# **Programming Language and Version:** Python 3.9.6

# **Environment Setup Variables:** None

# **Additional Information:** The project utilizes standard Python libraries and does not require any specific environment setup variables.

```
# Data Preprocessing

## Countries: China, USA, Brazil, Indonesia
## Crops: Rice, Wheat, Corn, Soya Beans
## Year: 2000-2021

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.impute import KNNImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from scipy.sparse import hstack, csr_matrix

import warnings
# Suppress FutureWarnings
warnings.simplefilter(action='ignore', category=FutureWarning)

"""## Crop_Production"""

Crop_Production_df = pd.read_csv("Crop_Production.csv")
Crop_Production_df.drop(['Domain Code', 'Domain', 'Area Code (M49)',
'Element Code',
                         'Item Code (CPC)', 'Year Code', 'Flag', 'Flag
Description', 'Note'], axis=1,inplace=True)
Crop_Production_df =
Crop_Production_df[Crop_Production_df['Year'].between(2000, 2021)]
Crop_Production_df.rename(columns={'Area': 'Country','Element':
'Prod_type', 'Item': 'Crop_Name', 'Value': 'Crop_Production_Value',
                                   'Unit': 'Crop_Production_Unit'},
inplace=True)

Crop_Production_df.head()

unique_values = Crop_Production_df['Year'].unique()
print(unique_values)

"""### *Pivoting the Prod_type Column*"""

Crop_Production_df['Prod_type'] = Crop_Production_df['Prod_type'] + '_' +
Crop_Production_df['Crop_Production_Unit']
```

```
# Pivot the DataFrame
Crop_Production_pivot_df = Crop_Production_df.pivot(index=['Country',
'Crop_Name', 'Year'], columns='Prod_type',
values='Crop_Production_Value').reset_index()

# Rename columns
Crop_Production_pivot_df.columns.name = None

# Display the pivoted DataFrame
Crop_Production_pivot_df.head()

null_values_count = Crop_Production_pivot_df.isnull().sum()
print(null_values_count)

# Data Types Check
print("Data Types Check:")
print(Crop_Production_pivot_df.dtypes)
print("\n")

# Data Distribution Check
print("Data Distribution Check:")
print(Crop_Production_pivot_df.describe())
print("\n")

## Crop_Trade

Crop_Trade_df = pd.read_csv("Crop_Trade.csv")
Crop_Trade_df.drop(['Domain Code', 'Domain', 'Area Code (M49)', 'Element
Code',
                         'Item Code (CPC)', 'Year Code', 'Flag', 'Flag
Description', 'Note'], axis=1,inplace=True)
Crop_Trade_df = Crop_Trade_df[Crop_Trade_df['Year'].between(2000, 2021)]
Crop_Trade_df.rename(columns={'Area': 'Country', 'Element': 'Trade_type',
'Item': 'Crop_Name',
                              'Value': 'Trade_Value',  'Unit':
'Trade_Unit'}, inplace=True)

Crop_Trade_df.head()

unique_values = Crop_Trade_df['Year'].unique()
print(unique_values)

"""### *Pivoting the Trade_type Column*"""

Crop_Trade_df['Trade_type'] = Crop_Trade_df['Trade_type'] + '_' +
Crop_Trade_df['Trade_Unit']


# Pivot the DataFrame
Crop_Trade_pivot_df = Crop_Trade_df.pivot(index=['Country', 'Crop_Name',
'Year'], columns='Trade_type', values='Trade_Value').reset_index()

# Rename columns
Crop_Trade_pivot_df.columns.name = None
```

```
# Display the pivoted DataFrame
Crop_Trade_pivot_df.head()

# Data Types Check
print("Data Types Check:")
print(Crop_Trade_pivot_df.dtypes)
print("\n")

null_values_count = Crop_Trade_pivot_df.isnull().sum()
print(null_values_count)

# Visualize distribution before handling missing values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(Crop_Trade_pivot_df["Export Quantity_t"], bins=20,
color='skyblue', edgecolor='black')
plt.title('Export Quantity Distribution (Before)')
plt.xlabel('Export Quantity (tonnes)')
plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
plt.hist(Crop_Trade_pivot_df["Export Value_1000 USD"], bins=20,
color='salmon', edgecolor='black')
plt.title('Export Value Distribution (Before)')
plt.xlabel('Export Value (1000 USD)')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

# Fill missing values with appropriate methods (e.g., mean, median, zero)
Crop_Trade_pivot_df["Export
Quantity_t"].fillna(Crop_Trade_pivot_df["Export Quantity_t"].mean(),
inplace=True)
Crop_Trade_pivot_df["Export Value_1000 USD"].fillna(0, inplace=True)  #
Assuming missing values indicate zero export value

# Check if there are any remaining missing values
missing_values = Crop_Trade_pivot_df.isnull().sum()
print("Remaining missing values:")
print(missing_values)

# Summary statistics after handling missing values
print("Summary statistics after handling missing values:")
print(Crop_Trade_pivot_df.describe())
print("\n")

# Visualize distribution after handling missing values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.hist(Crop_Trade_pivot_df["Export Quantity_t"], bins=20,
color='skyblue', edgecolor='black')
plt.title('Export Quantity Distribution (After)')
plt.xlabel('Export Quantity (tonnes)')
plt.ylabel('Frequency')

plt.subplot(1, 2, 2)
plt.hist(Crop_Trade_pivot_df["Export Value_1000 USD"], bins=20,
color='salmon', edgecolor='black')
plt.title('Export Value Distribution (After)')
plt.xlabel('Export Value (1000 USD)')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()

## Emission_from_Crops

Emission_from_Crops_df = pd.read_csv("Emission_from_Crops.csv")
Emission_from_Crops_df.drop(['Domain Code', 'Domain', 'Area Code (M49)',
'Element Code', 'Source', 'Source Code',
                    'Item Code (CPC)', 'Year Code', 'Flag', 'Flag
Description', 'Note'], axis=1,inplace=True)
Emission_from_Crops_df.rename(columns={'Area': 'Country','Element':
'Emission_type', 'Item': 'Crop_Name', 'Value': 'Emission_Value_kt',
'Unit': 'Emission_Unit'}, inplace=True)

Emission_from_Crops_df.head()

Emission_from_Crops_df.drop(columns = "Emission_Unit", axis = 1, inplace =
True)

Emission_from_Crops_df.shape

Emission_from_Crops_df.head()

null_values_count = Emission_from_Crops_df.isnull().sum()
print(null_values_count)

# Data Types Check
print("Data Types Check:")
print(Emission_from_Crops_df.dtypes)
print("\n")

# Summary statistics
print("Summary statistics:")
print(Emission_from_Crops_df.describe())
print("\n")

## Land_Use

Land_Use_df = pd.read_csv("Land_Use.csv")
Land_Use_df.drop(['Domain Code', 'Domain', 'Area Code (M49)', 'Element
Code', 'Element',
                    'Item Code', 'Year Code', 'Flag', 'Flag Description',
'Note'], axis=1,inplace=True)
```

```
Land_Use_df.rename(columns={'Area': 'Country','Item': 'Area_type',
'Value': 'Area_Value',  'Unit': 'Area_Unit'}, inplace=True)

Land_Use_df.head()

unique_values = Land_Use_df['Year'].unique()
print(unique_values)

# Rename the "Area_Value" column to "Area_Value_1000_ha"

Land_Use_df.rename(columns={'Area_Value': 'Area_Value_1000 ha'},
inplace=True)

# Drop the "Area_Unit" column
Land_Use_df.drop(columns="Area_Unit", inplace=True)

Land_Use_df.head()

null_values_count = Land_Use_df.isnull().sum()
print(null_values_count)

# Data Types Check
print("Data Types Check:")
print(Land_Use_df.dtypes)
print("\n")

# Summary statistics
print("Summary statistics:")
print(Land_Use_df.describe())
print("\n")

## Pesticides_Use


Pesticides_Use_df = pd.read_csv("Pesticides_Use.csv")
Pesticides_Use_df.drop(['Domain Code', 'Domain', 'Area Code (M49)',
'Element Code', 'Element',
                    'Item Code', 'Year Code', 'Flag', 'Flag Description',
'Note'], axis=1,inplace=True)
Pesticides_Use_df.rename(columns={'Area':'Country','Item':
'Pesticide_Type', 'Value': 'Pesticide_Value',  'Unit': 'Pesticide_Unit'},
inplace=True)

Pesticides_Use_df.head()

unique_values = Pesticides_Use_df['Year'].unique()
print(unique_values)

# Rename the "Pesticide_Value" column to "Pesticide_Value_t"

Pesticides_Use_df.rename(columns={'Pesticide_Value': 'Pesticide_Value_t'},
inplace=True)

# Drop the "Pesticide_Unit" column
```

```
Pesticides_Use_df.drop(columns="Pesticide_Unit", inplace=True)

Pesticides_Use_df.head()

null_values_count = Pesticides_Use_df.isnull().sum()
print(null_values_count)

# Data Types Check
print("Data Types Check:")
print(Pesticides_Use_df.dtypes)
print("\n")

# Summary statistics
print("Summary statistics:")
print(Pesticides_Use_df.describe())
print("\n")

## Value_of_Agricultural_production

Value_of_Agricultural_production_df =
pd.read_csv("Value_of_Agricultural_Production.csv")
Value_of_Agricultural_production_df.drop(['Domain Code', 'Domain', 'Area
Code (M49)', 'Element', 'Element Code', 'Unit',
                          'Item Code (CPC)', 'Year Code', 'Flag', 'Flag
Description'], axis=1,inplace=True)
Value_of_Agricultural_production_df =
Value_of_Agricultural_production_df[Value_of_Agricultural_production_df['Y
ear'].between(2000, 2021)]
Value_of_Agricultural_production_df.rename(columns={'Area':'Country','Item
': 'Crop_Name', 'Value': 'Gross_Production_Value_1000 USD',  'Unit':
'Agri_Prod_Unit'}, inplace=True)
Value_of_Agricultural_production_df.head()

unique_values = Value_of_Agricultural_production_df['Year'].unique()
print(unique_values)

null_values_count = Value_of_Agricultural_production_df.isnull().sum()
print(null_values_count)

# Data Types Check
print("Data Types Check:")
print(Value_of_Agricultural_production_df.dtypes)
print("\n")

# Summary statistics
print("Summary statistics:")
print(Value_of_Agricultural_production_df.describe())
print("\n")

## Cropland_Nutrient_Balance


Cropland_Nutrient_Balance_df =
pd.read_csv("Cropland_Nutrient_Balance.csv")
```

```
Cropland_Nutrient_Balance_df.drop(['Domain Code', 'Domain', 'Area Code
(M49)', 'Element Code', 'Unit',
                         'Item Code', 'Year Code', 'Flag', 'Flag
Description', 'Note'], axis=1,inplace=True)
Cropland_Nutrient_Balance_df =
Cropland_Nutrient_Balance_df[Cropland_Nutrient_Balance_df['Year'].between(
2000, 2021)]
Cropland_Nutrient_Balance_df.rename(columns={'Area':'Country','Element':
'Cropland_Nutrient_Management_Metrics','Item': 'Input_Factors', 'Value':
'Fertilizer_Usage_Value_kg/ha',  'Unit': 'Agri_Prod_Unit'}, inplace=True)
Cropland_Nutrient_Balance_df.head()

unique_values = Cropland_Nutrient_Balance_df['Year'].unique()
print(unique_values)

null_values_count = Cropland_Nutrient_Balance_df.isnull().sum()
print(null_values_count)

total_values = Cropland_Nutrient_Balance_df.shape[0]

percentage_missing = (null_values_count / total_values) * 100

# Display the result
print(percentage_missing)

# Visualize Fertilizer Usage Distribution
plt.figure(figsize=(10, 6))
sns.boxplot(x='Input_Factors', y='Fertilizer_Usage_Value_kg/ha',
data=Cropland_Nutrient_Balance_df)
plt.title('Distribution of Fertilizer Usage by Input Factors')
plt.xlabel('Input Factors')
plt.ylabel('Fertilizer Usage')
plt.xticks(rotation=45)
plt.show()

# Investigate Group Sizes
group_sizes = Cropland_Nutrient_Balance_df['Input_Factors'].value_counts()
plt.figure(figsize=(10, 6))
sns.barplot(x=group_sizes.index, y=group_sizes.values)
plt.title('Group Sizes by Input Factors')
plt.xlabel('Input Factors')
plt.ylabel('Group Size')
plt.xticks(rotation=45)
plt.show()

"""- The group sizes for different input factors vary significantly.
- "Mineral fertilizers" and "Manure applied to soils" have the largest
group sizes, followed by "Crop residual" and "Biological fixation."
- "Seed" and "Leaching" have relatively smaller group sizes.
"""

# Visual inspection using boxplot
plt.figure(figsize=(8, 6))
```

```
sns.boxplot(x=Cropland_Nutrient_Balance_df['Fertilizer_Usage_Value_kg/ha']
)
plt.title('Boxplot of Fertilizer Usage')
plt.xlabel('Fertilizer Usage')
plt.show()

"""- The boxplot reveals the presence of several outliers, with some
extreme values in the fertilizer usage data.
- The distribution of fertilizer usage appears to be right-skewed, with a
long tail towards higher values.
"""

# Visualize distribution before imputation
plt.figure(figsize=(8, 6))
plt.hist(Cropland_Nutrient_Balance_df["Fertilizer_Usage_Value_kg/ha"].drop
na(), bins=20, color='skyblue', edgecolor='black')
plt.title('Fertilizer Usage Distribution (Before Imputation)')
plt.xlabel('Fertilizer Usage (kg/ha)')
plt.ylabel('Frequency')
plt.show()

"""### *Filling Missing Data*"""

from scipy.stats.mstats import winsorize

winsorized_values =
winsorize(Cropland_Nutrient_Balance_df['Fertilizer_Usage_Value_kg/ha'],
                              limits=(0.05, 0.95))

# Convert the winsorized values array into a pandas Series
winsorized_series = pd.Series(winsorized_values)

# Fill the missing values in the original DataFrame with the winsorized
values
Cropland_Nutrient_Balance_df['Fertilizer_Usage_Value_kg/ha'] =
Cropland_Nutrient_Balance_df['Fertilizer_Usage_Value_kg/ha'].fillna(winsor
ized_series)

# Verify that there are no more missing values
missing_values_after_imputation =
Cropland_Nutrient_Balance_df.isnull().sum()
print("Missing values after imputation:")
print(missing_values_after_imputation)

# Visualize distribution before imputation
plt.figure(figsize=(8, 6))
plt.hist(Cropland_Nutrient_Balance_df["Fertilizer_Usage_Value_kg/ha"].drop
na(), bins=20, color='orange', edgecolor='black')
plt.title('Fertilizer Usage Distribution (After Imputation)')
plt.xlabel('Fertilizer Usage (kg/ha)')
plt.ylabel('Frequency')
plt.show()
```

```
merged_df = pd.merge(Crop_Production_pivot_df, Crop_Trade_pivot_df,
on=['Country', 'Year','Crop_Name'], how='inner')
merged_df = pd.merge(merged_df,Emission_from_Crops_df, on=['Country',
'Year','Crop_Name'], how='inner')
merged_df = pd.merge(merged_df,Value_of_Agricultural_production_df,
on=['Country', 'Year','Crop_Name'], how='inner')
merged_df = pd.merge(merged_df, Land_Use_df, on=['Country', 'Year'],
how='inner')
merged_df = pd.merge(merged_df,Pesticides_Use_df, on=['Country', 'Year'],
how='inner')
merged_df = pd.merge(merged_df,Cropland_Nutrient_Balance_df,
on=['Country', 'Year'], how='inner')

merged_df.head()

merged_df.shape

merged_df.info()

"""## Feature Engineering"""

year = merged_df['Year']

# Sinusoidal Transformation
merged_df['Year_sin'] = np.sin(2 * np.pi * year / max(year))
merged_df['Year_cos'] = np.cos(2 * np.pi * year / max(year))

# Display the modified DataFrame
print(merged_df[['Year', 'Year_sin', 'Year_cos']].head())

# Emission-related Interaction Features
merged_df['Emission_to_Production_ratio'] = merged_df['Emission_Value_kt']
/ merged_df['Production_t']
merged_df['Emission_to_Area_ratio'] = merged_df['Emission_Value_kt'] /
merged_df['Area harvested_ha']
merged_df['Emission_Value_per_Pesticide'] = merged_df['Emission_Value_kt']
/ merged_df['Pesticide_Value_t']

# Land Use-related Interaction Features
merged_df['Area_to_Production_ratio'] = merged_df['Area harvested_ha'] /
merged_df['Production_t']
merged_df['Area_Value_to_Pesticide_ratio'] = merged_df['Area_Value_1000
ha'] / merged_df['Pesticide_Value_t']

# Pesticide-related Interaction Features
merged_df['Pesticide_to_Production_ratio'] =
merged_df['Pesticide_Value_t'] / merged_df['Production_t']

# Trade related Interaction features
merged_df['Export_to_Production_ratio'] = merged_df['Export Quantity_t'] /
merged_df['Production_t']
merged_df['Import_to_Production_ratio'] = merged_df['Import Quantity_t'] /
merged_df['Production_t']
```

```
# Compute aggregate statistics across different groups

# Mean production by country
country_production_mean =
merged_df.groupby('Country')['Production_t'].mean()
merged_df['Country_Production_Mean'] =
merged_df['Country'].map(country_production_mean)

# Median production by crop name
crop_production_median =
merged_df.groupby('Crop_Name')['Production_t'].median()
merged_df['Crop_Production_Median'] =
merged_df['Crop_Name'].map(crop_production_median)

# Sum of pesticide usage by year
yearly_pesticide_sum =
merged_df.groupby('Year')['Pesticide_Value_t'].sum()
merged_df['Yearly_Pesticide_Sum'] =
merged_df['Year'].map(yearly_pesticide_sum)

# Standard deviation of yield by country
country_yield_std = merged_df.groupby('Country')['Yield_100 g/ha'].std()
merged_df['Country_Yield_Std'] =
merged_df['Country'].map(country_yield_std)

# Total emission value by crop name
crop_emission_total =
merged_df.groupby('Crop_Name')['Emission_Value_kt'].sum()
merged_df['Crop_Emission_Total'] =
merged_df['Crop_Name'].map(crop_emission_total)

# Dropping duplicates
merged_df.drop_duplicates(inplace=True)

merged_df.head()

merged_df.shape

merged_df.to_csv('merged_file.csv', index=False)
```

# Exploratory Data Analysis

```
merged_df.describe()

import matplotlib.pyplot as plt
import seaborn as sns

# List of numerical columns to plot
num_cols = ['Area harvested_ha', 'Production_t', 'Yield_100 g/ha',
            'Gross_Production_Value_1000 USD', 'Area_Value_1000 ha',
'Pesticide_Value_t', 'Fertilizer_Usage_Value_kg/ha',
            'Area_to_Production_ratio', 'Area_Value_to_Pesticide_ratio',
'Export_to_Production_ratio',
            'Country_Production_Mean', 'Crop_Production_Median',
'Yearly_Pesticide_Sum',
            'Country_Yield_Std', 'Crop_Emission_Total']

# Setting up the matplotlib figure
plt.figure(figsize=(18, 20))

# Looping through the numerical columns to create a histogram for each
for i, col in enumerate(num_cols, 1):
    plt.subplot(8, 3, i)  # Adjust the grid size based on the number of
columns
    sns.histplot(merged_df[col], kde=False)
    plt.title(col)
    plt.ylabel('Frequency')
    plt.xlabel(col)

plt.tight_layout()
plt.show()

# Grouping by 'Crop_Name' and summing the 'Production_t'
total_production_per_crop =
merged_df.groupby('Crop_Name')['Production_t'].sum().reset_index()

# Sorting the crops by total production in descending order
total_production_per_crop =
total_production_per_crop.sort_values('Production_t', ascending=False)

import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(8, 6))
sns.barplot(data=total_production_per_crop, x='Crop_Name',
y='Production_t', color='skyblue')
plt.title('Total Production by Crop')
plt.xlabel('Total Production (tons)')
plt.ylabel('Crop Name')
plt.show()

import matplotlib.pyplot as plt
import seaborn as sns

# Group by 'Country' and 'Crop_Name' and sum the production.
country_crop_production = merged_df.groupby(['Country',
'Crop_Name'])['Production_t'].sum().reset_index()

# Sort the production within each country and get the top N crops.
top_crops_per_country = country_crop_production.groupby('Country').apply(
    lambda x: x.sort_values('Production_t', ascending=False).head(5)  #
Taking the top 5 crops as an example
).reset_index(drop=True)

# Get a sorted list of unique countries
countries = top_crops_per_country['Country'].unique()
```

```
# Determine the number of subplot rows we'll need, with up to 4 plots per
row for instance
n_rows = len(countries) // 4 + (len(countries) % 4 > 0)

# Set up the matplotlib figure
fig, axes = plt.subplots(n_rows, 4, figsize=(20, 5 * n_rows))  # Adjust
the size as needed

# Flatten the axes array for easy indexing
axes = axes.flatten()

# Loop through the countries and create a plot for each
for i, country in enumerate(countries):
    # Filter the dataframe for the country
    country_data = top_crops_per_country[top_crops_per_country['Country']
== country]

    # Create the horizontal bar plot
    sns.barplot(x='Crop_Name', y='Production_t', data=country_data,
ax=axes[i], palette='coolwarm')
    axes[i].set_title(f'Top 5 Crops in {country}')
    axes[i].set_xlabel('Crop Name')
    axes[i].set_ylabel('Total Production (tons)')

# Hide any unused subplots
for j in range(i + 1, len(axes)):
    axes[j].set_visible(False)

# Adjust the layout
plt.tight_layout()
plt.show()

# Calculate the production per hectare for each row in the dataframe
merged_df['Production_per_ha'] = merged_df['Production_t'] /
merged_df['Area harvested_ha']

# Group by 'Crop_Name' and calculate the mean production per hectare for
each crop
productivity_per_crop =
merged_df.groupby('Crop_Name')['Production_per_ha'].mean().reset_index()

# Sort the crops by productivity in descending order
productivity_per_crop_sorted =
productivity_per_crop.sort_values('Production_per_ha', ascending=False)

plt.figure(figsize=(8, 5))
sns.barplot(x='Crop_Name', y='Production_per_ha',
data=productivity_per_crop_sorted, color='lightblue')
plt.title('Average Production (tons per hectare)')
plt.xlabel('Average Productivity per Hectare by Crop')
plt.ylabel('Crop Name')
plt.show()

# Calculate the production per hectare for each row in the dataframe
```

```
merged_df['Production_per_ha'] = merged_df['Production_t'] /
merged_df['Area harvested_ha']

# Group by 'Country' and calculate the mean production per hectare for
each country
production_per_ha_by_country =
merged_df.groupby('Country')['Production_per_ha'].mean().reset_index()

# Sort the countries by mean production per hectare in descending order
production_per_ha_by_country_sorted =
production_per_ha_by_country.sort_values('Production_per_ha',
ascending=False)

plt.figure(figsize=(8, 5))
sns.barplot(x='Country', y='Production_per_ha',
data=production_per_ha_by_country_sorted.head(20), color='green')  #
Adjust to display the number of top countries you'd like to show
plt.title('Top Countries by Average Crop Production per Hectare')
plt.xlabel('Country')
plt.ylabel('Average Production per Hectare (tons/ha)')
plt.show()

"""## Export and Import Analysis"""

trade_value = merged_df.groupby('Country')[['Export Value_1000 USD',
'Import Value_1000 USD']].sum().reset_index()


plt.figure(figsize=(8, 5))

# Setting the positions of the bars
barWidth = 0.4
r1 = range(len(trade_value))
r2 = [x + barWidth for x in r1]

# Create the bars for exports
plt.bar(r1, trade_value['Export Value_1000 USD'], color='blue',
width=barWidth, edgecolor='grey', label='Export Value')

# Create the bars for imports right next to the export bars
plt.bar(r2, trade_value['Import Value_1000 USD'], color='red',
width=barWidth, edgecolor='grey', label='Import Value')

# Add labels to the x-axis at the bar centers
plt.xlabel('Country', fontweight='bold')
plt.xticks([r + barWidth / 2 for r in range(len(trade_value))],
trade_value['Country'], rotation=90)

plt.ylabel('Value (1000 USD)')
plt.title('Export vs Import Value for Each Country')
plt.legend()
plt.show()

# Calculate the total export and import quantity for each country
```

```
trade_balance = merged_df.groupby('Country')[['Export Quantity_t', 'Import
Quantity_t']].sum().reset_index()

# Now let's plot the grouped bar chart
plt.figure(figsize=(8, 5))

# Create the bars for exports
plt.bar(r1, trade_balance['Export Quantity_t'], color='blue',
width=barWidth, edgecolor='grey', label='Export Quantity')

# Create the bars for imports
plt.bar(r2, trade_balance['Import Quantity_t'], color='red',
width=barWidth, edgecolor='grey', label='Import Quantity')

# Add labels to the x-axis at the bar centers
plt.xlabel('Country', fontweight='bold')
plt.xticks([r + barWidth for r in range(len(trade_balance))],
trade_balance['Country'], rotation=90)

plt.ylabel('Quantity (tons)')
plt.yscale('log')  # Apply logarithmic scale
plt.title('Export vs Import Quantity for Each Country (Logarithmic
Scale)')

# Create legend & Show graphic
plt.legend()
plt.show()

# Calculate the unit price for exports and imports
merged_df['Export_Unit_Price_USD_per_ton'] = (merged_df['Export Value_1000
USD'] * 1000) / merged_df['Export Quantity_t']
merged_df['Import_Unit_Price_USD_per_ton'] = (merged_df['Import Value_1000
USD'] * 1000) / merged_df['Import Quantity_t']

# Handle divisions by zero or missing data if any
merged_df['Export_Unit_Price_USD_per_ton'].fillna(0, inplace=True)
merged_df['Import_Unit_Price_USD_per_ton'].fillna(0, inplace=True)
merged_df.replace([np.inf, -np.inf], 0, inplace=True)

# Now we can calculate the average unit price for each country by
averaging these values
country_unit_prices = merged_df.groupby('Country').agg({
    'Export_Unit_Price_USD_per_ton': 'mean',
    'Import_Unit_Price_USD_per_ton': 'mean'
}).reset_index()

plt.figure(figsize=(8, 5))

# Setting the positions of the bars
barWidth = 0.35
r1 = range(len(country_unit_prices))
r2 = [x + barWidth for x in r1]

# Create the bars for export unit prices
```

```
plt.bar(r1, country_unit_prices['Export_Unit_Price_USD_per_ton'],
color='blue', width=barWidth, edgecolor='grey', label='Export Unit Price')

# Create the bars for import unit prices
plt.bar(r2, country_unit_prices['Import_Unit_Price_USD_per_ton'],
color='red', width=barWidth, edgecolor='grey', label='Import Unit Price')

# Add labels to the x-axis at the bar centers
plt.xlabel('Country', fontweight='bold')
plt.xticks([r + barWidth for r in range(len(country_unit_prices))],
country_unit_prices['Country'], rotation=90)

plt.ylabel('Unit Price (USD/ton)')
plt.title('Average Unit Price for Export vs Import per Country')
plt.legend()
plt.show()

# Calculate the total export and import values for each crop type
crop_trade_values = merged_df.groupby('Crop_Name').agg({
    'Export Value_1000 USD': 'sum',
    'Import Value_1000 USD': 'sum'
}).reset_index()

# Sort the crops by export value to see which are the highest
crop_trade_values.sort_values('Export Value_1000 USD', ascending=False,
inplace=True)

# You could visualize the top N crops by export value to see what is
contributing to export values
plt.figure(figsize=(8, 5))
sns.barplot(x='Crop_Name', y='Export Value_1000 USD',
data=crop_trade_values.head(20), color='blue', label='Export Value')
sns.barplot(x='Crop_Name', y='Import Value_1000 USD',
data=crop_trade_values.head(20), color='red', alpha=0.5, label='Import
Value')
plt.xticks(rotation=90)
plt.title('Crops by Export vs Import Value')
plt.legend()
plt.show()

# Calculate the trade balance by subtracting the import value from the
export value
trade_value['Trade_Balance_1000USD'] = trade_value['Export Value_1000
USD'] - trade_value['Import Value_1000 USD']

# Now let's plot the trade balance for each country using values
plt.figure(figsize=(8, 5))
sns.barplot(x='Country', y='Trade_Balance_1000USD', data=trade_value,
palette='vlag')
plt.xticks(rotation=90)
plt.title('Net Trade Balance of Agricultural Products per Country
(Value)')
plt.xlabel('Country')
plt.ylabel('Trade Balance (1000 USD)')
plt.show()

# Calculate the average yield for each crop and country
average_yield = merged_df.groupby(['Country', 'Crop_Name'])['Yield_100
g/ha'].mean().reset_index()

# Sort the results by yield in descending order
average_yield = average_yield.sort_values('Yield_100 g/ha',
ascending=False)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 10))
sns.scatterplot(data=average_yield, x='Country', y='Yield_100 g/ha',
hue='Crop_Name', size='Yield_100 g/ha', sizes=(20, 200))
plt.xticks(rotation=90)
plt.title('Average Crop Yield per Country')
plt.xlabel('Country')
plt.ylabel('Yield (100g per hectare)')
plt.legend(title='Crop Name', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()

# Calculate the efficiency of pesticide usage
merged_df['Pesticide_Efficiency'] = merged_df['Yield_100 g/ha'] /
merged_df['Pesticide_Value_t']

# Calculate the efficiency of fertilizer usage
merged_df['Fertilizer_Efficiency'] = merged_df['Yield_100 g/ha'] /
merged_df['Fertilizer_Usage_Value_kg/ha']

# Handle potential divisions by zero or missing data
merged_df.replace([np.inf, -np.inf], np.nan, inplace=True)
merged_df.fillna(0, inplace=True)

# Calculate correlation matrix for yield, pesticide usage, and fertilizer
usage
correlation_matrix = merged_df[['Yield_100 g/ha', 'Pesticide_Value_t',
'Fertilizer_Usage_Value_kg/ha']].corr()

# Use seaborn to visualize the correlation matrix
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(8, 4))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix for Yield and Input Usage')
plt.show()

# Calculate emissions efficiency
merged_df['Emissions_per_ton'] = merged_df['Emission_Value_kt'] /
merged_df['Production_t']
```

```
# Replace infinite values with NaN and then fill or remove them
merged_df.replace([np.inf, -np.inf], np.nan, inplace=True)
merged_df.dropna(subset=['Emissions_per_ton'], inplace=True)

# Calculate correlation matrix for production and emissions
correlation_matrix_prod_emiss = merged_df[['Production_t',
'Emission_Value_kt']].corr()

# Visualize the correlation matrix
plt.figure(figsize=(8, 6))
sns.heatmap(correlation_matrix_prod_emiss, annot=True, cmap='coolwarm',
center=0)
plt.title('Correlation Matrix for Production and Emissions')
plt.show()

# Group data by crop or country to see average emissions per ton
average_emissions_per_crop =
merged_df.groupby('Crop_Name')['Emissions_per_ton'].mean().reset_index()

# Sort and visualize the top crops by emissions efficiency
average_emissions_per_crop_sorted =
average_emissions_per_crop.sort_values('Emissions_per_ton',
ascending=False)

plt.figure(figsize=(8, 5))
sns.barplot(x='Emissions_per_ton', y='Crop_Name',
data=average_emissions_per_crop_sorted)
plt.title('Emissions Efficiency per Crop')
plt.xlabel('Emissions per ton (kt/ton)')
plt.ylabel('Crop Name')
plt.show()

# Assuming that 'Fertilizer_Usage_Value_kg/ha' represents the amount of
fertilizer applied per hectare
merged_df['Nutrient_Use_Efficiency'] = merged_df['Yield_100 g/ha'] /
merged_df['Fertilizer_Usage_Value_kg/ha']

# Example for a bar chart visualization
plt.figure(figsize=(8, 5))
sns.barplot(x='Crop_Name', y='Nutrient_Use_Efficiency', data=merged_df)
plt.title('Efficiency (Yield per kg/ha of Fertilizer)')
plt.xlabel('Nutrient Use Efficiency by Crop)')
plt.ylabel('Crop Name')
plt.show()

# For example, to look at production trends:
production_trends =
merged_df.groupby('Year')['Production_t'].sum().reset_index()

# For emissions trends:
emission_trends =
merged_df.groupby('Year')['Emission_Value_kt'].sum().reset_index()

import matplotlib.pyplot as plt
```

```
# Line chart for production trends
plt.figure(figsize=(10, 5))
plt.plot(production_trends['Year'], production_trends['Production_t'],
marker='o')
plt.title('Production Trends Over Time')
plt.xlabel('Year')
plt.ylabel('Total Production (tons)')
plt.grid(True)
plt.show()

# Line chart for emissions trends
plt.figure(figsize=(10, 5))
plt.plot(emission_trends['Year'], emission_trends['Emission_Value_kt'],
marker='o', color='red')
plt.title('Emission Trends Over Time')
plt.xlabel('Year')
plt.ylabel('Emissions (kt)')
plt.grid(True)
plt.show()

# Calculate the percentage of land used for each crop each year and its
impact on production.
# We will create a new feature 'Land_Use_Percentage' that represents the
percentage of land used for a particular crop each year.

# First, we find out the total land available each year (assuming it's the
sum of the area harvested for all crops).
total_land_each_year = merged_df.groupby('Year')['Area
harvested_ha'].sum().reset_index().rename(columns={'Area harvested_ha':
'Total_Area_ha'})


# Merge this total land data back into the original dataframe
merged_with_total_land = pd.merge(merged_df, total_land_each_year,
on='Year')

# Calculate the percentage of land used for each crop each year
merged_with_total_land['Land_Use_Percentage'] =
(merged_with_total_land['Area harvested_ha'] /
merged_with_total_land['Total_Area_ha']) * 100

# Group by year and crop to see the impact on production
land_use_impact_on_production = merged_with_total_land.groupby(['Year',
'Crop_Name']).agg({
    'Land_Use_Percentage': 'mean',  # Average percentage of land used for
the crop that year
    'Production_t': 'sum'           # Total production for the crop that
year
}).reset_index()

# Plotting the relationship between land use percentage and production
plt.figure(figsize=(14, 7))
```

```
sns.scatterplot(data=land_use_impact_on_production,
x='Land_Use_Percentage', y='Production_t', hue='Crop_Name')
plt.title('Impact of Land Use Percentage on Crop Production')
plt.xlabel('Land Use Percentage (%)')
plt.ylabel('Total Production (tons)')
plt.legend(title='Crop Name', bbox_to_anchor=(1.05, 1), loc=2)
plt.grid(True)
plt.show()

# Descriptive statistics
descriptive_stats = merged_df.describe()

# Calculate the IQR for each numeric feature
Q1 = merged_df.quantile(0.25)
Q3 = merged_df.quantile(0.75)
IQR = Q3 - Q1

# Identify outliers for each feature
is_outlier = (merged_df < (Q1 - 1.5 * IQR)) | (merged_df > (Q3 + 1.5 *
IQR))

import matplotlib.pyplot as plt
import seaborn as sns

# Assuming merged_df is your DataFrame and it has been preprocessed
correctly
numeric_features = merged_df.select_dtypes(include=['float64', 'int64'])

# Create violin plots for each numeric feature
for column in numeric_features.columns:
    plt.figure(figsize=(8, 4))
    sns.violinplot(data=merged_df, x=column)
    plt.title(f'Violin Plot for {column}')
    plt.show()

for column in is_outlier:
    if (is_outlier[column] == True).any():
        print(column, len(column))
```

# Model Building

```
merged_df = pd.read_csv("merged_file.csv")
numeric_cols = merged_df.select_dtypes(include=['float64',
'int64']).columns

# Define the percentile thresholds
lower_percentile = 0.02
upper_percentile = 0.98

# Capping the outliers for all numerical features in the DataFrame
for col in numeric_cols:
    # Compute the 1st and 99th percentiles
    lower_bound = merged_df[col].quantile(lower_percentile)
    upper_bound = merged_df[col].quantile(upper_percentile)
    # Cap values below the 1st percentile to the 1st percentile value
    # Cap values above the 99th percentile to the 99th percentile value
    merged_df[col] = np.where(merged_df[col] < lower_bound, lower_bound,
merged_df[col])
    merged_df[col] = np.where(merged_df[col] > upper_bound, upper_bound,
merged_df[col])

merged_df.head()

# List of relevant features
relevant_features = [
    'Country', 'Crop_Name', 'Year', 'Area harvested_ha', 'Production_t',
'Yield_100 g/ha',
    'Export Quantity_t', 'Export Value_1000 USD', 'Import Quantity_t',
'Import Value_1000 USD',
    'Emission_type', 'Emission_Value_kt', 'Gross_Production_Value_1000
USD', 'Area_Value_1000 ha',
    'Pesticide_Type', 'Pesticide_Value_t', 'Fertilizer_Usage_Value_kg/ha',
    'Emission_to_Production_ratio', 'Emission_to_Area_ratio',
'Emission_Value_per_Pesticide',
    'Area_to_Production_ratio'
]

# Create a new DataFrame with only the selected features
data_relevant = merged_df[relevant_features]

data_relevant.head()

# Select only numerical features for the correlation matrix
numerical_features = data_relevant.select_dtypes(include=['float64',
'int64']).columns
correlation_matrix = data_relevant[numerical_features].corr()

# Plot the correlation matrix using a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm',
cbar=True)
plt.title('Correlation Matrix of Numerical Features')
plt.show()

# Dropping highly correlated features based on heatmap analysis
columns_to_drop = ['Area harvested_ha', 'Export Quantity_t', 'Import
Value_1000 USD', 'Gross_Production_Value_1000 USD']
data_reduced = data_relevant.drop(columns=columns_to_drop)

# Function to calculate VIF for each feature
def calculate_vif(df):
    vif_data = pd.DataFrame()
    vif_data['feature'] = df.columns
    vif_data['VIF'] = [variance_inflation_factor(df.values, i) for i in
range(df.shape[1])]
    return vif_data
```

```
#recalculate the VIF for the reduced dataset
vif_scores_reduced =
calculate_vif(data_reduced.select_dtypes(include=['float64', 'int64',
'uint8']))
print(vif_scores_reduced.sort_values('VIF', ascending=False))

# Dropping the specified high VIF features
columns_to_drop_more = ['Yield_100 g/ha', 'Emission_to_Production_ratio',
'Emission_to_Area_ratio']
data_reduced_further = data_reduced.drop(columns=columns_to_drop_more)

data_reduced_further.head()

vif_scores_updated =
calculate_vif(data_reduced_further.select_dtypes(include=['float64',
'int64', 'uint8']))
print(vif_scores_updated.sort_values('VIF', ascending=False))

# Define target variables
target_vars = ['Production_t', 'Export Value_1000 USD']

# Define numerical and categorical columns, excluding the target columns
numerical_cols = data_reduced_further.select_dtypes(include=['int64',
'float64']).columns.difference(target_vars).tolist()
categorical_cols =
data_reduced_further.select_dtypes(include=['object']).columns.tolist()

# Create the preprocessing pipeline for numerical data
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())  # Standardize numerical features
])

# Create the preprocessing pipeline for categorical data
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))  # Encode
categorical features
])

# Combine all elements into a large transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Split data into training and testing sets based on the year
train = data_reduced_further[data_reduced_further['Year'] <= 2019]
test = data_reduced_further[data_reduced_further['Year'] > 2019]

# Drop the 'Year' column if it's no longer needed for modeling
X_train = train.drop(['Production_t', 'Export Value_1000 USD'], axis=1)
y_train_prod = train['Production_t']
y_train_export = train['Export Value_1000 USD']
```

```
X_test = test.drop(['Production_t', 'Export Value_1000 USD'], axis=1)
y_test_prod = test['Production_t']
y_test_export = test['Export Value_1000 USD']

"""## Linear Regression"""

# Create the full pipeline including the preprocessor and the Linear
Regression model
pipeline_lr_prod = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

# Training the pipeline on the training data
pipeline_lr_prod.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_lr_prod.predict(X_test)

# Evaluating the model
lr_mae = mean_absolute_error(y_test_prod, y_pred_prod)
lr_mae_percentage = (lr_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(lr_mae_percentage))

# Mean Squared Error (MSE)
lr_mse = mean_squared_error(y_test_prod, y_pred_prod)
lr_mse_percentage = (lr_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(lr_mse_percentage))

# Root Mean Squared Error (RMSE)
lr_rmse = np.sqrt(lr_mse)
lr_rmse_percentage = (lr_rmse / np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(lr_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

"""## Lasso Regression"""

pipeline_lasso = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', Lasso(alpha=1.0, random_state=42))
])

# Training the pipeline on the training data for Production
pipeline_lasso.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_lasso.predict(X_test)

# Evaluating the model

lasso_mae = mean_absolute_error(y_test_prod, y_pred_prod)
```

```
lasso_mae_percentage = (lasso_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(lasso_mae_percentage))

# Mean Squared Error (MSE)
lasso_mse = mean_squared_error(y_test_prod, y_pred_prod)
lasso_mse_percentage = (lasso_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(lasso_mse_percentage))

# Root Mean Squared Error (RMSE)
lasso_rmse = np.sqrt(lasso_mse)
lasso_rmse_percentage = (lasso_rmse / np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(lasso_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

"""## Ridge Regression"""

pipeline_ridge = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', Ridge(alpha=1.0, random_state=42))
])

# Training the pipeline on the training data for Production
pipeline_ridge.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_ridge.predict(X_test)

# Evaluating the model
ridge_mae = mean_absolute_error(y_test_prod, y_pred_prod)
ridge_mae_percentage = (ridge_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(ridge_mae_percentage))

# Mean Squared Error (MSE)
ridge_mse = mean_squared_error(y_test_prod, y_pred_prod)
ridge_mse_percentage = (ridge_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(ridge_mse_percentage))

# Root Mean Squared Error (RMSE)
ridge_rmse = np.sqrt(ridge_mse)
ridge_rmse_percentage = (ridge_rmse / np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(ridge_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

"""## SVM Regressor"""

pipeline_svr = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', SVR(kernel='poly'))
])

# Training the pipeline on the training data for Production
pipeline_svr.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_svr.predict(X_test)

# Evaluating the model

svr_mae = mean_absolute_error(y_test_prod, y_pred_prod)
svr_mae_percentage = (svr_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(svr_mae_percentage))

# Mean Squared Error (MSE)
svr_mse = mean_squared_error(y_test_prod, y_pred_prod)
svr_mse_percentage = (svr_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(svr_mse_percentage))

# Root Mean Squared Error (RMSE)
svr_rmse = np.sqrt(svr_mse)
svr_rmse_percentage = (svr_rmse / np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(svr_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

"""## XGBoost Regressor"""

pipeline_xgb = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(objective='reg:squarederror',
random_state=42))
])

# Training the pipeline on the training data for Production
pipeline_xgb.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_xgb.predict(X_test)

xgb_mae = mean_absolute_error(y_test_prod, y_pred_prod)
xgb_mae_percentage = (xgb_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(xgb_mae_percentage))

# Mean Squared Error (MSE)
xgb_mse = mean_squared_error(y_test_prod, y_pred_prod)
xgb_mse_percentage = (xgb_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(xgb_mse_percentage))

# Root Mean Squared Error (RMSE)
xgb_rmse = np.sqrt(xgb_mse)
xgb_rmse_percentage = (xgb_rmse / np.mean(y_test_prod)) * 100
```

```
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(xgb_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

# Define the parameter grid
param_grid = {
    'regressor__n_estimators': [100, 200],
    'regressor__max_depth': [3, 5, 7],
    'regressor__learning_rate': [0.01, 0.1],
    'regressor__subsample': [0.7, 0.9],
    'regressor__colsample_bytree': [0.7, 0.9]
}

# Setup the grid search
grid_search = GridSearchCV(pipeline_xgb, param_grid, cv=3,
scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train_prod)

print("Best parameters:", grid_search.best_params_)
print("Best cross-validation score: ", np.sqrt(-grid_search.best_score_))

# Update pipeline with the best parameters
pipeline_xgb_optimized = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', XGBRegressor(
        objective='reg:squarederror',
        colsample_bytree=0.9,
        learning_rate=0.1,
        max_depth=5,
        n_estimators=200,
        subsample=0.9,
        random_state=42))
])

# Training the optimized pipeline on the training data
pipeline_xgb_optimized.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_xgb_optimized.predict(X_test)

xgb_optimized_mae = mean_absolute_error(y_test_prod, y_pred_prod)
xgb_optimized_mae_percentage = (xgb_optimized_mae / np.mean(y_test_prod))
* 100
print("Mean Absolute Error (MAE):
{:.2f}%".format(xgb_optimized_mae_percentage))

# Mean Squared Error (MSE)
xgb_optimized_mse = mean_squared_error(y_test_prod, y_pred_prod)
xgb_optimized_mse_percentage = (xgb_optimized_mse / (np.mean(y_test_prod)
** 2)) * 100
print("Mean Squared Error (MSE):
{:.2f}%".format(xgb_optimized_mse_percentage))
```

```
# Root Mean Squared Error (RMSE)
xgb_optimized_rmse = np.sqrt(xgb_optimized_mse)
xgb_optimized_rmse_percentage = (xgb_optimized_rmse /
np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(xgb_optimized_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

"""## Random Forest"""

pipeline_rf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(n_estimators=100, random_state=42,
max_depth= 6))
])

# Training the pipeline on the training data for Production
pipeline_rf.fit(X_train, y_train_prod)

# Predicting on the test data
y_pred_prod = pipeline_rf.predict(X_test)

# Evaluating the model

rf_mae = mean_absolute_error(y_test_prod, y_pred_prod)
rf_mae_percentage = (rf_mae / np.mean(y_test_prod)) * 100
print("Mean Absolute Error (MAE): {:.2f}%".format(rf_mae_percentage))

# Mean Squared Error (MSE)
rf_mse = mean_squared_error(y_test_prod, y_pred_prod)
rf_mse_percentage = (rf_mse / (np.mean(y_test_prod) ** 2)) * 100
print("Mean Squared Error (MSE): {:.2f}%".format(rf_mse_percentage))

# Root Mean Squared Error (RMSE)
rf_rmse = np.sqrt(rf_mse)
rf_rmse_percentage = (rf_rmse / np.mean(y_test_prod)) * 100
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(rf_rmse_percentage))

r2 = r2_score(y_test_prod, y_pred_prod)
print("R2 Score:", r2)

# Calculate residuals
residuals = y_test_prod - y_pred_prod

# Plotting the histogram of residuals
plt.figure(figsize=(6, 4))
plt.hist(residuals, bins=30, alpha=0.5, color='g')
plt.title('Histogram of Prediction Errors')
plt.xlabel('Prediction Error')
plt.ylabel('Frequency')
plt.show()

# Boxplot of residuals
plt.figure(figsize=(6, 4))
plt.boxplot(residuals, vert=False)
plt.title('Boxplot of Prediction Errors')
plt.xlabel('Prediction Error')
plt.show()

plt.figure(figsize=(10, 6))
stats.probplot(residuals, dist="norm", plot=plt)
plt.title('Q-Q Plot')
plt.show()

plt.figure(figsize=(10, 6))
plt.scatter(y_pred_prod, residuals)
plt.title('Residuals vs. Predicted Values')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
plt.axhline(y=0, color='r', linestyle='--')
plt.show()

df_analysis = pd.DataFrame({
  'predicted_values': y_pred_prod,
  'residuals': residuals
})

# Define quantile-based ranges for predicted values
df_analysis['predicted_range'] = pd.qcut(df_analysis['predicted_values'],
4)

# Calculate summary statistics for each range
summary_stats =
df_analysis.groupby('predicted_range')['residuals'].agg(['mean', 'median',
'std', 'count'])

# Plot the distribution of residuals for each range
fig, axes = plt.subplots(nrows=4, figsize=(8, 12))
for (range_val, subset), ax in zip(df_analysis.groupby('predicted_range'),
axes):
    ax.hist(subset['residuals'], bins=10, alpha=0.7)
    ax.set_title(f'Residuals for Predicted Range: {range_val}')
    ax.set_xlabel('Residuals')
    ax.set_ylabel('Frequency')

plt.tight_layout()
plt.show()

# Identifying high error instances
high_error_threshold = residuals.abs().mean() + 1.5 *
residuals.abs().std()  # Using mean + 1.5*STD as threshold
high_error_indices = residuals[residuals.abs() >
high_error_threshold].index
high_error_data = X_test.loc[high_error_indices]
```

```
# Analyzing features of high error instances
feature_analysis = high_error_data.describe(include='all')
display(feature_analysis)

def calculate_metrics(y_t, y_p):
    model_mae = mean_absolute_error(y_t, y_p)
    model_mae_percentage = (model_mae / np.mean(y_t)) * 100

    # Mean Squared Error (MSE)
    model_mse = mean_squared_error(y_t, y_p)
    model_mse_percentage = (model_mse / (np.mean(y_t) ** 2)) * 100

    # Root Mean Squared Error (RMSE)
    model_rmse = np.sqrt(model_mse)
    model_rmse_percentage = (model_rmse / np.mean(y_t)) * 100

    model_r2 = r2_score(y_t, y_p)

    return model_mae_percentage, model_mse_percentage,
model_rmse_percentage, model_r2

"""## Residual Calculation and Grouping"""

df_test = []

df_test = pd.DataFrame({
    'actual': y_test_prod,
    'predicted': y_pred_prod
})

df_test = pd.concat([X_test, df_test], axis = 1)
df_test.shape
df_test['residuals'] = df_test['actual'] - df_test['predicted']
# Group residuals into bins (for example, by quartiles)
df_test['residual_group'] = pd.qcut(df_test['residuals'], 4, labels=['Q1',
'Q2', 'Q3', 'Q4'])

# Plotting the distribution of residuals within each bin
plt.figure(figsize=(12, 6))
sns.boxplot(x='residual_group', y='residuals', data=df_test)
plt.title('Distribution of Residuals by Group')
plt.xlabel('Residual Group')
plt.ylabel('Residuals')
plt.grid(True)
plt.show()

"""## Extract Data Points from Q1 and Q4:"""

# Filter data for Q1 and Q4 groups
df_q1 = df_test[df_test['residual_group'] == 'Q1']
df_q4 = df_test[df_test['residual_group'] == 'Q4']

# Descriptive statistics for Q1
```

```
print("Descriptive Statistics for Q1:")
display(df_q1.describe())

# Descriptive statistics for Q4
print("Descriptive Statistics for Q4:")
display(df_q4.describe())

# Correlation matrix for the Q1 group
corr_q1 = df_test[df_test['residual_group'] == 'Q1'].corr()
print("Correlation Matrix for Q1:")
display(corr_q1)

# Correlation matrix for the Q4 group
corr_q4 = df_test[df_test['residual_group'] == 'Q4'].corr()
print("Correlation Matrix for Q4:")
display(corr_q4)

# Heatmap visualization for Q1
plt.figure(figsize=(8, 6))
sns.heatmap(corr_q1.drop(['predicted', 'residuals'],
axis=0).drop(['actual', 'predicted', 'residuals'], axis=1),
            annot=True,
            cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap for Q1')
plt.show()

# Heatmap visualization for Q4
plt.figure(figsize=(10, 8))
sns.heatmap(corr_q4.drop(['predicted', 'residuals'],
axis=0).drop(['actual', 'predicted', 'residuals'], axis=1),
            annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Heatmap for Q4')
plt.show()


y_pred_train = pipeline_rf.predict(X_train)

mae_rf_train, mse_rf_train, rmse_rf_train, r2_rf_train =
calculate_metrics(y_train_prod, y_pred_train)


print(f"Segment 1 - MAE: {mae_rf_train}, MSE: {mse_rf_train}, RMSE:
{rmse_rf_train}, R2: {r2_rf_train}")

residual_train = y_train_prod - y_pred_train

df_test_train = pd.DataFrame({
    'actual': y_train_prod,
    'predicted': y_pred_train
})

df_test_train = pd.concat([X_train, df_test_train], axis = 1)

df_test_train.shape
```

```
df_test_train['residuals'] = df_test_train['actual'] -
df_test_train['predicted']

# Group residuals into bins (for example, by quartiles)
df_test_train['residual_group'] = pd.qcut(df_test_train['residuals'], 2,
labels=['Q1', 'Q2'])

# Plotting the distribution of residuals within each bin
plt.figure(figsize=(6, 5))
sns.boxplot(x='residual_group', y='residuals', data=df_test_train)
plt.title('Distribution of Residuals by Group')
plt.xlabel('Residual Group')
plt.ylabel('Residuals')
plt.grid(True)
plt.show()
```

# Model Optimization

```
"""
Based on the EDA, performing the following actions
"""

merged_df = pd.read_csv("merged_file.csv")

# Assuming 'merged_df' is your preprocessed DataFrame and you have already
identified numeric_cols
numeric_cols = merged_df.select_dtypes(include=['float64',
'int64']).columns

# Define the percentile thresholds
lower_percentile = 0.02
upper_percentile = 0.98

# Capping the outliers for all numerical features in the DataFrame
for col in numeric_cols:
    # Compute the 1st and 99th percentiles
    lower_bound = merged_df[col].quantile(lower_percentile)
    upper_bound = merged_df[col].quantile(upper_percentile)

    # Cap values below the 1st percentile to the 1st percentile value
    # Cap values above the 99th percentile to the 99th percentile value
    merged_df[col] = np.where(merged_df[col] < lower_bound, lower_bound,
merged_df[col])
    merged_df[col] = np.where(merged_df[col] > upper_bound, upper_bound,
merged_df[col])

merged_df.head()

# List of relevant features
relevant_features = [
    'Country', 'Crop_Name', 'Year', 'Area harvested_ha', 'Production_t',
'Yield_100 g/ha',
    'Export Quantity_t', 'Export Value_1000 USD', 'Import Quantity_t',
'Import Value_1000 USD',
    'Emission_type', 'Emission_Value_kt', 'Gross_Production_Value_1000
USD', 'Area_Value_1000 ha',
    'Pesticide_Type', 'Pesticide_Value_t', 'Fertilizer_Usage_Value_kg/ha',
    'Emission_to_Production_ratio', 'Emission_to_Area_ratio',
'Emission_Value_per_Pesticide',
    'Area_to_Production_ratio'
]

# Create a new DataFrame with only the selected features
data_relevant = merged_df[relevant_features]

data_relevant.head()

# Select only numerical features for the correlation matrix
numerical_features = data_relevant.select_dtypes(include=['float64',
'int64']).columns
correlation_matrix = data_relevant[numerical_features].corr()

# Plot the correlation matrix using a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm',
cbar=True)
plt.title('Correlation Matrix of Numerical Features')
plt.show()

# Dropping highly correlated features based on heatmap analysis
columns_to_drop = ['Area harvested_ha', 'Export Quantity_t', 'Import
Value_1000 USD', 'Gross_Production_Value_1000 USD']
data_reduced = data_relevant.drop(columns=columns_to_drop)

# Function to calculate VIF for each feature
def calculate_vif(df):
    vif_data = pd.DataFrame()
    vif_data['feature'] = df.columns
    vif_data['VIF'] = [variance_inflation_factor(df.values, i) for i in
range(df.shape[1])]
    return vif_data

#recalculate the VIF for the reduced dataset
vif_scores_reduced =
calculate_vif(data_reduced.select_dtypes(include=['float64', 'int64',
'uint8']))
print(vif_scores_reduced.sort_values('VIF', ascending=False))

# Dropping the specified high VIF features
columns_to_drop_more = ['Yield_100 g/ha', 'Emission_to_Production_ratio',
'Emission_to_Area_ratio']
data_reduced_further = data_reduced.drop(columns=columns_to_drop_more)

data_reduced_further.head()
```

```
vif_scores_updated =
calculate_vif(data_reduced_further.select_dtypes(include=['float64',
'int64', 'uint8']))
print(vif_scores_updated.sort_values('VIF', ascending=False))

# Define target variables
target_vars = ['Production_t']

# Define numerical and categorical columns, excluding the target columns
numerical_cols = data_reduced_further.select_dtypes(include=['int64',
'float64']).columns.difference(target_vars).tolist()
categorical_cols =
data_reduced_further.select_dtypes(include=['object']).columns.tolist()

# Create the preprocessing pipeline for numerical data
numeric_transformer = Pipeline(steps=[
    ('scaler', StandardScaler())  # Standardize numerical features
])

# Create the preprocessing pipeline for categorical data
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore'))  # Encode
categorical features
])

# Combine all elements into a large transformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])

# Split data into training and testing sets based on the year
train = data_reduced_further[data_reduced_further['Year'] <= 2019]
test = data_reduced_further[data_reduced_further['Year'] > 2019]

# Drop the 'Year' column if it's no longer needed for modeling
X_train = train.drop(['Production_t'], axis=1)
y_train = train['Production_t']

X_test = test.drop(['Production_t'], axis=1)
y_test = test['Production_t']

"""## Random Forest"""

def calculate_metrics(y_t, y_p):
    model_mae = mean_absolute_error(y_t, y_p)
    model_mae_percentage = (model_mae / np.mean(y_t)) * 100

    # Mean Squared Error (MSE)
    model_mse = mean_squared_error(y_t, y_p)
    model_mse_percentage = (model_mse / (np.mean(y_t) ** 2)) * 100

    # Root Mean Squared Error (RMSE)
    model_rmse = np.sqrt(model_mse)
    model_rmse_percentage = (model_rmse / np.mean(y_t)) * 100

    model_r2 = r2_score(y_t, y_p)

    return model_mae_percentage, model_mse_percentage,
model_rmse_percentage, model_r2

pipeline_rf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(n_estimators=100, random_state=42,
max_depth=8))
])

# Training the pipeline on the training data for Production
pipeline_rf.fit(X_train, y_train)

# Predicting on the train data
y_pred_train = pipeline_rf.predict(X_train)

rf_mae_train, rf_mse_train, rf_rmse_train, rf_r2_train =
calculate_metrics(y_train, y_pred_train)

print("Mean Absolute Error (MAE): {:.2f}%".format(rf_mae_train))
print("Mean Squared Error (MSE): {:.2f}%".format(rf_mse_train))
print("Root Mean Squared Error (RMSE): {:.2f}%".format(rf_rmse_train))
print("R2 Score:", rf_r2_train)

y_pred_test = pipeline_rf.predict(X_test)

rf_mae_test, rf_mse_test, rf_rmse_test, rf_r2_test =
calculate_metrics(y_test, y_pred_test)

print("Mean Absolute Error (MAE): {:.2f}%".format(rf_mae_test))
print("Mean Squared Error (MSE): {:.2f}%".format(rf_mse_test))
print("Root Mean Squared Error (RMSE): {:.2f}%".format(rf_rmse_test))
print("R2 Score:", rf_r2_test)

# Boxplot for target variable
plt.figure(figsize=(4, 4))
sns.boxplot(data_reduced_further['Production_t'])
plt.title('Box Plot of Target Variable')
plt.show()

# Scatter plot for target variable against another feature that you
suspect might be influencing outliers
plt.figure(figsize=(8, 4))
plt.scatter(data_reduced_further['Emission_Value_kt'],
data_reduced_further['Production_t'])
plt.xlabel('Feature')
plt.ylabel('Target')
plt.title('Scatter Plot of Feature vs. Target')
plt.show()
```

```
# Calculate IQR for the target variable
Q1 = data_reduced_further['Production_t'].quantile(0.05)
Q3 = data_reduced_further['Production_t'].quantile(0.95)
IQR = Q3 - Q1

# Define thresholds for outliers
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filtering the data
filtered_df = data_reduced_further[(data_reduced_further['Production_t']
>= lower_bound) & (data_reduced_further['Production_t'] <= upper_bound)]

print(f"Original data points: {len(data_reduced_further)}")
print(f"Data points after outlier removal: {len(filtered_df)}")

# Split data into training and testing sets based on the year
train_filter = filtered_df[filtered_df['Year'] <= 2019]
test_filter = filtered_df[filtered_df['Year'] > 2019]

X_train_filter = train_filter.drop(['Production_t'], axis=1)
y_train_filter = train_filter['Production_t']

X_test_filter = test_filter.drop(['Production_t'], axis=1)
y_test_filter = test_filter['Production_t']

y_pred_train_filter = pipeline_rf.predict(X_train_filter)

rf_mae_train_filter, rf_mse_train_filter, rf_rmse_train_filter,
rf_r2_train_filter = calculate_metrics(y_train_filter,
y_pred_train_filter)

print("Mean Absolute Error (MAE): {:.2f}%".format(rf_mae_train_filter))
print("Mean Squared Error (MSE): {:.2f}%".format(rf_mse_train_filter))

print("Root Mean Squared Error (RMSE):
{:.2f}%".format(rf_rmse_train_filter))

print("R2 Score:", rf_r2_train_filter)
y_pred_test_filter = pipeline_rf.predict(X_test_filter)

rf_mae_test_filter, rf_mse_test_filter, rf_rmse_test_filter,
rf_r2_test_filter = calculate_metrics(y_test_filter, y_pred_test_filter)

print("Mean Absolute Error (MAE): {:.2f}%".format(rf_mae_test_filter))
print("Mean Squared Error (MSE): {:.2f}%".format(rf_mse_test_filter))
print("Root Mean Squared Error (RMSE):
{:.2f}%".format(rf_rmse_test_filter))
print("R2 Score:", rf_r2_test_filter)

import matplotlib.pyplot as plt
plt.hist(filtered_df['Production_t'], bins=30)
plt.title('Histogram of Target Variable')
plt.show()
```

```
# Define thresholds
threshold = 1.5e8

# Create subsets
def divide_segments(threshold, segment_no):
    segment1 = filtered_df[filtered_df['Production_t'] <= threshold]
    segment2 = filtered_df[filtered_df['Production_t'] > threshold] #&
(filtered_df['Production_t'] <= threshold2)]
# segment3 = filtered_df[(filtered_df['Production_t'] > threshold)
    if segment_no == 1:
        return segment1
    else:
        return segment2

X1 = divide_segments(threshold,1).drop('Production_t', axis=1)
y1 = divide_segments(threshold,1)['Production_t']
X2 = divide_segments(threshold,2).drop('Production_t', axis=1)
y2 = divide_segments(threshold,2)['Production_t']

pipeline_lr = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
])

pipeline_lasso = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', Lasso(alpha=1.0, random_state=42))
])

# pipeline for segment 2: Ridge Regressor
pipeline_ridge = Pipeline(steps = [
    ('preprocessor', preprocessor),
    ('regressor', Ridge(alpha=1.0, random_state=42))
])
# Pipeline for segment 2: Random Forest Regressor
pipeline_rf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(n_estimators=100, random_state=42,
max_depth= 6))
])

# Pipeline for segment 3: Gradient Boosting Regressor
pipeline_gb = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', GradientBoostingRegressor(n_estimators=100,
random_state=42))
])

# Function to split data and train pipeline
def train_test_pipeline(X, y, pipeline, mode):
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
    pipeline.fit(X_train, y_train)
    y_pred_tr = pipeline.predict(X_train)
    y_pred = pipeline.predict(X_test)
    mae_tr, mse_tr, rmse_tr, r2_tr = calculate_metrics(y_train, y_pred_tr)
    mae, mse, rmse, r2 = calculate_metrics(y_test, y_pred)
    if mode == 'test':
        return mae, mse, rmse, r2
    if mode == 'train':
        return mae_tr, mse_tr, rmse_tr, r2_tr

# Applying pipelines
mae1_tr, mse1_tr, rmse1_tr, r21_tr = train_test_pipeline(X1, y1,
pipeline_gb, 'train')
mae2_tr, mse2_tr, rmse2_tr, r22_tr = train_test_pipeline(X2, y2,
pipeline_ridge, 'train')
# mae3_tr, mse2_tr, rmse2_tr, r23_tr =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1_tr}, MSE: {mse1_tr}, RMSE: {rmse1_tr}, R2:
{r21_tr}")
print(f"Segment 2 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr}, R2:
{r22_tr}")
# print(f"Segment 3 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr},
R2: {r22_tr}")

# Applying pipelines
mae1, mse1, rmse1, r21 = train_test_pipeline(X1,y1, pipeline_gb, 'test')
mae2, mse2, rmse2, r22 = train_test_pipeline(X2, y2, pipeline_ridge,
'test')
# mae3, mse2, rmse2, r23 =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1}, MSE: {mse1}, RMSE: {rmse1}, R2: {r21}")
print(f"Segment 2 - MAE: {mae2}, MSE: {mse2}, RMSE: {rmse2}, R2: {r22}")
# print(f"Segment 3 - MAE: {mae2}, MSE: {mse2}, RMSE: {rmse2}, R2: {r22}")

"""## Finding Threshold based on Clustering"""

from sklearn.cluster import DBSCAN

def find_density_based_threshold(data):
    values = data['Production_t'].values.reshape(-1, 1)
    clustering = DBSCAN(eps=0.5, min_samples=10).fit(values)
    labels = clustering.labels_
    # Find transition points between clusters
    unique_labels = np.unique(labels)
    core_samples = np.array([values[labels == label].mean() for label in
unique_labels if label != -1])
    if len(core_samples) > 1:
        sorted_samples = np.sort(core_samples)
        # Threshold between the first two clusters
        threshold = np.mean(sorted_samples[:2])
        return threshold
    return None

thrshold_db = find_density_based_threshold(filtered_df)
thrshold_db
X_db1 = divide_segments(thrshold_db,1).drop('Production_t', axis=1)
y_db1 = divide_segments(thrshold_db,1)['Production_t']
X_db2 = divide_segments(thrshold_db,2).drop('Production_t', axis=1)
y_db2 = divide_segments(thrshold_db,2)['Production_t']

# Applying pipelines
mae1_db_tr, mse1_db_tr, rmse1_db_tr, r21_db_tr =
train_test_pipeline(X_db1, y_db1,pipeline_rf, 'train')
mae2_db_tr, mse2_db_tr, rmse2_db_tr, r22_db_tr =
train_test_pipeline(X_db2, y_db2, pipeline_ridge, 'train')
# mae3_tr, mse2_tr, rmse2_tr, r23_tr =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1_db_tr}, MSE: {mse1_db_tr}, RMSE:
{rmse1_db_tr}, R2: {r21_db_tr}")
print(f"Segment 2 - MAE: {mae2_db_tr}, MSE: {mse2_db_tr}, RMSE:
{rmse2_db_tr}, R2: {r22_db_tr}")
# print(f"Segment 3 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr},
R2: {r22_tr}")

# Applying pipelines
mae1_db, mse1_db, rmse1_db, r21_db = train_test_pipeline(X_db1,
y_db1,pipeline_gb, 'test')
mae2_db, mse2_db, rmse2_db, r22_db = train_test_pipeline(X_db2, y_db2,
pipeline_ridge, 'test')
# mae3_tr, mse2_tr, rmse2_tr, r23_tr =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1_db}, MSE: {mse1_db}, RMSE: {rmse1_db}, R2:
{r21_db}")
print(f"Segment 2 - MAE: {mae2_db}, MSE: {mse2_db}, RMSE: {rmse2_db}, R2:
{r22_db}")
# print(f"Segment 3 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr},
R2: {r22_tr}")

"""## Change Point Detection"""
pip install ruptures
from ruptures import Pelt
import ruptures as rpt

def find_change_point_threshold(data):
    values = data['Production_t'].values
    algo = rpt.Pelt(model="l2").fit(values)
    result = algo.predict(pen=10)
    # Find the first significant change point as the threshold
    if result:
        threshold = values[result[0]]
        return threshold
    return None

thr_cpd = find_change_point_threshold(filtered_df)
thr_cpd

X_cpd1 = divide_segments(thr_cpd,1).drop('Production_t', axis=1)
y_cpd1 = divide_segments(thr_cpd,1)['Production_t']
X_cpd2 = divide_segments(thr_cpd,2).drop('Production_t', axis=1)
y_cpd2 = divide_segments(thr_cpd,2)['Production_t']

# Applying pipelines
mae1_cpd_tr, mse1_cpd_tr, rmse1_cpd_tr, r21_cpd_tr =
train_test_pipeline(X_cpd1, y_cpd1,pipeline_rf, 'train')
mae2_cpd_tr, mse2_cpd_tr, rmse2_cpd_tr, r22_cpd_tr =
train_test_pipeline(X_cpd2, y_cpd2, pipeline_rf, 'train')
# mae3_tr, mse2_tr, rmse2_tr, r23_tr =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1_cpd_tr}, MSE: {mse1_cpd_tr}, RMSE:
{rmse1_cpd_tr}, R2: {r21_cpd_tr}")
print(f"Segment 2 - MAE: {mae2_cpd_tr}, MSE: {mse2_cpd_tr}, RMSE:
{rmse2_cpd_tr}, R2: {r22_cpd_tr}")
# print(f"Segment 3 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr},
R2: {r22_tr}")

# Applying pipelines
mae1_cpd, mse1_cpd, rmse1_cpd, r21_cpd = train_test_pipeline(X_cpd1,
y_cpd1,pipeline_rf, 'train')
mae2_cpd, mse2_cpd, rmse2_cpd, r22_cpd = train_test_pipeline(X_cpd2,
y_cpd2, pipeline_ridge, 'train')
# mae3_tr, mse2_tr, rmse2_tr, r23_tr =
train_test_pipeline(segment3.drop('Production_t', axis=1),
segment3['Production_t'], pipeline_gb)

print(f"Segment 1 - MAE: {mae1_cpd}, MSE: {mse1_cpd}, RMSE: {rmse1_cpd},
R2: {r21_cpd}")
print(f"Segment 2 - MAE: {mae2_cpd}, MSE: {mse2_cpd}, RMSE: {rmse2_cpd},
R2: {r22_cpd}")
# print(f"Segment 3 - MAE: {mae2_tr}, MSE: {mse2_tr}, RMSE: {rmse2_tr},
R2: {r22_tr}")

"""### Segregating Train and Test based on Years"""
data_b_2019 = filtered_df[filtered_df["Year"] < 2019]

data_a_2019 = filtered_df[filtered_df["Year"] >= 2019]

import matplotlib.pyplot as plt
plt.hist(data_b_2019['Production_t'], bins=30)
plt.title('Histogram of Target Variable')
plt.show()

import matplotlib.pyplot as plt
plt.hist(data_a_2019['Production_t'], bins=30)
plt.title('Histogram of Target Variable')
plt.show()
thr_b_2019 = find_change_point_threshold(data_b_2019)
thr_b_2019
thr_a_2019 = find_change_point_threshold(data_a_2019)
thr_a_2019

def train_and_evaluate(X, y, pipeline):
    pipeline.fit(X, y)
    y_pred = pipeline.predict(X)
    return calculate_metrics(y, y_pred)

def evaluate_models(data, pipelines):
    X = data.drop(columns=['Production_t'])
    y = data['Production_t']

    results = {}
    for name, pipeline in pipelines.items():
        metrics = train_and_evaluate(X, y, pipeline)
        results[name] = metrics

    return results

# Segmenting data based on thresholds
data_b_2019_segment1 = data_b_2019[data_b_2019["Production_t"] <=
thr_b_2019]
data_b_2019_segment2 = data_b_2019[data_b_2019["Production_t"] >
thr_b_2019]

data_a_2019_segment1 = data_a_2019[data_a_2019["Production_t"] <=
thr_a_2019]
data_a_2019_segment2 = data_a_2019[data_a_2019["Production_t"] >
thr_a_2019]

pipelines = {
    'Linear Regression': Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', LinearRegression())
]),

    'Lasso': Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', Lasso(alpha=1.0, random_state=42))
]),

    'Ridge': Pipeline(steps = [
    ('preprocessor', preprocessor),
    ('regressor', Ridge(alpha=1.0, random_state=42))
]),

    'Random Forest': Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', RandomForestRegressor(n_estimators=100, random_state=42,
max_depth= 4))
]),

    'Gradient Boosting': Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', GradientBoostingRegressor(n_estimators=100,
random_state=42))
])
}

# Evaluate models for each segment
results_b_segment1 = evaluate_models(data_b_2019_segment1, pipelines)
results_b_segment2 = evaluate_models(data_b_2019_segment2, pipelines)
results_a_segment1 = evaluate_models(data_a_2019_segment1, pipelines)
results_a_segment2 = evaluate_models(data_a_2019_segment2, pipelines)

# Print results
print("Results for data before 2019, Segment 1:", results_b_segment1)
print("Results for data before 2019, Segment 2:", results_b_segment2)
print("Results for data after 2019, Segment 1:", results_a_segment1)
print("Results for data after 2019, Segment 2:", results_a_segment2)

def find_best_model(results):
    best_model_name = None
    best_model_score = -np.inf  # Initialize with negative infinity for
comparison
    best_model_metrics = None

    for model_name, metrics in results.items():
        r2_score = metrics[-1]  # The last metric in the tuple is R²
        if r2_score > best_model_score:
            best_model_score = r2_score
            best_model_name = model_name
            best_model_metrics = metrics

    return best_model_name, best_model_metrics

# Print the best model for each segment
def print_best_model_results(segment_name, results):
    best_model_name, best_model_metrics = find_best_model(results)
    print(f"Best model for {segment_name}:")
    print(f"Model: {best_model_name}")
    print(f"Metrics (MAE%, MSE%, RMSE%, R²): {best_model_metrics}\n")

print_best_model_results("data before 2019, Segment 1",
results_b_segment1)
print_best_model_results("data before 2019, Segment 2",
results_b_segment2)
print_best_model_results("data after 2019, Segment 1", results_a_segment1)
print_best_model_results("data after 2019, Segment 2", results_a_segment2)
```