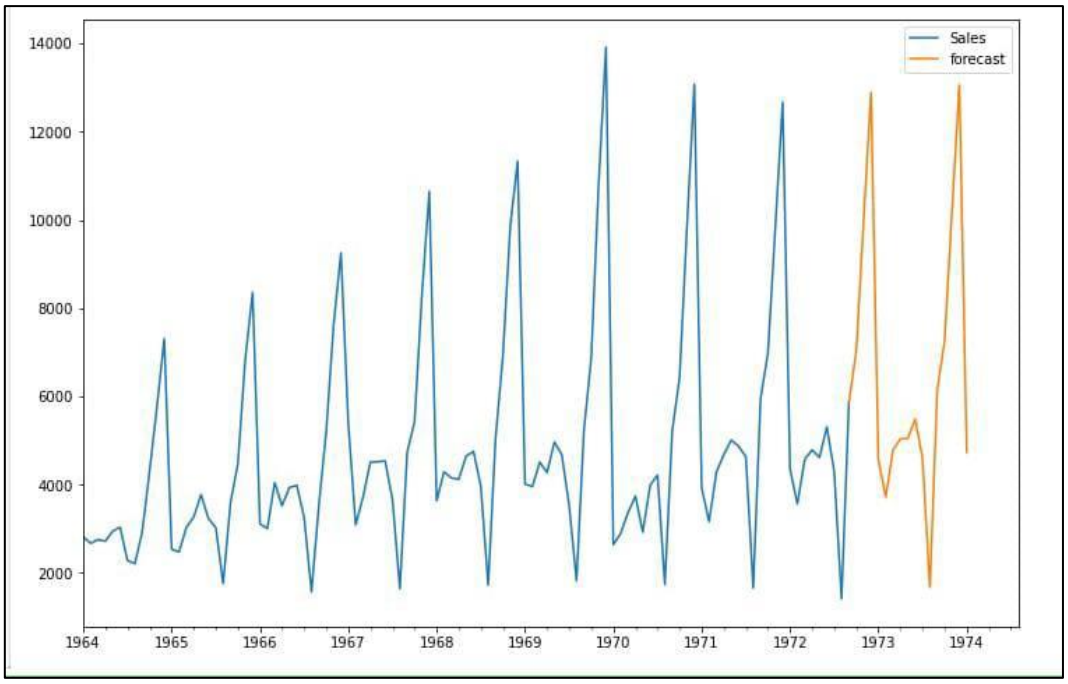
Institute of Data Engineering, Analytics and Science Foundation
Indian Statistical Institute, Kolkata

Implementing ML Algorithms in NPCYF



Mentor Name

Nushrat Hussain

Submitted By

Makhdum Hossain
Sananda Roy Chowdhury

TABLE OF CONTENTS

Contents

1. Introduction:
Rice, a staple food for a significant portion of India's population, plays a crucial role in the country's food security. Accurately forecasting rice production is essential for effective policy-making, resource allocation, and price stabilization.

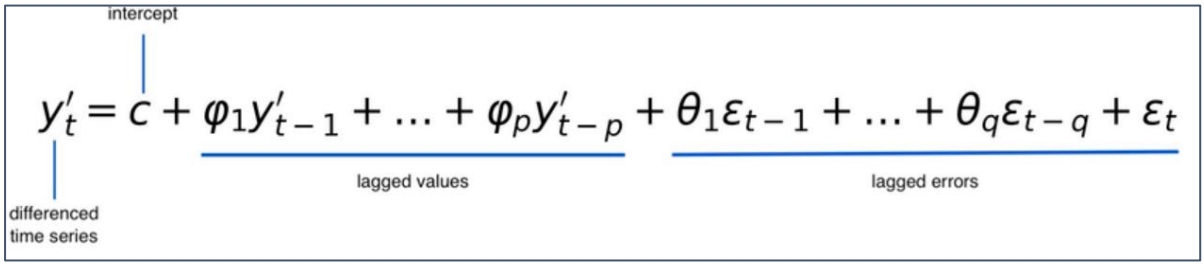
2. Problem Statement:
This report addresses the challenge of forecasting rice production in India by developing a robust forecasting model that incorporates historical data on rice yield, cultivated area, and their interdependencies at the state and district levels.

3. Objectives:
- To analyze historical trends and patterns in rice yield and cultivated area across Indian states and districts from 1997 to 2023.
- To develop and validate an ARIMA (Autoregressive Integrated Moving Average) model for forecasting rice yield and cultivated area at the district level, considering the temporal dependencies within the data.
- To estimate future rice production by integrating the forecasted yields and cultivated areas, providing insights into potential production levels for the coming years.
- To evaluate the accuracy and reliability of the proposed forecasting model using appropriate statistical metrics.

4. Methodology and Strategy:
To full all the objectives of the given problem statement, ARIMA model was picked so as to develop a robust forecasting model which predict

ARIMA MODEL- What and Why?
ARIMA stands for **Autoregressive Integrated Moving Average**. It's a powerful statistical model used for time series forecasting — predicting future values based on the patterns in past data points collected over regular intervals (e.g., daily stock prices, monthly sales, yearly rainfall).

Mathematical Equation of ARIMA Model



Components of ARIMA:
ARIMA models are defined by three key parameters: (p, d, q)

- **AR (Autoregressive):** Uses a linear combination of *past values* of the time series itself to predict future values. **The 'p' parameter determines how many past values (lags) are included.**
- **I (Integrated):** Accounts for the *differencing* of the time series to make it stationary (removing trends or seasonality). **'d' represents the number of times differencing is applied.**

- **MA (Moving Average):** Incorporates the average of *past forecast errors* to improve predictions. **The 'q' parameter indicates how many past errors are considered.**

Why Use ARIMA for Forecasting?

- **Effective for Time-Dependent Data:** ARIMA models excel at capturing temporal dependencies (how past values influence future ones) inherent in time series data.
- **Handles Trends and Seasonality:** The 'I' (integration) component helps address trends and seasonality, making the time series suitable for modelling.
- **Well-Established and Interpretable:** ARIMA models have a strong theoretical foundation and are relatively interpretable, allowing you to understand the factors driving your forecasts.

When to Use ARIMA:

- We have a time series dataset with a clear pattern of autocorrelation (correlation between past and present values).
- The time series is or can be made stationary (relatively consistent mean and variance over time) using ***Dickey Fuller Test***.
- We need a model that can provide both point forecasts (single value predictions) and confidence intervals.

RESULTS AND DISCUSSIONS:

DATASET:

Before we move into results and discussions, let us first learn a bit about the dataset we are using for better understanding.

	A	B	C	D	E	F
1	State	District	Year	Area(Hectare)	Production(Tonnes)	Yield(Tonne/Hectare)
2	Andaman and Nicobar Islands	Nicobars	2000 - 2001	102	321	3.15
3	Andaman and Nicobar Islands	Nicobars	2001 - 2002	83	300	3.61
4	Andaman and Nicobar Islands	Nicobars	2002 - 2003	189.2	510.84	2.7
5	Andaman and Nicobar Islands	Nicobars	2003 - 2004	52	90.17	1.73
6	Andaman and Nicobar Islands	Nicobars	2004 - 2005	52.94	72.57	1.37
7	Andaman and Nicobar Islands	Nicobars	2005 - 2006	2.09	12.06	5.77

Fig: Rice.xlsx(Download link: https://shorturl.at/jSE9s)

So the dataset that we are working with contains 6 columns namely **STATE**(which contains the Indian States), **DISTRICTS**(the districts of various states), **YEAR**(in YYYY-YYYY format), **AREA**(in Hectares) the area in which crop is grown, **PRODUCTION**(in tonnes) indicates how much rice in Tonnes is grown and Yield(Tonne/Hectares) implies the amount of rice grown from a certain area and it can also be obtained by dividing the columns**(YIELD/AREA).**

Summing up all, the dataset contains the information about the Yield of Rice(Yield is nothing but Amount of Rice grown(here in Tonnes)/Area in which it is grown(here in Hectare)) from a certain district of respective State in a tabular fashion.

NOTE: The dataset we are using should be cleaned and processed before applying ARIMA model for accurate result.

We will be using PYTHON for the entire code and let us walk you with certain IDEs(Integrated Development Environment) wherein we can code!!

IDEs(Integrated Development Environment):

PyCharm: PyCharm is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools for Python developers, tightly integrated to create a convenient environment for productive Python, web, and data science development. Download Link: https://www.jetbrains.com/pycharm/download/?section=windows

VSCode: Visual Studio Code, often referred to as VS Code, is a popular and versatile code editor developed by Microsoft. It's designed to be a lightweight yet powerful tool for various programming languages, including: Web Development, Backend Development, Data Science, Mobile Development. Download Link: https://code.visualstudio.com/download.

Jupyter Notebook: Interactive Computing Environment- Jupyter Notebook is a web-based application that allows you to create and share documents containing live code, equations, visualizations, and explanatory text Install Link: https://jupyter.org/install.

There are several other IDEs as well. Please consider your system information before downloading IDEs else it won't be compatible with your machine (System type: 32bit/64 bit and OS: WINDOWS/LINUX/MAC).

We have used Jupyter Notebook based on our ease of use.

Step1: Installing and Importing PYTHON Libraries.

Python libraries are collections of pre-written code that provide functions, classes, and methods to help developers perform various tasks without needing to write everything from scratch. Libraries can simplify coding by offering reusable components for specific functionalities.

1. **Pandas:** Pandas is a data-analysis library that provides high-level data structures and robust data analysis tools. It is used for data wrangling, cleaning, and preparation. It is designed to make data manipulation and analysis easy and intuitive.

   Install: pip install Pandas
   Use: import pandas as pd

2. **NumPy:** NumPy is a scientific computing library for Python. It provides powerful tools for manipulating and analysing numerical data. It is used for array-based computations, linear algebra, Fourier transforms random number functions, and more.

   Install: pip install numpy
   Use: import numpy as np

3. **Scikit-learn:** Scikit-learn is a machine-learning library for Python. It provides tools for supervised and unsupervised learning, data preprocessing, model selection, etc. It is designed to be easy to use, efficient, and robust.

   Install: pip install sklearn
   Use: import sklearn

4. **Seaborn:** Seaborn is a data visualization library for Python. It provides high-level plotting functions for creating attractive and informative visualizations. It is optimized for working with pandas' data structures and can integrate with NumPy and Scikit-learn.

Install: pip install seaborn
Use: import seaborn as sns

5. StatsModel: **Statsmodels** is a Python library used for statistical modelling, hypothesis testing, data exploration and forecasting data. It provides tools for estimating and testing statistical models, particularly in the context of time series analysis, regression analysis, and more.

 Augmented Dickey-Fuller Test (adfuller): This test is used to check if a time series is stationary. The null hypothesis is that the time series has a unit root, meaning it is non-stationary.
 Autocorrelation and Partial Autocorrelation Plots (plot_acf, plot_pacf): These plots help to visualize the autocorrelation and partial autocorrelation of a time series, which can assist in identifying the order of ARIMA models.
 ARIMA Model (ARIMA): This class is used to fit an ARIMA model to the time series data. ARIMA stands for AutoRegressive Integrated Moving Average and is a popular model for forecasting.
 Mean Squared Error (mean_squared_error): This metric helps assess the accuracy of your forecasts by measuring the average of the squares of the errors (differences between actual and predicted values).
 Mean Absolute Percentage Error (mean_absolute_percentage_error): This metric provides a percentage error measurement, making it easier to interpret the accuracy of forecasts across different scales.

6. **Matplotlib:** Matplotlib is a popular data visualization library in Python. It's often used for creating static, interactive, and animated visualizations in Python. Matplotlib allows you to generate plots, histograms, bar charts, scatter plots, etc., with just a few lines of code.

Install: Pip install matplotlib
Use: import matplotlib.pyplot as plt

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, mean_absolute_percentage_error
import warnings
warnings.filterwarnings('ignore')
```

Importing the warnings Module: This module provides a way to manage warnings in your code. The filterwarnings function allows you to specify how warnings should be handled. In this case, 'ignore' means that all warnings will be suppressed and not displayed.

Step 2: Data Loading and Pre-Processing:

```
def load_and_preprocess_data(file_path):
    df = pd.read_excel(file_path)
    df['Year'] = df['Year'].apply(lambda x: pd.to_datetime(x.split('-')[0] + '-01-01'))
    df.set_index('Year', inplace=True)
    df = df.dropna(subset=['Area(Hectare)', 'Production(Tonnes)', 'Yield(Tonne/Hectare)'], how='all')
    return df
```

Let us analyse what we have coded here:

A function named **" load_and_preprocess_data(file_path):"** that accepts one argument, **file_path**, which should be a string representing the path to the Excel file.

Inside the function, this line uses the "**pd.read_excel"** function to read the data from the Excel file specified by "**file_path"**. The data is stored in a pandas DataFrame named **df**.

Now we come to a crucial part where we convert "Year" column to datetime objects.

df['Year'] = df['Year'].apply(lambda x: pd.to_datetime(x.split('-')[0] + '-01-01'))

Imagine your initial DataFrame **df** looks like this:

	Year	Area(Hectare)	Production(Tonnes)
0	2020-21	100.0	500.0
1	2021-22	120.0	600.0
2	2022-23	150.0	750.0

This line is crucial for time series analysis. It assumes that the 'Year' column in your Excel file might be in a format like "2023-24" or "2022-23."

- It first splits the string in the 'Year' column by the hyphen ("-").
- Then, it takes the first part (e.g., "2023") and appends "-01-01" to it, creating a string in the format "YYYY-01-01."

- Finally, it uses **pd.to_datetime** to convert this string into a pandas datetime object, making it suitable for time-based indexing.

```
1          Year  Area(Hectare)  Production(Tonnes)
2  0  2020-01-01          100.0                 500.0
3  1  2021-01-01          120.0                 600.0
4  2  2022-01-01          150.0                 750.0
```

Then we, Set 'Year' as the index

The code **df.set_index('Year', inplace=True**) then takes the **'Year'** column (now with datetime objects) and sets it as the index of the DataFrame.

```
1             Area(Hectare)  Production(Tonnes)
2  Year
3  2020-01-01          100.0                 500.0
4  2021-01-01          120.0                 600.0
5  2022-01-01          150.0                 750.0
```

Now we have a time series DataFrame with the 'Year' as its index, making it easier to perform time-based operations for analysis and forecasting.

To make the data consistent, we have to remove rows with null. This line df = **df.dropna(subset=['Area(Hectare)', 'Production(Tonnes)', 'Yield(Tonne/Hectare)'], how='all')** removes any rows where *all* the columns **'Area(Hectare)', 'Production(Tonnes)'**, and **'Yield(Tonne/Hectare)'** have missing values (**NaN**). **how='all'** ensures that only rows with missing values in ***all*** **three columns** are removed. Following that we return the data frame using **return df.**

Hence our dataframe is now ready for time series analysis or other operations.

Step 3: Data Preparation for Time Series Analysis with ARIMA Model

Before applying the ARIMA model, it's essential to determine whether the time series data is stationary, as ARIMA performs best with stationary data. To verify stationarity, we use the Dickey-Fuller test, a standard statistical test to detect stationarity.

```python
def check_stationarity(timeseries):
    result = adfuller(timeseries, autolag='AIC')
    print('\nAugmented Dickey-Fuller Test Results:')
    print(f'ADF Statistic: {result[0]}')
    print(f'p-value: {result[1]}')
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')
    return result[1] <= 0.05
```

The **check_stationarity(timeseries)** function assesses the stationarity of a given time series by utilizing the Augmented Dickey-Fuller (ADF) test. This method is commonly employed in time series analysis and provides statistical insights to determine whether the series exhibits constant mean and variance over time.

The function performs the ADF test, and the results include:

- **ADF Statistic**: A value that helps assess stationarity.
- **p-value**: The p-value indicates if the series is stationary (a p-value ≤ 0.05 suggests stationarity).
- **Critical Values**: Thresholds for significance at different levels (1%, 5%, and 10%).

The function returns True if the series is stationary, and False otherwise.

```
def difference_series(series):
    return series.diff().dropna()
```

The **"difference_series(series)"** function is designed to help make a series stationary by applying first-order differencing.

The function works by subtracting each data point from the previous one, and it then uses **dropna()** to remove any resulting **NaN** values, preparing a stationary series for further analysis.

```
Checking stationarity of original series:

Augmented Dickey-Fuller Test Results:
ADF Statistic: -2.111496135694762
p-value: 0.2399171449681302
Critical Values:
        1%: -3.7238633119999998
        5%: -2.98648896
        10%: -2.6328004

Time series is not stationary. Applying differencing.

Augmented Dickey-Fuller Test Results:
ADF Statistic: -6.323310184857089
p-value: 3.028718835620696e-08
Critical Values:
        1%: -3.7377092158564813
        5%: -2.9922162731481485
        10%: -2.635746736111111
```

The output shows the results of the **Augmented Dickey-Fuller** test for stationarity. The original time series is not stationary. After applying **differencing**, the time series becomes stationary as the **p-value** is less than 0.05, suggesting that the **null hypothesis** of **non-stationarity** is **rejected.**

Since we have made the data points stationary, we now will soon start to forecast for the data. Before forecasting it will be best if we visualize the data.

Till this, the process of preparing the data remains same for all three parameters namely yield, area and production. Now onwards, we have to focus on individual parameters.

Step 4.1: FOR YIELD

Step 4.1.1: FILTERING THE DATA:

```
def analyze_agricultural_data(state, district, df):
    data = df[(df['State'] == state) & (df['District'] == district)].copy()

    if data.empty:
        print(f"No data found for {district}, {state}")
        return None

    yield_data = data['Yield(Tonne/Hectare)']

    if len(yield_data) < 10:
        print(f"Warning: Limited data available ({len(yield_data)} points). Some analyses may be limited.")
```

This line **def analyze_agricultural_data(state, district, df):** defines a function named **analyze_agricultural_data** that takes three parameters:

- **state**: A string representing the name of the state you want to analyze.
- **district**: A string representing the name of the district within that state.
- **df**: A pandas Data Frame that contains agricultural data, including yield information.

Now **data = df[(df['State'] == state) & (df['District'] == district)].copy()** filters the Data Frame df to create a new Data Frame called 'data'. The filtering is done using two conditions combined with the **&** operator:

- **df['State'] == state**: Checks if the 'State' column matches the given **state**.
- **df['District'] == district**: Checks if the 'District' column matches the given **district**.

The. copy**()** method is used to create a copy of the filtered Data Frame. This is important because it prevents any modifications made to **data** from affecting the original Data Frame **df.**

```
    if data.empty:
        print(f"No data found for {district}, {state}")
        return None
```

This block checks if the filtered Data Frame **data** is empty (i.e., it contains no rows).

- If **data** is empty, it means that there are no records for the specified **state** and **district**.
- The **print** statement outputs a message indicating that no data was found for the specified district and state.
- The **return None** statement exits the function early, indicating that further analysis cannot be performed due to the lack of data.

```
    yield_data = data['Yield(Tonne/Hectare)']

    if len(yield_data) < 10:
        print(f"Warning: Limited data available ({len(yield_data)} points). Some analyses may be limited.")
```

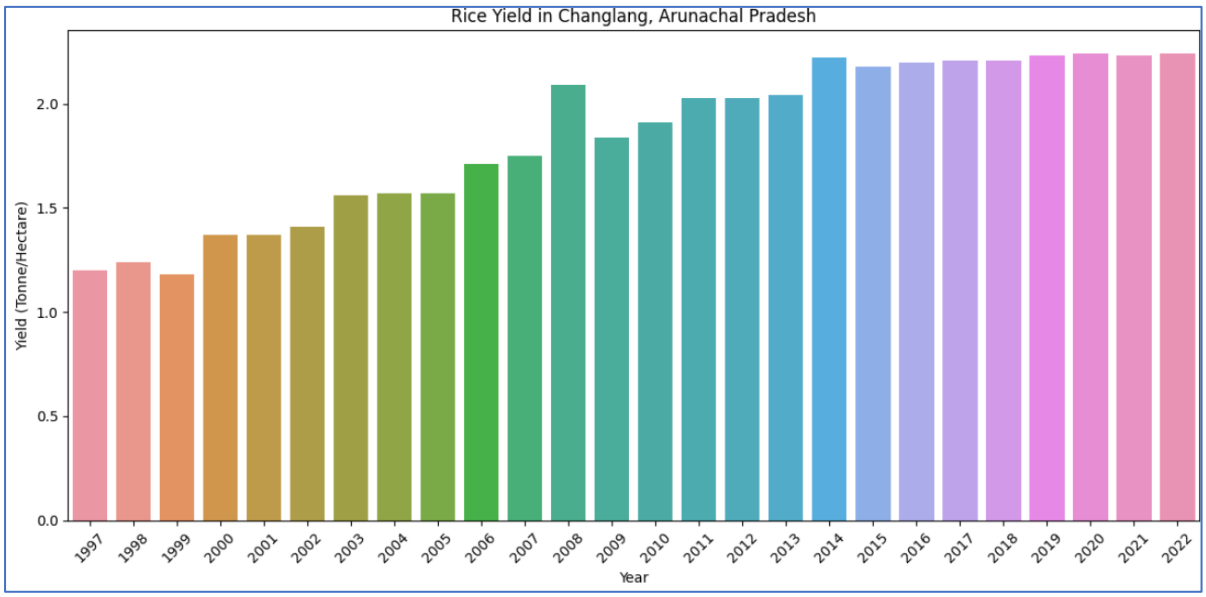
If the **data** Data Frame is not empty, this line extracts the 'Yield(Tonne/Hectare)' column from **data** and assigns it to a new variable called **yield_data**.

yield_data is now a pandas Series that contains the yield values for the specified district and state.

The **len(yield_data)** function returns the number of entries in the Series. If there are fewer than 10 data points, it prints a warning message indicating that the available data is limited and that some analyses may not be reliable or may be limited in scope.

Step 4.1.2: Visualizing the data:

The overall purpose of this code is to visually represent the historical yield data of rice for a specific district and state, making it easier to analyze trends over time. The bar plot provides a clear and informative visualization of how rice yields have varied across different years.



1. Figure Creation:
    - A new figure is created with a specified size of 12 inches wide and 6 inches tall using plt.figure(fig size=(12, 6)).
2. Bar Plot Generation: A bar plot is created using Seaborn's **sns.barplot()** function:
    - The **x-axis represents** the **years extracted from the index of the yield_data Series.**
    - The **y-axis represents** the **yield values (in Tonne/Hectare)** from yield_data.
3. Title and Labels:
    - The plot is titled "Rice Yield in {district}, {state}", dynamically including the district and state.
    - The **x-axis is labelled "Year" and the y-axis is labeled "Yield (Tonne/Hectare)"**.
4. X-axis Tick Rotation:
    - The x-axis tick labels are rotated by 45 degrees to enhance readability.
5. Layout Adjustment:

- **plt.tight_layout()** is called to automatically adjust the subplot parameters for better spacing and to prevent overlapping of elements.

6. Display the Plot:
    - Finally, **plt.show()** is called to render and display the plot.

Step 4.1.3 : ACF and PACF plot for ARIMA Model Selection includes the value of p,q and d.

Before proceeding let us first study about autocorrelation and partial autocorrelation that would help us strengthen our concept.

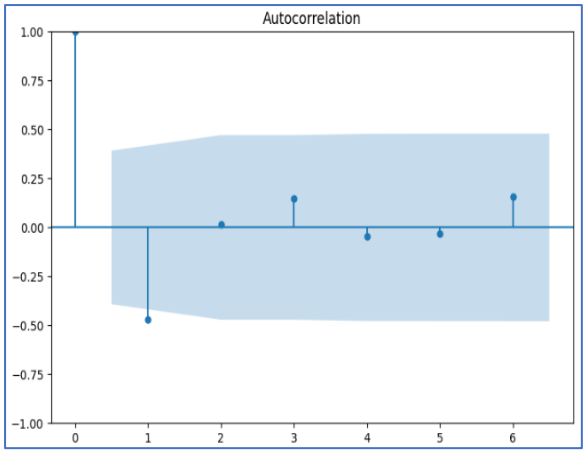
- **ACF** provides insights into the overall correlation structure of a time series, indicating how past values influence current values.
- **PACF** focuses on the direct relationships at specific lags, helping to identify the order of autoregressive terms in time series models.

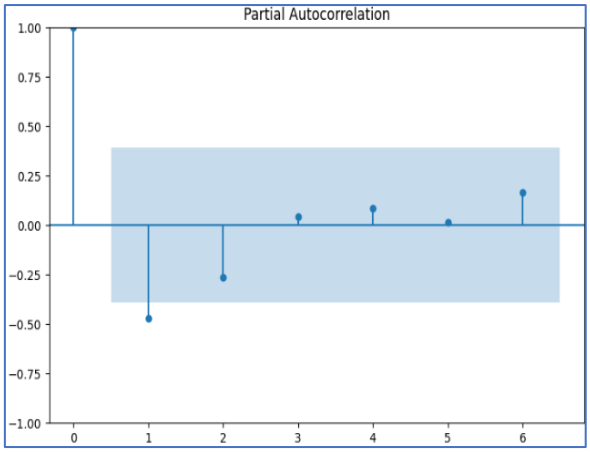
```
max_lags = max(1, int(len(yield_diff) * 0.3) - 1)

if len(yield_diff) > 4:
    try:
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
        plot_acf(yield_diff, ax=ax1, lags=max_lags)
        plot_pacf(yield_diff, ax=ax2, lags=max_lags)
        plt.tight_layout()
        plt.show()
    except Exception as e:
        print(f"\nCould not generate ACF/PACF plots: {str(e)}")

p = min(1, max_lags)
q = min(1, max_lags)
```

The line max_lags = max(1, int(len(yield_diff) * 0.3) - 1) calculates the maximum number of lags to consider for ACF and PACF plots. It takes 30% of the length of **yield_diff**, converts it to an integer, and subtracts 1. The result is ensured to be at least 1.

Then we check the length of the number of datapoints and if the no of datapoints are more than 4 then it is sufficient for data analysis.

Now we come to the autocorrelation and correlation function and if the length condition is met, it attempts to create two subplots side by side: one for the autocorrelation function (ACF) and one for the partial autocorrelation function (PACF) of **yield_diff**. If an error occurs during plotting, it catches the exception and prints an error message for that we have used exclusively a try and catch block.





Step 4.1.4: ARIMA model plot for the 'YIELD' Parameter:

```
try:
    model = ARIMA(yield_data, order=(p, d, q))
    results = model.fit()

    forecast_steps = 5
    forecast = results.forecast(steps=forecast_steps)

    # Scatter plot with trend line for historical data and forecast
    plt.figure(figsize=(12, 6))
    sns.regplot(x=np.arange(len(yield_data)), y=yield_data.values, scatter_kws={'s': 50}, label='Historical')

    last_date = yield_data.index[-1]
    future_dates = pd.date_range(start=last_date, periods=forecast_steps + 1, freq='Y')[1:]

    plt.scatter(np.arange(len(yield_data), len(yield_data) + forecast_steps), forecast, color='red', s=50, label='Forecast')
    plt.plot(np.arange(len(yield_data), len(yield_data) + forecast_steps), forecast, color='red', linestyle='--')

    plt.title(f'Rice Yield Trend and Forecast for {district}, {state}')
    plt.xlabel('Time Steps')
    plt.ylabel('Yield (Tonne/Hectare)')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

ARIMA(yield_data, order=(p, d, q)): This initializes the ARIMA model with parameters (p) (autoregressive order), (d) (degree of differencing), and (q) (moving average order).
results = model.fit(): This fits the ARIMA model to the data.

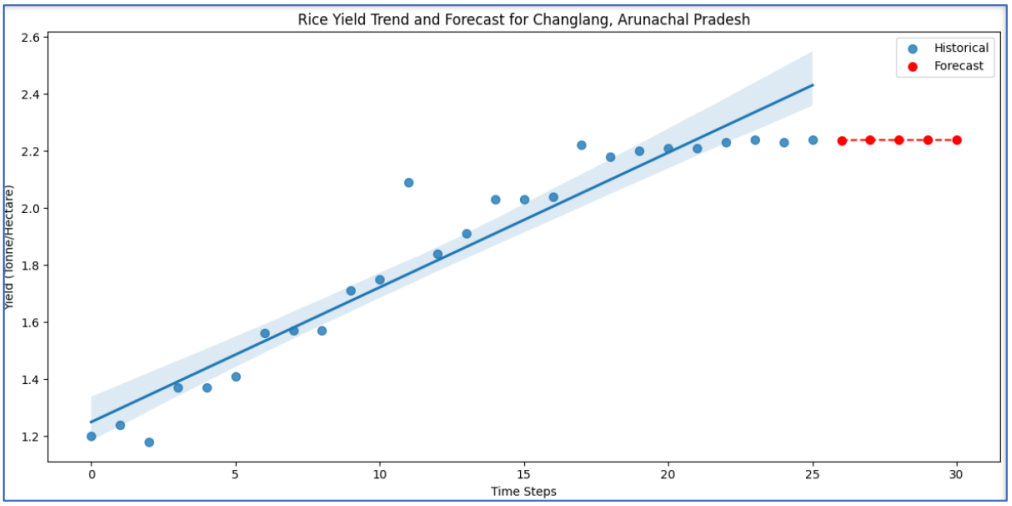
Now since we want to forecast for next 5 years we have put the value of forecast_steps=5.
forecast = results.forecast(steps=forecast_steps): This generates the forecast for the specified number of steps ahead

Following that we have made a scatter plot for plotting the historial data as well as the forecast.

- **plt.figure(figsize=(12, 6))**: Sets the size of the figure for the plot.
- **sns.regplot(...)**: This creates a scatter plot of the historical data points with a regression line. The **x** values are the indices of **yield_data**, and **y** values are the actual yield values.
- plt.scatter(...): Plots the forecasted values as red dots.

- plt.plot(...): Connects the forecasted points with a dashed line.

Now the scatter plot looks something like this:



Step 4.1.5 : Splitting the data into training and testing set

```
    if len(yield_data) >= 5:
        train_size = int(len(yield_data) * 0.8)
        train, test = yield_data[:train_size], yield_data[train_size:]

        model_test = ARIMA(train, order=(p, d, q))
        results_test = model_test.fit()
        forecast_test = results_test.forecast(steps=len(test))
```

First, we check if the data set has more than 5 sets which is necessary for splitting the data into training and testing set.

- **train_size = int(len(yield_data) * 0.8)**: Calculates the size of the training set as 80% of the total data.
- **train, test = yield_data[:train_size], yield_data[train_size:]**: Splits the data into training (first 80%) and testing (remaining 20%).

We then fit the training set into the ARIMA model and forecast for the length of the test set.

Step 4.1.6: Performance Metric Calculation

```
    rmse = np.sqrt(mean_squared_error(test, forecast_test))
    mape = mean_absolute_percentage_error(test, forecast_test) * 100

    print("\nModel Performance Metrics:")
    print(f"RMSE: {rmse:.2f}")
    print(f"MAPE: {mape:.2f}%")
else:
    print("\nInsufficient data for performance metrics calculation")
```

1. **RMSE**: Calculates the root mean squared error between the actual test values and the forecasted values.
2. **MAPE**: Calculates the mean absolute percentage error, providing a percentage error measure

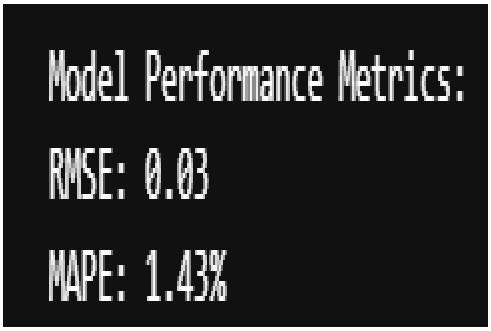


Fig: MAPE and RMSE score of our model

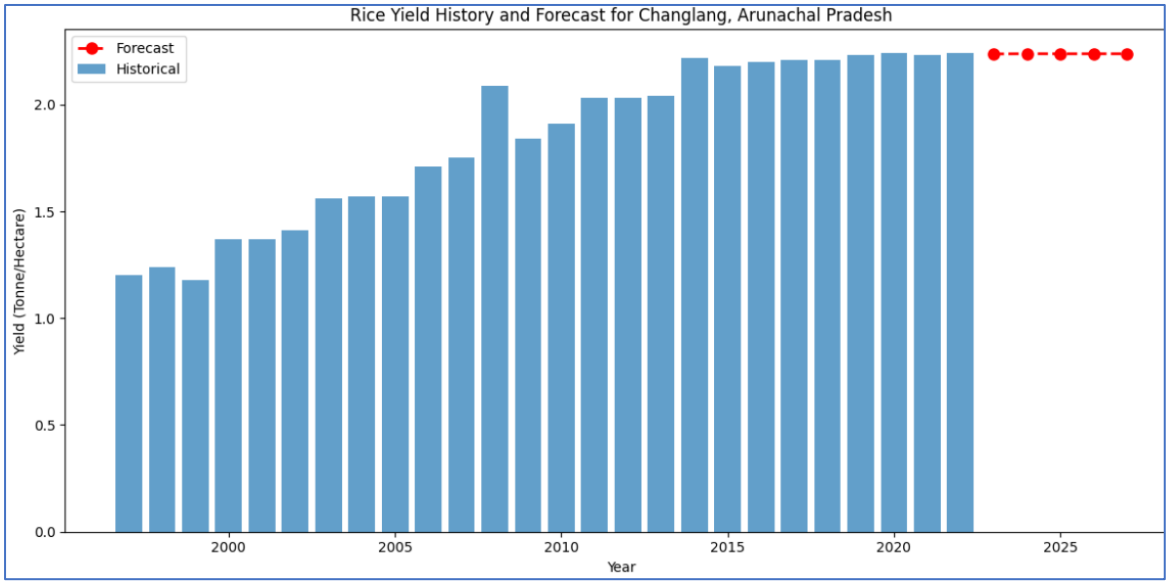
- **RMSE:**
  - Good scores are context-dependent; lower values are better.
  - Compare RMSE to the scale of the target variable.
- **MAPE:**
  - **0% to 10%**: Excellent
  - **10% to 20%**: Good
  - **20% to 50%**: Acceptable
  - **Above 50%**: Poor

Fig: Standard MAPE and RMSE score template

Step 4.1.7: Bar and Line Plot for forecast:

```
# Bar and line plot for forecast
plt.figure(figsize=(12, 6))
plt.bar(yield_data.index.year, yield_data.values, alpha=0.7, label='Historical')
plt.plot(future_dates.year, forecast, color='red', marker='o', linestyle='--', linewidth=2, markersize=8, label='Forecast')
plt.title(f'Rice Yield History and Forecast for {district}, {state}')
plt.xlabel('Year')
plt.ylabel('Yield (Tonne/Hectare)')
plt.legend()
plt.tight_layout()
plt.show()
```

The bar and line plot visualizes the historical rice yield data and its forecast for the next five years. The vertical bars represent the historical yield (in tonnes per hectare) for each year, while the red dashed line indicates the forecasted yield for the upcoming years, marked with circular points. This combined visualization allows for easy comparison between past performance and future expectations, providing insights into trends in rice yield over time. The title specifies the district and state, giving context to the data presented.



```
    print("\nYield Forecast for the next 5 years:")
    for year, value in zip(future_dates, forecast):
        print(f"{year.year}: {value:.2f} tonne/hectare")
```

This code snippet is to present the forecasted yield data in a clear and organized manner. It provides users with an easy-to-read summary of expected yields for the next five years, which can be useful for planning and decision-making in agricultural contexts.

```
Yield Forecast for the next 5 years:
2023: 2.12 tonne/hectare
2024: 2.13 tonne/hectare
2025: 2.12 tonne/hectare
2026: 2.13 tonne/hectare
2027: 2.13 tonne/hectare
```

Step 4.1.8: Creating the Search Interface and the Main Method():

```
def main():
    try:
        file_path = "rice.xlsx"  # Replace with your file path
        print("\nLoading and preprocessing data...")
        df = load_and_preprocess_data(file_path)

        print("\nAgricultural Yield Analysis")
        print("---------------------------")
        state = input("Enter the state name: ")
        district = input("Enter the district name: ")

        data = analyze_agricultural_data(state, district, df)

    except Exception as e:
        print(f"\nAn error occurred: {str(e)}")
        print("Please check your input data and try again.")

if __name__ == "__main__":
    main()
```

This part of the code helps create user a search box wherein they can enter the name of the state and the district.

Let us see how it works!

```
Loading and preprocessing data...

Agricultural Yield Analysis
---------------------------
Enter the state name:  Arunachal Pradesh
Enter the district name: [↑↓ for history. Search history with c-↑/c-↓                ]
```

The Up and Down Arrow key helps user access the history of their search as well.

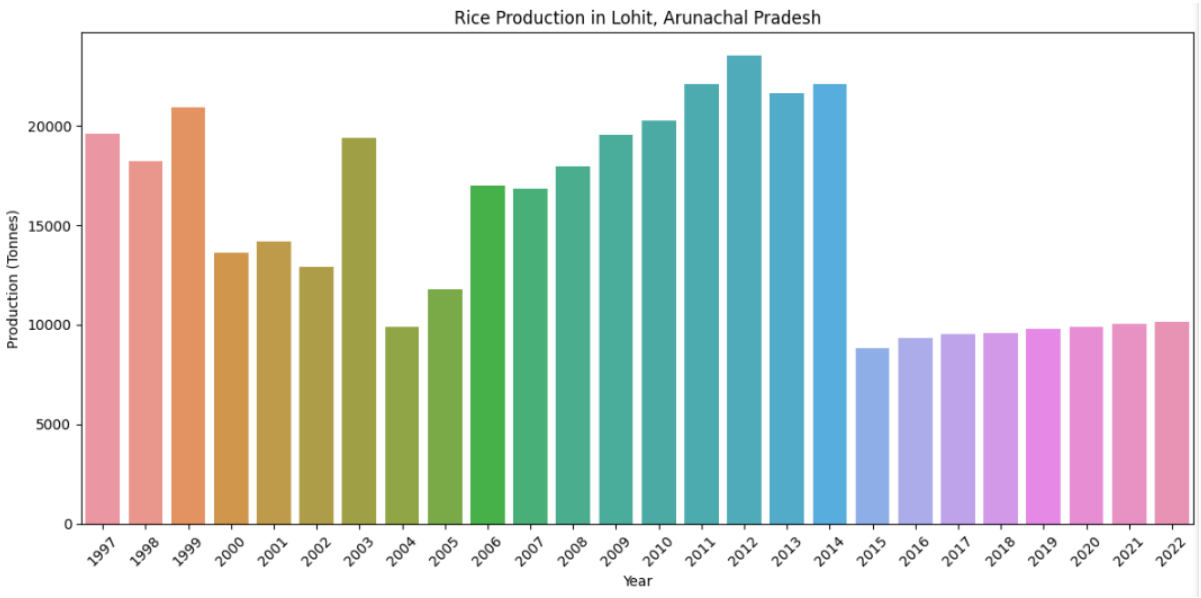
Step 4.2: FOR PRODUCTION

Step 4.2.1: FILTERING THE DATA:

- The function “analyze_agricultural_data”, first filters rows in ‘df’ where both the 'State' and 'District' columns match the specified `state` and `district` parameters. ‘data’ is a copy of this filtered dataset, containing only records for the chosen state and district.
- If `data` is empty (meaning no records were found for the specified location), the function displays a message indicating no data is available for that particular district and state, then exits early.
- The function then extracts the 'Production (Tonnes)' column from the filtered data, which represents the production figures in tonnes.
- It checks if there are fewer than 10 data points in this column, warning the user if data is limited.

Step 4.2.2: Visualizing the data for PRODUCTION:

This step creates a bar plot to visualize historical rice production data for a specific district and state, making it easier to observe production trends over the years.



1. Set Plot Size: Define the plot size as **12x6 inches** for readability.

2. Generate Bar Plot:

- X-axis: Use `**production.index.year**` to show years.
- Y-axis: Use `**production.values**` for production in tonnes.

3. Add Title and Labels:

- Title: Show location (district and state).
- X-axis Label: "**Year**."
- Y-axis Label: "**Production (Tonnes)**."

4. Format and Display:

- Rotate x-axis labels 45°.
- Use `plt.tight_layout()` to adjust spacing.
- Display plot with `plt.show()`.

This visual summarizes rice production over the years for the chosen district and state.

Step 4.2.3: ACF and PACF plot for ARIMA Model Selection includes the value of p,q and d.

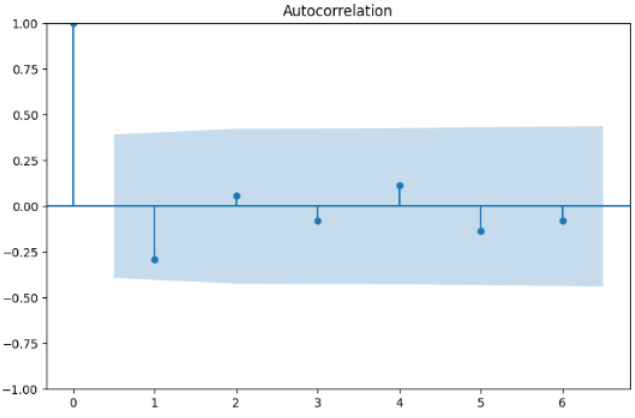
This step generates **Autocorrelation Function (ACF)** and **Partial Autocorrelation Function (PACF**) plots to aid in selecting the `p' (AR term) and `q' (MA term) parameters for an ARIMA model.

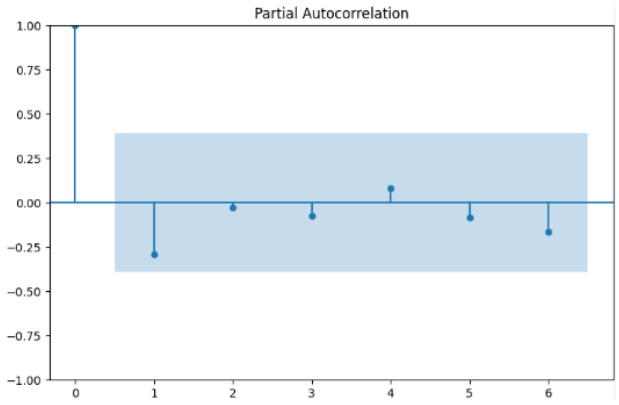
1. Set 'max_lags':

- 'max_lags' determines the number of lags for the ACF and PACF plots. It is set to roughly 30% of the data length, with a minimum of 1.

2. Generate ACF and PACF Plots:

- If `production_diff` (differenced production data) has more than 4 data points, the code attempts to plot the ACF and PACF.
- `plot_acf` and `plot_pacf` functions display these plots in a side-by-side layout:
- **ACF Plot**(left) shows correlation of `production_diff` with lagged versions of itself.
- **PACF Plot** (right) helps identify potential AR terms by removing the influence of intermediate lags.





3. Parameter Initialization:

- `p` and `q` values are initialized as 1 or the maximum number of lags (whichever is smaller) as a preliminary choice for the ARIMA model.

4. Display and Exception Handling:

- `plt.tight_layout()` ensures plots are properly spaced.
- The code handles errors gracefully, notifying if the plots can't be generated.

This step aids in the visual selection of `p`, `q`, and `d` parameters for a well-fitted ARIMA model.

Step 4.2.4: ARIMA model plot for the 'PRODUCTION' Parameter:

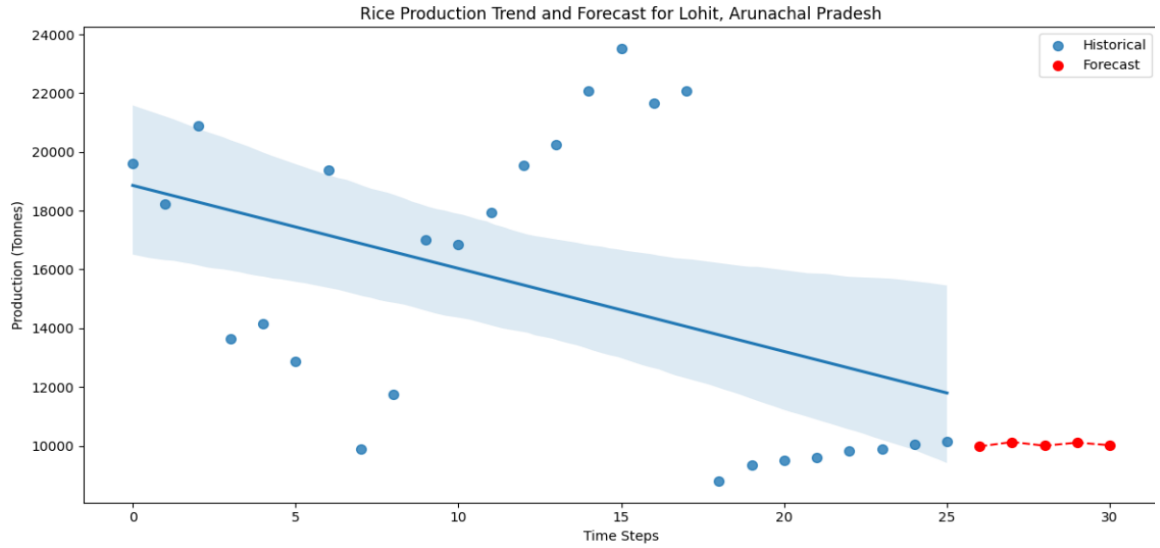
This segment fits an ARIMA model to production data, forecasts future values, and visualizes both historical and forecasted trends for easier analysis.

1. Model Fitting and Forecasting:

- An ARIMA model is created with `(p, d, q)` parameters based on prior analysis.
- The model is fitted to `production` data, and it forecasts production for the next 5 time points.

2. Plotting Historical and Forecasted Data:

- Sets the plot size for readability.
- A scatter plot with a trend line shows historical production data.
- Forecasted data points are added as red markers, with a red dashed line connecting them to illustrate the future trend.



This plot provides a clear view of historical production trends alongside future predictions, aiding in understanding potential changes in production.

Step 4.2.5: Splitting the data into training and testing set

1. Define Training and Testing Sets: If there are at least 5 data points, the data is split with 80% as training (`train`) and 20% as testing (`test`).

2. Train ARIMA Model: An ARIMA model is fitted to the training set (`train`) with the `(p, d, q)` parameters.

3. Generate Forecast for Testing Set: The model forecasts values for the length of the test set (`test`) to assess model accuracy on unseen data.

Step 4.2.6: Performance Metric Calculation

The model's performance is evaluated using two key metrics**: RMSE (Root Mean Squared Error)** to measure prediction accuracy, and **MAPE (Mean Absolute Percentage Error)** to understand the error in percentage terms.

```
Model Performance Metrics:
RMSE: 1384.47
MAPE: 13.19%
```

- **RMSE**:
  - Good scores are context-dependent; lower values are better.
  - Compare RMSE to the scale of the target variable.
- **MAPE**:
  - **0% to 10%**: Excellent
  - **10% to 20%**: Good
  - **20% to 50%**: Acceptable
  - **Above 50%**: Poor

Step 4.1.7: Bar and Line Plot for forecast:

1. Historical Production as Bar Plot:

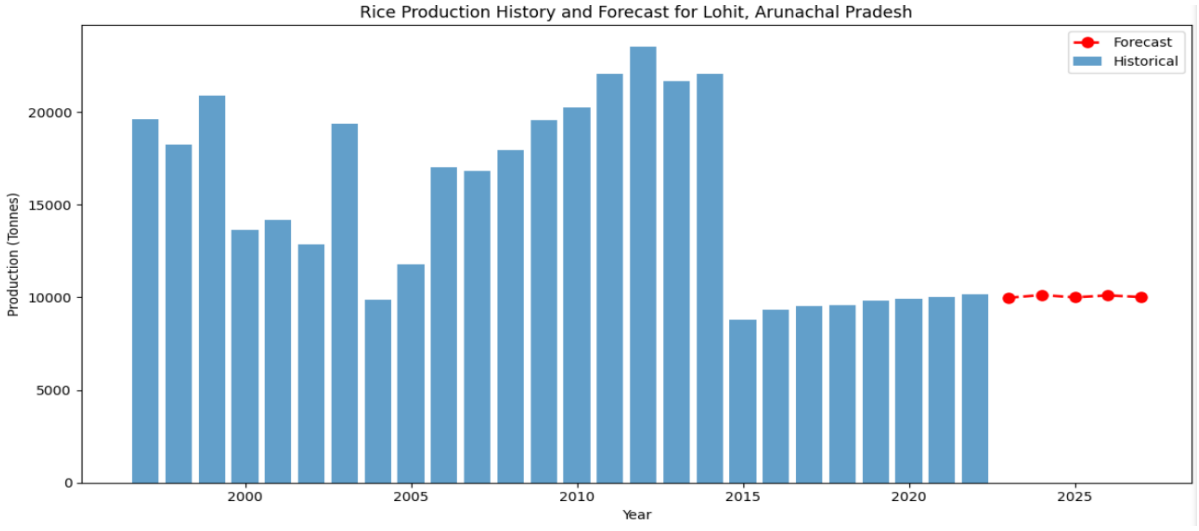
- Displays past production values year by year using bars.
- Bars are slightly transparent (`alpha=0.7`) to make the historical data distinct from the forecast.

2. Forecasted Data with Line and Markers:

- Forecasted values are represented with red markers connected by a dashed line.
- This highlights the projected production trend, making future changes more visible.

3. Detailed Plot Customization:

- Title includes district and state for easy reference.
- X-axis shows the years, while the y-axis represents production in tonnes.
- A legend helps distinguish historical data from forecasted values.



4. Displaying Forecasted Values:

- The forecasted production values for each of the next five years are printed after the plot.
- This provides a straightforward summary of future projections, aiding in analysis and planning.

```
Forecast for the next 5 years:
2023: 9,977.04 tonnes
2024: 10,127.04 tonnes
2025: 10,000.82 tonnes
2026: 10,107.03 tonnes
2027: 10,017.66 tonnes
```

Step 4.1.8: Creating the Search Interface and the Main Method():

1. Load Data and Preprocess:

- Specifies the file path (`rice.xlsx`) for loading the data.
- Calls `load_and_preprocess_data()` to prepare the data for analysis.

2. Collect User Input for Analysis:

- Prompts the user to enter a state and district for targeted analysis.
- Passes this information along with the preprocessed data to `analyze_agricultural_data()`.

3. Error Handling:

- Catches and displays any exceptions that occur during execution.
- Provides an error message, suggesting the user check inputs if something goes wrong.

This structure ensures data is loaded, user input is processed, and potential errors are handled smoothly.

Work Products and Deliverables:

We have created a Github Repository which contains the following:

1. Dataset as .xlsx file
2. Entire Code as .ipynb file in the form of a notebook

One can also refer to this video to access the code and see the output- **https://youtu.be/x7SuYveb_hc**. Here we have searched giving State- Arunachal Pradesh and District- Lohit. Try out with the one you need.

GitHub Repo Link: https://shorturl.at/5DPlu

Scope and Limitations:

1. The study focuses on rice production in India, utilizing data from 1997 to 2023.
2. The ARIMA model assumes that historical patterns and relationships within the data will persist in the future.
3. External factors, such as climate change, technological advancements, and policy changes, are not explicitly considered in the model but can influence future production levels.

Conclusion:

This analysis of historical rice production trends in India reveals significant fluctuations influenced by climate, agricultural practices, and economic factors. Understanding these trends is crucial for stakeholders focused on enhancing productivity and ensuring food security.

We developed a validated ARIMA-based forecasting model to predict rice yield and cultivated area, enabling accurate district-level production estimates for the coming years. These forecasts empower policymakers, farmers, and agricultural organizations to prepare for market fluctuations and implement targeted support for regions facing challenges.

The study also offers essential recommendations for improving India's rice production system. Policymakers should address regional issues such as water scarcity and soil degradation by promoting sustainable agricultural practices and investing in research and development. Encouraging collaboration among farmers, government agencies, and research institutions will foster innovation and knowledge sharing.

In summary, leveraging data-driven insights and strategic planning can significantly enhance India's rice production capabilities, ensuring food security and economic stability for millions who depend on this vital crop. By adopting these strategies, stakeholders can work together to build a more resilient and productive rice production system in India.

References:

- *Asteriou, Dimitros; Hall, Stephen G. (2011). "ARIMA Models and the Box–Jenkins Methodology". Applied Econometrics (Second ed.). Palgrave MacMillan. pp. 265–286. ISBN 978-0-230-27182-1.*
- *Mills, Terence C. (1990). Time Series Techniques for Economists. Cambridge University Press. ISBN 978-0-521-34339-8.*
- *Percival, Donald B.; Walden, Andrew T. (1993). Spectral Analysis for Physical Applications. Cambridge University Press. ISBN 978-0-521-35532-2.*
- Shumway R.H. and Stoffer, D.S. (2017). *Time Series Analysis and Its Applications: With R Examples*. Springer. DOI: 10.1007/978-3-319-52452-8
- ARIMA Models in Python. Become an expert in fitting ARIMA (autoregressive integrated moving average) models to time series data using Python.

.