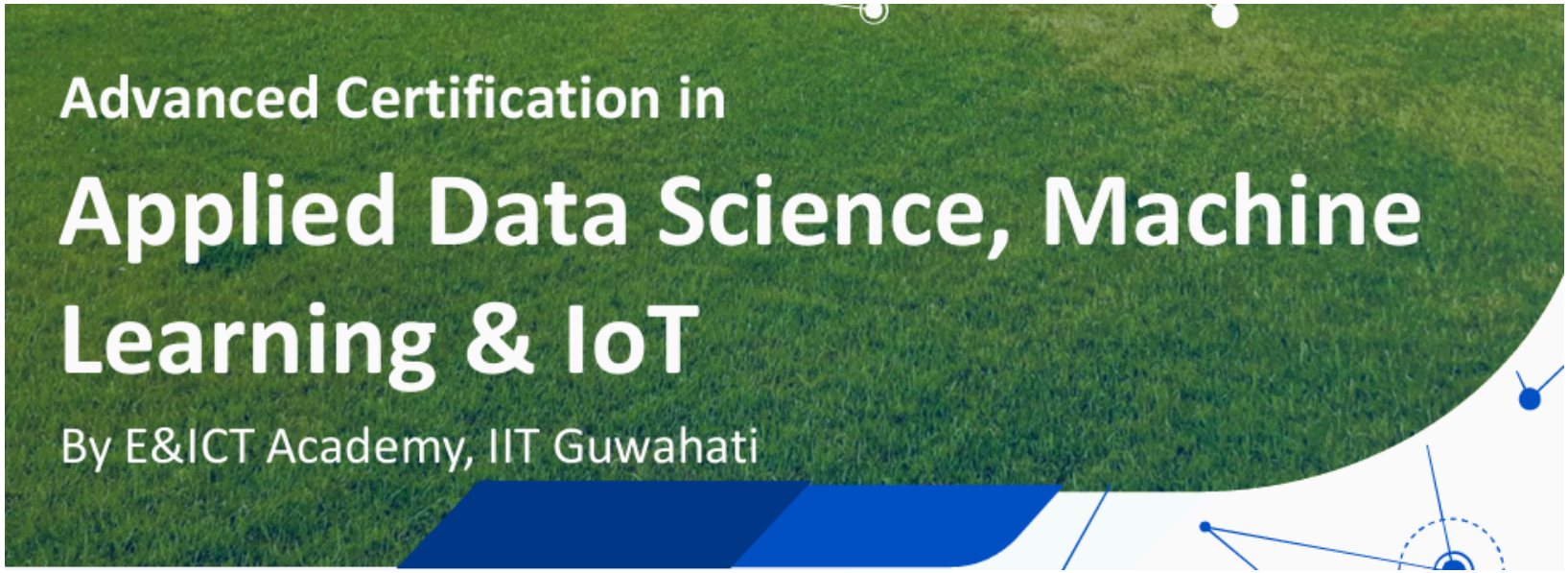


Capstone Project - 1

Name: Rajesh Bisht

E-mail Id: rbisht.india@gmail.com

Git Link source code: https://github.com/RajeshBisht28/CapstonjeProectOne

In [ ]:

Project (Data Set) Overview:

This project aims to predict the yield of different crops by analyzing historical (from 1997 to 2020) 24-years agricultural data. By using Machine earning and statistical method that models the relationship between a dependent variable and one or more independent variables, we can forecast future crop production. This is particularly valuable for farmers, policymakers, and

agricultural businesses who need reliable predictions to make informed decisions about crop planning, resource allocation, and market strategies.

Crop Prediction Dataset

Its about the Indian based Crops, the dataset is designed to analyze various factors influencing crop production and yield. Below is a detailed description of each feature in the dataset. Target feature is Yield.

Feature Name	Description
Crop	The type of crop being analyzed (e.g., Arecanut, Cotton).
Crop_Year	The year of crop production (1997 to 2020)
Season	The season in which the crop is grown (e.g., Kharif, Whole Year).
State	The region or state where the crop is cultivated (e.g, Assam, Bihar, Delhi).
Area	The area of cultivation, measured in hectares.
Production	The total production output, measured in tons.
Annual_Rainfall	The average annual rainfall in the region, measured in millimeters.
Fertilizer	The total amount of fertilizer used, measured in kilograms.
Pesticide	The total amount of pesticide applied, measured in kilograms.
Yield	The calculated yield of the crop (production per unit area).

Data Characteristics:

- Temporal data (years from 1997 to 2020).
- Geographical data (states in India).
- Categorical data (Crop as name / type of crop).
- seasonal data can be categorized as temporal data.
- Agricultural factors affecting yield.

Import the Data set

In [2]:
```
import pandas as pd
import numpy as np
# Load dataset
df_original = pd.read_csv(r'crop_yield.csv')
```

In [3]:
```
df_original.head()
```

Out[3]:

	Crop	Crop_Year	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
0	Arecanut	1997	Whole Year	Assam	73814.0	56708	2051.4	7024878.38	22882.34	0.796087
1	Arhar/Tur	1997	Kharif	Assam	6637.0	4685	2051.4	631643.29	2057.47	0.710435
2	Castor seed	1997	Kharif	Assam	796.0	22	2051.4	75755.32	246.76	0.238333
3	Coconut	1997	Whole Year	Assam	19656.0	126905000	2051.4	1870661.52	6093.36	5238.051739
4	Cotton(lint)	1997	Kharif	Assam	1739.0	794	2051.4	165500.63	539.09	0.420909

Find Correlation b/w Yield vs Crop Year

In [4]:
```
correlation = df_original['Yield'].corr(df_original['Crop_Year'])
print(f"Correlation b/w Yield vs Crop Year : {correlation}")
```

```
Correlation b/w Yield vs Crop Year : 0.0025391242148657452
```

A correlation coefficient of Crop_Year is 0.0025 indicates a very weak positive correlation. So we can eliminate Crop_Year feature.

In [5]:
```
### Remove Crop_Year column
df_rc = df_original.copy()
df_rc = df_original.drop(columns=['Crop_Year'])
df_rc.head()
```

Out[5]:

	Crop	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
0	Arecanut	Whole Year	Assam	73814.0	56708	2051.4	7024878.38	22882.34	0.796087
1	Arhar/Tur	Kharif	Assam	6637.0	4685	2051.4	631643.29	2057.47	0.710435
2	Castor seed	Kharif	Assam	796.0	22	2051.4	75755.32	246.76	0.238333
3	Coconut	Whole Year	Assam	19656.0	126905000	2051.4	1870661.52	6093.36	5238.051739
4	Cotton(lint)	Kharif	Assam	1739.0	794	2051.4	165500.63	539.09	0.420909

Check Total NULL values

In [6]:
```
total_nulls = df_rc.isnull().sum().sum()
print(f"Total null values are: {total_nulls}")
```

```
Total null values are: 0
```

Get Information of Dataset, As below Table, Rows count 19689 and 9 columns. (After eliminate Crop_Year column)

In [7]:
```
import pandas as pd
from io import StringIO

buffer = StringIO()
df_rc.info(buf=buffer)
info_str = buffer.getvalue()

# Parse the info string to create a DataFrame
lines = info_str.strip().split('\n')
```

```python
# Adjust the headers based on the actual output
headers = ["#", "Column", "Non-Null Count", "Dtype"]
data = []

# Extract the relevant lines and split appropriately
for line in lines[3:]:
    parts = line.split()
    if len(parts) > 1:
        # Handle the format of the lines correctly
        column_name = parts[1]
        non_null_count = parts[2]
        dtype = parts[-1]
        row = [parts[0], column_name, non_null_count, dtype]
        data.append(row)

# Create DataFrame from parsed data
info_df = pd.DataFrame(data, columns=headers)

# Style the DataFrame
styled_info_df = info_df.style.set_table_styles(
    [
        {'selector': 'thead th', 'props': [('background-color', '#3E4149'), ('color', 'white'), ('text-align', 'cente
        {'selector': 'tbody td', 'props': [('border', '1px solid black'), ('text-align', 'center')]}
    ]
).set_properties(**{'text-align': 'center'})

# Display the styled DataFrame
styled_info_df
```

Out[7]:

	#	Column	Non-Null Count	Dtype
0	#	Column	Non-Null	Dtype
1	---	------	--------------	-----
2	0	Crop	19689	object
3	1	Season	19689	object
4	2	State	19689	object
5	3	Area	19689	float64
6	4	Production	19689	int64
7	5	Annual_Rainfall	19689	float64
8	6	Fertilizer	19689	float64
9	7	Pesticide	19689	float64
10	8	Yield	19689	float64
11	dtypes:	float64(5),	int64(1),	object(3)
12	memory	usage:	1.4+	MB

Removing non-alphanumeric characters from "Crop", "State", "Season" values

In [8]:
```
df_rc_filter = df_rc.copy()
df_rc_filter['Crop'] = df_rc['Crop'].str.replace(r'\W+', ' ', regex=True)
df_rc_filter['State'] = df_rc['State'].str.replace(r'\W+', ' ', regex=True)
df_rc_filter['Season'] = df_rc['Season'].str.replace(r'\W+', ' ', regex=True)
```

In [21]:
```
df_rc_filter.head(10)
```

Out[21]:

	Crop	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
0	Arecanut	Whole Year	Assam	73814.0	56708	2051.4	7024878.38	22882.34	0.796087
1	Arhar Tur	Kharif	Assam	6637.0	4685	2051.4	631643.29	2057.47	0.710435
2	Castor seed	Kharif	Assam	796.0	22	2051.4	75755.32	246.76	0.238333
3	Coconut	Whole Year	Assam	19656.0	126905000	2051.4	1870661.52	6093.36	5238.051739
4	Cotton lint	Kharif	Assam	1739.0	794	2051.4	165500.63	539.09	0.420909
5	Dry chillies	Whole Year	Assam	13587.0	9073	2051.4	1293074.79	4211.97	0.643636
6	Gram	Rabi	Assam	2979.0	1507	2051.4	283511.43	923.49	0.465455
7	Jute	Kharif	Assam	94520.0	904095	2051.4	8995468.40	29301.20	9.919565
8	Linseed	Rabi	Assam	10098.0	5158	2051.4	961026.66	3130.38	0.461364
9	Maize	Kharif	Assam	19216.0	14721	2051.4	1828786.72	5956.96	0.615652

In [ ]:

Label encoding: There are some feature / columns are object :

Label encode for Crop, Season, State.

In [22]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
### Copy of original dataset with new data set : df_encode
df_encode = df_rc_filter.copy()
# Label encoding
label_encoder = LabelEncoder()

# Custom mapping to avoid zeros
df_encode['Crop'] = label_encoder.fit_transform(df_original['Crop'])+1
df_encode['Season'] = label_encoder.fit_transform(df_original['Season'])+1
df_encode['State'] = label_encoder.fit_transform(df_original['State'])+1
```

In [23]:
```
df_rc_filter.head(3)
```

Out[23]:

	Crop	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
0	Arecanut	Whole Year	Assam	73814.0	56708	2051.4	7024878.38	22882.34	0.796087
1	Arhar Tur	Kharif	Assam	6637.0	4685	2051.4	631643.29	2057.47	0.710435
2	Castor seed	Kharif	Assam	796.0	22	2051.4	75755.32	246.76	0.238333

In [ ]:
```

```

Avoid E-notation / Scientific notation, Round up values.

In [24]:
```
df_encode = df_encode.round(2)
df_encode.head(5)
```

Out[24]:

	Crop	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
0	1	5	3	73814.0	56708	2051.4	7024878.38	22882.34	0.80
1	2	2	3	6637.0	4685	2051.4	631643.29	2057.47	0.71
2	9	2	3	796.0	22	2051.4	75755.32	246.76	0.24
3	10	5	3	19656.0	126905000	2051.4	1870661.52	6093.36	5238.05
4	12	2	3	1739.0	794	2051.4	165500.63	539.09	0.42

Finding correlation matrix for 'Yield'

In [25]:
```
correlation_matrix = df_encode.corr()
print(correlation_matrix['Yield'])
```

```
Crop              -0.110894
Season             0.141791
State              0.009668
Area               0.001859
Production         0.570809
Annual_Rainfall    0.020761
Fertilizer         0.002862
Pesticide          0.001782
Yield              1.000000
Name: Yield, dtype: float64
```

Crop : is simply name an identifier, and the negative correlation doesn't have any meaningful interpretation.

Feature	Comment
Crop	This indicates a weak, negative correlation between crop type and yield. As the crop type changes, the yield slightly decreases.
Season	There is a weak, positive correlation between season and yield, suggesting that different seasons have a minor impact on yield.
State	A very weak, positive correlation between state and yield. The state in which the crop is grown has an almost negligible effect on yield.
Area	There's an extremely weak, positive correlation between the area of cultivation and yield, indicating that area has virtually no impact on yield.
Production	This indicates a moderate, positive correlation between production and yield. Higher production is moderately associated with higher yield.
Annual_Rainfall	A very weak, positive correlation between annual rainfall and yield, suggesting that rainfall has little to no effect on yield.

Feature	Comment
Fertilizer	There's an extremely weak, positive correlation between fertilizer use and yield, implying fertilizer use does not significantly affect yield.
Pesticide	A very weak, positive correlation between pesticide use and yield, indicating minimal impact on yield.

In [26]:
```
print(df_encode.min())
```

```
Crop                 1.00
Season               1.00
State                1.00
Area                 0.50
Production           0.00
Annual_Rainfall    301.30
Fertilizer          54.17
Pesticide            0.09
Yield                0.00
dtype: float64
```

There is no any -ve values , Apply Box-Cox Transformation to relevant columns (excluding categorical ones)

In [27]:
```
import pandas as pd
import numpy as np
from scipy.stats import boxcox
# Apply Box-Cox Transformation to relevant columns (excluding categorical ones)
df_boxcox = df_encode.copy()

for column in ['Area', 'Production', 'Annual_Rainfall', 'Fertilizer', 'Pesticide', 'Yield']:
    df_boxcox[column] = np.where(df_boxcox[column] < 0, 1e-7, df_boxcox[column])  # Handle negatives
    df_boxcox[column], _ = boxcox(df_boxcox[column] + 1)
```

See below: After apply Transformation Minimum values not in negative.

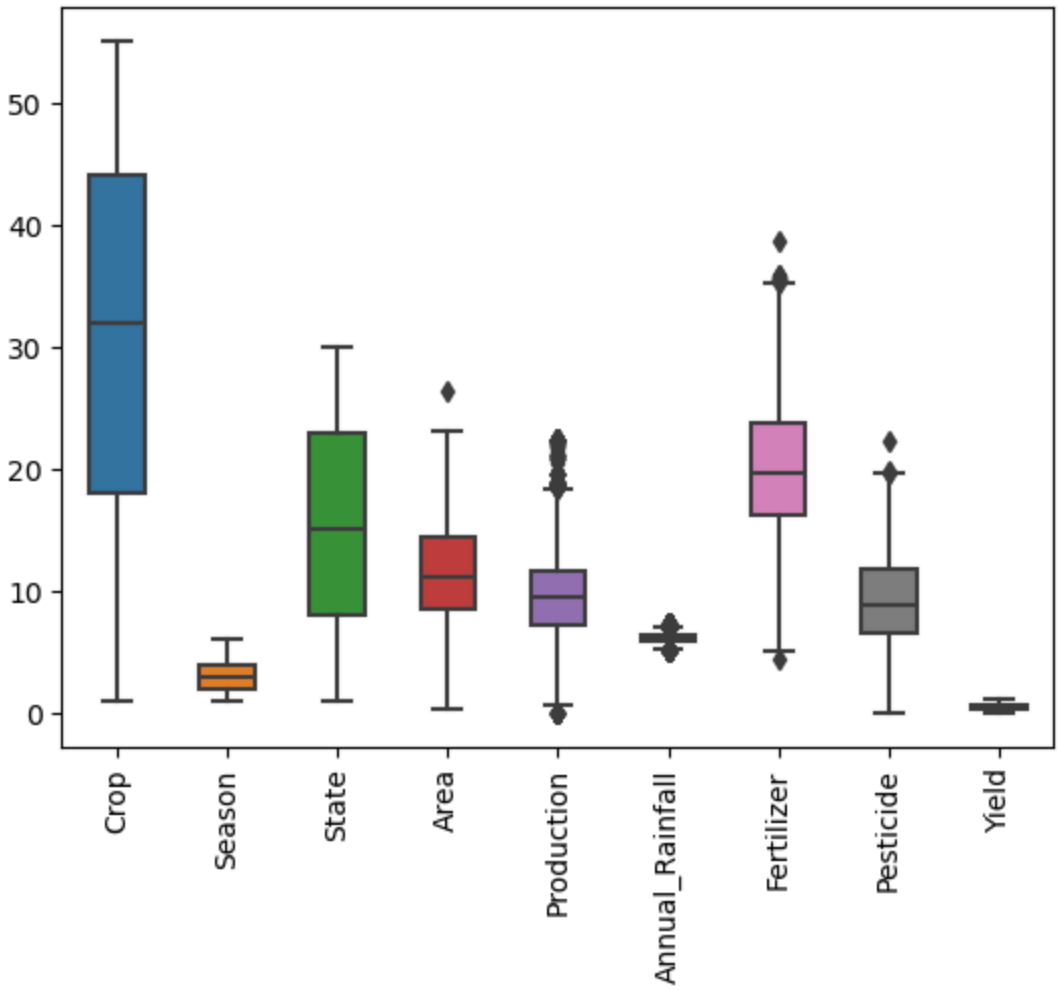
In [61]:
```
print(df_boxcox.min())
```

```
Crop               1.000000
Season             1.000000
State              1.000000
Area               0.408941
Production         0.000000
Annual_Rainfall    5.078614
Fertilizer         4.399786
Pesticide          0.086303
Yield              0.000000
dtype: float64
```

In [ ]:
```

```

Check for outliers using boxplot

In [28]:
```
import seaborn as sns
import matplotlib.pyplot as plt
# Check for outliers using boxplot
sns.boxplot(data=df_boxcox, width=.5)
plt.xticks(rotation=90)
plt.show()
```



In [ ]:

Caping outliers using percentile method.

In [29]:
```
import pandas as pd
import numpy as np

columns = ['Area','Production', 'Annual_Rainfall', 'Fertilizer', 'Pesticide', 'Yield']
df_remove_outliers = df_boxcox.copy()
# Cap the outliers at 1st and 99th percentile
```

```
for column in columns:
    if df_boxcox[column].dtype != 'object':  # Ignore categorical columns
        lower_percentile = df_boxcox[column].quantile(0.01)
        upper_percentile = df_boxcox[column].quantile(0.99)
        df_remove_outliers[column] = np.where(df_boxcox[column] < lower_percentile, lower_percentile, df_boxcox[colur
        df_remove_outliers[column] = np.where(df_boxcox[column] > upper_percentile, upper_percentile, df_boxcox[colur
```

In [30]:
```
#print(df_remove_outliers)
row_count = len(df_remove_outliers)
print(f"Number of rows: {row_count}")
```

```
Number of rows: 19689
```

Calculate the absolute differences after caping outliers, Checking how the outlier treatment progressed.

In [31]:
```
# Calculate the absolute differences
differences = df_remove_outliers[columns] - df_boxcox[columns]

# Check where the values were capped (non-zero difference indicates a change)
modified_data = differences != 0

# Summary of how many rows were affected for each column
modification_count = modified_data.sum()
print("Number of rows modified in each column:")
print(modification_count)
```

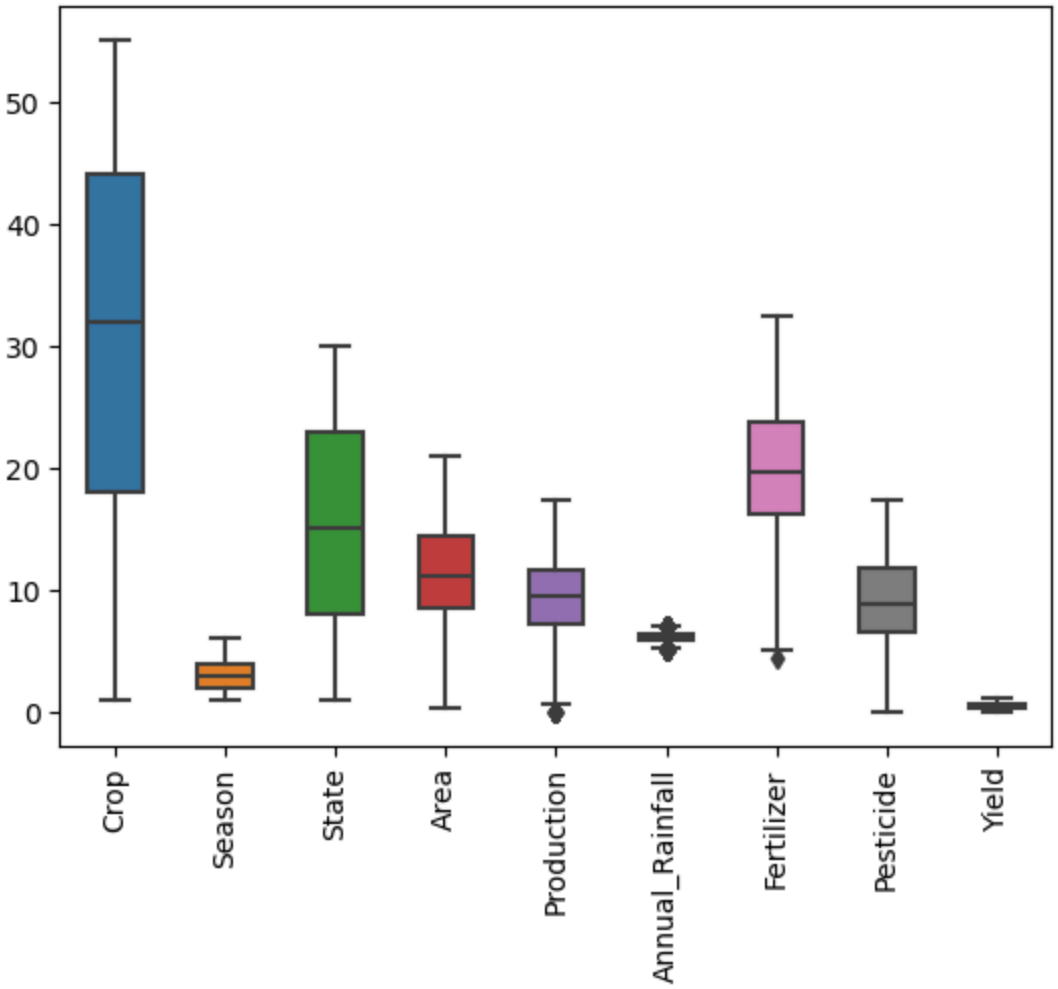
```
Number of rows modified in each column:
Area               197
Production         197
Annual_Rainfall    170
Fertilizer         197
Pesticide          197
Yield              197
dtype: int64
```

In [ ]:
```

```

After Capping Outlier: Check for outliers using boxplot.

In [32]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# Check for outliers using boxplot
sns.boxplot(data=df_remove_outliers, width=.5)
plt.xticks(rotation=90)
plt.show()
```



Correlation Table among all features. (After removes outliers)

In [33]:
```
#  Heatmap to show correlations
import pandas as pd
correlation_matrix = df_remove_outliers.corr()
```

```
correlation_matrix_styled = correlation_matrix.style.background_gradient(cmap='coolwarm').format(precision=2)
correlation_matrix_styled
```

Out[33]:

	Crop	Season	State	Area	Production	Annual_Rainfall	Fertilizer	Pesticide	Yield
Crop	1.00	0.04	0.04	0.02	0.03	0.05	0.02	0.02	0.06
Season	0.04	1.00	-0.04	-0.03	0.11	0.10	-0.03	-0.04	0.30
State	0.04	-0.04	1.00	-0.11	-0.08	0.11	-0.10	-0.10	0.02
Area	0.02	-0.03	-0.11	1.00	0.90	-0.23	1.00	0.99	0.09
Production	0.03	0.11	-0.08	0.90	1.00	-0.18	0.90	0.90	0.47
Annual_Rainfall	0.05	0.10	0.11	-0.23	-0.18	1.00	-0.23	-0.23	0.05
Fertilizer	0.02	-0.03	-0.10	1.00	0.90	-0.23	1.00	0.99	0.10
Pesticide	0.02	-0.04	-0.10	0.99	0.90	-0.23	0.99	1.00	0.10
Yield	0.06	0.30	0.02	0.09	0.47	0.05	0.10	0.10	1.00

Check Multicollinearity, using Variance Inflation Factor (VIF) table.

In [34]:

```
import pandas as pd
from IPython.display import HTML
from statsmodels.stats.outliers_influence import variance_inflation_factor
# VIF Calculation
def calculate_vif(df):
    vif = pd.DataFrame()
    vif["variables"] = df.columns
    vif["VIF"] = [variance_inflation_factor(df.values, i) for i in range(df.shape[1])]
    return vif

vif_data = calculate_vif(df_remove_outliers)
html_content = vif_data.to_html(index=False)
HTML(html_content)
```

Out[34]:

variables	VIF
Crop	4.847405
Season	8.146031
State	3.997719
Area	1844.399900
Production	286.787289
Annual_Rainfall	132.026735
Fertilizer	1833.027128
Pesticide	541.583695
Yield	35.549511

Skewness Measure for Features.

In [ ]:

In [35]:
```
import pandas as pd

# Assuming df is your DataFrame
# Calculate skewness for all columns
skewness = df_remove_outliers.skew().reset_index()
skewness.columns = ['Column', 'Skewness']

# Create a styled DataFrame
styled_skewness = skewness.style.set_table_styles(
    [
        {'selector': 'thead th', 'props': [('background-color', '#3E4149'), ('color', 'white'), ('text-align', 'cente
        {'selector': 'tbody td', 'props': [('border', '1px solid black'), ('text-align', 'center')]}
    ]
).set_properties(**{'text-align': 'center'})

# Display the styled DataFrame
styled_skewness
```

Out[35]:

	Column	Skewness
0	Crop	-0.221717
1	Season	0.739242
2	State	0.050369
3	Area	-0.036974
4	Production	-0.166441
5	Annual_Rainfall	-0.073746
6	Fertilizer	-0.038491
7	Pesticide	-0.039400
8	Yield	0.403356

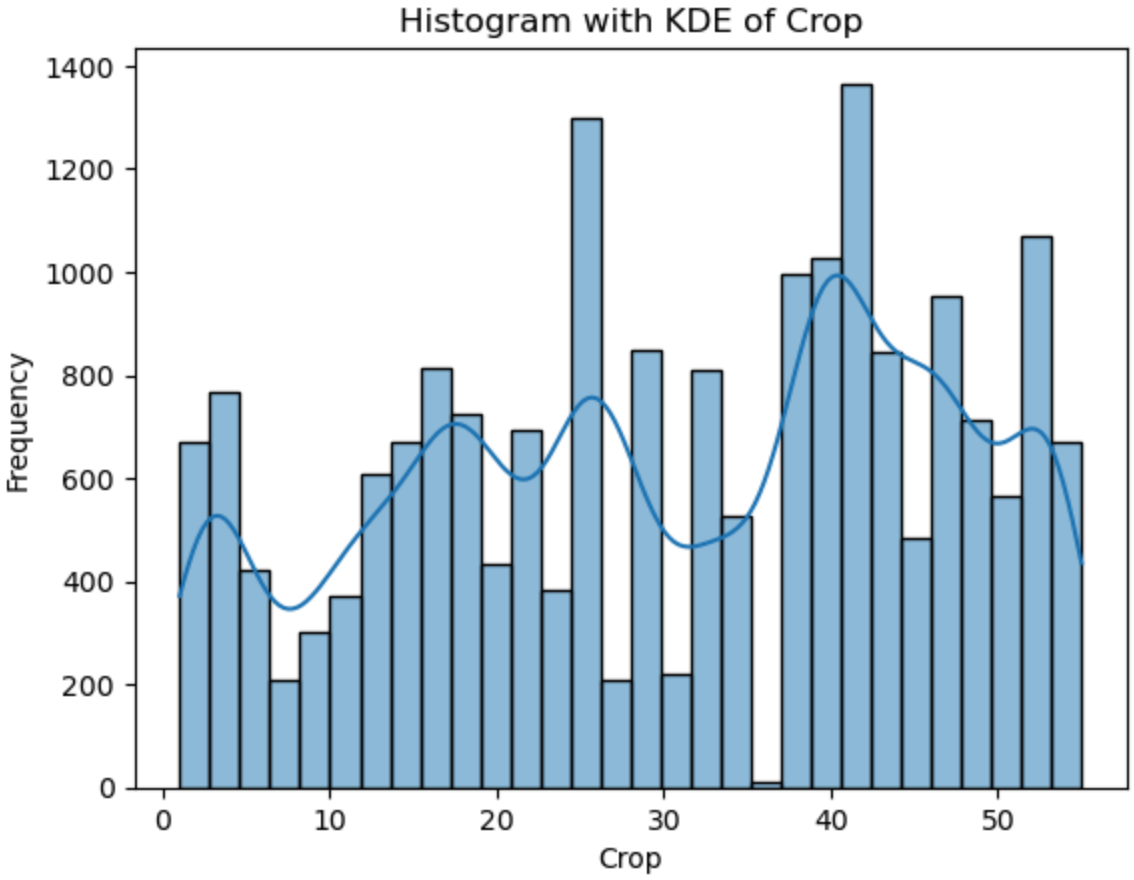
State, Area, Annual_Rainfall, Fertilizer, Pesticide all have skewness values close to 0, indicating a nearly symmetrical distribution. Production have slightly negatively skewed. Seems major effective feature are symmetrical distribution, So Model might perform better and the results will be more reliable.

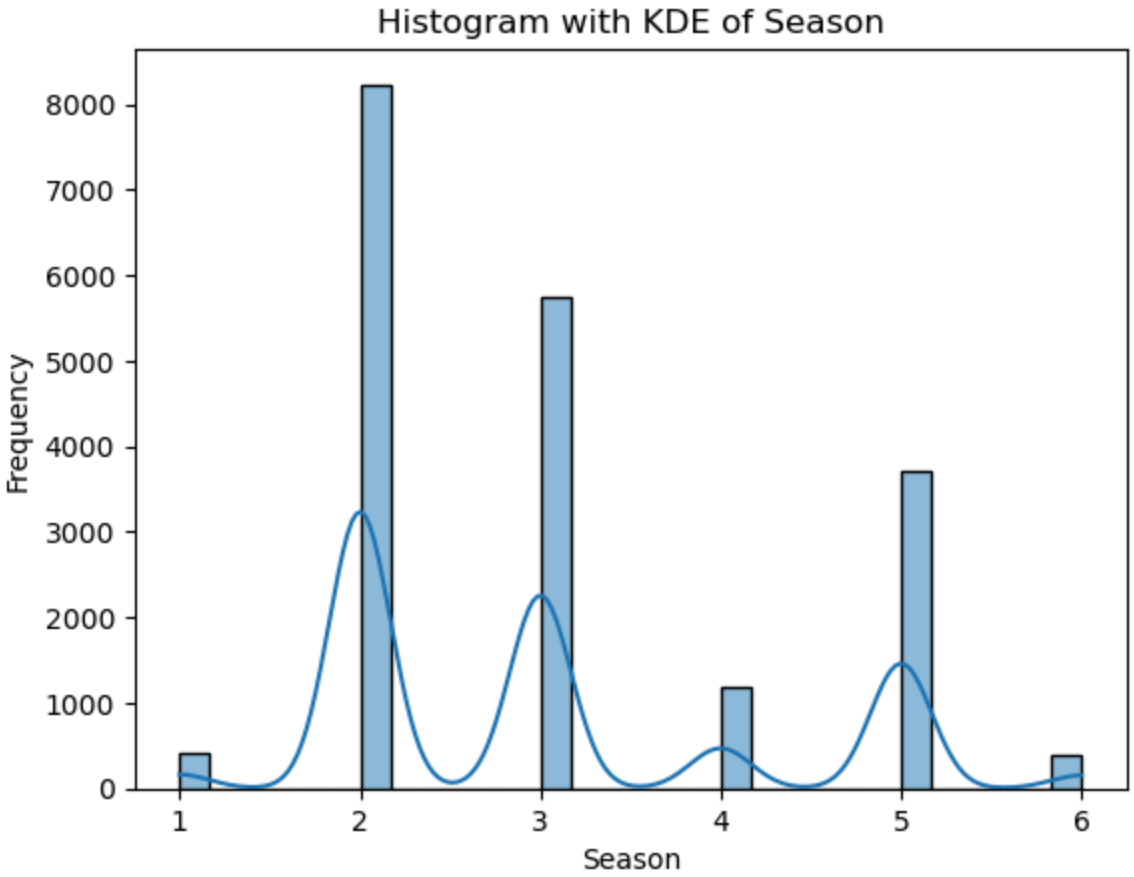
Visualize : distribution, Skewness, peaks : via : Histogram KDE.

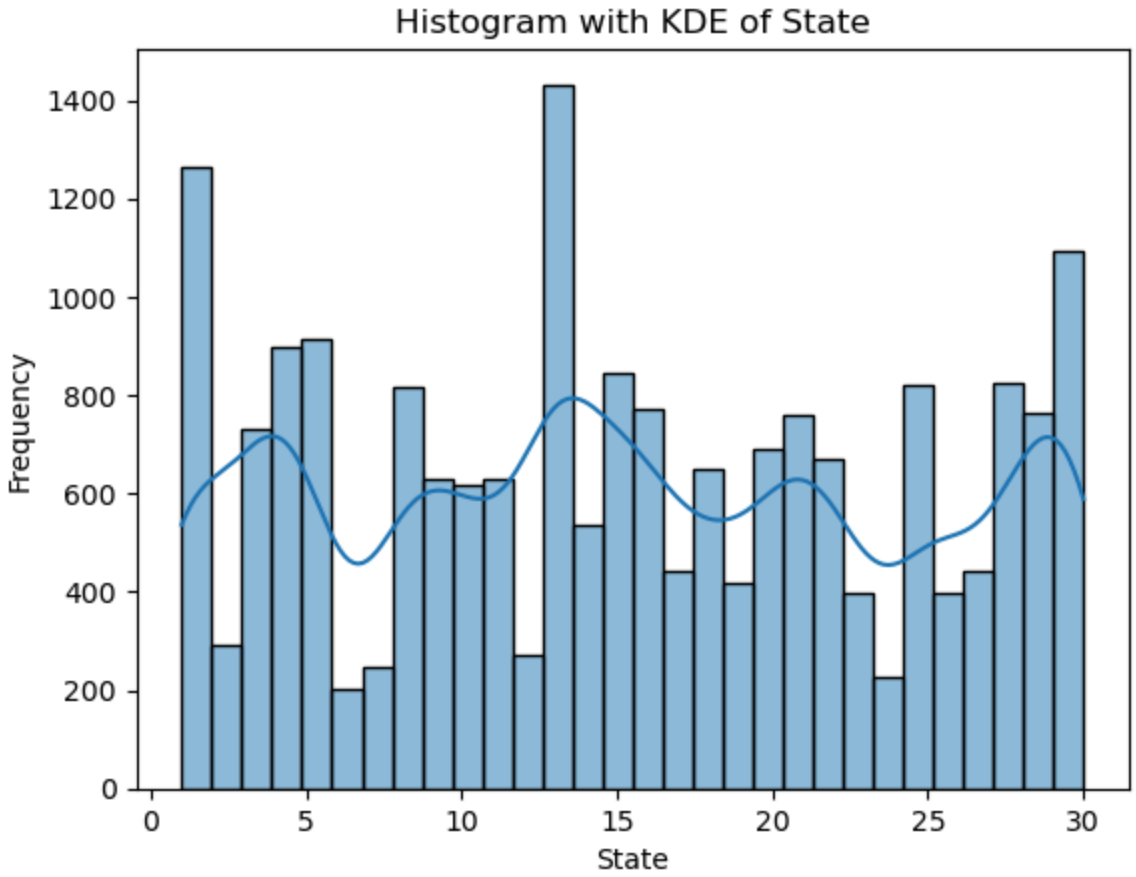
In [36]:

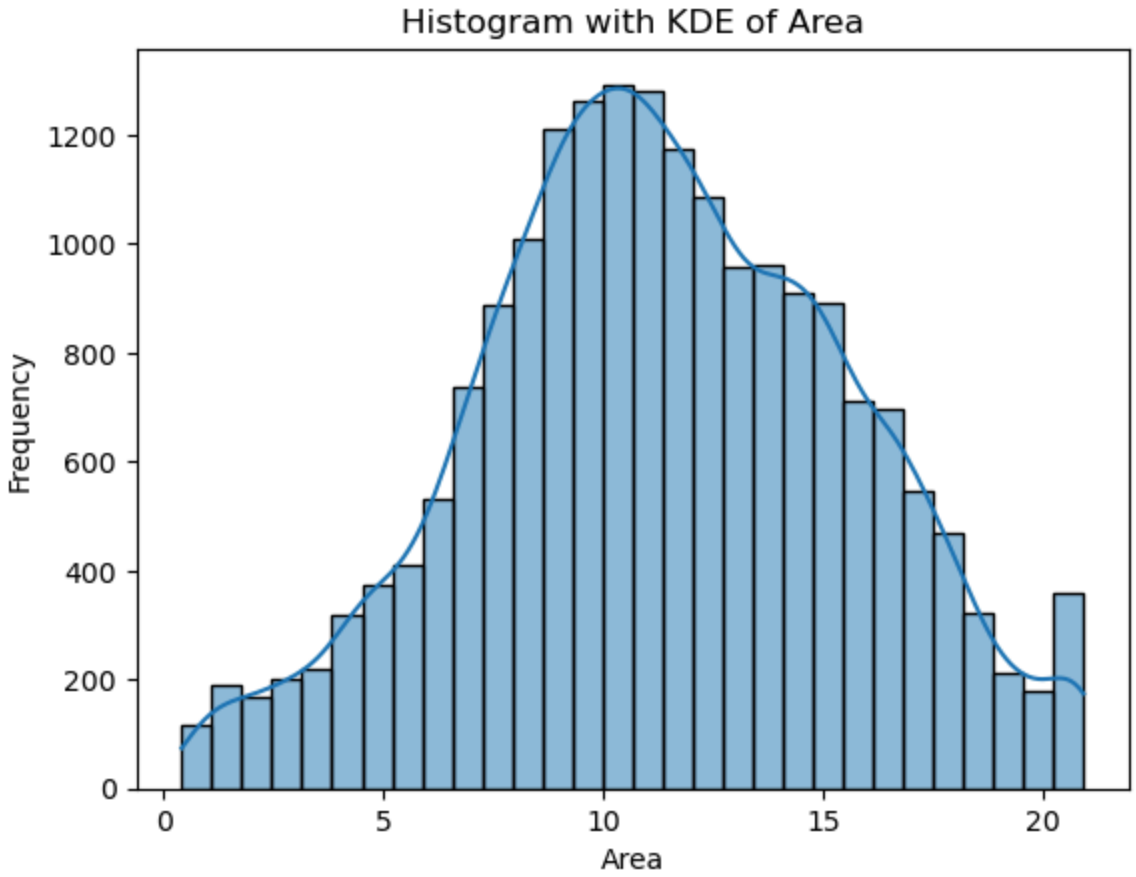
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import warnings

# Replace inf values with NaN
df_remove_outliers.replace([np.inf, -np.inf], np.nan, inplace=True)
warnings.simplefilter(action='ignore', category=FutureWarning)
# List of columns
columns = df_boxcox.columns

# Plot histogram with KDE for each column
for column in columns:
```

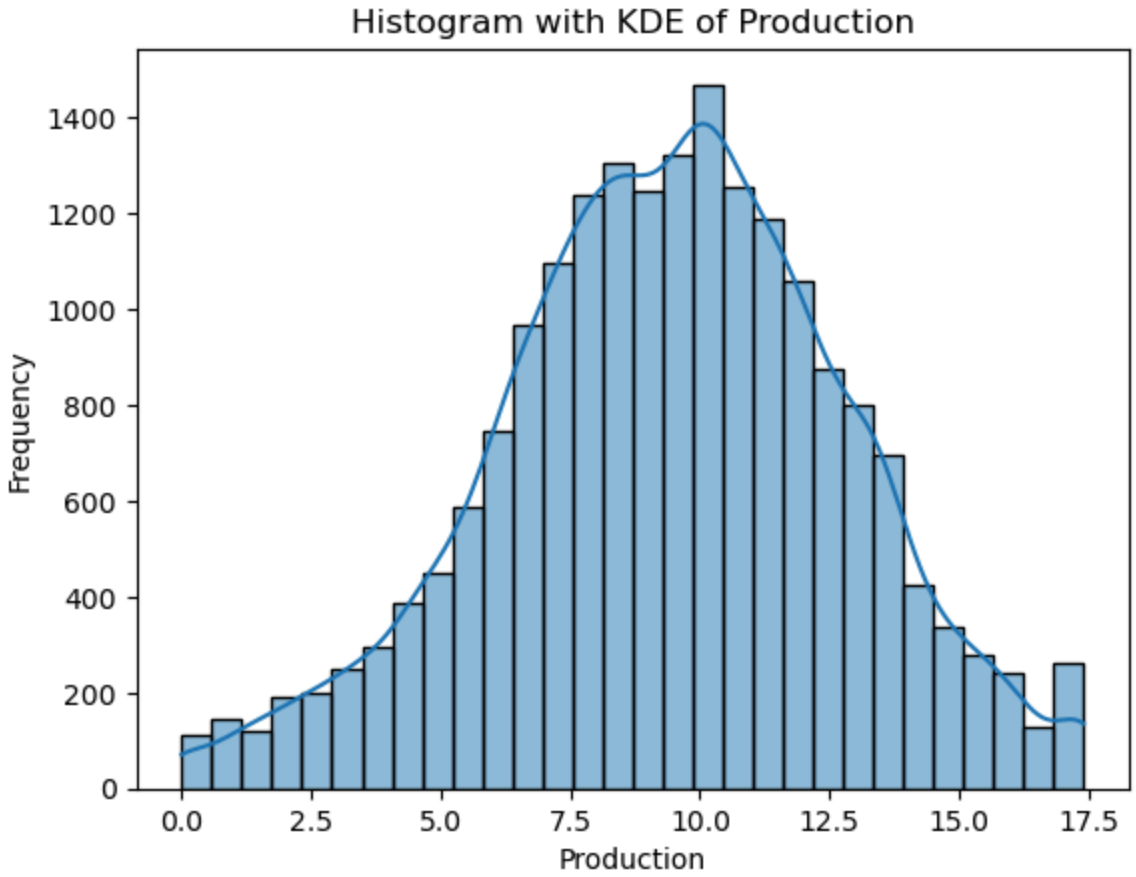
```
    plt.figure()
    sns.histplot(df_remove_outliers[column].dropna(), kde=True, bins=30)
    plt.title(f'Histogram with KDE of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.show()
```

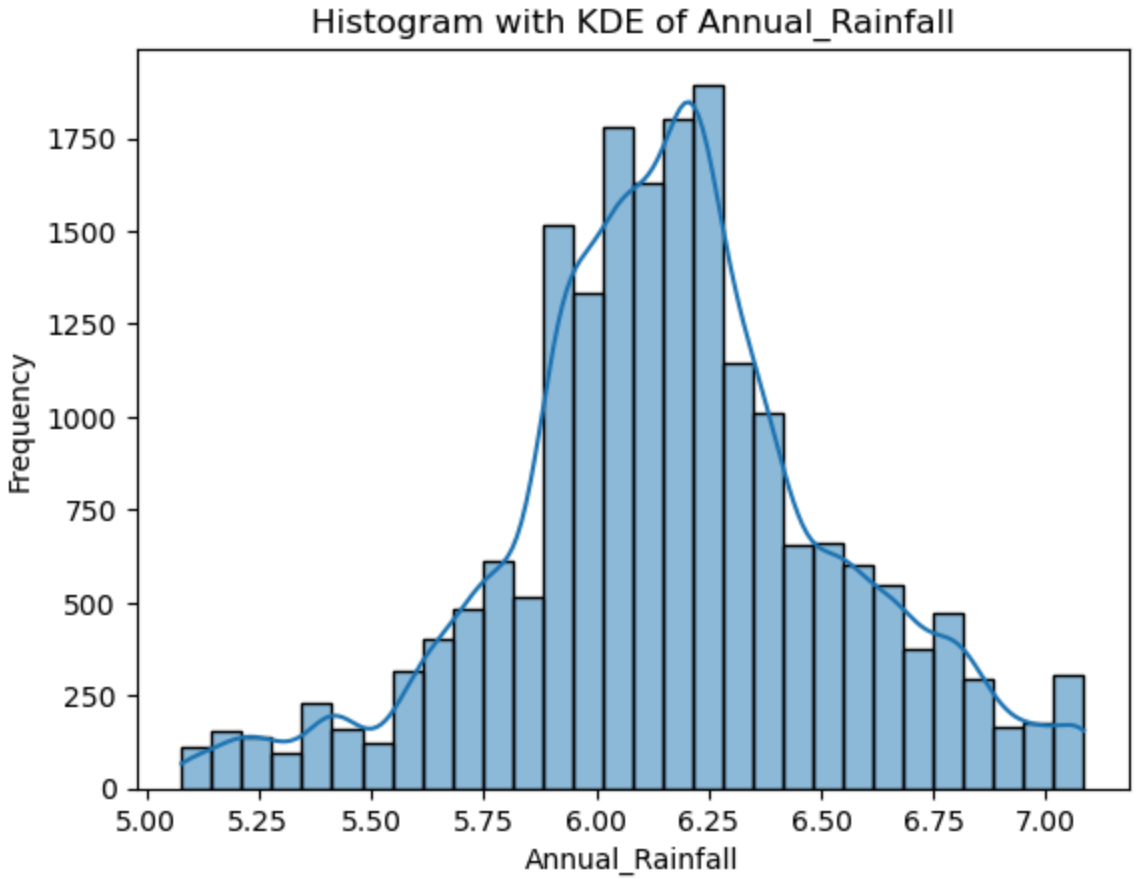


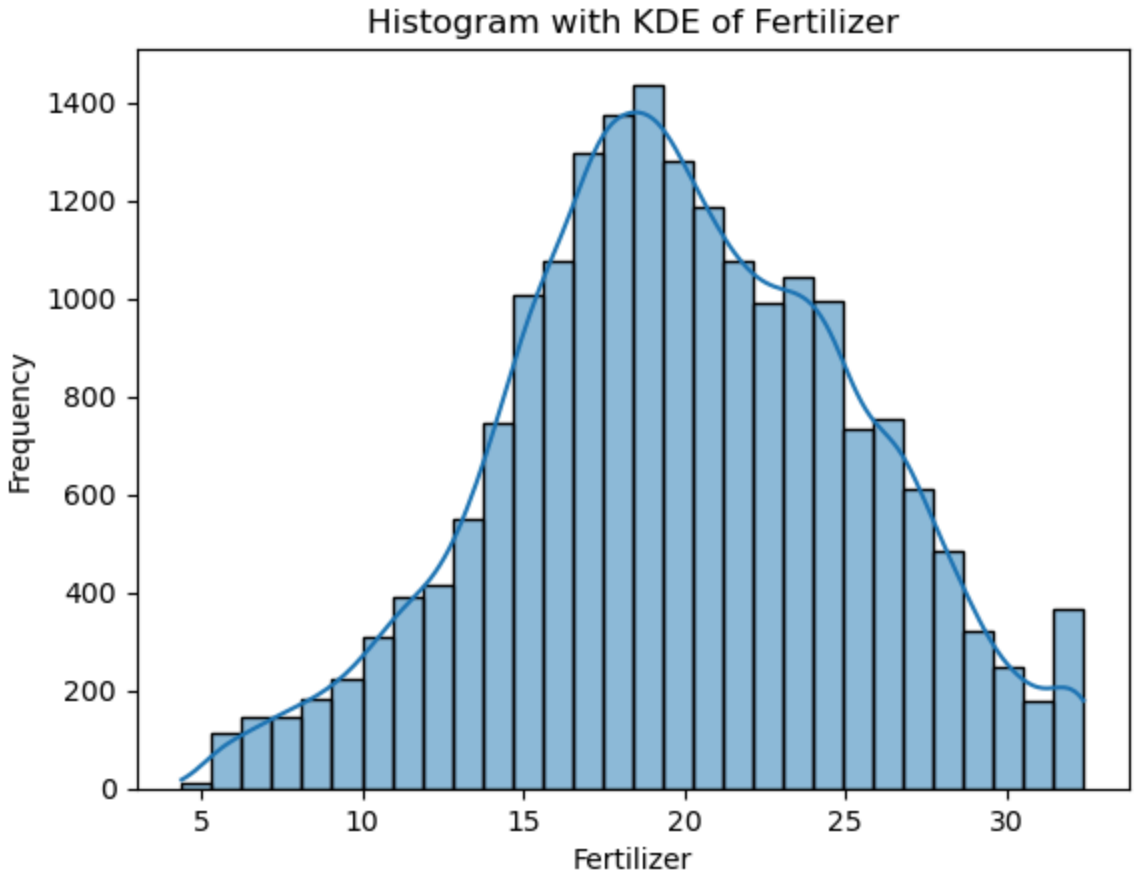

Histogram with KDE of Season
Frequency
Season
0
1000
2000
3000
4000
5000
6000
7000
8000
1
2
3
4
5
6

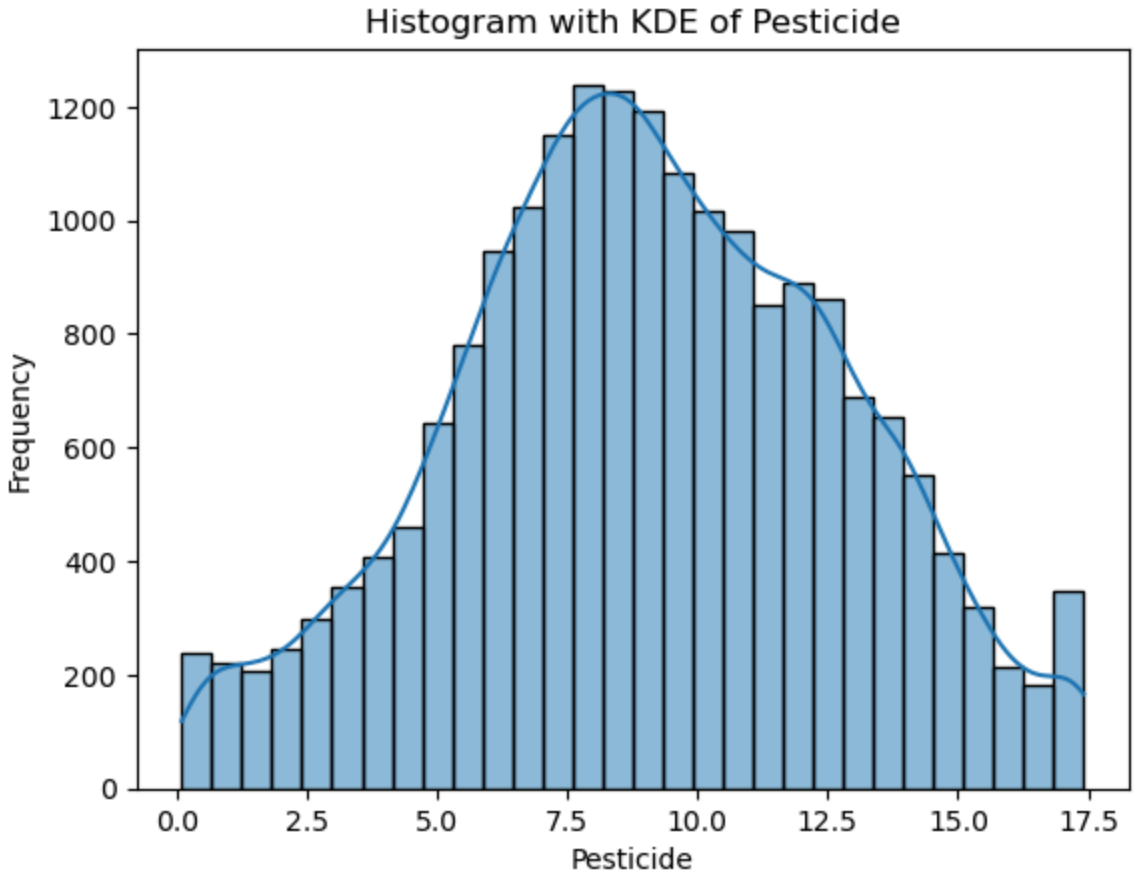

Histogram with KDE of State
Frequency
State
0
200
400
600
800
1000
1200
1400
0
5
10
15
20
25
30

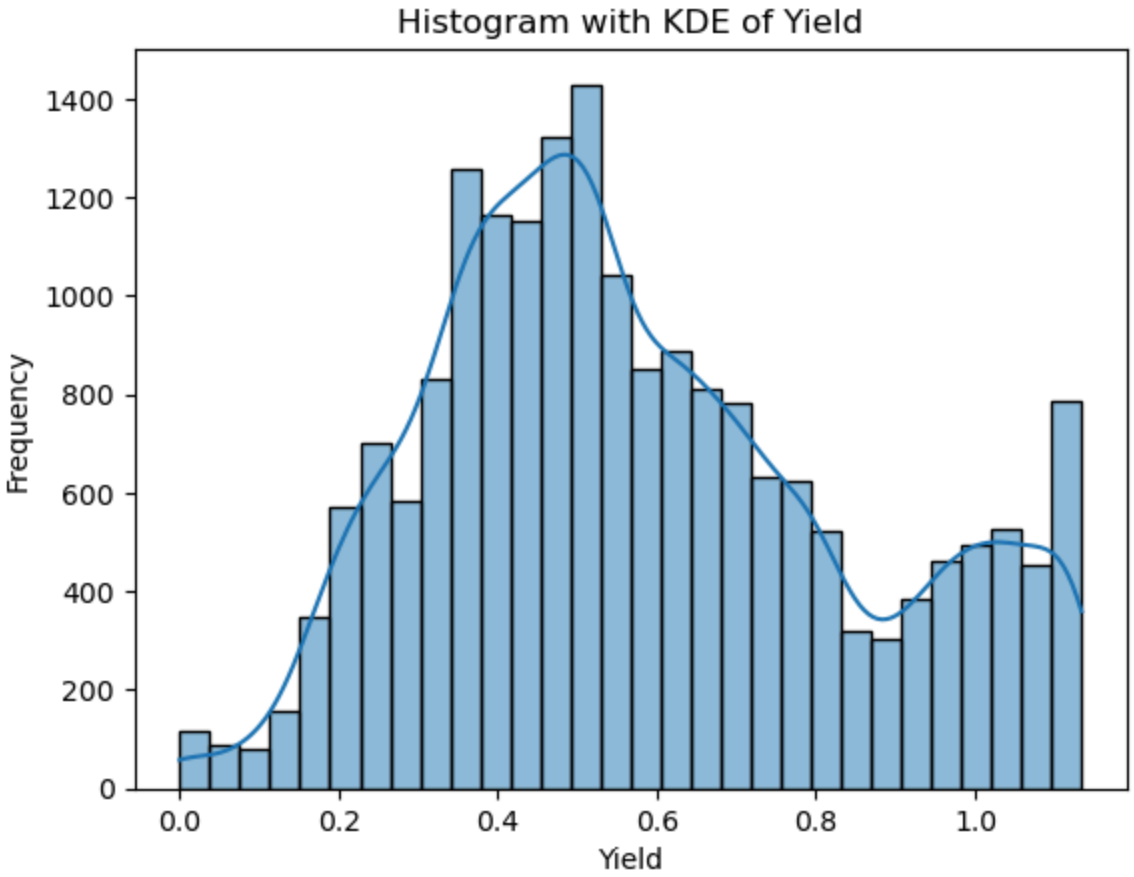

Histogram with KDE of Area
Frequency
Area
0
200
400
600
800
1000
1200
0
5
10
15
20


Histogram with KDE of Production
Frequency
Production
0
200
400
600
800
1000
1200
1400
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Histogram with KDE of Annual_Rainfall
Frequency
0
250
500
750
1000
1250
1500
1750
5.00
5.25
5.50
5.75
6.00
6.25
6.50
6.75
7.00
Annual_Rainfall


Histogram with KDE of Fertilizer
Frequency
Fertilizer
0
200
400
600
800
1000
1200
1400
5
10
15
20
25
30


Histogram with KDE of Pesticide
Frequency
0
200
400
600
800
1000
1200
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Pesticide



In [ ]:

Model's Algorithm Selection Note

There are total 8 independent features, Multicollinearity occurs, when one feature is highly linearly related to one or more other features, which can inflate the variance of the regression coefficients and make the model unstable. But features have weak correllation.

When a dataset exhibits weak pairwise correlations but high multicollinearity, it indicates a complex relationship between the independent variables.

When most features have weak correlations with the target variable, there is "Weak Correlation and High Multicollinearity" effectiveness of a "Decision Tree Regressor" has some reasons why it can be beneficial:

Decision Trees are less sensitive to multicollinearity because they split the data based on feature values. This reduces the direct impact of correlated features on model performance. Decision Trees can inherently select important features while ignoring less relevant ones during the splits, which might help in achieving high accuracy even in the presence of multicollinearity.

1. **Capturing Interactions:** Even though individual features have weak correlations; a Decision Tree can capture interactions between multiple features. These interactions might reveal patterns that are not obvious when looking at features in isolation.
2. **Handling Non-Linearity:** Decision Trees are good at capturing non- linear relationships. Features with weak linear correlations might still have non-linear relationships with the target variable that a Decision Tree can uncover.
3. **Feature Importance:** Decision Trees provide a ranking of feature importance. This can help identify which features contribute the most to the prediction, even if their individual correlations are weak.
4. **Flexibility:** Decision Trees can adapt to various kinds of data (categorical and numerical) and different scales, which is useful in datasets with diverse features.
5. **Combination with Ensemble Methods:** While a single Decision Tree might be limited, using ensemble methods like Random Forests or Gradient Boosting Machines can improve performance. These methods aggregate the predictions of multiple trees, reducing overfitting and increasing accuracy.

In summary, even with weak individual feature correlations and high multicollinearity a Decision Tree Regressor can be powerful by capturing complex interactions and non-linear relationships. It's also worth exploring ensemble methods to enhance model performance. Always consider comparing multiple models to ensure the best fit for your dataset.

In [ ]:

Scale the data using robust scaling.

In [29]:
```
from sklearn.preprocessing import RobustScaler, PowerTransformer, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

scaler = RobustScaler()
# Apply RobustScaler to the entire dataset
scaled_data = scaler.fit_transform(df_remove_outliers)
# Convert back to DataFrame for convenience
scaled_df = pd.DataFrame(scaled_data, columns=df_remove_outliers.columns)
```

Define a function for calculating Matrix for evaluate model.

In [30]:
```
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
# Evaluate metrics
def evaluate_model(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)
    return mse, rmse, mae, r2
```

In [ ]:
```
```

Define features and target

In [31]:
```
X = scaled_df.drop(columns=['Yield'])  # Features
y = scaled_df['Yield']  # Target
```

Split Data for Testing and Training with ratio 20:80

Purpose of random_state: For Reproducibility and Consistency, Ensures that you get the same results every time you run your code and allows them to reproduce your results exactly. This is important for research, debugging, and comparing model performance.

In [32]:
```
# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [33]:
```
# Train a Decision Tree Regressor
model = DecisionTreeRegressor(random_state=42)
model.fit(X_train, y_train)
```

Out[33]:
```
DecisionTreeRegressor(random_state=42)
```

In [34]:
```
# Predict on training and testing sets
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)
```

In [35]:
```
train_metrics = evaluate_model(y_train, y_train_pred)
test_metrics = evaluate_model(y_test, y_test_pred)
```

In [ ]:
```

```

In [36]:
```
# Create a DataFrame for tabular display
compare_df = pd.DataFrame({
    'Metric': ['Mean Squared Error', 'Root Mean Squared Error', 'Mean Absolute Error', 'R-squared'],
    'Training': [round(val, 2) for val in train_metrics],
    'Testing': [round(val, 2) for val in test_metrics]
})
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
print("       Decision Tree Training and Testing Metrics ")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")

print(compare_df)
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       Decision Tree Training and Testing Metrics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

	Metric	Training	Testing
0	Mean Squared Error	0.0	0.02
1	Root Mean Squared Error	0.0	0.15
2	Mean Absolute Error	0.0	0.09
3	R-squared	1.0	0.95

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Decision Tree Regressor: Visualize with Bar chart For Comparison of Training and Testing Metrics Accuracy.

In [37]:
```python
import matplotlib.pyplot as plt

# Plotting the bar chart
compare_df.set_index('Metric').plot(kind='bar', figsize=(8, 5))

plt.title('Decision Tree: Comparison of Training and Testing Metrics')
plt.xlabel('Metrics')
plt.ylabel('Values')
plt.xticks(rotation=45)
plt.legend(loc='center')
plt.show()
```

Analysis "Decision Tree" Model Accuracy And Explore Alternative:

Implemented a Decision Tree Regression model and achieved impressive results: Model provides **100% training accuracy and 95% testing accuracy.** While these numbers are promising, **it's essential to consider potential overfitting** and the model's generalization capability. **Random Forest Regression can often address these issues** and further enhance performance. This diversity helps prevent overfitting, leading to better generalization on unseen data.

Let's Improving Model Performance with Random Forest Regression.

Apply Random Forest Regression.

In [38]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Distribute Feature and Target
X = scaled_df.drop(columns=['Yield'])
y = scaled_df['Yield']

# Split into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the Random Forest model
rf = RandomForestRegressor(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Predictions for random forest
y_train_pred = rf.predict(X_train)
y_test_pred = rf.predict(X_test)
```

In [40]:
```
train_rf_metrics = evaluate_model(y_train, y_train_pred)
test_rf_metrics = evaluate_model(y_test, y_test_pred)
```

In [41]:
```
# Create a DataFrame for tabular display
compare_rf = pd.DataFrame({
    'Metric': ['Mean Squared Error', 'Root Mean Squared Error', 'Mean Absolute Error', 'R-squared'],
    'Training': [round(val, 2) for val in train_rf_metrics],
    'Testing': [round(val, 2) for val in test_rf_metrics]
})
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
print("       Random Forest: Training and Testing Metrics ")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
```

```
print(compare_rf)
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
       Random Forest: Training and Testing Metrics
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

	Metric	Training	Testing
0	Mean Squared Error	0.00	0.01
1	Root Mean Squared Error	0.04	0.10
2	Mean Absolute Error	0.02	0.06
3	R-squared	1.00	0.98

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

In [ ]:

:: Key Considerations ::

The "Random Forest Regression" potentially improve the model by preventing overfitting. Training data 100% accuracy and improving accuracy of the unseen data , testing data accuracy improving from 95% to 98%. Suggesting it might generalize better to new data. This typically suggests the model is balanced.

Acceptability: Yes, the model can be considered acceptable for deployment, as it achieves high accuracy on both training and testing data.

Interpretation of Accuracy: In terms of variance explained, it can approximate the testing performance as "98% accurate" and training as "100% accurate."

In [ ]: