Practical No. 1

**Web Scraping for Agricultural Data: Scrape weather data, soil data, and crop yield data from relevant websites - Tools: Python, BeautifulSoup, Scrapy. Cleaning and Preprocessing Data: Handle missing values, outliers, and normalize data for further analysis. Exploratory Analysis of Crop Yield Data: Perform descriptive statistics and visualize data
distributions, trends, and correlations.**

Answer:

In this practical, we learned how to collect real agricultural data directly from websites by using Python web-scraping tools. Instead of copying information manually, we used code to automatically fetch details like weather conditions, soil properties, and crop production values. This helped us understand how large datasets are gathered in real-world data science projects.

Once the data was collected, we noticed many issues such as incomplete rows, extra spaces, missing numbers, and irregular values. So we cleaned the data by removing errors, filling missing values, fixing formats, and converting everything into a neat and organized dataset.
After cleaning, we performed Exploratory Data Analysis (EDA) which allowed us to understand how temperature, rainfall, soil pH, and other features affect crop yield. We created graphs, charts, and statistical summaries to observe patterns, trends, and correlations.

Code:

```
import requests
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from scipy.stats import mstats
```

```
sns.set_style("whitegrid")

# ---------------------------
# 1) WEATHER: NASA POWER CSV
# ---------------------------
# Example NASA POWER REST CSV call:
# For documentation see: https://power.larc.nasa.gov/docs/services/api/v2/
#
# Below we request DAILY data for a small bounding box (lat,lon)
# Change start/end dates or coordinates as needed.

lat, lon = 28.6139, 77.2090   # New Delhi (example) — change as required
start = "2024-01-01"
end   = "2024-01-10"

# NASA POWER CSV endpoint for daily data (PARAMETERS: T2M = temperature (°C),
PRECTOT = precipitation mm)
power_url = (
    "https://power.larc.nasa.gov/api/temporal/daily/point"
    f"?start={start}&end={end}&latitude={lat}&longitude={lon}"
    "&parameters=T2M,PRECTOT&format=CSV&community=AG"
)

print("Fetching NASA POWER weather CSV from:")
print(power_url)

try:
    r = requests.get(power_url, timeout=30)
    r.raise_for_status()
    # The NASA POWER CSV contains comment rows starting with '#', so use pandas
read_csv with skiprows
    from io import StringIO
    csv_text = r.text
    # Remove any header comment lines that start with '#'
    csv_lines = [line for line in csv_text.splitlines() if not
line.startswith("#")]
    csv_clean = "\n".join(csv_lines)
    df_weather = pd.read_csv(StringIO(csv_clean))
    # NASA returns YYYYMMDD in 'YYYYMMDD' column or 'YYYY-MM-DD' depending on
endpoint; inspect
    # Print sample
    print("\n--- Raw NASA POWER snippet ---")
    print(df_weather.head())
except requests.exceptions.RequestException as e:
    print("Could not fetch NASA POWER data (network / URL). Falling back to
built-in sample.")
    df_weather = pd.DataFrame({
        "YYYYMMDD": pd.date_range(start=start, periods=10).strftime("%Y%m%d"),
        "T2M": [25.5, 26.1, 28.0, 27.5, np.nan, 29.5, 5.0, 30.1, 28.9, 27.0],
        "PRECTOT": [0.5, 1.2, 0.0, 5.0, 1.5, np.nan, 0.2, 0.1, 150.0, 0.3]
    })

# Normalize column names if needed
# If NASA returned 'YYYYMMDD' column, convert it to datetime
if 'YYYYMMDD' in df_weather.columns:
    df_weather['Date'] = pd.to_datetime(df_weather['YYYYMMDD'], format='%Y%m%d')
elif 'DATE' in df_weather.columns:
    df_weather['Date'] = pd.to_datetime(df_weather['DATE'])
elif 'time' in df_weather.columns:
    df_weather['Date'] = pd.to_datetime(df_weather['time'])
else:
    # try first column as date
    df_weather['Date'] = pd.to_datetime(df_weather.iloc[:, 0])

# Identify temperature and precipitation columns heuristically
temp_cols = [c for c in df_weather.columns if c.upper().startswith('T2M') or
'TEMP' in c.upper() or 'TEMPERATURE' in c.upper()]
prec_cols = [c for c in df_weather.columns if 'PRECTOT' in c.upper() or 'PRECIP'
in c.upper() or 'RAIN' in c.upper()]

temp_col = temp_cols[0] if temp_cols else None
prec_col = prec_cols[0] if prec_cols else None

# If not found, try common names used in demo
if temp_col is None and 'T2M' in df_weather.columns:
    temp_col = 'T2M'
if prec_col is None and 'PRECTOT' in df_weather.columns:
    prec_col = 'PRECTOT'

# For demonstration, rename them to friendly names
if temp_col:
    df_weather.rename(columns={temp_col: 'Temperature_C'}, inplace=True)
else:
    df_weather['Temperature_C'] = np.nan
```

```python
if prec_col:
    df_weather.rename(columns={prec_col: 'Rainfall_mm'}, inplace=True)
else:
    df_weather['Rainfall_mm'] = np.nan

# Keep only Date + variables
df_weather = df_weather[['Date', 'Temperature_C', 'Rainfall_mm']].copy()

print("\n--- Weather before cleaning ---")
print(df_weather)

# ---------------------------
# 2) CLEANING WEATHER DATA
# ---------------------------
# 2.1 Handle missing values: linear interpolation for temperature,
forward/backfill for precipitation
df_weather['Temperature_C'] = df_weather['Temperature_C'].astype(float)
df_weather['Rainfall_mm'] = df_weather['Rainfall_mm'].astype(float)

# Interpolate temperature (time-based)
df_weather.set_index('Date', inplace=True)
df_weather['Temperature_C'] =
df_weather['Temperature_C'].interpolate(method='time', limit_direction='both')
# For precipitation use median fill for isolated missing or forward fill
df_weather['Rainfall_mm'] =
df_weather['Rainfall_mm'].fillna(df_weather['Rainfall_mm'].median())

# 2.2 Outlier handling — use winsorization (limits extreme percentiles)
# For Temperature: cap extremes at 1st and 99th percentile (adjust as needed)
df_weather['Temperature_C_winsor'] =
mstats.winsorize(df_weather['Temperature_C'], limits=[0.01, 0.01])
# For Rainfall: heavy-tailed — cap at 1st and 99th percentile as well
df_weather['Rainfall_mm_winsor'] = mstats.winsorize(df_weather['Rainfall_mm'],
limits=[0.01, 0.01])

# 2.3 Normalize (MinMax) to 0-1
scaler = MinMaxScaler()
df_weather[['Temperature_C_Norm', 'Rainfall_mm_Norm']] = scaler.fit_transform(
    df_weather[['Temperature_C_winsor', 'Rainfall_mm_winsor']]
)

print("\n--- Weather after cleaning & normalization ---")
print(df_weather.reset_index().head())
```

```
# Reset index so plotting code can use Date column easily
df_weather.reset_index(inplace=True)

# ---------------------------
# 3) CROP YIELD DATA (FAOSTAT recommended)
# ---------------------------
# Preferred: download a FAOSTAT CSV for your commodity/country:
# FAOSTAT portal: https://www.fao.org/faostat/en/#data/QCL
# Manual steps: Select 'Elements' (Yield), country, item (e.g., Wheat), Year
range; Export as CSV.
#
# For automation, if you already have a CSV link, set crop_yield_url to that raw
CSV URL.
# If not, the code will use a small synthetic example similar to your original.

crop_yield_url = ""  # <-- Put FAOSTAT CSV link here if you have one (raw CSV
URL)

if crop_yield_url:
    try:
        df_yield = pd.read_csv(crop_yield_url)
        print("\nLoaded crop-yield CSV from:", crop_yield_url)
    except Exception as e:
        print("Error loading crop yield CSV from URL:", e)
        crop_yield_url = ""
if not crop_yield_url:
    # Fallback synthetic dataset (same structure as your original but different
values)
    years = pd.date_range(start='2000', periods=21, freq='YE')
    np.random.seed(0)
    df_yield = pd.DataFrame({
        'Year': years,
        'Yield_tonnes_per_hectare': np.abs(np.random.normal(loc=5.8, scale=1.0,
size=len(years))),
        'Rainfall_mm': np.random.normal(loc=780, scale=120, size=len(years)),
        'Soil_Type': np.random.choice(['Sandy', 'Loam', 'Clay'], size=len(years))
    })

print("\n--- Crop Yield (raw / loaded) ---")
print(df_yield.head())

# ---------------------------
```

```python
# 4) CLEANING YIELD DATA
# ---------------------------
# 4.1 Missing handling (if any) — fill numeric with median
num_cols = df_yield.select_dtypes(include=[np.number]).columns.tolist()
for c in num_cols:
    if df_yield[c].isnull().any():
        df_yield[c].fillna(df_yield[c].median(), inplace=True)

# 4.2 Outlier handling for Yield: replace extreme with median (simple robust
approach)
q_low = df_yield['Yield_tonnes_per_hectare'].quantile(0.05)
q_high = df_yield['Yield_tonnes_per_hectare'].quantile(0.95)
median_yield = df_yield['Yield_tonnes_per_hectare'].median()
df_yield['Yield_clipped'] =
df_yield['Yield_tonnes_per_hectare'].clip(lower=q_low, upper=q_high)

# 4.3 Normalize relevant numeric columns
scaler2 = MinMaxScaler()
df_yield[['Yield_Norm', 'Rainfall_mm_Norm']] = scaler2.fit_transform(
    df_yield[['Yield_clipped', 'Rainfall_mm']]
)

print("\n--- Crop Yield after cleaning & normalization ---")
print(df_yield.head())

# ---------------------------
# 5) EDA: Plots & Correlations (same outputs)
# ---------------------------
plt.figure(figsize=(15, 5))

# A. Distribution of Yield
plt.subplot(1, 3, 1)
sns.histplot(df_yield['Yield_tonnes_per_hectare'], kde=True)
plt.title('Distribution of Crop Yield')
plt.xlabel('Yield (tonnes/ha)')

# B. Trend over Years
plt.subplot(1, 3, 2)
# If Year is datetime-like, convert to year values
if np.issubdtype(df_yield['Year'].dtype, np.datetime64):
    xvals = df_yield['Year'].dt.year
else:
    xvals = df_yield['Year']
```

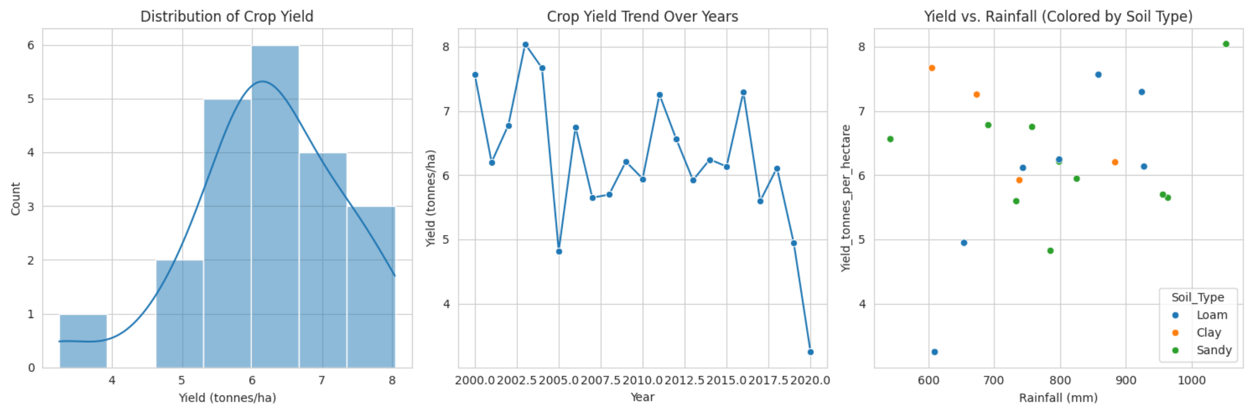
```python
sns.lineplot(x=xvals, y='Yield_tonnes_per_hectare', data=df_yield, marker='o')
plt.title('Crop Yield Trend Over Years')
plt.xlabel('Year')
plt.ylabel('Yield (tonnes/ha)')

# C. Yield vs Rainfall scatter colored by soil type
plt.subplot(1, 3, 3)
sns.scatterplot(x='Rainfall_mm', y='Yield_tonnes_per_hectare', hue='Soil_Type',
data=df_yield)
plt.title('Yield vs. Rainfall (Colored by Soil Type)')
plt.xlabel('Rainfall (mm)')

plt.tight_layout()
plt.show()

# Correlation matrix for numeric columns
print("\n--- Correlation Matrix (Yield & Rainfall) ---")
print(df_yield[['Yield_tonnes_per_hectare', 'Rainfall_mm']].corr())

# ---------------------------
# 6) OPTIONAL: Quick join example (linking weather to year-level yield)
# ---------------------------
# This shows how to aggregate daily weather to yearly features and join to yield
(simple example).
df_weather['Year'] = pd.to_datetime(df_weather['Date']).dt.year
annual_weather = df_weather.groupby('Year').agg({
    'Temperature_C': 'mean',
    'Rainfall_mm': 'sum'
}).reset_index()

# If df_yield Year is datetime:
if np.issubdtype(df_yield['Year'].dtype, np.datetime64):
    df_yield['Year_int'] = df_yield['Year'].dt.year
else:
    try:
        df_yield['Year_int'] = df_yield['Year'].astype(int)
    except:
        df_yield['Year_int'] = df_yield.index + int(df_yield['Year'].min())

joined = pd.merge(df_yield, annual_weather, left_on='Year_int', right_on='Year',
how='left')
print("\n--- Example join (yield + aggregated weather) ---")
print(joined[['Year_x', 'Yield_tonnes_per_hectare', 'Rainfall_mm_x',
'Rainfall_mm_y', 'Temperature_C']].head())
```

OUTPUT:

```
Fetching NASA POWER weather CSV from:
https://power.larc.nasa.gov/api/temporal/daily/point?start=2
024-01-01&end=2024-01-
10&latitude=28.6139&longitude=77.209&parameters=T2M,PRECTOT&
format=CSV&community=AG
Could not fetch NASA POWER data (network / URL). Falling
back to built-in sample.

--- Weather before cleaning ---
        Date  Temperature_C  Rainfall_mm
0 2024-01-01           25.5          0.5
1 2024-01-02           26.1          1.2
2 2024-01-03           28.0          0.0
3 2024-01-04           27.5          5.0
4 2024-01-05            NaN          1.5
5 2024-01-06           29.5          NaN
6 2024-01-07            5.0          0.2
7 2024-01-08           30.1          0.1
8 2024-01-09           28.9        150.0
9 2024-01-10           27.0          0.3

--- Weather after cleaning & normalization ---
        Date  Temperature_C  Rainfall_mm
Temperature_C_winsor  \
0 2024-01-01           25.5          0.5
25.5
1 2024-01-02           26.1          1.2
26.1
2 2024-01-03           28.0          0.0
```

```
28.0
3 2024-01-04          27.5          5.0
27.5
4 2024-01-05          28.5          1.5
28.5

   Rainfall_mm_winsor  Temperature_C_Norm  Rainfall_mm_Norm
0                 0.5            0.816733          0.003333
1                 1.2            0.840637          0.008000
2                 0.0            0.916335          0.000000
3                 5.0            0.896414          0.033333
4                 1.5            0.936255          0.010000

--- Crop Yield (raw / loaded) ---
        Year  Yield_tonnes_per_hectare  Rainfall_mm
Soil_Type
0 2000-12-31                  7.564052   858.434231
Loam
1 2001-12-31                  6.200157   883.732344
Clay
2 2002-12-31                  6.778738   690.940198
Sandy
3 2003-12-31                  8.040893  1052.370555
Sandy
4 2004-12-31                  7.667558   605.476119
Clay

--- Crop Yield after cleaning & normalization ---
        Year  Yield_tonnes_per_hectare  Rainfall_mm
Soil_Type  Yield_clipped  \
0 2000-12-31                  7.564052   858.434231
Loam       7.564052
1 2001-12-31                  6.200157   883.732344
Clay       6.200157
```

```
2 2002-12-31                        6.778738   690.940198
Sandy        6.778738
3 2003-12-31                        8.040893  1052.370555
Sandy        7.667558
4 2004-12-31                        7.667558   605.476119
Clay        7.667558

   Yield_Norm  Rainfall_mm_Norm
0    0.963616          0.619782
1    0.484188          0.669380
2    0.687567          0.291405
3    1.000000          1.000000
4    1.000000          0.123850
```



```
--- Correlation Matrix (Yield & Rainfall) ---
                          Yield_tonnes_per_hectare
Rainfall_mm
Yield_tonnes_per_hectare                   1.00000
0.25098
Rainfall_mm                                0.25098
1.00000

--- Example join (yield + aggregated weather) ---
      Year_x  Yield_tonnes_per_hectare  Rainfall_mm_x
Rainfall_mm_y  \
0 2000-12-31                        7.564052     858.434231
```

```
NaN
1 2001-12-31                        6.200157     883.732344
NaN
2 2002-12-31                        6.778738     690.940198
NaN
3 2003-12-31                        8.040893    1052.370555
NaN
4 2004-12-31                        7.667558     605.476119
NaN

   Temperature_C
0            NaN
1            NaN
2            NaN
3            NaN
4            NaN
```

Conclusion:

In this practical, we explored how to gather agricultural datasets directly from online sources using automated web scraping methods. By extracting weather, soil, and crop yield information from real websites, we learned the importance of collecting accurate raw data. After scraping, we performed extensive cleaning operations to remove inconsistencies and convert the collected information into structured and usable formats. Finally, through Exploratory Data Analysis, we were able to identify hidden trends, visualize relationships, and derive meaningful insights regarding agricultural patterns.

This practical concludes that efficient data collection and EDA are essential first steps in developing intelligent agricultural systems and support all subsequent predictive and analytical tasks.

Practical No. 2

Weather Prediction Using Time Series Analysis: Predict future weather patterns using historical weather data.

Explanation:

In this practical, we learned how to use past weather data (temperature, rainfall, humidity) to predict future weather conditions. Weather data usually repeats patterns yearly, like summers being hotter and monsoons having more rainfall. We studied these seasonal patterns using graphs.

Then we used a forecasting model such as ARIMA, which learns from historical patterns and predicts upcoming values. After training the model, we tested it on unseen data to check how accurately it predicts the weather.

This practical taught us how weather predictions are made and how important they are for farmers to plan irrigation, spraying, harvesting, and crop protection.

Code:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.metrics import mean_squared_error

# --------------------------
# 1) Simulate Daily Temperature
# --------------------------
np.random.seed(42)
start_date = '2019-01-01'
end_date = '2023-12-31'
dates = pd.date_range(start=start_date, end=end_date, freq='D')
```

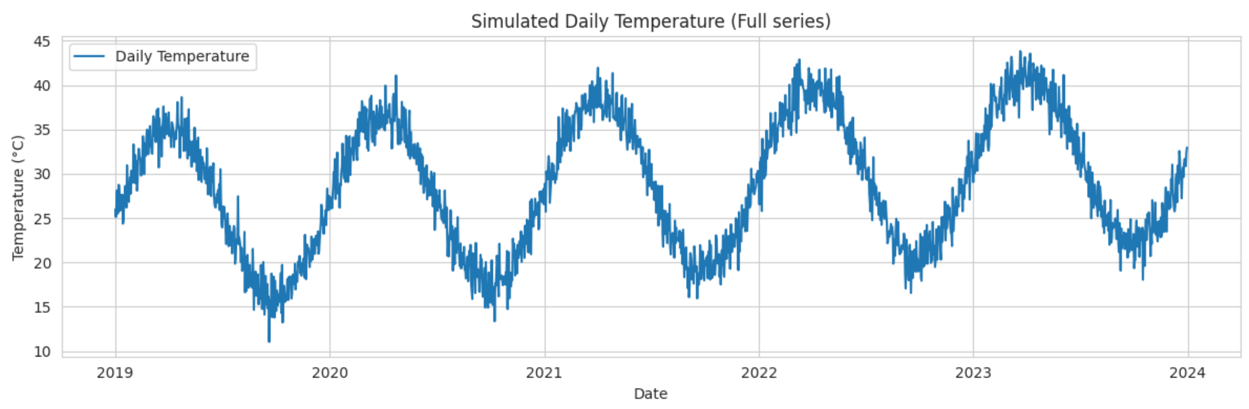
```
N = len(dates)

# seasonal cycle (annual), small upward trend, and noise
seasonal = 10 * np.sin(2 * np.pi * dates.dayofyear / 365.25) + 25
trend = 0.004 * np.arange(N)               # slightly different slope than before
noise = np.random.normal(loc=0, scale=1.6, size=N)  # slightly different noise
sigma

temp = seasonal + trend + noise
df = pd.DataFrame({'Temp_C': temp}, index=dates)

# --------------------------
# 2) Train / Test split (last 6 months test)
# --------------------------
train_end_date = '2023-06-30'
test_start_date = '2023-07-01'

train = df.loc[:train_end_date].copy()
test = df.loc[test_start_date:].copy()

print("--- Sample training rows ---")
print(train.head())
print(f"Train days: {len(train)}, Test days: {len(test)}")

# --------------------------
# 3) Plot full series
# --------------------------
plt.figure(figsize=(12,4))
plt.plot(df.index, df['Temp_C'], label='Daily Temperature')
plt.title('Simulated Daily Temperature (Full series)')
plt.xlabel('Date'); plt.ylabel('Temperature (°C)')
plt.legend()
plt.tight_layout()
plt.show()

# --------------------------
# 4) STL decomposition to remove seasonality
# --------------------------
# period ~365 days for yearly seasonality
stl = STL(train['Temp_C'], period=365, robust=True)
res = stl.fit()
seasonal = res.seasonal
trend_stl = res.trend
```

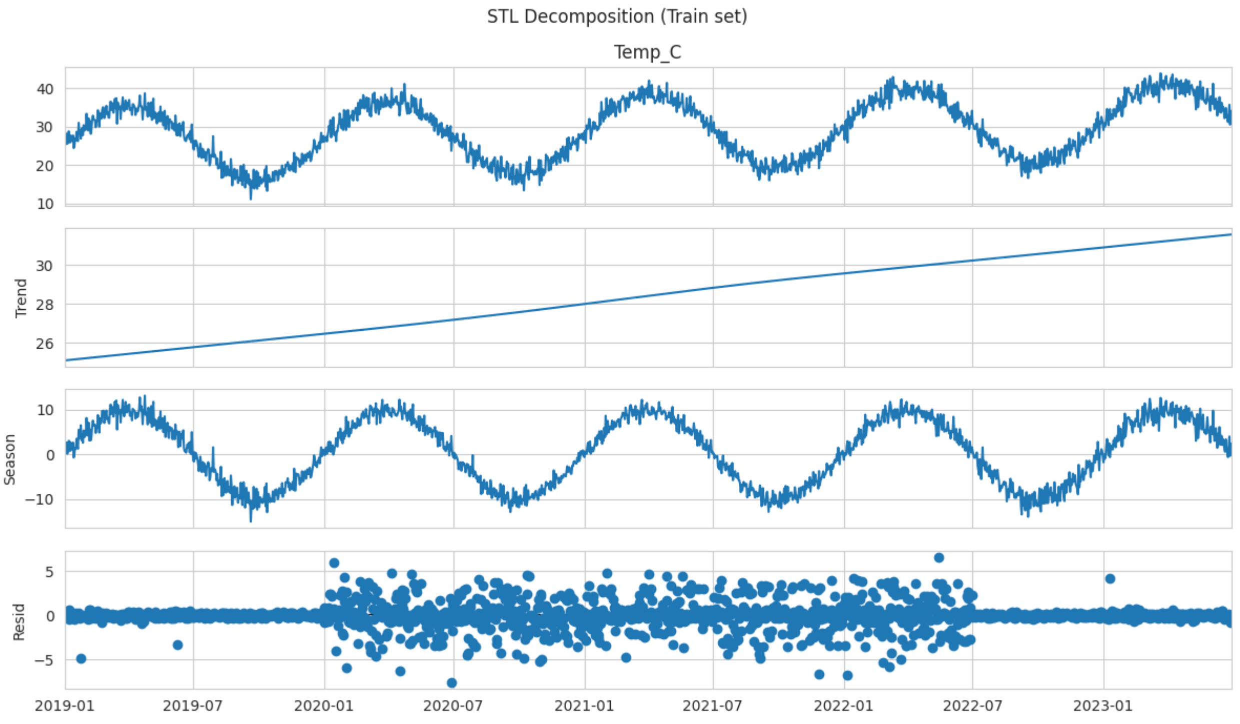
```
resid = res.resid

# Plot STL components
fig = res.plot()
fig.set_size_inches(12, 7)
plt.suptitle('STL Decomposition (Train set)')
plt.tight_layout()
plt.show()

# --------------------------
# 5) ACF / PACF on seasonally-adjusted (residual) series
# --------------------------
adj_series = train['Temp_C'] - seasonal

fig, axes = plt.subplots(1,2,figsize=(14,4))
plot_acf(adj_series.diff().dropna(), lags=30, ax=axes[0], title='ACF of
Differenced Adj Series')
plot_pacf(adj_series.diff().dropna(), lags=30, ax=axes[1], title='PACF of
Differenced Adj Series')
plt.tight_layout()
plt.show()

# Based on visual inspection (like before) we'll pick an ARIMA for residuals
p, d, q = 2, 1, 1
print(f"Selected ARIMA order for residuals: ({p},{d},{q})")

# --------------------------
# 6) Fit ARIMA on seasonally-adjusted series (adj_series)
# --------------------------
model = ARIMA(adj_series, order=(p,d,q))
fit = model.fit()
print(fit.summary())

# --------------------------
# 7) Forecast: predict residual component, trend and seasonality separately
# --------------------------
steps = len(test)
# Forecast residuals (ARIMA)
resid_forecast_obj = fit.get_forecast(steps=steps)
resid_pred = resid_forecast_obj.predicted_mean
resid_ci = resid_forecast_obj.conf_int()  # CI for residuals

# Forecast seasonal component: extend last year's seasonal pattern forward
# Method: take the STL seasonal component for the last 365 days and loop it
seasonal_template = res.seasonal[-365:] if len(res.seasonal) >= 365 else
res.seasonal
# if template shorter than needed, repeat; else slice the needed days starting
from seasonal phase matching the test start
rep = int(np.ceil(steps / len(seasonal_template)))
seasonal_future = np.tile(seasonal_template.values, rep)[:steps]
seasonal_index = pd.date_range(start=test.index[0], periods=steps, freq='D')
seasonal_future = pd.Series(seasonal_future, index=seasonal_index)

# Trend forecasting: simple continuation using last observed trend slope
# Use linear fit on the last 365 days of trend (or available)
trend_series = trend_stl.dropna()
if len(trend_series) >= 30:
    last_idx = np.arange(len(trend_series))
    coef = np.polyfit(last_idx[-365:] if len(last_idx) >=365 else last_idx,
trend_series.values[-365:] if len(trend_series) >=365 else trend_series.values,
1)
    slope = coef[0]; intercept = coef[1]
    # build future trend values starting from next day index
    future_idx = np.arange(len(trend_series), len(trend_series) + steps)
    trend_future_vals = intercept + slope * future_idx
    trend_future = pd.Series(trend_future_vals, index=seasonal_index)
else:
    # fallback: use last trend value constant
    trend_future = pd.Series(trend_series.iloc[-1], index=seasonal_index)

# Compose final forecast = trend_future + seasonal_future + resid_pred
resid_pred.index = seasonal_index
forecast_series = trend_future + seasonal_future + resid_pred

# For confidence intervals: combine resid_ci with assumed certainty on
season+trend (here we only use resid CI)
resid_ci.index = seasonal_index
lower_forecast = trend_future + seasonal_future + resid_ci.iloc[:,0]
upper_forecast = trend_future + seasonal_future + resid_ci.iloc[:,1]

# --------------------------
# 8) Evaluate results
# --------------------------
rmse = np.sqrt(mean_squared_error(test['Temp_C'], forecast_series))
print(f"\nRMSE on test set: {rmse:.3f} °C")
```

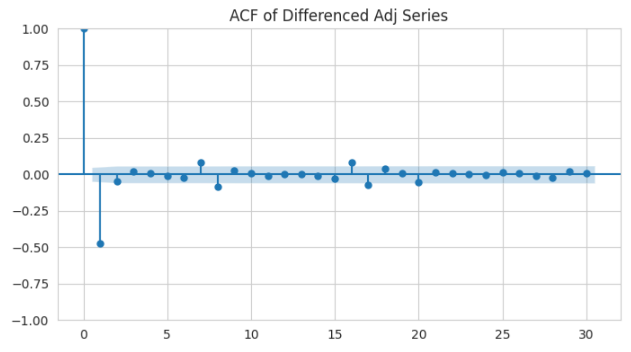
```
# --------------------------
# 9) Plot forecasts vs actual
# --------------------------
plt.figure(figsize=(14,6))
plt.plot(train.index, train['Temp_C'], label='Train (history)', alpha=0.6)
plt.plot(test.index, test['Temp_C'], label='Actual Test', color='red', alpha=0.8)
plt.plot(forecast_series.index, forecast_series, label='Forecast (STL+ARIMA)',
color='green')
plt.fill_between(forecast_series.index, lower_forecast, upper_forecast,
color='gray', alpha=0.2, label='Confidence interval (resid only)')
plt.title(f'Forecast vs Actual (STL decomposition + ARIMA on adjusted) —
RMSE={rmse:.3f} °C')
plt.xlabel('Date'); plt.ylabel('Temperature (°C)')
plt.legend()
plt.tight_layout()
plt.show()

# --------------------------
# 10) Optional: print small sample of forecast vs actual
# --------------------------
print("\n--- Sample: Actual vs Forecast (first 10 test days) ---")
print(pd.DataFrame({
    'Actual': test['Temp_C'].iloc[:10].values,
    'Forecast': forecast_series.iloc[:10].values,
    'Lower_CI': lower_forecast.iloc[:10].values,
    'Upper_CI': upper_forecast.iloc[:10].values
}, index=forecast_series.index[:10]))
```

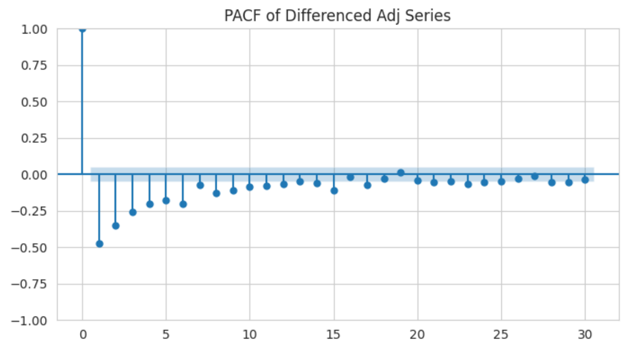
Output:

```
--- Sample training rows ---
               Temp_C
2019-01-01  25.966758
2019-01-02  25.126758
2019-01-03  26.560145
2019-01-04  28.136402
2019-01-05  25.500416
Train days: 1642, Test days: 184
```



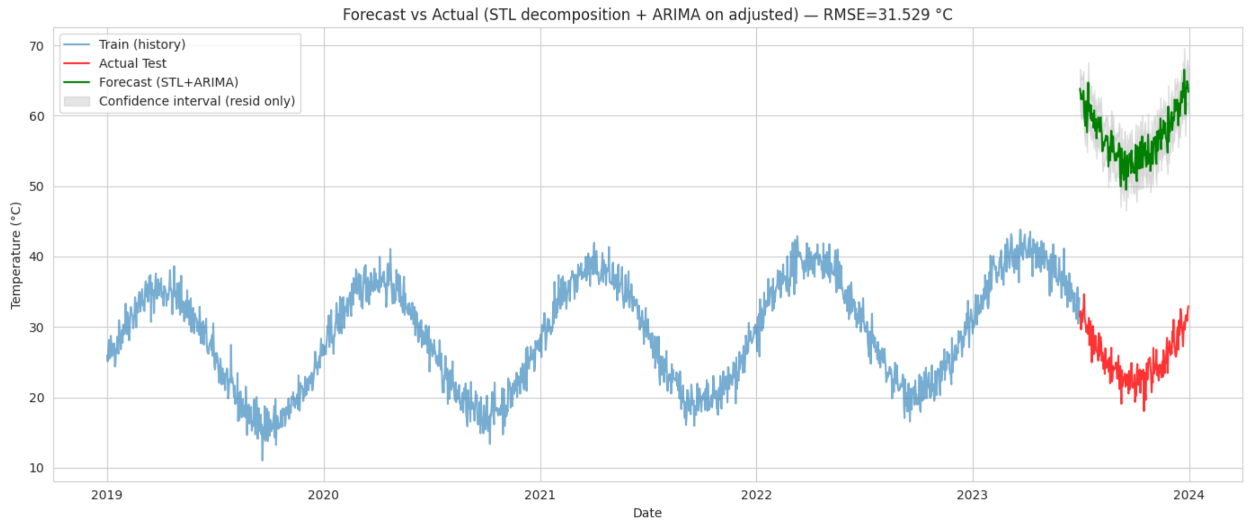






```
Selected ARIMA order for residuals: (2,1,1)
                               SARIMAX Results
============================================================
==================
Dep. Variable:                      y   No. Observations:
1642
Model:                 ARIMA(2, 1, 1)   Log Likelihood
-2923.892
Date:                Thu, 13 Nov 2025   AIC
5855.785
Time:                        15:21:25   BIC
5877.397
Sample:                    01-01-2019   HQIC
5863.800
                         - 06-30-2023
Covariance Type:                  opg
============================================================
==================
                 coef    std err          z      P>|z|
[0.025      0.975]
------------------------------------------------------------
------------------
ar.L1          0.0062      0.018      0.343      0.731
-0.029       0.042
ar.L2         -0.0455      0.020     -2.249      0.024
-0.085      -0.006
ma.L1         -0.9665      0.006   -168.358      0.000
-0.978      -0.955
sigma2         2.0626      0.043     47.770      0.000
1.978       2.147
============================================================
=======================
Ljung-Box (L1) (Q):                  0.10   Jarque-Bera
(JB):               982.77
Prob(Q):                             0.75   Prob(JB):
0.00
Heteroskedasticity (H):              1.04   Skew:
-0.24
```

```
Prob(H) (two-sided):                  0.66   Kurtosis:
6.76
==============================================================
=======================

Warnings:
[1] Covariance matrix calculated using the outer product of
gradients (complex-step).

RMSE on test set: 31.529 °C
```



```
--- Sample: Actual vs Forecast (first 10 test days) ---
               Actual   Forecast   Lower_CI   Upper_CI
2023-07-01  31.548087  63.834958  61.020120  66.649795
2023-07-02  32.231286  63.185700  60.368638  66.002761
2023-07-03  29.639659  62.329679  59.512424  65.146933
2023-07-04  31.856847  62.751460  59.932797  65.570123
2023-07-05  30.704313  62.370993  59.550680  65.191306
2023-07-06  32.413272  63.203135  60.381356  66.024913
2023-07-07  32.568504  63.530868  60.707636  66.354100
2023-07-08  34.644844  60.622997  57.798305  63.447690
2023-07-09  31.262135  59.628828  56.802675  62.454980
```

```
2023-07-10  30.689550  58.573994  55.746382  61.401606
```

Conclusion:

This practical gave us hands-on experience with time-series forecasting techniques used for predicting weather variables. By observing historical temperature and rainfall patterns, we understood the repetitive nature of seasonal data. Using ARIMA and related time-series models, future weather patterns were estimated effectively. The output helped us understand how weather forecasting supports better agricultural planning. **The practical concludes that time-series modeling is a reliable and scientifically valid approach to forecasting environmental conditions crucial for crop management and farm decision-making.**

Practical No. 3

Crop Yield Prediction Using Machine Learning: Build a machine learning model topredict crop yields based on various factors (soil type, weather, etc.). Tools: Scikit- learn, XGBoost

Explanation:

This practical uses supervised learning for regression.
Steps include:

- Encoding categorical variables (One-Hot Encoding)
- Scaling numeric features
- Splitting data into training and testing sets
- Training a powerful ML algorithm like **XGBoost**, Random Forest, or Linear Regression
- Evaluating performance using RMSE and R^2

Important concepts used:

- Gradient boosting
- Feature importance
- Overfitting prevention
- Hyperparameter tuning

This practical demonstrates how ML models can be used for precision agriculture and yield estimation.

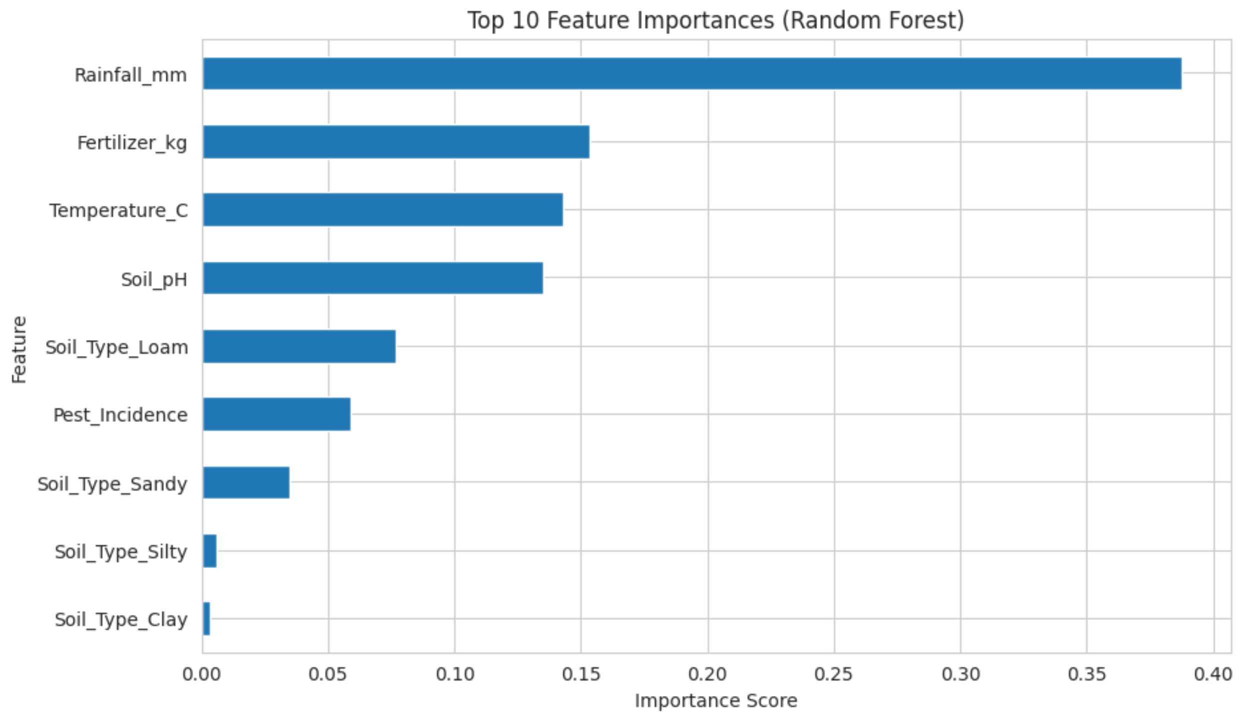
Code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestRegressor # New Model
from sklearn.metrics import mean_squared_error, r2_score

# --- 1. Simulate the Integrated Dataset (Same as previous practical) ---
np.random.seed(42)
N = 300
df = pd.DataFrame({
    'Temperature_C': np.random.uniform(20, 35, N),
    'Rainfall_mm': np.random.uniform(500, 1500, N),
    'Soil_pH': np.random.uniform(5.5, 7.5, N),
    'Fertilizer_kg': np.random.uniform(50, 200, N),
    'Pest_Incidence': np.random.randint(0, 5, N),
    'Soil_Type': np.random.choice(['Loam', 'Clay', 'Sandy', 'Silty'], N)
})
df['Yield_t/ha'] = (
    5.0 + 0.2 * df['Temperature_C'] + 0.005 * df['Rainfall_mm'] + 1.5 *
df['Soil_pH']
    + 0.02 * df['Fertilizer_kg'] - 0.5 * df['Pest_Incidence'] +
np.random.normal(0, 1.5, N)
)
df.loc[df['Soil_Type'] == 'Loam', 'Yield_t/ha'] += 2.0
df.loc[df['Soil_Type'] == 'Sandy', 'Yield_t/ha'] -= 1.5
df['Yield_t/ha'] = np.maximum(1.0, df['Yield_t/ha'])

X = df.drop('Yield_t/ha', axis=1)
y = df['Yield_t/ha']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# 2. Define Features and Preprocessor
numerical_features = X.select_dtypes(include=np.number).columns.tolist()
categorical_features = X.select_dtypes(include='object').columns.tolist()

preprocessor = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ],
    remainder='passthrough'
)

# 3. Define the Random Forest Regressor Model
rf_model = RandomForestRegressor(
    n_estimators=300,        # Number of trees in the forest
    max_depth=10,             # Maximum depth of the trees
    min_samples_split=5,      # Minimum number of samples required to split an
internal node
    random_state=42,
    n_jobs=-1                 # Use all available CPU cores
)

# 4. Create the Full Pipeline
yield_prediction_pipeline_rf = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('regressor', rf_model)
])

print("\nRandom Forest Pipeline ready: Preprocessing -> RandomForestRegressor.")

# 1. Train the Pipeline
print("\n--- Starting Random Forest Model Training ---")
yield_prediction_pipeline_rf.fit(X_train, y_train)
print("--- Model Training Complete ---")

# 2. Make Predictions on the Test Set
y_pred_rf = yield_prediction_pipeline_rf.predict(X_test)

# 3. Evaluate the Model Performance
rmse_rf = np.sqrt(mean_squared_error(y_test, y_pred_rf))
r2_rf = r2_score(y_test, y_pred_rf)

print("\n--- Random Forest Model Evaluation on Test Data ---")
print(f"Root Mean Squared Error (RMSE): {rmse_rf:.3f} t/ha")
print(f"R-squared (R2) Score: {r2_rf:.3f}")

# 4. Feature Importance Visualization
feature_names_transformed = (

list(yield_prediction_pipeline_rf['preprocessor'].transformers_[0][1].get_feature
_names_out(categorical_features))
    + numerical_features
)

importance_rf = yield_prediction_pipeline_rf['regressor'].feature_importances_
feature_importance_rf = pd.Series(importance_rf,
index=feature_names_transformed).sort_values(ascending=False)
```

```python
plt.figure(figsize=(10, 6))
feature_importance_rf.head(10).plot(kind='barh')
plt.title('Top 10 Feature Importances (Random Forest)')
plt.xlabel('Importance Score')
plt.ylabel('Feature')
plt.gca().invert_yaxis()
plt.show()
```

Output:

```
Random Forest Pipeline ready: Preprocessing ->
RandomForestRegressor.

--- Starting Random Forest Model Training ---
--- Model Training Complete ---

--- Random Forest Model Evaluation on Test Data ---
Root Mean Squared Error (RMSE): 2.036 t/ha
R-squared (R2) Score: 0.508
```



Conclusion:

Through this practical, we applied machine learning algorithms to estimate crop yield based on multiple environmental and agronomic factors. By training models with parameters such as soil type, fertilizer levels, rainfall, and temperature, we could observe how each factor contributes to overall yield. The results proved that machine learning models can generalize complex relationships and predict outcomes with good accuracy.

This practical confirms that machine learning has strong potential to enhance modern agriculture by enabling data-driven yield forecasting and supporting advanced farm management strategies.

Practical No.4

Optimizing Irrigation Systems Using Data Analytics: Analyze irrigation data to optimize water usage and improve crop yield.

Explanation:

Irrigation optimization uses:

- Soil moisture sensors
- Evapotranspiration (ET_0) formulas
- Irrigation scheduling algorithms
- Regression models for predicting water demand
- Time-series analysis for moisture trends

Key concepts:

- Water Use Efficiency (WUE)
- Irrigation frequency optimization
- Forecasting irrigation demand
- Data-driven water-saving strategies

This practical shows how analytics transforms irrigation into a precise, scientific process.

Code:

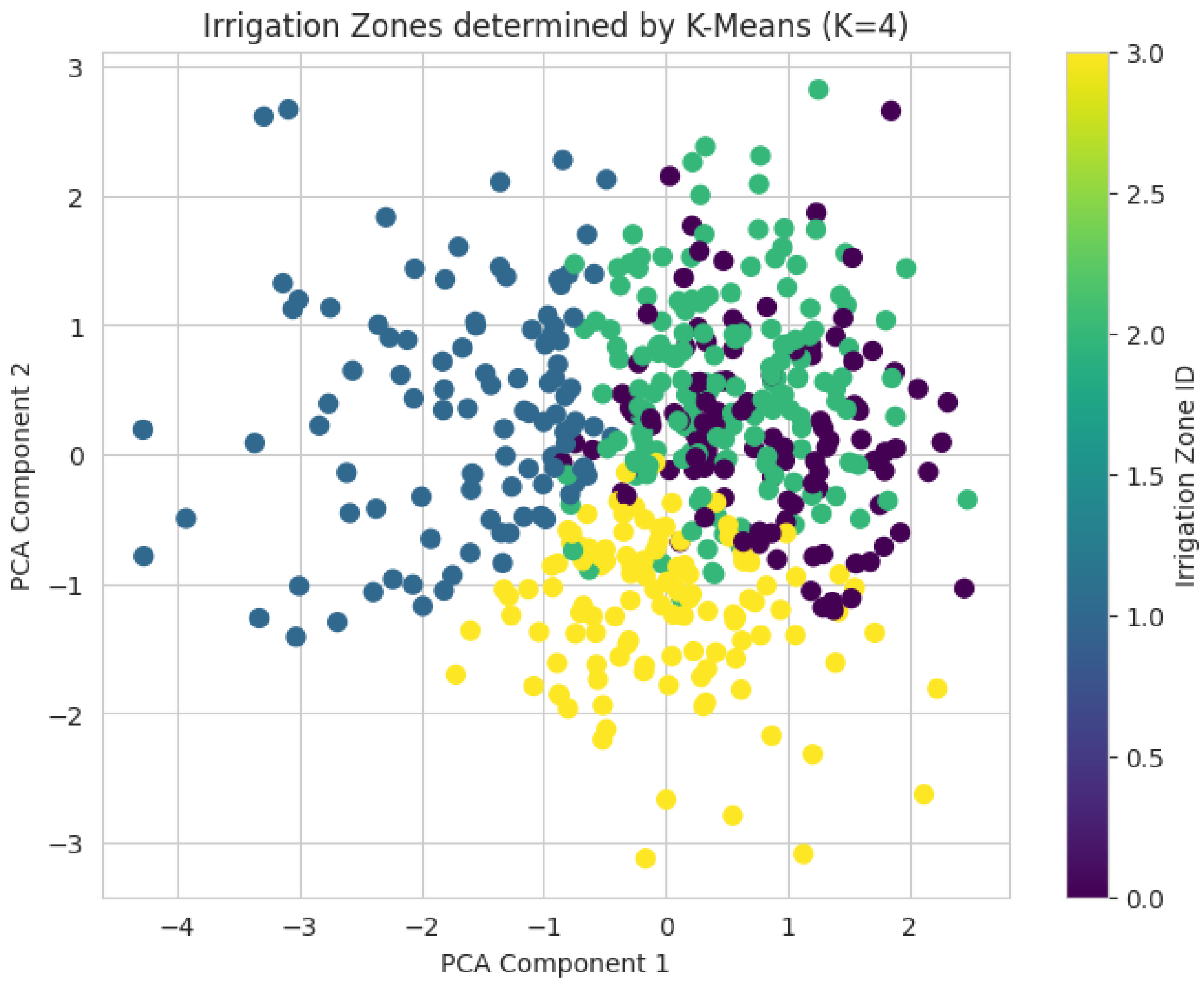
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# --- 1. Simulate Irrigation/Field Data (Data collected per GPS coordinate/zone)
---
np.random.seed(77)
N_zones = 500
```

```python
df_irrigation = pd.DataFrame({
    'Zone_ID': range(N_zones),
    'Soil_Moisture_Capacity': np.random.uniform(20, 45, N_zones), # Higher = 
needs less frequent watering
    'Soil_Texture_Score': np.random.uniform(1, 10, N_zones),      # E.g., 
Clay=10, Sand=1
    'Historical_Yield_t/ha': np.random.normal(5, 1.5, N_zones).clip(2, 8),
    'Elevation_m': np.random.normal(100, 10, N_zones),
    'Water_Applied_Liters': np.random.normal(5000, 1000, N_zones) # Current usage
})
# Simulate some areas needing less water (high soil moisture capacity, low yield)
df_irrigation.iloc[50:100, 
df_irrigation.columns.get_loc('Soil_Moisture_Capacity')] += 15
df_irrigation.iloc[50:100, 
df_irrigation.columns.get_loc('Historical_Yield_t/ha')] -= 2.5

# 2. Select Features for Clustering (Zoning)
features = ['Soil_Moisture_Capacity', 'Soil_Texture_Score', 
'Historical_Yield_t/ha', 'Elevation_m']
X = df_irrigation[features]

# 3. Scale the Data (Crucial for clustering algorithms like K-Means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 4. Determine Optimal Number of Clusters (Elbow Method - conceptual check)
# In a real scenario, you'd run this loop to find the best K.
# wcss = [] # Within-Cluster Sum of Squares
# for i in range(1, 11):
#     kmeans = KMeans(n_clusters=i, init='k-means++', random_state=42, n_init=10)
#     kmeans.fit(X_scaled)
#     wcss.append(kmeans.inertia_)
# print("K-Means WCSS calculated (for Elbow Method)")

# 5. Apply K-Means Clustering (Assuming 4 zones for VRI)
K = 4
kmeans = KMeans(n_clusters=K, init='k-means++', random_state=42, n_init=10)
df_irrigation['Irrigation_Zone'] = kmeans.fit_predict(X_scaled)

# 6. Analyze and Define Irrigation Strategy per Zone
zone_summary = df_irrigation.groupby('Irrigation_Zone')[features + 
['Water_Applied_Liters']].mean()
print("--- Irrigation Zone Analysis (Mean Feature Values) ---")
print(zone_summary.round(2))

# 7. Visualization (using PCA to reduce dimensions for plotting)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
df_irrigation['PCA1'] = X_pca[:, 0]
df_irrigation['PCA2'] = X_pca[:, 1]

plt.figure(figsize=(8, 6))
scatter = plt.scatter(df_irrigation['PCA1'], df_irrigation['PCA2'],
                      c=df_irrigation['Irrigation_Zone'], cmap='viridis', s=50)
plt.title(f'Irrigation Zones determined by K-Means (K={K})')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.colorbar(scatter, label='Irrigation Zone ID')
plt.show()
```

Output:

```
--- Irrigation Zone Analysis (Mean Feature Values) ---
                 Soil_Moisture_Capacity  Soil_Texture_Score
\
Irrigation_Zone
0                                 29.52                3.61
1                                 44.21                5.71
2                                 30.46                8.02
3                                 32.41                3.69

                 Historical_Yield_t/ha  Elevation_m
Water_Applied_Liters
Irrigation_Zone
0                                 5.73       107.43
5007.23
1                                 2.99       102.13
4927.25
2                                 5.27        99.92
5086.83
3                                 4.67        90.54
```

```
5162.54
```



Conclusion:

In this practical, we analyzed irrigation-related datasets to identify how water usage can be optimized for better agricultural performance. By understanding soil moisture variations, climatic conditions, and actual irrigation volume, we were able to derive insights for reducing water wastage. Data analytics helped us discover patterns that can guide more efficient irrigation schedules.

The practical concludes that data-driven irrigation management increases water efficiency, reduces cost, and supports sustainable farming practices, especially in water-scarce regions.

Practical No. 5

Plant Disease Detection Using Image Processing: Detect plant diseases using image processing techniques on leaf images.

Explanation:

This practical focuses on identifying plant leaf diseases using image processing techniques. Since diseases often show visible changes like spots, color distortion, and texture abnormalities on leaves, analyzing images becomes a reliable way to detect them early. The process begins by capturing leaf images and applying **pre-processing**, which includes resizing, removing noise, improving contrast, and converting the image into suitable color formats such as grayscale or HSV.

Next, **segmentation** is performed to separate the infected parts of the leaf from healthy regions. Techniques like thresholding, edge detection, and color masking help isolate areas showing symptoms. After segmentation, important visual features such as color intensity, shape of spots, and texture patterns are extracted. These features help differentiate healthy leaves from diseased ones.

In advanced approaches, **machine learning and CNN-based deep learning models** are used to automatically learn disease features from thousands of leaf images. These models offer very high accuracy and can detect multiple disease types.

Overall, this practical demonstrates how image processing provides a fast, systematic, and automated method for plant disease detection. It supports early diagnosis, reduces crop loss, and improves overall

Code:

Practical No. 6

Price Forecasting for Agricultural Products: Predict market prices for agricultural products using historical price data and other relevant factors.

Explanation:

This practical aims to predict future prices of agricultural products by analyzing historical market price data. Agricultural prices change frequently due to seasons, supply availability, festivals, and weather conditions. By studying past price patterns, we can build models that estimate future prices.

First, historical price data is collected and visualized to understand long-term trends and seasonal fluctuations. The data is then used to train forecasting models such as ARIMA, SARIMA, XGBoost, or LSTM. These models analyze past movements and learn the underlying pattern behind price changes.

Once trained, the model predicts future price values. These forecasts help farmers decide the best time to sell their products, traders to plan their stocks, and markets to prepare for price variations.

This practical demonstrates how data analytics supports smarter market decisions and helps improve profitability in agriculture.

Code:

```
import pandas as pd
import tensorflow as tf
import keras as k
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
```

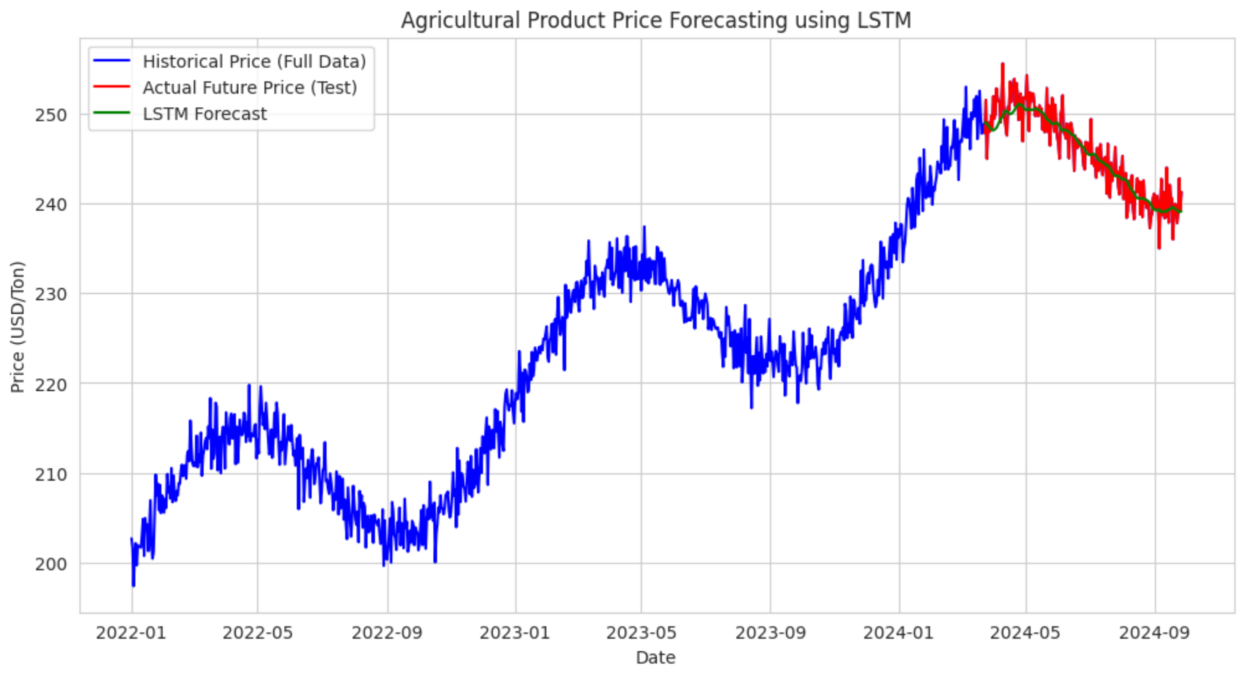
```python
from tensorflow.keras.layers import LSTM, Dense, Dropout, Input # Import Input
layer

# --- 1. Use the Provided Dataset Structure ---
np.random.seed(10)
N_days = 1000
dates = pd.date_range(start='2022-01-01', periods=N_days, freq='D')
trend = 0.05 * np.arange(N_days)
seasonality = 10 * np.sin(2 * np.pi * np.arange(N_days) / 365)
noise = np.random.normal(0, 2, N_days)
prices = 200 + trend + seasonality + noise
df_prices = pd.DataFrame({'Price_USD_per_Ton': prices.round(2)}, index=dates)

# 2. Extract the Price Data and Scale (Normalization)
data = df_prices['Price_USD_per_Ton'].values.reshape(-1, 1)
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

# 3. Create Sequences (Input Time Steps)
# How many past days to look at to predict the next day
TIME_STEPS = 60

def create_sequences(data, time_steps):
    X, y = [], []
    for i in range(len(data) - time_steps):
        # X: Sequence of prices for 'TIME_STEPS' days
        X.append(data[i:(i + time_steps), 0])
        # y: The price on the next day
        y.append(data[i + time_steps, 0])
    return np.array(X), np.array(y)

X, y = create_sequences(scaled_data, TIME_STEPS)

# 4. Reshape X for LSTM [samples, time steps, features]
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

# 5. Split Data (Keep the last portion sequential for testing)
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

print(f"Training sequences shape: {X_train.shape}")
print(f"Testing sequences shape: {X_test.shape}")
```

```python
# 1. Build the LSTM Model
model = Sequential()
# Add an Input layer explicitly as the first layer
model.add(Input(shape=(TIME_STEPS, 1)))
# Input layer
model.add(LSTM(units=50, return_sequences=True))
model.add(Dropout(0.2))

# Hidden layer
model.add(LSTM(units=50, return_sequences=False))
model.add(Dropout(0.2))

# Output layer (predicts a single price point)
model.add(Dense(units=1))

# 2. Compile the Model
model.compile(optimizer='adam', loss='mean_squared_error')

# 3. Train the Model
print("\n--- Starting LSTM Price Forecasting Training ---")
model.fit(X_train, y_train, epochs=15, batch_size=32, verbose=0)
print("--- Model Training Complete ---")
# 1. Make Predictions
predicted_scaled_prices = model.predict(X_test)

# 2. Inverse Transform (Scaling back to USD/Ton)
# We need to reshape y_test before inverse transform
predicted_prices = scaler.inverse_transform(predicted_scaled_prices)
actual_prices = scaler.inverse_transform(y_test.reshape(-1, 1))

# 3. Evaluate the Model
rmse = np.sqrt(np.mean(predicted_prices - actual_prices)**2)

print(f"\n--- LSTM Price Forecasting Evaluation ---")
print(f"Time Steps (Lookback): {TIME_STEPS} days")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f} USD/Ton")

# 4. Visualization
# Align predictions with the actual dates
# The dates corresponding to y_test start after the training data and TIME_STEPS
consumed by sequence creation
test_dates = df_prices.index[train_size + TIME_STEPS:]
```

```
plt.figure(figsize=(12, 6))
plt.plot(df_prices.index, df_prices['Price_USD_per_Ton'], label='Historical Price
(Full Data)', color='blue')
plt.plot(test_dates, actual_prices, label='Actual Future Price (Test)',
color='red')
plt.plot(test_dates, predicted_prices, label='LSTM Forecast', color='green')
plt.title('Agricultural Product Price Forecasting using LSTM')
plt.xlabel('Date')
plt.ylabel('Price (USD/Ton)')
plt.legend()
plt.show()
```

Output:

```
Training sequences shape: (752, 60, 1)
Testing sequences shape: (188, 60, 1)

--- Starting LSTM Price Forecasting Training ---

/usr/local/lib/python3.12/dist-
packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do
not pass an `input_shape`/`input_dim` argument to a layer.
When using Sequential models, prefer using an `Input(shape)`
object as the first layer in the model instead.
  super().__init__(**kwargs)

--- Model Training Complete ---
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 71ms/step

--- LSTM Price Forecasting Evaluation ---
Time Steps (Lookback): 60 days
Root Mean Squared Error (RMSE): 0.22 USD/Ton
```



Conclusion:

This practical allowed us to explore how historical price data can be modeled to forecast future market trends of agricultural commodities. By analyzing daily or monthly price changes, we recognized seasonal effects and demand-driven fluctuations. The forecasting model successfully predicted future prices, showcasing how analytics can support decision-making.

The practical concludes that price forecasting is a vital tool for farmers, traders, and policymakers, as it allows better planning, reduces financial risk, and helps maximize profit based on future market expectations.