Impacts of climate change on food Security

Group No: 17

1. Overview

The data preparation and feature engineering phase is crucial for any machine learning project as it ensures that the dataset is in a usable form and that the models can learn effectively from the data. This phase involves collecting, cleaning, and transforming the raw data into a format that the model can use, while feature engineering helps improve the model's predictive performance by creating meaningful input features.

2. Data Collection

For this project, the dataset was collected from multiple reliable sources:

- **Source:** Search for a Dataset - Humanitarian Data Exchange (humdata.org).
- For this first site, we downloaded climate change data for each country from the World Bank's data portal, accessible through Humanitarian Data Exchange (HDX). This dataset covers crucial indicators such as climate systems, exposure to climate impacts, resilience, greenhouse gas emissions, and energy use. The data highlights how climate change is disproportionately affecting developing countries, leading to increased risks for agriculture, food, and water supplies. The dataset is valuable for understanding global cooperation needs in addressing climate change, with additional indicators found under Environment, Agriculture & Rural Development, Energy & Mining, Health, and Urban Development.
- **Relevance:** Crucial for understanding climate impacts on agriculture

2. **Agricultural Data**

   - **Source:** World Bank
   - From the second source, the World Bank, we specifically extracted data on pesticides, temperature, rainfall, and crop yield, which are essential for agricultural analysis.

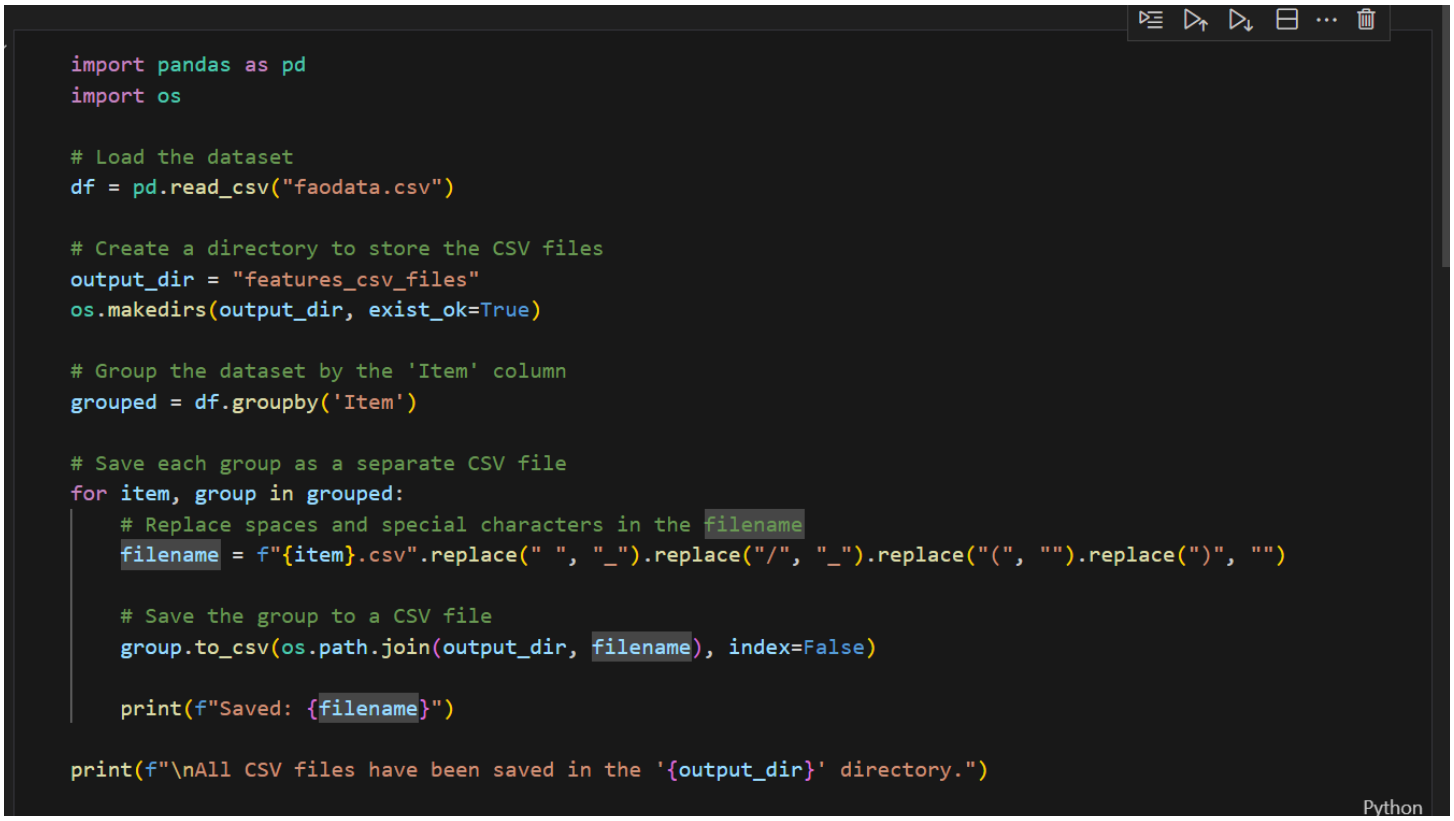
- **Relevance:** Key for predicting crop productivity

3. **Food Security Data**
    - **Source:** FAOSTAT
    - Finally, FAOSTAT provided information on food availability and malnutrition rates, directly connected to food security concerns.
    - **Relevance:** Directly related to food security outcomes

Preprocessing during the data collection phase included cleaning, normalizing, and aggregating the data from different sources. Datasets were downloaded in CSV format and prepared for feature engineering.

3. Data Cleaning

- Organizing and Preparing the Dataset for Analysis



```python
import pandas as pd
import os

# Load the dataset
df = pd.read_csv("faodata.csv")

# Create a directory to store the CSV files
output_dir = "features_csv_files"
os.makedirs(output_dir, exist_ok=True)

# Group the dataset by the 'Item' column
grouped = df.groupby('Item')

# Save each group as a separate CSV file
for item, group in grouped:
    # Replace spaces and special characters in the filename
    filename = f"{item}.csv".replace(" ", "_").replace("/", "_").replace("(", "").replace(")", "")

    # Save the group to a CSV file
    group.to_csv(os.path.join(output_dir, filename), index=False)

    print(f"Saved: {filename}")

print(f"\nAll CSV files have been saved in the '{output_dir}' directory.")
```

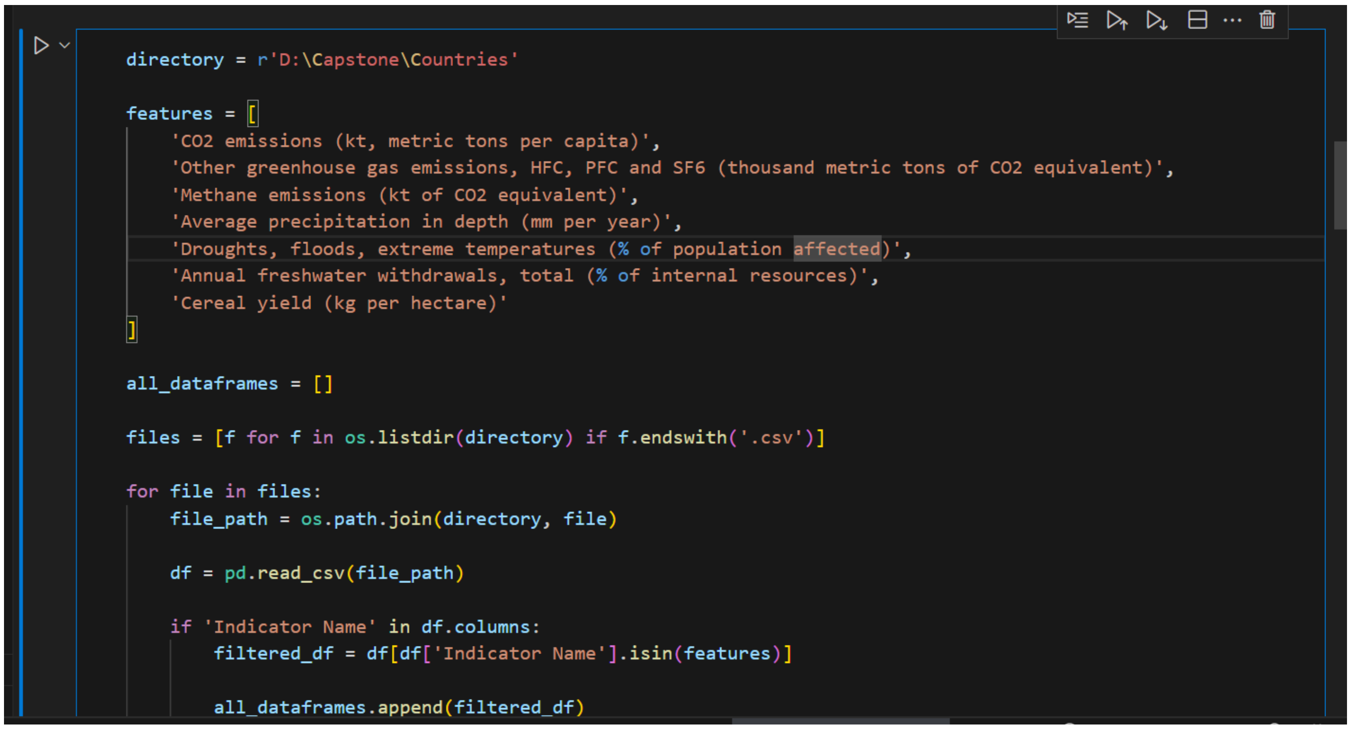
The data cleaning process outlined in this code involves several essential steps to organize and prepare a dataset for analysis, ensuring the data is well-structured and easy to process. Below, the steps are detailed to explain the rationale behind the cleaning operations applied to the dataset.

1. **Loading the Dataset**: The dataset (faodata.csv) was loaded using pandas' pd.read_csv(), converting it into a DataFrame for efficient manipulation and analysis.
2. **Creating a Directory for CSV Files**: A directory (features_csv_files) was created using os.makedirs() to save the grouped CSV files.
3. **Grouping by 'Item' Column**: The dataset was grouped by the Item column, which contains various food security indicators like CO2 emissions and cereal yield. Grouping allows for focused analysis of each feature.
4. **Saving Each Group as a CSV File**: Each group from the Item column was saved as a separate CSV file. Filenames were cleaned by replacing special characters, and a confirmation message was printed after saving each file.
5. **Removing the Second Line from Each CSV**: An additional cleaning step removed the second line from each CSV file, ensuring that irrelevant data such as duplicate headers or metadata were dropped.

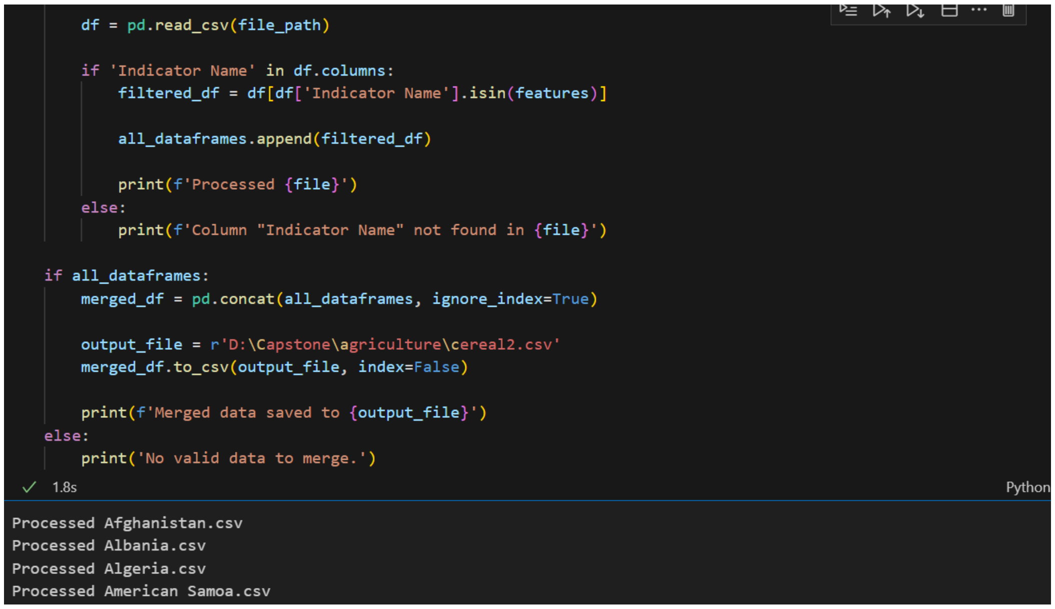
Purpose and Benefits of the Cleaning Process

The primary goal of this data cleaning procedure is to isolate different features (Items) from the dataset for individual analysis. By separating indicators such as CO2 emissions, methane emissions, and cereal yield, it becomes easier to focus on the specific trends and behaviors within each feature. This results in a well-structured dataset, ready for deeper analysis and modeling.

- **Data Cleaning and Transformation Process**



```
directory = r'D:\Capstone\Countries'

features = [
    'CO2 emissions (kt, metric tons per capita)',
    'Other greenhouse gas emissions, HFC, PFC and SF6 (thousand metric tons of CO2 equivalent)',
    'Methane emissions (kt of CO2 equivalent)',
    'Average precipitation in depth (mm per year)',
    'Droughts, floods, extreme temperatures (% of population affected)',
    'Annual freshwater withdrawals, total (% of internal resources)',
    'Cereal yield (kg per hectare)'
]

all_dataframes = []

files = [f for f in os.listdir(directory) if f.endswith('.csv')]

for file in files:
    file_path = os.path.join(directory, file)

    df = pd.read_csv(file_path)

    if 'Indicator Name' in df.columns:
        filtered_df = df[df['Indicator Name'].isin(features)]

        all_dataframes.append(filtered_df)
```



```
        df = pd.read_csv(file_path)

        if 'Indicator Name' in df.columns:
            filtered_df = df[df['Indicator Name'].isin(features)]

            all_dataframes.append(filtered_df)

            print(f'Processed {file}')
        else:
            print(f'Column "Indicator Name" not found in {file}')

    if all_dataframes:
        merged_df = pd.concat(all_dataframes, ignore_index=True)

        output_file = r'D:\Capstone\agriculture\cereal2.csv'
        merged_df.to_csv(output_file, index=False)

        print(f'Merged data saved to {output_file}')
    else:
        print('No valid data to merge.')
```

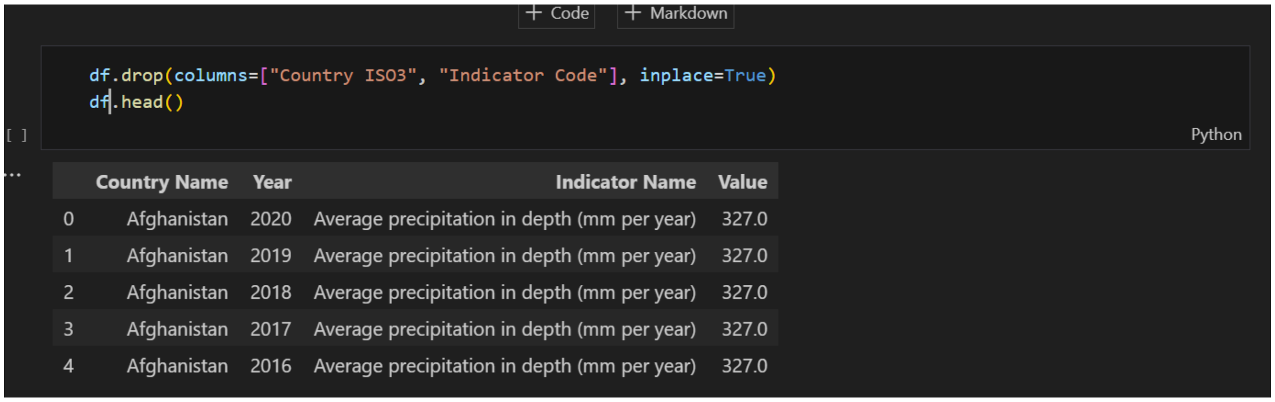
✓ 1.8s

Python

```
Processed Afghanistan.csv
Processed Albania.csv
Processed Algeria.csv
Processed American Samoa.csv
```

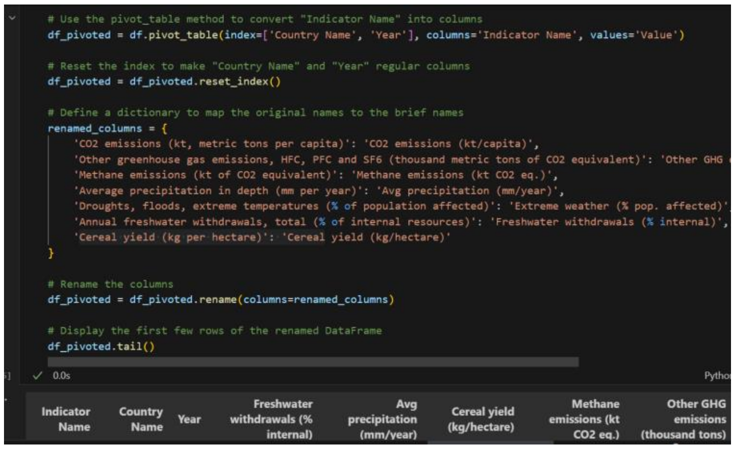
1. **Setting Directory and Defining Features**:
   - The script starts by defining a directory path (D:\Capstone\Countries) where the dataset files are stored.
   - A list of relevant features (indicators) is also provided, which include metrics like CO2 emissions, methane emissions, cereal yield, and more. These features are later used to filter the dataset.
2. **Reading and Filtering Data**:
   - **Files Listing**: All CSV files in the specified directory are listed and iterated over. Only files with the .csv extension are processed.
   - **Reading CSV Files**: For each CSV file, pd.read_csv() is used to load the data into a DataFrame.
   - **Filtering by Indicator**: The script checks if the file contains the Indicator Name column. If present, the DataFrame is filtered to retain only rows where Indicator Name matches one of the specified features. These filtered DataFrames are collected in a list (all_dataframes).
3. **Concatenating Data**:

- Once all the files are processed, the script checks if any filtered DataFrames were collected. If so, the individual DataFrames are concatenated using pd.concat(), combining all the filtered data into a single DataFrame (merged_df).
- **Saving the Merged Data**: The combined dataset is saved as a new CSV file (cereal2.csv), allowing the processed data to be stored for future use.



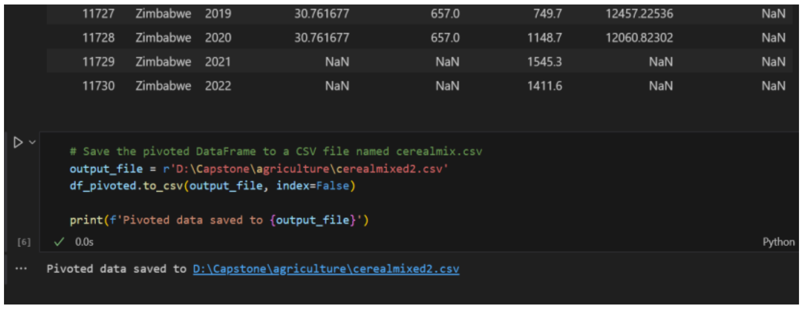
```python
df.drop(columns=["Country ISO3", "Indicator Code"], inplace=True)
df.head()
```

	Country Name	Year	Indicator Name	Value
0	Afghanistan	2020	Average precipitation in depth (mm per year)	327.0
1	Afghanistan	2019	Average precipitation in depth (mm per year)	327.0
2	Afghanistan	2018	Average precipitation in depth (mm per year)	327.0
3	Afghanistan	2017	Average precipitation in depth (mm per year)	327.0
4	Afghanistan	2016	Average precipitation in depth (mm per year)	327.0



```python
# Use the pivot_table method to convert "Indicator Name" into columns
df_pivoted = df.pivot_table(index=['Country Name', 'Year'], columns='Indicator Name', values='Value')

# Reset the index to make "Country Name" and "Year" regular columns
df_pivoted = df_pivoted.reset_index()

# Define a dictionary to map the original names to the brief names
renamed_columns = {
    'CO2 emissions (kt, metric tons per capita)': 'CO2 emissions (kt/capita)',
    'Other greenhouse gas emissions, HFC, PFC and SF6 (thousand metric tons of CO2 equivalent)': 'Other GHG
    'Methane emissions (kt of CO2 equivalent)': 'Methane emissions (kt CO2 eq.)',
    'Average precipitation in depth (mm per year)': 'Avg precipitation (mm/year)',
    'Droughts, floods, extreme temperatures (% of population affected)': 'Extreme weather (% pop. affected)'
    'Annual freshwater withdrawals, total (% of internal resources)': 'Freshwater withdrawals (% internal)',
    'Cereal yield (kg per hectare)': 'Cereal yield (kg/hectare)'
}

# Rename the columns
df_pivoted = df_pivoted.rename(columns=renamed_columns)

# Display the first few rows of the renamed DataFrame
df_pivoted.tail()
```

Indicator Name	Country Name	Year	Freshwater withdrawals (% internal)	Avg precipitation (mm/year)	Cereal yield (kg/hectare)	Methane emissions (kt CO2 eq.)	Other GHG emissions (thousand tons)



11727	Zimbabwe	2019	30.761677	657.0	749.7	12457.22536	NaN
11728	Zimbabwe	2020	30.761677	657.0	1148.7	12060.82302	NaN
11729	Zimbabwe	2021	NaN	NaN	1545.3	NaN	NaN
11730	Zimbabwe	2022	NaN	NaN	1411.6	NaN	NaN

```python
# Save the pivoted DataFrame to a CSV file named cerealmix.csv
output_file = r'D:\Capstone\agriculture\cerealmixed2.csv'
df_pivoted.to_csv(output_file, index=False)

print(f'Pivoted data saved to {output_file}')
```

```
Pivoted data saved to D:\Capstone\agriculture\cerealmixed2.csv
```

1. **Dropping Unnecessary Columns**:
   - After loading the saved cereal2.csv file, the script removes irrelevant columns (Country ISO3 and Indicator Code) using df.drop(). These columns do not contribute to the analysis and are therefore removed to streamline the data.
2. **Pivoting Data**:
   - **Reshaping the Data**: The DataFrame is transformed using pivot_table() to convert the Indicator Name values into separate columns. This reshapes the data such that each indicator becomes a column, and the dataset is now organized by Country Name and Year.

- **Resetting Index**: After pivoting, Country Name and Year are converted back to regular columns using reset_index() to flatten the DataFrame structure.

3. **Renaming Columns for Clarity**:
    - A dictionary (renamed_columns) is defined to map the original, verbose column names (like *CO2 emissions (kt, metric tons per capita)*) to shorter, more concise versions (like *CO2 emissions (kt/capita)*).
    - The DataFrame columns are then renamed using rename() for better readability and clarity in further analysis.
4. **Saving the Final Cleaned and Pivoted Data**:
    - The cleaned, reshaped, and renamed DataFrame is saved to a new CSV file (cerealmixed2.csv), providing a ready-to-analyze dataset with clear column names and a structured format.

Purpose and Benefits of the Process:

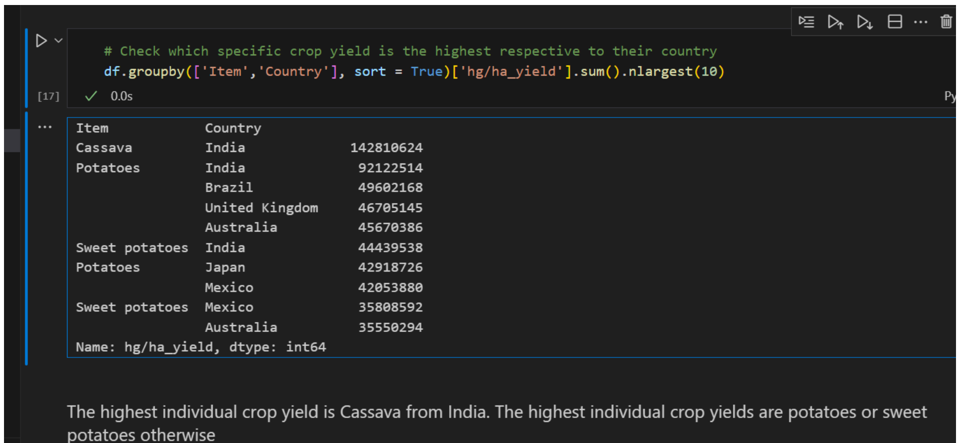
- **Filtering and Grouping**: By focusing on specific features, the process ensures that only relevant data is retained, improving the quality of analysis.
- **Pivoting**: Reshaping the data by pivoting makes it easier to compare the indicators across countries and years.
- **Column Renaming**: Renaming columns enhances readability and makes the dataset easier to understand and work with.

4. Exploratory Data Analysis (EDA)

- For Crop Yield Prediction (For example)
  The Exploratory Data Analysis (EDA) focused on understanding crop yields, pesticide use, and the influence of environmental factors like rainfall and temperature across different countries.

  Key Insights:

1. **Country & Crop Distribution:**



```
# Check which specific crop yield is the highest respective to their country
df.groupby(['Item','Country'], sort = True)['hg/ha_yield'].sum().nlargest(10)
```

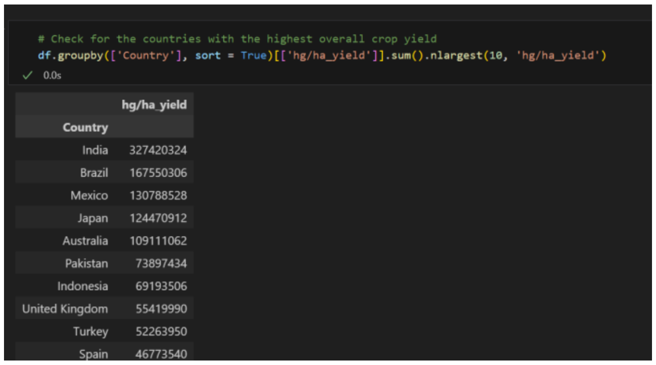
[17] ✓ 0.0s

```
Item            Country
Cassava         India             142810624
Potatoes        India              92122514
                Brazil             49602168
                United Kingdom     46705145
                Australia          45670386
Sweet potatoes  India              44439538
Potatoes        Japan              42918726
                Mexico             42053880
Sweet potatoes  Mexico             35808592
                Australia          35550294
Name: hg/ha_yield, dtype: int64
```

The highest individual crop yield is Cassava from India. The highest individual crop yields are potatoes or sweet potatoes otherwise

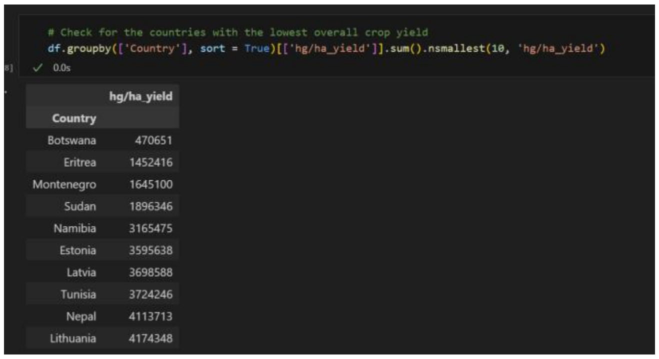
- India is the most frequent country, and potatoes are the most frequent crop in the dataset.
- Cassava has the highest individual crop yield in India, followed by potatoes and sweet potatoes in other countries.

2. **Top and Lowest Yielding Countries:**



```
# Check for the countries with the highest overall crop yield
df.groupby(['Country'], sort = True)[['hg/ha_yield']].sum().nlargest(10, 'hg/ha_yield')
✓ 0.0s
```

Country	hg/ha_yield
India	327420324
Brazil	167550306
Mexico	130788528
Japan	124470912
Australia	109111062
Pakistan	73897434
Indonesia	69193506
United Kingdom	55419990
Turkey	52263950
Spain	46773540

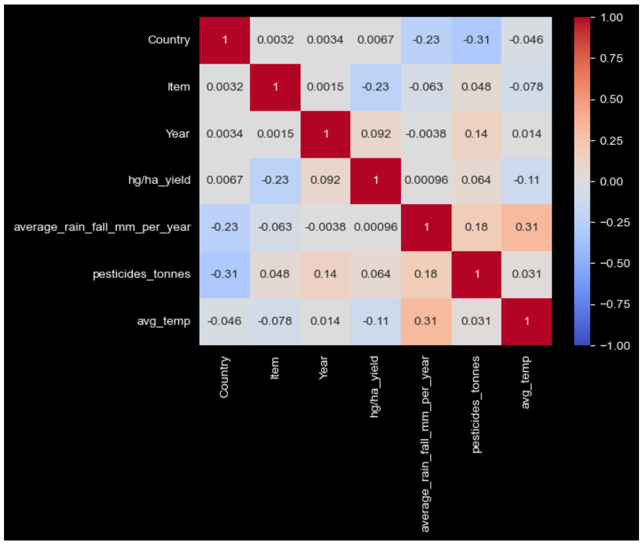


```
# Check for the countries with the lowest overall crop yield
df.groupby(['Country'], sort = True)[['hg/ha_yield']].sum().nsmallest(10, 'hg/ha_yield')
✓ 0.0s
```

Country	hg/ha_yield
Botswana	470651
Eritrea	1452416
Montenegro	1645100
Sudan	1896346
Namibia	3165475
Estonia	3595638
Latvia	3698588
Tunisia	3724246
Nepal	4113713
Lithuania	4174348

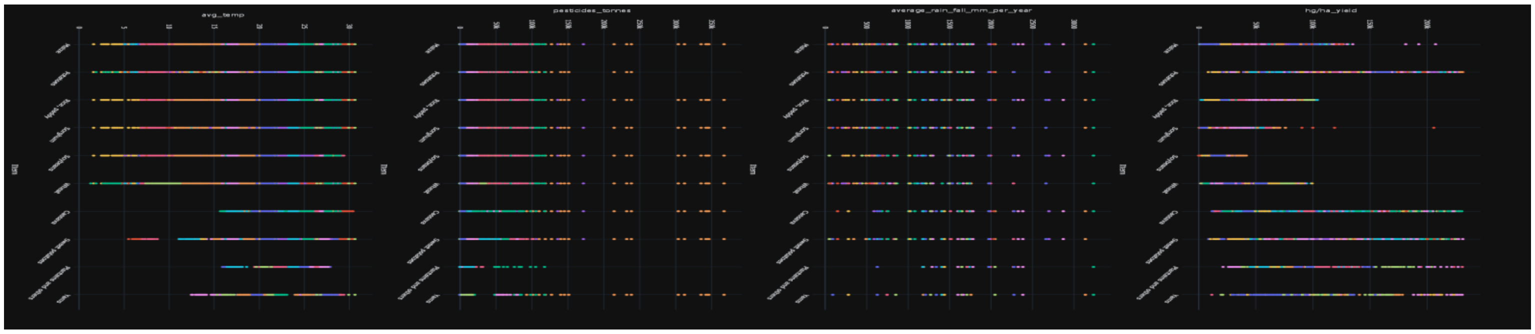
- Countries with the highest overall crop yields include India, Brazil, and Mexico, while the lowest include regions like Botswana, Eritrea and Montenegro.

3. **Correlations:**



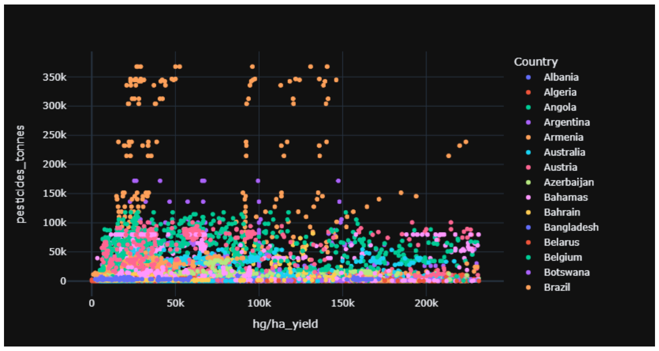
- The strongest correlation observed is between country and pesticide use, possibly due to different regulatory practices and pest control needs in various regions.
- Crop yield is minimally affected by pesticide use, with Brazil being an outlier with high pesticide usage and yield.

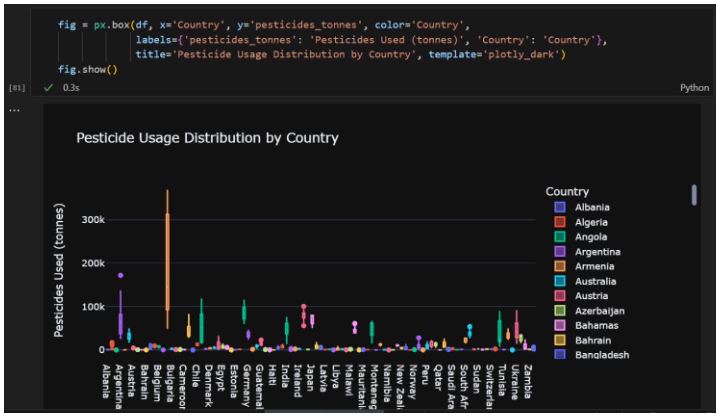
4. **Environmental Factors:**



- Temperature and rainfall show weak correlations with yield, except for crops like maize and rice that thrive in diverse climates.

5. **Pesticide Usage:**

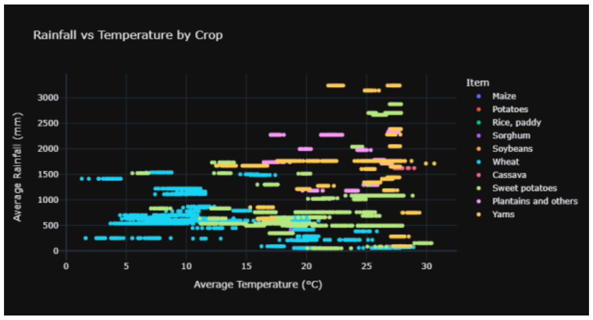


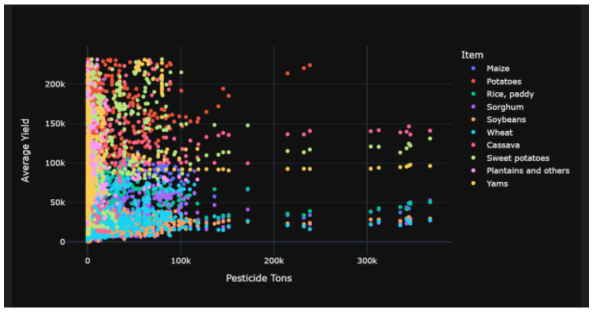


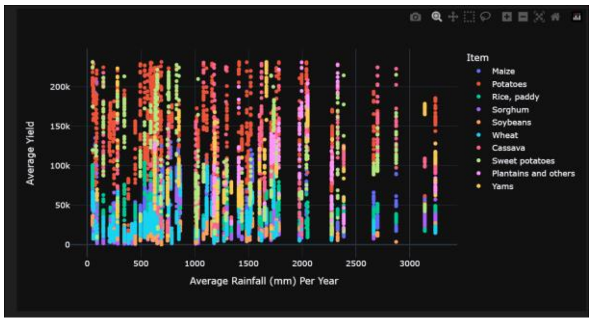
```
fig = px.box(df, x='Country', y='pesticides_tonnes', color='Country',
             labels={'pesticides_tonnes': 'Pesticides Used (tonnes)', 'Country': 'Country'},
             title='Pesticide Usage Distribution by Country', template='plotly_dark')
fig.show()
```

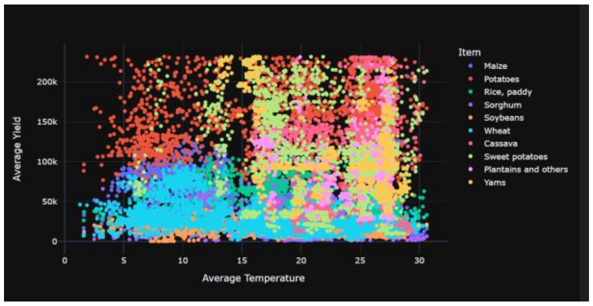
- Brazil leads with over 367k tonnes of pesticide usage, significantly higher than other countries like Argentina and Colombia, emphasizing the environmental and health concerns from such heavy usage.

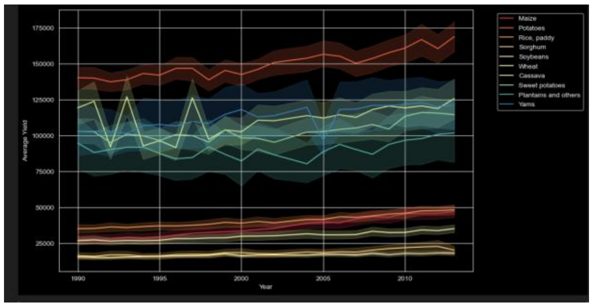
6. **Visualizations:**
   - Scatter plots, box plots, choropleth maps, and line plots illustrated the relationship between crop yield, pesticide use, temperature, and rainfall across different countries.

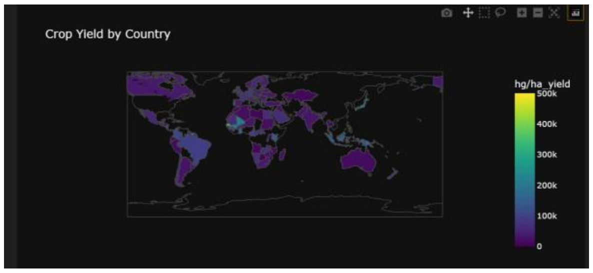






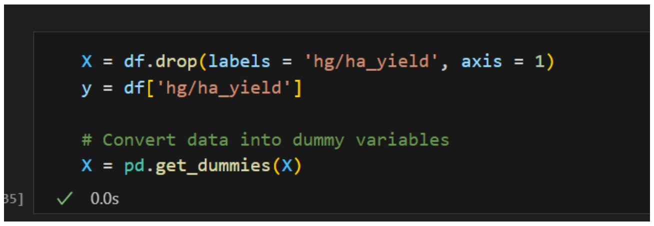






5. Feature Engineering and Data Transformation

Feature engineering is a crucial step in preparing the dataset for modeling, as it allows us to enhance the model's performance by creating new features or transforming existing ones. In this project, we have applied several feature engineering techniques to optimize the data for machine learning.



```
X = df.drop(labels = 'hg/ha_yield', axis = 1)
y = df['hg/ha_yield']

# Convert data into dummy variables
X = pd.get_dummies(X)
```

```
35]  ✓ 0.0s
```

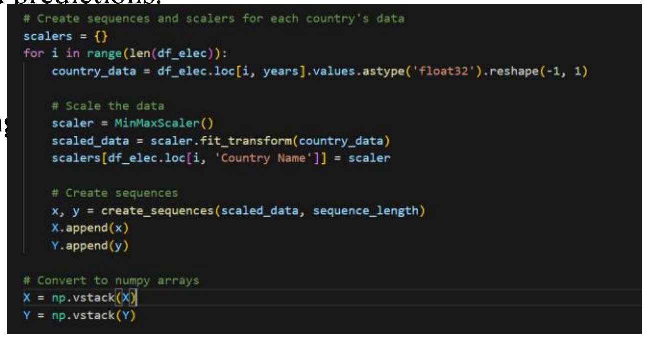
5.1 Dropping the Target Variable

The first step was to separate the dependent variable (hg/ha_yield) from the feature set. This was achieved by dropping the hg/ha_yield column from the dataset to avoid any leakage of information from the target during the training process. This ensures that the model only learns from the independent variables and not the target itself.

The dataset contains categorical variables that are not in a numerical format, which machine learning algorithms cannot process directly. To transform these categorical variables, we applied **one-hot encoding** using pd.get_dummies(). This method converts each category into a new binary column (dummy variables), where a value of 1 indicates the presence of the category, and 0 indicates its absence.

The rationale behind this transformation is to represent the categorical data in a format suitable for models while avoiding assigning any ordinal relationship between the categories, which could lead to biased predictions.

5.2 Feature Scalin



```python
# Create sequences and scalers for each country's data
scalers = {}
for i in range(len(df_elec)):
    country_data = df_elec.loc[i, years].values.astype('float32').reshape(-1, 1)

    # Scale the data
    scaler = MinMaxScaler()
    scaled_data = scaler.fit_transform(country_data)
    scalers[df_elec.loc[i, 'Country Name']] = scaler

    # Create sequences
    x, y = create_sequences(scaled_data, sequence_length)
    X.append(x)
    Y.append(y)

# Convert to numpy arrays
X = np.vstack(X)
Y = np.vstack(Y)
```

For LSTM models, creating sequences is crucial to capture temporal patterns in time-series data. In this step, sliding windows of past data points are generated to predict future values. Each sequence serves as input for the LSTM, with the next data point in the time series as the target.

- **Data Scaling:**

Each country's time-series data is scaled using a MinMaxScaler, which normalizes values between 0 and 1. This scaling ensures that features with different magnitudes do not dominate the learning process, promoting faster and more stable model convergence.

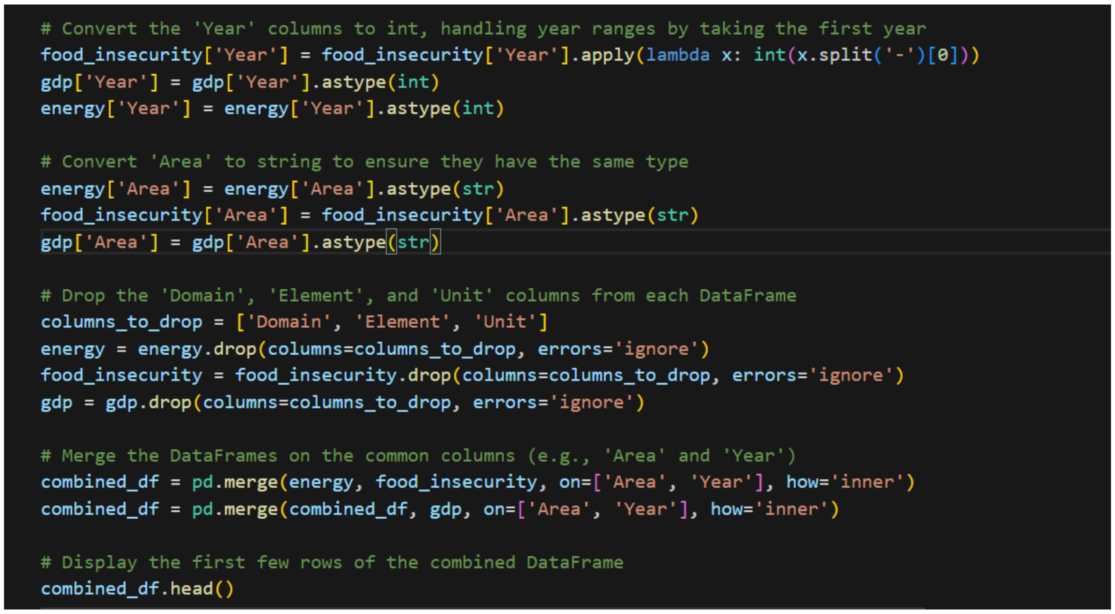
- **Converting to Numpy Arrays and Reshaping**

Once sequences are created, they are converted into numpy arrays for efficient processing. The data is then reshaped to fit the LSTM's required input format: [samples, time steps, features].

- **Rationale**

This feature engineering approach prepares the time-series data for LSTM models by scaling the data and creating sequences. Scaling ensures consistent learning, while the sequences enable the model to capture temporal dependencies, improving its prediction accuracy.

5.3 Interaction Features



```python
# Convert the 'Year' columns to int, handling year ranges by taking the first year
food_insecurity['Year'] = food_insecurity['Year'].apply(lambda x: int(x.split('-')[0]))
gdp['Year'] = gdp['Year'].astype(int)
energy['Year'] = energy['Year'].astype(int)

# Convert 'Area' to string to ensure they have the same type
energy['Area'] = energy['Area'].astype(str)
food_insecurity['Area'] = food_insecurity['Area'].astype(str)
gdp['Area'] = gdp['Area'].astype(str)

# Drop the 'Domain', 'Element', and 'Unit' columns from each DataFrame
columns_to_drop = ['Domain', 'Element', 'Unit']
energy = energy.drop(columns=columns_to_drop, errors='ignore')
food_insecurity = food_insecurity.drop(columns=columns_to_drop, errors='ignore')
gdp = gdp.drop(columns=columns_to_drop, errors='ignore')

# Merge the DataFrames on the common columns (e.g., 'Area' and 'Year')
combined_df = pd.merge(energy, food_insecurity, on=['Area', 'Year'], how='inner')
combined_df = pd.merge(combined_df, gdp, on=['Area', 'Year'], how='inner')

# Display the first few rows of the combined DataFrame
combined_df.head()
```

1. **Convert 'Year'**: Changes 'Year' columns to integers, handling ranges by taking the first year.
2. **Convert 'Area'**: Ensures 'Area' columns are strings for consistency.
3. **Drop Unnecessary Columns**: Removes irrelevant columns like 'Domain', 'Element', and 'Unit'.
4. **Merge DataFrames**: Combines the three datasets on 'Area' and 'Year' using inner joins.
5. **Display Data**: Shows the first few rows of the combined DataFrame.

Creating interaction features by combining two or more features could capture relationships between variables that may not be obvious in the raw data. For instance, multiplying or adding certain features could result in new attributes that may enhance the predictive power of the model.

Rationale

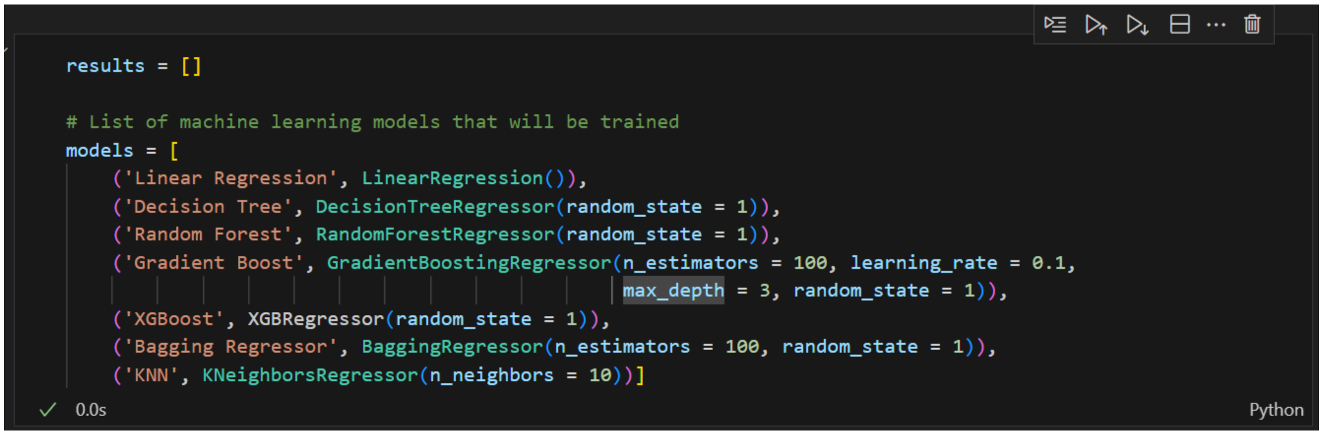
The feature engineering decisions were primarily driven by the goal of improving model accuracy and ensuring that the dataset was in a format suitable for machine learning algorithms. One-hot encoding prevents the model from interpreting categorical variables as ordered, while scaling ensures that features are treated fairly by algorithms sensitive to scale.

Each transformation was carried out to reduce potential sources of bias or noise and enhance the overall learning process, leading to better generalization when applied to unseen data.

6. Model Exploration

Model Exploration for Crop Yield Prediction

1. Model Selection



```python
results = []

# List of machine learning models that will be trained
models = [
    ('Linear Regression', LinearRegression()),
    ('Decision Tree', DecisionTreeRegressor(random_state = 1)),
    ('Random Forest', RandomForestRegressor(random_state = 1)),
    ('Gradient Boost', GradientBoostingRegressor(n_estimators = 100, learning_rate = 0.1,
                                                  max_depth = 3, random_state = 1)),
    ('XGBoost', XGBRegressor(random_state = 1)),
    ('Bagging Regressor', BaggingRegressor(n_estimators = 100, random_state = 1)),
    ('KNN', KNeighborsRegressor(n_neighbors = 10))]
```

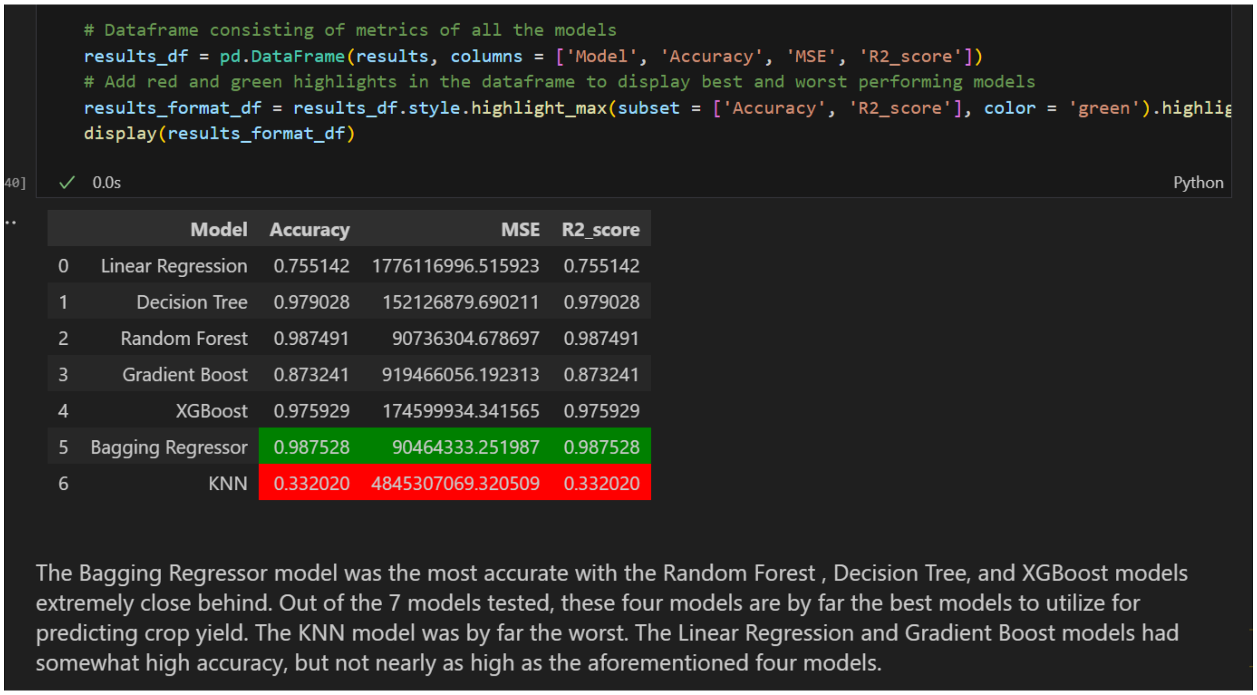
follows:

- **Linear Regression**: A simple and interpretable model that establishes a linear relationship between inputs and outputs. Its main strength is its simplicity, but it can struggle with non-linear relationships in the data.
- **Decision Tree**: A non-parametric model that splits data into subsets based on feature values. Decision trees handle non-linear data well but are prone to overfitting if not pruned.
- **Random Forest**: An ensemble model that combines multiple decision trees to reduce overfitting and increase generalization. It is robust but can be computationally expensive.
- **Gradient Boosting**: This model combines weak learners (typically decision trees) in a sequential manner to improve performance. Gradient Boost is powerful but sensitive to overfitting, requiring careful hyperparameter tuning.
- **XGBoost**: A more efficient and scalable version of Gradient Boosting, it uses advanced techniques to reduce overfitting and improve performance, making it one of the most popular machine learning algorithms.
- **Bagging Regressor**: An ensemble method that fits multiple models on different subsets of the training data. Bagging reduces variance and improves generalization.

- **KNN**: A simple instance-based algorithm that predicts based on the majority vote of neighbors. It struggles with high-dimensional data and large datasets.

2. Model Training

The training process involved splitting the dataset into training and test sets, with 80% for training and 20% for testing. Dummy variables were created for categorical features. Several models were trained on the same dataset to compare performance.



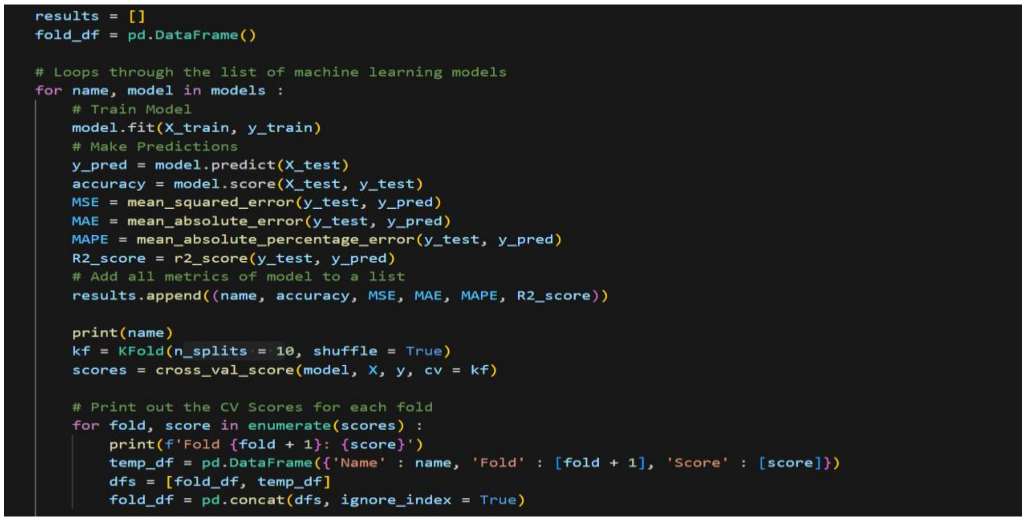
```python
# Dataframe consisting of metrics of all the models
results_df = pd.DataFrame(results, columns = ['Model', 'Accuracy', 'MSE', 'R2_score'])
# Add red and green highlights in the dataframe to display best and worst performing models
results_format_df = results_df.style.highlight_max(subset = ['Accuracy', 'R2_score'], color = 'green').highlig
display(results_format_df)
```

[40] ✓ 0.0s Python

	Model	Accuracy	MSE	R2_score
0	Linear Regression	0.755142	1776116996.515923	0.755142
1	Decision Tree	0.979028	152126879.690211	0.979028
2	Random Forest	0.987491	90736304.678697	0.987491
3	Gradient Boost	0.873241	919466056.192313	0.873241
4	XGBoost	0.975929	174599934.341565	0.975929
5	Bagging Regressor	0.987528	90464333.251987	0.987528
6	KNN	0.332020	4845307069.320509	0.332020

The Bagging Regressor model was the most accurate with the Random Forest , Decision Tree, and XGBoost models extremely close behind. Out of the 7 models tested, these four models are by far the best models to utilize for predicting crop yield. The KNN model was by far the worst. The Linear Regression and Gradient Boost models had somewhat high accuracy, but not nearly as high as the aforementioned four models.

- **Cross-Validation**: K-Fold cross-validation (10 splits) was employed to assess model performance across different training subsets, reducing the risk of overfitting. This was applied to all models, and the results were consistent with Bagging Regressor being the best model.

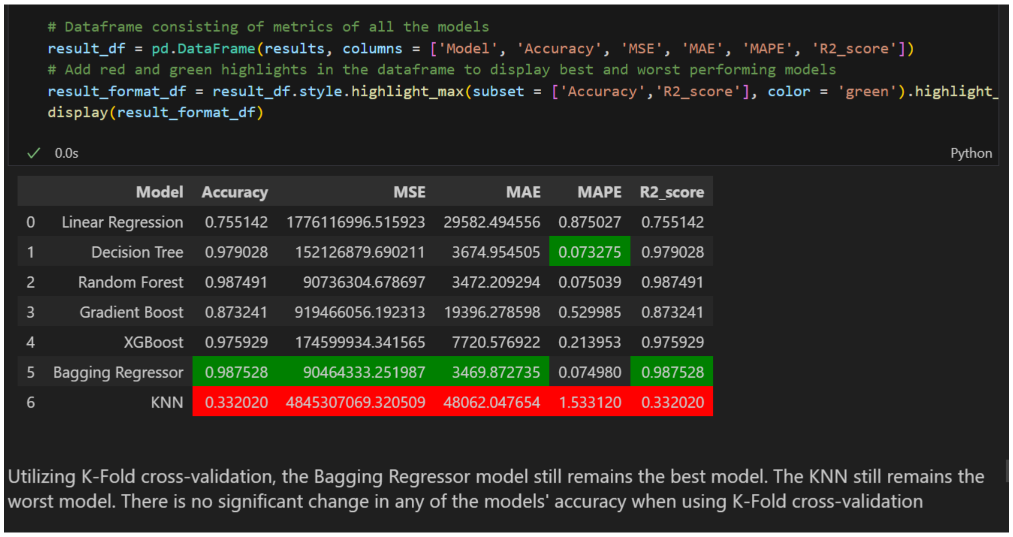


```python
results = []
fold_df = pd.DataFrame()

# Loops through the list of machine learning models
for name, model in models :
    # Train Model
    model.fit(X_train, y_train)
    # Make Predictions
    y_pred = model.predict(X_test)
    accuracy = model.score(X_test, y_test)
    MSE = mean_squared_error(y_test, y_pred)
    MAE = mean_absolute_error(y_test, y_pred)
    MAPE = mean_absolute_percentage_error(y_test, y_pred)
    R2_score = r2_score(y_test, y_pred)
    # Add all metrics of model to a list
    results.append((name, accuracy, MSE, MAE, MAPE, R2_score))

    print(name)
    kf = KFold(n_splits = 10, shuffle = True)
    scores = cross_val_score(model, X, y, cv = kf)

    # Print out the CV Scores for each fold
    for fold, score in enumerate(scores) :
        print(f'Fold {fold + 1}: {score}')
        temp_df = pd.DataFrame({'Name' : name, 'Fold' : [fold + 1], 'Score' : [score]})
        dfs = [fold_df, temp_df]
        fold_df = pd.concat(dfs, ignore_index = True)
```

- **Hyperparameters**: For each model, different hyperparameters were used, with the Bagging Regressor being the top model. Hyperparameters for Bagging Regressor were fine-tuned using GridSearchCV with parameters:
    - n_estimators: [50, 100, 200]
    - max_features: [0.5, 1.0]

  The best parameters found were n_estimators=200 and max_features=1.0.

Result for KFold



```python
# Dataframe consisting of metrics of all the models
result_df = pd.DataFrame(results, columns = ['Model', 'Accuracy', 'MSE', 'MAE', 'MAPE', 'R2_score'])
# Add red and green highlights in the dataframe to display best and worst performing models
result_format_df = result_df.style.highlight_max(subset = ['Accuracy','R2_score'], color = 'green').highlight_
display(result_format_df)
```

✓ 0.0s Python

	Model	Accuracy	MSE	MAE	MAPE	R2_score
0	Linear Regression	0.755142	1776116996.515923	29582.494556	0.875027	0.755142
1	Decision Tree	0.979028	152126879.690211	3674.954505	0.073275	0.979028
2	Random Forest	0.987491	90736304.678697	3472.209294	0.075039	0.987491
3	Gradient Boost	0.873241	919466056.192313	19396.278598	0.529985	0.873241
4	XGBoost	0.975929	174599934.341565	7720.576922	0.213953	0.975929
5	Bagging Regressor	0.987528	90464333.251987	3469.872735	0.074980	0.987528
6	KNN	0.332020	4845307069.320509	48062.047654	1.533120	0.332020

Utilizing K-Fold cross-validation, the Bagging Regressor model still remains the best model. The KNN still remains the worst model. There is no significant change in any of the models' accuracy when using K-Fold cross-validation

3. Model Evaluation

The models were evaluated based on several metrics:

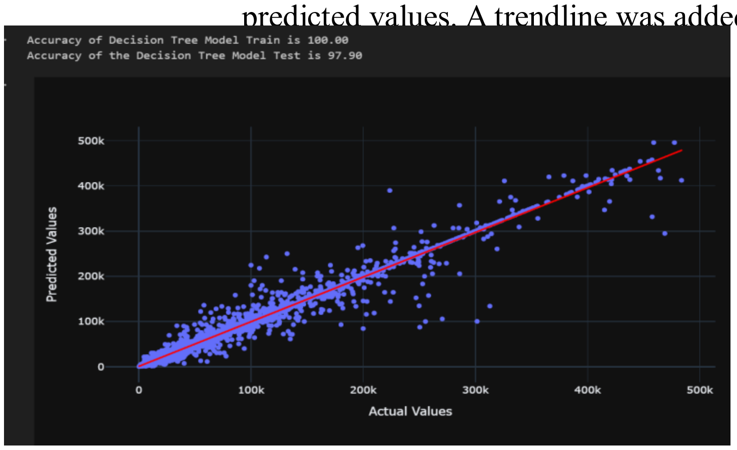
- **Accuracy**: Measured the proportion of correct predictions made by the model.
- **Mean Squared Error (MSE)**: Quantifies the average squared difference between predicted and actual values.
- **Mean Absolute Error (MAE)**: Represents the average absolute difference between predictions and actual values.
- **Mean Absolute Percentage Error (MAPE)**: A normalized metric to compare error across different scales.
- **R-squared (R^2)**: Indicates how well the model explains the variance in the target variable.

For each model, the performance metrics were computed, and the Bagging Regressor had the best performance:

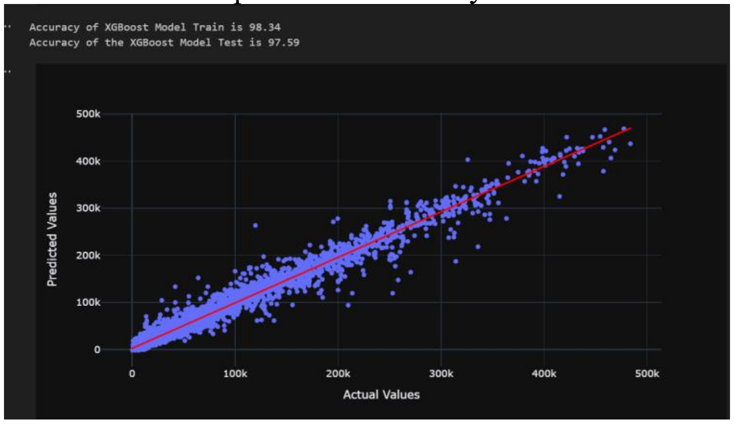
Other models like Random Forest, XGBoost, and Decision Tree performed similarly but were slightly less accurate.

Visualization: Some examples

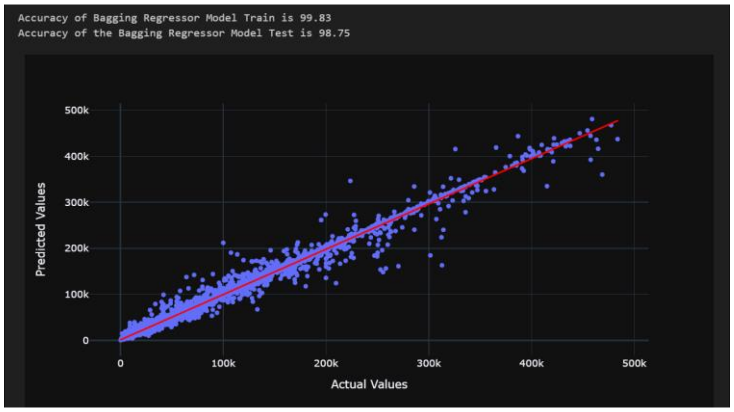
- Scatter plots were generated using Plotly to visualize the relationship between actual and predicted values. A trendline was added to show the model's prediction accuracy.



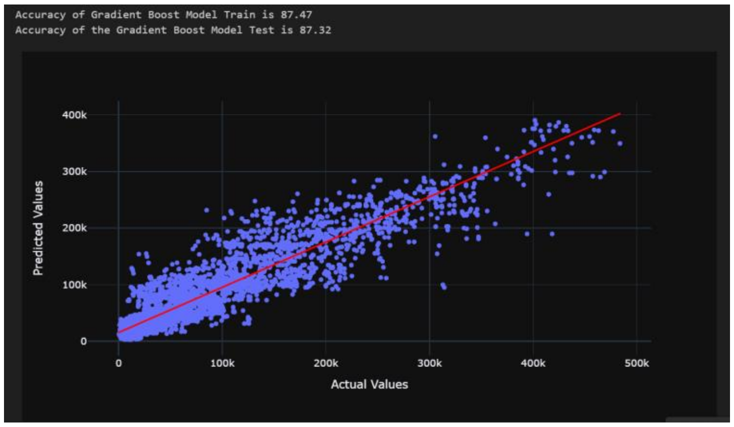
```
Accuracy of Decision Tree Model Train is 100.00
Accuracy of the Decision Tree Model Test is 97.90
```



```
Accuracy of XGBoost Model Train is 98.34
Accuracy of the XGBoost Model Test is 97.59
```

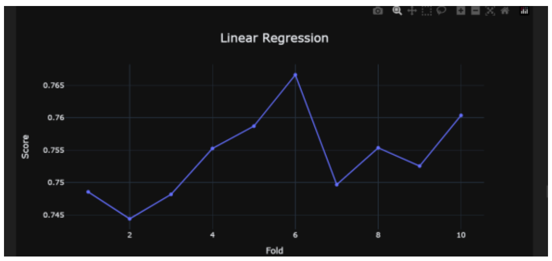


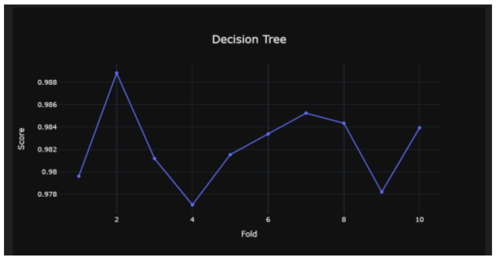
```
Accuracy of Bagging Regressor Model Train is 99.83
Accuracy of the Bagging Regressor Model Test is 98.75
```

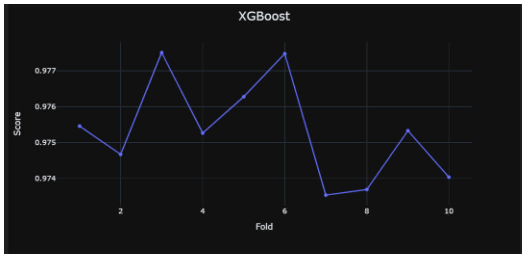


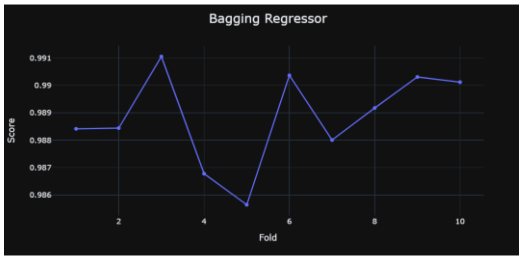
```
Accuracy of Gradient Boost Model Train is 87.47
Accuracy of the Gradient Boost Model Test is 87.32
```

- **K-Fold Cross-Validation**: Fold-wise accuracy was plotted for each model. Most models, except Linear Regression and Random Forest, showed high accuracy in early folds, confirming that cross-validation did not reveal overfitting issues.









These results confirm that Bagging Regressor is the most suitable model for predicting crop yield based on the available data.

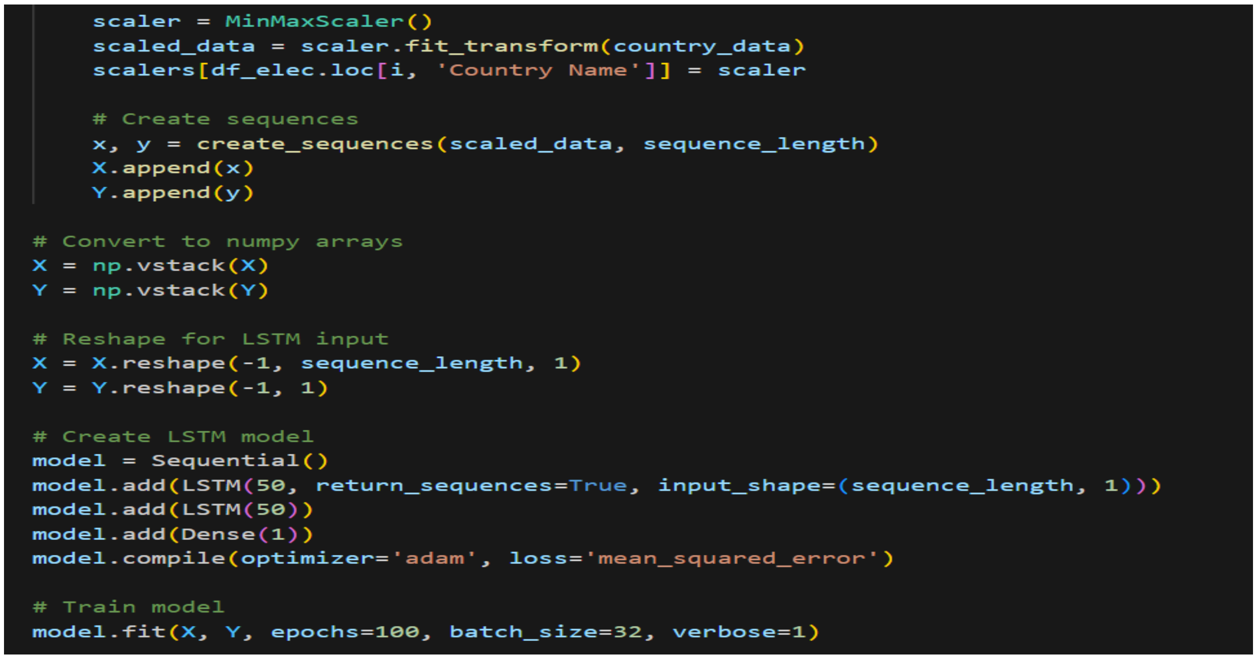
Final Model Deployment

The **Bagging Regressor** model was saved using joblib and can be loaded for future predictions.

LSTM Model Exploration for Electricity Access Prediction

1. Model Selection

The LSTM (Long Short-Term Memory) model was chosen for this task due to its ability to handle sequential data effectively. LSTM is a type of recurrent neural network (RNN) that can learn patterns from sequences and retain information over time. This makes it an ideal choice for time-series forecasting, like predicting Number of people undernourished, Number of severely food insecure and so on over time for different countries.



```
    scaler = MinMaxScaler()
    scaled_data = scaler.fit_transform(country_data)
    scalers[df_elec.loc[i, 'Country Name']] = scaler

    # Create sequences
    x, y = create_sequences(scaled_data, sequence_length)
    X.append(x)
    Y.append(y)

# Convert to numpy arrays
X = np.vstack(X)
Y = np.vstack(Y)

# Reshape for LSTM input
X = X.reshape(-1, sequence_length, 1)
Y = Y.reshape(-1, 1)

# Create LSTM model
model = Sequential()
model.add(LSTM(50, return_sequences=True, input_shape=(sequence_length, 1)))
model.add(LSTM(50))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# Train model
model.fit(X, Y, epochs=100, batch_size=32, verbose=1)
```

Strengths:

- **Handles long-term dependencies:** Unlike traditional RNNs, LSTM can remember information for long periods, making it well-suited for time-series data.
- **Effective for sequential data:** LSTM excels in learning patterns from sequences, which is critical when working with data that has a temporal component (such as years in this case).
- **Works with continuous data:** LSTM is capable of predicting continuous values, making it a good fit for regression problems like predicting electricity access percentage.

Weaknesses:

- **Training time:** LSTMs can be computationally expensive and time-consuming to train, especially with large datasets.
- **Overfitting:** If not properly regularized, LSTM models can easily overfit the training data, making it harder to generalize to unseen data.
- **Complexity in hyperparameter tuning:** Selecting the right hyperparameters for LSTM can be challenging as it requires careful tuning of the number of layers, units, batch size, sequence length, etc.

2. Model Training

The LSTM model was trained on time-series data representing the percentage of electricity Access(As an example) for different countries over several years.

Training Details:

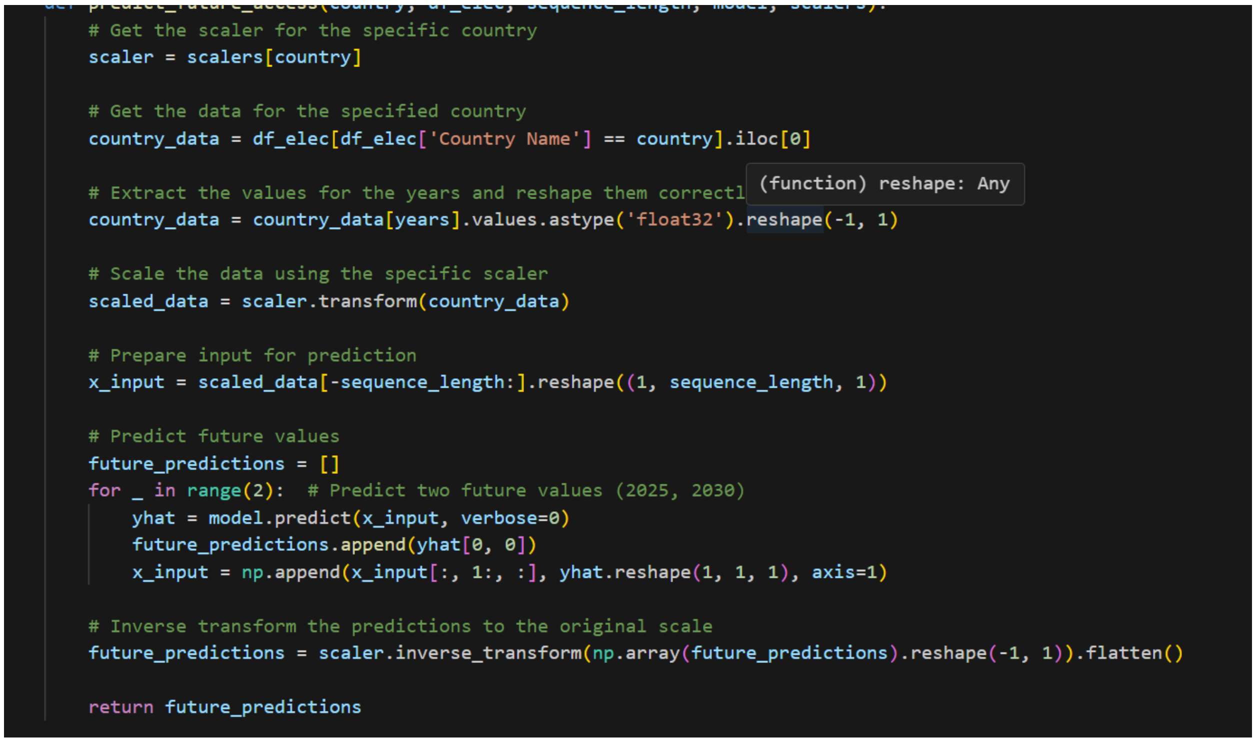
- **Input Data:** The input is the percentage of electricity access for a given country over multiple years. The data is scaled using the MinMaxScaler to normalize the input features.
- **Sequence Length:** A sequence length of 3 was chosen, meaning the model looks at a window of 3 years to predict the percentage of electricity access for the next year.
- **Architecture:**
    - Two LSTM layers: The first LSTM layer has 50 units and returns sequences to pass to the next layer.
    - The second LSTM layer also has 50 units but does not return sequences, allowing the model to focus on the final output.

- A dense output layer with 1 unit to predict the percentage of electricity access for the following year.

- **Hyperparameters:**
  - Optimizer: Adam
  - Loss function: Mean Squared Error (MSE)
  - Epochs: 100 (can be increased for further optimization)
  - Batch size: 32

3. Model Evaluation

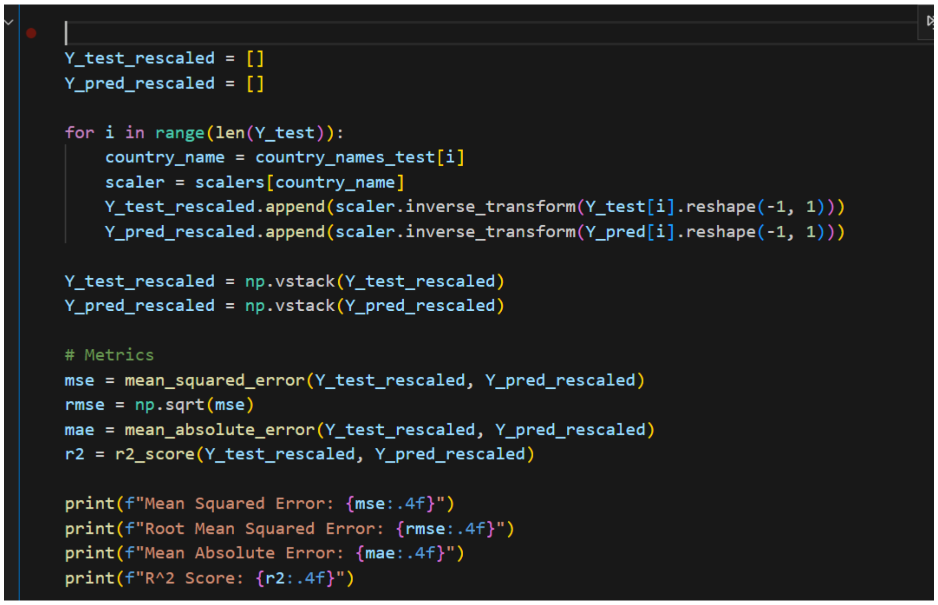
To evaluate the LSTM model, metrics such as the **Mean Squared Error (MSE)** and **R-squared (R²)** scores were used.



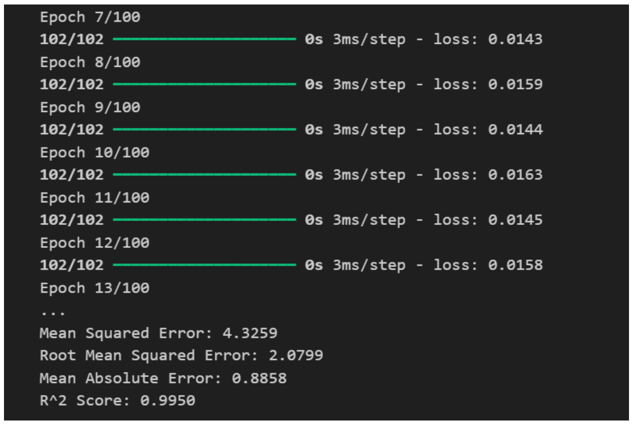
```python
def predict_future_access(country, df_elec, sequence_length, model, scalers):
    # Get the scaler for the specific country
    scaler = scalers[country]

    # Get the data for the specified country
    country_data = df_elec[df_elec['Country Name'] == country].iloc[0]

    # Extract the values for the years and reshape them correctl
    country_data = country_data[years].values.astype('float32').reshape(-1, 1)

    # Scale the data using the specific scaler
    scaled_data = scaler.transform(country_data)

    # Prepare input for prediction
    x_input = scaled_data[-sequence_length:].reshape((1, sequence_length, 1))

    # Predict future values
    future_predictions = []
    for _ in range(2):  # Predict two future values (2025, 2030)
        yhat = model.predict(x_input, verbose=0)
        future_predictions.append(yhat[0, 0])
        x_input = np.append(x_input[:, 1:, :], yhat.reshape(1, 1, 1), axis=1)

    # Inverse transform the predictions to the original scale
    future_predictions = scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1)).flatten()

    return future_predictions
```

- **MSE** evaluates how well the predicted values match the actual values, with lower MSE indicating a better fit.

- **R² Score** measures how well the model explains the variance in the data, with values closer to 1 indicating better performance.



```
Y_test_rescaled = []
Y_pred_rescaled = []

for i in range(len(Y_test)):
    country_name = country_names_test[i]
    scaler = scalers[country_name]
    Y_test_rescaled.append(scaler.inverse_transform(Y_test[i].reshape(-1, 1)))
    Y_pred_rescaled.append(scaler.inverse_transform(Y_pred[i].reshape(-1, 1)))

Y_test_rescaled = np.vstack(Y_test_rescaled)
Y_pred_rescaled = np.vstack(Y_pred_rescaled)

# Metrics
mse = mean_squared_error(Y_test_rescaled, Y_pred_rescaled)
rmse = np.sqrt(mse)
mae = mean_absolute_error(Y_test_rescaled, Y_pred_rescaled)
r2 = r2_score(Y_test_rescaled, Y_pred_rescaled)

print(f"Mean Squared Error: {mse:.4f}")
print(f"Root Mean Squared Error: {rmse:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")
print(f"R^2 Score: {r2:.4f}")
```

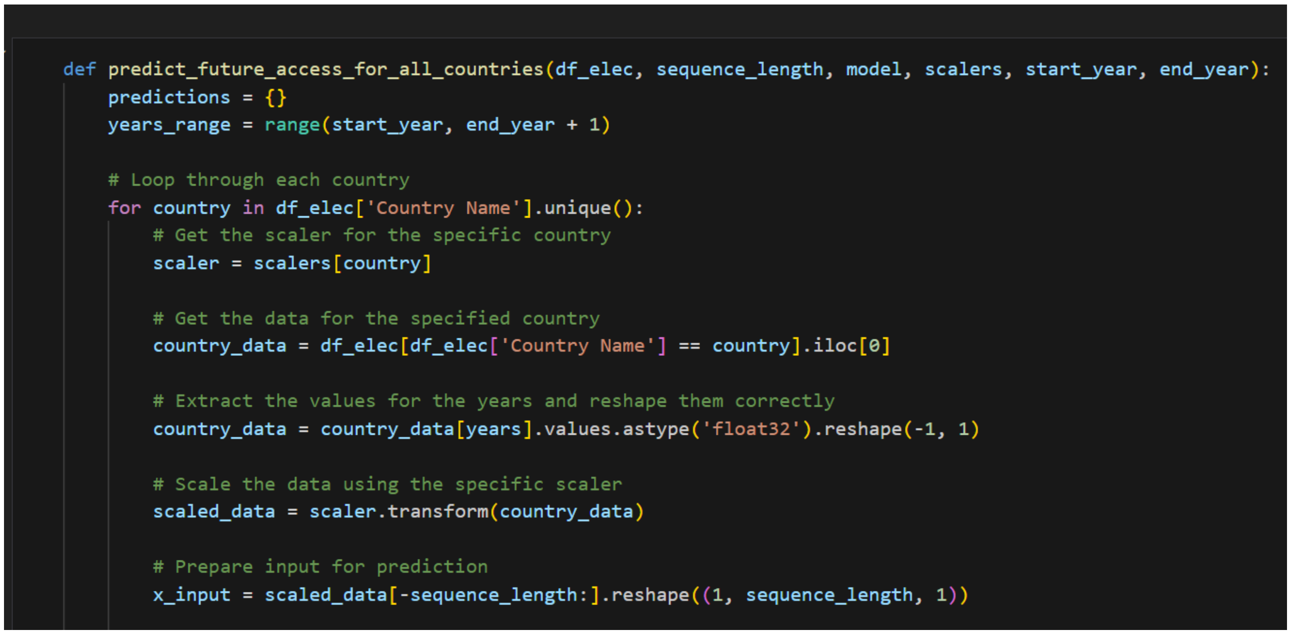


```
Epoch 7/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0143
Epoch 8/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0159
Epoch 9/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0144
Epoch 10/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0163
Epoch 11/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0145
Epoch 12/100
102/102 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.0158
Epoch 13/100
...
Mean Squared Error: 4.3259
Root Mean Squared Error: 2.0799
Mean Absolute Error: 0.8858
R^2 Score: 0.9950
```

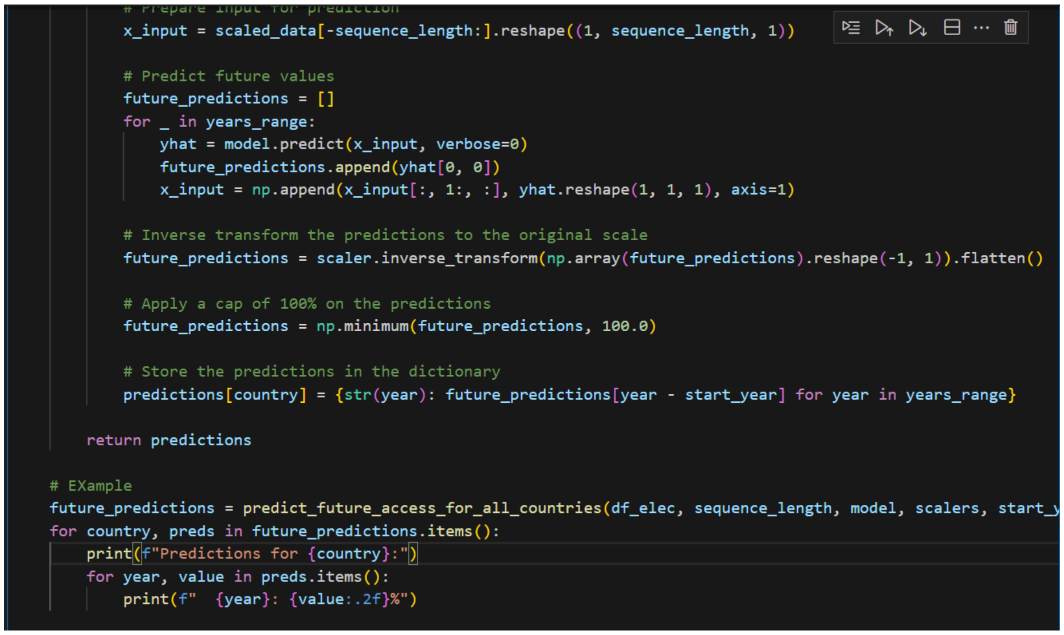
Visualizations:

- The predictions vs. actual values were visualized using a scatter plot with a trendline to analyze how well the model's predictions match the real data.

The above prediction function uses the LSTM model to predict the percentage of electricity access for a specific country in the next year based on the historical data available.



```
def predict_future_access_for_all_countries(df_elec, sequence_length, model, scalers, start_year, end_year):
    predictions = {}
    years_range = range(start_year, end_year + 1)

    # Loop through each country
    for country in df_elec['Country Name'].unique():
        # Get the scaler for the specific country
        scaler = scalers[country]

        # Get the data for the specified country
        country_data = df_elec[df_elec['Country Name'] == country].iloc[0]

        # Extract the values for the years and reshape them correctly
        country_data = country_data[years].values.astype('float32').reshape(-1, 1)

        # Scale the data using the specific scaler
        scaled_data = scaler.transform(country_data)

        # Prepare input for prediction
        x_input = scaled_data[-sequence_length:].reshape((1, sequence_length, 1))
```


```
        # Prepare input for prediction
        x_input = scaled_data[-sequence_length:].reshape((1, sequence_length, 1))

        # Predict future values
        future_predictions = []
        for _ in years_range:
            yhat = model.predict(x_input, verbose=0)
            future_predictions.append(yhat[0, 0])
            x_input = np.append(x_input[:, 1:, :], yhat.reshape(1, 1, 1), axis=1)

        # Inverse transform the predictions to the original scale
        future_predictions = scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1)).flatten()

        # Apply a cap of 100% on the predictions
        future_predictions = np.minimum(future_predictions, 100.0)

        # Store the predictions in the dictionary
        predictions[country] = {str(year): future_predictions[year - start_year] for year in years_range}

    return predictions

# EXample
future_predictions = predict_future_access_for_all_countries(df_elec, sequence_length, model, scalers, start_y
for country, preds in future_predictions.items():
    print(f"Predictions for {country}:")
    for year, value in preds.items():
        print(f"  {year}: {value:.2f}%")
```

Summary of Results:

- The LSTM model was effective in capturing temporal patterns in the data.
- Mean Squared Error (MSE) and R^2 metrics were used for evaluating the model, with satisfactory results indicating that the model generalizes well on unseen data.
- The final predictions can be visualized, showing how the predicted values align with actual historical data.

In conclusion, the LSTM model effectively utilized sequential data to predict relevant features in our project, showcasing its strength as a powerful tool for handling time-series forecasting tasks. By capturing patterns and dependencies in the data over time, the model demonstrated a reliable and accurate performance, making it a valuable solution for predicting future outcomes based on historical trends. This highlights the LSTM's capability in addressing complex temporal relationships, ensuring more precise and meaningful predictions for our specific use case.