Code Explanation (Important Functions/Sections)

Let's break down some of the key parts of your Python code:

1. **Data Loading and Initial Inspection:**

library\_add

content\_copy

df=pd.read\_csv('/content/Crop\_production.csv')  
    df.head()  
    df.info()

-   `pd.read\_csv()`: This function from the pandas library is used to load your dataset from a CSV file into a DataFrame.  
-   `df.head()`: Displays the first few rows of the DataFrame, giving you a quick look at the data structure.  
-   `df.info()`: Provides a concise summary of the DataFrame, including the index dtype and column dtypes, non-null values, and memory usage. This is crucial for checking for missing data and understanding data types.

1. **Handling Missing Values:**

library\_add

content\_copy

df.dropna(inplace=True)

-   `df.dropna()`: This function is used to remove rows or columns with missing values (NaN).  
-   `inplace=True`: This argument modifies the DataFrame `df` directly without needing to reassign the result. Using `inplace=True` is convenient but can sometimes make code harder to read or debug compared to `df = df.dropna()`.

1. **Categorical Feature Encoding:**

library\_add

content\_copy

from sklearn.preprocessing import OrdinalEncoder  
  
    encoder = OrdinalEncoder()  
  
    df['Crop\_Type\_encoded'] = encoder.fit\_transform(df[['Crop\_Type']])  
    df['State\_Name\_encoded'] = encoder.fit\_transform(df[['State\_Name']])  
    # Note: You later try to use 'Crop\_encoded\_label' and 'State\_Name\_encoded\_label'  
    # and 'Crop\_Type\_encoded\_label' in the ipywidgets section.  
    # You only encoded 'Crop\_Type' and 'State\_Name' with OrdinalEncoder and  
    # named the new columns 'Crop\_Type\_encoded' and 'State\_Name\_encoded'.  
    # You would need to apply encoding to the 'Crop' column as well and name  
    # the columns consistently, perhaps using LabelEncoder as the variable names suggest,  
    # although OrdinalEncoder also assigns integer labels.

-   `OrdinalEncoder`: This scikit-learn class is used to convert categorical features into numerical ones by assigning an integer to each unique category. It's suitable when the categorical variable has a natural order (though in this case, State Name, Crop Type, and Crop might not have a natural ordinal relationship, One-Hot Encoding might be another option depending on the model).  
-   `encoder.fit\_transform(df[['column\_name']])`: `fit()` learns the unique categories in the specified column, and `transform()` applies the mapping to create the new encoded column. The `[['column\_name']]` is necessary because `fit\_transform` expects a 2D array-like input.

1. **Dropping Original and Unnecessary Columns:**

library\_add

content\_copy

df = df.drop('Unnamed: 0', axis=1)  
    df = df.drop('State\_Name', axis=1)  
    df = df.drop('Crop\_Type', axis=1)  
    df = df.drop('Crop', axis=1)  
    df1 = df.copy()

-   `df.drop('column\_name', axis=1)`: This function removes the specified column (`axis=1`). You are dropping the original categorical columns after creating the encoded ones. You also drop 'Unnamed: 0', which is often an index column loaded from the CSV.  
-   `df1 = df.copy()`: Creates a shallow copy of the DataFrame `df` into a new variable `df1`.

1. **Data Type Conversion:**

library\_add

content\_copy

columns\_to\_convert = ['N', 'P', 'K']  
    for col in columns\_to\_convert:  
        if col in df.columns and df[col].dtype == np.int64:  
            df[col] = df[col].astype(np.int32)  
  
    columns\_to\_convert = ['pH', 'rainfall', 'temperature', 'Production\_in\_tons', 'Yield\_ton\_per\_hec','Area\_in\_hectares']  
    for col in columns\_to\_convert:  
        if col in df.columns and df[col].dtype == np.float64:  
            df[col] = df[col].astype(np.int32)  
    # ... similar conversion for encoded columns ...

-   `.astype(dtype)`: Converts the data type of a pandas Series (a DataFrame column) to the specified dtype (e.g., `np.int32`). This is done to potentially save memory or ensure compatibility with certain models. \*\*Caution:\*\* Converting float columns with decimal values to integers (`astype(np.int32)`) will truncate the decimal part, which might lead to loss of precision. For features like `pH`, `rainfall`, `temperature`, `Production\_in\_tons`, `Yield\_ton\_per\_hec`, and `Area\_in\_hectares`, it's generally better to keep them as float types (`float32` or `float64`) unless there's a specific reason to discard decimal precision.

1. **Splitting Data into Features and Target:**

library\_add

content\_copy

X = df.drop('Yield\_ton\_per\_hec', axis=1)  # Features  
    y = df['Yield\_ton\_per\_hec']  # Target variable

-   Separates the DataFrame into `X` (the features used for prediction, all columns except the target) and `y` (the target variable you want to predict).

1. **Splitting Data into Training and Testing Sets:**

library\_add

content\_copy

from sklearn.model\_selection import train\_test\_split  
    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

-   `train\_test\_split()`: This essential scikit-learn function splits your data into four subsets: training features (`X\_train`), testing features (`X\_test`), training target (`y\_train`), and testing target (`y\_test`).  
-   `test\_size=0.2`: Specifies that 20% of the data should be allocated to the testing set, and 80% to the training set.  
-   `random\_state=42`: Ensures that the splitting is the same every time you run the code, making your results reproducible.

1. **Handling Infinite Values and NaN in Training Data:**

library\_add

content\_copy

y\_train = y\_train.replace([np.inf, -np.inf], np.nan)  
    y\_train = y\_train.fillna(y\_train.mean()) # Impute NaN in target  
  
    # Inside the loop for models:  
    X\_train = X\_train.replace([np.inf, -np.inf], np.nan)  
    X\_train = X\_train.fillna(X\_train.mean()) # Impute NaN in features

-   `.replace([np.inf, -np.inf], np.nan)`: Replaces infinite values (positive and negative) with NaN. Infinity values can occur in data, especially after certain calculations, and many models cannot handle them.  
-   `.fillna(y\_train.mean())`: Fills any remaining NaN values in the training target variable with the mean of the non-NaN values. This is a simple imputation strategy. You also apply similar imputation to the features `X\_train` within the model training loop.

1. **Model Training and Evaluation Loop:**

library\_add

content\_copy

models = [RandomForestRegressor(), LinearRegression(), ElasticNet(), KNeighborsRegressor(), xgb.XGBRegressor(), Ridge()]  
    # scores = dict() # This dictionary isn't used later  
  
    for model in models:  
        # ... (imputation code) ...  
        model.fit(X\_train, y\_train) # Train the model  
        y\_pred = model.predict(X\_test) # Make predictions on the test set  
  
        print(f'model: {str(model)}')  
        print(f'RMSE: {mean\_squared\_error(y\_test, y\_pred)}') # Calculate and print RMSE  
        print(f'MAE: {mean\_absolute\_error(y\_test, y\_pred)}') # Calculate and print MAE  
        print('-' \* 30, '\n')

-   This loop iterates through a list of different regression models.  
-   `model.fit(X\_train, y\_train)`: This is where the model learns the relationship between the training features (`X\_train`) and the training target (`y\_train`).  
-   `model.predict(X\_test)`: After training, the `predict` method is used to make predictions on the unseen test features (`X\_test`).  
-   `mean\_squared\_error(y\_test, y\_pred)`: Calculates the Mean Squared Error, a common regression metric. The square root of this (RMSE) is often preferred as it's in the same units as the target variable.  
-   `mean\_absolute\_error(y\_test, y\_pred)`: Calculates the Mean Absolute Error, another robust regression metric that represents the average absolute difference between actual and predicted values.

1. **Interactive Prediction Interface:**

library\_add

content\_copy

import ipywidgets as widgets  
    from IPython.display import display  
  
    # ... Define input widgets ...  
  
    def predict\_on\_input(b):  
        # ... Create input\_data DataFrame ...  
        # ... Ensure column order matches X\_train.columns ...  
        predictions = model.predict(input\_data) # Make prediction using the LAST trained model  
        print("Predicted Value:", predictions[0])  
  
    predict\_button = widgets.Button(description="Predict")  
    predict\_button.on\_click(predict\_on\_input)  
  
    display(...) # Display widgets [on](https://g.co/legal/generative-code)

-   `ipywidgets`: A library that allows you to create interactive user interface controls (like text boxes, buttons, sliders) directly in a Jupyter Notebook.  
-   `widgets.IntText`, `widgets.FloatText`, `widgets.Button`: Examples of different widget types.

-   `predict\_on\_input(b)`: This function is triggered when the `predict\_button` is clicked. It reads values from the input widgets, constructs a pandas DataFrame (`input\_data`) in the correct column order, and uses the `predict` method of the `model` variable to make a prediction.  
-   `model.predict(input\_data)`: \*\*Important:\*\* The `model` variable used here will be the \*last\* model trained in your loop (which is `Ridge()` in your current code, but you have a separate cell where you re-initialize `model = xgb.XGBRegressor()`). Ensure `model` is set to the specific model you want to use for the interactive prediction.  
-   `predict\_button.on\_click(predict\_on\_input)`: Links the `predict\_on\_input` function to the button click event.  
-   `display()`: Renders the specified widgets in the notebook output.