import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)

import seaborn as sns

import matplotlib.pyplot as plt

NumPy (np):

* NumPy is a fundamental package for scientific computing in Python.
* It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
* NumPy is often used for numerical computations and data analysis.
* Common tasks include array manipulation, mathematical operations, linear algebra, random number generation, and more.

Pandas (pd):

* Pandas is a powerful data analysis and manipulation library for Python.
* It provides data structures and functions designed to make working with structured data (such as CSV files or SQL databases) fast, easy, and expressive.
* Pandas primarily deals with two main data structures: Series (1-dimensional labeled array) and DataFrame (2-dimensional labeled data structure with columns of potentially different types).
* Typical tasks with Pandas include data loading, cleaning, manipulation, aggregation, and analysis.

Seaborn:

* Seaborn is a statistical data visualization library based on Matplotlib.
* It provides a high-level interface for drawing attractive and informative statistical graphics.
* Seaborn simplifies the process of creating complex visualizations like scatter plots, histograms, box plots, heatmaps, and more.
* It offers additional features compared to Matplotlib, such as built-in themes, color palettes, and concise syntax for common statistical plots.

Matplotlib.pyplot (plt):

* Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
* Matplotlib.pyplot is a collection of command style functions that make Matplotlib work like MATLAB.
* It provides a convenient interface for creating plots and visualizations with just a few lines of code.
* With Matplotlib.pyplot, you can create various types of plots, customize their appearance, add labels and annotations, and save or display them.

In summary, NumPy and Pandas are primarily used for data manipulation and analysis, while Seaborn and Matplotlib.pyplot are used for data visualization. They are often used together in data science workflows to explore, analyze, and communicate insights from data.

df = pd.read\_csv('yield\_df.csv')

The line **df = pd.read\_csv('yield\_df.csv')** reads data from a CSV file named **'yield\_df.csv'** and stores it in a Pandas DataFrame called **df**.

df.head()

**df.head()** displays the first few rows of the DataFrame **df**, giving a quick overview of its structure and contents.

df.drop('Unnamed: 0',axis=1,inplace=True)

This code drops the column labeled **'Unnamed: 0'** from the DataFrame **df**.

df.shape

**df.shape** returns a tuple representing the dimensions of the DataFrame **df**, where the first element is the number of rows and the second element is the number of columns.

df.info()

**df.info()** is a method in Pandas used to print a concise summary of a DataFrame's structure and content.

Here's what it provides:

* The total number of entries in the DataFrame.
* The data type of each column.
* The number of non-null values in each column.
* Memory usage information.

df.isnull().sum()

**df.isnull().sum()** is a Pandas DataFrame method that calculates the number of missing (null) values in each column of the DataFrame **df** and returns a Series containing these counts.

This method is useful for quickly identifying columns with missing values and assessing the extent of missing data in each column.

df.duplicated().sum()

**df.duplicated().sum()** is a Pandas DataFrame method that calculates the number of duplicated rows in the DataFrame **df** and returns the sum of these duplicated rows.

This method is useful for identifying and quantifying the presence of duplicate rows in a DataFrame, which may need to be addressed depending on the analysis or processing requirements.

df.drop\_duplicates(inplace=True)

This line of code removes duplicate rows from the DataFrame **df** and modifies **df** in place.

**Transforming average\_rain\_fall\_mm\_per\_year**

In summary, this code identifies the indices of rows in the DataFrame df where the values in the column 'average\_rain\_fall\_mm\_per\_year' are not numeric strings. These rows can be considered for removal or further processing, depending on the specific use case.

def isStr(obj):

    try:

        float(obj)

        return False

    except:

        return True

to\_drop = df[df['average\_rain\_fall\_mm\_per\_year'].apply(isStr)].index

This code defines a function **isStr(obj)** that checks if a given object **obj** can be converted to a float. If the conversion to float is successful (i.e., it's a numeric value), the function returns **False**, indicating it's not a string. If an error occurs during the conversion (indicating it's not numeric), the function returns **True**, indicating it's a string.

Then, it uses this function to identify the indices of rows in the DataFrame **df** where the value in the column **'average\_rain\_fall\_mm\_per\_year'** is a string. These indices are stored in the variable **to\_drop**.

df = df.drop(to\_drop)

his line of code removes rows from the DataFrame **df** whose indices are stored in the variable **to\_drop**. It effectively drops the rows where the value in the column

df

**df** is a common placeholder name for a Pandas DataFrame in Python

df['average\_rain\_fall\_mm\_per\_year'] = df['average\_rain\_fall\_mm\_per\_year'].astype(np.float64)

This line of code converts the data type of the column **'average\_rain\_fall\_mm\_per\_year'** in the DataFrame **df** to **float64** using the **astype()** method. This method is a Pandas function used to cast the data type of a Series to a specified data type. In this case, it's converting the data type to a 64-bit floating-point number (**np.float64**), which is a numeric data type suitable for storing decimal numbers with higher precision. This conversion is useful when you want to perform mathematical operations or analysis on the data in that column, as floating-point numbers are typically used for numerical computations.

Top of Form

Graph Frequency vs Area

len(df['Area'].unique())

This code calculates the number of unique values in the 'Area' column of the DataFrame **df** using the **unique()** method and then determines the length of this unique set using the **len()** function.

plt.figure(figsize=(15,20))

sns.countplot(y=df['Area'])

plt.show()

This code creates a countplot using Seaborn to visualize the distribution of values in the 'Area' column of the DataFrame **df**.

* **plt.figure(figsize=(15,20))**: This sets the size of the figure for the plot to 15 units in width and 20 units in height.
* **sns.countplot(y=df['Area'])**: This creates a countplot where the counts of each unique value in the 'Area' column are plotted vertically along the y-axis.
* **plt.show()**: This displays the plot.

The countplot is a type of bar plot that shows the frequency of each category in a categorical variable, which in this case is the 'Area' column.

(df['Area'].value\_counts() < 500).sum()

This code calculates the number of unique values in the 'Area' column of the DataFrame **df** that have a count less than 500 using the **value\_counts()** method and then applying a condition to check if each count is less than 500. Finally, it sums up the counts that satisfy this condition.

Yield\_per\_country

country = df['Area'].unique()

yield\_per\_country = []

for state in country:

    yield\_per\_country.append(df[df['Area']==state]['hg/ha\_yield'].sum())

This code appears to be iterating over unique values in the 'Area' column of a DataFrame (presumably named df). For each unique value (presumably representing a country or state), it calculates the sum of the 'hg/ha\_yield' column within that country or state and appends it to the list yield\_per\_country. Essentially, it's aggregating the total yield per country or state.

df['hg/ha\_yield'].sum()

So, **df['hg/ha\_yield'].sum()** computes the total sum of all values in the 'hg/ha\_yield' column of the DataFrame **df**. It effectively aggregates all the values in that column into a single scalar value, representing the total yield.

yield\_per\_country

Certainly! **yield\_per\_country** is a list that contains the total yield for each unique country or region in the dataset. Each element in the list corresponds to the sum of yields for a specific country or region, computed from the 'hg/ha\_yield' column in the DataFrame.

Yield\_per\_country Graph

plt.figure(figsize=(15, 20))

sns.barplot(y=country, x=yield\_per\_country)

This code snippet utilizes matplotlib and seaborn libraries to create a bar plot. Here's a brief explanation:

1. **plt.figure(figsize=(15, 20))**: This line sets the size of the figure that will contain the bar plot. The **figsize** parameter specifies the width and height of the figure in inches.
2. **sns.barplot(y=country, x=yield\_per\_country)**: This line creates a bar plot using seaborn's **barplot** function. It specifies the **country** list as the values for the y-axis (typically the categorical axis) and **yield\_per\_country** list as the values for the x-axis (typically the numerical axis). This will create a horizontal bar plot where each bar represents the total yield for a specific country or region.

Overall, this code generates a horizontal bar plot showing the total yield for each country or region in the dataset.

**Graph Frequency per item**

sns.countplot(y=df['Item'])

This line of code creates a count plot using seaborn's **countplot** function. It displays the count of each unique value in the 'Item' column of the DataFrame **df** on the y-axis. Each bar in the plot represents the frequency of occurrence of a specific item in the dataset.

**Yield vs Item**

crops = df['Item'].unique()

yield\_per\_crop = []

for crop in crops:

    yield\_per\_crop.append(df[df['Item']==crop]['hg/ha\_yield'].sum())

This code calculates the total yield for each unique crop present in the DataFrame **df**. Here's a brief explanation:

1. **crops = df['Item'].unique()**: This line retrieves all unique values in the 'Item' column of the DataFrame **df** and assigns them to the variable **crops**. It likely represents a list of different crop types.
2. **yield\_per\_crop = []**: This initializes an empty list named **yield\_per\_crop** which will store the sum of yields for each crop.
3. **for crop in crops:**: This initiates a loop where **crop** iterates over each unique crop type in the **crops** list.
4. **yield\_per\_crop.append(df[df['Item']==crop]['hg/ha\_yield'].sum())**: Within the loop, this line calculates the sum of yields for each crop and appends it to the **yield\_per\_crop** list. Let's break it down further:
   * **df[df['Item']==crop]**: This filters the DataFrame **df** to only include rows where the 'Item' column matches the current **crop**.
   * **['hg/ha\_yield']**: This selects the 'hg/ha\_yield' column from the filtered DataFrame.
   * **.sum()**: This calculates the sum of all values in the 'hg/ha\_yield' column for the filtered DataFrame.
   * **append()**: This adds the calculated sum to the **yield\_per\_crop** list.

After the loop completes, **yield\_per\_crop** will contain the sum of yields for each crop type present in the DataFrame **df**

sns.barplot(y=crops,x=yield\_per\_crop)

This line of code creates a bar plot using seaborn's **barplot** function. It displays the total yield for each unique crop type present in the dataset. Here's a brief explanation:

* **sns.barplot(y=crops, x=yield\_per\_crop)**: This line specifies the crop types (**crops**) as values for the y-axis (typically the categorical axis) and **yield\_per\_crop** as values for the x-axis (typically the numerical axis). It creates a vertical bar plot where each bar represents the total yield for a specific crop type.

**Train test split rearranging column**

col = ['Year', 'average\_rain\_fall\_mm\_per\_year','pesticides\_tonnes', 'avg\_temp', 'Area', 'Item', 'hg/ha\_yield']

df = df[col]

X = df.iloc[:, :-1]

y = df.iloc[:, -1]

This code selects specific columns from the DataFrame **df** and then prepares the data for a machine learning model:

1. **col = ['Year', 'average\_rain\_fall\_mm\_per\_year', 'pesticides\_tonnes', 'avg\_temp', 'Area', 'Item', 'hg/ha\_yield']**: This line creates a list **col** containing the names of columns to be selected from the DataFrame **df**.
2. **df = df[col]**: This selects only the columns specified in the list **col** from the DataFrame **df**, effectively filtering the DataFrame to contain only the columns of interest.
3. **X = df.iloc[:, :-1]**: This line selects all rows and all columns except the last one (which is assumed to be the target variable) from the filtered DataFrame **df** and assigns it to the variable **X**. This typically represents the feature matrix, containing the independent variables used for prediction.
4. **y = df.iloc[:, -1]**: This line selects all rows and only the last column from the filtered DataFrame **df** and assigns it to the variable **y**. This typically represents the target variable, the variable we want to predict in a supervised learning task.

Overall, **X** contains the feature matrix (input features) and **y** contains the target variable (output). These are the data structures typically used in supervised machine learning tasks.

df.head(3)

The **df.head(3)** function displays the first 3 rows of the DataFrame **df**. It's a quick way to inspect the structure and content of the DataFrame, providing a concise overview of its data.

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=0.8, random\_state=0, shuffle=True)

This code uses scikit-learn's **train\_test\_split** function to split the dataset into training and testing sets:

1. **from sklearn.model\_selection import train\_test\_split**: This line imports the **train\_test\_split** function from the **model\_selection** module of scikit-learn, which is a widely used library for machine learning in Python.
2. **X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=0.8, random\_state=0, shuffle=True)**: This line splits the feature matrix **X** and the target variable **y** into training and testing sets. Here's a breakdown of the parameters:
   * **X**: The feature matrix.
   * **y**: The target variable.
   * **train\_size=0.8**: Specifies that 80% of the data will be used for training.
   * **random\_state=0**: Sets the random seed for reproducibility. By setting this parameter, the same random splits will be generated each time the code is run.
   * **shuffle=True**: Specifies whether to shuffle the data before splitting. In this case, the data will be shuffled before splitting.

After execution, **X\_train** and **y\_train** will contain the training feature matrix and target variable, respectively, while **X\_test** and **y\_test** will contain the testing feature matrix and target variable, respectively.

**Converting categorical to numerical and scaling the values**

from sklearn.preprocessing import OneHotEncoder

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import StandardScaler

ohe = OneHotEncoder(drop='first')

scale = StandardScaler()

preprocesser = ColumnTransformer(

        transformers = [

            ('StandardScale', scale, [0, 1, 2, 3]),

            ('OHE', ohe, [4, 5]),

        ],

        remainder='passthrough'

)

This code snippet defines a preprocessing pipeline using scikit-learn's **ColumnTransformer**:

1. **One-Hot Encoding (OHE):** The **OneHotEncoder** from scikit-learn is used to convert categorical variables into binary vectors. **drop='first'** is set to drop the first level of each categorical variable to avoid multicollinearity issues.
2. **Standard Scaling:** The **StandardScaler** standardizes numerical features by removing the mean and scaling to unit variance. This helps to ensure that all numerical features have the same scale, which can be important for certain machine learning algorithms.
3. **ColumnTransformer:** This is used to apply different transformations to different columns of the dataset. It specifies which columns should undergo each transformation. In this case:
   * The first four columns (index 0 to 3) are standardized using **StandardScaler**.
   * The fifth and sixth columns (index 4 and 5) are one-hot encoded using **OneHotEncoder**.
   * The parameter **remainder='passthrough'** indicates that columns not specified should be passed through without any transformation.

Overall, this preprocessing pipeline standardizes numerical features and performs one-hot encoding on categorical features, preparing the data for machine learning models.

X\_train\_dummy = preprocesser.fit\_transform(X\_train)

X\_test\_dummy = preprocesser.transform(X\_test)

These lines of code transform the training and testing feature matrices using the defined preprocessing pipeline:

1. **Training Data Transformation (X\_train\_dummy):**
   * **preprocesser.fit\_transform(X\_train)**: This fits the **preprocesser** (ColumnTransformer) to the training data (**X\_train**) and then transforms it. It applies the specified transformations (standard scaling and one-hot encoding) to the training data.
2. **Testing Data Transformation (X\_test\_dummy):**
   * **preprocesser.transform(X\_test)**: This applies the same transformations learned from the training data to the testing data (**X\_test**). It uses the fitted **preprocesser** to transform the testing data, ensuring consistency with the preprocessing applied to the training data.

In short, these lines ensure that both the training and testing data undergo the same preprocessing steps, maintaining consistency between the datasets and preparing them for use in machine learning models.

preprocesser.get\_feature\_names\_out(col[:-1])

The **preprocesser.get\_feature\_names\_out(col[:-1])** line retrieves the names of the features after applying the preprocessing steps defined in the **preprocesser** ColumnTransformer.

* **col[:-1]**: This part specifies the list of column names (features) before preprocessing, excluding the last column, which is typically the target variable.
* **preprocesser.get\_feature\_names\_out(...)**: This method of the **preprocesser** object returns the names of the transformed features after preprocessing.

In short, this line gives the names of the features after they have been preprocessed, which can be useful for understanding the transformed dataset and interpreting the results of machine learning models.

Top of Form

**Model training**

#linear regression

from sklearn.linear\_model import LinearRegression,Lasso,Ridge

from sklearn.neighbors import KNeighborsRegressor

from sklearn.tree import DecisionTreeRegressor

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

models = {

    'lr':LinearRegression(),

    'lss':Lasso(),

    'Rid':Ridge(),

    'Dtr':DecisionTreeRegressor()

}

for name, md in models.items():

    md.fit(X\_train\_dummy,y\_train)

    y\_pred = md.predict(X\_test\_dummy)

    print(f"{name} : mae : {mean\_absolute\_error(y\_test,y\_pred)} score : {r2\_score(y\_test,y\_pred)}")

This code trains several regression models and evaluates their performance using Mean Absolute Error (MAE) and R-squared (R2) score:

1. **Model Initialization:**
   * The code initializes several regression models including Linear Regression, Lasso Regression, Ridge Regression, and Decision Tree Regression.
2. **Model Training and Evaluation:**
   * The code iterates over each model, fits it to the training data (**X\_train\_dummy**, **y\_train**), and predicts the target variable for the testing data (**X\_test\_dummy**).
   * For each model, it calculates and prints both Mean Absolute Error (MAE) and R-squared (R2) score to evaluate the model's performance on the testing data.
   * Mean Absolute Error (MAE) measures the average absolute difference between the predicted and actual values.
   * R-squared (R2) score measures the proportion of the variance in the dependent variable that is predictable from the independent variables.
3. **Output:**
   * For each model, the output displays its abbreviation (**name**), the Mean Absolute Error (MAE), and the R-squared (R2) score.

In short, this code trains multiple regression models, evaluates their performance using MAE and R2 score, and prints the results for comparison.

**Select Model**

dtr = DecisionTreeRegressor()

dtr.fit(X\_train\_dummy,y\_train)

dtr.predict(X\_test\_dummy)

This code trains a Decision Tree Regressor (**dtr**) model using the training data (**X\_train\_dummy**, **y\_train**) and then predicts the target variable for the testing data (**X\_test\_dummy**).

* **Model Training:**
  + **dtr = DecisionTreeRegressor()**: This line initializes a Decision Tree Regressor model.
  + **dtr.fit(X\_train\_dummy, y\_train)**: This line fits the Decision Tree Regressor model to the training data. The model learns from the features (**X\_train\_dummy**) and their corresponding target values (**y\_train**).
* **Prediction:**
  + **dtr.predict(X\_test\_dummy)**: This line predicts the target variable for the testing data (**X\_test\_dummy**) using the trained Decision Tree Regressor model (**dtr**). The predicted values are returned.

In short, this code trains a Decision Tree Regressor model and uses it to predict the target variable for the testing data.

**Predictive system**

def prediction(Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, Item):

    # Create an array of the input features

    features = np.array([[Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, Item]], dtype=object)

    # Transform the features using the preprocessor

    transformed\_features = preprocesser.transform(features)

    # Make the prediction

    predicted\_yield = dtr.predict(transformed\_features).reshape(1, -1)

    return predicted\_yield[0]

Year = 1990

average\_rain\_fall\_mm\_per\_year =1485.0

pesticides\_tonnes = 121.00

avg\_temp = 16.37

Area = 'Albania'

Item = 'Maize'

result = prediction(Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, Item)

This code defines a function **prediction** that takes input features, preprocesses them, and predicts the yield using the trained Decision Tree Regressor (**dtr**) model:

* **Function Explanation:**
  + **prediction(Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, Item)**: This function takes input features such as year, average rainfall, pesticides usage, average temperature, area, and item type.
  + **features = np.array([[Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, Item]], dtype=object)**: This line creates an array of the input features.
  + **transformed\_features = preprocesser.transform(features)**: This line transforms the features using the preprocessor (**preprocesser**) defined earlier.
  + **predicted\_yield = dtr.predict(transformed\_features).reshape(1, -1)**: This line makes the prediction using the trained Decision Tree Regressor (**dtr**) model after transforming the features. The predicted yield is reshaped into a one-dimensional array.
  + **return predicted\_yield[0]**: This line returns the predicted yield as the result of the function.
* **Input Features:**
  + **Year**: Year of observation.
  + **average\_rain\_fall\_mm\_per\_year**: Average rainfall in millimeters per year.
  + **pesticides\_tonnes**: Amount of pesticides used in tonnes.
  + **avg\_temp**: Average temperature.
  + **Area**: Area or location (e.g., country).
  + **Item**: Type of crop or item.
* **Output:**
  + The **result** variable will contain the predicted yield based on the input features provided to the function.

In short, this code defines a function to predict crop yield based on input features using a pre-trained Decision Tree Regressor model and a preprocessing pipeline.

result

The **result** variable contains the predicted yield of maize in Albania for the year 1990, based on the provided input features such as average rainfall, pesticides usage, and average temperature. The prediction is made using a pre-trained Decision Tree Regressor model and a preprocessing pipeline.

entry1 = (1990, 1485.0, 121.00, 16.37, "Albania", "Maize", 36613)

entry2 = (2013, 657.0, 2550.07, 19.76, "Zimbabwe", "Sorghum", 3066)

hese are two data entries represented as tuples:

1. **entry1:**
   * Year: 1990
   * Average Rainfall (mm/year): 1485.0
   * Pesticides Usage (tonnes): 121.00
   * Average Temperature (°C): 16.37
   * Area: Albania
   * Item: Maize
   * hg/ha\_yield: 36613 (assumed to be yield per hectare)
   * This entry provides information about maize cultivation in Albania for the year 1990, including environmental factors like rainfall and temperature, as well as agricultural inputs like pesticides usage.
2. **entry2:**
   * Year: 2013
   * Average Rainfall (mm/year): 657.0
   * Pesticides Usage (tonnes): 2550.07
   * Average Temperature (°C): 19.76
   * Area: Zimbabwe
   * Item: Sorghum
   * hg/ha\_yield: 3066 (assumed to be yield per hectare)
   * This entry provides information about sorghum cultivation in Zimbabwe for the year 2013, including environmental factors like rainfall and temperature, as well as agricultural inputs like pesticides usage.

In short, each entry represents data points for agricultural production in specific locations (Albania and Zimbabwe) for different crops (Maize and Sorghum) during different years (1990 and 2013), including relevant environmental and agricultural input factors.

**Pickle files**

import pickle

pickle.dump(dtr,open('dtr.pkl','wb'))

pickle.dump(preprocesser,open('preprocessor.pkl','wb'))

These lines of code use the **pickle** module in Python to serialize and save the trained Decision Tree Regressor model (**dtr**) and the preprocessing pipeline (**preprocesser**) to files:

1. **Saving the Decision Tree Regressor Model:**
   * **pickle.dump(dtr, open('dtr.pkl', 'wb'))**: This line serializes and saves the trained Decision Tree Regressor model (**dtr**) to a file named **'dtr.pkl'** in binary mode (**'wb'**).
2. **Saving the Preprocessing Pipeline:**
   * **pickle.dump(preprocesser, open('preprocessor.pkl', 'wb'))**: This line serializes and saves the preprocessing pipeline (**preprocesser**) to a file named **'preprocessor.pkl'** in binary mode (**'wb'**).

In short, these lines store the trained regression model and the preprocessing steps as files on disk using the pickle serialization method. This allows you to later reload and reuse them without needing to retrain the model or redefine the preprocessing steps.

<!doctype html>

<html lang="en">

  <head>

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>Bootstrap demo</title>

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-9ndCyUaIbzAi2FUVXJi0CjmCapSmO7SnpJef0486qhLnuZ2cdeRhO02iuK6FUUVM" crossorigin="anonymous">

  </head>

  <body>

    <h1 class="text-center text-success">Crop Yield Prediction Per Country</h1>

<!--  Year  average\_rain\_fall\_mm\_per\_year pesticides\_tonnes avg\_temp  Area  Item-->

  <div class="container my-4 mt-4" style="background-color: rgba(0, 0, 0, 0.5); border-radius: 20px; color:white">

    <h1 class="text-center text-danger">Input All Features Here</h1>

    <form action="/predict" method="post">

    <div class="form-group">

        <label for="Year">Year</label>

        <input type="number" class="form-control" name="Year" step="any" value="2013">

    </div>

    <div class="form-group">

        <label for="average\_rain\_fall\_mm\_per\_year">average\_rain\_fall\_mm\_per\_year</label>

        <input type="number" class="form-control" name="average\_rain\_fall\_mm\_per\_year" step="any" >

    </div>

    <div class="form-group">

        <label for="pesticides\_tonnes">pesticides\_tonnes</label>

        <input type="number" class="form-control" name="pesticides\_tonnes" step="any" >

    </div>

    <div class="form-group">

        <label for="avg\_temp">avg\_temp</label>

        <input type="number" class="form-control" name="avg\_temp" step="any">

    </div>

    <div class="form-group">

        <label for="Area">Area</label>

        <input type="text" class="form-control" name="Area" >

    </div>

    <div class="form-group">

        <label for="Item">Item</label>

        <input type="text" class="form-control" name="Item" >

    </div>

    <button type="submit" class="btn btn-danger btn-lg mt-2 btn-block">Predict</button>

</form>

      {% if prediction %}

      <h1 class="text-center"> Predicted Yield: <br>{{prediction}}</h1>

      {% endif %}

</div>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js" integrity="sha384-geWF76RCwLtnZ8qwWowPQNguL3RmwHVBC9FhGdlKrxdiJJigb/j/68SIy3Te4Bkz" crossorigin="anonymous"></script>

  </body>

</html>

1. The document structure is defined within **<!doctype html>** and **<html lang="en">**.
2. In the **<head>** section, metadata like character set and viewport settings are defined. Bootstrap CSS is included via a CDN link.
3. Within the **<body>** section:
   * There's a title "Crop Yield Prediction Per Country" in a centered heading (**<h1>**).
   * A form is provided for inputting features required for prediction. It has fields for Year, average\_rain\_fall\_mm\_per\_year, pesticides\_tonnes, avg\_temp, Area, and Item.
   * Each input field is wrapped in a **<div>** with the class **form-group**.
   * The form action is set to "/predict" using the POST method.
   * A submit button with the label "Predict" is provided.
   * A conditional statement (**{% if prediction %}**) is used to display the prediction result if available.
4. The Bootstrap JavaScript bundle is included at the end of the body for enhanced functionality.

Overall, this code sets up a webpage for predicting crop yield per country, allowing users to input relevant features and displaying the predicted yield upon submission.

from flask import Flask,request, render\_template

import numpy as np

import pickle

import sklearn

print(sklearn.\_\_version\_\_)

#loading models

dtr = pickle.load(open('dtr.pkl','rb'))

preprocessor = pickle.load(open('preprocessor.pkl','rb'))

#flask app

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

    return render\_template('index.html')

@app.route("/predict",methods=['POST'])

def predict():

    if request.method == 'POST':

        Year = request.form['Year']

        average\_rain\_fall\_mm\_per\_year = request.form['average\_rain\_fall\_mm\_per\_year']

        pesticides\_tonnes = request.form['pesticides\_tonnes']

        avg\_temp = request.form['avg\_temp']

        Area = request.form['Area']

        Item  = request.form['Item']

        features = np.array([[Year,average\_rain\_fall\_mm\_per\_year,pesticides\_tonnes,avg\_temp,Area,Item]],dtype=object)

        transformed\_features = preprocessor.transform(features)

        prediction = dtr.predict(transformed\_features).reshape(1,-1)

        return render\_template('index.html',prediction = prediction)

if \_\_name\_\_=="\_\_main\_\_":

    app.run(debug=True)

This is a Flask web application for predicting crop yield. Here's a brief explanation:

1. The application imports necessary libraries: Flask, NumPy, pickle (for model loading), and sklearn for model version checking.
2. It loads the trained decision tree regression model (**dtr.pkl**) and the preprocessor (**preprocessor.pkl**) using pickle.
3. A Flask application is initialized.
4. The root route ('/') renders the index HTML template, which contains the form for inputting features.
5. The '/predict' route handles the prediction logic:
   * It retrieves the input data from the form submitted via POST request.
   * The input features are transformed using the preprocessor.
   * The transformed features are used to make a prediction using the decision tree regression model.
   * The prediction is then passed back to the index template for display.
6. If the script is executed directly (**\_\_name\_\_=="\_\_main\_\_"**), the Flask application runs in debug mode.

Overall, this script sets up a web server that accepts input features related to crop yield prediction, processes them using a pre-trained model, and returns the prediction to the user.

1. **dtr.pkl**: This file contains a trained Decision Tree Regression (DTR) model. The DTR model is a machine learning algorithm used for regression tasks, such as predicting numerical values based on input features. In this case, the DTR model has been trained on historical data related to crop yields, considering factors like year, rainfall, pesticide usage, temperature, area, and type of crop. When a new set of input features is provided, the DTR model utilizes its learned patterns to predict the crop yield for those conditions.
2. **preprocessor.pkl**: This file contains a preprocessor object. Preprocessing is a crucial step in machine learning pipelines, especially when dealing with real-world data that might be messy or inconsistent. The preprocessor used here likely includes steps such as scaling numerical features, encoding categorical variables, handling missing values, and possibly feature engineering. Before making predictions with the DTR model, the input features need to undergo the same preprocessing steps that were applied to the training data. This ensures that the input data is in the same format and scale as the data used to train the model, allowing for accurate predictions.

In summary, **dtr.pkl** holds the trained regression model responsible for predicting crop yields, while **preprocessor.pkl** contains the necessary preprocessing steps to transform input data into a suitable format for prediction by the model. Together, they enable the Flask application to accept user input, preprocess it, make predictions, and display the predicted crop yield to the user.

Top of Form

Top of Form

Top of Form