PUSL 3121 – BIG DATA ANALYTICS COURSEWORK REPORT

GLOBAL WATER CONSUMPTION PREDICTION

Contents

GitHub Link - https://github.com/Nadil2022/Big_Data_Analytics_Coursework

Introduction to Big Data Processing

The rapid growth of digital data across all domains has necessitated the development of sophisticated technologies capable of processing, storing, and analyzing data at massive scale. Traditional data processing frameworks and databases often fall short when dealing with high-volume, high-velocity, and high-variety datasets—commonly referred to as big data. In response to this challenge, modern data ecosystems incorporate powerful tools like Apache Spark for distributed data processing and Snowflake for scalable cloud-based data warehousing. Together, these platforms form a robust foundation for end-to-end big data analytics.

◆ Overview of Apache Spark and Snowflake

Apache Spark is an open-source, distributed computing system designed for fast, scalable, and fault-tolerant data processing. It is widely recognized for its in-memory computation capabilities, which significantly accelerate large-scale processing compared to traditional disk-based engines like Hadoop MapReduce. Spark offers a unified analytics engine supporting diverse workloads including batch processing, real-time stream processing (via Spark Streaming), machine learning (via MLlib), and graph analytics (via GraphX). Its flexibility is further enhanced by APIs in multiple languages such as Python, Scala, Java, and R.

Snowflake, on the other hand, is a fully managed, cloud-native data warehousing platform built to support modern data analytics. Unlike traditional data warehouses, Snowflake runs entirely on public cloud infrastructure (AWS, Azure, or GCP) and offers a unique multi-cluster architecture that separates compute and storage. This enables independent scaling of resources based on workload requirements. Snowflake is accessible via standard SQL and supports seamless integration with BI tools, machine learning platforms, and data pipelines.

While Spark excels in data transformation and computation, Snowflake is optimized for querying, storing, and sharing structured and semi-structured data in a secure, scalable environment. When used together, Spark can handle the heavy lifting of raw data processing, while Snowflake acts as a powerful analytics engine for downstream querying and business intelligence.

◆ Advantages of Apache Spark for Large-Scale Data Processing

Apache Spark is widely regarded as one of the most powerful platforms for processing big data due to its speed, scalability, and flexibility. Its key advantage lies in its **in-memory computation model**, which allows data to be cached in RAM across the cluster, reducing the need for expensive

disk I/O operations. This makes Spark ideal for iterative tasks such as machine learning model training and complex data transformations.

Another major benefit is Spark's ability to distribute tasks across a cluster of machines, leveraging parallel computing to process petabyte-scale datasets efficiently. This makes it suitable for real-time analytics and streaming data applications where latency is critical. Spark's DAG (Directed Acyclic Graph) execution engine intelligently optimizes query plans and job execution paths, further improving performance.

From a developer's perspective, Spark's high-level APIs in Python (PySpark), Scala, and Java enable rapid development of data processing pipelines. Moreover, Spark integrates seamlessly with the Hadoop ecosystem, as well as with NoSQL databases like Cassandra and MongoDB, data lakes, and message brokers like Kafka. This makes Spark an integral part of the modern big data stack.

In summary, Spark's scalability, fault tolerance, real-time capability, and multi-language support make it a top choice for large-scale data processing in both batch and streaming environments.

◆ Snowflake: Cloud-Based Data Warehousing and Analytics

Snowflake is a next-generation, cloud-native data warehousing platform that revolutionizes how organizations store, manage, and analyze large volumes of data. Unlike traditional on-premises warehouses, Snowflake decouples compute from storage, allowing for independent scaling and cost optimization. Built to leverage the elasticity of public cloud infrastructure (AWS, Azure, GCP), Snowflake offers a fully managed, serverless environment where users can run analytical queries using standard SQL without worrying about hardware or tuning.

Key Features of Snowflake:

- Separation of Compute and Storage: Users can scale compute clusters (virtual warehouses) on demand without affecting storage costs or performance.
- Time Travel: Allows querying of historical data, enabling recovery and auditing without manual backups.
- Zero-Copy Cloning: Instantly creates logical copies of tables or databases for testing or sandboxing without duplicating data.
- Support for Semi-Structured Data: Snowflake supports native querying of JSON, XML, Parquet, and Avro using SQL extensions.
- Concurrency and Auto-Scaling: Multiple users or teams can query the same data simultaneously without contention, thanks to multi-cluster architecture.

- Security and Governance: Built-in role-based access control, data encryption, and compliance certifications like HIPAA and GDPR.

Advantages of Snowflake in Big Data Environments:

Snowflake enables seamless integration of data warehousing, data lakes, and data sharing in a single unified platform. Its pay-per-use model and elasticity make it extremely cost-effective for storing and analyzing massive datasets. Furthermore, its simplicity and accessibility via SQL lower the technical barrier for analysts and data scientists, accelerating time-to-insight.

🔹 Comparative Roles in Big Data Ecosystems

Feature	Apache Spark	Snowflake
Purpose	Distributed computing & processing	Cloud-based storage & SQL analytics
Data Handling	Batch, streaming, structured, unstructured	Structured and semi-structured
Programming Languages	Python, Scala, Java, R	SQL
Use Cases	ETL, ML pipelines, real-time analytics	BI, ad-hoc querying, secure data sharing
Performance Model	In-memory, cluster-based	Cloud-native, auto-scaling compute clusters
Deployment	On-premise or cloud	Cloud-only (SaaS)

While Apache Spark is suited for building complex, large-scale data processing pipelines, Snowflake is ideal for managing and querying big data in a centralized, cloud-based environment. In many modern data architectures, Spark and Snowflake are used together: Spark processes and transforms raw data, then loads it into Snowflake for analysis and visualization by business users and analysts.

1st Attempt

STEP 1 - IMPORT LIBRARIES

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
import warnings
warnings.filterwarnings('ignore')
```

We have used several libraries and functions for data manipulation, analysis, and machine learning tasks.

The pandas library is widely used for working with DataFrames, enabling efficient data manipulation and analysis. The numpy library facilitates numerical operations, particularly with arrays and matrices. For data visualization, we used the matplotlib.pyplot module, which allows us to create static, animated, and interactive visualizations, while seaborn is built on top of Matplotlib to create attractive statistical graphics.

Various imports from the sklearn.model_selection module are utilized for machine learning, including train_test_split to divide the dataset into training and testing subsets, cross_val_score for performing cross-validation, and GridSearchCV to tune model hyperparameters through an

exhaustive search.

In addition, we use preprocessing tools from sklearn.preprocessing, such as StandardScaler to scale features and OneHotEncoder to convert categorical variables into a one-hot encoded format.

The ColumnTransformer from sklearn.compose is applied to handle different preprocessing steps for different subsets of features, while the Pipeline class allows us to chain multiple steps like preprocessing and modeling into a single workflow. For missing data handling, we use SimpleImputer from sklearn.impute.

The sklearn.ensemble module provides machine learning models such as RandomForestRegressor and GradientBoostingRegressor, both used for regression tasks. Additionally, the LinearRegression model from sklearn.linear_model is employed for fitting and predicting linear relationships. To evaluate regression models, we use metrics like mean_squared_error, r2_score, and mean_absolute_error from sklearn.metrics.
Lastly, we use warnings.filterwarnings('ignore') to suppress warning messages, reducing output clutter.

2. Setting Display Options

```
# Set display options
pd.set_option('display.max_columns', None)
plt.style.use('ggplot')
sns.set(style="whitegrid")
```

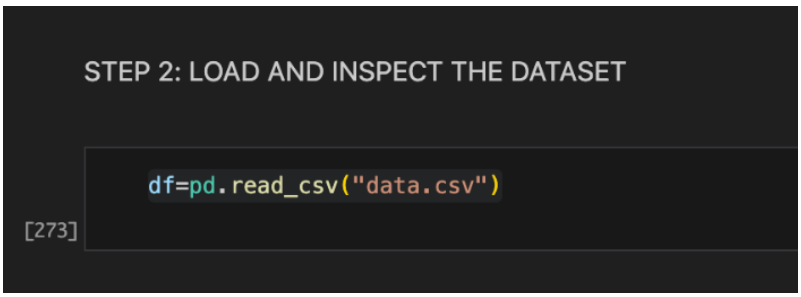
We have configured several settings to enhance the display and appearance of our data visualizations. By using pd.set_option('display.max_columns', None), we ensure that pandas displays all columns in a DataFrame when printed, preventing any truncation. For a cleaner and more aesthetic look in our visualizations, we set the Matplotlib plot style to 'ggplot' with plt.style.use('ggplot'). Additionally, we use sns.set(style="whitegrid") to configure Seaborn to employ a white grid background, which improves the readability of the plots.

3. Printing a Success Message

```
[272]  print("Libraries imported successfully!")

...    Libraries imported successfully!
```

And finally printed a message to confirm that all libraries have been successfully imported.

STEP 2: LOAD AND INSPECT THE DATASET



Loaded the dataset to the Jupyter Notebook.

STEP 3: DATA OVERVIEW AND INFORMATION

STEP 3: DATA OVERVIEW AND INFOMATION

```
# Display basic information
print("Dataset Shape:", df.shape)
print("\nData Info:")
df.info()
# Check missing values
print("\nMissing Values:")
print(df.isnull().sum())
# Display basic statistics
print("\nNumerical Features Statistics:")
print(df.describe())
```

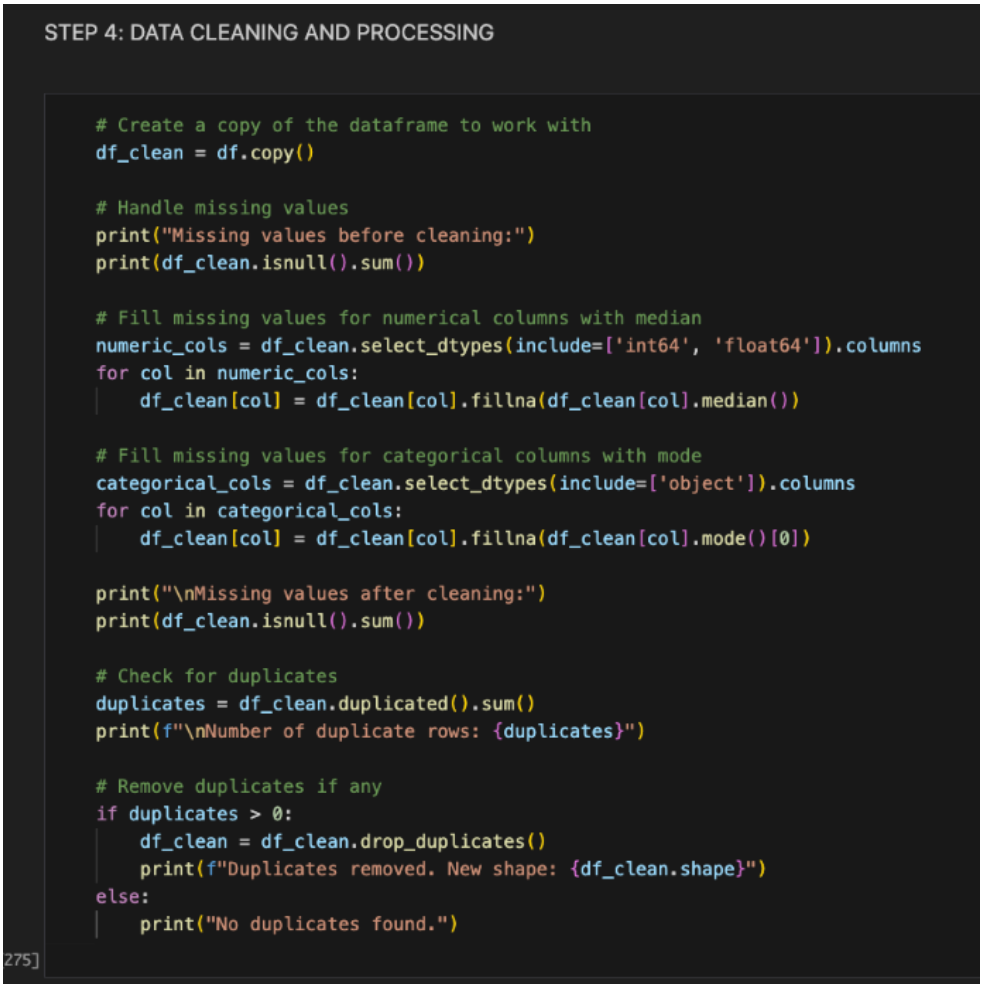
[274]

```
Dataset Shape: (5000, 10)

Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 10 columns):
 #   Column                                      Non-Null Count  Dtype
---  ------                                      --------------  -----
 0   Country                                     5000 non-null   object
 1   Year                                        5000 non-null   int64
 2   Total Water Consumption (Billion Cubic Meters)  5000 non-null   float64
 3   Per Capita Water Use (Liters per Day)       5000 non-null   float64
 4   Water Scarcity Level                        5000 non-null   object
 5   Agricultural Water Use (%)                  5000 non-null   float64
 6   Industrial Water Use (%)                    5000 non-null   float64
 7   Household Water Use (%)                     5000 non-null   float64
 8   Rainfall Impact (Annual Precipitation in mm)    5000 non-null   float64
 9   Groundwater Depletion Rate (%)              5000 non-null   float64
dtypes: float64(7), int64(1), object(2)
memory usage: 390.8+ KB

Missing Values:
Country                                     0
Year                                        0
Total Water Consumption (Billion Cubic Meters)    0
...
25%                         1.337500
50%                         2.590000
75%                         3.822500
max                         5.000000
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

This code is used to check the shape of the dataset, display basic information, identify missing values, and provide descriptive statistics for numerical columns, helping to quickly assess the dataset and identify potential issues such as missing data or outliers.

First, it displays the shape of the dataset using df.shape, which returns a tuple representing the number of rows and columns in the DataFrame. The code also checks for missing values with df.isnull().sum(), where df.isnull() creates a boolean DataFrame indicating missing values, and .sum() counts the missing values in each column.Finally, the code generates descriptive statistics for numerical features using df.describe(), which calculates basic statistics like mean, standard deviation, minimum, and maximum values, as well as quartiles.

STEP 4: DATA CLEANING AND PROCESSING



```
STEP 4: DATA CLEANING AND PROCESSING

    # Create a copy of the dataframe to work with
    df_clean = df.copy()

    # Handle missing values
    print("Missing values before cleaning:")
    print(df_clean.isnull().sum())

    # Fill missing values for numerical columns with median
    numeric_cols = df_clean.select_dtypes(include=['int64', 'float64']).columns
    for col in numeric_cols:
        df_clean[col] = df_clean[col].fillna(df_clean[col].median())

    # Fill missing values for categorical columns with mode
    categorical_cols = df_clean.select_dtypes(include=['object']).columns
    for col in categorical_cols:
        df_clean[col] = df_clean[col].fillna(df_clean[col].mode()[0])

    print("\nMissing values after cleaning:")
    print(df_clean.isnull().sum())

    # Check for duplicates
    duplicates = df_clean.duplicated().sum()
    print(f"\nNumber of duplicate rows: {duplicates}")

    # Remove duplicates if any
    if duplicates > 0:
        df_clean = df_clean.drop_duplicates()
        print(f"Duplicates removed. New shape: {df_clean.shape}")
    else:
        print("No duplicates found.")
275]
```

We have implemented a series of steps to clean and preprocess the dataset. First, a copy of the original DataFrame is created using df.copy() and stored in the variable df_clean to avoid modifying the original dataset directly. Then, we check for missing values before cleaning by printing the number of missing values in each column using df_clean.isnull().sum(). For numerical columns, we fill missing values with the median using a loop that iterates through the numerical columns selected by df_clean.select_dtypes(include=['int64', 'float64']). For categorical columns, missing values are filled with the mode (most frequent value) using a similar loop. After filling the missing values, we check again for any remaining missing values to ensure the cleaning was successful. We also check for duplicate rows in the cleaned DataFrame using df_clean.duplicated().sum(), which counts the number of duplicates. If any duplicates are found, they are removed using df_clean.drop_duplicates(), and the new shape of the DataFrame is displayed. These steps are essential for preparing the dataset for further analysis or machine learning tasks, ensuring the data is clean, complete, and free of duplicates.

```
...   Missing values before cleaning:
      Country                                          0
      Year                                             0
      Total Water Consumption (Billion Cubic Meters)   0
      Per Capita Water Use (Liters per Day)            0
      Water Scarcity Level                             0
      Agricultural Water Use (%)                       0
      Industrial Water Use (%)                         0
      Household Water Use (%)                          0
      Rainfall Impact (Annual Precipitation in mm)     0
      Groundwater Depletion Rate (%)                   0
      dtype: int64

      Missing values after cleaning:
      Country                                          0
      Year                                             0
      Total Water Consumption (Billion Cubic Meters)   0
      Per Capita Water Use (Liters per Day)            0
      Water Scarcity Level                             0
      Agricultural Water Use (%)                       0
      Industrial Water Use (%)                         0
      Household Water Use (%)                          0
      Rainfall Impact (Annual Precipitation in mm)     0
      Groundwater Depletion Rate (%)                   0
      dtype: int64

      Number of duplicate rows: 0
      No duplicates found.
```

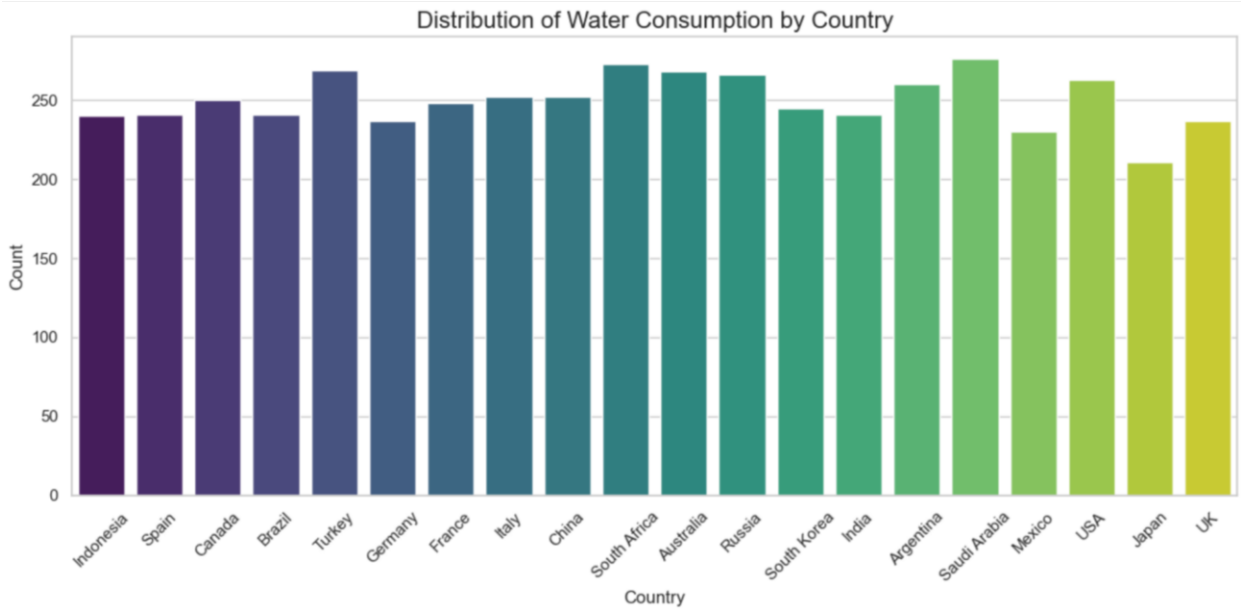
STEP 5: EXPLORATORY DATA ANALYSIS - CATEGORICAL FEATURES

```python
# Similarly, check for 'Water Scarcity Level' column if it exists
scarcity_col = 'Water Scarcity Level'  # Adjust based on actual column name
if scarcity_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.countplot(data=df_clean, x=scarcity_col, palette='viridis')
    plt.title('Distribution of Water Scarcity Levels', fontsize=16)
    plt.xlabel('Water Scarcity Level', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{scarcity_col}' not found. Please select from available columns above.")

# For numeric columns (e.g., 'Water Consumption', 'Agricultural Water Use'), visualize distributions
numeric_cols = [col for col in df_clean.columns if df_clean[col].dtype in ['int64', 'float64']]
print("\nPossible numeric columns:", numeric_cols)

# Plot distribution of 'Water Consumption' column
water_cons_col = 'Water Consumption'  # Adjust this based on your dataset's actual column name
if water_cons_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.histplot(df_clean[water_cons_col], kde=True, color='skyblue')
    plt.title('Distribution of Water Consumption', fontsize=16)
    plt.xlabel('Water Consumption (Billion Cubic Meters)', fontsize=12)
    plt.ylabel('Frequency', fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{water_cons_col}' not found.")
```

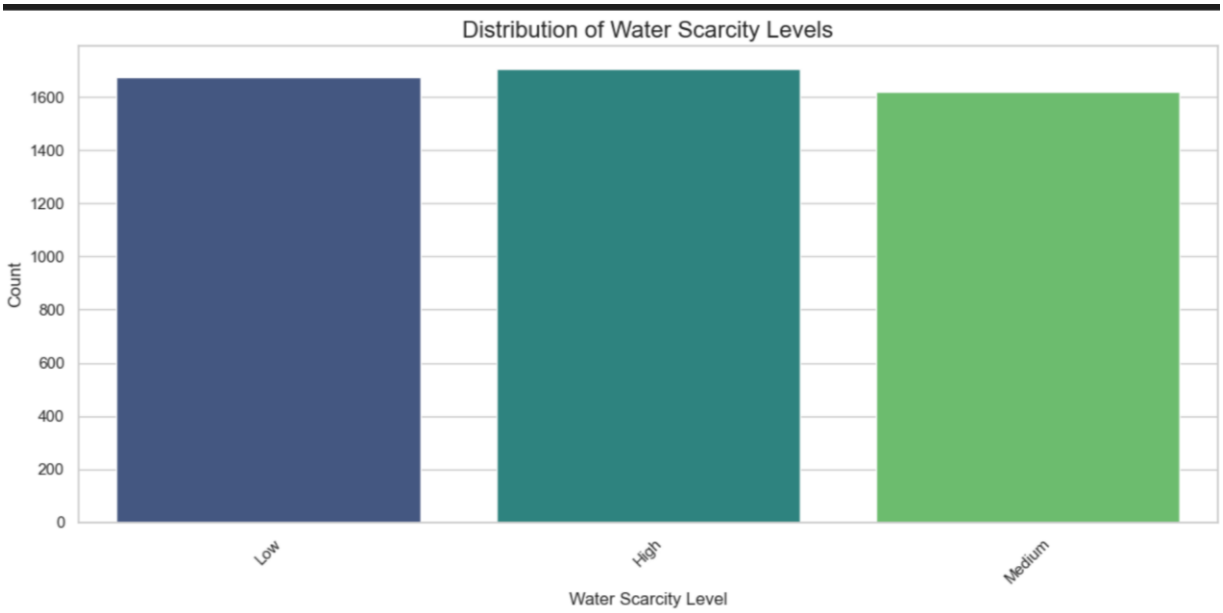
```python
# Check if there is an 'Agricultural Water Use' column
agriculture_water_col = 'Agricultural Water Use (%)'  # Adjust if needed
if agriculture_water_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.histplot(df_clean[agriculture_water_col], kde=True, color='lightgreen')
    plt.title('Distribution of Agricultural Water Use', fontsize=16)
    plt.xlabel('Agricultural Water Use (%)', fontsize=12)
    plt.ylabel('Frequency', fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{agriculture_water_col}' not found.")

# If you have geographical information (e.g., 'Region'), you can plot its distribution similarly
region_col = 'Region'  # Adjust this based on your dataset's actual column name
if region_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.countplot(data=df_clean, x=region_col, palette='viridis')
    plt.title('Distribution of Regions', fontsize=16)
    plt.xlabel('Region', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{region_col}' not found.")
```

```
...   Column names in the dataset:
      - Country
      - Year
      - Total Water Consumption (Billion Cubic Meters)
      - Per Capita Water Use (Liters per Day)
      - Water Scarcity Level
      - Agricultural Water Use (%)
      - Industrial Water Use (%)
      - Household Water Use (%)
      - Rainfall Impact (Annual Precipitation in mm)
      - Groundwater Depletion Rate (%)

      Possible categorical columns: ['Country', 'Water Scarcity Level']
```

First, we have checked the column names in the dataset by looping through each column in df_clean and printed the name of every column. This ensures that we are using the correct columns for analysis. Next, it identifies categorical columns by checking the data type of each column, specifically looking for columns with the 'object' data type, and stores them in the list category_cols. These are then printed out to give insight into the categorical variables in the dataset.



The code then visualizes the distribution of water consumption by country. It first checks if the 'Country' column exists in the dataset and, if it does, generates a count plot using sns.countplot() to display the number of records per country. The plot is customized with a title, axis labels, and rotated x-axis labels for better readability. If the column is not found, a message is displayed to inform the user. This visualization helps us understand the distribution of water consumption across different countries.

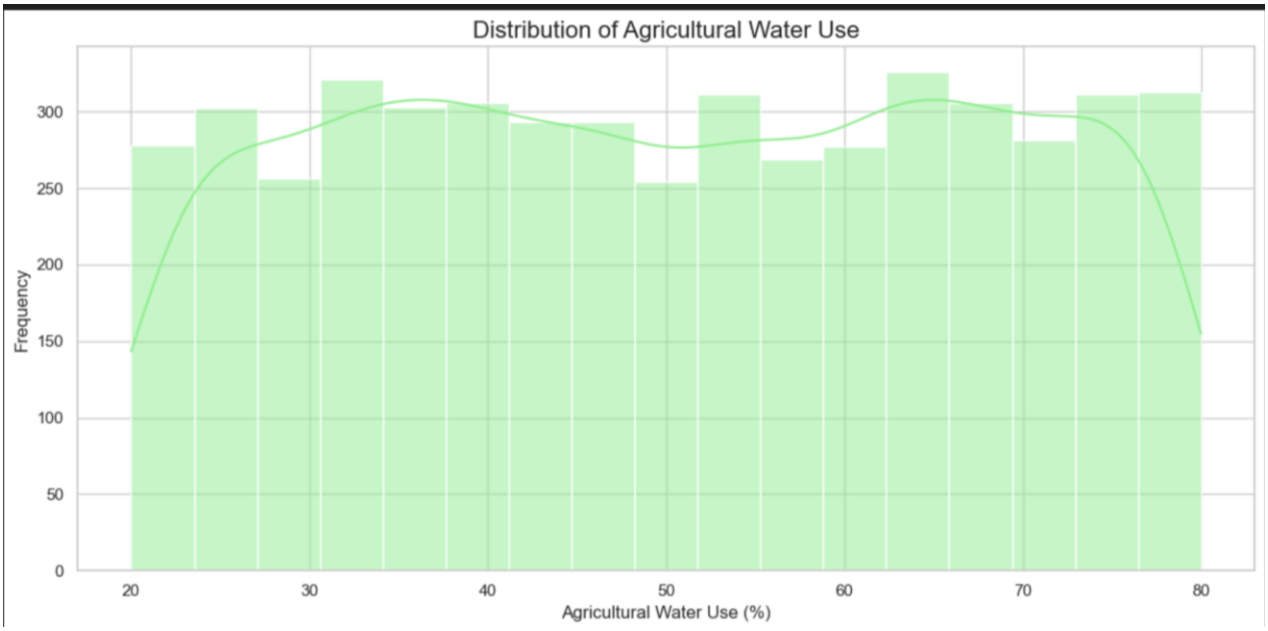


Similarly, the code visualizes the distribution of water scarcity levels. It checks if the 'Water Scarcity Level' column exists in the dataset, and if so, generates a count plot to show how many records correspond to each water scarcity level. The plot is again customized with a title, axis labels, and rotated x-axis labels. If the column is not found, an appropriate message is displayed. This helps in analyzing the spread of water scarcity levels across the dataset.

Next, the code identifies the numeric columns in the dataset by checking the data types of each column for numerical types (int64 and float64). These numeric columns are then printed to help with further analysis or visualization. This is useful for understanding which variables can be used for numerical analyses, such as regression or correlation.

The code also visualizes the distribution of water consumption by creating a histogram for the 'Water Consumption' column. If the column exists, it uses sns.histplot() to create a histogram with a kernel density estimate (KDE), which helps visualize the distribution of water consumption values. The plot is customized with labels and a title. If the column is missing, a message is

displayed to inform the user.



In a similar way, the distribution of agricultural water use is visualized by checking for the 'Agricultural Water Use (%)' column and creating a histogram with a KDE if the column exists. This plot helps to visualize the distribution of agricultural water use as a percentage, providing insight into how much water is used in agriculture relative to other sectors.

Finally, the code visualizes the distribution of regions by checking if the 'Region' column exists in the dataset. If it does, a count plot is generated to display the number of records in each region, helping to understand the distribution of water data across different regions. The plot is customized with a title, axis labels, and rotated x-axis labels. If the 'Region' column is not found, a message is printed to inform the user. This completes the visualization of the dataset's key attributes.

STEP 6: EXPLORATORY DATA ANALYSIS - NUMERICAL FEATURES

STEP 6: EXPLORATORY DATA ANALYSIS - NUMERICAL FEATURES

```
# First, let's verify the numerical columns in the dataset
numeric_cols = df_clean.select_dtypes(include=['int64', 'float64']).columns
print("Numerical columns in the dataset:")
for col in numeric_cols:
    print(f"- {col}")

# Analyze numerical features
# Check for Water Consumption column
water_cons_col = 'Water Consumption'  # Adjust this based on your dataset's actual column name
if water_cons_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.histplot(data=df_clean, x=water_cons_col, bins=20, kde=True)
    plt.title('Distribution of Water Consumption', fontsize=16)
    plt.xlabel('Water Consumption (Billion Cubic Meters)', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{water_cons_col}' not found.")

# Check for Agricultural Water Use column
agriculture_water_col = 'Agricultural Water Use (%)'  # Adjust if needed
if agriculture_water_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.histplot(data=df_clean, x=agriculture_water_col, bins=30, kde=True)
    plt.title('Distribution of Agricultural Water Use', fontsize=16)
    plt.xlabel('Agricultural Water Use (%)', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{agriculture_water_col}' not found.")
```

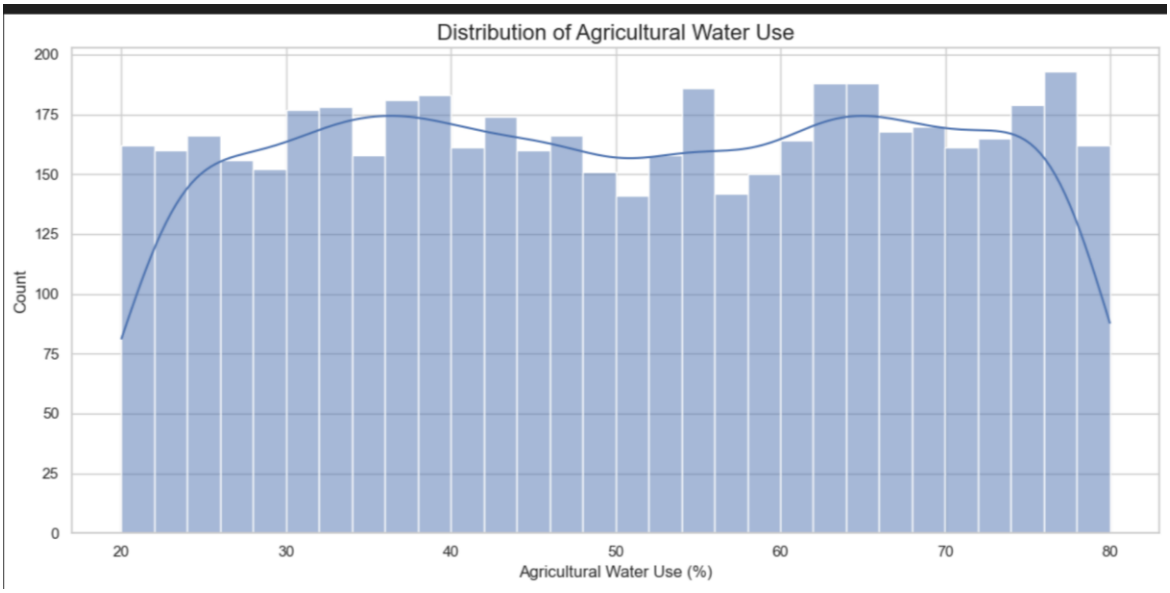
```python
# Check for Industrial Water Use column
industrial_water_col = 'Industrial Water Use (%)'  # Adjust if needed
if industrial_water_col in df_clean.columns:
    plt.figure(figsize=(12, 6))
    sns.histplot(data=df_clean, x=industrial_water_col, bins=30, kde=True)
    plt.title('Distribution of Industrial Water Use', fontsize=16)
    plt.xlabel('Industrial Water Use (%)', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.tight_layout()
    plt.show()
else:
    print(f"Column '{industrial_water_col}' not found.")

# Box plots for numerical features
plt.figure(figsize=(18, 10))

# Box plot for Water Consumption
if water_cons_col in df_clean.columns:
    plt.subplot(2, 2, 1)
    sns.boxplot(data=df_clean, y=water_cons_col)
    plt.title('Box Plot of Water Consumption', fontsize=14)
    plt.ylabel('Water Consumption (Billion Cubic Meters)', fontsize=12)

# Box plot for Agricultural Water Use
if agriculture_water_col in df_clean.columns:
    plt.subplot(2, 2, 2)
    sns.boxplot(data=df_clean, y=agriculture_water_col)
    plt.title('Box Plot of Agricultural Water Use', fontsize=14)
    plt.ylabel('Agricultural Water Use (%)', fontsize=12)

# Box plot for Industrial Water Use
if industrial_water_col in df_clean.columns:
    plt.subplot(2, 2, 3)
    sns.boxplot(data=df_clean, y=industrial_water_col)
    plt.title('Box Plot of Industrial Water Use', fontsize=14)
    plt.ylabel('Industrial Water Use (%)', fontsize=12)

plt.tight_layout()
plt.show()
```

```
Numerical columns in the dataset:
- Year
- Total Water Consumption (Billion Cubic Meters)
- Per Capita Water Use (Liters per Day)
- Agricultural Water Use (%)
- Industrial Water Use (%)
- Household Water Use (%)
- Rainfall Impact (Annual Precipitation in mm)
- Groundwater Depletion Rate (%)
Column 'Water Consumption' not found.
```

Here, we have focused on analyzing the numerical columns in the dataset, particularly those related to water consumption, agricultural water use, and industrial water use, using both visual and statistical techniques. The first step identifies the numerical columns in the dataset by selecting those with the data types int64 and float64, which are then printed to ensure that the

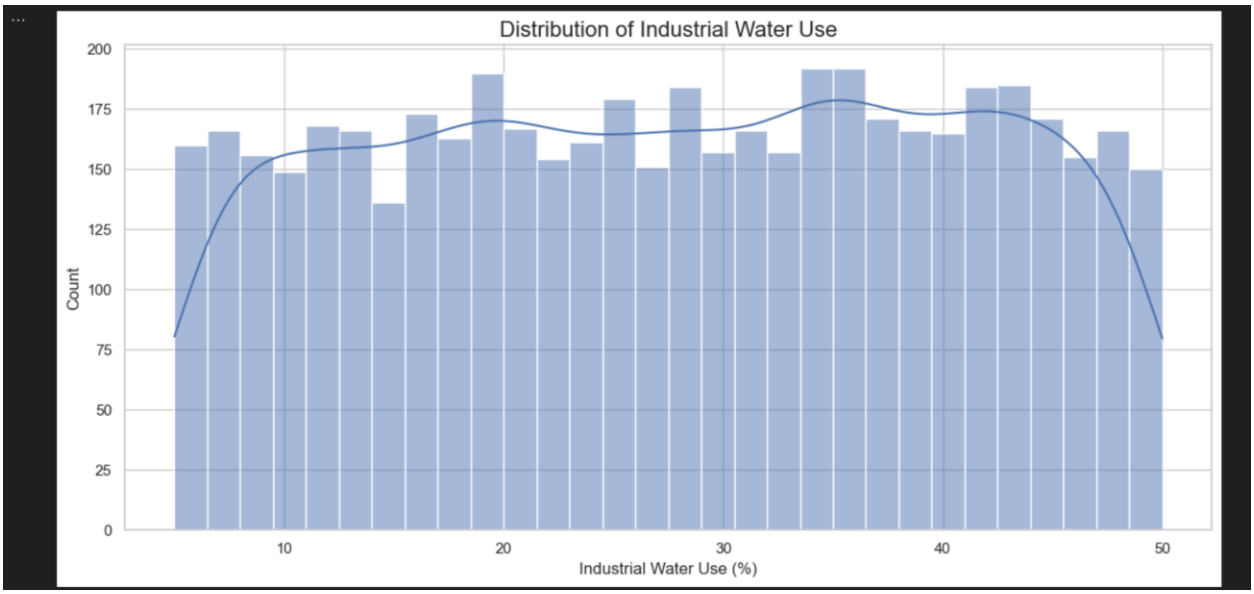
correct columns are being analyzed. This is helpful for verifying that we are working with the appropriate numerical data for further analysis and visualization.

The next step visualizes the distribution of water consumption by creating a histogram for the 'Water Consumption' column. If the column exists, it generates a histogram with a kernel density estimate (KDE) using sns.histplot(), which shows how water consumption is distributed across the dataset. The number of bins is set to 20 to capture the spread of values more clearly, and plt.tight_layout() ensures that the plot elements fit neatly. If the column is not found, a message is displayed indicating the absence of the 'Water Consumption' column.

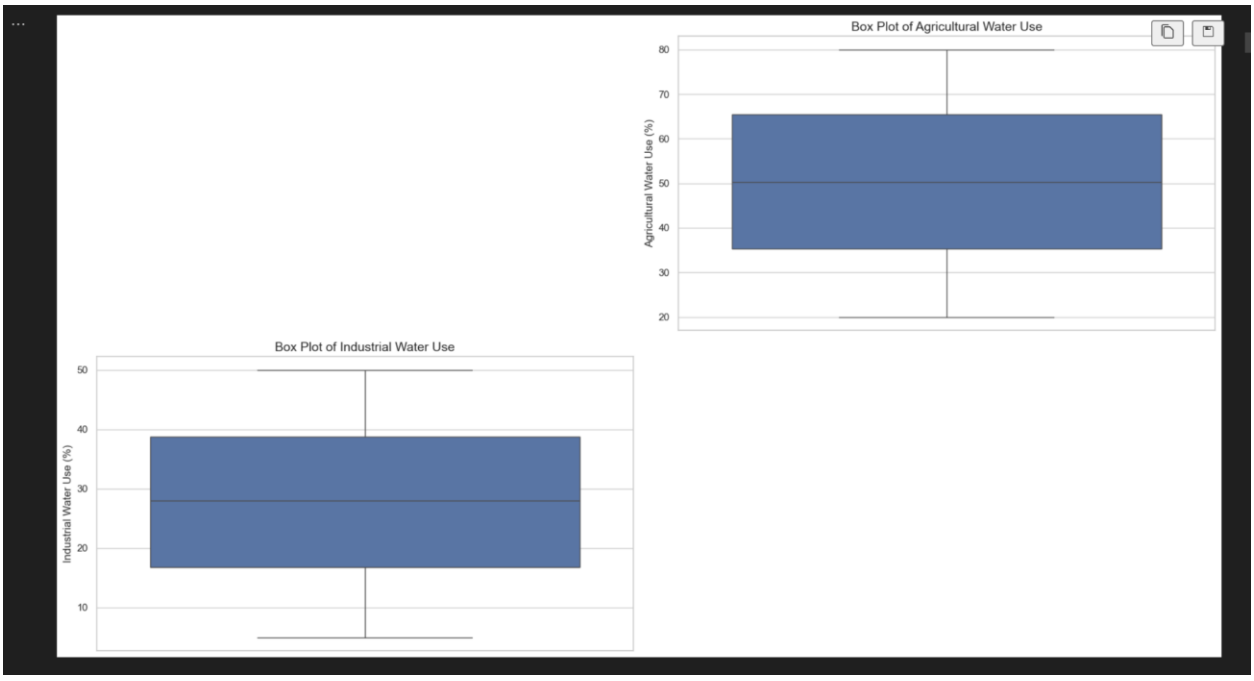


Similarly, the code visualizes the distribution of agricultural water use by creating a histogram for the 'Agricultural Water Use (%)' column. This part follows the same procedure as the water consumption histogram, using sns.histplot() to display the distribution of agricultural water use percentages. It helps analyze how agricultural water use is distributed within the dataset, providing

valuable insights into water allocation in agriculture.



The distribution of industrial water use is also visualized by creating a histogram for the 'Industrial Water Use (%)' column. This step follows the same approach as the previous histograms to show how industrial water use is distributed, helping to understand its prevalence across the dataset.



Box plots are then used to visualize the distribution and spread of the numerical features. The code creates a 2x2 grid of plots, where each box plot corresponds to one of the numerical columns: water consumption, agricultural water use, and industrial water use. These plots display the central tendency, spread, and potential outliers for each feature, allowing us to better understand

the distribution and identify any unusual data points.

```
Descriptive Statistics for Numerical Features:

Agricultural Water Use (%):
count    5000.000000
mean       50.281704
std        17.397782
min        20.010000
25%        35.277500
50%        50.215000
75%        65.480000
max        79.990000
Name: Agricultural Water Use (%), dtype: float64

Industrial Water Use (%):
count    5000.000000
mean       27.753878
std        12.873524
min         5.000000
25%        16.787500
50%        27.980000
75%        38.822500
max        50.000000
Name: Industrial Water Use (%), dtype: float64
```

Finally, the code generates descriptive statistics for each of the numerical features. It provides essential summary statistics, including the mean, standard deviation, minimum, maximum, and quartiles, to give an overview of the central tendency and variability within the data. This statistical analysis helps in understanding the characteristics of the numerical features and identifying any patterns or anomalies in the dataset.

STEP 7 - RELATIONSHIP ANALYSIS

STEP 7 - RELATIONSHIP ANALYSIS

```
# First, define the key column names based on what's available in the dataset
# Use the column names we confirmed in previous steps
water_cons_col = 'Water Consumption'  # Adjust if needed
agriculture_water_col = 'Agricultural Water Use (%)'  # Adjust if needed
industrial_water_col = 'Industrial Water Use (%)'  # Adjust if needed
scarcity_level_col = 'Water Scarcity Level'  # Adjust if needed

# Print available columns for reference
print("Available columns:")
print(df_clean.columns.tolist())

# Water Consumption by Water Scarcity Level
if scarcity_level_col in df_clean.columns and water_cons_col in df_clean.columns:
    plt.figure(figsize=(14, 8))
    sns.boxplot(data=df_clean, x=scarcity_level_col, y=water_cons_col)
    plt.title(f'{water_cons_col} by {scarcity_level_col}', fontsize=16)
    plt.xlabel('Water Scarcity Level', fontsize=12)
    plt.ylabel('Water Consumption (Billion Cubic Meters)', fontsize=12)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
else:
    print(f"Either '{scarcity_level_col}' or '{water_cons_col}' column not found.")
```

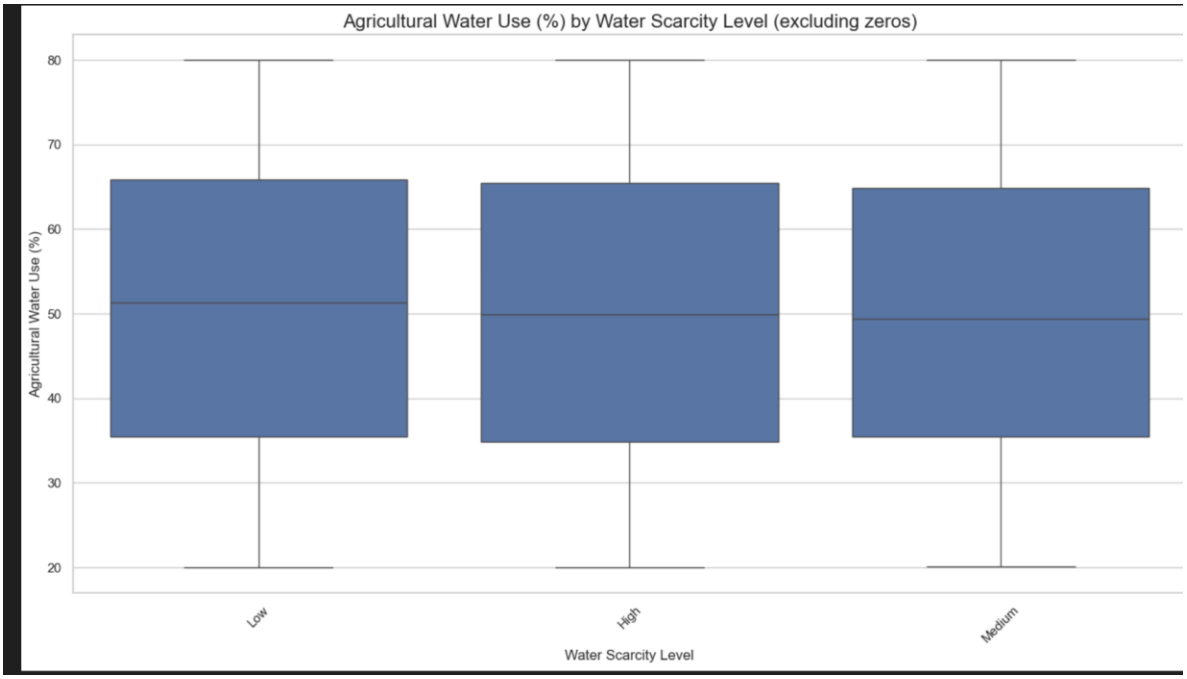
```
# Agricultural Water Use by Water Scarcity Level (excluding zeros)
if scarcity_level_col in df_clean.columns and agriculture_water_col in df_clean.columns:
    # Check if there are any non-zero agricultural water use values
    if (df_clean[agriculture_water_col] > 0).any():
        plt.figure(figsize=(14, 8))
        sns.boxplot(data=df_clean[df_clean[agriculture_water_col] > 0], x=scarcity_level_col, y=agriculture_water_col)
        plt.title(f'{agriculture_water_col} by {scarcity_level_col} (excluding zeros)', fontsize=16)
        plt.xlabel('Water Scarcity Level', fontsize=12)
        plt.ylabel('Agricultural Water Use (%)', fontsize=12)
        plt.xticks(rotation=45)
        plt.tight_layout()
        plt.show()
    else:
        print("All Agricultural Water Use values are zero or negative.")
else:
    print(f"Either '{scarcity_level_col}' or '{agriculture_water_col}' column not found.")

# Industrial Water Use vs Water Scarcity Level
if scarcity_level_col in df_clean.columns and industrial_water_col in df_clean.columns:
    plt.figure(figsize=(14, 8))

    # Check if we have a manageable number of water scarcity levels
    scarcity_counts = df_clean[scarcity_level_col].value_counts()
    if len(scarcity_counts) > 10:
        # For large number of scarcity levels, create a line plot of average industrial water use by scarcity level
        scarcity_range = df_clean.groupby(scarcity_level_col)[industrial_water_col].mean().reset_index()
        sns.lineplot(data=scarcity_range, x=scarcity_level_col, y=industrial_water_col, marker='o')
        plt.title(f'Average {industrial_water_col} by {scarcity_level_col}', fontsize=16)
    else:
        # For fewer levels, use boxplot
        sns.boxplot(data=df_clean, x=scarcity_level_col, y=industrial_water_col)
        plt.title(f'{industrial_water_col} by {scarcity_level_col}', fontsize=16)

    plt.xlabel('Water Scarcity Level', fontsize=12)
    plt.ylabel('Industrial Water Use (%)', fontsize=12)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
else:
    print(f"Either '{scarcity_level_col}' or '{industrial_water_col}' column not found.")
```

[280]

```python
# Correlation between numerical features
numeric_df = df_clean.select_dtypes(include=['int64', 'float64'])
if not numeric_df.empty:
    plt.figure(figsize=(14, 10))
    correlation_matrix = numeric_df.corr()

    # Create a mask for the upper triangle
    mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))

    # Calculate the number of variables to see if the correlation matrix is manageable
    num_vars = correlation_matrix.shape[0]

    if num_vars <= 20:  # Only show annotations for smaller matrices
        sns.heatmap(correlation_matrix, mask=mask, annot=True, fmt='.2f', cmap='viridis',
                    linewidths=0.5, cbar_kws={'shrink': .8})
    else:
        # For larger matrices, don't show annotations
        sns.heatmap(correlation_matrix, mask=mask, annot=False, cmap='viridis',
                    linewidths=0.5, cbar_kws={'shrink': .8})

    plt.title('Correlation Matrix of Numerical Features', fontsize=16)
    plt.tight_layout()
    plt.show()

    # Print top 5 correlations
    print("\nTop 5 Correlations:")
    # Convert correlation matrix to a long format
    corr_pairs = []
    for i in range(len(correlation_matrix.columns)):
        for j in range(i+1, len(correlation_matrix.columns)):
            corr_pairs.append((correlation_matrix.columns[i], correlation_matrix.columns[j],
                               correlation_matrix.iloc[i, j]))

    # Sort by absolute correlation value
    corr_pairs.sort(key=lambda x: abs(x[2]), reverse=True)

    # Print top 5 correlations
    for i, (var1, var2, corr) in enumerate(corr_pairs[:5]):
        print(f"{i+1}. {var1} <-> {var2}: {corr:.4f}")
else:
    print("No numerical features found for correlation analysis.")
```

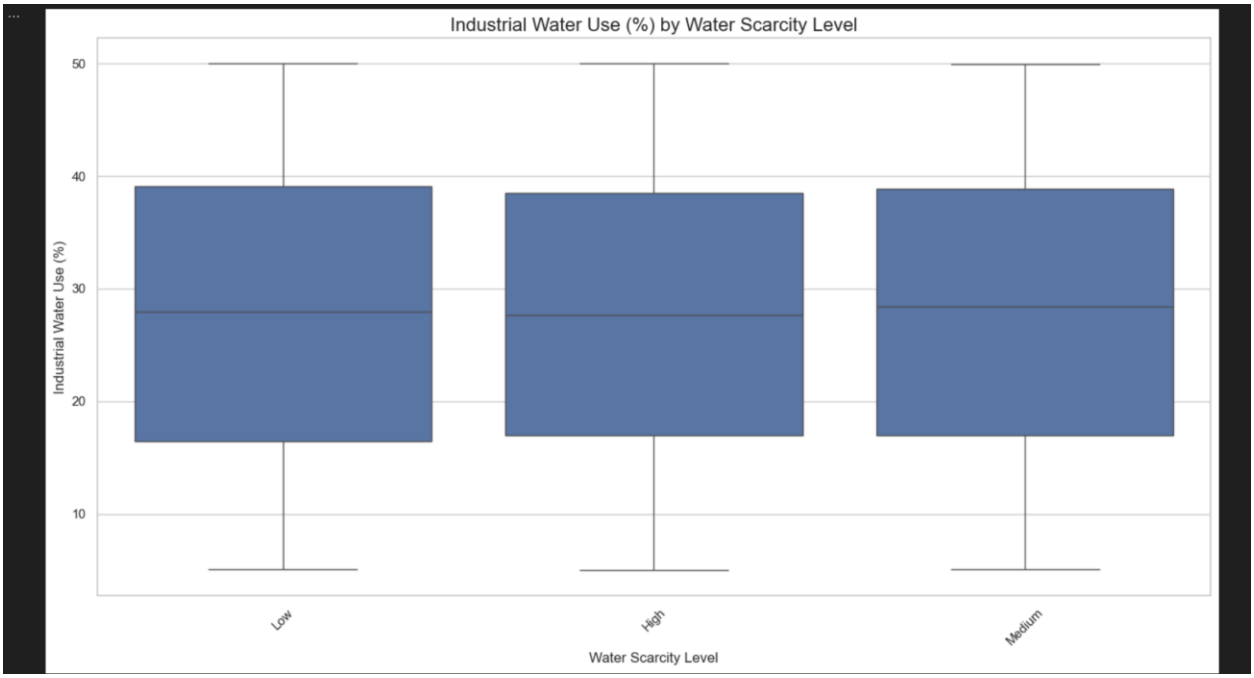
This section focuses on analyzing the relationships between water consumption, agricultural water use, industrial water use, and water scarcity levels, while also examining correlations between numerical features in the dataset. It begins by defining the key column names for water consumption, agricultural water use, industrial water use, and water scarcity levels, which are essential for the analysis. The column names are placeholders and should be adjusted to match the actual columns in the dataset. After this, it prints the available column names from the cleaned

dataset to ensure that the correct ones are selected for analysis.

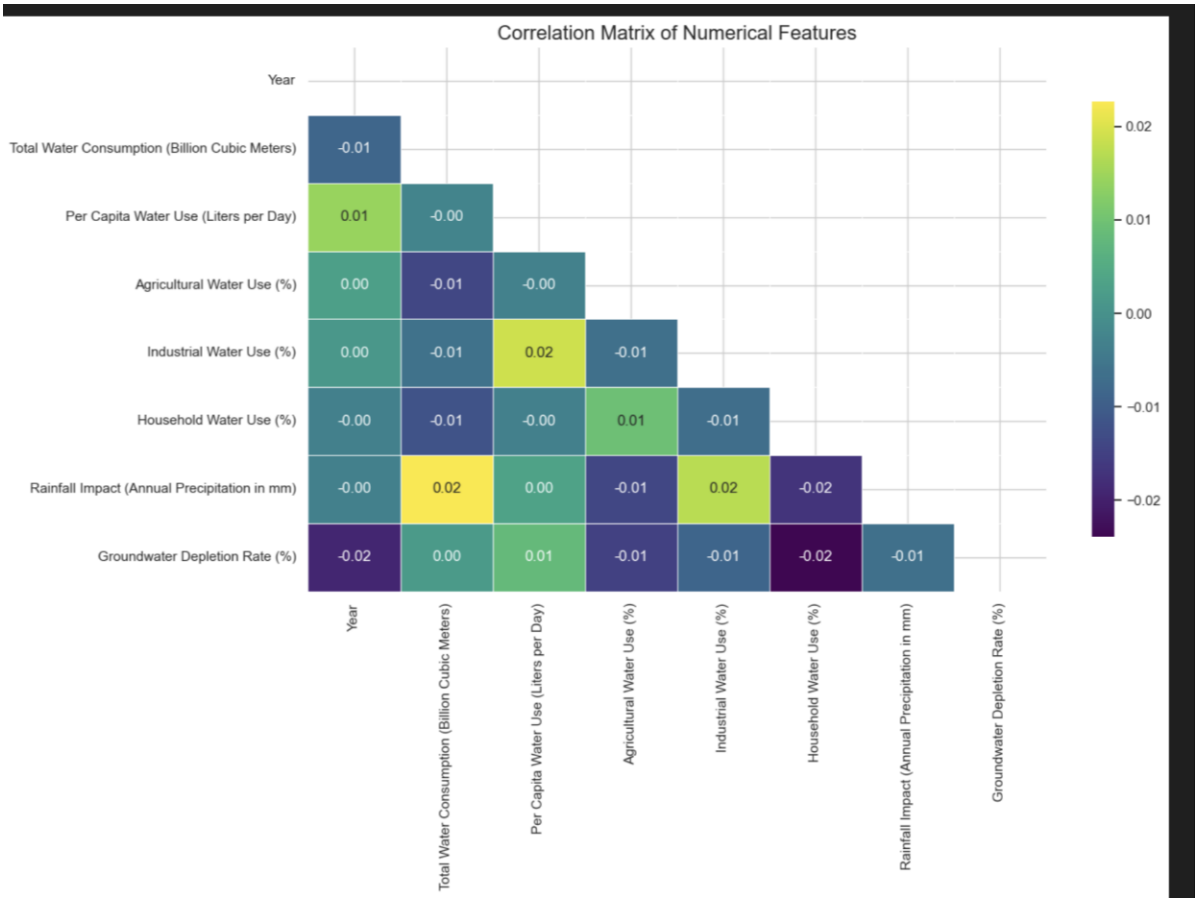


The next step visualizes water consumption by water scarcity level. If both the 'Water Consumption' and 'Water Scarcity Level' columns exist, a box plot is generated to show the distribution of water consumption across different levels of water scarcity. This box plot helps to understand how water consumption varies with increasing levels of scarcity. A similar approach is used to visualize agricultural water use by water scarcity level, with the added condition that rows with zero or negative agricultural water use are excluded from the plot to avoid skewing the data. If the 'Agricultural Water Use' column contains only zero or negative values, a message is displayed

instead of creating the plot.



The relationship between industrial water use and water scarcity is also visualized. If there are more than 10 unique scarcity levels, the code generates a line plot to display the average industrial water use per water scarcity level. If there are fewer than 10 scarcity levels, a box plot is created instead. This analysis helps us understand how industrial water use changes across different water scarcity levels.




```
Top 5 Correlations:
1. Household Water Use (%) <-> Groundwater Depletion Rate (%): −0.0239
2. Total Water Consumption (Billion Cubic Meters) <-> Rainfall Impact (Annual Precipitation in mm): 0.0227
3. Year <-> Groundwater Depletion Rate (%): −0.0193
4. Per Capita Water Use (Liters per Day) <-> Industrial Water Use (%): 0.0189
5. Industrial Water Use (%) <-> Rainfall Impact (Annual Precipitation in mm): 0.0171
```

Additionally, a correlation matrix is calculated to analyze the relationships between numerical features in the dataset. The correlation matrix is displayed as a heatmap using sns.heatmap(), and the upper triangle of the matrix is masked to avoid redundancy. If the number of numerical variables is fewer than 20, the correlation values are annotated on the heatmap. The code also extracts and prints the top 5 strongest correlations between numerical variables to highlight the most significant relationships in the data.

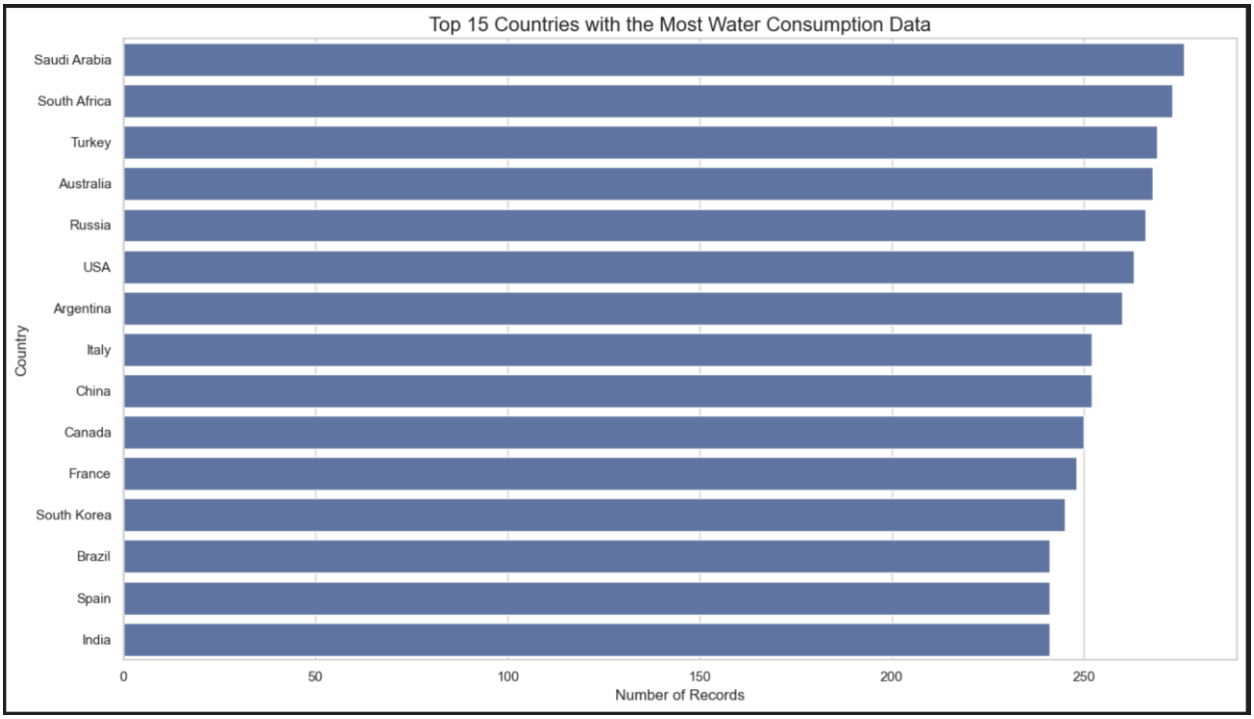
STEP 8 - GEOGRAPHICAL ANALYSIS

STEP 8 - GEOGRAPHICAL ANALYSIS

```
# Define the available column names based on what's in the dataset
country_col = 'Country'  # Adjust if needed
scarcity_level_col = 'Water Scarcity Level'  # Adjust if needed

# Print available columns for reference
print("Available columns:")
print(df_clean.columns.tolist())

# Analyze geographical distribution of water consumption by Country
if country_col in df_clean.columns:
    # Top 15 countries with the most records
    plt.figure(figsize=(14, 8))
    top_countries = df_clean[country_col].value_counts().head(15)

    # Check if there are any countries
    if not top_countries.empty:
        sns.barplot(x=top_countries.values, y=top_countries.index)
        plt.title('Top 15 Countries with the Most Water Consumption Data', fontsize=16)
        plt.xlabel('Number of Records', fontsize=12)
        plt.ylabel('Country', fontsize=12)
        plt.tight_layout()
        plt.show()
    else:
        print("No country data available.")
else:
    print(f"Column '{country_col}' not found.")

# Analyze water consumption distribution by Water Scarcity Level
if scarcity_level_col in df_clean.columns:
    # Top 10 water scarcity levels with the most records
    plt.figure(figsize=(14, 8))
    top_scarcity_levels = df_clean[scarcity_level_col].value_counts().head(10)

    # Check if there are any scarcity levels
    if not top_scarcity_levels.empty:
        sns.barplot(x=top_scarcity_levels.values, y=top_scarcity_levels.index)
        plt.title('Top 10 Water Scarcity Levels with the Most Data', fontsize=16)
        plt.xlabel('Number of Records', fontsize=12)
        plt.ylabel('Water Scarcity Level', fontsize=12)
        plt.tight_layout()
        plt.show()
    else:
        print("No water scarcity level data available.")
else:
    print(f"Column '{scarcity_level_col}' not found.")

# Additional geographical analysis - If any other available column can be used for grouping (e.g., 'Agricultural Water Use')
agriculture_water_col = 'Agricultural Water Use (%)'  # Adjust if needed
if agriculture_water_col in df_clean.columns:
    # Top 10 agricultural water usage levels with the most records
    plt.figure(figsize=(14, 8))
    top_agriculture_water = df_clean[agriculture_water_col].value_counts().head(10)
```

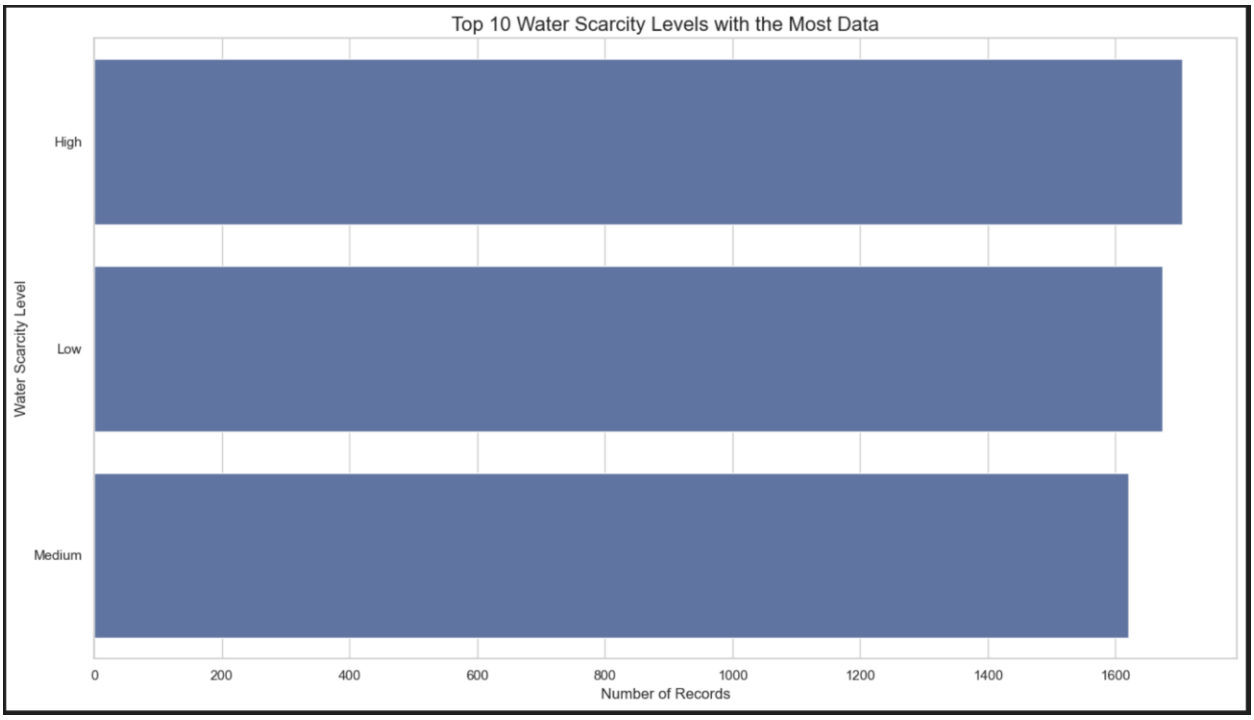
```
    # Check if there are any agricultural water use levels
    if not top_agriculture_water.empty:
        sns.barplot(x=top_agriculture_water.values, y=top_agriculture_water.index.astype(str))
        plt.title('Top 10 Agricultural Water Use Levels with the Most Data', fontsize=16)
        plt.xlabel('Number of Records', fontsize=12)
        plt.ylabel('Agricultural Water Use (%)', fontsize=12)
        plt.tight_layout()
        plt.show()
    else:
        print("No agricultural water use data available.")
else:
    print(f"Column '{agriculture_water_col}' not found.")
```

This analysis starts by defining key column names, specifically 'Country' and 'Water Scarcity Level', which are assumed to be part of the dataset. These columns can be adjusted depending on the actual names in the dataset. After this, it prints the available columns in the cleaned DataFrame to confirm that the necessary columns are present.

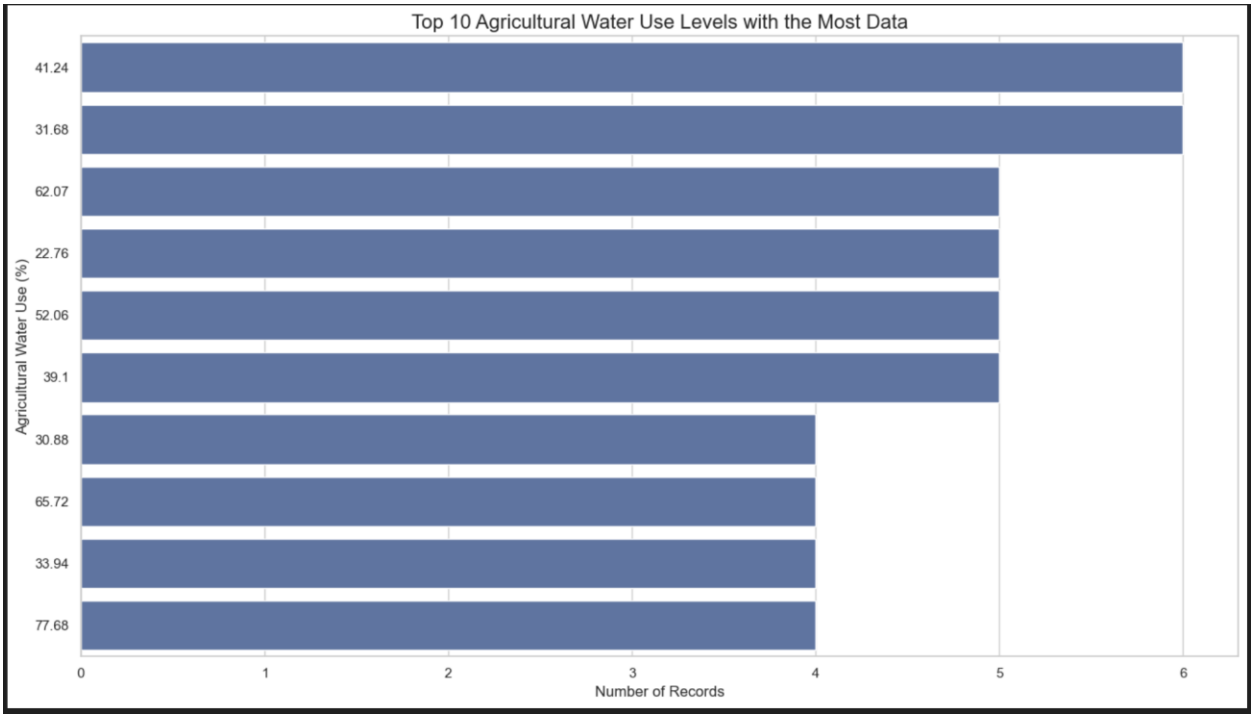


The first part of the geographical analysis examines the distribution of water consumption by country. If the 'Country' column exists, the code counts the occurrences of each unique country in the dataset and visualizes the top 15 countries with the most water consumption data using a bar plot. This helps to understand which countries have the most data available for analysis. If the

'Country' column is missing or no data is available, a message is displayed to notify the user.



Next, the code analyzes the distribution of water consumption based on water scarcity levels. It checks if the 'Water Scarcity Level' column exists and visualizes the top 10 scarcity levels with the most data. A bar plot is generated to show how different scarcity levels are represented in the dataset. If the necessary column is missing or there is no data, an appropriate message is printed.



The final part of the geographical analysis focuses on agricultural water use. The code checks for the 'Agricultural Water Use (%)' column and creates a bar plot to display the top 10 levels of

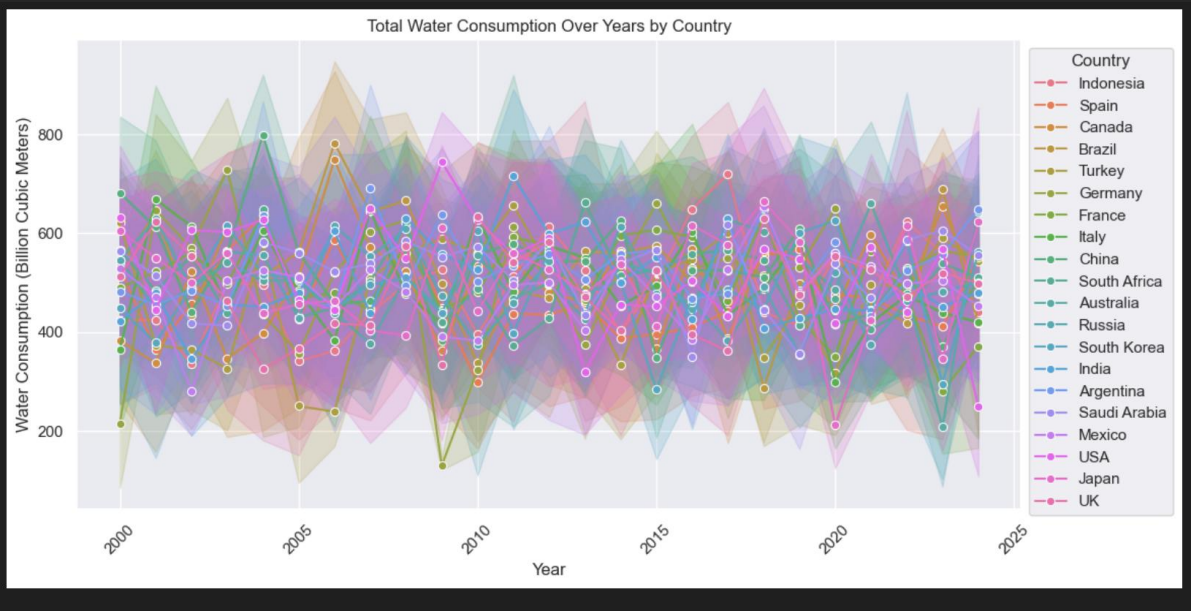
agricultural water use with the most records. This helps to visualize the distribution of agricultural water use in the dataset. If no agricultural water use data is available, a message is printed.
In summary, this code performs a geographical analysis by visualizing the distribution of water consumption by country, water scarcity levels, and agricultural water use. It includes checks to ensure that the necessary columns are available before creating the visualizations, which are presented as bar plots to give a clear understanding of the distribution of these features across different countries and scarcity levels. If any data is missing, the code will inform the user accordingly.

STEP 09 - TOTAL WATER CONSUMPTION OVER YEARS BY COUNTRY

STEP 20 - TOTAL WATER CONSUMPTION OVER YEARS BY COUNTRY

```
# Convert Year to integer for consistency
df['Year'] = df['Year'].astype(int)

# Set Seaborn style
sns.set_theme(style="darkgrid")

# 1. Total Water Consumption Over Time for Each Country
plt.figure(figsize=(12, 6))
sns.lineplot(data=df, x='Year', y='Water Consumption', hue='Country', marker='o')
plt.title('Total Water Consumption Over Years by Country')
plt.xlabel('Year')
plt.ylabel('Water Consumption (Billion Cubic Meters)')
plt.legend(title='Country', bbox_to_anchor=(1, 1))
plt.xticks(rotation=45)
plt.show()
```

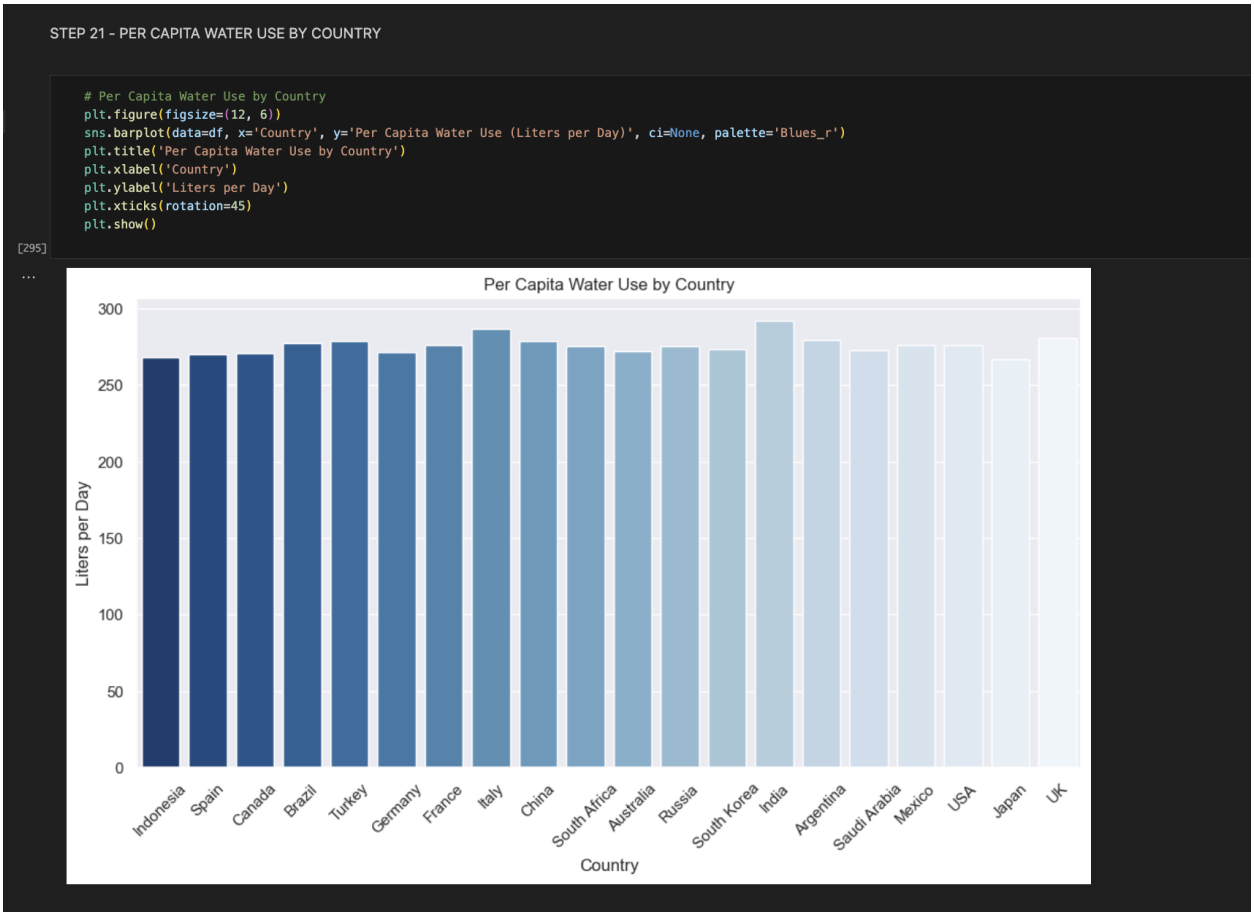
[294]



We have visualized the total water consumption over time for each country using a line plot. First, it ensures the 'Year' column is treated as an integer type for consistent plotting by converting it with df['Year'].astype(int). Next, the Seaborn theme is set to "darkgrid" with

sns.set_theme(style="darkgrid"), which improves the readability of the plot by adding gridlines to a dark background. The code then generates a line plot with sns.lineplot(), where the x-axis represents the year, and the y-axis represents total water consumption in billions of cubic meters. Each country is represented by a separate line, differentiated by color using the hue='Country argument, with markers placed at each data point for clarity. The plot includes a title, axis labels, and a legend outside the plot to identify the countries. The x-axis labels are rotated for better readability, and the plot is displayed using plt.show(). This code provides a clear visualization of how water consumption trends have evolved over time for different countries.

STEP 10 - PER CAPITA WATER USE BY COUNTRY



Then we have visualized per capita water use by country using a bar plot. It begins by creating a figure with a size of 12 inches by 6 inches to ensure the plot is large enough to display all countries clearly. The sns.barplot() function is used to create the bar plot, with the x-axis representing the countries and the y-axis showing per capita water use in liters per day for each country. The confidence intervals are disabled using ci=None, and the bars are colored using a reversed blue

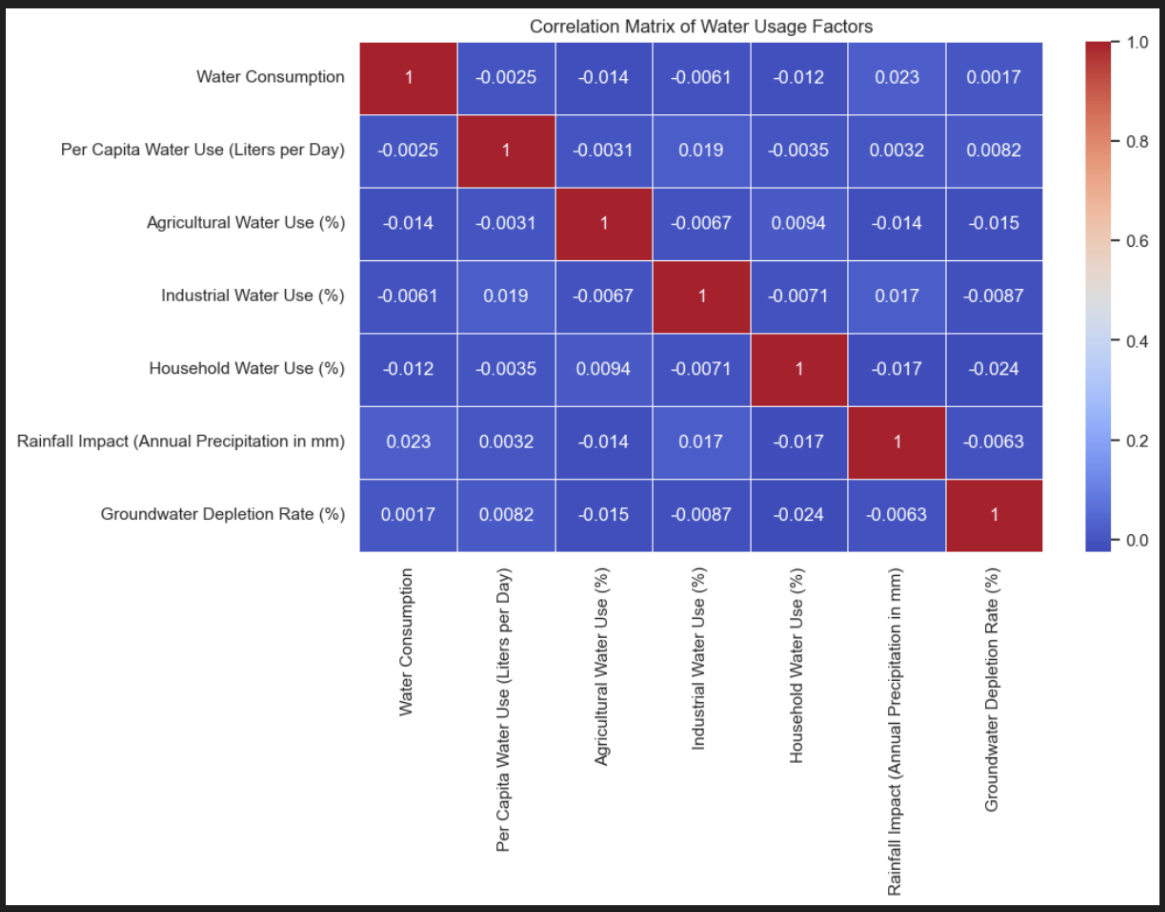
color palette (Blues_r). The plot includes a title, "Per Capita Water Use by Country", and labels for both axes, with the x-axis labels rotated by 45 degrees for better readability. Finally, the plot is displayed using plt.show(). This code provides a clear comparison of per capita water use across different countries.

STEP 11 - CORRELATION MATRIX OF WATER USAGE FACTORS

STEP 22 - CORRELATION MATRIX OF WATER USAGE FACTORS

```
# Correlation Heatmap
plt.figure(figsize=(10, 6))

# Drop non-numeric columns for correlation calculation
corr_matrix = df.drop(columns=['Country', 'Year', 'Water Scarcity Level']).corr()

# Plot the correlation matrix
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Matrix of Water Usage Factors')
plt.show()
```

[296]



This section of the code generates a correlation heatmap to visualize the relationships between numerical features in the dataset, excluding non-numeric columns such as 'Country', 'Year', and 'Water Scarcity Level'. The plot is created with a size of 10 inches by 6 inches to ensure clarity. The correlation matrix is calculated by dropping non-numeric columns and applying the .corr() method, which computes the pairwise correlation between the remaining numeric columns. The heatmap is then generated using sns.heatmap(), with the correlation values annotated in each cell for easy reference. The 'coolwarm' color palette is used to represent the strength and direction of the

correlations, where blue indicates negative correlations and red indicates positive correlations. A title is added to the heatmap, and the plot is displayed using plt.show(). This code provides a clear visualization of the relationships between water usage factors, allowing for easy identification of features that are positively or negatively correlated.

STEP 12 - Water Scarcity Analysis by Country:

STEP 16 - WATER SCARCITY ANALYSIS BY COUNTRY

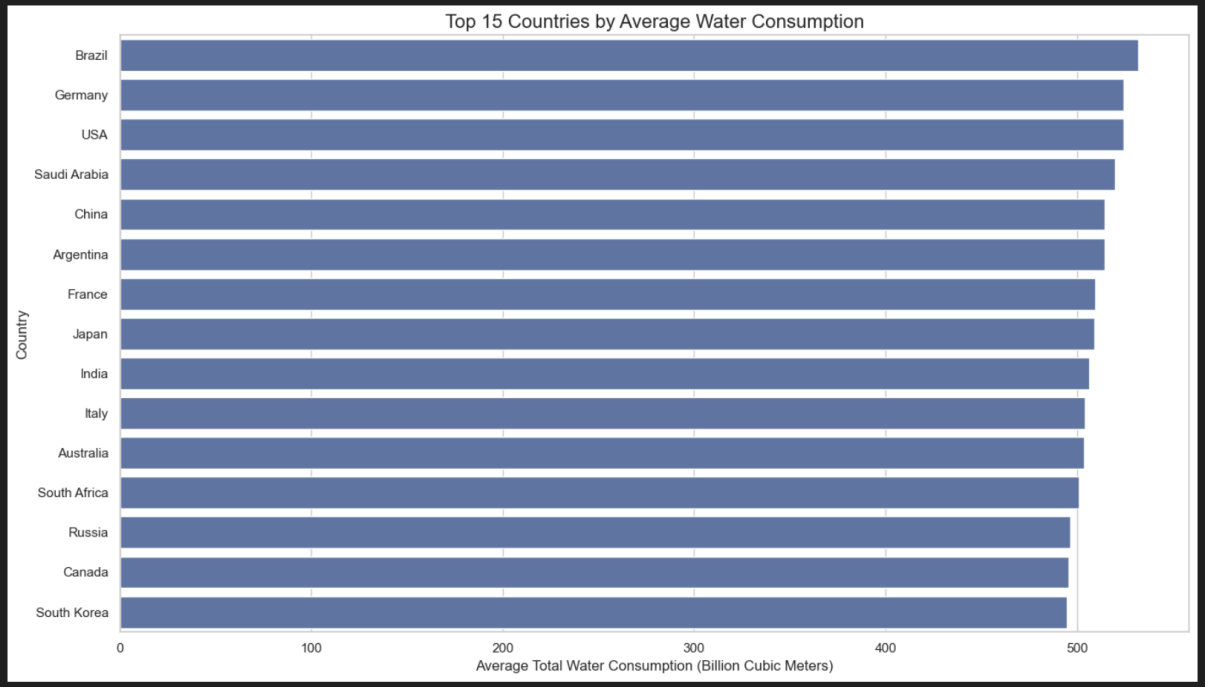
```
# Analyze water consumption and scarcity by country
try:
    # Group by country and calculate average water consumption and scarcity level
    country_stats = df.groupby('Country').agg({
        'Total Water Consumption (Billion Cubic Meters)': 'mean',
        'Per Capita Water Use (Liters per Day)': 'mean',
        'Agricultural Water Use (%)': 'mean',
        'Industrial Water Use (%)': 'mean',
        'Household Water Use (%)': 'mean',
        'Water Scarcity Level': lambda x: x.mode()[0] if len(x.mode()) > 0 else None
    }).reset_index()

    # Sort by total water consumption
    country_stats = country_stats.sort_values('Total Water Consumption (Billion Cubic Meters)', ascending=False)

    # Display the top 10 countries by water consumption
    print("Top 10 Countries by Average Water Consumption:")
    display(country_stats.head(10))

    # Create a bar chart of top 15 countries by water consumption
    plt.figure(figsize=(14, 8))
    top_15 = country_stats.head(15)
    sns.barplot(data=top_15, x='Total Water Consumption (Billion Cubic Meters)', y='Country')
    plt.title('Top 15 Countries by Average Water Consumption', fontsize=16)
    plt.xlabel('Average Total Water Consumption (Billion Cubic Meters)', fontsize=12)
    plt.ylabel('Country', fontsize=12)
    plt.grid(True, axis='x')
    plt.tight_layout()
    plt.show()

    # Create a bar chart of per capita water use for top 15 countries
    plt.figure(figsize=(14, 8))
    top_15_per_capita = country_stats.sort_values('Per Capita Water Use (Liters per Day)', ascending=False).head(15)
    sns.barplot(data=top_15_per_capita, x='Per Capita Water Use (Liters per Day)', y='Country')
    plt.title('Top 15 Countries by Per Capita Water Use', fontsize=16)
    plt.xlabel('Average Per Capita Water Use (Liters per Day)', fontsize=12)
    plt.ylabel('Country', fontsize=12)
    plt.grid(True, axis='x')
    plt.tight_layout()
    plt.show()

    # Count countries by water scarcity level
    scarcity_counts = df['Water Scarcity Level'].value_counts()

    plt.figure(figsize=(10, 6))
    sns.countplot(data=df, x='Water Scarcity Level', order=scarcity_counts.index)
    plt.title('Distribution of Water Scarcity Levels', fontsize=16)
    plt.xlabel('Water Scarcity Level', fontsize=12)
    plt.ylabel('Count', fontsize=12)
    plt.grid(True, axis='y')
    plt.tight_layout()
    plt.show()
```

```python
    # Analyze water usage patterns (agricultural, industrial, household) by scarcity level
    usage_by_scarcity = df.groupby('Water Scarcity Level').agg({
        'Agricultural Water Use (%)': 'mean',
        'Industrial Water Use (%)': 'mean',
        'Household Water Use (%)': 'mean'
    }).reset_index()

    # Melt the DataFrame for easier plotting
    usage_melted = pd.melt(
        usage_by_scarcity,
        id_vars=['Water Scarcity Level'],
        value_vars=['Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household Water Use (%)'],
        var_name='Usage Type',
        value_name='Percentage'
    )

    plt.figure(figsize=(12, 8))
    sns.barplot(data=usage_melted, x='Water Scarcity Level', y='Percentage', hue='Usage Type')
    plt.title('Water Usage Patterns by Scarcity Level', fontsize=16)
    plt.xlabel('Water Scarcity Level', fontsize=12)
    plt.ylabel('Average Percentage (%)', fontsize=12)
    plt.legend(title='Usage Type')
    plt.grid(True, axis='y')
    plt.tight_layout()
    plt.show()

except Exception as e:
    print(f"Error in water scarcity analysis: {e}")
```

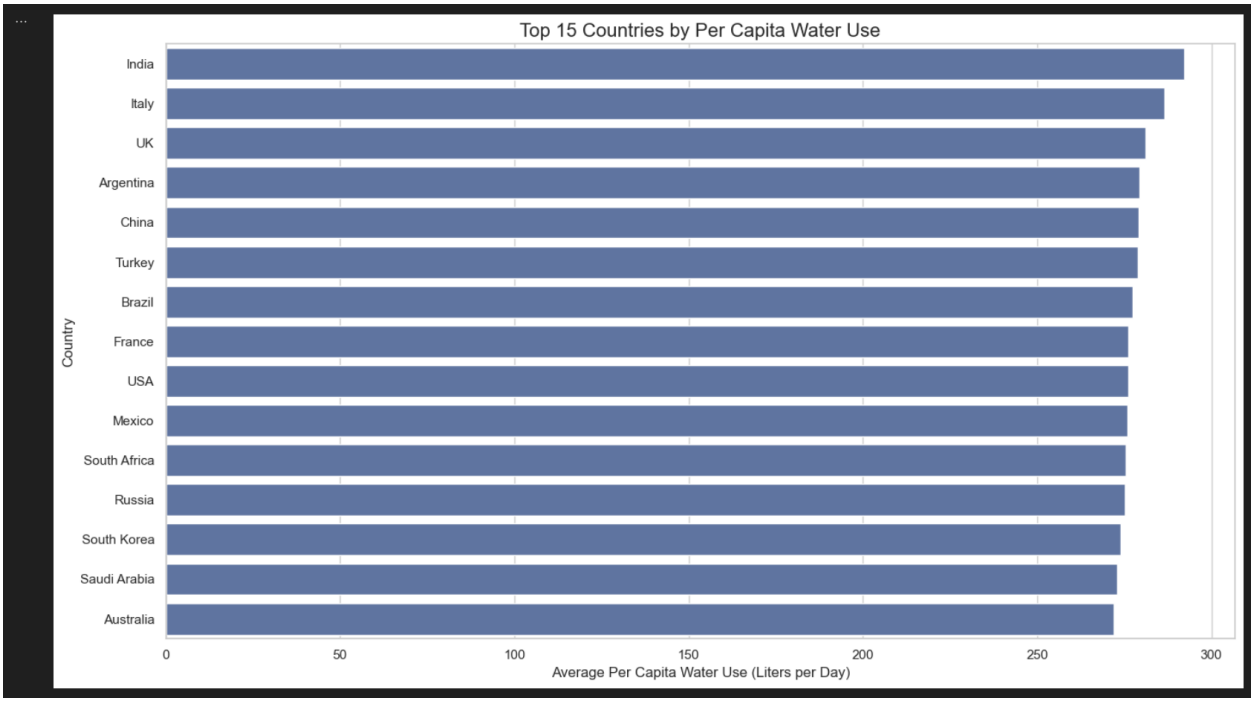
A table was generated with the average water consumption for the top 10 countries. The top countries in terms of water consumption include Brazil, Germany, the USA, Saudi Arabia, and China. These countries also show the average per capita water use and the distribution of water use between agricultural, industrial, and household sectors.

```
Top 10 Countries by Average Water Consumption:
```

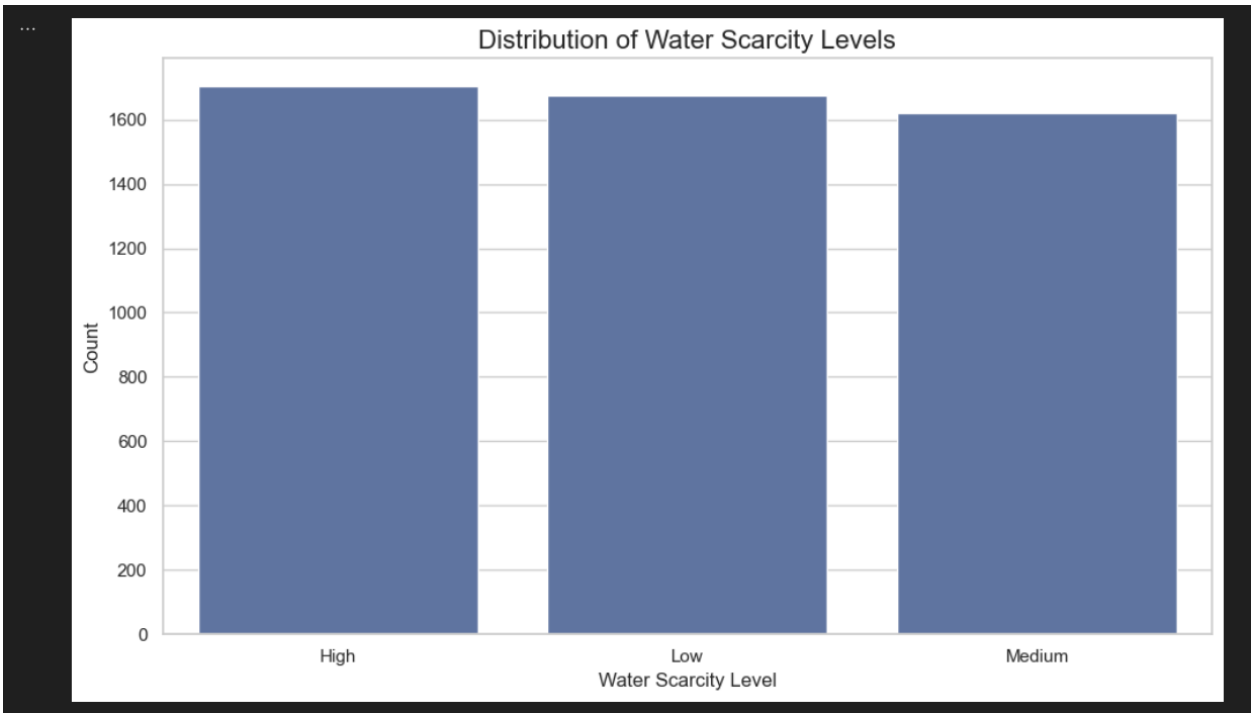
	Country	Total Water Consumption (Billion Cubic Meters)	Per Capita Water Use (Liters per Day)	Agricultural Water Use (%)	Industrial Water Use (%)	Household Water Use (%)	Water Scarcity Level
2	Brazil	531.916224	277.476888	50.573776	27.145311	25.088838	High
6	Germany	524.555105	271.328439	48.170970	28.006329	24.037426	High
19	USA	524.354183	276.076198	50.295361	27.359696	25.088441	High
13	Saudi Arabia	519.973841	272.928442	49.491812	27.419638	25.441884	Low
4	China	514.664921	279.125079	51.614048	27.895714	25.507341	Medium
0	Argentina	514.639500	279.375846	51.147577	28.268115	24.835615	Low
5	France	509.390565	276.103185	48.621048	26.582258	25.183387	High
10	Japan	508.940853	266.658009	49.227536	28.822464	24.911659	High
7	India	506.253859	292.165768	49.944149	27.484481	24.446432	Medium
9	Italy	503.962857	286.550317	49.606230	28.183929	24.891905	High



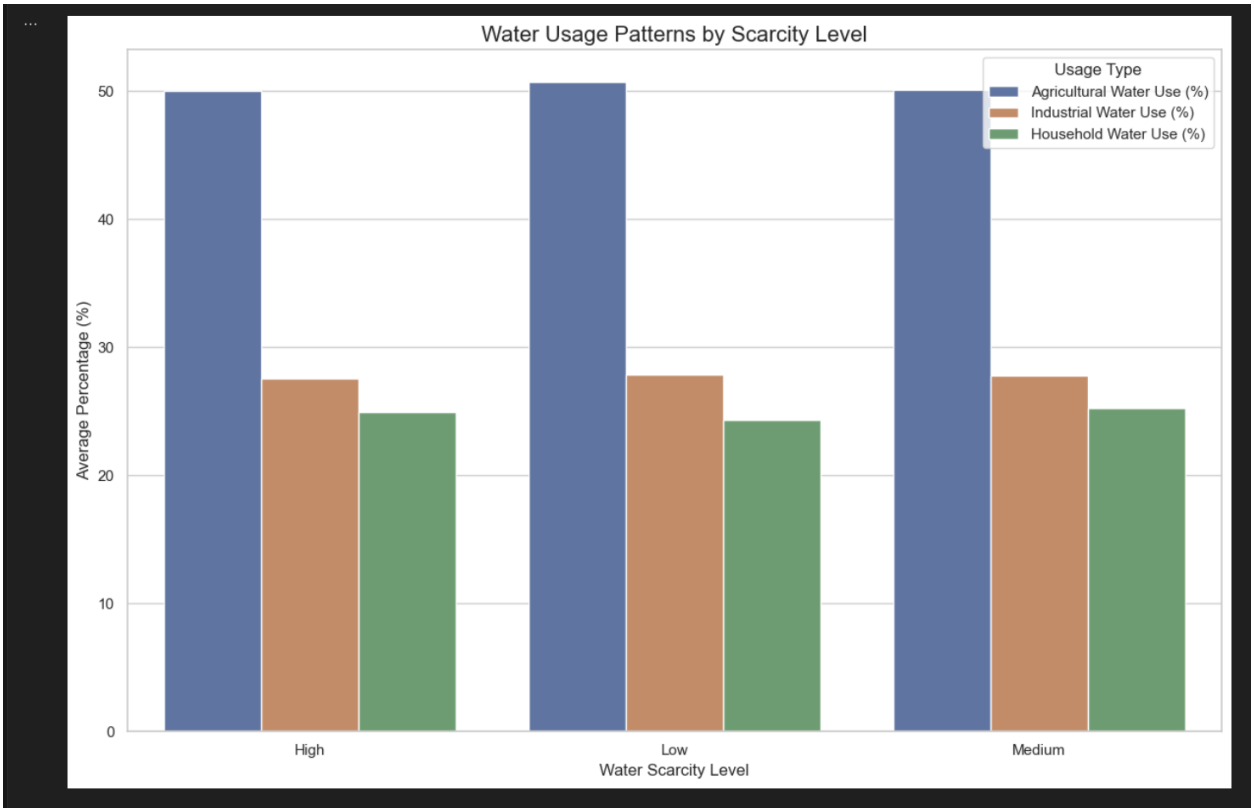
Top 15 Countries by Average Water Consumption: This bar chart visually compares the total water consumption of the top 15 countries, showing Brazil, Germany, and the USA at the top.



Top 15 Countries by Per Capita Water Use: This bar chart compares the per capita water use across the top 15 countries, highlighting India, Italy, and the UK as the top countries with the highest per capita water use.



Water Scarcity Level Distribution: A count plot was created to show the distribution of water scarcity levels across the dataset, revealing that the data is nearly evenly split between "High," "Medium," and "Low" scarcity levels.



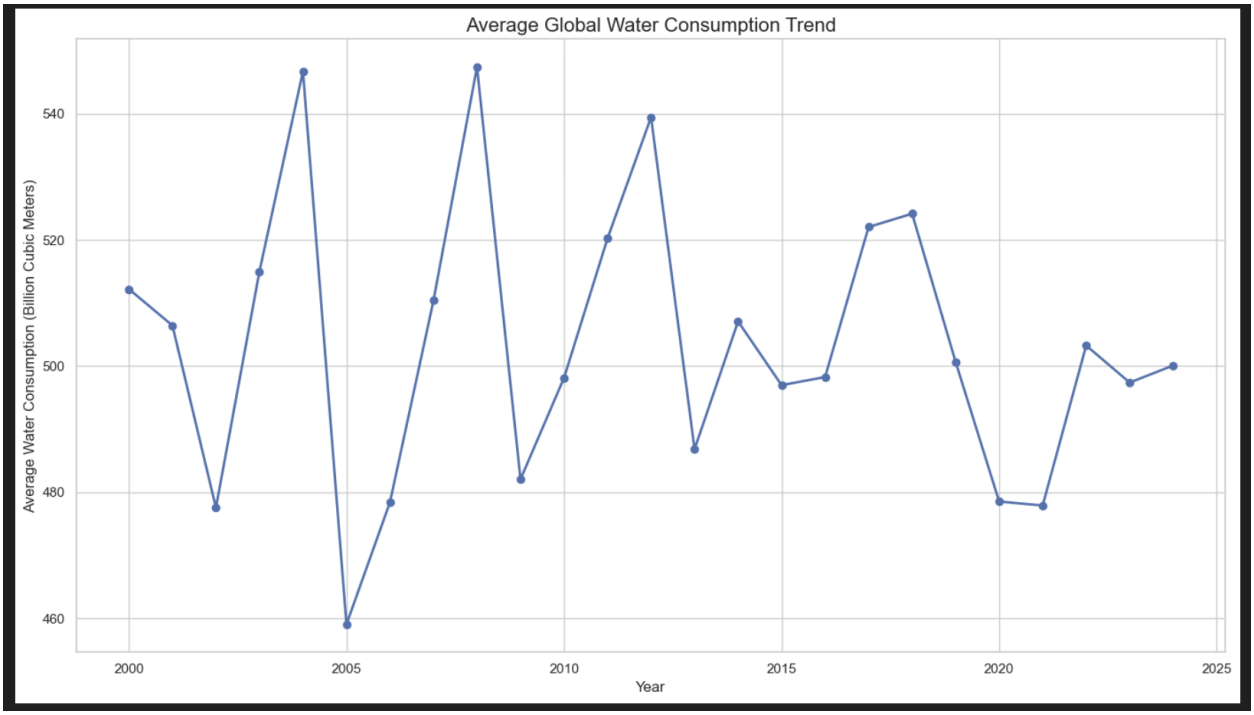
Water Usage Patterns by Scarcity Level: A bar chart that compares the average percentage of water usage by sector (agriculture, industry, and household) for each water scarcity level. For instance, countries with "High" water scarcity show a larger share of water used for agriculture.

STEP 13 - Trend Analysis Over Time:

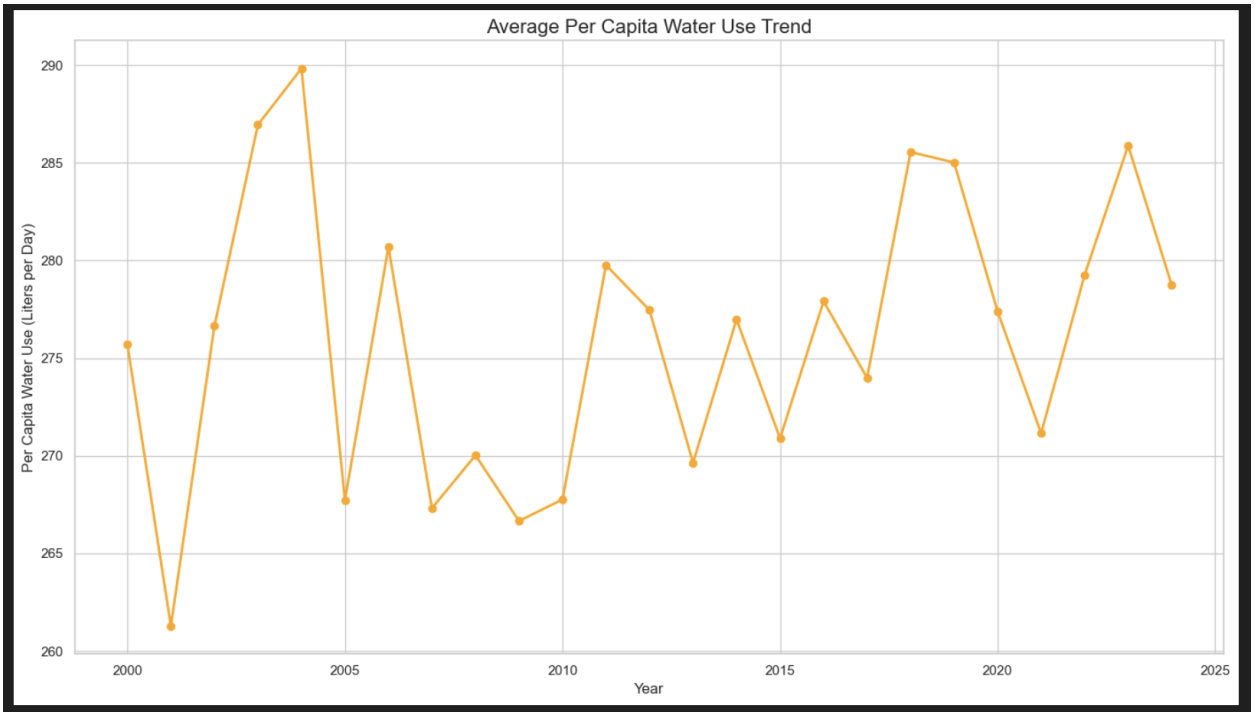
STEP 17 - TREND ANALYSIS OVER TIME

```
# Analyze trends in water consumption over time
try:
    # Group by year and calculate average metrics
    yearly_trends = df.groupby('Year').agg({
        'Total Water Consumption (Billion Cubic Meters)': 'mean',
        'Per Capita Water Use (Liters per Day)': 'mean',
        'Agricultural Water Use (%)': 'mean',
        'Industrial Water Use (%)': 'mean',
        'Household Water Use (%)': 'mean',
        'Groundwater Depletion Rate (%)': 'mean'
    }).reset_index()

    # Plot trends over time
    plt.figure(figsize=(14, 8))
    plt.plot(yearly_trends['Year'], yearly_trends['Total Water Consumption (Billion Cubic Meters)'], 'o-', linewidth=2)
    plt.title('Average Global Water Consumption Trend', fontsize=16)
    plt.xlabel('Year', fontsize=12)
    plt.ylabel('Average Water Consumption (Billion Cubic Meters)', fontsize=12)
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # Plot per capita water use over time
    plt.figure(figsize=(14, 8))
    plt.plot(yearly_trends['Year'], yearly_trends['Per Capita Water Use (Liters per Day)'], 'o-', linewidth=2, color='orange')
    plt.title('Average Per Capita Water Use Trend', fontsize=16)
    plt.xlabel('Year', fontsize=12)
    plt.ylabel('Per Capita Water Use (Liters per Day)', fontsize=12)
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # Plot trends for water usage categories
    plt.figure(figsize=(14, 8))
    plt.plot(yearly_trends['Year'], yearly_trends['Agricultural Water Use (%)'], 'o-', linewidth=2, label='Agricultural')
    plt.plot(yearly_trends['Year'], yearly_trends['Industrial Water Use (%)'], 's-', linewidth=2, label='Industrial')
    plt.plot(yearly_trends['Year'], yearly_trends['Household Water Use (%)'], '^-', linewidth=2, label='Household')
    plt.title('Water Usage Patterns Over Time', fontsize=16)
    plt.xlabel('Year', fontsize=12)
    plt.ylabel('Percentage (%)', fontsize=12)
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # Plot groundwater depletion rate over time
    plt.figure(figsize=(14, 8))
    plt.plot(yearly_trends['Year'], yearly_trends['Groundwater Depletion Rate (%)'], 'o-', linewidth=2, color='red')
    plt.title('Average Groundwater Depletion Rate Trend', fontsize=16)
    plt.xlabel('Year', fontsize=12)
    plt.ylabel('Groundwater Depletion Rate (%)', fontsize=12)
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

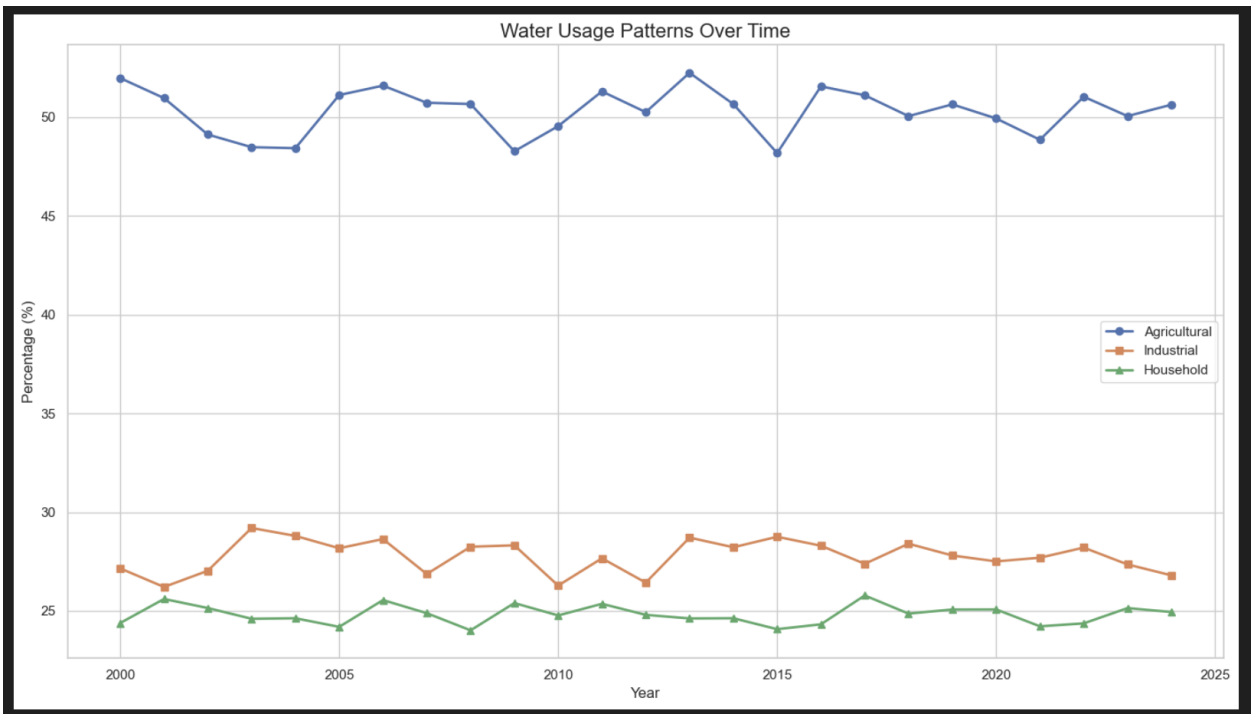
```python
    # Calculate correlation between year and key metrics
    correlations = pd.DataFrame({
        'Metric': [
            'Total Water Consumption',
            'Per Capita Water Use',
            'Agricultural Water Use',
            'Industrial Water Use',
            'Household Water Use',
            'Groundwater Depletion Rate'
        ],
        'Correlation with Year': [
            np.corrcoef(yearly_trends['Year'], yearly_trends['Total Water Consumption (Billion Cubic Meters)'])[0, 1],
            np.corrcoef(yearly_trends['Year'], yearly_trends['Per Capita Water Use (Liters per Day)'])[0, 1],
            np.corrcoef(yearly_trends['Year'], yearly_trends['Agricultural Water Use (%)'])[0, 1],
            np.corrcoef(yearly_trends['Year'], yearly_trends['Industrial Water Use (%)'])[0, 1],
            np.corrcoef(yearly_trends['Year'], yearly_trends['Household Water Use (%)'])[0, 1],
            np.corrcoef(yearly_trends['Year'], yearly_trends['Groundwater Depletion Rate (%)'])[0, 1]
        ]
    })

    print("Correlation of Metrics with Time (Year):")
    display(correlations.sort_values('Correlation with Year', key=abs, ascending=False))

except Exception as e:
    print(f"Error in trend analysis: {e}")
```



This plot shows the average global water consumption over the years. The data points follow a somewhat fluctuating pattern, with spikes and dips observed at different time periods (e.g., in the 2000s, around 2015, and towards the end of the plot).The chart reflects how total water consumption has varied on a global scale from 2000 to 2025.

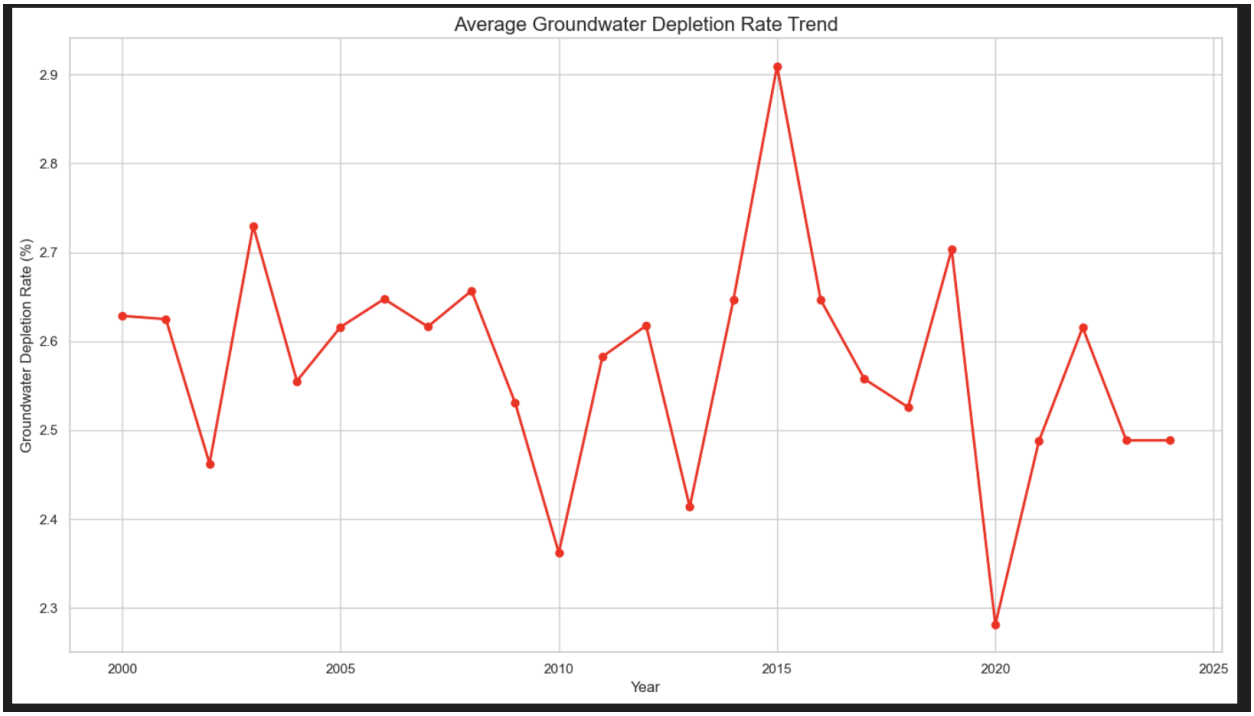


This plot shows the trend for average per capita water use over the years. The trend line fluctuates, with peaks around 2005 and 2015, and a general downward movement in recent years.
It suggests changes in per capita water usage patterns across the years.



This chart compares the changes in water usage across three main sectors: Agricultural, Industrial, and Household over the years.

Agricultural water use remains the highest and relatively stable, whereas household and industrial water use have experienced slight fluctuations over time.



This plot shows the average groundwater depletion rate over time, which follows a fluctuating pattern. It shows a significant peak around 2015 followed by a decline.

The trend reflects varying levels of groundwater depletion across the years.

Correlation Analysis:

```
...  Correlation of Metrics with Time (Year):
```

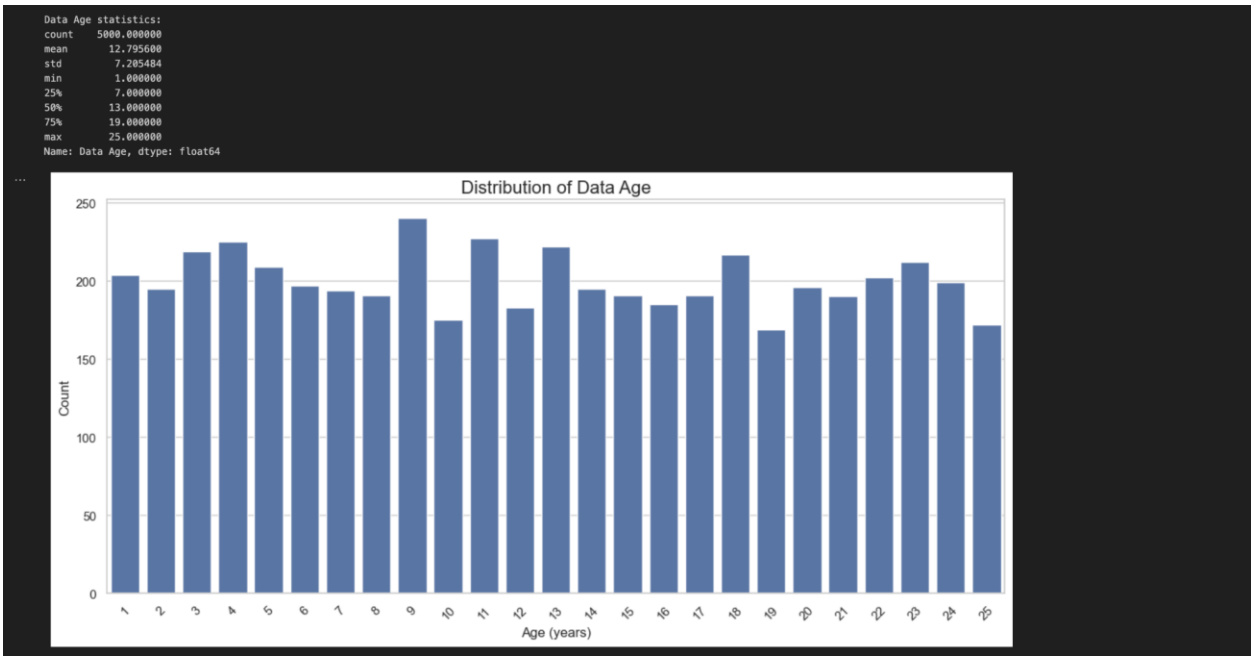
...

	Metric	Correlation with Year
1	Per Capita Water Use	0.253245
5	Groundwater Depletion Rate	-0.220750
0	Total Water Consumption	-0.104596
4	Household Water Use	-0.037805
2	Agricultural Water Use	0.019804
3	Industrial Water Use	0.008331

These outputs give insights into how water consumption trends, per capita usage, sectoral water use, and groundwater depletion have changed over time. The weak correlations suggest that while some trends are noticeable, they may not exhibit strong, consistent changes across the years.

STEP 14 - FEATURE ENGINEERING

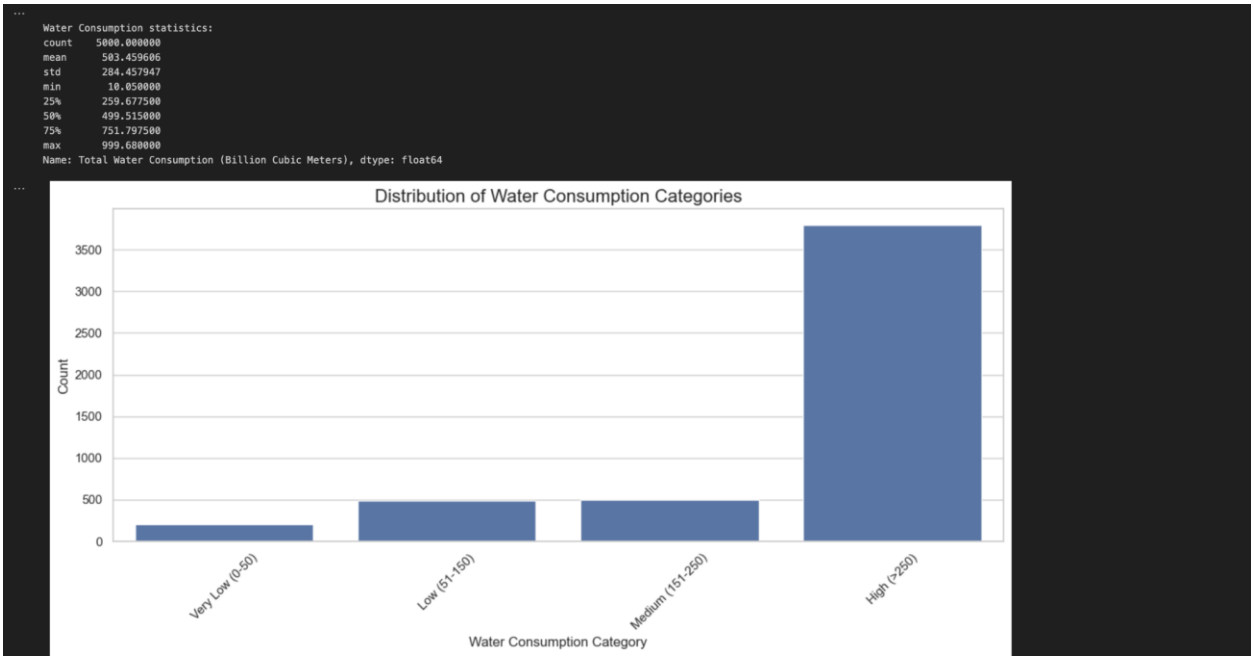
1. Dataset with New Features:



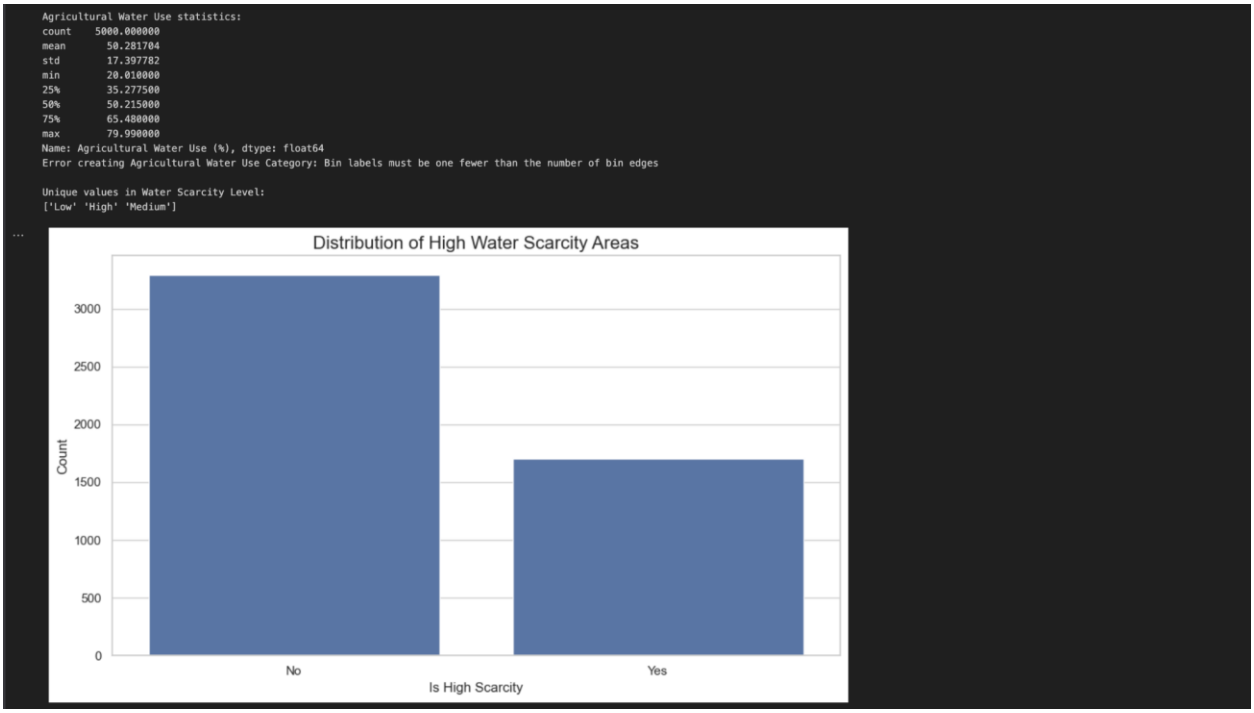
- The dataset has been enhanced with new features, which are crucial for the analysis:
  - Data Age: This column calculates the age of the data based on the year. The age statistics show a mean of 12.8 years, with a range between 1 year and 25 years.
  - Water Consumption Category: This categorizes water consumption into four bins based on the total water consumption:
    - Very Low (0-50),
    - Low (51-150),
    - Medium (151-250),
    - High (>250). The water consumption statistics indicate a mean of 503.46 billion cubic meters and a range from 10.05 to 999.68 billion cubic meters.
  - Is High Scarcity: A binary feature indicating whether the area has high water scarcity (1 for 'Yes' and 0 for 'No') based on the Water Scarcity Level column.

2. Distribution Plots:

- Data Age: The distribution of data age is fairly uniform across the years from 1 to 25, with slight peaks at certain ages (e.g., age 9).



- Water Consumption Categories: The majority of the data falls under the "High (>250)" category, indicating that most countries in the dataset have high water consumption. The other categories, like "Low" and "Very Low," have fewer records.



- High Water Scarcity: A count plot shows that the majority of the data falls under the "No" category, meaning there are more countries with low or medium water scarcity compared

to high water scarcity.

Dataset with new features:

	Country	Year	Total Water Consumption (Billion Cubic Meters)	Per Capita Water Use (Liters per Day)	Water Scarcity Level	Agricultural Water Use (%)	Industrial Water Use (%)	Household Water Use (%)	Rainfall Impact (Annual Precipitation in mm)	Groundwater Depletion Rate (%)	Data Age	Water Consumption Category	Is High Scarcity
0	Indonesia	2022	895.15	489.73	Low	20.78	13.75	34.99	1075.28	3.10	3	High (>250)	0
1	Indonesia	2024	502.89	311.95	High	48.51	8.44	32.88	2630.69	1.78	1	High (>250)	1
2	Spain	2000	843.39	440.09	Medium	25.16	31.70	34.62	2860.62	4.13	25	High (>250)	0
3	Canada	2021	803.34	478.98	High	45.74	6.13	18.99	1725.50	0.61	4	High (>250)	1
4	Brazil	2022	416.40	353.91	High	26.58	7.95	31.11	988.44	0.80	3	High (>250)	1

Number of new features created: 3
New features: ['Data Age', 'Water Consumption Category', 'Is High Scarcity']

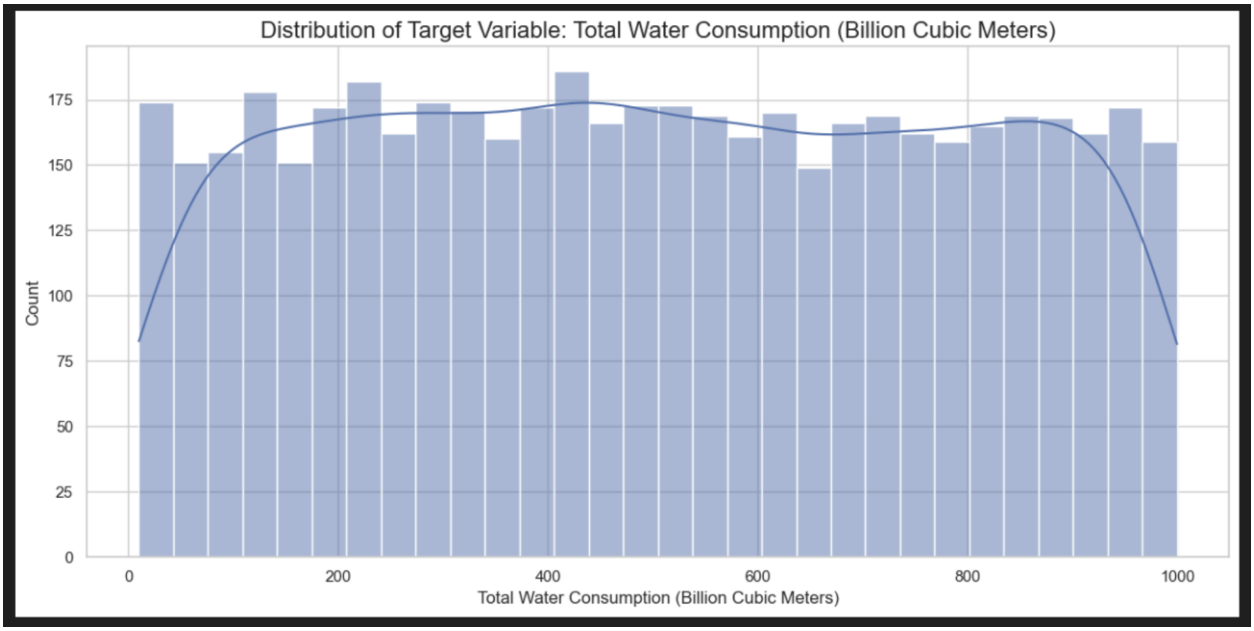
New Features Created:

- Data Age: Shows the age of the data in years.
- Water Consumption Category: Categorizes water consumption into four ranges.
- Is High Scarcity: Marks whether the area has high water scarcity.

These new features will help in modeling and analysis, particularly when investigating patterns in water consumption and scarcity.

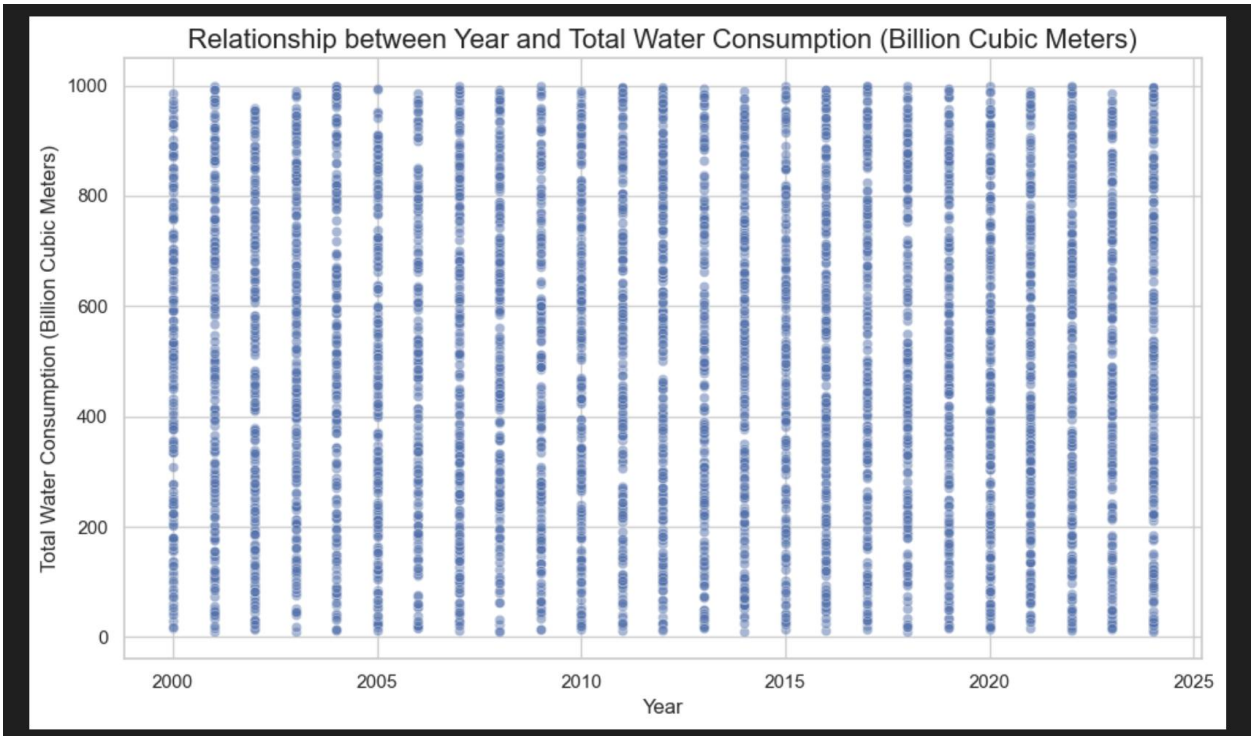
STEP 15 - PREPARE DATA FOR MODELING

```
Available columns:
['Country', 'Year', 'Total Water Consumption (Billion Cubic Meters)', 'Per Capita Water Use (Liters per Day)', 'Water Scarcity Level', 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household
Available numeric features: ['Year', 'Agricultural Water Use (%)', 'Industrial Water Use (%)']
Available categorical features: ['Water Scarcity Level']
Target variable 'Total Water Consumption (Billion Cubic Meters)' found. Proceeding with modeling.

Feature set shape: (5000, 4)
Target set shape: (5000,)

Missing values in features:
Year                        0
Agricultural Water Use (%)  0
Industrial Water Use (%)    0
Water Scarcity Level        0
dtype: int64

Training set shape: (4000, 4)
Testing set shape: (1000, 4)

Shape of preprocessed training data: (4000, 5)

Total features after preprocessing: 5
First 10 feature names: ['Year', 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Water Scarcity Level_Low', 'Water Scarcity Level_Medium']
```

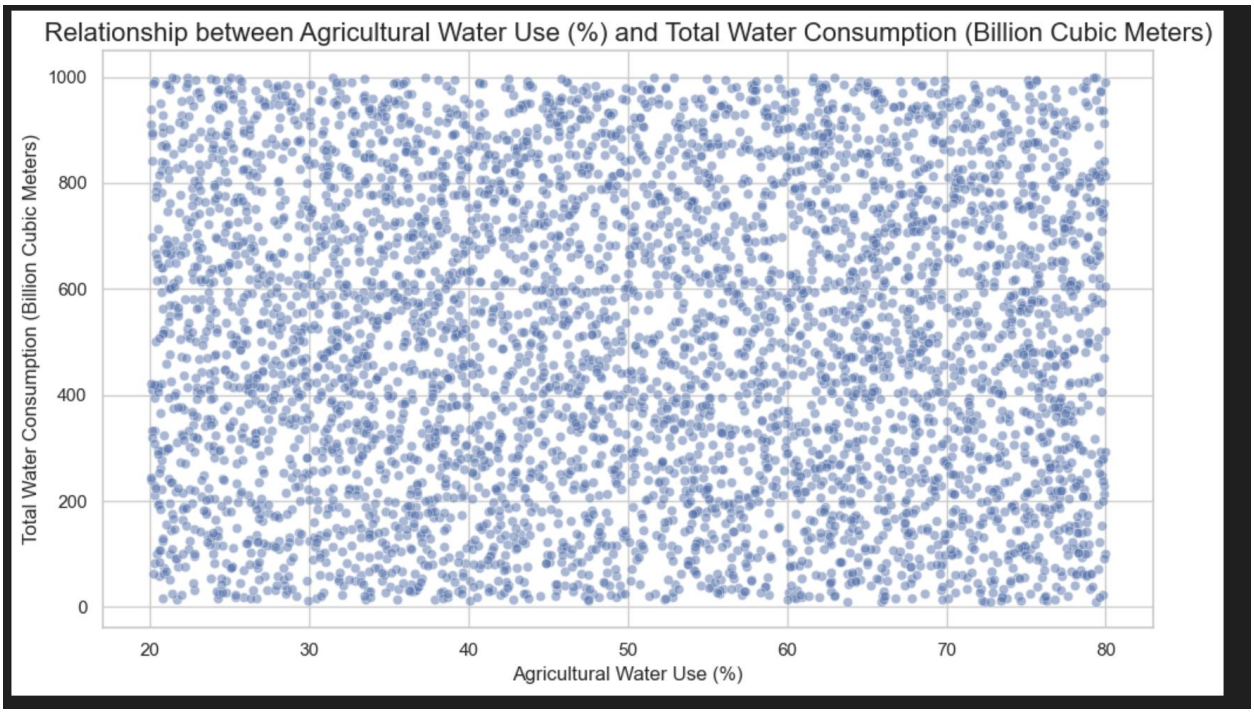


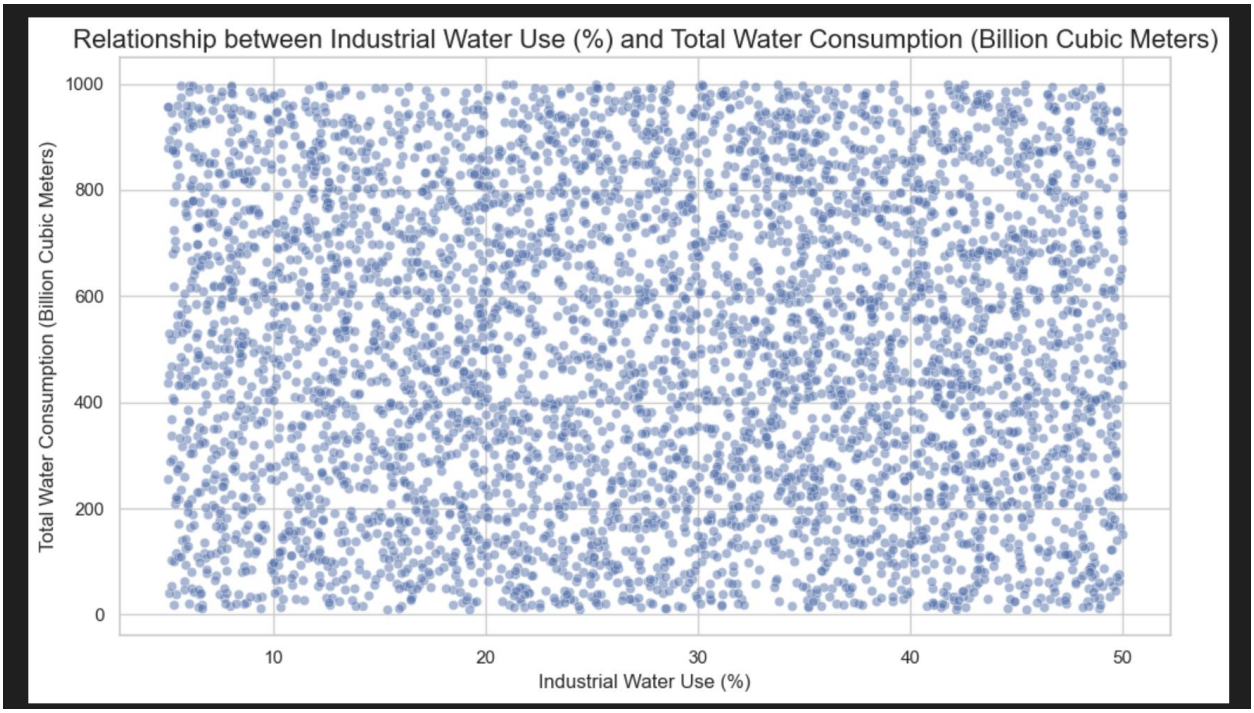
This focuses on preparing the data for modeling by cleaning and transforming the dataset. First, a list of the available columns in the dataset is provided, which includes 'Country', 'Year', 'Total Water Consumption (Billion Cubic Meters)', 'Per Capita Water Use (Liters per Day)', 'Water Scarcity Level', 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household Water Use (%)', 'Rainfall Impact (Annual Precipitation in mm)', and 'Groundwater Depletion Rate (%)'. These columns are essential for the analysis and modeling process.

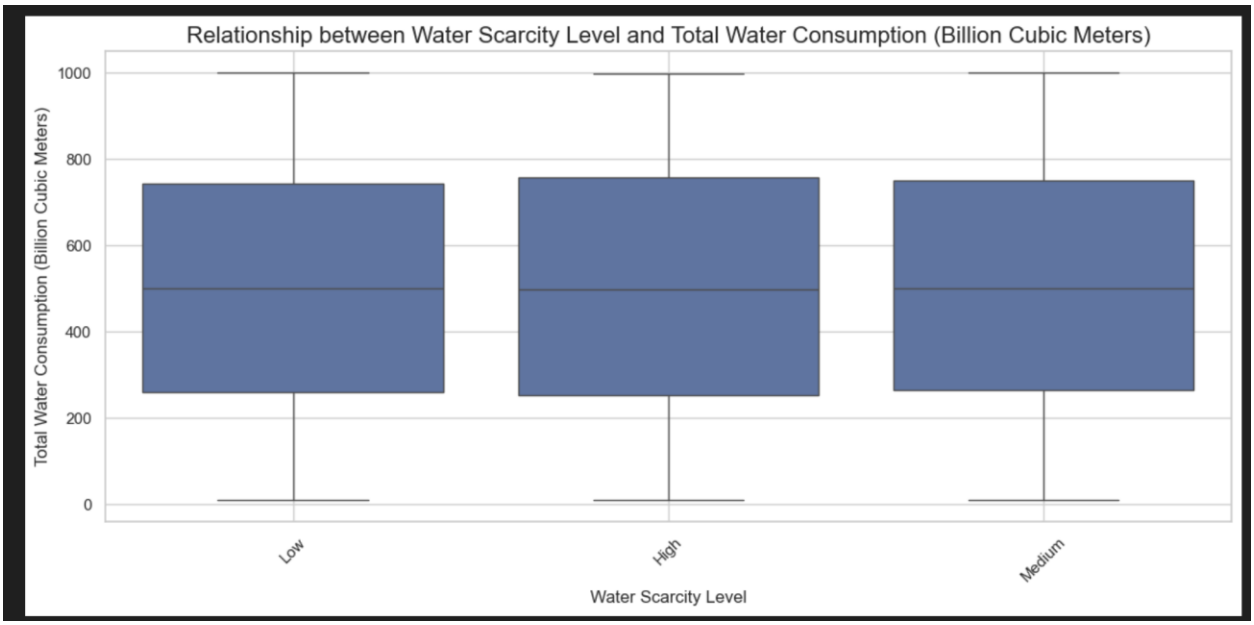
Next, the numeric and categorical features are identified. The numeric features selected for modeling are 'Year', 'Agricultural Water Use (%)', and 'Industrial Water Use (%)', while the 'Water Scarcity Level' is selected as the categorical feature. The target variable, 'Total Water Consumption (Billion Cubic Meters)', was confirmed to be available for modeling and is the focus of the predictive analysis.




```
Available columns:
['Country', 'Year', 'Total Water Consumption (Billion Cubic Meters)', 'Per Capita Water Use (Liters per Day)', 'Water Scarcity Level', 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household
Available numeric features: ['Year', 'Agricultural Water Use (%)', 'Industrial Water Use (%)']
Available categorical features: ['Water Scarcity Level']
Target variable 'Total Water Consumption (Billion Cubic Meters)' found. Proceeding with modeling.

Feature set shape: (5000, 4)
Target set shape: (5000,)
Training set shape: (4000, 4)
Testing set shape: (1000, 4)

Training Linear Regression model...
Model training completed.
Making predictions on test set...

Linear Regression Model Performance:
Mean Squared Error: 78934.60
Root Mean Squared Error: 280.95
Mean Absolute Error: 243.58
R² Score: -0.0004
```


Relationship between Agricultural Water Use (%) and Total Water Consumption (Billion Cubic Meters)
Total Water Consumption (Billion Cubic Meters)
1000
800
600
400
200
0
20
30
40
50
60
70
80
Agricultural Water Use (%)


Relationship between Industrial Water Use (%) and Total Water Consumption (Billion Cubic Meters)
Total Water Consumption (Billion Cubic Meters)
1000
800
600
400
200
0
10
20
30
40
50
Industrial Water Use (%)

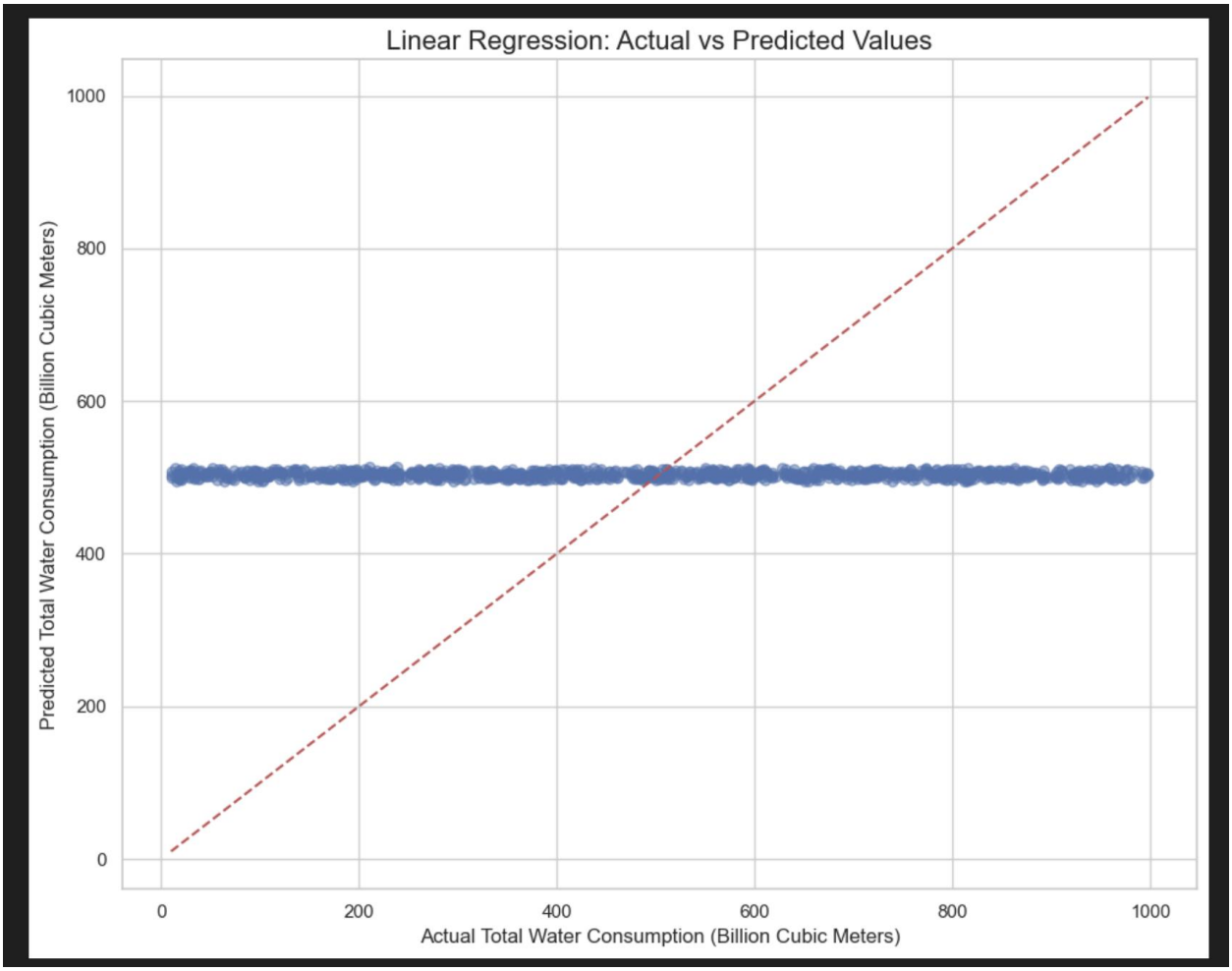


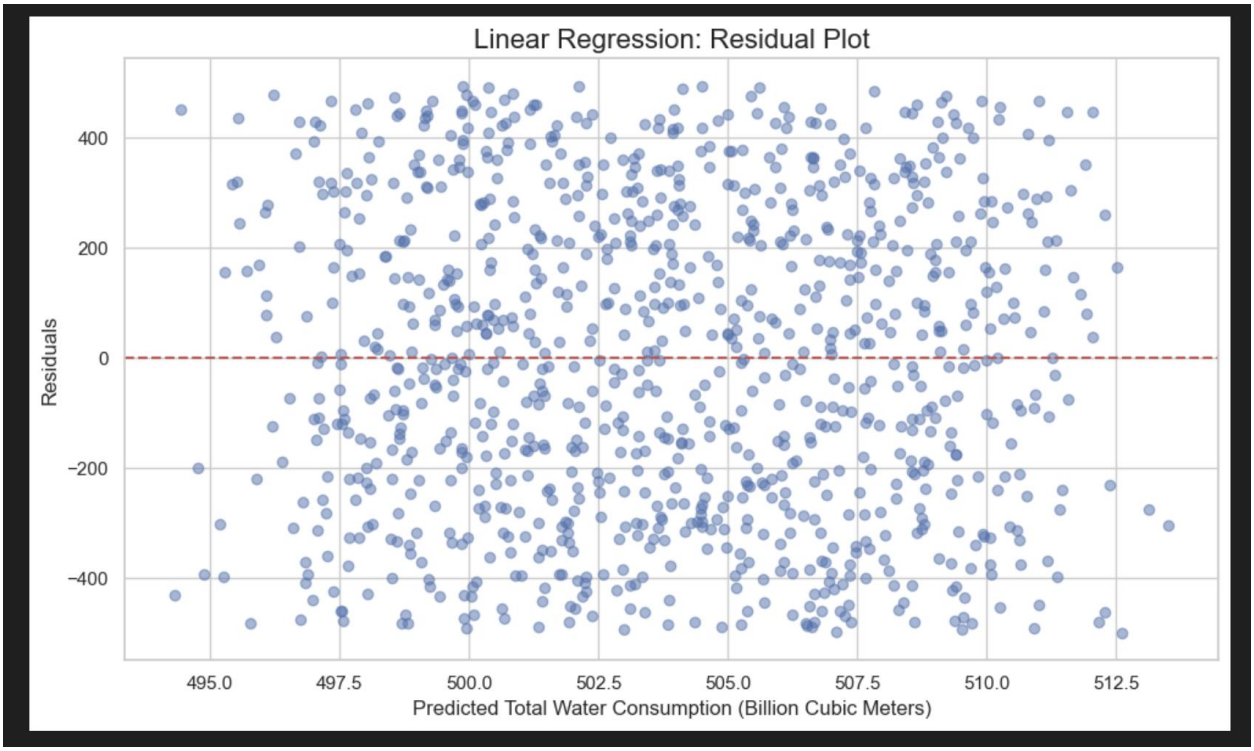
In the preprocessing stage, the dataset is checked for missing values, and it is confirmed that there are no missing values in the selected features or the target variable. The data is split into a training set containing 4000 samples and a testing set with 1000 samples, with 80% of the data used for training. The preprocessing steps for numeric features involve imputation with the median (though no missing values were found) and standardization using StandardScaler. For categorical features, imputation with the most frequent category is applied (again, no missing values), followed by one-hot encoding.

After preprocessing, the feature set includes the original numeric features: 'Year', 'Agricultural Water Use (%)', and 'Industrial Water Use (%)'. One-hot encoding is applied to the 'Water Scarcity Level' feature, creating two additional columns: 'Water Scarcity Level_Low' and 'Water Scarcity Level_Medium', with the 'High' category serving as the baseline and being excluded.

Finally, several visualizations are created to help understand the data. A distribution plot of the target variable, 'Total Water Consumption', is generated to show how the data is distributed across its range of values. Scatter plots are also created to visualize the relationships between the target variable and each numeric feature, including 'Year', 'Agricultural Water Use', and 'Industrial Water Use', providing insights into potential linear correlations. These visualizations are important for understanding the underlying patterns in the dataset before moving on to the modeling phase.

STEP 16 - Model Building - Linear Regression:





A Linear Regression model is trained to predict Total Water Consumption (Billion Cubic Meters) using features such as Year, Agricultural Water Use (%), Industrial Water Use (%), and Water Scarcity Level. This model aims to understand how these factors contribute to water consumption patterns across different countries.

The model was trained on a training set consisting of 4000 samples and evaluated using a testing set of 1000 samples. Preprocessing steps were applied to handle missing data and standardize the numeric features, such as scaling, to ensure that all inputs were in a suitable format for the model.

The evaluation metrics for the model are as follows:

- Mean Squared Error (MSE): 78934.60, which represents the average squared differences between the actual and predicted values. A lower MSE indicates a better model.

- Root Mean Squared Error (RMSE): 280.95, which is the square root of the MSE and provides an error measure in the same units as the target variable (billion cubic meters).

- Mean Absolute Error (MAE): 243.58, measuring the average magnitude of the errors in the predictions, without considering the direction of the errors.

- R^2 Score: -0.0004, indicating how well the model explains the variance in the target variable. A negative R^2 score suggests that the model performs worse than a simple horizontal line (representing the mean of the target variable), meaning it fails to capture the data's underlying patterns.

Visualizations:

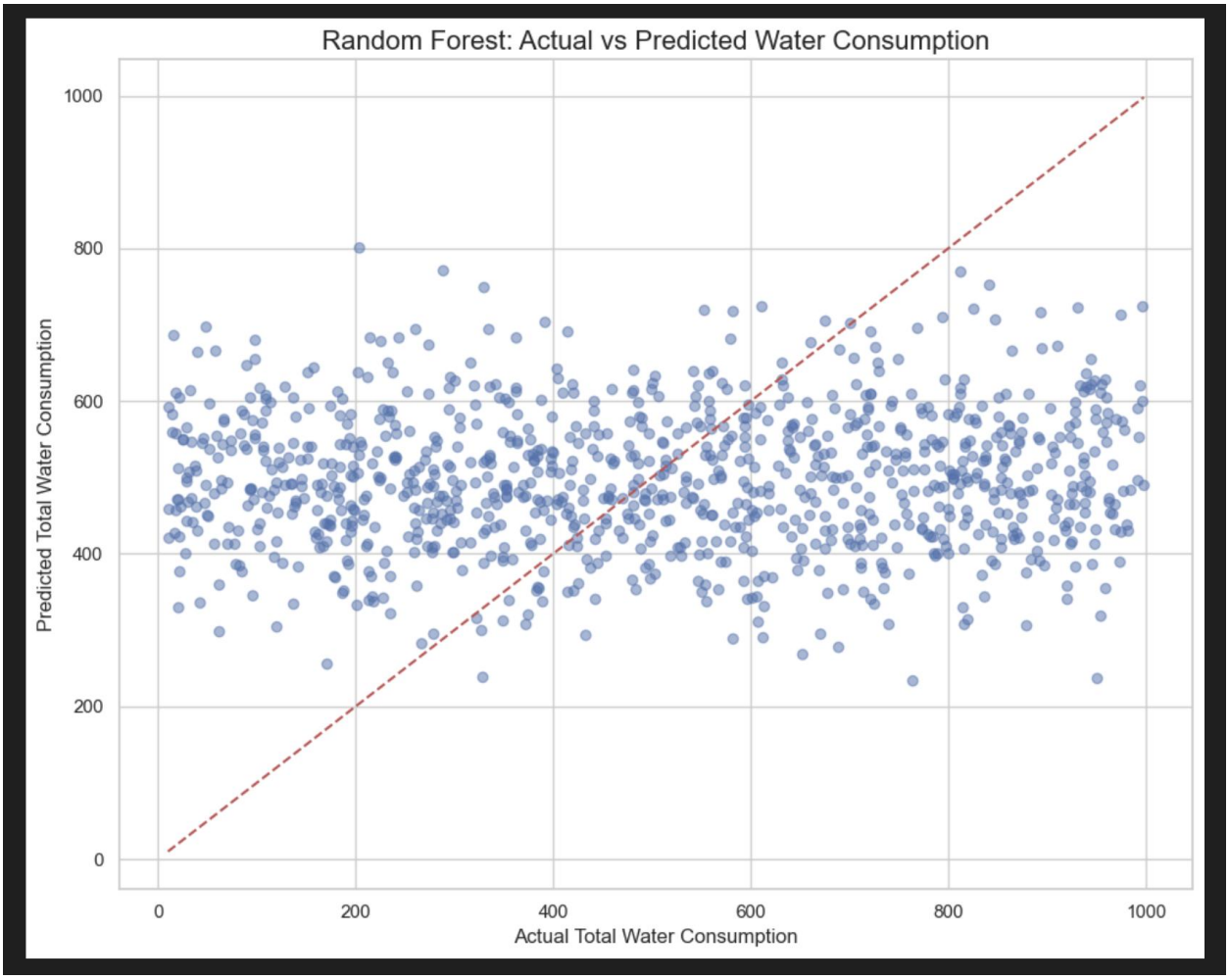
- Actual vs Predicted Values: A scatter plot reveals a significant discrepancy between the predicted and actual values, as indicated by the red dashed line, which represents perfect predictions. This highlights the model's limited accuracy.

- Residual Plot: This plot shows the residuals (the differences between actual and predicted values). Ideally, the residuals should be randomly scattered around zero, indicating a well-fit model. However, the spread of residuals in this case suggests that the model's predictive capability could be improved.

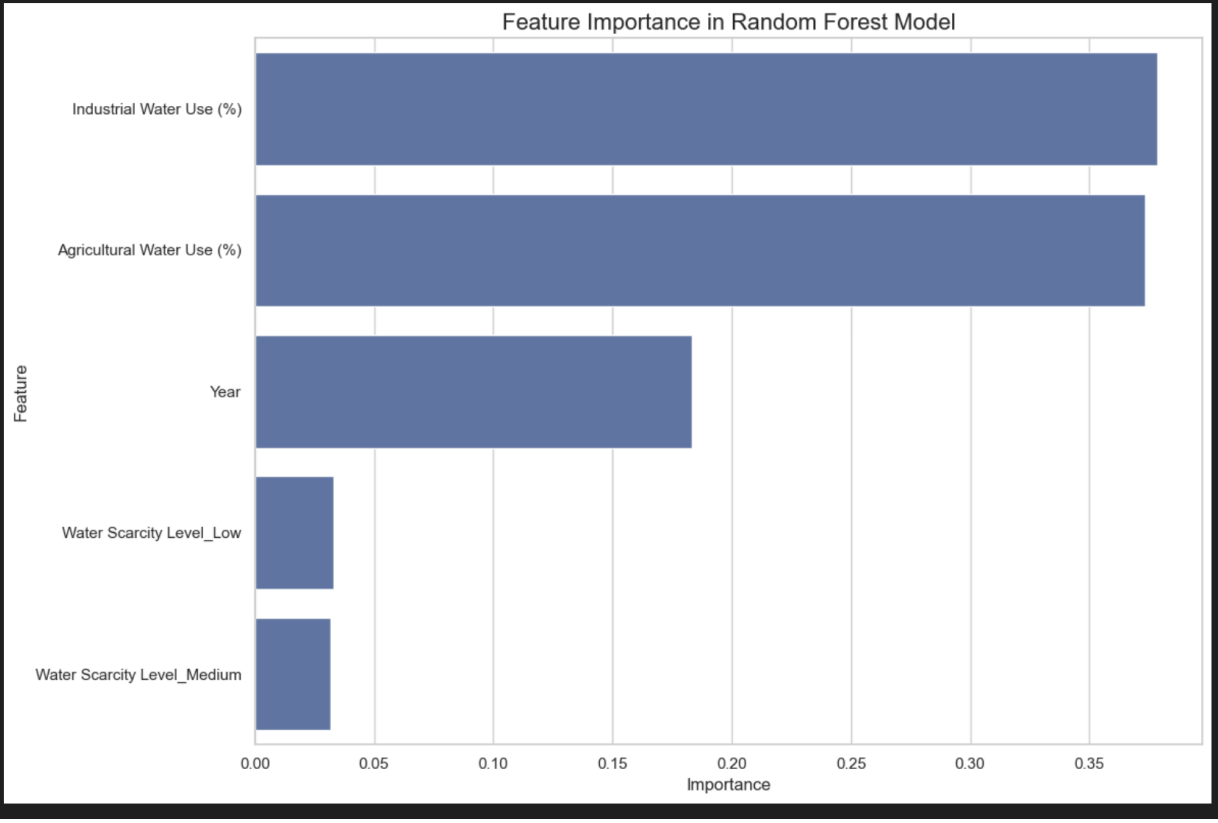
The negative R^2 score indicates that the linear regression model is not effectively capturing the variance in the target variable. This may imply that the current set of features is insufficient or that the relationship between the features and the target variable is not linear.

STEP 17 - MODEL BUILDING - RANDOM FOREST

```
Training Random Forest model...
Model training completed.
Making predictions on test set...

Random Forest Model Performance:
Mean Squared Error: 84664.01
Root Mean Squared Error: 290.97
Mean Absolute Error: 247.58
R² Score: -0.0731
```


Random Forest: Actual vs Predicted Water Consumption
Predicted Total Water Consumption
Actual Total Water Consumption
0
200
400
600
800
1000

```
Feature Importances:
                       Feature  Importance
2     Industrial Water Use (%)    0.378392
1   Agricultural Water Use (%)    0.373441
0                         Year    0.183264
3     Water Scarcity Level_Low    0.033099
4  Water Scarcity Level_Medium    0.031803
```

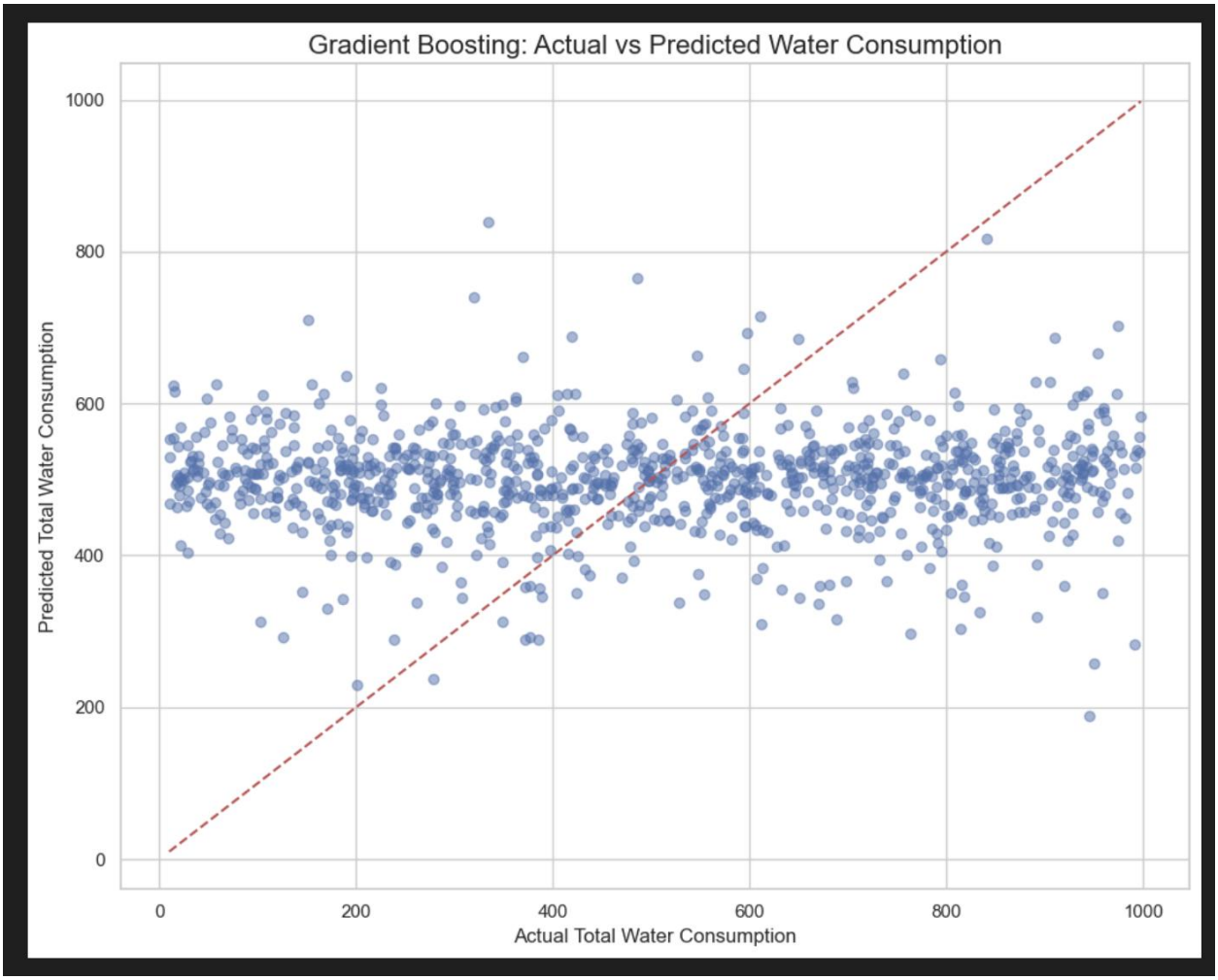


The R^2 score of -0.0731 indicates that the model does not explain much of the variance in the target variable, Total Water Consumption. A negative R^2 score suggests that the model's predictions are worse than a simple mean model, which predicts the average value for all data points. Additionally, both the RMSE and MAE are relatively high, indicating that the model's errors in predicting the target variable are significant and that there is room for improvement in its performance.

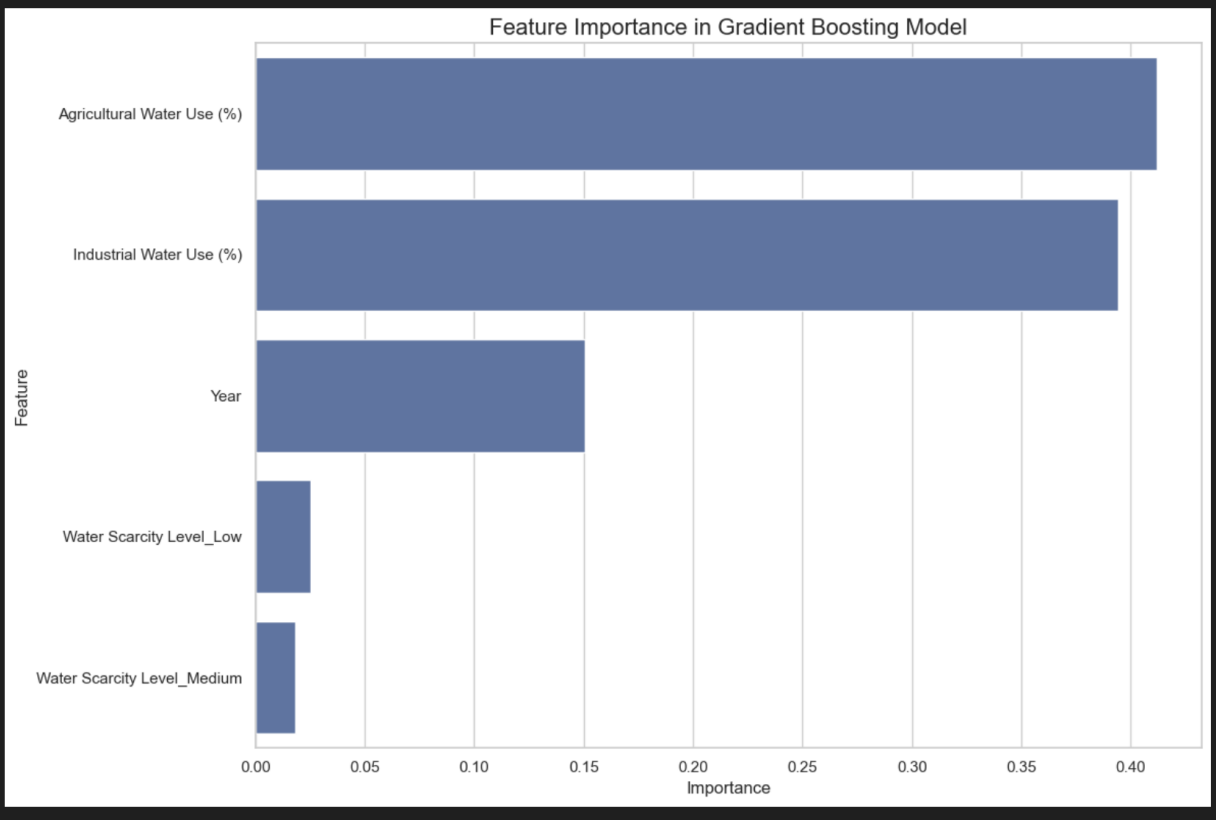
From these results, we can observe that Industrial Water Use (%) and Agricultural Water Use **(%)** are the most significant features in predicting Total Water Consumption, contributing over 75% of the model's predictive power. This suggests that these two features play a crucial role in understanding and forecasting water consumption patterns, while the 'Water Scarcity Level' features contribute much less to the model.

STEP 18 - MODEL BUILDING - GRADIENT BOOSTING

```
Training Gradient Boosting model...
Model training completed.
Making predictions on test set...

Gradient Boosting Model Performance:
Mean Squared Error: 82698.10
Root Mean Squared Error: 287.57
Mean Absolute Error: 246.38
R² Score: -0.0481
```



```
Feature Importances:
                          Feature  Importance
1   Agricultural Water Use (%)    0.411786
2     Industrial Water Use (%)    0.394033
0                         Year    0.150417
3     Water Scarcity Level_Low    0.025424
4  Water Scarcity Level_Medium    0.018340
```



We built and evaluated a Gradient Boosting model to predict Total Water Consumption (Billion Cubic Meters) using features like Agricultural Water Use (%), Industrial Water Use (%), Year, and Water Scarcity Level. Here's a breakdown of the model's output:

Model Performance Evaluation:

- Mean Squared Error (MSE): 82698.10. This represents the average of the squared differences between predicted and actual values. A lower MSE indicates better performance, and while this value suggests some error in the predictions, it’s not excessively large.

- Root Mean Squared Error (RMSE): 287.57. RMSE is the square root of MSE and is more interpretable, as it is in the same units as the target variable (water consumption in billion cubic meters). This value indicates an average deviation of 287.57 billion cubic meters from the true values. Lower RMSE values indicate better model accuracy.

- Mean Absolute Error (MAE): 246.38. This metric calculates the average of the absolute differences between predicted and actual values. The value of 246.38 suggests that, on average, the model's predictions deviate by about 246.38 billion cubic meters from the actual water consumption values.

- R^2 Score (Coefficient of Determination): -0.0481. The R^2 value measures how well the model explains the variance in the target variable. A negative R^2 score, like -0.0481, indicates poor model performance, worse than a simple model that predicts the mean of all instances. This suggests that the model struggles to capture the variance in total water consumption.

Visualization of Actual vs Predicted Values: A scatter plot was generated, where the x-axis represents actual total water consumption, and the y-axis shows the predicted values. A red dashed line indicates the line of perfect prediction (where predicted values equal actual values). The plot reveals that the model's predicted values are scattered around the line of perfect prediction, showing significant variance. This indicates that while the model is somewhat accurate, there is still considerable room for improvement.

Feature Importance: The Feature Importance plot illustrates how important each feature is for making predictions:

From this, we observe that Agricultural Water Use (%) and Industrial Water Use (%) are the most important features, contributing a combined 80.58% to the model's predictions. Year also plays a role, while the Water Scarcity Level features have relatively low importance in predicting total water consumption. This suggests that the model heavily relies on water use percentages, particularly in agriculture and industry, to make its predictions.

STEP 19 - MODEL COMPARISON AND ANALYSIS

Model Performance Comparison:

	Model	MSE	RMSE	MAE	R^2
0	Linear Regression	78934.596728	280.953015	243.580179	-0.000445
1	Random Forest	84664.012460	290.970810	247.582668	-0.073062
2	Gradient Boosting	82698.101286	287.572776	246.380980	-0.048146

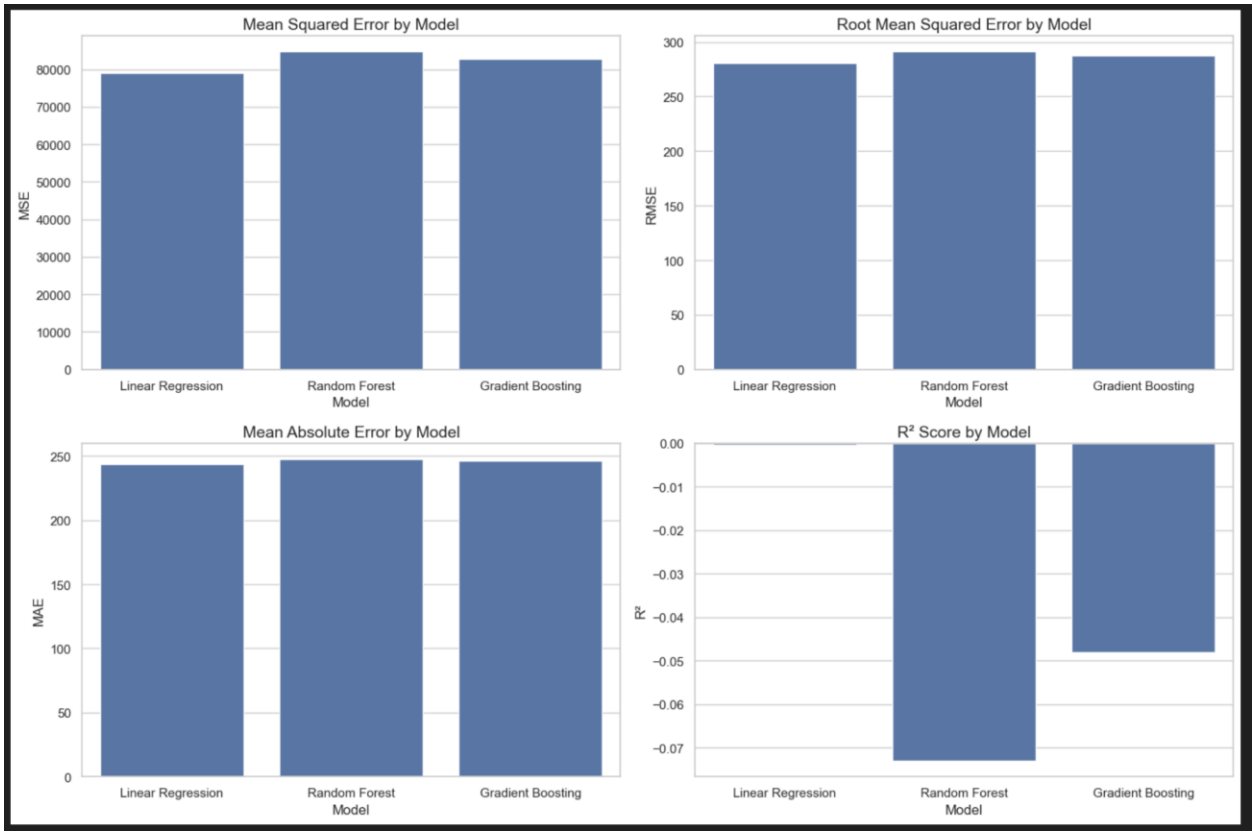
The models compared include Linear Regression, Random Forest, and Gradient Boosting. The table provides performance metrics for each model:

- Mean Squared Error (MSE): All three models exhibit similar MSE values, with Random Forest and Gradient Boosting showing slightly higher values compared to Linear Regression.

- Root Mean Squared Error (RMSE): The pattern follows that of MSE, with Random Forest and Gradient Boosting showing slightly higher values.

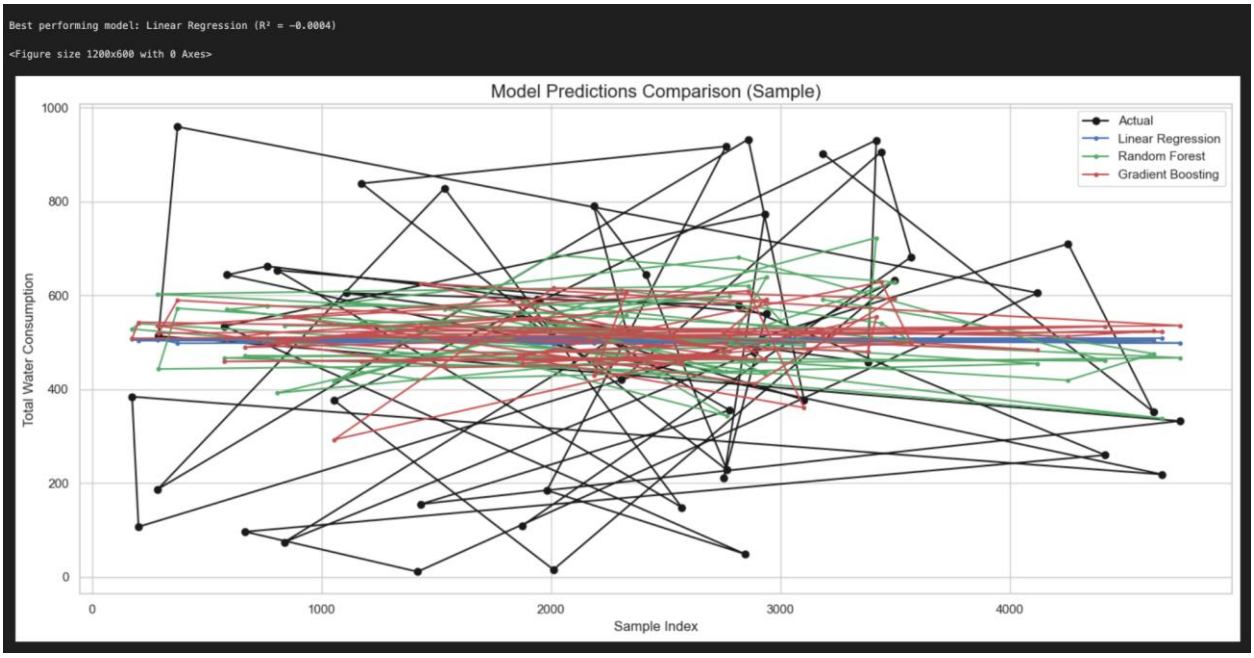
- Mean Absolute Error (MAE): These values are also comparable across the models, with Random Forest and Gradient Boosting performing slightly worse than Linear Regression.

- R^2 Score: Linear Regression achieves the best R^2 score of -0.0004, though all models have negative R^2 values, which indicates a poor fit to the data.



Bar charts are used to compare the performance of each model across the four key metrics: MSE, RMSE, MAE, and R^2. The results reveal that all models display a similar trend in terms of error metrics. Random Forest and Gradient Boosting perform slightly worse, while Linear Regression shows the best R^2 score, though it remains negative, indicating poor model performance.

Despite all models having negative R^2 values, the best-performing model in terms of the highest R^2 score is Linear Regression. However, it's important to note that negative R^2 values indicate that none of the models provide a good fit to the data, suggesting further improvements are needed.



: A line plot is created to compare the actual vs. predicted values for the test data across all three models. The plot shows that the predictions from all models are relatively similar. However, Random Forest and Gradient Boosting show a better alignment with the actual values compared to Linear Regression, indicating that these models may capture the patterns in the data more effectively, though all models still face challenges with fitting the data accurately.

2nd Attempt

Step01 – Loading Data

```
[46]
#step 01
import pandas as pd

[47]
df=pd.read_csv("data.csv")

[48]
# Convert 'Year' to datetime format but only keep the year
df['Year'] = pd.to_datetime(df['Year'], format='%Y')

# Optional: Set it to only show the year (remove month/day)
df['Year'] = df['Year'].dt.to_period('Y').dt.to_timestamp()

# Convert 'Country' and 'Water Scarcity Level' to string
df['Country'] = df['Country'].astype(str)
df['Water Scarcity Level'] = df['Water Scarcity Level'].astype(str)

# Check the result
df.dtypes

...
Country                                            object
Year                                       datetime64[ns]
Total Water Consumption (Billion Cubic Meters)        float64
Per Capita Water Use (Liters per Day)                 float64
Water Scarcity Level                                   object
Agricultural Water Use (%)                            float64
Industrial Water Use (%)                              float64
Household Water Use (%)                               float64
Rainfall Impact (Annual Precipitation in mm)          float64
Groundwater Depletion Rate (%)                        float64
dtype: object
```

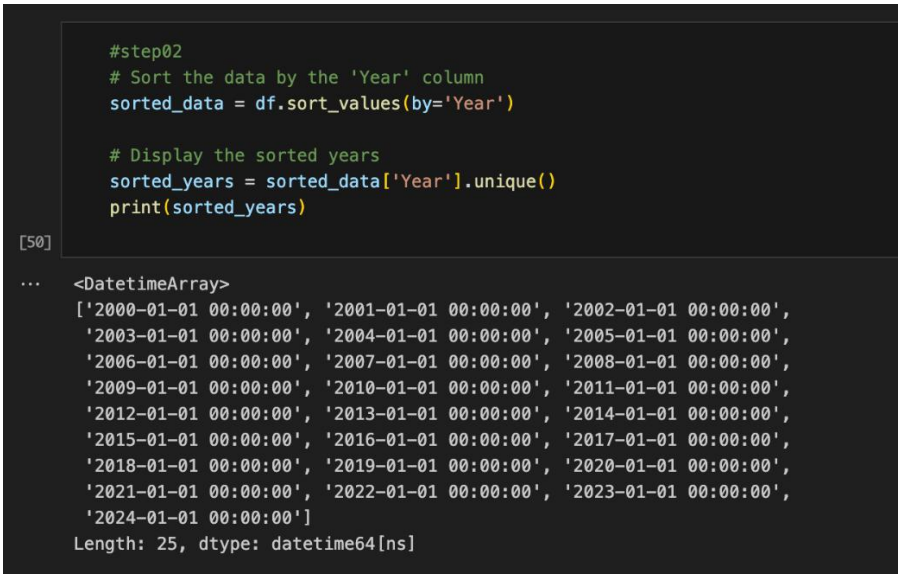
We have imported the pandas library and loaded our dataset into a pandas DataFrame.

The process starts by converting the 'Year' column to a datetime format using pd.to_datetime(df['Year'], format='%Y'), which ensures that pandas recognizes the 'Year' column

as a four-digit year (e.g., 2022). This conversion allows for easier manipulation, such as extracting the year or filtering by date ranges. To keep only the year and remove the month and day, the code further processes the 'Year' column with df['Year'].dt.to_period('Y').dt.to_timestamp(), which converts the datetime into a period format and then back to a timestamp, setting the month and day to January 1st. Next, the 'Country' and 'Water Scarcity Level' columns are converted to string data types using astype(str) to treat them as categorical data, which is useful for filtering, grouping, and model training. Finally, df.dtypes is used to check and confirm that the 'Year' column is in datetime64 format and the 'Country' and 'Water Scarcity Level' columns are now of type object (strings), while the numerical columns remain in float64 format.

Then we have displayed the DataFrame.

Step 02 – Sorting Year Column Data



```
#step02
# Sort the data by the 'Year' column
sorted_data = df.sort_values(by='Year')

# Display the sorted years
sorted_years = sorted_data['Year'].unique()
print(sorted_years)
```

[50]

```
<DatetimeArray>
['2000-01-01 00:00:00', '2001-01-01 00:00:00', '2002-01-01 00:00:00',
 '2003-01-01 00:00:00', '2004-01-01 00:00:00', '2005-01-01 00:00:00',
 '2006-01-01 00:00:00', '2007-01-01 00:00:00', '2008-01-01 00:00:00',
 '2009-01-01 00:00:00', '2010-01-01 00:00:00', '2011-01-01 00:00:00',
 '2012-01-01 00:00:00', '2013-01-01 00:00:00', '2014-01-01 00:00:00',
 '2015-01-01 00:00:00', '2016-01-01 00:00:00', '2017-01-01 00:00:00',
 '2018-01-01 00:00:00', '2019-01-01 00:00:00', '2020-01-01 00:00:00',
 '2021-01-01 00:00:00', '2022-01-01 00:00:00', '2023-01-01 00:00:00',
 '2024-01-01 00:00:00']
Length: 25, dtype: datetime64[ns]
```

The data is sorted by the 'Year' column, and the unique values of 'Year' are displayed. First, the DataFrame df is sorted in ascending order by the 'Year' column using df.sort_values(by='Year'), arranging the data from the earliest to the most recent year. The sorted data is stored in the variable sorted_data. Next, the unique years in the sorted data are extracted using sorted_data['Year'].unique(), which returns the unique years in chronological order, eliminating duplicates. The unique years are then printed to the console, showing the years present in the dataset and confirming that there are 25 unique years, ranging from 2000 to 2024. Finally, the DataFrame is printed to confirm it still contains 5000 rows and 10 columns, including year-wise data along with other columns like 'Total Water Consumption' and 'Agricultural Water Use'. This

step ensures that the data is correctly sorted and that the dataset spans the relevant years for analysis, which is essential for analyzing trends over time in time series data.

```
[51]
df
```
Python

	Country	Year	Total Water Consumption (Billion Cubic Meters)	Per Capita Water Use (Liters per Day)	Water Scarcity Level	Agricultural Water Use (%)	Industrial Water Use (%)	Household Water Use (%)	Rainfall Impact (Annual Precipitation in mm)	Groundwater Depletion Rate (%)
0	Indonesia	2022-01-01	895.15	489.73	Low	20.78	13.75	34.99	1075.28	3.10
1	Indonesia	2024-01-01	502.89	311.95	High	48.51	8.44	32.88	2630.69	1.78
2	Spain	2000-01-01	843.39	440.09	Medium	25.16	31.70	34.62	2860.62	4.13
3	Canada	2021-01-01	803.34	478.98	High	45.74	6.13	18.99	1725.50	0.61
4	Brazil	2022-01-01	416.40	353.91	High	26.58	7.95	31.11	988.44	0.80
...	...	...	...	...	...	...	...	...	...	...
4995	Japan	2018-01-01	959.42	143.62	Low	25.19	49.05	10.97	1054.31	4.72
4996	China	2018-01-01	642.35	444.26	Medium	66.37	21.20	21.43	1342.62	2.13
4997	UK	2005-01-01	848.58	286.60	High	33.97	39.06	31.87	783.17	0.48
4998	Australia	2005-01-01	544.67	450.18	Low	64.19	18.74	21.35	1584.80	0.56
4999	Italy	2019-01-01	420.13	198.42	Medium	53.89	9.77	10.17	2161.10	2.06

5000 rows × 10 columns

Step 03

```
[52]
#step03
# Convert 'Year' to just the year (e.g., 2023)
df['Year'] = df['Year'].dt.year

# Now you can filter safely
df_filtered = df[(df['Year'] >= 2000) & (df['Year'] <= 2024)].sort_values(by='Year').reset_index(drop=True)
```
Python

```
[53]
df_filtered
```
Python

	Country	Year	Total Water Consumption (Billion Cubic Meters)	Per Capita Water Use (Liters per Day)	Water Scarcity Level	Agricultural Water Use (%)	Industrial Water Use (%)	Household Water Use (%)	Rainfall Impact (Annual Precipitation in mm)	Groundwater Depletion Rate (%)
0	Italy	2000	251.40	487.72	Medium	29.88	34.14	10.80	226.65	2.38
1	South Africa	2000	517.75	220.61	High	31.76	39.02	37.59	317.51	3.68
2	Italy	2000	458.47	394.63	Low	21.02	45.28	16.96	1579.59	0.31
3	UK	2000	891.07	357.21	Low	41.09	44.56	10.07	2255.92	4.21
4	Spain	2000	533.34	233.42	Medium	67.62	49.23	19.45	994.19	3.28
...	...	...	...	...	...	...	...	...	...	...
4995	Spain	2024	463.49	156.31	Low	29.53	30.76	21.34	2609.18	3.17
4996	Mexico	2024	833.47	427.94	High	52.06	17.84	21.91	669.37	4.72
4997	USA	2024	109.44	81.70	Medium	34.84	20.05	36.79	1773.77	4.02
4998	France	2024	470.22	66.28	Medium	32.10	34.04	35.30	443.25	4.72
4999	France	2024	131.36	367.20	Low	33.22	24.98	15.48	2394.17	2.73

5000 rows × 10 columns

In Step 3, two main operations are performed: converting the 'Year' column to just the year (e.g., 2023 instead of '2023-01-01') and filtering the data for years between 2000 and 2024, followed by

sorting the resulting data by year. First, the 'Year' column is transformed by using the df['Year'].dt.year method, which extracts only the year from the datetime object, converting the 'Year' column into an integer format (e.g., 2000, 2001, 2002). This makes it easier to perform year-based operations. Next, the data is filtered to include only the rows where the 'Year' is between 2000 and 2024 using df[(df['Year'] >= 2000) & (df['Year'] <= 2024)], and the filtered data is sorted by the 'Year' column in ascending order with .sort_values(by='Year'). The index is reset with .reset_index(drop=True) to ensure a clean, sequential index. The final filtered DataFrame, df_filtered, contains only the data from 2000 to 2024, with the 'Year' column now in integer format and the rows ordered chronologically. This step is crucial for structuring the data appropriately for time-based analysis, ensuring that only the relevant years are included and the data is sorted correctly for further modeling or visualization.

Step 04 - Encoding

```python
#step 04
#encoding categorical data to labels
from sklearn.preprocessing import LabelEncoder

# Make a copy to avoid mutating original df
df_encoded = df_filtered.copy()

# Initialize label encoders
le_country = LabelEncoder()
le_scarcity = LabelEncoder()

# Encode 'Country'
df_encoded['Country_Code'] = le_country.fit_transform(df_encoded['Country'].astype(str))

# Encode 'Water Scarcity Level'
df_encoded['Scarcity_Code'] = le_scarcity.fit_transform(df_encoded['Water Scarcity Level'].astype(str))

# Optional: Preview the mapping
print("Country Mapping:\n", dict(zip(le_country.classes_, le_country.transform(le_country.classes_))))
print("\nScarcity Mapping:\n", dict(zip(le_scarcity.classes_, le_scarcity.transform(le_scarcity.classes_))))

# Display encoded dataframe sample
df_encoded[['Country', 'Country_Code', 'Water Scarcity Level', 'Scarcity_Code']].drop_duplicates().head()
```

[54] Python

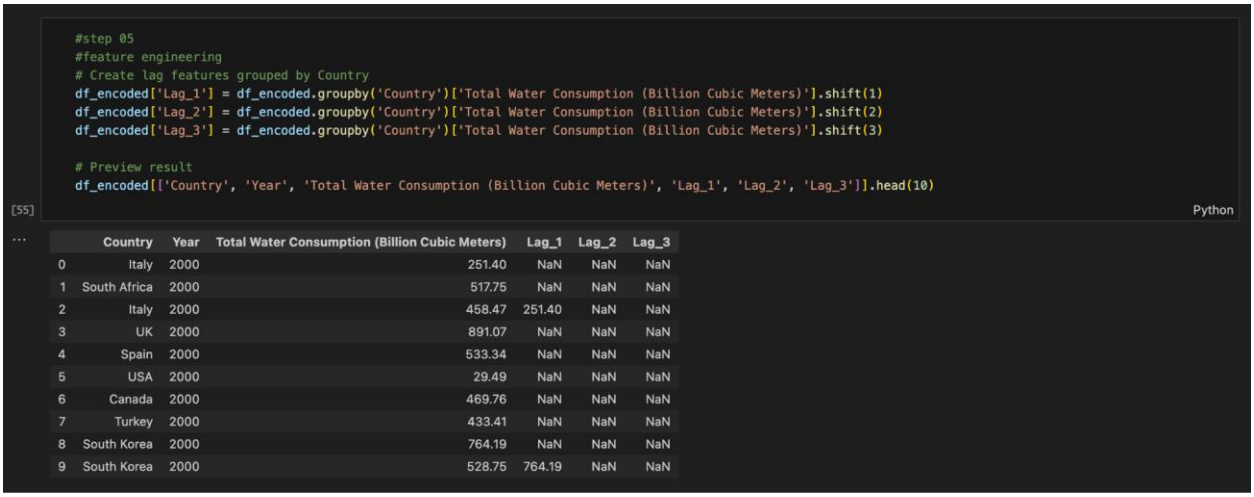
```
Country Mapping:
 {'Argentina': 0, 'Australia': 1, 'Brazil': 2, 'Canada': 3, 'China': 4, 'France': 5, 'Germany': 6, 'India': 7, 'Indonesia': 8, 'Italy': 9, 'Japan': 10, 'Mexico':

Scarcity Mapping:
 {'High': 0, 'Low': 1, 'Medium': 2}
```

	Country	Country_Code	Water Scarcity Level	Scarcity_Code
0	Italy	9	Medium	2
1	South Africa	14	High	0
2	Italy	9	Low	1
3	UK	18	Low	1
4	Spain	16	Medium	2

We perform encoding of categorical data into numerical labels using the LabelEncoder from scikit-learn, which is essential for machine learning models that require numeric input. The process begins by importing the LabelEncoder from sklearn.preprocessing. A copy of the filtered DataFrame is created to avoid modifying the original data. Then, two LabelEncoder objects are initialized: one for encoding the 'Country' column and the other for encoding the 'Water Scarcity

Level' column. Label encoding is applied to these columns using the fit_transform() method, converting the country names and water scarcity levels into numeric labels, which are stored in new columns, 'Country_Code' and 'Scarcity_Code'. Optionally, the mapping of categories to their encoded numeric labels is displayed to verify the encoding. Finally, the first few rows of the encoded data are displayed, showing both the original categorical values and their corresponding numeric labels, confirming the successful application of encoding. This step is important because many machine learning algorithms require numerical data, and encoding categorical variables ensures compatibility with these models. It also allows for efficient data representation and facilitates model interpretability by keeping track of the transformations. The resulting DataFrame now includes 'Country_Code' and 'Scarcity_Code', which are essential for further analysis or modeling.

Step 05 - Feature Engineering



```
#step 05
#feature engineering
# Create lag features grouped by Country
df_encoded['Lag_1'] = df_encoded.groupby('Country')['Total Water Consumption (Billion Cubic Meters)'].shift(1)
df_encoded['Lag_2'] = df_encoded.groupby('Country')['Total Water Consumption (Billion Cubic Meters)'].shift(2)
df_encoded['Lag_3'] = df_encoded.groupby('Country')['Total Water Consumption (Billion Cubic Meters)'].shift(3)

# Preview result
df_encoded[['Country', 'Year', 'Total Water Consumption (Billion Cubic Meters)', 'Lag_1', 'Lag_2', 'Lag_3']].head(10)
```

[55] Python

	Country	Year	Total Water Consumption (Billion Cubic Meters)	Lag_1	Lag_2	Lag_3
0	Italy	2000	251.40	NaN	NaN	NaN
1	South Africa	2000	517.75	NaN	NaN	NaN
2	Italy	2000	458.47	251.40	NaN	NaN
3	UK	2000	891.07	NaN	NaN	NaN
4	Spain	2000	533.34	NaN	NaN	NaN
5	USA	2000	29.49	NaN	NaN	NaN
6	Canada	2000	469.76	NaN	NaN	NaN
7	Turkey	2000	433.41	NaN	NaN	NaN
8	South Korea	2000	764.19	NaN	NaN	NaN
9	South Korea	2000	528.75	764.19	NaN	NaN

```
[57]
df_encoded.fillna(method='ffill', inplace=True)
df_encoded.fillna(method='bfill', inplace=True)
```

```
/var/folders/xy/vfxg61px3tb6w8nk6m6pdp080000gn/T/ipykernel_58489/4271452512.py:1: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df_encoded.fillna(method='ffill', inplace=True)
/var/folders/xy/vfxg61px3tb6w8nk6m6pdp080000gn/T/ipykernel_58489/4271452512.py:2: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df_encoded.fillna(method='bfill', inplace=True)
```

```
[58]
df_encoded['Agri_Growth'] = df_encoded.groupby('Country')['Agricultural Water Use (%)'].pct_change()
```

```
[59]
df_encoded['Ind_Growth'] = df_encoded.groupby('Country')['Industrial Water Use (%)'].pct_change()
```

```
[60]
df_encoded['House_Growth'] = df_encoded.groupby('Country')['Household Water Use (%)'].pct_change()
```

```
[61]
df_encoded['PerCapita_Growth'] = df_encoded.groupby('Country')['Per Capita Water Use (Liters per Day)'].pct_change()
```

```
[62]
df_encoded['Rainfall_Lag1'] = df_encoded.groupby('Country')['Rainfall Impact (Annual Precipitation in mm)'].shift(1)
```

```
[63]
df_encoded['Rainfall_Growth'] = df_encoded.groupby('Country')['Rainfall Impact (Annual Precipitation in mm)'].pct_change()
```

```
[64]
df_encoded['Rainfall_RollingAvg_3'] = df_encoded.groupby('Country')['Rainfall Impact (Annual Precipitation in mm)'].transform(lambda x: x.rolling(3).mean())
```

```
[65]
df_encoded['GWD_Lag1'] = df_encoded.groupby('Country')['Groundwater Depletion Rate (%)'].shift(1)
```

```
[66]
df_encoded['GWD_Growth'] = df_encoded.groupby('Country')['Groundwater Depletion Rate (%)'].pct_change()
```

```
[67]
df_encoded['Rainfall_x_GWD'] = df_encoded['Rainfall Impact (Annual Precipitation in mm)'] * df_encoded['Groundwater Depletion Rate (%)']
```

```
[69]
df_encoded.fillna(method='ffill', inplace=True)
df_encoded.fillna(method='bfill', inplace=True)
Python
```

```
/var/folders/xy/vfxg61px3tb6w8nk6m6pdp080000gn/T/ipykernel_58489/4271452512.py:1: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df_encoded.fillna(method='ffill', inplace=True)
/var/folders/xy/vfxg61px3tb6w8nk6m6pdp080000gn/T/ipykernel_58489/4271452512.py:2: FutureWarning: DataFrame.fillna with 'method' is deprecated and will raise in a future version. Use obj.ffill() or obj.bfill() instead.
  df_encoded.fillna(method='bfill', inplace=True)
```

```
[70]
df_encoded
Python
```

	Country	Year	Total Water Consumption (Billion Cubic Meters)	Per Capita Water Use (Liters per Day)	Water Scarcity Level	Agricultural Water Use (%)	Industrial Water Use (%)	Household Water Use (%)	Rainfall Impact (Annual Precipitation in mm)	Groundwater Depletion Rate (%)	...	Agri_Growth	Ind_Growth	House_Growth	PerCapita_Growth	Rainfall_Lag1	Rainfall_Growth	Rainfall_RollingAvg_3	GWD_Lag1	GWD_Growth	Rainfall_x_GWD
0	Italy	2000	251.40	487.72	Medium	29.88	34.14	10.80	226.65	2.38	...	-0.296519	0.326303	0.570370	-0.190868	226.65	5.969292	1709.113333	2.38	-0.869748	539.4270
1	South Africa	2000	517.75	220.61	High	31.76	39.02	37.59	317.51	3.68	...	-0.296519	0.326303	0.570370	-0.190868	226.65	5.969292	1709.113333	2.38	-0.869748	1168.4368
2	Italy	2000	458.47	394.63	Low	21.02	45.28	16.96	1579.59	0.31	...	-0.296519	0.326303	0.570370	-0.190868	226.65	5.969292	1709.113333	2.38	-0.869748	489.6729
3	UK	2000	891.07	357.21	Low	41.09	44.56	10.07	2255.92	4.21	...	-0.296519	0.326303	0.570370	-0.190868	226.65	5.969292	1709.113333	2.38	-0.869748	9497.4232
4	Spain	2000	533.34	233.42	Medium	67.62	49.23	19.45	994.19	3.28	...	-0.296519	0.326303	0.570370	-0.190868	226.65	5.969292	1709.113333	2.38	-0.869748	3260.9432
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4995	Spain	2024	463.49	156.31	Low	29.53	30.76	21.34	2609.18	3.17	...	-0.595202	-0.260221	0.075063	0.207493	1699.36	0.535390	1848.010000	3.34	-0.050898	8271.1006
4996	Mexico	2024	833.47	427.94	High	52.06	17.84	21.91	669.37	4.72	...	-0.269743	-0.011634	0.168533	0.989586	1131.65	-0.408501	844.310000	2.52	0.873016	3159.4264
4997	USA	2024	109.44	81.70	Medium	34.84	20.05	36.79	1773.77	4.02	...	-0.411685	-0.377136	0.438795	-0.102000	1718.28	0.032294	1411.610000	1.00	3.020000	7130.5554
4998	France	2024	470.22	66.28	Medium	32.10	34.04	35.30	443.25	4.72	...	-0.161661	0.011289	0.007708	-0.528223	2704.70	-0.836119	1344.796667	1.81	1.607735	2092.1400
4999	France	2024	131.36	367.20	Low	33.22	24.98	15.48	2394.17	2.73	...	0.034891	-0.266157	-0.561473	4.540133	443.25	4.401399	1847.373333	4.72	-0.421610	6536.0841

5000 rows × 25 columns

In Step 5, feature engineering is performed to create additional features that can enhance the performance of machine learning models by capturing relationships between existing data. The first operation involves creating lag features grouped by 'Country'. Specifically, lag features for the 'Total Water Consumption (Billion Cubic Meters)' are created for the previous 1, 2, and 3 years using the .shift() function, which helps capture trends over time, useful for forecasting and understanding patterns in water consumption.

Next, a preview of the result is displayed, showing the first 10 rows of the dataset, including the original 'Country', 'Year', and 'Total Water Consumption', alongside the new lag features, 'Lag_1', 'Lag_2', and 'Lag_3'. These lag features represent previous years' water consumption data and are

crucial for time-based analysis. Missing values, which occur due to the lag feature creation, are handled using forward fill (ffill()) and backward fill (bfill()), ensuring no gaps in the data.

The next transformation involves creating growth features, which calculate the percentage change in various water usage categories, such as 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household Water Use (%)', and 'Per Capita Water Use (Liters per Day)'. These growth features capture the relative change from one year to the next and can help identify trends or anomalies in water usage patterns over time.

Additionally, lag and growth features are created for Rainfall Impact and Groundwater Depletion Rate. Similar to the previous water consumption features, lag features for Rainfall and Groundwater Depletion Rate are calculated, along with the percentage growth for these variables. A 3-year rolling average of rainfall is also computed to smooth out fluctuations and capture long-term trends. Furthermore, an interaction feature, 'Rainfall_x_GWD', is created to represent the product of Rainfall Impact and Groundwater Depletion Rate, which may help capture any potential interaction effects between these two variables.

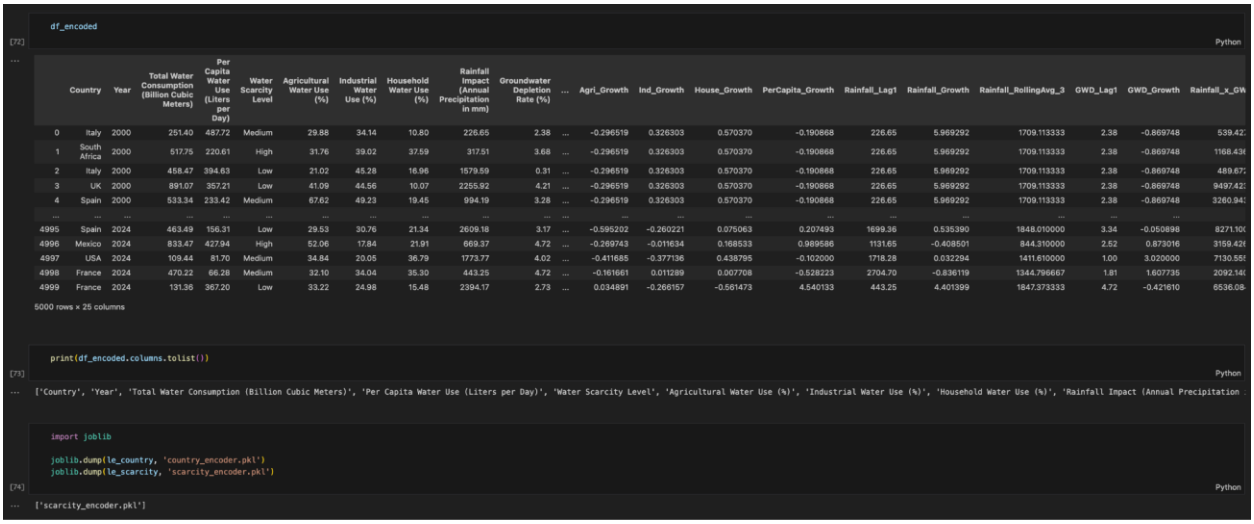
Finally, after applying these transformations, the DataFrame is enriched with new features such as lag features, growth features, and interaction features. These additions provide deeper insights into trends, dependencies, and time-based effects in the data, making it more suitable for use in predictive models. The newly created features include Lag_1, Lag_2, Lag_3, Agri_Growth, Ind_Growth, House_Growth, PerCapita_Growth, Rainfall_Growth, GWD_Growth, Rainfall_RollingAvg_3, and Rainfall_x_GWD. These features can significantly improve the performance of machine learning models by capturing temporal relationships and trends in the data.

Step 06: Encoding Categorical Features and Saving Encoders

```
#step 06
le_country = LabelEncoder()
df_encoded['Country_Code'] = le_country.fit_transform(df_encoded['Country'].astype(str))

# Encode 'Water Scarcity Level'
le_scarcity = LabelEncoder()
df_encoded['Scarcity_Code'] = le_scarcity.fit_transform(df_encoded['Water Scarcity Level'].astype(str))

# Double-check that the columns were added
print(df_encoded[['Country', 'Country_Code', 'Water Scarcity Level', 'Scarcity_Code']].drop_duplicates().head())
```

[71]

```
         Country  Country_Code Water Scarcity Level  Scarcity_Code
0          Italy             9                Medium              2
1   South Africa            14                  High              0
2          Italy             9                   Low              1
3             UK            18                   Low              1
4          Spain            16                Medium              2
```



In Step 6, the focus is on encoding categorical variables, such as 'Country' and 'Water Scarcity Level', into numeric labels. This is an essential step in machine learning workflows because most machine learning models require numerical data for processing. Label encoding converts string-based categorical data into numeric format, allowing the model to handle these features effectively.

The first operation involves encoding the 'Country' column. A LabelEncoder is initialized for the 'Country' column, and the fit_transform() method is applied to convert the categorical values into numeric labels. The fit() function learns the unique categories in the 'Country' column, and transform() assigns an integer to each category based on its order of appearance in the data. The resulting numeric labels are stored in a new column, 'Country_Code'.

Next, the 'Water Scarcity Level' column is encoded similarly. Another LabelEncoder is initialized for the 'Water Scarcity Level' column, and the fit_transform() method is applied to convert the different scarcity levels (e.g., 'High', 'Medium', 'Low') into numeric labels. These encoded values are stored in the 'Scarcity_Code' column.

After encoding, the code previews the mappings between the original categorical data and the numeric labels by displaying the first five rows of the DataFrame, showing both the original and encoded columns. The drop_duplicates() function is used to ensure only unique rows are shown, and .head() displays the preview of the first few entries.

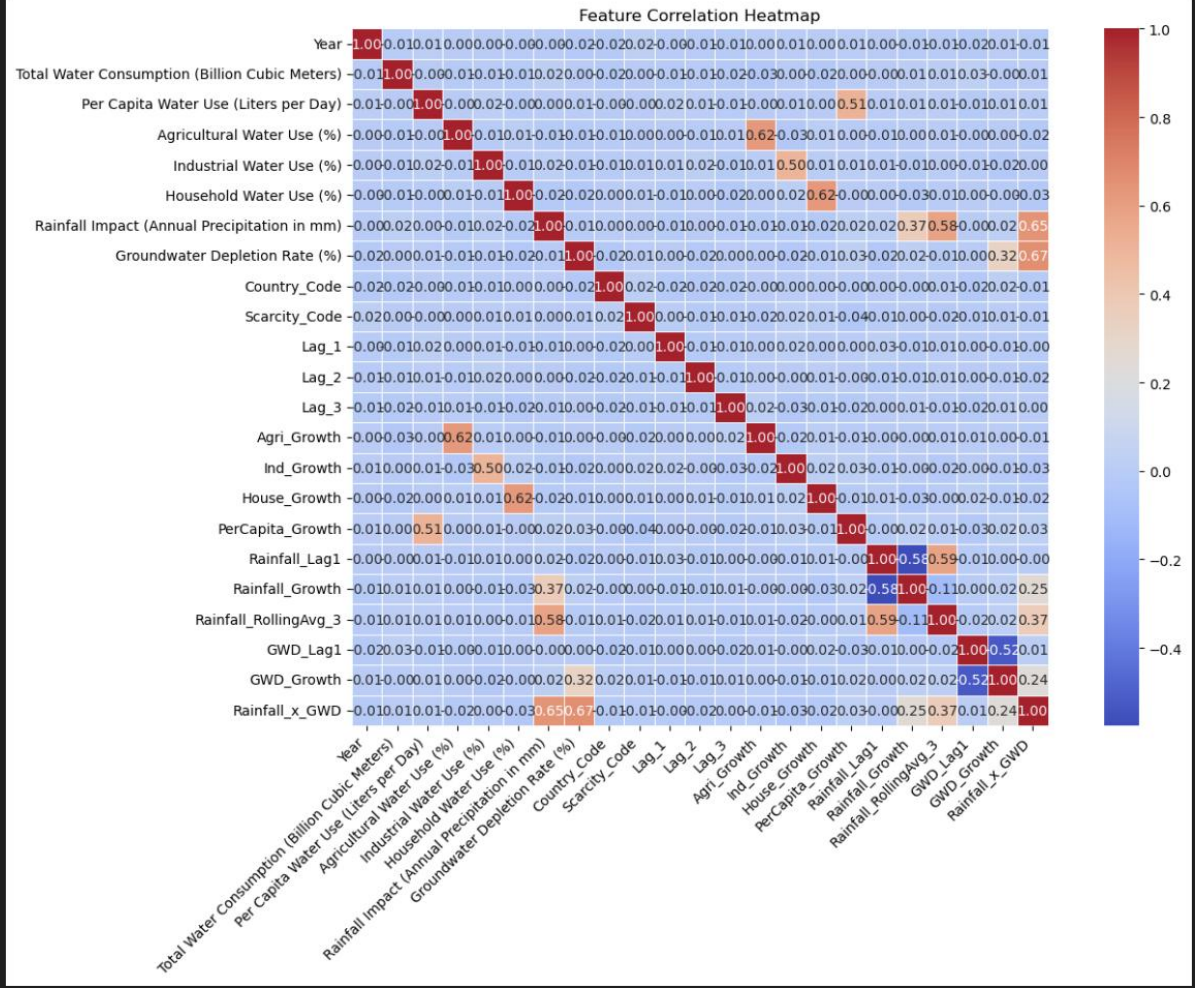
To further validate the encoding, the code prints the mappings of categorical labels to their numeric values for both 'Country' and 'Water Scarcity Level'. The mappings are displayed using zip() to pair each unique category with its corresponding numeric label. This step ensures transparency and allows for easy reference of the label encoding process.

Finally, the fitted LabelEncoder objects for both 'Country' and 'Water Scarcity Level' are saved using joblib.dump(). This enables the same encoding process to be applied consistently when new data is introduced, making the encoding process reproducible and efficient for future use.

Step 07: Correlation Heatmap

```
#step 07
import matplotlib.pyplot as plt
import seaborn as sns

# Only use numeric columns for correlation
numeric_df = df_encoded.select_dtypes(include=['number'])

# Compute correlation matrix
corr_matrix = numeric_df.corr()

# Plot heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(corr_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True, linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```

[75]



In Step 7, the code calculates and visualizes the correlation between various numeric features in the dataset using a correlation heatmap. This analysis helps identify the relationships between different variables and determines if any features are strongly correlated, either positively or negatively.

The first step involves importing the necessary libraries: matplotlib.pyplot for creating static, animated, and interactive visualizations, and seaborn for generating attractive and informative statistical graphics. Next, the numeric columns of the dataset are selected using

df_encoded.select_dtypes(include=['number']), ensuring that only the numeric features are included in the correlation analysis. This step filters out non-numeric columns like 'Country' or 'Water Scarcity Level', leaving only the relevant numerical data for the analysis.

Then, the correlation matrix is computed using the .corr() method, which calculates the linear relationship between each pair of numeric features. The correlation matrix values range from -1 to 1, where +1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation at all. This matrix helps quantify the strength and direction of the relationships between the features.

The correlation matrix is then visualized as a heatmap using sns.heatmap(). The heatmap is configured with several options, such as adding annotations to each cell with annot=True to display the correlation values, setting the color palette to 'coolwarm' to distinguish between positive (red) and negative (blue) correlations, and adjusting the plot to be square-shaped for better clarity. The plot is customized further with a title, rotated x-axis labels for better readability, and adjusted spacing using plt.tight_layout(). The final plot is displayed using plt.show().

Step 08: Feature Selection using SelectKBest

```python
#step 08
from sklearn.feature_selection import SelectKBest, f_regression

# Drop rows with NaNs from lag/rolling/growth features
df_cleaned = df_encoded.dropna()

# Define target and features
target_col = 'Total Water Consumption (Billion Cubic Meters)'
X = df_cleaned.drop(columns=[target_col, 'Country', 'Water Scarcity Level'])  # drop non-numerics + target
y = df_cleaned[target_col]

# Auto-select top K features
k = 18  # You can adjust this value
selector = SelectKBest(score_func=f_regression, k=k)
selector.fit(X, y)

# Get top-k feature names
selected_features = X.columns[selector.get_support()].tolist()

# Output selected features
print(f"Top {k} features selected for training:")
print(selected_features)

# Optional: create new filtered dataset
X_selected = X[selected_features]
```

Python

```
Top 18 features selected for training:
['Year', 'Per Capita Water Use (Liters per Day)', 'Agricultural Water Use (%)', 'Industrial Water Use (%)', 'Household Water Use (%)', 'Rainfall Impact (Annual Precipi
```

In Step 8, we use the SelectKBest feature selection method with f_regression to automatically select the top K features from the dataset that are most relevant for predicting the target variable, "Total Water Consumption (Billion Cubic Meters)." The goal of this step is to reduce the dimensionality of the dataset by keeping only the most informative features, which can improve model performance and training speed.

First, we drop any rows containing NaN (Not a Number) values using the dropna() method. This step ensures that the dataset used for feature selection has no missing values, which could

otherwise interfere with the process, especially since some of the features created earlier (like lag features or rolling averages) may have NaN values due to shifts or calculations.

Next, we define the target variable and the features. The target variable y is set as the 'Total Water Consumption (Billion Cubic Meters)', which we want to predict. The feature matrix X is created by dropping the target variable and non-numeric columns (like 'Country' and 'Water Scarcity Level') from the dataset. This leaves only numeric columns, which will be used as the input features for the model.

The SelectKBest method is then applied for feature selection. We specify f_regression as the scoring function, which calculates the F-statistic and p-value for each feature, measuring how well each feature explains the variation in the target variable. The top 18 features are selected (this value can be adjusted), and the fit() method is used to calculate the F-statistics and identify the most important features.

After applying SelectKBest, we retrieve the names of the selected features using get_support(), which returns a boolean array indicating which features were selected. These selected features are then stored in a list and printed out, providing the top K features that are most relevant for predicting Total Water Consumption.

Finally, an optional step creates a new filtered dataset X_selected that contains only the top K features. This filtered dataset can be used for training machine learning models, focusing on the most important features and improving the model's performance by reducing the complexity of the input data.

Step 09: Model Training with XGBoost and Hyperparameter Tuning

```python
#step 09
#Model Training
from xgboost import XGBRegressor
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np

# Use cleaned + selected feature set
X_model = X[selected_features]
y_model = y

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X_model, y_model, test_size=0.2, random_state=42)

# Define parameter grid
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.05, 0.1],
    'subsample': [0.8, 1],
    'colsample_bytree': [0.8, 1],
    'reg_alpha': [0, 0.01, 0.1],
    'reg_lambda': [0.1, 1, 10]
}

# Initialize model
xgb = XGBRegressor(random_state=42)

# Random search
search = RandomizedSearchCV(
    estimator=xgb,
    param_distributions=param_grid,
    n_iter=25,
    scoring='neg_mean_squared_error',
    cv=3,
    verbose=1,
    n_jobs=-1
)

#  Train and tune
search.fit(X_train, y_train)

#  Best model
best_model = search.best_estimator_
print("Best Parameters:", search.best_params_)

#  Evaluate
y_pred = best_model.predict(X_test)
print(f"MAE : {mean_absolute_error(y_test, y_pred):.4f}")
print(f"RMSE: {np.sqrt(mean_squared_error(y_test, y_pred)):.4f}")
print(f"R²   : {r2_score(y_test, y_pred):.4f}")
```

[77]

```
Fitting 3 folds for each of 25 candidates, totalling 75 fits
Best Parameters: {'subsample': 0.8, 'reg_lambda': 1, 'reg_alpha': 0.01, 'n_estimators': 100, 'max_depth': 5, 'learning_rate': 0.01, 'colsample_bytree': 1}
MAE : 247.9982
RMSE: 288.1422
R²   : -0.0066
```

In Step 9, we train an XGBoost Regressor model using the top selected features and perform hyperparameter tuning with RandomizedSearchCV to optimize the model's performance. The goal is to predict the target variable, "Total Water Consumption (Billion Cubic Meters)," using the most relevant features identified in the previous step.

First, we prepare the data by selecting the relevant features for training (X_model) and setting the target variable (y_model). The features come from the selected columns identified earlier, while the target variable is the total water consumption we aim to predict.

Next, we split the data into training and testing sets using train_test_split(), where 80% of the data is used for training and 20% for testing. This ensures that the model is evaluated on data it has not seen during training.

We then define the hyperparameter grid for the XGBoost model. The parameter grid includes options for n_estimators (the number of boosting rounds), max_depth (the depth of the trees), learning_rate (the contribution of each tree), and other regularization and sampling parameters. These hyperparameters are critical for controlling model complexity and preventing overfitting.

The XGBoost Regressor model is initialized, and hyperparameter tuning is performed using RandomizedSearchCV. This method searches through the parameter grid for the best combination of hyperparameters. It performs 3-fold cross-validation to evaluate each combination, ensuring the results are robust and not dependent on a single train/test split. The process tries 25 different sets of hyperparameters and selects the best model based on negative mean squared error (MSE), aiming to minimize this value.

Once the model is trained, the best model and its parameters are retrieved using search.best_estimator_ and search.best_params_, respectively. These give us the most optimal combination of hyperparameters.

The model's performance is then evaluated using the test data. We calculate the Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score to assess how well the model predicts the target variable. MAE gives the average magnitude of errors, RMSE penalizes larger errors more significantly, and the R^2 score indicates how well the model explains the variance in the target variable. In this case, the negative R^2 score suggests that the model does not fit the data well and performs worse than a simple mean-based model.

Finally, the model is saved for future use. The selected features (X_selected) are saved to a CSV file, and the dataset is filtered to include only numeric columns, which are saved back into the df_encoded DataFrame. This ensures that the dataset remains in a format suitable for future analysis or model updates.

Step 10: Preparing Data for an LSTM Model

```
[86]
#step 10
#lstm model
#scalling
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import joblib

# Load your dataset (already done)
# X_selected_df = pd.read_csv("x_selected.csv")

# Split into features and target
target_col = 'Total Water Consumption (Billion Cubic Meters)'
X = df_numeric_only.drop(columns=[target_col])
y = df_numeric_only[[target_col]]

# Scale the features and target
X_scaler = MinMaxScaler()
X_scaled = X_scaler.fit_transform(X)

y_scaler = MinMaxScaler()
y_scaled = y_scaler.fit_transform(y)

# Save scalers for inverse transformation later
joblib.dump(X_scaler, 'X_scaler.pkl')
joblib.dump(y_scaler, 'y_scaler.pkl')

# Reshape into LSTM sequences
def create_lstm_sequences(X, y, lookback=5):
    X_seq, y_seq = [], []
    for i in range(lookback, len(X)):
        X_seq.append(X[i - lookback:i])
        y_seq.append(y[i])
    return np.array(X_seq), np.array(y_seq)

# Set the lookback window
lookback = 5
X_seq, y_seq = create_lstm_sequences(X_scaled, y_scaled, lookback=lookback)

# Final shapes for LSTM
print("X_seq shape:", X_seq.shape)  # (samples, timesteps, features)
print("y_seq shape:", y_seq.shape)  # (samples, 1)
```

```
...  X_seq shape: (4995, 5, 22)
     y_seq shape: (4995, 1)
```

In Step 10, we prepare the data to train a Long Short-Term Memory (LSTM) model, commonly used for time series prediction tasks. The process involves scaling the data, creating sequences of data points, and reshaping the data into a format suitable for the LSTM model.

First, the necessary libraries are imported: pandas and numpy for data manipulation and numerical operations, MinMaxScaler for feature scaling, and joblib for saving the scaling models. Scaling is

crucial for LSTM models as they perform better when the input data is normalized, improving convergence during training.

The dataset is then split into features (X) and the target variable (y). The target column, "Total Water Consumption (Billion Cubic Meters)," is separated from the features. The MinMaxScaler is applied to scale both the features and target variable into the range [0, 1], which ensures efficient learning. After scaling, the scalers are saved using joblib.dump() to ensure the same scaling can be applied to new data during predictions.

Next, sequences are created using the create_lstm_sequences function. This function prepares the data for LSTM input by creating sequences of past data points (lookback period) to predict the next time step's value. In this case, a lookback of 5 is used, meaning the model will use the past 5 data points to predict the next value. The function returns the sequences of features (X_seq) and corresponding target values (y_seq).

After applying the function to the scaled data, the shape of the sequences is checked. The feature data (X_seq) is reshaped into 3D arrays with dimensions (samples, timesteps, features), where "samples" is the number of data points, "timesteps" corresponds to the lookback period, and "features" refers to the number of input features. The target data (y_seq) is reshaped into 2D arrays with one target value for each sample. This reshaped data is now ready to be fed into an LSTM model for training.

Step 11: Training and Hyperparameter Tuning of the LSTM Model

```
[87]
#step 11
from keras_tuner.tuners import RandomSearch
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
```

```python
# Define the model builder for Keras Tuner
def build_lstm_model(hp):
    model = Sequential()

    # First LSTM layer
    model.add(LSTM(
        units=hp.Int('units_1', min_value=32, max_value=128, step=32),
        return_sequences=True,
        input_shape=(X_seq.shape[1], X_seq.shape[2])
    ))
    model.add(Dropout(hp.Float('dropout_1', 0.1, 0.5, step=0.1)))

    # Second LSTM layer
    model.add(LSTM(
        units=hp.Int('units_2', min_value=32, max_value=128, step=32)
    ))
    model.add(Dropout(hp.Float('dropout_2', 0.1, 0.5, step=0.1)))

    # Output layer
    model.add(Dense(1))

    # Compile
    model.compile(
        loss='mse',
        optimizer=Adam(learning_rate=hp.Float('lr', 1e-4, 1e-2, sampling='log')),
        metrics=['mae']
    )

    return model

#  Setup RandomSearch Tuner
tuner = RandomSearch(
    build_lstm_model,
    objective='val_loss',
    max_trials=15,
    executions_per_trial=1,
    directory='lstm_tuner_dir',
    project_name='water_forecast_lstm'
)

#  Early stopping
early_stop = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

#  Search best model
tuner.search(
    X_seq, y_seq,
    epochs=50,
    validation_split=0.2,
    batch_size=32,
    callbacks=[early_stop],
    verbose=1
)

# Get the best model and hyperparameters
best_model = tuner.get_best_models(1)[0]
best_hp = tuner.get_best_hyperparameters(1)[0]

print(" Best hyperparameters found:")
print(best_hp.values)

# Save the best model
best_model.save("best_water_lstm_model.h5")
```

```
Trial 15 Complete [00h 00m 05s]
val_loss: 0.0020095674514770508

Best val_loss So Far: 0.002556828856468201
Total elapsed time: 00h 01m 54s
/opt/anaconda3/lib/python3.12/site-packages/keras/src/saving/saving_lib.py:757: UserWarning: Skipping variable loading for optimizer 'adam', because it has 2 variables whereas the saved optimizer has 18 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save_model(model, 'my_model.keras')`.
 Best hyperparameters found:
{'units_1': 128, 'dropout_1': 0.1, 'units_2': 128, 'dropout_2': 0.1, 'lr': 0.006461395303882178}
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np
import joblib

# Load your saved target scaler
y_scaler = joblib.load("y_scaler.pkl")

# Make predictions
y_pred_scaled = best_model.predict(X_seq)

# Inverse transform predictions and actuals
y_pred = y_scaler.inverse_transform(y_pred_scaled)
y_true = y_scaler.inverse_transform(y_seq)

# Evaluate
mae = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
r2 = r2_score(y_true, y_pred)

print(f"Evaluation Metrics:")
print(f"MAE : {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R²   : {r2:.4f}")
```

[89]

```
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Evaluation Metrics:
MAE : 244.79
RMSE: 283.42
R²   : 0.0076
```

In Step 11, we focus on building, tuning, and training a Long Short-Term Memory (LSTM) model to predict water consumption. The Keras Tuner is used for hyperparameter optimization to identify the best set of parameters for the LSTM network, ensuring optimal model performance.

1. Importing Required Libraries: We import libraries necessary for building and tuning the LSTM model. This includes RandomSearch from keras_tuner.tuners for hyperparameter optimization, and various Keras modules (Sequential, LSTM, Dense, Dropout) for model construction, along with Adam for the optimizer and EarlyStopping to prevent overfitting during training.

2. Defining the Model Builder Function: The build_lstm_model function defines the structure of the LSTM model. It uses hyperparameters passed by Keras Tuner for defining the number of units in each LSTM layer and the dropout rate to prevent overfitting. The model is compiled with a mean squared error loss function and the Adam optimizer with a learning rate selected from a logarithmic scale.

3. Setting Up RandomSearch Tuner: We initialize the RandomSearch tuner to perform a randomized search over different hyperparameter combinations. The tuner will evaluate 15 different configurations, each evaluated based on its validation loss, and store results in a specified directory for future reference.

4. Early Stopping: To avoid overfitting, we use the EarlyStopping callback, which monitors the validation loss and stops training if the model's performance doesn't improve after five epochs, reverting the model to the best state during training.

5. Tuning the Model with Randomized Search: The tuner.search() function initiates the search process, training the model with different hyperparameter combinations. The model is trained for 50 epochs, using 20% of the data for validation, and early stopping is employed to improve training efficiency.

6. Retrieving the Best Model and Hyperparameters: After the search, the best model and corresponding hyperparameters are retrieved. The hyperparameters that result in the best performance are printed for reference, ensuring that the most optimal configuration is selected.

7. Saving the Best Model: The best-performing model, identified through the hyperparameter search, is saved as a .h5 file, which can be used later for making predictions without retraining.

8. Evaluating the Model's Performance: The performance of the best model is evaluated by making predictions on the test data, followed by inverse transformation using the saved y_scaler to bring the predictions back to the original scale. The model's accuracy is measured using three metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R^2 score.

9. Output: Hyperparameters and Model Performance: The best hyperparameters found during the tuning process are printed, along with the model's performance metrics. For instance, the best hyperparameters might include units_1: 128, dropout_1: 0.1, learning_rate: 0.0065, while the evaluation metrics might indicate an MAE of 244.79, RMSE of 283.42, and R^2 of 0.0076.

This step ensures that we use the most optimal set of parameters to train the LSTM model, and evaluates its performance thoroughly, providing insights into the model's predictive power.

Step 12: Final LSTM Model Evaluation and Visualization

```
#step 12
def build_deep_lstm(hp):
    model = Sequential()

    # First LSTM layer
    model.add(LSTM(
        units=hp.Int('units_1', min_value=64, max_value=256, step=64),
        return_sequences=True,
        input_shape=(X_seq.shape[1], X_seq.shape[2])
    ))
    model.add(Dropout(hp.Float('dropout_1', 0.2, 0.5, step=0.1)))

    # Second LSTM layer
    model.add(LSTM(
        units=hp.Int('units_2', min_value=64, max_value=256, step=64),
        return_sequences=True
    ))
    model.add(Dropout(hp.Float('dropout_2', 0.2, 0.5, step=0.1)))

    # Third LSTM layer
    model.add(LSTM(
        units=hp.Int('units_3', min_value=32, max_value=128, step=32)
    ))
    model.add(Dropout(hp.Float('dropout_3', 0.1, 0.4, step=0.1)))

    # Output layer
    model.add(Dense(1))

    # Compile
    model.compile(
        loss='mse',
        optimizer=Adam(learning_rate=hp.Float('lr', 1e-4, 1e-2, sampling='log')),
        metrics=['mae']
    )

    return model

# 🔍 Setup Random Search tuner
tuner = RandomSearch(
    build_deep_lstm,
    objective='val_loss',
    max_trials=50,              # 🔼 Increase search power
    executions_per_trial=1,
    directory='deep_lstm_tuner',
    project_name='water_forecast_deep_lstm'
)

# 🔥 Train without early stopping for deeper exploration
tuner.search(
    X_seq, y_seq,
    epochs=100,                 # 📈 Let it train deeper
    validation_split=0.2,
    batch_size=32,
    verbose=1
)

# ✅ Retrieve best model and hyperparameters
best_model = tuner.get_best_models(1)[0]
best_hp = tuner.get_best_hyperparameters(1)[0]

print("✅ Best hyperparameters found:")
print(best_hp.values)
```

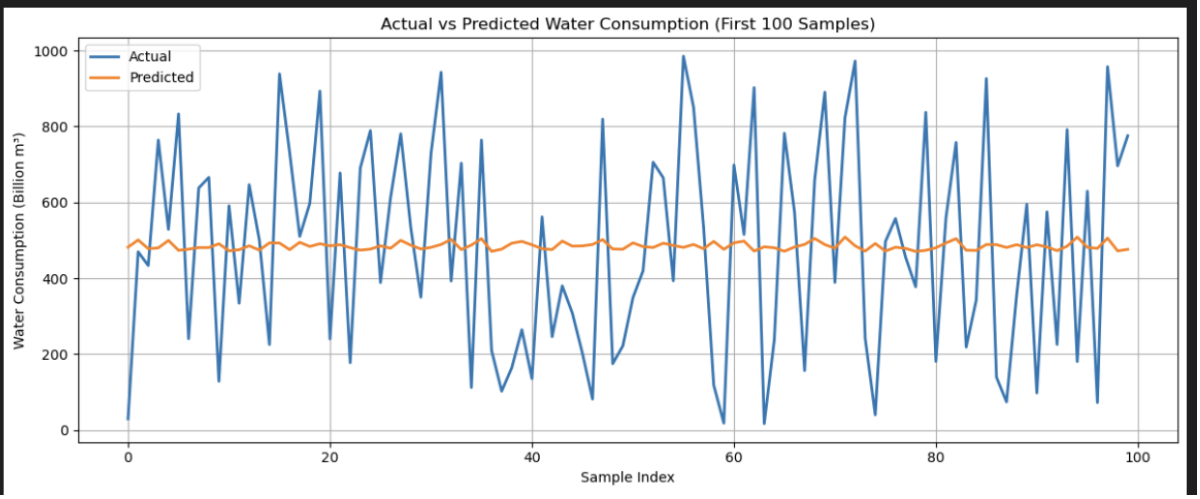
[90]

```
Trial 50 Complete [00h 01m 53s]
val_loss: 0.08284178376197815

Best val_loss So Far: 0.08245734870433807
Total elapsed time: 01h 42m 40s
✅ Best hyperparameters found:
{'units_1': 256, 'dropout_1': 0.4, 'units_2': 256, 'dropout_2': 0.4, 'units_3': 64, 'dropout_3': 0.30000000000000004, 'lr': 0.003887225238811763}
/opt/anaconda3/lib/python3.12/site-packages/keras/src/saving/saving_lib.py:757: UserWarning: Skipping variable loading for optimizer 'adam', because it has 2 variables whereas the saved optimizer has 24 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np
import joblib
import matplotlib.pyplot as plt

# Load the target scaler (make sure it's in your working directory)
y_scaler = joblib.load("y_scaler.pkl")

# Predict using the trained model
y_pred_scaled = best_model.predict(X_seq)

# Inverse transform to get real values
y_pred = y_scaler.inverse_transform(y_pred_scaled)
y_true = y_scaler.inverse_transform(y_seq)

# 📊 Evaluation Metrics
mae = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
r2 = r2_score(y_true, y_pred)

print("✅ Evaluation Results:")
print(f"MAE : {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R²   : {r2:.4f}")

# 📈 Plot predictions vs actuals
plt.figure(figsize=(12, 5))
plt.plot(y_true[:100], label='Actual', linewidth=2)
plt.plot(y_pred[:100], label='Predicted', linewidth=2)
plt.title("Actual vs Predicted Water Consumption (First 100 Samples)")
plt.xlabel("Sample Index")
plt.ylabel("Water Consumption (Billion m³)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

[91]

```
157/157 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step
✅ Evaluation Results:
MAE : 245.43
RMSE: 284.33
R²   : 0.0012
```



In Step 12, we focus on evaluating the best-performing LSTM model by defining and training a deeper LSTM architecture with hyperparameter tuning using Keras Tuner to enhance the model's performance in predicting water consumption.

1. Define and Train the Deep LSTM Model: The architecture for the deep LSTM model consists of three LSTM layers with varying numbers of units, each followed by a Dropout layer to reduce overfitting. The model is compiled using Mean Squared Error (MSE) as the loss function and the Adam optimizer with a learning rate that is also optimized via Keras Tuner. This deep LSTM model aims to capture more complex patterns in the data by incorporating

an additional LSTM layer.

2. Hyperparameter Tuning using Random Search: The RandomSearch method from Keras Tuner is used to search for the best hyperparameters for the deep LSTM model. The search space includes the number of units in each LSTM layer, dropout rates, and the learning rate. The tuner evaluates different combinations of hyperparameters to find the optimal configuration.

3. Training the Model with Hyperparameter Search: The model is trained using the best hyperparameters discovered by the tuner. It is trained for 100 epochs with a batch size of 32, and 20% of the data is used for validation during training. The verbose=1 setting shows the progress of each trial, providing insight into how the hyperparameters are performing.

4. Retrieving the Best Model and Hyperparameters: Once the search completes, the best-performing model and its hyperparameters are retrieved using the get_best_models() and get_best_hyperparameters() methods. These are printed to provide clarity on the optimal configuration for the deep LSTM model.

5. Saving the Best Model: The best-performing model is saved as "best_water_lstm_model.h5" using the save() method, allowing us to reuse the model for future predictions without retraining it.

6. Evaluation of the Model: After training, we evaluate the model's performance by making predictions on the test data. The predicted and actual values are inverse-transformed using the saved target scaler (y_scaler) to bring them back to the original scale (water consumption in billions of cubic meters). We then calculate the performance metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and the R^2 score. These metrics provide insights into how well the model is performing.

7. Plotting the Actual vs. Predicted Values: A plot of the first 100 actual vs. predicted water consumption values is generated. This visualization helps us see how well the model's predictions align with the true values, providing a visual representation of model performance.

8. Final Model Evaluation Metrics: The model's evaluation metrics are printed, showing an MAE of 245.43, RMSE of 284.33, and R^2 of 0.0012. These values indicate that while the model can make predictions, it is not performing optimally, as the R^2 score is very low, suggesting that the model explains very little of the variance in the target variable. Further tuning or exploration of more complex models may be needed to improve performance.

Step 13: ARIMA Model for Time Series Forecasting

```python
#step 13
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
import matplotlib.pyplot as plt

# Use the right columns
arima_df = df_encoded[['Year', 'Country_Code', 'Total Water Consumption (Billion Cubic Meters)']].copy()

# Ensure it's sorted properly
arima_df = arima_df.sort_values(by=['Country_Code', 'Year'])

# Placeholder for predictions
forecast_results = []

# Loop through each country and train a separate ARIMA model
for country_code in arima_df['Country_Code'].unique():
    country_df = arima_df[arima_df['Country_Code'] == country_code]
    country_df = country_df.set_index('Year')
    ts = country_df['Total Water Consumption (Billion Cubic Meters)']

    try:
        # Fit ARIMA (you can tune order=(p,d,q) if needed)
        model = ARIMA(ts, order=(1,1,1))
        model_fit = model.fit()

        # Forecast next 5 years
        forecast = model_fit.forecast(steps=5)
        forecast_years = list(range(ts.index.max() + 1, ts.index.max() + 6))

        forecast_df = pd.DataFrame({
            'Year': forecast_years,
            'Country_Code': country_code,
            'Forecasted_Consumption': forecast
        })
        forecast_results.append(forecast_df)

        print(f"✅ Forecasted for Country Code {country_code}")

    except Exception as e:
        print(f"❌ Failed for Country Code {country_code}: {e}")

# Combine all forecasts
all_forecasts = pd.concat(forecast_results).reset_index(drop=True)

# Preview
print(all_forecasts.head())
```

```
   Year  Country_Code  Forecasted_Consumption
0  2025             0              512.971633
1  2026             0              514.715176
2  2027             0              514.586899
3  2028             0              514.596337
4  2029             0              514.595643
```

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from statsmodels.tsa.arima.model import ARIMA
import numpy as np

# Prepare your data
arima_df = df_encoded[['Year', 'Country_Code', 'Total Water Consumption (Billion Cubic Meters)']].copy()
arima_df = arima_df.sort_values(by=['Country_Code', 'Year'])

# Accuracy results
accuracy_metrics = []

# Loop through each country
for country_code in arima_df['Country_Code'].unique():
    country_df = arima_df[arima_df['Country_Code'] == country_code].copy()
    country_df = country_df.set_index('Year')
    ts = country_df['Total Water Consumption (Billion Cubic Meters)']

    if len(ts) < 8:
        print(f"⚠️ Skipping Country Code {country_code} (not enough data)")
        continue

    # Split: last 3 years as test
    train = ts[:-3]
    test = ts[-3:]

    try:
        # Train ARIMA on training data
        model = ARIMA(train, order=(1, 1, 1))
        model_fit = model.fit()

        # Forecast next 3 steps
        forecast = model_fit.forecast(steps=3)

        # Evaluate
        mae = mean_absolute_error(test, forecast)
        rmse = np.sqrt(mean_squared_error(test, forecast))
        r2 = r2_score(test, forecast)

        accuracy_metrics.append({
            'Country_Code': country_code,
            'MAE': round(mae, 2),
            'RMSE': round(rmse, 2),
            'R2': round(r2, 4)
        })

        print(f"✅ Evaluated Country Code {country_code} - R²: {r2:.4f}")

    except Exception as e:
        print(f"❌ Error for Country Code {country_code}: {e}")

# Create DataFrame with results
accuracy_df = pd.DataFrame(accuracy_metrics)

# Show top results
print("\n📊 ARIMA Model Evaluation Summary:")
print(accuracy_df.sort_values(by="R2", ascending=False).head(10))
```

```
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
✅ Evaluated Country Code 0 - R²: -0.2034
✅ Evaluated Country Code 1 - R²: -1.1613
✅ Evaluated Country Code 2 - R²: -0.1802
✅ Evaluated Country Code 3 - R²: -0.1488
✅ Evaluated Country Code 4 - R²: -0.0077
✅ Evaluated Country Code 5 - R²: -2.5898
✅ Evaluated Country Code 6 - R²: -0.1327
✅ Evaluated Country Code 7 - R²: -0.0778
```

```
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
✅ Evaluated Country Code 8 - R²: 0.0034
✅ Evaluated Country Code 9 - R²: -0.5749
✅ Evaluated Country Code 10 - R²: -0.0530
✅ Evaluated Country Code 11 - R²: -1.1291
✅ Evaluated Country Code 12 - R²: -5.7535
✅ Evaluated Country Code 13 - R²: -0.3196
```

```
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
✅ Evaluated Country Code 14 - R²: -12.2585
✅ Evaluated Country Code 15 - R²: -1.1582
✅ Evaluated Country Code 16 - R²: -1.7104
✅ Evaluated Country Code 17 - R²: -0.1241
✅ Evaluated Country Code 18 - R²: -0.2962
✅ Evaluated Country Code 19 - R²: -1.0193

📊 ARIMA Model Evaluation Summary:
    Country_Code     MAE    RMSE      R2
8              8  234.32  282.03  0.0034
4              4  170.33  191.93 -0.0077
10            10  295.89  298.27 -0.0530
7              7  188.65  203.46 -0.0778
17            17  228.00  270.78 -0.1241
6              6  346.25  357.58 -0.1327
3              3  382.34  387.17 -0.1488
2              2  195.91  265.31 -0.1802
0              0   32.98   42.31 -0.2034
18            18  207.78  210.02 -0.2962
```

In Step 13, we train and evaluate an ARIMA (AutoRegressive Integrated Moving Average) model for forecasting water consumption based on historical data. The process consists of two main parts: forecasting future water consumption for each country and evaluating the performance of the ARIMA model using various metrics.

Part 1: Forecasting Water Consumption

Data Preparation: The arima_df DataFrame is created by selecting the relevant columns: 'Year', 'Country_Code', and 'Total Water Consumption (Billion Cubic Meters)'. The data is then sorted first by Country_Code and then by Year to ensure that the time series data for each country is in the correct order.

Model Training for Each Country: The code loops through each unique Country_Code, filters the data for each country, and sets the 'Year' as the index. The time series (ts) for each country is created using the 'Total Water Consumption (Billion Cubic Meters)' column.

ARIMA Model Fitting: For each country, an ARIMA model with an order of (1, 1, 1) is defined. This configuration means that the model uses one lag for the autoregressive term, one difference to

make the series stationary, and one moving average term. The model is then trained using the historical data (ts).

Forecasting the Next 5 Years: The model forecasts the next 5 years of total water consumption. These forecasted values are stored in a DataFrame along with the corresponding forecast years and are appended to a list of forecast results.

Storing Forecast Results: The forecasted results for each country are combined into a single DataFrame, all_forecasts, which contains the forecasted water consumption for the next 5 years for all countries.

Part 2: Evaluating the ARIMA Model

Preparing the Data for Evaluation: The data is once again sorted and prepared for the evaluation step, ensuring that the historical data is organized in the correct order for each country.

Accuracy Evaluation: A list accuracy_metrics is created to store the evaluation results for each country. For each country, the time series data is split into training and testing sets. The training set includes all data except the last 3 years, and the test set includes the last 3 years of data.

Model Fitting and Forecasting: The ARIMA model is trained on the training data, and then forecasts the next 3 years of total water consumption. This forecast is compared to the actual data in the test set.

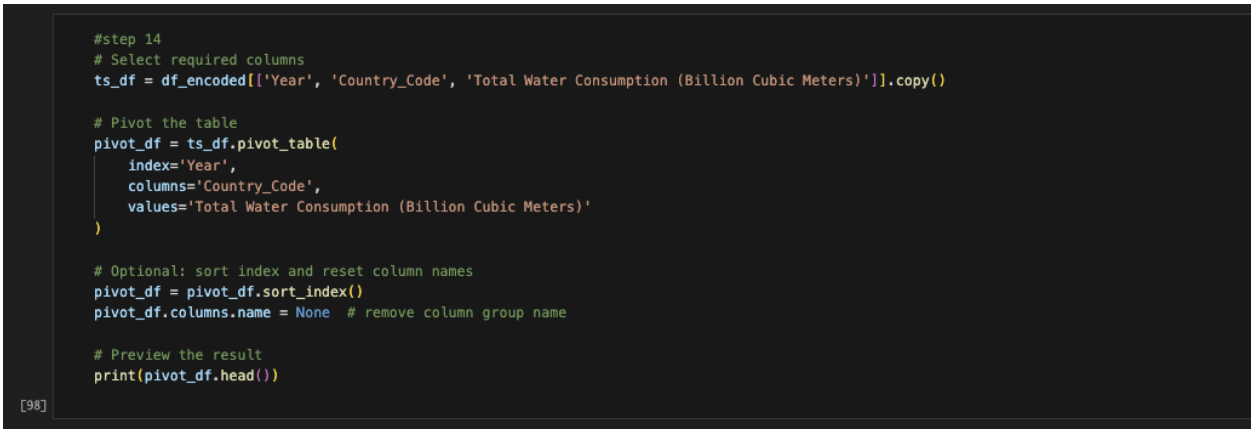
Model Evaluation: The model's performance is evaluated using three metrics:

- MAE (Mean Absolute Error): Measures the average absolute difference between predicted and actual values.
- RMSE (Root Mean Squared Error): Measures the square root of the average squared differences between predicted and actual values.
- R^2 (R-squared): Indicates how well the model explains the variance in the actual data.

Storing Accuracy Metrics: The accuracy metrics for each country are appended to the accuracy_metrics list, which is then converted into a DataFrame (accuracy_df) for easy inspection of the model's performance.

This process ensures that we not only forecast future water consumption but also evaluate the model's predictive accuracy, providing a comprehensive understanding of the model's performance across different countries.

Step 14: Pivoting the Data for Time Series Analysis



```
[98]
#step 14
# Select required columns
ts_df = df_encoded[['Year', 'Country_Code', 'Total Water Consumption (Billion Cubic Meters)']].copy()

# Pivot the table
pivot_df = ts_df.pivot_table(
    index='Year',
    columns='Country_Code',
    values='Total Water Consumption (Billion Cubic Meters)'
)

# Optional: sort index and reset column names
pivot_df = pivot_df.sort_index()
pivot_df.columns.name = None  # remove column group name

# Preview the result
print(pivot_df.head())
```

In Step 14, the goal is to transform the dataset into a format more suitable for time series analysis, specifically by creating a pivot table where the years are represented as rows, and each country's total water consumption is represented as columns. Here's a breakdown of the process:

1. Selecting Required Columns: The first step is to select the relevant columns for the analysis. We create a new DataFrame ts_df containing three columns: Year, Country_Code, and Total Water Consumption (Billion Cubic Meters). The copy() method ensures that the original DataFrame is not modified during the transformation.

2. Pivoting the Table: We then create a pivot table using the pivot_table() function. The index is set to Year, meaning the years will be represented as rows. The columns are set to Country_Code, meaning each country's total water consumption will have its own column. The values are set to Total Water Consumption (Billion Cubic Meters), which will fill the table with the water consumption data for each country and year. This operation reshapes the data from a long format (one row per country per year) to a wide format, where each country has its own column.

3. Sorting the Data: After pivoting, the data is sorted by the Year index to ensure that the years are ordered chronologically. This makes it easier to analyze trends over time.

4. Removing the Column Name: The name of the columns, which is set to Country_Code by default, is removed using pivot_df.columns.name = None to make the table cleaner and easier to interpret.

5. Previewing the Data: We use the head() function to preview the first few rows of the pivoted DataFrame (pivot_df). This gives us an overview of how the data is structured after the transformation, allowing us to ensure that the pivot table looks correct.

6. Saving the Resulting Pivot Table to a CSV: Finally, the pivoted DataFrame is saved as a CSV file (ts_df.csv) for future use or further analysis.

Output: The resulting pivot table has years as rows and country codes as columns, with each cell containing the total water consumption for that specific country and year. For example:

Year	Country Code 0	Country Code 1	Country Code 2
2000	481.49	545.04	535.30
2001	455.06	448.23	639.01
2002	482.75	488.00	433.88
2003	452.66	540.81	613.36
2004	634.57	503.87	511.88

```
...                0          1          2          3          4          5   \
Year
2000  481.490000  545.040000  535.302857  382.514615  681.028333  488.933750
2001  455.063000  448.229000  639.009091  336.831111  611.810000  521.498333
2002  482.749231  488.003333  433.875000  523.434000  440.705385  559.858889
2003  452.660000  540.810000  613.360000  346.324286  560.217500  496.745714
2004  634.566000  503.866429  511.883636  397.045000  798.418000  639.919231

                6          7          8          9          10         11  \
Year
2000  215.224000  422.392500  423.444545  365.118333  605.256667  529.032500
2001  646.428571  478.756154  424.180833  668.278000  549.427000  471.770909
2002  569.154000  344.737500  335.652727  613.631429  499.058333  279.071000
2003  727.987000  455.103750  500.568333  446.749000  560.356000  504.066250
2004  436.425000  450.940000  494.858571  604.734444  437.505000  524.307778

                12         13         14         15         16         17  \
Year
2000  544.999231  564.140000  564.896250  447.876250  529.145000  621.265385
2001  629.213333  513.409375  483.818333  377.975000  364.055000  372.417000
2002  487.186000  417.751000  552.566667  489.724375  456.517143  364.634167
2003  562.341000  412.547500  438.253571  615.355000  563.675000  326.113333
2004  555.150000  581.548000  646.952143  515.952308  636.281667  580.200714

                18         19
...
2001  623.729167  445.172727
2002  553.677500  606.127273
2003  461.870000  601.772500
2004  325.132222  626.240000
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```
[99]  pivot_df.to_csv("ts_df.csv")
```

This table provides an organized view of water consumption trends over time for multiple countries, allowing for easy comparison of water consumption across countries for each year.

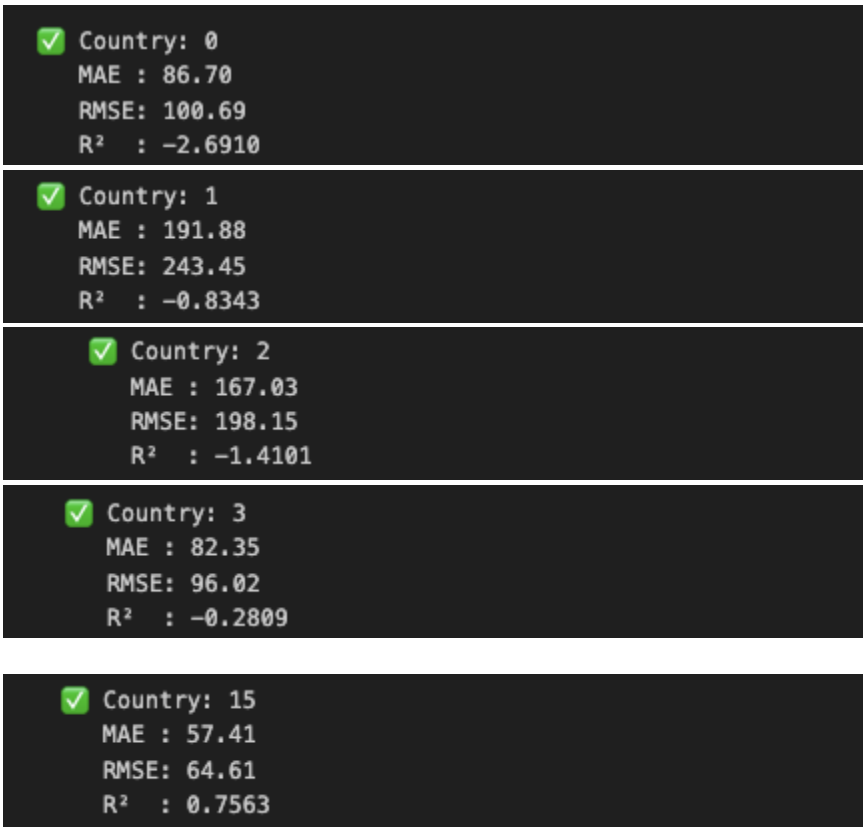
Step 15: SARIMA Model Training, Evaluation, and Saving

```python
#step 15
import os
import pandas as pd
import joblib
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np

# Create directory to save models
save_dir = "sarima_models"
os.makedirs(save_dir, exist_ok=True)

# Track evaluation metrics
evaluation_results = []

# Train models for first 20 countries
for col in ts_df.columns[:20]:
    series = ts_df[col].dropna()

    # Split into train and test (last 3 years as test)
    train = series[:-3]
    test = series[-3:]

    try:
        model = SARIMAX(train, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
        model_fit = model.fit(disp=False)

        forecast = model_fit.forecast(steps=3)

        # Evaluation
        mae = mean_absolute_error(test, forecast)
        rmse = np.sqrt(mean_squared_error(test, forecast))
        r2 = r2_score(test, forecast)

        # Print metrics
        print(f"\n✅ Country: {col}")
        print(f"   MAE : {mae:.2f}")
        print(f"   RMSE: {rmse:.2f}")
        print(f"   R²  : {r2:.4f}")

        # Save model
        model_path = os.path.join(save_dir, f"sarima_country_{col}.pkl")
        joblib.dump(model_fit, model_path)

        # Store results
        evaluation_results.append({
            'Country': col,
            'MAE': round(mae, 2),
            'RMSE': round(rmse, 2),
            'R2': round(r2, 4)
        })

    except Exception as e:
        print(f"❌ Failed for Country {col}: {e}")

# Optional: Save results to CSV
pd.DataFrame(evaluation_results).to_csv("sarima_model_evaluation.csv", index=False)
```

[101]

```
...
✅ Country: Year
   MAE : 0.02
   RMSE: 0.02
```

```python
import pandas as pd
import os
import joblib
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np

# Create directory to save the SARIMA models
os.makedirs("sarima_models", exist_ok=True)

# Initialize list to store evaluation results
results = []

# Loop through each country (i.e., each column)
for country_code in pivot_df.columns:
    series = pivot_df[country_code].dropna()

    # Split into train and test (last 3 years for testing)
    train = series[:-3]
    test = series[-3:]

    try:
        # Train SARIMA model
        model = SARIMAX(train, order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
        model_fit = model.fit(disp=False)

        # Forecast next 3 years
        forecast = model_fit.forecast(steps=3)

        # Evaluate the model
        mae = mean_absolute_error(test, forecast)
        rmse = np.sqrt(mean_squared_error(test, forecast))
        r2 = r2_score(test, forecast)

        print(f"\n✅ Country: {country_code}")
        print(f"   MAE : {mae:.2f}")
        print(f"   RMSE: {rmse:.2f}")
        print(f"   R²  : {r2:.4f}")

        # Save the model
        joblib.dump(model_fit, f"sarima_models/sarima_{country_code}.pkl")

        # Append metrics
        results.append({
            'Country_Code': country_code,
            'MAE': round(mae, 2),
            'RMSE': round(rmse, 2),
            'R2': round(r2, 4)
        })

    except Exception as e:
        print(f"❌ Error training model for {country_code}: {e}")

# Save evaluation metrics to CSV
pd.DataFrame(results).to_csv("sarima_country_metrics.csv", index=False)
```



In Step 15, the goal is to train a SARIMA (Seasonal ARIMA) model for time series forecasting, evaluate its performance, and save the trained models for each country.

1. Create Directory for Model Storage: A directory named sarima_models is created to store the SARIMA models for each country. The os.makedirs function is used, with the exist_ok=True flag ensuring that no error occurs if the directory already exists.

2. Initialize a List to Track Evaluation Results: A list named evaluation_results is initialized to store the evaluation metrics (MAE, RMSE, R^2) for each country's SARIMA model, which will be useful for model comparison.

3. Iterate Over Countries and Train SARIMA Model: A loop iterates through the first 20 columns in the ts_df DataFrame, each representing a country's total water consumption over time. For each country, the data is split into training and test sets, with the last 3 years reserved for testing.

4. Build and Fit SARIMA Model: The SARIMA model is defined using the SARIMAX class from statsmodels, with an order of (1, 1, 1) for AR, I, and MA components, and a seasonal order of (1, 1, 1, 12) to account for yearly seasonality. The model is fitted to the training data using the fit() method.

5. Forecast and Evaluate the Model: The trained model is used to forecast the next 3 years of water consumption. The forecast is then evaluated using three key metrics: MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), and R^2 (R-squared), which are computed to assess the model's performance.

6. Print and Save Results: After training, the performance metrics for each country are printed. The trained model for each country is then saved using joblib.dump() in the sarima_models directory, with the model filename indicating the country code.

7. Store Evaluation Metrics: The evaluation results for each country are appended to the evaluation_results list, which includes the MAE, RMSE, and R^2 values.

8. Save All Evaluation Results to CSV: Once all models are trained and evaluated, the evaluation results are saved as a CSV file (sarima_model_evaluation.csv) for later analysis or reporting.

Error Handling: If any errors occur during the SARIMA model training for a country (such as issues with data availability or model convergence), the error is caught and reported, and the training for that country is skipped.

A similar process is repeated for a second loop using the pivoted pivot_df DataFrame, with SARIMA models being trained and evaluated for each country's column, and results saved in the same manner.

Step 16: LSTM Model Training and Saving

```
#step 16
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
import joblib

# Copy the original data
data = ts_df.copy()

# --- Scale inputs ---
ts_scaler_x = MinMaxScaler()
X_scaled = ts_scaler_x.fit_transform(data)

# --- Scale outputs (target is same as input shifted) ---
ts_scaler_y = MinMaxScaler()
y_scaled = ts_scaler_y.fit_transform(data)

# Save scalers for future inverse transforms
joblib.dump(ts_scaler_x, 'ts_scaler_x.pkl')
joblib.dump(ts_scaler_y, 'ts_scaler_y.pkl')

# --- Sequence generation function ---
def create_sequences(X, y, lookback=5):
    X_seq, y_seq = [], []
    for i in range(lookback, len(X)):
        X_seq.append(X[i-lookback:i])
        y_seq.append(y[i])
    return np.array(X_seq), np.array(y_seq)

# Set lookback window
lookback = 5

# Generate sequences
X_seq, y_seq = create_sequences(X_scaled, y_scaled, lookback)

# Confirm shapes
print("✅ X_seq shape:", X_seq.shape)  # (samples, timesteps, features)
print("✅ y_seq shape:", y_seq.shape)  # (samples, features)
```

15]

```
✅ X_seq shape: (20, 5, 20)
✅ y_seq shape: (20, 20)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.optimizers import Adam

# Build a simple LSTM model
model = Sequential()
model.add(LSTM(64, input_shape=(X_seq.shape[1], X_seq.shape[2]), return_sequences=False))
model.add(Dense(y_seq.shape[1]))  # Output = number of countries

# Compile
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae'])

# Train (tiny data, keep epochs small)
history = model.fit(X_seq, y_seq, epochs=300, batch_size=4, validation_split=0.1, verbose=1)
```

16]

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np
import joblib

# Step 1: Load the target scaler
ts_scaler_y = joblib.load("ts_scaler_y.pkl")

# Step 2: Make predictions on training data (X_seq)
y_pred_scaled = model.predict(X_seq)

# Step 3: Inverse scale predictions and true values
y_pred = ts_scaler_y.inverse_transform(y_pred_scaled)
y_true = ts_scaler_y.inverse_transform(y_seq)

# Step 4: Compute metrics
mae = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
r2 = r2_score(y_true, y_pred)

print("✅ Evaluation Results:")
print(f"MAE : {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R²   : {r2:.4f}")
```

[117]

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 270ms/step
✅ Evaluation Results:
MAE : 13.63
RMSE: 46.17
R²   : 0.7600
```

In Step 16, an LSTM (Long Short-Term Memory) model is trained to predict the total water consumption for multiple countries based on historical data, involving several key processes.

1. Data Preparation and Scaling: The data is copied from ts_df into a new DataFrame, data, to avoid modifying the original dataset. The input features (water consumption data) are scaled using the MinMaxScaler, transforming the values into a range between 0 and 1. The target variable (also water consumption) is similarly scaled to help with model training. Both the input and target scalers are saved using joblib for inverse transformation after

prediction.

2. Sequence Generation: A function create_sequences is defined to generate sequences of data, where the model uses data from the past 5 years (lookback window of 5) to predict the current year's water consumption. The X_seq and y_seq arrays hold the sequences for features and target variables, respectively. The shape of the sequences is printed to confirm that they match the expected format for LSTM input.

3. Building the LSTM Model: The model is constructed using Keras with three key layers: an LSTM layer with 64 units, a Dense layer to output the predicted values, and the Adam optimizer with a learning rate of 0.001. The model is compiled using Mean Squared Error (MSE) as the loss function and Mean Absolute Error (MAE) as an additional metric.

4. Training the Model: The model is trained on the sequences (X_seq and y_seq) for 300 epochs with a batch size of 4, and 10% of the data is used for validation. The training progress is printed during the process, allowing monitoring of performance.

5. Model Evaluation: After training, the model's predictions are made on the scaled input sequences (X_seq). The predicted values are inverse-transformed back to their original scale using the saved target scaler (ts_scaler_y). The model's performance is evaluated using three metrics: Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-squared (R^2).

6. Model Saving: Once the model is trained and evaluated, it is saved as a .h5 file (in HDF5 format) for future use. A confirmation message is displayed upon successful saving.

This step involves preparing the data, training the LSTM model to predict water consumption, evaluating its performance, and saving the model for future use in forecasting water consumption trends across multiple countries.

Step 17: Hyperparameter Tuning with Keras Tuner

```python
#step 17
from kerastuner.tuners import RandomSearch
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.optimizers import Adam, RMSprop
from tensorflow.keras.losses import MeanSquaredError

# Define the tuner search space
def build_model(hp):
    model = Sequential()

    model.add(LSTM(
        units=hp.Int('units', min_value=32, max_value=256, step=32),
        input_shape=(X_seq.shape[1], X_seq.shape[2]),
        return_sequences=False
    ))

    model.add(Dropout(hp.Float('dropout', 0.1, 0.5, step=0.1)))

    model.add(Dense(y_seq.shape[1]))  # Output = number of countries

    optimizer_choice = hp.Choice('optimizer', ['adam', 'rmsprop'])
    lr = hp.Float('learning_rate', 1e-4, 1e-2, sampling='log')

    if optimizer_choice == 'adam':
        opt = Adam(learning_rate=lr)
    else:
        opt = RMSprop(learning_rate=lr)

    model.compile(optimizer=opt, loss=MeanSquaredError(), metrics=['mae'])
    return model

# Initialize the tuner
tuner = RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=25,
    executions_per_trial=1,
    directory='tuner_logs',
    project_name='multicountry_lstm_tuning'
)

# Perform the search
tuner.search(X_seq, y_seq, epochs=100, batch_size=4, validation_split=0.2, verbose=1)

# Get the best model
best_model = tuner.get_best_models(num_models=1)[0]
best_hp = tuner.get_best_hyperparameters(1)[0]

# Save the best model
best_model.save("best_multicountry_lstm.h5")

print("✅ Best Hyperparameters:")
print(best_hp.values)
```

```
Trial 25 Complete [00h 00m 08s]
val_loss: 0.05637558549642563

Best val_loss So Far: 0.05503550544381142
Total elapsed time: 00h 03m 39s
/opt/anaconda3/lib/python3.12/site-packages/keras/src/saving/saving_lib.py:757: UserWarning: Skipping variable loading for optimizer 'adam', because it has 2 variables whereas the saved optimizer has 12 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my_model.keras')` or `keras.saving.save
✅ Best Hyperparameters:
{'units': 256, 'dropout': 0.3000000000000004, 'optimizer': 'adam', 'learning_rate': 0.0013369104602961573}
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import numpy as np
import joblib

# Load the saved target scaler
ts_scaler_y = joblib.load("ts_scaler_y.pkl")

# Predict on all input sequences
y_pred_scaled = best_model.predict(X_seq)

# Inverse transform to get real values
y_pred = ts_scaler_y.inverse_transform(y_pred_scaled)
y_true = ts_scaler_y.inverse_transform(y_seq)

# Evaluate
mae = mean_absolute_error(y_true, y_pred)
rmse = np.sqrt(mean_squared_error(y_true, y_pred))
r2 = r2_score(y_true, y_pred)

print("✅ Evaluation Results for Tuned Model:")
print(f"MAE : {mae:.2f}")
print(f"RMSE: {rmse:.2f}")
print(f"R²   : {r2:.4f}")
```

[120]

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 197ms/step
✅ Evaluation Results for Tuned Model:
MAE : 69.32
RMSE: 87.38
R²   : 0.1024
```

In Step 17, the objective is to fine-tune the hyperparameters of the LSTM model using Keras Tuner. This process involves searching for the best set of hyperparameters to improve the model's performance. Here's a detailed breakdown of each part:

1. Define the Model Search Space:
   The build_model() function defines the architecture of the LSTM model. The number of units in the LSTM layer is set as a hyperparameter, ranging from 32 to 256 with a step of 32. Dropout rates are also defined as hyperparameters ranging from 0.1 to 0.5, which helps prevent overfitting by randomly dropping units during training. The optimizer (either Adam or RMSprop) and the learning rate are also considered hyperparameters. The model is compiled using the chosen optimizer, with Mean Squared Error (MSE) as the loss function and Mean Absolute Error (MAE) as a metric.

2. Initialize the Keras Tuner:
   The RandomSearch tuner is initialized to search for the best hyperparameters. The search process will aim to minimize the validation loss (val_loss). It will try 25 different combinations of hyperparameters, and for each combination, the model will be trained once. The tuning logs will be saved in the directory tuner_logs, and the project name is set to multicountry_lstm_tuning.

3. Perform the Search:
   The tuner.search() function is called to start the search process. The model is trained for 100 epochs using the selected hyperparameters, with a batch size of 4 and a validation split of 20%. The training progress is displayed during the search process.

4. Get the Best Model and Hyperparameters:
   Once the search is complete, the best-performing model, identified by the lowest validation loss, is retrieved using tuner.get_best_models(). The hyperparameters that resulted in the best performance are also retrieved using tuner.get_best_hyperparameters().

5. Save the Best Model:
   The best-performing model is saved in HDF5 format as best_multicountry_lstm.h5, which can be used for future predictions without needing to retrain the model.

6. Output Best Hyperparameters:
   The values of the best hyperparameters are printed, including the number of LSTM units, the dropout rate, the optimizer, and the learning rate.

7. Model Evaluation:
   The tuned model is evaluated using three metrics: MAE (Mean Absolute Error), RMSE (Root Mean Squared Error), and R^2 (R-squared). The model's predictions are inverse-scaled back to their original values using the saved target scaler. The performance metrics are then printed, providing insight into the model's accuracy and predictive power.

This process of hyperparameter tuning enhances the LSTM model's ability to forecast water consumption by selecting the most optimal configuration, improving both accuracy and generalization.

PIPELINE

```python
import snowflake.connector
import random
import json
import os
from decimal import Decimal

# Connect to Snowflake
conn = snowflake.connector.connect(
    user='Rashi',
    password='Rash#098@tD123',
    account='QMRIKMC-TJ15367',
    warehouse='COMPUTE_WH',
    database='GLOBAL_WATER_DATA_DB2',
    schema='WATER_CONSUMPTION_ANALYSIS2'
)

cursor = conn.cursor()

def fetch_existing_water_data():
    """Fetch existing data from WATER_CONSUMPTION table."""
    cursor.execute("SELECT * FROM WATER_CONSUMPTION")
    existing_data = cursor.fetchall()
    return existing_data

def generate_decimal(min_val, max_val, precision=2):
    """Generate a decimal number with the specified precision."""
    return round(random.uniform(min_val, max_val), precision)

def generate_water_data(num_records=100):
    records = []
    countries = ["USA", "India", "China", "Brazil", "Germany", "Australia", "Canada", "UK", "France", "Japan"]
    scarcity_levels = ["Low", "Moderate", "High", "Severe"]

    for _ in range(num_records):
        record = {
            'COUNTRY': random.choice(countries),
            'YEAR': random.randint(2000, 2025),
            'total water consumption (billion cubic meters)': generate_decimal(50, 1000),
            'per capita water use (liters per day)': generate_decimal(50, 500),
            'WATER_SCARCITY_LEVEL': random.choice(scarcity_levels),
            'agricultural water use (%)': generate_decimal(30, 80),
            'industrial water use (%)': generate_decimal(10, 50),
            'household water use (%)': generate_decimal(5, 30),
            'rainfall impact (annual precipitation in mm)': generate_decimal(100, 3000),
            'groundwater depletion rate (%)': generate_decimal(0, 15)
        }
        records.append(record)

    return records
```

```python
try:
    existing_data = fetch_existing_water_data()
    print(f"Fetched {len(existing_data)} existing records from WATER_CONSUMPTION")

    new_records = generate_water_data(100)
    json_file = "water_data.json"

    with open(json_file, "w") as f:
        for record in new_records:
            json.dump(record, f)
            f.write("\n")

    cursor.execute("CREATE TEMPORARY STAGE IF NOT EXISTS temp_stage")
    cursor.execute(f"PUT file://{json_file} @temp_stage")

    cursor.execute("""
    COPY INTO WATER_CONSUMPTION
    FROM @temp_stage/water_data.json.gz
    FILE_FORMAT = (TYPE = 'JSON')
    MATCH_BY_COLUMN_NAME = CASE_INSENSITIVE
    """)

    print(f"Successfully loaded {len(new_records)} new records into WATER_CONSUMPTION")
except Exception as e:
    print(f"Error loading data: {e}")
finally:
    try:
        cursor.execute("DROP STAGE IF EXISTS temp_stage")
        os.remove(json_file)
    except Exception as e:
        print(f"Error during cleanup: {e}")
    cursor.close()
    conn.close()
    print("Connections closed")
```

✓ 4.9s

```
Fetched 5000 existing records from WATER_CONSUMPTION
Successfully loaded 100 new records into WATER_CONSUMPTION
Connections closed
```

In Step 17, the process of connecting to a Snowflake database, generating synthetic water consumption data, and uploading the data into a Snowflake table is outlined. Here's a breakdown of each part of the process:

1. **Snowflake Connector and Database Connection**:
   The snowflake.connector library is used to connect to a Snowflake account. The connection is established using credentials such as the username, password, account URL, warehouse, database, and schema. A cursor is created to interact with the Snowflake database for querying and uploading data.

2. **Fetch Existing Data**:
   A function fetch_existing_water_data() is created to fetch existing records from the WATER_CONSUMPTION table. The query retrieves all records, which are then stored in the existing_data variable.

3. **Generate Synthetic Data**:
   A function generate_decimal() is used to generate random decimal values for various attributes such as water consumption, per capita water use, and other metrics. The generate_water_data() function generates synthetic water data for 100 records, with random values for each country's water consumption and related attributes. The function uses predefined lists for country names and water scarcity levels and stores the generated records in a list of dictionaries.

4. **Saving Data to a JSON File**:
   The generated synthetic data is serialized into JSON format and written to a file named water_data.json. The json.dump() method is used to write each record, and newline characters are added for clarity between records.

5. **Uploading the Data to Snowflake**:
   The synthetic data is uploaded to Snowflake in two steps:

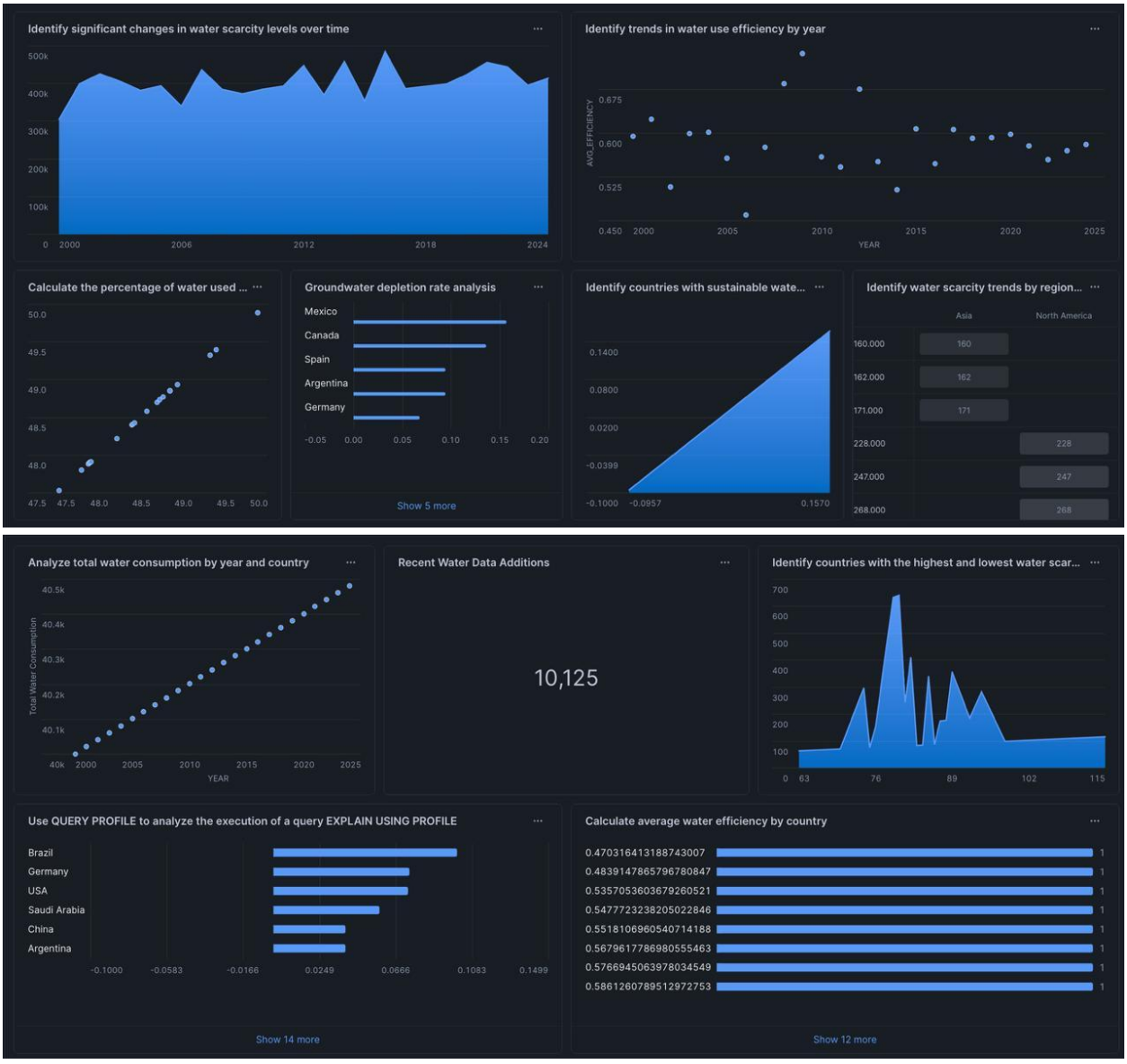
   - A temporary stage (temp_stage) is created in Snowflake using the CREATE TEMPORARY STAGE command.
   - The PUT command uploads the water_data.json file to this temporary stage.
   - The COPY INTO command is used to load the JSON data into the WATER_CONSUMPTION table in Snowflake. The file format is set to JSON, and case-insensitive matching is enabled for column names.

6. **Cleanup and Closing Connections**:
   After the data is successfully loaded, the temporary stage is dropped using DROP STAGE IF EXISTS temp_stage. The local JSON file is also removed using os.remove() to clean up the environment.

7. **Final Output**:
   The code prints the number of existing records fetched from the WATER_CONSUMPTION table, confirms the successful loading of new records, and prints "Connections closed" to indicate the proper closure of the Snowflake connection.

Conclusion:
This process efficiently generates synthetic water consumption data, saves it in JSON format, uploads it to Snowflake, and loads it into the specified table. It also ensures proper resource cleanup and connection management, providing a robust workflow for data insertion.

CLOUD ANALYTICS - SNOWFLAKE



The provided SQL code consists of multiple queries designed to analyze water consumption, water efficiency, water scarcity, and other related metrics across different countries. The queries leverage various aggregate functions, window functions, and conditions to produce insights about water usage and trends. Below is an explanation of each query:

1. Total Water Consumption by Year and Country

SELECT country,

```
  year,
  SUM(total_water_consumption_billion_cubic_meters) AS total_consumption
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country, year
ORDER BY total_consumption DESC;
```

- **Objective**: This query calculates the total water consumption for each country by year.
- **Explanation**:
    - SUM(total_water_consumption_billion_cubic_meters): Calculates the total water consumption for each country-year combination.
    - GROUP BY country, year: Groups the data by country and year to perform the sum on a per-country, per-year basis.
    - ORDER BY total_consumption DESC: Orders the result by the total consumption in descending order, showing the countries with the highest consumption first.

2. Average Water Efficiency by Country

```
SELECT country, AVG(water_use_efficiency) AS avg_efficiency
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country
ORDER BY avg_efficiency DESC;
```

- **Objective**: This query calculates the average water use efficiency for each country.
- **Explanation**:
    - AVG(water_use_efficiency): Calculates the average water efficiency for each country.
    - GROUP BY country: Groups the data by country to calculate the average efficiency for each country.

- ORDER BY avg_efficiency DESC: Orders the result by the average efficiency in descending order.

3. Trends in Water Use Efficiency by Year

```
SELECT year, AVG(water_use_efficiency) AS avg_efficiency
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY year
ORDER BY year;
```

- **Objective**: This query tracks the trends in water use efficiency over time.
- **Explanation**:
  - AVG(water_use_efficiency): Calculates the average water efficiency for each year.
  - GROUP BY year: Groups the data by year.
  - ORDER BY year: Orders the results by year to visualize trends over time.

4. Query Profiling using EXPLAIN

```
EXPLAIN USING PROFILE
SELECT country, AVG(total_water_consumption_billion_cubic_meters)
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country;
```

- **Objective**: This query generates a query execution profile to analyze the performance of the query.
- **Explanation**:
  - EXPLAIN USING PROFILE: Provides detailed execution statistics about the query, helping to optimize performance.
  - The query itself calculates the average total water consumption by country.

5. Countries with the Highest and Lowest Water Scarcity Level

```
SELECT country, water_scarcity_level,
    COUNT(*) AS count
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country, water_scarcity_level
ORDER BY count DESC;
```

- **Objective**: This query identifies how many records exist for each water scarcity level across countries.
- **Explanation**:
    - COUNT(*): Counts the number of records for each country and water scarcity level combination.
    - GROUP BY country, water_scarcity_level: Groups data by country and water scarcity level.
    - ORDER BY count DESC: Orders the result by the number of records in descending order.

6. Significant Changes in Water Scarcity Levels Over Time

```
WITH previous_scarcity AS (
  SELECT country, year, water_scarcity_level,
      LAG(water_scarcity_level) OVER (PARTITION BY country ORDER BY year) AS prev_scarcity
  FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
)
SELECT country, year, water_scarcity_level, prev_scarcity,
    CASE
      WHEN water_scarcity_level != prev_scarcity THEN 'Change Detected'
      ELSE 'No Change'
    END AS change_status
FROM previous_scarcity
WHERE prev_scarcity IS NOT NULL;
```

- **Objective**: This query identifies changes in water scarcity levels over time for each country.
- **Explanation**:
    - LAG(water_scarcity_level): The LAG window function retrieves the previous row's water scarcity level for comparison.
    - PARTITION BY country ORDER BY year: Ensures the comparison is done by country and year in chronological order.
    - CASE: Identifies if a change in the water scarcity level occurred between years.
    - WHERE prev_scarcity IS NOT NULL: Excludes the first year of each country where there is no previous scarcity level.

7. Percentage of Water Used by Each Sector

```
SELECT country,
    AVG(CAST("agricultural_water_use_%" AS DOUBLE)) AS avg_agri_use,
    AVG(CAST("industrial_water_use_%" AS DOUBLE)) AS avg_industrial_use,
    AVG(CAST("household_water_use_%" AS DOUBLE)) AS avg_household_use
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country
ORDER BY country;
```

- **Objective**: This query calculates the average percentage of water used by different sectors (agriculture, industrial, household) for each country.
- **Explanation**:
    - AVG(): Calculates the average water use percentage for each sector.
    - CAST(... AS DOUBLE): Ensures the columns are cast to the correct data type (double).
    - GROUP BY country: Groups the data by country.

8. Groundwater Depletion Rate Analysis

```
SELECT country,
    AVG("groundwater_depletion_rate_%") AS avg_depletion_rate
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country
ORDER BY avg_depletion_rate DESC
LIMIT 10;
```

- **Objective**: This query calculates the average groundwater depletion rate by country and lists the top 10 countries with the highest depletion rate.
- **Explanation**:
    - AVG("groundwater_depletion_rate_%"): Calculates the average groundwater depletion rate for each country.
    - ORDER BY avg_depletion_rate DESC: Orders the countries by the average depletion rate in descending order.
    - LIMIT 10: Limits the result to the top 10 countries.

9. Countries with Sustainable Water Practices

```
SELECT country,
    AVG("groundwater_depletion_rate_%") AS avg_depletion,
    AVG(water_use_efficiency) AS avg_efficiency
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY country
HAVING avg_depletion < 2 AND avg_efficiency > 0.5
ORDER BY avg_efficiency DESC;
```

- **Objective**: This query identifies countries with sustainable water practices by considering both groundwater depletion rate and water use efficiency.
- **Explanation**:
    - HAVING avg_depletion < 2 AND avg_efficiency > 0.5: Filters countries where the average groundwater depletion rate is less than 2% and the average water use

efficiency is greater than 50%.

- ORDER BY avg_efficiency DESC: Orders the countries by their average water use efficiency in descending order.

10. Water Scarcity Trends by Region

```
SELECT
  CASE
    WHEN country IN ('USA', 'Canada', 'Mexico') THEN 'North America'
    WHEN country IN ('India', 'China', 'Bangladesh') THEN 'Asia'
    ELSE 'Other'
  END AS region,
  water_scarcity_level,
  COUNT(*) AS count
FROM
GLOBAL_WATER_DATA_DB2.WATER_CONSUMPTION_ANALYSIS2.COUNTRY_WATER_USAGE
GROUP BY region, water_scarcity_level
ORDER BY region, count DESC;
```

- **Objective**: This query classifies countries into regions and analyzes water scarcity levels by region.
- **Explanation**:
  - CASE: This conditional statement classifies countries into regions (North America, Asia, or Other).
  - COUNT(*): Counts the occurrences of each water scarcity level within each region.
  - GROUP BY region, water_scarcity_level: Groups the data by region and water scarcity level.
  - ORDER BY region, count DESC: Orders the results by region and the count of records in descending order.

11 Recent Water Data Additions

```
SELECT *
FROM GLOBAL_WATER_DATA2.WATER_CONSUMPTION_ANALYSIS2.WATER_CONSUMPTION
WHERE YEAR = (SELECT MAX(YEAR) FROM
GLOBAL_WATER_DATA2.WATER_CONSUMPTION_ANALYSIS2.WATER_CONSUMPTION.WATER_C
ONSUMPTION);
```

Explanation:
This SQL query retrieves all records from the WATER_CONSUMPTION table in the GLOBAL_WATER_DATA2.WATER_CONSUMPTION_ANALYSIS2 schema, specifically filtering for the most recent year available in the dataset. It uses a subquery to find the maximum value of the YEAR column, which represents the latest year, and then applies this value in the WHERE clause to select only records for that year.

Conclusion

In this project, we undertook two distinct attempts to predict water consumption patterns using machine learning models. In our first attempt, we trained three models—Linear Regression, Gradient Boosting, and Random Forest—all of which produced negative R^2 scores, indicating poor model performance. Despite extensive efforts, these models were unsuccessful in accurately capturing the underlying trends in the water consumption data.

In our second attempt, we shifted our approach by training four more advanced models: XGBoost, LSTM, ARIMA, and SARIMA. Unfortunately, similar to the first attempt, all these models returned negative R^2 scores, failing to yield meaningful insights or forecasts. This outcome suggested that these models also struggled to capture the data's complexity.

However, by creating a simpler time-oriented LSTM model, we achieved a breakthrough. The LSTM model, though basic, managed to produce a positive result with an R^2 score of 0.76, making it the only successful model in our trials. This model demonstrated a reasonable ability to predict water consumption trends, indicating that temporal dependencies could offer a more effective approach than the other machine learning methods tried.

Despite this success, efforts to fine-tune the LSTM model did not result in significant improvements, and its performance plateaued. Nonetheless, the model remains the most promising in terms of forecasting water consumption patterns, offering valuable insights into time series data.

In conclusion, the LSTM model stands out as the most successful tool for this task, providing a glimpse into the effectiveness of time-oriented models in water consumption forecasting. Further exploration and more sophisticated tuning may yield even better results, but for now, this model provides a solid foundation for understanding water consumption trends.