**corr\_matrix = df.corr()**

**# plot the heat map**

**fig, ax = plt.subplots(figsize=(20, 15))**

**sns.heatmap(corr\_matrix, annot=True, ax=ax, cmap='coolwarm')**

**plt.show() meaning samjhao**

1. corr\_matrix = df.corr()

This line calculates the correlation matrix for the dataset (df).

The correlation matrix is a table showing correlation coefficients between variables (columns). Each cell in the table shows the correlation between two variables.

Correlation values range from -1 to 1:

+1: Perfect positive correlation (as one variable increases, the other also increases).

0: No correlation (no relationship between the variables).

-1: Perfect negative correlation (as one variable increases, the other decreases).

2. fig, ax = plt.subplots(figsize=(20, 15))

This line creates a new figure (fig) and a set of subplots (ax) with a specified size.

figsize=(20, 15) sets the size of the figure to be 20 inches wide and 15 inches tall.

3. sns.heatmap(corr\_matrix, annot=True, ax=ax, cmap='coolwarm')

This line creates a heatmap of the correlation matrix using Seaborn (sns).

corr\_matrix: The data (correlation matrix) to be visualized.

annot=True: This adds the actual correlation coefficients as text annotations on each cell of the heatmap.

ax=ax: Specifies which subplot to draw the heatmap on (in this case, the one created earlier).

cmap='coolwarm': This sets the color palette for the heatmap, with colors ranging from cool (blue) for negative correlations to warm (red) for positive correlations.

4. plt.show()

This line displays the plot. It’s the final command that shows the heatmap on the screen.

Meaning:

The whole block of code is used to create a visual representation of the correlation between different variables in your dataset.

The heatmap will help you quickly identify which variables are strongly correlated (either positively or negatively) with each other, making it easier to spot patterns or relationships in the data.

**from matplotlib import pyplot as grph**

**df.hist(figsize=(18, 14), color='blue', edgecolor='black')**

**grph.tight\_layout()**

**grph.show() meaning**

1. from matplotlib import pyplot as grph

This line imports the pyplot module from the matplotlib library and gives it the alias grph.

matplotlib.pyplot is a collection of functions that make matplotlib work like MATLAB, allowing for easy plotting and customization of plots.

2. df.hist(figsize=(18, 14), color='blue', edgecolor='black')

df.hist(...): This method creates histograms for all numerical columns in the DataFrame df.

figsize=(18, 14): Specifies the overall size of the plot, setting it to be 18 inches wide and 14 inches tall.

color='blue': Sets the color of the bars in the histograms to blue.

edgecolor='black': Adds a black border around each bar in the histograms, making them easier to distinguish.

3. grph.tight\_layout()

This function adjusts the spacing between subplots to prevent overlapping labels and ensure that everything fits nicely within the figure area.

It optimizes the layout so that all elements are displayed clearly and neatly.

4. grph.show()

This command displays the figure on the screen. It’s the final step that renders the histograms for viewing.

Meaning:

The code generates histograms for each numerical column in your DataFrame (df).

Each histogram shows the distribution of values in that column, which helps in understanding the frequency of data points within different ranges.

The histograms are color-coded in blue with black edges for clarity, and the layout is adjusted to ensure a clean presentation.

This visualization is useful for getting a sense of how your data is distributed, identifying skewness, outliers, and the general shape of the data distribution.

**from sklearn.metrics import accuracy\_score**

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

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

**import matplotlib.pyplot as plt**

**from sklearn.preprocessing import PolynomialFeatures**

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

1. from sklearn.metrics import accuracy\_score

This function is used to calculate the accuracy of a classification model.

Accuracy is the ratio of correctly predicted instances to the total instances. It's a common evaluation metric for classification models.

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

This function is used to split a dataset into two subsets: a training set and a test set.

The training set is used to train the model, and the test set is used to evaluate its performance.

It helps in validating how well the model generalizes to unseen data.

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

mean\_absolute\_error: This metric measures the average magnitude of errors in predictions, without considering their direction. It is the average of the absolute differences between predicted and actual values.

mean\_squared\_error: This metric measures the average of the squares of the errors between predicted and actual values. It’s more sensitive to outliers than mean absolute error.

r2\_score: Also known as the coefficient of determination, it represents how well the model's predictions fit the actual data. An

𝑅

2

R

2

value of 1 indicates perfect prediction, while 0 indicates that the model does no better than simply predicting the mean.

4. import matplotlib.pyplot as plt

This is the standard import for the matplotlib plotting library, which is used for creating static, interactive, and animated visualizations in Python.

pyplot provides a MATLAB-like interface for creating figures and plots.

5. from sklearn.preprocessing import PolynomialFeatures

This function is used to transform features into polynomial features, enabling the creation of polynomial regression models.

For example, if you have a feature

𝑥

x, using PolynomialFeatures can generate features like

𝑥

2

x

2

,

𝑥

3

x

3

, etc., which can be used to model more complex relationships between the input variables and the output variable.

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

These metrics are repeated from above and are used to evaluate the performance of regression models.

mean\_squared\_error (already explained) measures the average squared difference between predicted and actual values.

r2\_score (already explained) indicates how well the model's predictions approximate the actual data.

Summary:

These imports are used to handle various tasks in building and evaluating machine learning models.

The accuracy\_score is used for classification tasks, while the mean\_absolute\_error, mean\_squared\_error, and r2\_score are used for regression tasks.

train\_test\_split is essential for splitting data into training and testing sets, ensuring that model evaluation is fair.

PolynomialFeatures allows you to transform your data to fit polynomial regression models.

matplotlib.pyplot is used to visualize the data and model outputs.

**from sklearn.linear\_model import LogisticRegression**

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

**# logistic regression**

**from sklearn.linear\_model import LogisticRegression**

**model = LogisticRegression()**

1. from sklearn.linear\_model import LogisticRegression

This line imports the LogisticRegression class from the sklearn.linear\_model module.

Logistic Regression is a type of regression analysis used for predicting binary outcomes (e.g., 0 or 1, true or false). It's commonly used for classification problems.

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

This line imports the train\_test\_split function, which is used to split the dataset into training and testing sets.

3. model = LogisticRegression()

This line creates an instance of the LogisticRegression model.

The variable model now holds this logistic regression model, which can be trained on data and used to make predictions.

**X=df.copy()**

**X.drop(columns=['FloodProbability'],inplace=True)**

**y=df[['FloodProbability']]**

**print(df)**

**#splitting the data into training data and test data ( aorund 20% of data will be used for testing the results whereas 80% for training the model)**

**#X\_train contains training data**

**# y\_train cotains result set for training data ( both X\_train and y\_train will train the model)**

**# X\_test contains the test data**

**# y\_test contains test expected Results**

**#X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)**

**X\_train\_original=X\_train**

**X\_test\_original=X\_test**

**y\_train\_original=y\_train**

**y\_test\_original=y\_test**

**X\_train\_original**

In the code snippet you provided, X\_train\_original represents the training data features before any modifications or preprocessing steps are applied. Let me break it down for you:

Data Preparation:

X is a copy of the original DataFrame df with the column 'FloodProbability' removed.

y is a DataFrame containing only the 'FloodProbability' column.

Splitting Data:

train\_test\_split splits the dataset into training and testing sets.

X\_train and y\_train contain the features and target values for the training set.

X\_test and y\_test contain the features and target values for the testing set.

Original Data Variables:

X\_train\_original, X\_test\_original, y\_train\_original, and y\_test\_original are simply variables that hold the original training and testing data. These are not modified or processed further in this snippet.

So, X\_train\_original is the original training feature set that you can use to refer back to the initial split data if needed.

import numpy as np

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

# Example data

X = np.random.rand(100, 10) # 100 samples, 10 features

y = np.random.randint(0, 2, 100) # 100 labels

# Incorrect split (example) - this should be avoided

# x\_train, y\_train = X[:80], y[:90]

# Correct split

x\_train, x\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

print(f"Length of x\_train: {len(x\_train)}")

print(f"Length of y\_train: {len(y\_train)}")

# Ensure consistent length before fitting

if len(x\_train) == len(y\_train):

from sklearn.linear\_model import LogisticRegression

model = LogisticRegression()

model.fit(x\_train, y\_train)

else:

print("Lengths of x\_train and y\_train do not match.")

In your code, you’re using train\_test\_split to split your data into training and testing sets. Here’s a step-by-step explanation of each part and the checks performed:

Generating Example Data:

X = np.random.rand(100, 10) creates a dataset with 100 samples and 10 features.

y = np.random.randint(0, 2, 100) creates a target array with 100 labels (binary classification in this case).

Incorrect Split Example:

x\_train, y\_train = X[:80], y[:90] is an incorrect split. The lengths of x\_train (80 samples) and y\_train (90 labels) are mismatched, which will cause issues when fitting a model.

Correct Split:

train\_test\_split correctly splits X and y into training and testing sets. With test\_size=0.2, 20% of the data is used for testing, and the remaining 80% is used for training.

x\_train and y\_train will have the same number of samples (80), and x\_test and y\_test will have the same number of samples (20).

Check for Consistent Length:

The if statement checks whether the lengths of x\_train and y\_train are the same. If they are, it proceeds to fit a logistic regression model.

If the lengths don’t match, it prints an error message. This ensures that your feature and target arrays are aligned before training the model.

In summary, this script ensures that the training data and corresponding labels are correctly matched in length before fitting the model, avoiding potential issues with model training.

**Hyperparameter tuning**

**from sklearn.ensemble import RandomForestRegressor**

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

**import pandas as pd**

**# Load your dataset**

**df = pd.read\_csv('flood.csv')**

**# Define features and target variable**

**X = df.drop(columns=['FloodProbability'])**

**y = df['FloodProbability']**

**# Split the data into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**# Define the Random Forest model**

**rf = RandomForestRegressor(random\_state=42)**

**# Define a smaller hyperparameters grid to search**

**param\_grid = {**

**'n\_estimators': [100, 200],**

**'max\_depth': [10, 20],**

**'min\_samples\_split': [2, 5],**

**'min\_samples\_leaf': [1, 2]**

**}**

**# Set up the Grid Search with cross-validation**

**grid\_search = GridSearchCV(estimator=rf, param\_grid=param\_grid,**

**cv=3, n\_jobs=-1, verbose=2, scoring='neg\_mean\_squared\_error')**

**# Fit the Grid Search to the data**

**grid\_search.fit(X\_train, y\_train)**

**# Get the best parameters and the best score**

**best\_params = grid\_search.best\_params\_**

**best\_score = grid\_search.best\_score\_**

**print("Best parameters found: ", best\_params)**

**print("Best score: ", best\_score)**

Your code sets up a RandomForestRegressor model and performs hyperparameter tuning using GridSearchCV. Here’s a breakdown of what each part of your code does:

Load the Dataset:

df = pd.read\_csv('flood.csv') reads your dataset from a CSV file.

Define Features and Target Variable:

X is created by dropping the 'FloodProbability' column, which will be used as features.

y is the 'FloodProbability' column, which is the target variable.

Split the Data:

train\_test\_split divides the data into training and testing sets with 20% of the data reserved for testing. random\_state=42 ensures reproducibility.

Define the Model:

rf = RandomForestRegressor(random\_state=42) initializes a Random Forest Regressor with a fixed random seed.

Set Up Hyperparameter Grid:

param\_grid defines a small grid of hyperparameters to search. These include:

n\_estimators: Number of trees in the forest.

max\_depth: Maximum depth of the trees.

min\_samples\_split: Minimum number of samples required to split an internal node.

min\_samples\_leaf: Minimum number of samples required to be at a leaf node.

Grid Search with Cross-Validation:

GridSearchCV performs an exhaustive search over the specified parameter grid with cross-validation (cv=3 means 3-fold cross-validation).

n\_jobs=-1 allows the process to use all available cores for faster computation.

verbose=2 provides detailed output during the fitting process.

scoring='neg\_mean\_squared\_error' evaluates the model based on the negative mean squared error (the more negative, the better the performance).

Fit the Grid Search:

grid\_search.fit(X\_train, y\_train) performs the grid search on the training data.

Get Best Parameters and Score:

grid\_search.best\_params\_ retrieves the best combination of hyperparameters.

grid\_search.best\_score\_ retrieves the best score achieved during cross-validation.

The print statements at the end display the best hyperparameters and the corresponding score.

**SVM Hyperparameter Tuning**

**from sklearn.svm import SVR**

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

**import pandas as pd**

**# Define features and target variable**

**X = df.drop(columns=['FloodProbability'])**

**y = df['FloodProbability']**

**# Split the data into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**# Define the SVM model**

**svm = SVR()**

**# Define the hyperparameters grid to search**

**param\_grid = {**

**'C': [0.1, 1, 10, 100],**

**'gamma': [1, 0.1, 0.01, 0.001]**

**}**

**# Set up the Grid Search with cross-validation**

**grid\_search = GridSearchCV(estimator=svm, param\_grid=param\_grid,**

**cv=3, n\_jobs=-1, verbose=2, scoring='neg\_mean\_squared\_error')**

**# Fit the Grid Search to the data**

**grid\_search.fit(X\_train, y\_train)**

**# Get the best parameters and the best score**

**best\_params = grid\_search.best\_params\_**

**best\_score = grid\_search.best\_score\_**

**print("Best parameters found: ", best\_params)**

**print("Best score: ", best\_score)**

Your code uses SVR (Support Vector Regression) from scikit-learn and performs hyperparameter tuning with GridSearchCV. Here’s a step-by-step explanation:

Define Features and Target Variable:

X contains the features (all columns except 'FloodProbability').

y is the target variable ('FloodProbability').

Split the Data:

train\_test\_split divides the data into training and testing sets, reserving 20% for testing and using 80% for training. random\_state=42 ensures that the split is reproducible.

Define the SVM Model:

svm = SVR() initializes a Support Vector Regression model. SVR is used for regression tasks and is based on Support Vector Machines (SVM).

Set Up the Hyperparameter Grid:

param\_grid defines the hyperparameters to tune:

C: Regularization parameter. The strength of the regularization is inversely proportional to C. Higher values of C make the model fit the training data more closely.

gamma: Kernel coefficient for ‘rbf’, ‘poly’, and ‘sigmoid’. The higher the value, the closer other samples must be to be considered in the decision boundary.

Grid Search with Cross-Validation:

GridSearchCV performs an exhaustive search over the specified parameter grid.

cv=3 means 3-fold cross-validation is used.

n\_jobs=-1 allows the use of all available CPU cores for faster computation.

verbose=2 provides detailed output during the fitting process.

scoring='neg\_mean\_squared\_error' evaluates the model based on the negative mean squared error. The more negative the score, the better the model’s performance.

Fit the Grid Search:

grid\_search.fit(X\_train, y\_train) performs the grid search on the training data, evaluating each combination of hyperparameters using cross-validation.

Retrieve Best Parameters and Score:

grid\_search.best\_params\_ returns the hyperparameters that achieved the best performance.

grid\_search.best\_score\_ provides the best score (negative mean squared error) found during the grid search.

The print statements at the end show the best hyperparameters found and the corresponding performance score of the model.

**Gradient Descent-based Optimizers**

**import tensorflow as tf**

**from tensorflow.keras.models import Sequential**

**from tensorflow.keras.models import tensorflow.keras.wrappers**

**from tensorflow.keras.layers import Dense**

**from tensorflow.keras.wrappers.scikit\_learn import KerasRegressor**

**from tensorflow.keras.optimizers import SGD, Adam**

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

**# Define features and target variable**

**X = df.drop(columns=['FloodProbability'])**

**y = df['FloodProbability']**

**# Split the data into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**# Define a function to create the model**

**def create\_model(optimizer='adam', learning\_rate=0.01):**

**model = Sequential()**

**model.add(Dense(64, input\_dim=X\_train.shape[1], activation='relu'))**

**model.add(Dense(32, activation='relu'))**

**model.add(Dense(1, activation='linear'))**

**if optimizer == 'adam':**

**opt = Adam(learning\_rate=learning\_rate)**

**else:**

**opt = SGD(learning\_rate=learning\_rate)**

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

**return model**

**# Wrap the Keras model so it can be used by scikit-learn**

**model = KerasRegressor(build\_fn=create\_model, verbose=0)**

**# Define the hyperparameters grid to search**

**param\_grid = {**

**'optimizer': ['adam', 'sgd'],**

**'learning\_rate': [0.001, 0.01, 0.1],**

**'epochs': [50, 100],**

**'batch\_size': [10, 20]**

**}**

**# Set up the Grid Search with cross-validation**

**grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid,**

**cv=3, verbose=2, scoring='neg\_mean\_squared\_error')**

**# Fit the Grid Search to the data**

**grid\_search.fit(X\_train, y\_train)**

**# Get the best parameters and the best score**

**best\_params = grid\_search.best\_params\_**

**best\_score = grid\_search.best\_score\_**

**print("Best parameters found: ", best\_params)**

**print("Best score: ", best\_score)**