Data Processing Script (`data_preprocessing.py`)

This script processes bird monitoring data from two Excel files, performs data cleaning, transformations, and stores the cleaned data into a PostgreSQL database.

Step 1: Load and Clean Data

python
CopyEdit

```
def load_and_clean_data():
    forest_sheets = pd.read_excel('Bird_Monitoring_Data_FOREST.xlsx', sheet_name=None)
    grassland_sheets = pd.read_excel('Bird_Monitoring_Data_GRASSLAND.xlsx', sheet_name=None)
```

- **Explanation:**
  - Reads two Excel files into Python using `pandas`. The `sheet_name=None` option reads all sheets into a dictionary.
  - `forest_sheets` contains all sheets from the 'Bird_Monitoring_Data_FOREST.xlsx' file, and similarly for `grassland_sheets`.

python
CopyEdit

```
   forest_dfs = [df for df in forest_sheets.values() if not df.empty]
    grassland_dfs = [df for df in grassland_sheets.values() if not df.empty]
```

- **Explanation:**
  - Filters out any empty sheets from both datasets, ensuring only non-empty sheets are kept.

python
CopyEdit

```
   forest_df = pd.concat(forest_dfs, ignore_index=True) if forest_dfs else pd.DataFrame()
    grassland_df = pd.concat(grassland_dfs, ignore_index=True) if grassland_dfs else pd.DataFrame()
```

- **Explanation:**
  - Combines the filtered sheets for both the forest and grassland data into single `DataFrame` objects using `pd.concat()`.
  - If there are no non-empty sheets, it creates an empty `DataFrame`.

python
CopyEdit

```
   combined_df = pd.concat([forest_df, grassland_df],
ignore_index=True)
```

- **Explanation:**
  - Merges the two data sets into one `DataFrame`, `combined_df`.

python
CopyEdit

```
   if combined_df.empty:
        raise ValueError("No valid data found in the Excel sheets.")
```

- **Explanation:**
  - Checks if the combined dataset is empty and raises an error if so.

python
CopyEdit

```
   combined_df = combined_df.dropna(subset=['Scientific_Name'])
```

- **Explanation:**
  - Removes any rows where the `Scientific_Name` column has missing (NaN) values.

python
CopyEdit

```
   combined_df['Temperature'] =
combined_df['Temperature'].fillna(combined_df['Temperature'].mean())
    combined_df['Humidity'] =
combined_df['Humidity'].fillna(combined_df['Humidity'].mean())
```

- **Explanation:**
  - Fills any missing values in `Temperature` and `Humidity` columns with the mean value of those columns.

python
CopyEdit

```
combined_df['Date'] = pd.to_datetime(combined_df['Date'])
 combined_df['Year'] = combined_df['Date'].dt.year
 combined_df['Month'] = combined_df['Date'].dt.month
```

- **Explanation:**
  - Converts the `Date` column to a `datetime` format, then extracts the `Year` and `Month` as separate columns.

python
CopyEdit

```
combined_df['Interval_Length'] =
pd.to_numeric(combined_df['Interval_Length'], errors='coerce')
```

- **Explanation:**
  - Converts the `Interval_Length` column to numeric values. Non-convertible values are turned into NaN.

python
CopyEdit

```
def categorize_interval(length):
    if pd.isnull(length):
        return 'Unknown'
    elif 0 <= length <= 2.5:
        return '0-2.5 min'
    elif 2.5 < length <= 5:
        return '2.5-5 min'
    elif 5 < length <= 7.5:
        return '5-7.5 min'
    elif 7.5 < length <= 10:
        return '7.5-10 min'
    else:
        return '10+ min'
 combined_df['Interval_Length'] =
combined_df['Interval_Length'].apply(categorize_interval)
```

- **Explanation:**

- Defines a function to categorize `Interval_Length` into ranges, then applies this function to the `Interval_Length` column to classify each entry.

python
CopyEdit

```
combined_df['Sky'] = combined_df['Sky'].fillna('Unknown')
 combined_df['Wind'] = combined_df['Wind'].fillna('Unknown')
```

- **Explanation:**
  - Fills missing values in the `Sky` and `Wind` columns with `'Unknown'`.

python
CopyEdit

```
columns_to_keep = [
        'Admin_Unit_Code', 'Location_Type', 'Interval_Length',
'ID_Method', 'Year', 'Month', 'Date',
        'Scientific_Name', 'Common_Name', 'Temperature', 'Humidity',
'Distance', 'Flyover_Observed',
        'Sex', 'PIF_Watchlist_Status', 'Regional_Stewardship_Status',
'Disturbance', 'Plot_Name',
        'Sky', 'Wind', 'Observer', 'Visit'
    ]
    combined_df = combined_df[columns_to_keep]
```

- **Explanation:**
  - Filters out unnecessary columns, retaining only the relevant ones specified in the list `columns_to_keep`.

Step 2: Connect to PostgreSQL Database

python
CopyEdit

```
def connect_to_postgres():
    conn = psycopg2.connect(
        dbname="bird_db",
        user="postgres",
        password="Phani@1pk",
        host="localhost",
```

```
        port="5432"
    )
    return conn
```

- **Explanation:**
    - Establishes a connection to the PostgreSQL database using the `psycopg2` library with specified credentials.
    - Returns the connection object.

Step 3: Store Data in PostgreSQL

python
CopyEdit

```
def store_data_in_postgres(df):
    conn = connect_to_postgres()
    cursor = conn.cursor()
```

- **Explanation:**
    - Connects to the PostgreSQL database and creates a cursor for executing queries.

python
CopyEdit

```
   drop_table_query = "DROP TABLE IF EXISTS bird_observations;"
    cursor.execute(drop_table_query)
    conn.commit()
```

- **Explanation:**
    - Drops the existing `bird_observations` table if it exists. This ensures that the table is recreated each time with the latest schema.

python
CopyEdit

```
   create_table_query = """
    CREATE TABLE bird_observations (
        Admin_Unit_Code VARCHAR(50),
        Location_Type VARCHAR(50),
        Interval_Length VARCHAR(50),
```

```
        ID_Method VARCHAR(50),
        Year INT,
        Month INT,
        Date DATE,
        Scientific_Name VARCHAR(100),
        Common_Name VARCHAR(100),
        Temperature FLOAT,
        Humidity FLOAT,
        Distance VARCHAR(50),
        Flyover_Observed BOOLEAN,
        Sex VARCHAR(50),
        PIF_Watchlist_Status BOOLEAN,
        Regional_Stewardship_Status BOOLEAN,
        Disturbance VARCHAR(100),
        Plot_Name VARCHAR(100),
        Sky VARCHAR(50),
        Wind VARCHAR(50),
        Observer VARCHAR(100),
        Visit INT
    );
    """
    cursor.execute(create_table_query)
    conn.commit()
```

- **Explanation:**
    - Creates a new `bird_observations` table in the PostgreSQL database with the specified columns.

python
CopyEdit

```
   for _, row in df.iterrows():
        insert_query = sql.SQL("""INSERT INTO bird_observations (...)
VALUES (%s, %s, %s, ...);""")
        cursor.execute(insert_query, tuple(row))
    conn.commit()
```

- **Explanation:**

- Iterates over each row in the cleaned DataFrame (`df`), inserting the data into the `bird_observations` table.
- Commits the transaction to persist the changes.

Streamlit.py

1. Imports

python
CopyEdit

```
import pandas as pd
import streamlit as st
import psycopg2
import plotly.express as px
```

Here, you're importing:

- `pandas` for data manipulation.
- `streamlit` for creating the web app and user interface.
- `psycopg2` for connecting to a PostgreSQL database.
- `plotly.express` for creating interactive plots and visualizations.

---

2. Database Connection

python
CopyEdit

```
def connect_to_postgres():
    try:
        conn = psycopg2.connect(
            dbname="bird_db",
            user="postgres",  # Replace with actual username
            password="Phani@1pk",  # Replace with actual
password
            host="localhost",
```

```
            port="5432"
        )
        return conn
    except Exception as e:
        st.error(f"Database connection failed: {e}")
        return None
```

This function connects to a PostgreSQL database called `bird_db` using the provided credentials. If successful, it returns the connection object `conn`; if an error occurs, it displays an error message using Streamlit.

---

3. Query Data from PostgreSQL

python
CopyEdit

```
def query_data_from_postgres(query):
    conn = connect_to_postgres()
    if conn is None:
        return pd.DataFrame()  # Return empty DataFrame if
connection fails

    try:
        df = pd.read_sql(query, conn)
        return df
    except Exception as e:
        st.error(f"Failed to execute query: {e}")
        return pd.DataFrame()
    finally:
        conn.close()
```

This function queries data from PostgreSQL:

- It first establishes a connection to the database using `connect_to_postgres()`.

- Then, it executes the SQL query and reads the result into a DataFrame using `pd.read_sql()`.
- In case of failure, it handles errors and closes the connection at the end.

---

4. Exploratory Data Analysis (EDA)

This is where all the actual data analysis and visualizations are done. Let's go over each part.

Temporal Analysis (Observations by Date)

python
CopyEdit

```
if "date" in df.columns:
    df["date"] = pd.to_datetime(df["date"])  # Ensure date is in datetime format
    date_counts = df["date"].value_counts().sort_index()
    fig = px.line(x=date_counts.index, y=date_counts.values, labels={'x': 'Date', 'y': 'Number of Observations'})
    st.plotly_chart(fig)
    st.write("**Summary:** The temporal analysis shows the number of bird observations over time. Peaks in the graph indicate periods of higher bird activity, which may correlate with migration or breeding seasons.")
```

- **Purpose**: Visualize how the number of bird observations changes over time.
- **Steps**:
    - Check if the `date` column exists.
    - Convert the `date` column into a datetime object.
    - Count the occurrences of each date (number of observations).
    - Plot the data as a line graph using Plotly Express (`px.line`).
    - Display the chart and a summary.

Spatial Analysis (Species Diversity by Location Type)

python
CopyEdit

```
if "location_type" in df.columns and "scientific_name" in
df.columns:
    location_diversity =
df.groupby('location_type')['scientific_name'].nunique().reset_i
ndex()
    location_diversity.columns = ['Location Type', 'Number of
Species']
    fig_species_diversity = px.bar(location_diversity,
x='Location Type', y='Number of Species', title='Species
Richness by Location Type', color='Location Type')
    st.plotly_chart(fig_species_diversity)
    st.write("**Summary:** This chart compares species richness
across different location types (e.g., forest, grassland). It
highlights which habitats support the highest biodiversity.")
```

- **Purpose**: Analyze how species richness varies by location type (forest, grassland, etc.).
- **Steps**:
    - Group data by `location_type` and count unique species (`scientific_name`).
    - Create a bar chart showing species richness by location type.
    - Display the plot and a summary.

---

5. Creating a Dashboard

python
CopyEdit

```
def create_dashboard(df):
    st.title("Bird Species Observation Analysis")
    st.write("This dashboard provides insights into bird species
distribution and diversity across forests and grasslands.")
```

- **Purpose**: Set up the title and description for the Streamlit app.
- **Steps**:
    - Display the app title using `st.title()`.

- Show a brief description using `st.write()`.

---

6. Filters for Sidebar

python
CopyEdit

```
st.sidebar.header("Filters")
```

This creates a sidebar header that will hold the filters.

Location Type Filter

python
CopyEdit

```
if "location_type" in df.columns:
    location_type = st.sidebar.selectbox("Select Location Type",
["All", "Forest", "Grassland"])
else:
    st.sidebar.write("Location Type data not available.")
    location_type = "All"
```

- If `location_type` exists in the DataFrame, a select box will allow the user to choose between "All", "Forest", or "Grassland".
- If it doesn't exist, the user is informed, and the filter is set to "All".

Admin Unit Code Filter

python
CopyEdit

```
if "admin_unit_code" in df.columns:
    admin_units = df["admin_unit_code"].unique()
    selected_admin_unit = st.sidebar.selectbox("Select Admin
Unit Code", ["All"] + list(admin_units))
else:
    selected_admin_unit = "All"
```

- If `admin_unit_code` exists, a select box allows the user to filter by different admin units. "All" is the default.
- If it doesn't exist, the filter is set to "All".

Date Range Filter

python
CopyEdit

```
if "date" in df.columns:
    df["date"] = pd.to_datetime(df["date"])
    min_date, max_date = df["date"].min(), df["date"].max()
    date_range = st.sidebar.date_input("Select Date Range",
[min_date, max_date], min_value=min_date, max_value=max_date)
else:
    st.sidebar.write("Date data not available.")
    date_range = None
```

- If `date` exists, a date input allows the user to filter the data based on a selected date range.

---

7. Applying Filters to the Data

python
CopyEdit

```
filtered_df = df
if selected_admin_unit != "All":
    filtered_df = filtered_df[filtered_df['admin_unit_code'] ==
selected_admin_unit]
if location_type and location_type != "All":
    filtered_df = filtered_df[filtered_df['location_type'] ==
location_type]
if date_range and len(date_range) == 2:
    start_date, end_date = pd.to_datetime(date_range[0]),
pd.to_datetime(date_range[1])
    filtered_df = filtered_df[(filtered_df['date'] >=
start_date) & (filtered_df['date'] <= end_date)]
```

- Filters are applied to the DataFrame based on the selected options in the sidebar.

---

8. Displaying Filtered Data

python
CopyEdit

```
st.subheader("Filtered Data")
st.write(filtered_df)
```

- Displays the filtered dataset after the sidebar filters are applied.

---

9. Perform EDA on Filtered Data

python
CopyEdit

```
perform_eda(filtered_df)
```

- This function performs the EDA on the filtered data and generates visualizations.

---

10. Main Function

python
CopyEdit

```
if __name__ == "__main__":
    st.sidebar.title("Data Source")

    # Query data from PostgreSQL
    query = "SELECT * FROM bird_observations;"
    df_from_postgres = query_data_from_postgres(query)

    # Create Streamlit dashboard
    create_dashboard(df_from_postgres)
```

- This section is the entry point of the application.
- It queries the data from PostgreSQL and creates the Streamlit dashboard with the retrieved data.

Summary

This code sets up a Streamlit web app to display bird observation data. It connects to a PostgreSQL database, fetches the data, and performs various analyses like temporal, spatial, and species analysis. Users can filter data by location, date, and other criteria using a sidebar. The app provides an interactive dashboard with visualizations for insights into bird species distribution and biodiversity.