Dataset Extraction

Performed web scraping to gather data about movies from Box Office India, processes it, and save it to an Excel file. Here's a step-by-step breakdown:

1. **Imports**: The script uses BeautifulSoup to parse HTML, Requests to make HTTP requests, Pandas for data manipulation, and OS and UUID modules for file handling and unique ID generation.
2. **Save to Excel Function (`save_movies_to_excel`)**: This function saves the scraped movie data to an Excel file:
    - It first checks if an Excel file (default is "movies3.xlsx") already exists.
    - If it does, it loads the existing data; otherwise, it initializes an empty DataFrame.
    - It then identifies any new columns in the incoming data and adds them to the existing DataFrame.
    - Finally, it appends the new data to the existing data and saves it back to the Excel file.
3. **Fetch Movie Links Function (`fetch_movie_links`)**: This function scrapes movie links from specified pages:
    - For each page in the `page_urls` list, it sends a GET request, parses the HTML, and locates anchor tags (`<a>`) with a class named `anchormob`.
    - Each link found is completed with the domain prefix and added to `all_links`.
4. **Scrape Movie Details**:
    - Each link is processed in a loop to gather information about each movie:
        - Requests the page content and finds sections containing movie details (`div` tags with specific classes).
        - Extracts key details such as movie name, release date, runtime, genre, verdict, and box office figures.
        - Collects weekly data for movie territories by following additional links on the page.
5. **Weekly and Territory Collections**:
    - A helper function, `get_weekly_data`, scrapes weekly collection data for each movie link.
    - For each weekly link, it parses the content and collects data on specific territories, storing it in `weekly_data`.
6. **Production Banner and Cast/Crew Information**:
    - Extracts the production banner (companies) and cast and crew details from specific HTML structures.
    - For the crew, it organizes the data by role and adds each person involved in that role.
7. **Revenue Collection by Region**:
    - This part gathers financial information such as total collections, nett, and share for different regions (both domestic and overseas).
8. **Final Data Compilation**:

- After gathering all necessary data points, each movie's information is stored as a dictionary (`movie_data`), which is appended to the `all_movies` list.

9. **Save to Excel**:
   - After scraping all movies, `save_movies_to_excel` saves the data to an Excel file.

This script effectively scrapes, processes, and structures data from Box Office India, enabling analysis of movie performance across multiple dimensions in an Excel format.

Dataset Preprocessing

Objective

The purpose of this data preprocessing pipeline is to prepare the box office data for analysis by cleaning, formatting, and transforming various fields. This involves parsing dates, converting data types, and applying currency conversion based on historical exchange rates. This document details each preprocessing step executed within the PySpark environment to ensure efficient handling of the dataset.

Environment Setup

1. **Install Java and PySpark**:
   - Java is installed to support PySpark's runtime requirements.
   - PySpark is installed for large-scale data processing and analytics within the environment.

```
# Install Java
!apt-get install openjdk-8-jdk-headless -qq > /dev/null

# Install PySpark
!pip install pyspark
```

2. **Initialize Java Environment**:
   - The JAVA_HOME environment variable is set to configure the Java runtime path.

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
```

Data Loading and Initialization

1. **Start Spark Session**:
    - A Spark session is initialized with memory configurations set to optimize processing.

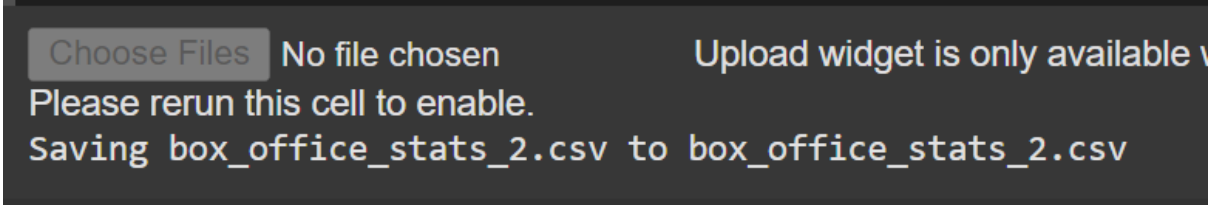
```
# Import necessary PySpark modules
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("Box Office Data Analysis").config("spark.executor.memory", "16g") \
    .config("spark.driver.memory", "16g").getOrCreate()
```

2. **Upload and Load Data**:
    - The primary box office data file (box_office_stats_2.csv) is uploaded, and the dataset is loaded as a PySpark DataFrame with inferred schema to handle different data types automatically.

```
from google.colab import files

# This will open a file upload dialog
uploaded = files.upload()
```



Data Cleaning and Transformation

1. **Parse Date Format**:

- The legacy time parser policy is set to handle variations in date formats.
- The release_date column is parsed from dd-MMM-yy format to a uniform dd/MM/yyyy format.

```
# Step 1: Set the legacy time parser policy
spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")

# Step 2: Parse 'release_date' using the specified format
bo_data_df = bo_data_df.withColumn(
    'release_Date', F.date_format(F.to_date(F.col('release_date'), 'dd-MMM-yy'), 'dd/MM/yyyy')
)
```

2. **Convert runtime to Integer**:
   - The runtime field is processed to retain only the numerical values (in minutes) and is cast to an integer. The original runtime column is then dropped.

```
# Step 3: Convert 'runtime' to integer and drop the original 'runtime' column
bo_data_df = bo_data_df.withColumn(
    'runtime_in_mins', F.split(F.col('runtime'), " ")[0].cast(IntegerType())
).drop('runtime')
```

3. **Clean and Convert Columns to Integer**:
   - A list of numeric columns is defined, including revenue, share, and regional box office data fields.
   - A custom function, clean_and_convert_to_integer, is applied to these columns to remove non-numeric characters and cast each field to integer format.

```
# Step 4: Clean and convert other fields to integer as done previously
def clean_and_convert_to_integer(column_name):
    return F.regexp_replace(column_name, "[^\\d]", "").cast(IntegerType())
```

```
# Apply cleaning function to columns
for column in columns_to_clean:
    bo_data_df = bo_data_df.withColumn(column, clean_and_convert_to_integer(F.col(column)))

bo_data_df.show(10)
```

4. **Convert release_Date to Date Type**:

- The release_Date column, initially formatted as a string, is converted to the DateType for accurate temporal analysis.

```
# Convert the 'release_Date' column from string to date format
bo_data_df = bo_data_df.withColumn('release_Date', to_date(F.col('release_Date'), 'dd/MM/yyyy'))

bo_data_df.show(10)
```

Currency Conversion Integration

1. **Upload Conversion Rate Data**:
   - A file containing currency conversion rates (conversion_rate.csv) is uploaded, which includes USD-to-INR and GBP-to-INR conversion rates.

```
[ ]  # This will open a file upload dialog
     uploaded = files.upload()

     Choose Files  No file chosen    Upload widget is only available when the cell has been executed in the current browser session.
     Please rerun this cell to enable.
     Saving converstion_rate.csv to converstion_rate.csv

[ ]  conversion_df = spark.read.format('csv').option("header", "true").option("inferSchema", "true").load("conversti

[ ]  conversion_df.printSchema()

     root
      |-- year: integer (nullable = true)
      |-- usd_to_inr: double (nullable = true)
      |-- gbp_to_inr: double (nullable = true)
```

2. **Extract Year and Join Conversion Rates**:
   - The release year is extracted from release_Date and stored in a new column, release_year.
   - The conversion rate data is then joined with the main DataFrame on the release_year column to apply year-specific conversion rates.

```python
# Extract the year from 'release_Date' in bo_data_df
bo_data_df = bo_data_df.withColumn("release_year", year(F.col("release_Date")))

# Ensure conversion_df has the correct data types
conversion_df = conversion_df.withColumn("usd_to_inr", conversion_df["usd_to_inr"].cast(FloatType())) \
                             .withColumn("gbp_to_inr", conversion_df["gbp_to_inr"].cast(FloatType())) \
                             .withColumn("year", conversion_df["year"].cast(IntegerType()))

# Join conversion rates with the main DataFrame on the extracted year
bo_data_df = bo_data_df.join(conversion_df, bo_data_df["release_year"] == conversion_df["year"], "left")

bo_data_df.show(10)
```

3. **Convert Overseas Gross to INR**:
    - Columns containing overseas box office figures in USD or GBP are converted to INR using the appropriate year-specific conversion rate.
    - USD-based columns are renamed to _in_dollars, and GBP-based columns to _in_pounds. Converted values in INR are saved in new columns with _in_inr suffix.

```python
# USD and GBP columns for conversion
usd_columns = [
    "overseas_gross", "usa_/_canada_overseas_first_weekend", "usa_/_canada_overseas_total_gross",
    "gulf_overseas_first_weekend", "gulf_overseas_total_gross", "australia_overseas_first_weekend",
    "australia_overseas_total_gross"
]
gbp_columns = [
    "united_kingdom_overseas_first_weekend", "united_kingdom_overseas_total_gross"
]
```

```python
# Convert USD values to INR using the year-specific rate
for column in usd_columns:
    bo_data_df = bo_data_df.withColumn(
        f"{column}_in_inr",
        F.when(F.col(column).isNotNull(), F.col(column) * F.col("usd_to_inr")).otherwise(F.lit(None))
    ).withColumnRenamed(column, f"{column}_in_dollars")

# Convert GBP values to INR using the year-specific rate
for column in gbp_columns:
    bo_data_df = bo_data_df.withColumn(
        f"{column}_in_inr",
        F.when(F.col(column).isNotNull(), F.col(column) * F.col("gbp_to_inr")).otherwise(F.lit(None))
    ).withColumnRenamed(column, f"{column}_in_pounds")
```

4. **Sample Filtering**:
    - A sample DataFrame, new_df, is created, containing records of selected movies for preliminary inspection and validation of currency conversion.

```
new_df = bo_data_df.filter((col("name")=='Kabir Singh') | (col("name")=='Sanju') | (col("name")=='Tees Maar Khan') | (col("name")=='Bodyguard') | (col("name")=='Ek Tha Tiger') | (col("name")=='Chennai Express'))

new_df.select("name", "usd_to_inr", "gbp_to_inr", "year",
              "usa_/_canada_overseas_first_weekend_in_dollars", "usa_/_canada_overseas_total_gross_in_dollars",
              "usa_/_canada_overseas_first_weekend_in_inr", "usa_/_canada_overseas_total_gross_in_inr",
    "gulf_overseas_first_weekend_in_dollars", "gulf_overseas_total_gross_in_dollars", "gulf_overseas_first_weekend_in_inr", "gulf_overseas_total_gross_in_inr",
              "australia_overseas_first_weekend_in_dollars", "australia_overseas_total_gross_in_dollars", "australia_overseas_first_weekend_in_inr", "australia_overseas_total_gross_in_inr",
              "united_kingdom_overseas_first_weekend_in_pounds", "united_kingdom_overseas_total_gross_in_pounds",
              "united_kingdom_overseas_first_weekend_in_inr", "united_kingdom_overseas_total_gross_in_inr").show()
```

Splitting lists of fields into individual columns

```
# Set the maximum number of columns you want for each category
max_actors = 4
max_directors = 2
max_producers = 4
max_banners = 3
max_music_directors = 2
max_lyricists = 2

# Apply column splitting logic to bo_data_df
bo_data_df = bo_data_df.select(
   "name","release_Date","genre","verdict", "usd_to_inr", "gbp_to_inr", "year",
   "screens","first_day","opening_note","budget","first_weekend","first_week","total_nett_gross","india_gross","overseas_gross_in_dollars","wor
   "india_total_nett","india_share","mumbai_total_nett","mumbai_share","delhi_/_up_total_nett","delhi_/_up_share","east_punjab_total_nett","eas
   "united_kingdom_overseas_first_weekend_in_pounds","united_kingdom_overseas_total_gross_in_pounds","usa_/_canada_overseas_first_weekend_in_do
   "week1_mumbai","week1_delhi_up","week1_east_punjab","week1_rajasthan","week1_cp_berar","week1_ci","week1_nizam_andhra","week1_mysore","week1
   "week2_mumbai","week2_delhi_up","week2_east_punjab","week2_rajasthan","week2_cp_berar","week2_ci","week2_nizam_andhra","week2_mysore","week2
   "week3_mumbai","week3_delhi_up","week3_east_punjab","week3_rajasthan","week3_cp_berar","week3_ci","week3_nizam_andhra","week3_mysore","week3
   "week4_mumbai","week4_delhi_up","week4_east_punjab","week4_rajasthan","week4_cp_berar","week4_ci","week4_nizam_andhra","week4_mysore","week4
   "week5_mumbai","week5_delhi_up","week5_east_punjab","week5_rajasthan","week5_cp_berar","week5_ci","week5_nizam_andhra","week5_mysore","week5
   "week6_mumbai","week6_delhi_up","week6_east_punjab","week6_rajasthan","week6_cp_berar","week6_ci","week6_nizam_andhra","week6_mysore","week6
   "week7_mumbai","week7_delhi_up","week7_east_punjab","week7_rajasthan","week7_cp_berar","week7_ci","week7_nizam_andhra","week7_mysore","week7
   "week8_mumbai","week8_delhi_up","week8_east_punjab","week8_rajasthan","week8_cp_berar","week8_ci","week8_nizam_andhra","week8_mysore","week8
   "week9_mumbai","week9_delhi_up","week9_east_punjab","week9_rajasthan","week9_cp_berar","week9_ci","week9_nizam_andhra","week9_mysore","week9
   "week10_mumbai","week10_delhi_up","week10_east_punjab","week10_rajasthan","week10_cp_berar","week10_ci","week10_nizam_andhra","week10_mysore
   "week11_mumbai","week11_delhi_up","week11_east_punjab","week11_rajasthan","week11_cp_berar","week11_ci","week11_nizam_andhra","week11_mysore
   *[F.expr(f"create_columns(actors, {max_actors})").getItem(i).alias(f"Actor_{i+1}") for i in range(max_actors)],
    *[F.expr(f"create_columns(Director, {max_directors})").getItem(i).alias(f"Director_{i+1}") for i in range(max_directors)],
    *[F.expr(f"create_columns(Producer, {max_producers})").getItem(i).alias(f"Producer_{i+1}") for i in range(max_producers)],
    *[F.expr(f"create_columns(banners, {max_banners})").getItem(i).alias(f"Banner_{i+1}") for i in range(max_banners)],
    *[F.expr(f"create_columns(Music, {max_music_directors})").getItem(i).alias(f"MusicDirector_{i+1}") for i in range(max_music_directors)],
    *[F.expr(f"create_columns(Lyrics, {max_lyricists})").getItem(i).alias(f"Lyricist_{i+1}") for i in range(max_lyricists)],
)
```

Column Renaming

You rename columns with special characters to follow a consistent format:

```
temp_df = temp_df.withColumnRenamed("delhi_/_up_total_nett", "delhi_up_total_nett") \
       .withColumnRenamed("delhi_/_up_share", "delhi_up_share") \
       .withColumnRenamed("nizam_/_andhra_total_nett", "nizam_andhra_total_nett") \
       .withColumnRenamed("nizam_/_andhra_share", "nizam_andhra_share") \
       .withColumnRenamed("tamil_n_/_kerala_total_nett", "tamil_n_kerala_total_nett") \
       .withColumnRenamed("tamil_n_/_kerala_share", "tamil_n_kerala_share")
```

This renaming will make these columns easier to reference in PySpark.

Filling Distributor Shares

The fill_distributor_share function fills missing values for each territory's distributor share using 47.2% of the territory's total_nett value if it is available:

```python
# Function to fill the distributor share
def fill_distributor_share(df, territory):
    total_nett_col = f"{territory}_total_nett"
    share_col = f"{territory}_share"

    # Fill distributor share (47.2% of total_nett if total_nett is available and share is null)
    df = df.withColumn(
        share_col,
        F.when(
            F.col(share_col).isNull() & F.col(total_nett_col).isNotNull(),
            F.col(total_nett_col) * 0.472
        ).otherwise(F.col(share_col))
    )

    return df
```

Weekly Data Handling

The fill_week_columns function fills weekly gross values:

- **Weeks 6-11** are set to 0 if all week columns are null.
- **Weeks 1-5** are filled with a fraction of total_nett if they are null. This approach preserves existing values in weekly columns:

```python
# Function to fill week columns only if they are null and handle weeks 6 to 11
def fill_week_columns(df, territory):
    total_nett_col = f"{territory}_total_nett"
    week_columns = [f"week{i}_{territory}" for i in range(1, 12)]  # Weeks 1 to 11

    # Step 1: Set weeks 6 to 11 to 0 only if all week fields (week1 to week11) are NULL
    all_weeks_null_condition = F.lit(True)
    for week_col in week_columns:
        all_weeks_null_condition = all_weeks_null_condition & F.col(week_col).isNull()

    for week_col in week_columns[5:]:  # For weeks 6 to 11
        df = df.withColumn(
            week_col,
            F.when(
                all_weeks_null_condition,  # Set weeks 6-11 to 0 only if all weeks 1 to 11 are NULL
                F.lit(0)
            ).otherwise(F.col(week_col))  # Retain original value if it's not null
        )

    # Step 2: Distribute the total_nett across weeks 1 to 5 only if those weeks are NULL
    for i, week_col in enumerate(week_columns[:5], 1):  # For weeks 1 to 5
        df = df.withColumn(
            week_col,
            F.when(
                F.col(week_col).isNull() & F.col(total_nett_col).isNotNull(),  # Only fill if week_col
                F.col(total_nett_col) / 5
            ).otherwise(F.col(week_col))  # Retain original value if it exists
        )

    return df
```

Handling Null and NaN Counts

To handle and check for nulls in the DataFrame:

```
# Fetch null counts

# List to store the null count expressions
null_counts = []

# Loop through columns, handling numeric and non-numeric columns separately
for column in temp_df.columns:
    if dict(temp_df.dtypes)[column] in ['float', 'double']:
        # Check for NaN and Null in numeric columns
        null_counts.append(count(when(isnan(col(column)) | col(column).isNull(), column)).alias(column))
    else:
        # Only check for Null in non-numeric columns
        null_counts.append(count(when(col(column).isNull(), column)).alias(column))

# Apply the select transformation with the null_counts list
temp_df.select(null_counts).show()
```

This counts nulls and NaNs separately for numeric columns.

Gross Calculations

You calculate overseas_gross_in_inr by subtracting india_gross from worldwide_gross:

```
# Step 1: Compute overseas_gross_in_inr if it's null
temp_df = temp_df.withColumn(
    "overseas_gross_in_inr",
    F.when(
        F.col("overseas_gross_in_inr").isNull() & F.col("worldwide_gross").isNotNull() & F.col("india_gross").isNotNull(),
        F.col("worldwide_gross") - F.col("india_gross")
    ).otherwise(F.col("overseas_gross_in_inr"))
)

# Step 2: Compute worldwide_gross if it's null
temp_df = temp_df.withColumn(
    "worldwide_gross",
    F.when(
        F.col("worldwide_gross").isNull() & F.col("overseas_gross_in_inr").isNotNull() & F.col("india_gross").isNotNull(),
        F.col("overseas_gross_in_inr") + F.col("india_gross")
    ).otherwise(F.col("worldwide_gross"))
)
```

If worldwide_gross is missing, it is recalculated using india_gross and overseas_gross_in_inr.

Overseas Gross in INR/Dollars/Pounds

The code calculates missing gross amounts across territories based on conversion rates:

```python
# Step 3: Convert overseas_gross_in_inr to overseas_gross_in_dollars
temp_df = temp_df.withColumn(
    "overseas_gross_in_dollars",
    F.when(
        F.col("overseas_gross_in_dollars").isNull() & F.col("overseas_gross_in_inr").isNotNull(),
        F.col("overseas_gross_in_inr") / F.col("usd_to_inr")
    ).otherwise(F.col("overseas_gross_in_dollars"))
)

# Show the resulting DataFrame with the relevant columns
temp_df.select("worldwide_gross", "india_gross", "overseas_gross_in_inr", "overseas_gross_in_dollars").show()
```

Suggestions:

1. **Efficiency**: Consider caching or checkpointing if data size is large.
2. **Testing**: Verify null counts and calculations to ensure correct results.
3. **Documentation**: Document the function inputs/outputs to clarify data expectations.

This script provides a robust solution for handling missing data and calculated fields for a mo