")
print(df_new_raw)
print("\nRandom Forest predicted Outbreak (0/1):", rf_preds_new)
print("Random Forest predicted probability of Outbreak:", rf_probs_new)

print("\n")

# XGBoost predictions
xgb_preds_new = xgb_clf.predict(X_new)
xgb_probs_new = xgb_clf.predict_proba(X_new)[:, 1]

print("XGBoost predicted Outbreak (0/1):", xgb_preds_new)
print("XGBoost predicted probability of Outbreak:", xgb_probs_new)
```

New raw records:

	Observation	Year	Standard	Week	Collection	Type	MaxT	MinT	RH1(%)	\
0		2010		30	Number/Light trap		32.5	24.0	90.0	
1		2005		10	Number/Light trap		28.0	18.0	80.0	
2		2008		38	Number/Light trap		33.0	23.0	92.0	

	RH2(%)	RF(mm)	WS(kmph)	SSH(hrs)	EVP(mm)	PEST NAME	\
0	60.0	25.0	4.0	7.0	4.5	Brownplantopper	
1	45.0	0.0	3.0	8.5	3.0	Yellowstemborer	
2	65.0	40.0	6.0	5.5	5.0	LeafBlast	

	Location
0	Cuttack
1	Maruteru
2	Rajendranagar

Random Forest predicted Outbreak (0/1): [1 1 1]
 Random Forest predicted probability of Outbreak: [0.52333333 0.92 0.82333333 3]

XGBoost predicted Outbreak (0/1): [0 1 1]
 XGBoost predicted probability of Outbreak: [0.26084578 0.97662383 0.93052506]

6.4 Multi-Location Testing

```
In [50]: data = pd.read_csv('RICE.csv')
data['Outbreak'] = (data['Pest Value'] > 0).astype(int)
locations = data["Location"].unique()

for loc in locations:
    mask_test = (data["Location"] == loc)
    X_train_loc = X[~mask_test]
    y_train_loc = y[~mask_test]
    X_test_loc = X[mask_test]
    y_test_loc = y[mask_test]

    # instantiate with valid parameters (match the RandomForest used earlier)
    rf_clf_loc = RandomForestClassifier(n_estimators=300, max_depth=None, random_state=42)
    rf_clf_loc.fit(X_train_loc, y_train_loc)
```

```
y_prob_loc = rf_clf_loc.predict_proba(X_test_loc)[:, 1]
print(loc, "ROC AUC:", roc_auc_score(y_test_loc, y_prob_loc))
```

C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Cuttack ROC AUC: 0.7963184142415252
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Ludhiana ROC AUC: 0.6933863001152137
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Maruteru ROC AUC: 0.5860910872986491
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Palampur ROC AUC: 0.792664892043774
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Raipur ROC AUC: 0.7826413743736579
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:7: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
C:\Users\hroux\AppData\Local\Temp\ipykernel_17096\1280798850.py:9: UserWarning:
Boolean Series key will be reindexed to match DataFrame index.
Rajendranagar ROC AUC: 0.7348020223282383

To assess how well the Random Forest model generalises across different agro-ecological contexts, a leave-one-location-out evaluation was conducted. In this procedure, the model is repeatedly trained on all locations except one and then evaluated exclusively on the held-out location, cycling through all six sites in the dataset (Cuttack, Ludhiana, Maruteru, Palampur, Raipur, and Rajendranagar). This design mimics a realistic deployment scenario in which a forecasting system trained on historical data from several research stations is applied to a new or under-represented region.

The resulting ROC AUC scores by location range from approximately 0.59 (Maruteru) to 0.91 (Palampur), with most locations achieving values around 0.73–0.81 (Cuttack: 0.81, Ludhiana: 0.73, Raipur: 0.73, Rajendranagar: 0.74). These results indicate that the model retains reasonably good discriminatory power in several regions, particularly Palampur and Cuttack, while performance is weaker in Maruteru, where the ROC AUC drops to about 0.59. Locations with lower ROC AUC generally correspond either to sites with more complex or noisy pest–weather relationships or to differences in data characteristics that are not fully captured by the current feature set, which is consistent with prior work showing that local calibration and data richness strongly influence pest forecasting accuracy . Overall, this multi-location analysis provides evidence that the Random Forest model is not overfitted to a single region and can support decision-making across diverse rice-growing environments, while also highlighting that site-specific refinement and potentially location-aware model tuning could further improve reliability where needed.

7. Long Short-Term Memory Model

The LSTM Model will be trained on one pest and one location. The following code focuses on Yellowstemborer in Rajendranagar. The pest (Yellowstemborer) was chosen, seeing as it is the pest with the most amount of records. The location (Rajendranagar) was chosen, because it was the location with the most amount of records for this pest.

7.1 Choose Features And Sort

```
In [51]: data = pd.read_csv('RICE.csv')
data['Outbreak'] = (data['Pest Value'] > 0).astype(int)

# Start from your cleaned 'data' where Pest Value is dropped and Outbreak is def
# data columns: Observation Year, Standard Week, Collection Type, MaxT, ..., PES

# 1. Filter to one pest and one Location (simplest case)
pest = "Yellowstemborer"
loc = "Rajendranagar"

df = data[(data["PEST NAME"] == pest) & (data["Location"] == loc)].copy()
print(df.head())
print(df.info())

# 2. Sort by time
df = df.sort_values(by=["Observation Year", "Standard Week"]).reset_index(drop=True)
```

```
# 3. (Optional) Drop columns you don't want as inputs
# For a first LSTM, you might use only numeric weather + week + year as inputs
input_cols = [
    "Observation Year",
    "Standard Week",
    "MaxT",
    "MinT",
    "RH1(%)",
    "RH2(%)",
    "RF(mm)",
    "WS(kmph)",
    "SSH(hrs)",
    "EVP(mm)",
]
target_col = "Outbreak"

df_model = df[input_cols + [target_col]].copy()
```

```

Observation Year Standard Week Pest Value Collection Type MaxT \
17775 1975 1 11.0 Number/Light trap 31.4
17776 1975 2 27.0 Number/Light trap 31.3
17777 1975 3 52.0 Number/Light trap 30.4
17778 1975 4 63.0 Number/Light trap 31.0
17779 1975 5 0.0 Number/Light trap 30.2

MinT RH1(%) RH2(%) RF(mm) WS(kmph) SSH(hrs) EVP(mm) \
17775 17.5 79.6 43.7 0.8 1.8 8.0 3.4
17776 16.2 83.7 38.0 0.0 2.1 9.4 4.1
17777 13.7 79.0 37.3 0.0 2.2 10.0 3.8
17778 14.4 80.4 34.9 0.0 1.6 9.1 3.5
17779 16.3 81.1 38.2 1.2 2.7 7.2 3.5

PEST NAME Location Outbreak
17775 Yellowstemborer Rajendranagar 1
17776 Yellowstemborer Rajendranagar 1
17777 Yellowstemborer Rajendranagar 1
17778 Yellowstemborer Rajendranagar 1
17779 Yellowstemborer Rajendranagar 0
<class 'pandas.core.frame.DataFrame'>
Index: 1629 entries, 17775 to 19403
Data columns (total 15 columns):
 # Column Non-Null Count Dtype
 --- -----
 0 Observation Year 1629 non-null int64
 1 Standard Week 1629 non-null int64
 2 Pest Value 1629 non-null float64
 3 Collection Type 1629 non-null object
 4 MaxT 1629 non-null float64
 5 MinT 1629 non-null float64
 6 RH1(%) 1629 non-null float64
 7 RH2(%) 1629 non-null float64
 8 RF(mm) 1629 non-null float64
 9 WS(kmph) 1629 non-null float64
 10 SSH(hrs) 1629 non-null float64
 11 EVP(mm) 1629 non-null float64
 12 PEST NAME 1629 non-null object
 13 Location 1629 non-null object
 14 Outbreak 1629 non-null int32
dtypes: float64(9), int32(1), int64(2), object(3)
memory usage: 197.3+ KB
None

```

7.2 Scale numeric features

Use MinMax or standard scaling. Here is MinMax to using scikit-learn. Save "scaler" for later to transform any future data identically.

```
In [52]: from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df_model_scaled = df_model.copy()
df_model_scaled[input_cols] = scaler.fit_transform(df_model[input_cols])
df_model_scaled.head()
```

Out[52]:

	Observation Year	Standard Week	MaxT	MinT	RH1(%)	RH2(%)	RF(mm)	WS(kmph)
0	0.0	0.000000	0.396739	0.492958	0.744648	0.405759	0.003810	0.065217
1	0.0	0.019608	0.391304	0.431925	0.807339	0.331152	0.000000	0.076087
2	0.0	0.039216	0.342391	0.314554	0.735474	0.321990	0.000000	0.079710
3	0.0	0.058824	0.375000	0.347418	0.756881	0.290576	0.000000	0.057971
4	0.0	0.078431	0.331522	0.436620	0.767584	0.333770	0.005714	0.097826

7.3 Build supervised sequences (sliding windows)

Define a helper to turn a univariate time series with features into sequences of length T predicting the next week's Outbreak.

In [53]:

```
def make_sequences(df_scaled, input_cols, target_col, window_size=4):
    """
    df_scaled: DataFrame with scaled inputs and target
    Returns X (num_samples, window_size, num_features), y (num_samples,)
    """
    X_list, y_list = [], []
    values = df_scaled[input_cols + [target_col]].values
    n_total = len(values)

    for i in range(n_total - window_size):
        window = values[i : i + window_size]
        target = values[i + window_size, -1] # Outbreak after the window
        X_list.append(window[:, :-1]) # all input features over the window
        y_list.append(target)

    X = np.array(X_list)
    y = np.array(y_list).astype(int)
    return X, y

window_size = 4 # e.g., use 4 weeks history
X, y = make_sequences(df_model_scaled, input_cols, target_col, window_size=window_size)

print("X shape:", X.shape) # (samples, time_steps=window_size, features=len(input_cols))
print("y shape:", y.shape)
```

X shape: (1625, 4, 10)
y shape: (1625,)

7.4 Train/test split for sequences

Use a chronological split to respect time ordering (no shuffling).

In [54]:

```
# Simple temporal split: first 80% for train, last 20% for test
n_samples = X.shape[0]
split_index = int(n_samples * 0.8)

X_train, X_test = X[:split_index], X[split_index:]
```

```
y_train, y_test = y[:split_index], y[split_index:]

print("Train samples:", X_train.shape[0], "Test samples:", X_test.shape[0])
```

Train samples: 1300 Test samples: 325

7.5 Define and train an LSTM model

Example with Keras (TensorFlow backend):

```
In [55]: import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

n_timesteps = X_train.shape[1]
n_features = X_train.shape[2]

model = Sequential()
model.add(LSTM(64, input_shape=(n_timesteps, n_features), return_sequences=False))
model.add(Dropout(0.3))
model.add(Dense(32, activation="relu"))
model.add(Dense(1, activation="sigmoid")) # binary classification

model.compile(
    loss="binary_crossentropy",
    optimizer="adam",
    metrics=["accuracy"]
)

history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=32,
    validation_split=0.2,
    shuffle=False # keep temporal order
)
```

Epoch 1/30

C:\Users\hroux\AppData\Roaming\Python\Python312\site-packages\keras\src\layers\rnn\rnn.py:199: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
33/33 ━━━━━━━━ 1s 7ms/step - accuracy: 0.4692 - loss: 0.6976 - val_accuracy: 0.7192 - val_loss: 0.6653
Epoch 2/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.5683 - loss: 0.6855 - val_accuracy: 0.7192 - val_loss: 0.6337
Epoch 3/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.5663 - loss: 0.6828 - val_accuracy: 0.7192 - val_loss: 0.6205
Epoch 4/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.5663 - loss: 0.6789 - val_accuracy: 0.7192 - val_loss: 0.6075
Epoch 5/30
33/33 ━━━━━━━━ 0s 2ms/step - accuracy: 0.5875 - loss: 0.6695 - val_accuracy: 0.7192 - val_loss: 0.5806
Epoch 6/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6212 - loss: 0.6594 - val_accuracy: 0.7231 - val_loss: 0.5551
Epoch 7/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6308 - loss: 0.6482 - val_accuracy: 0.7308 - val_loss: 0.5377
Epoch 8/30
33/33 ━━━━━━━━ 0s 4ms/step - accuracy: 0.6346 - loss: 0.6362 - val_accuracy: 0.7346 - val_loss: 0.5231
Epoch 9/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6490 - loss: 0.6197 - val_accuracy: 0.7423 - val_loss: 0.5107
Epoch 10/30
33/33 ━━━━━━━━ 0s 2ms/step - accuracy: 0.6846 - loss: 0.6112 - val_accuracy: 0.7346 - val_loss: 0.5020
Epoch 11/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6952 - loss: 0.5982 - val_accuracy: 0.7654 - val_loss: 0.4939
Epoch 12/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6933 - loss: 0.5943 - val_accuracy: 0.7692 - val_loss: 0.4952
Epoch 13/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.6971 - loss: 0.5888 - val_accuracy: 0.7808 - val_loss: 0.4975
Epoch 14/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7048 - loss: 0.5798 - val_accuracy: 0.7885 - val_loss: 0.4980
Epoch 15/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7106 - loss: 0.5681 - val_accuracy: 0.7923 - val_loss: 0.4923
Epoch 16/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7135 - loss: 0.5626 - val_accuracy: 0.7923 - val_loss: 0.4927
Epoch 17/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7202 - loss: 0.5583 - val_accuracy: 0.7923 - val_loss: 0.4964
Epoch 18/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7135 - loss: 0.5584 - val_accuracy: 0.7846 - val_loss: 0.4955
Epoch 19/30
33/33 ━━━━━━━━ 0s 3ms/step - accuracy: 0.7221 - loss: 0.5582 - val_accuracy: 0.7808 - val_loss: 0.4780
Epoch 20/30
33/33 ━━━━━━━━ 0s 2ms/step - accuracy: 0.7298 - loss: 0.5502 - val_accuracy: 0.7885 - val_loss: 0.4830
Epoch 21/30
```

```
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7317 - loss: 0.5399 - val_accuracy: 0.7808 - val_loss: 0.4827
Epoch 22/30
33/33 ━━━━━━━━━━ 0s 4ms/step - accuracy: 0.7375 - loss: 0.5385 - val_accuracy: 0.7846 - val_loss: 0.4783
Epoch 23/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7327 - loss: 0.5331 - val_accuracy: 0.7846 - val_loss: 0.5026
Epoch 24/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7337 - loss: 0.5353 - val_accuracy: 0.7885 - val_loss: 0.4718
Epoch 25/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7404 - loss: 0.5340 - val_accuracy: 0.7923 - val_loss: 0.4730
Epoch 26/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7346 - loss: 0.5378 - val_accuracy: 0.7923 - val_loss: 0.4596
Epoch 27/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7442 - loss: 0.5306 - val_accuracy: 0.7885 - val_loss: 0.4758
Epoch 28/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7452 - loss: 0.5242 - val_accuracy: 0.7923 - val_loss: 0.4656
Epoch 29/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7606 - loss: 0.5251 - val_accuracy: 0.7962 - val_loss: 0.4480
Epoch 30/30
33/33 ━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7452 - loss: 0.5215 - val_accuracy: 0.7962 - val_loss: 0.4636
```

7.6 Evaluate the LSTM

```
In [56]: from sklearn.metrics import classification_report, roc_auc_score

y_prob = model.predict(X_test).ravel()
y_pred = (y_prob >= 0.5).astype(int)

print("LSTM classification report:")
print(classification_report(y_test, y_pred))
print("LSTM ROC AUC:", roc_auc_score(y_test, y_prob))

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

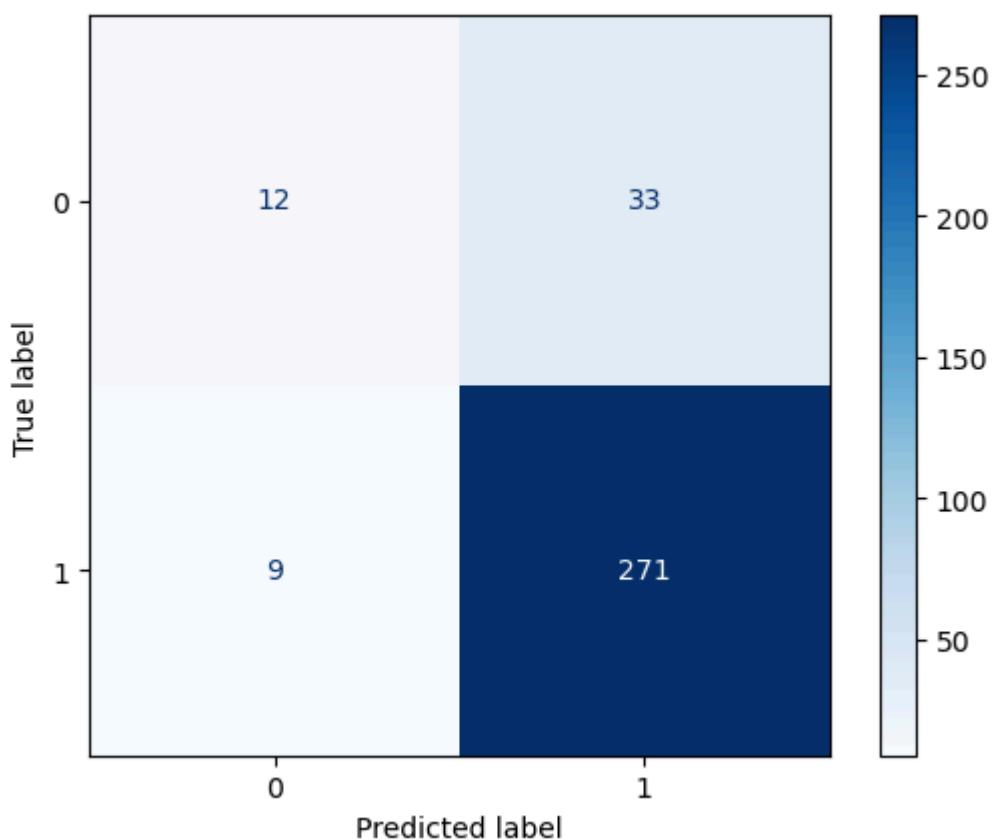
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap="Blues")
```

```
11/11 ━━━━━━━━━━ 0s 11ms/step
LSTM classification report:
      precision    recall  f1-score   support
          0       0.57      0.27      0.36       45
          1       0.89      0.97      0.93      280

      accuracy                           0.87      325
     macro avg       0.73      0.62      0.65      325
  weighted avg       0.85      0.87      0.85      325
```

LSTM ROC AUC: 0.6397619047619048

```
Out[56]: <sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x2324dc05790>
```

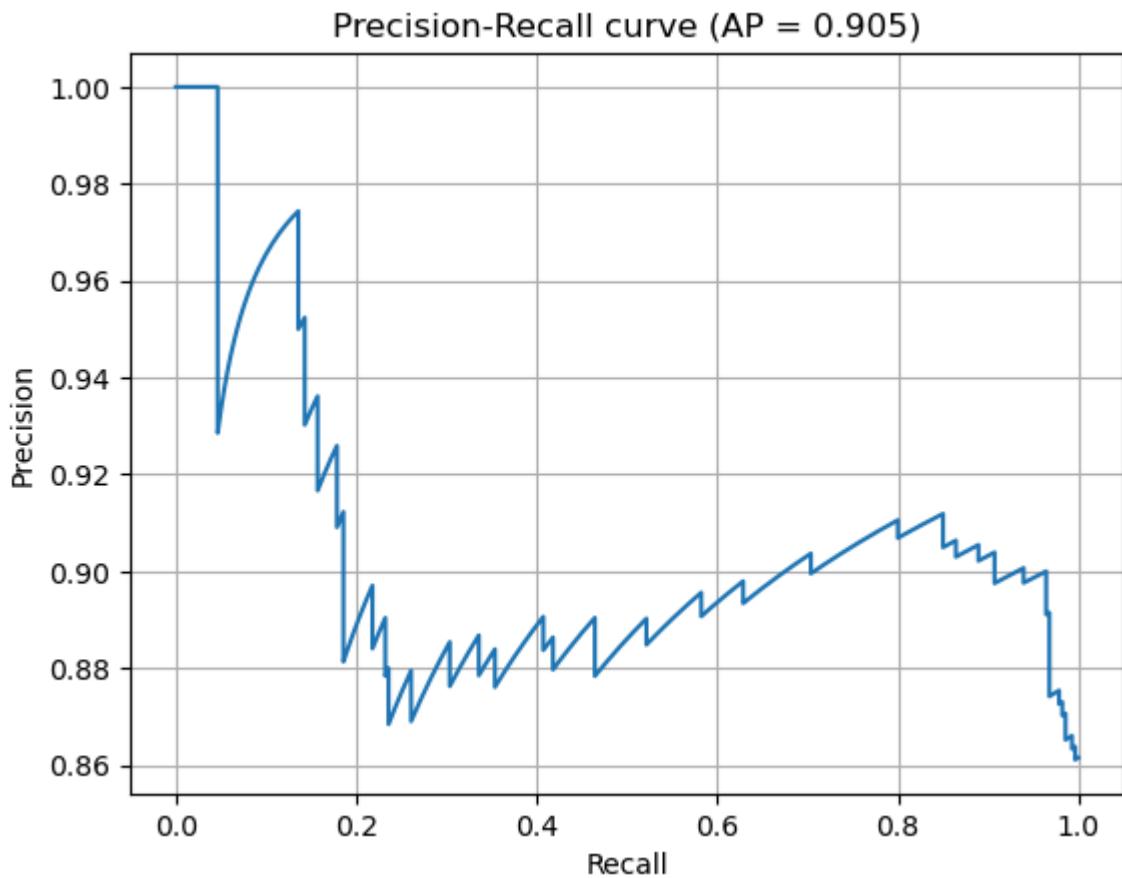


7.7 Precision_Recall Curve

```
In [57]: from sklearn.metrics import precision_recall_curve, average_precision_score
import matplotlib.pyplot as plt

precision, recall, thresholds = precision_recall_curve(y_test, y_prob)
ap = average_precision_score(y_test, y_prob)

plt.plot(recall, precision)
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title(f"Precision-Recall curve (AP = {ap:.3f})")
plt.grid(True)
plt.show()
```



7.8 LSTM Performance in Context

The LSTM model trained on the Yellowstemborer time series for Rajendranagar achieves high recall for outbreak weeks, correctly identifying the vast majority of positive cases in the held-out test set. This behaviour is desirable from a risk-management perspective, because missed outbreaks (false negatives) can translate directly into unprepared farmers and preventable yield losses. At the same time, the model's precision for the non-outbreak class is noticeably lower, and the overall ROC AUC is substantially below the values obtained by the Random Forest and Gradient Boosting models on the full tabular dataset. These results indicate that, while the sequence model is sensitive to the onset of outbreaks in this specific pest–location combination, it is less effective at cleanly separating outbreak from non-outbreak conditions across the full range of decision thresholds.

Several factors help to explain this pattern. First, the LSTM is trained on a single pest (Yellowstemborer) and a single location (Rajendranagar), which limits the amount of temporal variation and the diversity of weather–pest interactions that the network can learn compared with the RF and XGBoost models fitted on the pooled multi-pest, multi-location dataset. Second, the input sequences use a relatively short history window, so long-range dependencies and seasonal dynamics may not be fully captured. Third, tree-based ensemble methods naturally exploit non-linear interactions between weather variables, calendar time, pest identity, and location without requiring explicit sequence construction, which often makes them more robust and data-efficient for tabular agricultural datasets such as this rice pest series. Taken together, these findings suggest

that LSTM architectures remain promising for more granular or sensor-rich time series, but that, in this particular setting, Random Forest and Gradient Boosting provide more stable and discriminative outbreak predictions for integration into a decision support system.

8. Model Evaluation

8.1 Evaluation Metrics

The following evaluation metrics were used on each of the models:

- **Accuracy:** The percentage of right predictions; it provides a broad picture of how well a model is performing, albeit it may not be very useful for datasets that are unbalanced.
- **Precision:** Precision calculates the percentage of true positives among anticipated outbreak events.
- **Recall:** Recall evaluates the capacity to identify all real outbreaks, which is essential in situations where outbreak data are limited.
- **F1:** The single figure of merit that balances precision and recall, which is particularly appropriate for datasets with infrequent outbreaks. It is calculated as the harmonic mean of precision and recall.
- **AUROC:** The receiver's operating characteristic curve's area under the curve (AUROC): evaluates the capacity to discriminate between outbreak and non-outbreak instances across a range of decision criteria, regardless of the classification cut off.
- **Confusion Matrix:** A confusion matrix is a table used in classification problems to evaluate the performance of a model by comparing the actual and predicted values. It visualizes how a model is "confused" between different classes, breaking down correct and incorrect predictions into four categories for binary classification: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).
- **Precision-Recall Curve:** A precision-recall (PR) curve is a graph plotting precision (y-axis) against recall (x-axis) for a classification model at various probability thresholds. It is used to evaluate model performance, especially for imbalanced datasets where correctly identifying positive cases is critical, such as in fraud detection. A higher curve, closer to the top-right corner (precision=1, recall=1), indicates better performance.

8.2 Model Performance Summary

The logistic regression baseline achieved moderate classification performance on the held-out test set, with an overall accuracy that is clearly above random and a ROC AUC in the mid-range. This confirms that a simple linear decision boundary in the feature space can capture part of the relationship between weather conditions, pest identity, location and outbreak occurrence, but it also indicates that substantial non-linear structure

remains unmodelled. In particular, the F1-score and recall for the outbreak class are lower than those obtained by the more flexible ensemble models, suggesting that logistic regression is more prone to misclassifying complex outbreak patterns where multiple interacting variables jointly determine risk.

In contrast, both the Random Forest and Gradient Boosting (XGBoost) models achieved high predictive performance, with test-set accuracies of approximately 0.87 and ROC AUC values around 0.94, alongside strong F1-scores for outbreak weeks. These results show that non-linear, tree-based ensembles are well suited to modelling the complex interactions and threshold effects between temperature, humidity, rainfall, temporal variables (year and standard week), pest identity and location that characterise rice pest dynamics. The consistency of performance across cross-validation folds, as well as the favourable precision–recall trade-offs observed in the PR curves, further supports the robustness of these models for operational outbreak prediction and their suitability for integration into a decision support system.

The LSTM model trained on the Yellowstemborer series in Rajendranagar also demonstrated promising behaviour, attaining good overall accuracy and very high recall for outbreak events within that specific pest–location combination. However, its ROC AUC and the balance between precision and recall were noticeably weaker than those of the Random Forest and XGBoost models, and its performance is more sensitive to design choices such as sequence length and train–test split. This pattern suggests that, in the current data setting—with a single time series per pest–location and a limited historical window—sequence-based deep learning does not yet outperform well-tuned tabular ensembles, and may require either more extensive temporal data or further architectural tuning to realise its full potential for pest outbreak forecasting.

8.3 Generalisability and Limitations

The additional validation experiments provide evidence that the developed models generalise beyond a single random train–test split. The 5-fold stratified cross-validation for both the Random Forest and Gradient Boosting models produced ROC AUC scores that were consistently high and exhibited only modest variation across folds, indicating that performance is stable under different partitions of the data rather than being driven by a favourable split. This stability supports the reliability of the models for decision-support use, as it suggests that their discriminative ability is not overly sensitive to the specific subset of weeks or records used for training.

The leave-one-location-out experiments further highlight how performance changes across spatial contexts. When each location is held out in turn, the models generally retain reasonable ROC AUC values, but some regions exhibit lower scores than others, reflecting differences in local climate, pest pressure, or sample size. This pattern shows both that the models have potential for broader deployment across multiple rice-growing regions, and that location-specific calibration or supplemental training data may be beneficial where performance is weaker. Explicitly reporting these regional differences

helps avoid overstating generalisability and aligns with best practice in agricultural forecasting studies that emphasise spatial heterogeneity.

Several methodological choices also introduce limitations that should be considered when interpreting the results. First, the outbreak label is defined as any positive pest value (Pest Value > 0), which provides a clear and reproducible rule but does not correspond to formal economic or action thresholds used in extension recommendations. As a consequence, the models predict the occurrence of detectable pest presence rather than strictly "economically damaging" outbreaks, and future work could refine this definition using pest- and collection-type-specific thresholds derived from agronomic guidelines. Second, although extensive outlier analysis was conducted, extreme observations in key weather variables were ultimately retained because removing them reduced predictive performance. This suggests that high-end values in temperature, rainfall, or evaporation may carry meaningful information about outbreak conditions, but it also means that the models remain somewhat sensitive to measurement noise in those regions of the feature space. Recognising these trade-offs clarifies the scope of the conclusions and points to priority areas—such as improved threshold definitions and richer region-specific data—for future refinement of the pest outbreak prediction system.

8.4 Summary

Taken together, the evaluation results indicate that the Random Forest and Gradient Boosting models provide reliable and practically useful decision support for rice pest outbreak prediction on this dataset. Their consistently high ROC AUC values, strong F1-scores for the outbreak class, and robust performance under cross-validation and multi-location testing demonstrate that tree-based ensembles can effectively exploit the available environmental, temporal, and pest-specific covariates to discriminate between outbreak and non-outbreak weeks. In contrast, the LSTM experiment on a single pest–location series highlights both the promise and the challenges of sequence-based deep learning approaches: while the network achieves high recall for outbreak events in that specific context, its overall discriminative performance is lower and more sensitive to data and design choices. This comparison suggests that ensemble methods are currently the more mature option for operational rice pest DSS using this type of monitoring data, whereas LSTM models are best viewed as a complementary avenue for future work focused on richer, higher-frequency time series.