DOCUMENT

```
# First, we bring in NumPy and give it the alias 'np'.
# NumPy is like Python’s supercharged calculator for working with numbers.
# It’s especially good at handling large lists of numbers (arrays) and
# performing math on them very quickly — much faster than plain Python lists.

import numpy as np

# Next, we import Pandas and call it 'pd'.
# Pandas is our go-to tool for working with structured data — like tables or spreadsheets.
# It gives us the DataFrame and Series objects, which make it easy to load, explore,
# clean, and analyze data without having to manually write loops or messy indexing.

import pandas as pd

# Now we bring in Matplotlib's 'pyplot' module and call it 'plt'.
# Think of pyplot as the artist’s toolkit — it lets us create all kinds of charts:
# line graphs, bar charts, scatter plots, and more.
# It can handle simple static plots, as well as more dynamic and interactive visualizations.

import matplotlib.pyplot as plt

# Next, we import Seaborn and call it 'sb'.
# Seaborn is like Matplotlib’s stylish cousin — it builds on top of Matplotlib
# but comes with beautiful default themes, color palettes, and higher-level functions.
# It makes it easier to create attractive, statistically meaningful plots with less code.

import seaborn as sb

# From scikit-learn's model_selection module, we import 'train_test_split'.
# This is a helper function that takes your dataset and splits it into two parts:
# one for training the model and another for testing it, so we can check how well it performs.

from sklearn.model_selection import train_test_split

# From scikit-learn's preprocessing module, we import 'StandardScaler'.
# This tool standardizes your data — it rescales each feature so they have
# a mean of 0 and a standard deviation of 1.
# This helps many machine learning models work better and faster.

from sklearn.preprocessing import StandardScaler

# Finally, we import the 'metrics' module from scikit-learn.
# This is our evaluation toolkit — it contains functions to measure
# how well a model is performing, such as accuracy, precision, recall, and F1-score.
```

```
from sklearn import metrics

# From scikit-learn's svm (Support Vector Machine) module, we import 'SVC'.
# SVC stands for Support Vector Classifier — a powerful algorithm for classification tasks.
# It works by finding the optimal boundary (hyperplane) that separates different classes
# in the feature space, and it can handle both linear and non-linear decision boundaries.

from sklearn.svm import SVC

# Next, from the XGBoost library, we import 'XGBClassifier'.
# This is an implementation of Extreme Gradient Boosting —
# a fast, efficient, and highly accurate tree-based ensemble method.
# It's well-known for winning many machine learning competitions because of its
performance.

from xgboost import XGBClassifier

# Finally, from scikit-learn's linear_model module, we import 'LogisticRegression'.
# Logistic Regression is one of the simplest yet effective models for binary (and multi-class)
classification.
# It predicts probabilities using a logistic (sigmoid) function, making it easy to interpret.

from sklearn.linear_model import LogisticRegression

# From the imbalanced-learn (imblearn) library, we import 'RandomOverSampler'.
# This tool helps when our dataset has an imbalance — meaning some classes
# have far fewer samples than others.
# RandomOverSampler fixes this by randomly duplicating samples from the minority class
# until all classes have roughly equal representation, which can improve model fairness and
accuracy.

from imblearn.over_sampling import RandomOverSampler

# We import Python’s built-in 'warnings' module.
# This module controls warning messages that Python may show during execution
# — for example, deprecation warnings or harmless library alerts.

import warnings

# Here, we tell Python to ignore all warnings.
# This keeps the output cleaner, especially in notebooks or demo scripts,
# where too many warnings can clutter results.
# However, be careful — ignoring warnings means you might miss useful information
# about potential issues in your code.

warnings.filterwarnings('ignore')

# We use Pandas' 'read_csv' function to load data from a CSV file named 'Rainfall.csv'.
```

```
# This reads the file into a Pandas DataFrame (df) — a 2D table-like structure
# with labeled rows and columns.
# Once loaded, we can easily explore, clean, and analyze the dataset using Pandas
functions.

df = pd.read_csv('Rainfall.csv')

# We call the 'head()' method on our DataFrame (df).
# By default, 'head()' returns the first 5 rows of the dataset.
# This is a quick way to preview the data, check column names,
# and verify that the file loaded correctly.

df.head()

# We use the 'shape' attribute of the DataFrame (df).
# This returns a tuple (rows, columns), telling us the dimensions of the dataset.
# For example, (100, 5) would mean 100 rows and 5 columns.

df.shape

# We call the 'info()' method on our DataFrame (df).
# This gives us a concise summary of the dataset, including:
# - The total number of entries (rows)
# - The number of columns and their names
# - The data types of each column (e.g., int64, float64, object)
# - How many non-null (non-missing) values each column has
# This is useful for quickly understanding the structure and cleanliness of the data.

df.info()

# We call the 'describe()' method on our DataFrame (df).
# 'describe()' generates summary statistics for numeric columns, such as:
# count, mean, standard deviation, minimum, maximum, and quartiles.
# By default, the results are displayed with statistics as rows and columns as headers.

# Adding '.T' at the end transposes the table —
# this flips rows and columns so that each column's statistics are shown vertically.
# This often makes the output easier to read, especially for datasets with many columns.

df.describe().T

# We use the 'isnull()' method on our DataFrame (df).
# 'isnull()' returns a DataFrame of the same shape as df,
# with True where values are missing (NaN) and False otherwise.

# Then we call '.sum()' on the result.
# Since True is treated as 1 and False as 0 in Python,
# summing gives us the total number of missing values in each column.
```

```
# This helps us quickly identify which columns have null data and how many entries are
missing.

df.isnull().sum()

# We use the 'columns' attribute of our DataFrame (df).
# This returns an Index object containing all the column names in the dataset.
# It’s useful for quickly checking or iterating over the dataset’s features.

df.columns

# We use the 'rename()' method on our DataFrame (df) to clean up the column names.

df.rename(
    str.strip,          # Apply Python's string method .strip() to each column name,
                        # which removes any leading or trailing spaces.
    axis='columns',     # Specify that we’re renaming along the columns axis (not the rows
index).
    inplace=True        # Make the change directly in the original DataFrame without creating a
copy.
)

# After renaming, we check 'df.columns' again.
# This returns an Index object containing the updated column names.
# It’s a quick way to verify that our stripping of leading/trailing spaces worked.

df.columns

# Loop through each column in the DataFrame
for col in df.columns:

    # Check if the current column has any missing (NaN) values
    if df[col].isnull().sum() > 0:

        # Calculate the mean of the column.
        # By default, Pandas ignores NaN values in this calculation.
        val = df[col].mean()

        # Replace all NaN values in this column with the calculated mean.
        # This is a simple imputation technique that works best for numeric data.
        df[col] = df[col].fillna(val)

# First, 'df.isnull()' creates a DataFrame of the same shape as df,
# with True where values are missing (NaN) and False where values are present.

# Then, '.sum()' on this DataFrame counts the number of True values per column
# (since True = 1 and False = 0 in Python).
```

```
# Finally, the second '.sum()' adds up those column-wise counts to give
# the total number of missing values in the entire DataFrame.

df.isnull().sum().sum()

# We create a pie chart to visualize the distribution of values in the 'rainfall' column.

plt.pie(
   df['rainfall'].value_counts().values,  # The sizes of each pie slice —
                                          # taken from the counts of each unique value in 'rainfall'.

   labels=df['rainfall'].value_counts().index,  # The labels for the slices —
                                                # these are the unique values in 'rainfall'.

   autopct='%1.1f%%'  # Format for displaying the percentage of each slice.
                      # '%1.1f%%' means one decimal place followed by a percent sign.
)


# We call 'plt.show()' to display the plot.
# While some environments (like Jupyter notebooks) may auto-display plots,
# explicitly calling plt.show() ensures that:
# - The plot actually appears, especially in scripts or certain IDEs.
# - All previously defined plotting commands are rendered together.

plt.show()

# Create a new figure and set its size to 10x10 inches.
# This controls how large the plot will appear on screen or in saved output.
plt.figure(figsize=(10, 10))

# Create a heatmap using Seaborn.
sb.heatmap(
   df.corr() > 0.8,   # First, 'df.corr()' computes the correlation matrix for all numeric columns.
                      # Then, '> 0.8' turns it into a True/False matrix where True means
                      # the correlation is stronger than 0.8 (high correlation).

   annot=True,        # Annotate each cell with its True/False value.

   cbar=False         # Hide the color bar (since we're just showing True/False,
                      # a gradient color scale is not as meaningful here).
)

# Display the heatmap we just created.
# In many coding environments, plt.show() ensures the plot is actually rendered.
# It's especially important when running scripts outside of Jupyter notebooks,
# where plots won't appear automatically.
```

```
plt.show()

# We use the 'drop()' method to remove specific columns from our DataFrame.

df.drop(
   ['maxtemp', 'mintemp'],  # A list of column names to drop.

   axis=1,                # axis=1 means we're dropping columns (axis=0 would mean rows).

   inplace=True             # Make the change directly to the existing DataFrame
                         # instead of creating and returning a new one.
)


AFTER MODEL TRAINING

# We create a new DataFrame called 'features' that contains only the input variables (X).

features = df.drop(
   ['day', 'rainfall'],  # Columns to drop:
                      # 'day' might just be an identifier (not useful for prediction),
                      # and 'rainfall' is our target variable, which we don't include in features.

   axis=1               # axis=1 means we're removing columns (not rows).
)

# We create a variable called 'target' that stores our output variable (y).
# Here, 'df.rainfall' selects the 'rainfall' column from the DataFrame.
# This column will be used as the label we want our model to predict.

target = df.rainfall

# We use 'train_test_split' to split our dataset into training and validation sets.
This separates your data into training (to fit the model) and validation (to check
performance) sets, while keeping the class distribution balanced.

X_train, X_val, Y_train, Y_val = train_test_split(
   features,       # The independent variables (X) — what the model will learn from.
   target,         # The dependent variable (y) — what the model will try to predict.

   test_size=0.2,  # 20% of the data goes into the validation (or test) set,
                # and 80% will be used for training the model.

   stratify=target,# Ensures that the proportion of classes in 'target'
                # is preserved in both the training and validation sets
                # (important for classification problems with imbalanced classes).

   random_state=2  # Sets a seed for random number generation, so the split is reproducible
```

```
                # and you get the same data split every time you run the code.
)
# Making a RandomOverSampler object.
# This will duplicate examples from the smaller class until it's the same size as the bigger
one.
# 'minority' means only the smaller class gets oversampled.
# random_state=22 just makes sure you get the same result each time.

ros = RandomOverSampler(sampling_strategy='minority', random_state=22)


# Using the RandomOverSampler to balance the training data.
# fit_resample() learns which class is smaller and then duplicates its rows
# until both classes have the same number of samples.
# X and Y now hold the oversampled training data.

X, Y = ros.fit_resample(X_train, Y_train)


# Making a StandardScaler to normalize the features (mean=0, std=1).
scaler = StandardScaler()

# Fit it on the training data and scale it.
X = scaler.fit_transform(X)

# Scale the validation data using the same scaler.
X_val = scaler.transform(X_val)

# List of models to test.
models = [
    LogisticRegression(),   # Simple linear model.
    XGBClassifier(),        # Gradient boosting, usually very accurate.
    SVC(kernel='rbf', probability=True)  # SVM with RBF kernel, can output probabilities.
]

# Loop through numbers 0, 1, 2 — one for each model in the list.
for i in range(3):

# Train the i-th model using the oversampled training data.
models[i].fit(X, Y)

# Print the name/details of the current model.
print(f'{models[i]} : ')

# Get the predicted probabilities for the training data from the current model.
train_preds = models[i].predict_proba(X)

# Print the training AUC score — measures how well the model separates the classes.
```

```
# train_preds[:,1] is the probability of the positive class.
print('Training Accuracy : ', metrics.roc_auc_score(Y, train_preds[:, 1]))
# Get the predicted probabilities for the validation data.
val_preds = models[i].predict_proba(X_val)

# Print the validation AUC score — shows how well the model performs on unseen data.
print('Validation Accuracy : ', metrics.roc_auc_score(Y_val, val_preds[:, 1]))

# Blank line for readability between model outputs.
print()

# Import Matplotlib for plotting.
import matplotlib.pyplot as plt

# Import ConfusionMatrixDisplay to easily show confusion matrices.
from sklearn.metrics import ConfusionMatrixDisplay

# Import metrics module from sklearn — has accuracy, precision, recall, etc.
from sklearn import metrics

# Show the confusion matrix for the SVC model on validation data.
ConfusionMatrixDisplay.from_estimator(
    models[2],  # The trained SVC model (3rd in the list).
    X_val,      # Validation features.
    Y_val       # True validation labels.
)

# Display the confusion matrix plot.
plt.show()

# Print a classification report for the SVC model on validation data.
# Shows precision, recall, f1-score, and support for each class.
print(metrics.classification_report(
    Y_val,                # True validation labels.
    models[2].predict(X_val)  # Predicted labels from the SVC model.
))
```