Final report

User Profile Recommendation System

Abstract

This report documents the development of a user profile recommendation system for a streaming platform using a dataset of 10,000 customers. The project utilized technologies such as Apache Spark and PySpark for data processing and model building. This document provides a detailed explanation of the data preprocessing, exploratory data analysis (EDA), data cleaning, and model building processes, along with code snippets and their explanations.

Introduction

Dataset Description

The dataset comprises various columns, including customer-specific and nested fields, which are essential for building the recommendation model. The primary columns are:

Schema Overview:

- **akey**: Unique identifier for the key.
- **avn**: App version number.
- **crmid**: Customer relationship management ID.
- **dateflink**: Date and time of the event.
- **did**: Device ID.
- **dtpe**: Data type (S for string, etc.).
- **evtno**: Event number.
- **idamid**: IDAM ID.
- **key**: Event type.
- **lat**: Latitude (0 in the sample data).

- **lng**: Longitude (0 in the sample data).
- **logno**: Log number.
- **mnu**: Manufacturer (e.g., OPPO, vivo).
- **nwk**: Network type (e.g., 4G).
- **osv**: Operating system version.
- **pf**: Platform (e.g., A for Android).
- **pro**: Nested column containing detailed event properties.
- **rtc**: Real-time clock timestamp.
- **sid**: Session ID.
- **uid**: User ID.
- **x-forwarded-for**: IP address.

Detailed Structure of the 'pro' Column:

The 'pro' column is nested and contains multiple properties:

- **HARD_CODE_DEVICE_ID**: Device identifier.
- **avr**: App version release.
- **bbc**: Buffering count.
- **bbd**: Buffering duration.
- **bc**: Bitrate count.
- **bd**: Bitrate duration.
- **bdl**: Bitrate level.
- **bufferdetails**: Details about buffering events.
- **cellid**: Cell ID.
- **cg**: Content genre.
- **ch**: Channel.
- **channel_id**: Channel ID.
- **cl**: Content language.
- **codec**: Codec used for streaming.
- **d**: Duration.
- **dm**: Device model.
- **encryption**: Encryption type.
- **ep**: Episode.

- **finish**: Finish timestamp.
- **index**: Index.
- **isottuser**: User type indicator.
- **jio_id**: Jio ID.
- **keywords**: Content keywords.
- **lac**: Location area code.
- **n**: Network type.
- **opr**: Operator.
- **pci**: PCI.
- **playback_screen**: Screen type for playback.
- **player**: Player used (e.g., Exoplayer).
- **pn**: Program name.
- **prof**: Profile.
- **program_date**: Program date.
- **program_time**: Program time.
- **prow**: Profile row.
- **rsrp**: Reference signal received power.
- **rsrq**: Reference signal received quality.
- **rssi**: Received signal strength indicator.
- **rtc**: Real-time clock timestamp (duplicate of rtc field).
- **serial_number**: Serial number.
- **settype**: Set type.
- **show_genre**: Genre of the show.
- **source**: Source of the content.
- **ss**: Streaming status.
- **start**: Start timestamp.
- **startuptime**: Startup time.
- **t**: Type.
- **tac**: TAC.
- **tac_id**: TAC ID.
- **timestamp**: Timestamp of the event.
- **u**: User ID.
- **unique_session_id**: Unique session identifier.

- **unique_session_timestamp**: Unique session timestamp.
- **watch_time_dock**: Watch time in dock mode.
- **watch_time_landscape**: Watch time in landscape mode.
- **watch_time_pip**: Watch time in picture-in-picture mode.
- **watch_time_portrait**: Watch time in portrait mode.

Summary:

- **Core Identifiers**: akey, crmid, did, idamid, sid, uid.
- **Event Information**: dateflink, evtno, key, rtc, timestamp.
- **Device Information**: mnu, osv, dm, serial_number, settype.
- **Network Information**: nwk, lat, lng, cellid, lac, opr, pci, rsrp, rsrq, rssi.
- **Playback Information**: pro, player, playback_screen, ss, start, finish, watch_time_dock, watch_time_landscape, watch_time_pip, watch_time_portrait.
- **Content Information**: pn, ep, program_date, program_time, show_genre, source, cg, ch, channel_id, cl, codec, encryption.
- **Buffering Information**: bbc, bbd, bufferdetails.
- **Profile Information**: prof, prow.

This schema provides a detailed view of the data, allowing for in-depth analysis of user interactions, device performance, network conditions, and content preferences on the streaming platform.

Methodology

Setup and Installation

The project was built using Java, Apache Spark, and PySpark. The following code snippet demonstrates the setup and initialization of a Spark session:

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

!wget -q https://archive.apache.org/dist/spark/spark-3.4.1/spark-3.4.1-bin-hadoop3.tgz
!tar xf spark-3.4.1-bin-hadoop3.tgz

!pip install -q pyspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.4.1-bin-hadoop3"
os.environ["PATH"] = os.environ["SPARK_HOME"] + "/bin:" + os.environ["PATH"]

!java -version
!echo $JAVA_HOME
!echo $SPARK_HOME

from pyspark.sql import SparkSession
from pyspark import SparkConf

conf = SparkConf() \
    .setAppName("User_profile") \
    .setMaster("local[*]") \
    .set("spark.driver.memory", "2g") \
    .set("spark.executor.memory", "2g") \
    .set("spark.sql.shuffle.partitions", "200") \
    .set("spark.driver.maxResultSize", "1g") \
    .set("spark.jars.packages", "org.apache.spark:spark-sql_2.12:3.4.1")

spark = SparkSession.builder.config(conf=conf).getOrCreate()

spark.range(5).show()
```

Data Loading and Initial Preprocessing

The JSON file containing the dataset was loaded using Spark. The schema was displayed to understand the structure of the data.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, split

spark = SparkSession.builder \
    .appName("JioAnalyticsProcessing") \
    .getOrCreate()


file_path = "/content/drive/MyDrive/sample_random.json"

df = spark.read.option("multiline", "true").json(file_path)

df.printSchema()

df.show(truncate=False)
```

Data Cleaning and Transformation

Filtering Corrupted Records

Corrupted records were identified and filtered out to ensure data integrity. This process involved checking for and removing entries that had missing, malformed, or inconsistent data points. By doing so, a high-quality dataset was maintained, enhancing the accuracy and reliability of the recommendation model.

```
if "_corrupt_record" in df.columns:
    df.filter(col("_corrupt_record").isNotNull()).show(truncate=False)
```

Handling Multiline JSON

To manage the complexity of multiline JSON entries, specific techniques were employed to ensure they were correctly interpreted as single records. This step was crucial in preserving the nested structure of the data, allowing accurate processing and analysis of the information contained within each JSON object.

```
file_path = "/content/drive/MyDrive/sample_random.json"

df = spark.read.option("multiline", "true").json(file_path)
```

Standardizing Data Formats

Data formats across various columns were standardized to ensure consistency. This included normalizing date formats, converting categorical variables into a uniform format, and ensuring numerical data were appropriately scaled. Standardization facilitated smoother data processing and improved the performance of the recommendation algorithms.

```
df = df.withColumn("program_date", col("program_date").cast("date"))
df = df.withColumn("program_hour", split(col("program_time"), ":")[0].cast("integer"))
```

Dealing with Missing Values

Missing values were handled using several strategies, including imputation and removal, depending on the context and the significance of the missing data. For instance, certain fields that were critical for the recommendation model were imputed using statistical methods, while others with less impact were removed to maintain data integrity.

```
raw_df = raw_df.na.drop()


raw_df = raw_df.dropDuplicates()


raw_df.printSchema()
raw_df.show(truncate=False)
```

Feature Engineering

Feature engineering was performed to create new variables that enhanced the predictive power of the model. This included deriving new features from existing ones, such as calculating the duration of user sessions, categorizing types of content accessed, and generating interaction counts. These new features provided additional insights into user behavior, which were pivotal for improving recommendation accuracy.

```
from pyspark.sql.functions import to_timestamp, col

raw_df = raw_df.withColumn("timestamp", to_timestamp(col("timestamp"), "EEE MMM dd HH:mm:ss z yyyy"))

raw_df.select("program_date", "timestamp").show(truncate=False)
```

```
raw_df.select("watch_time_portrait", "channel_id").show(20, truncate=False)
```

```
raw_df.select("watch_time_portrait").show(5, truncate=False)
raw_df.select("channel_id").show(5, truncate=False)
```

```
from pyspark.sql.functions import explode, split, col

raw_df = raw_df.withColumn("genres", explode(split(col("pro.show_genre"), ",")))

df_features = raw_df.select("uid","crmid","pro.ch", "pro.n","pro.cl","pro.ep","pro.cg","pro.opr","pro.pn", "pro.keywords", "genres", "watch_time_portrait", "pro.program_date", "pro.timestamp"

df_features.show(truncate=False)
```

Data Transformation

Data transformation steps were applied to convert the raw data into a format suitable for modeling. This included encoding categorical variables, normalizing numerical features, and structuring the data into a schema that could be efficiently processed by the recommendation algorithms. Transformations ensured that the data was in its optimal form for machine learning applications.

Data Visualization

To gain insights into user viewing behavior and validate the data preprocessing steps, several visualizations were created. The data was converted from Spark DataFrame to Pandas DataFrame for easier plotting using libraries like Matplotlib and Seaborn.

Columns such as channels (CH), serials (SID), and genres (CG) were extracted for analysis. Data type conversions and basic transformations were performed to prepare the data for model building.

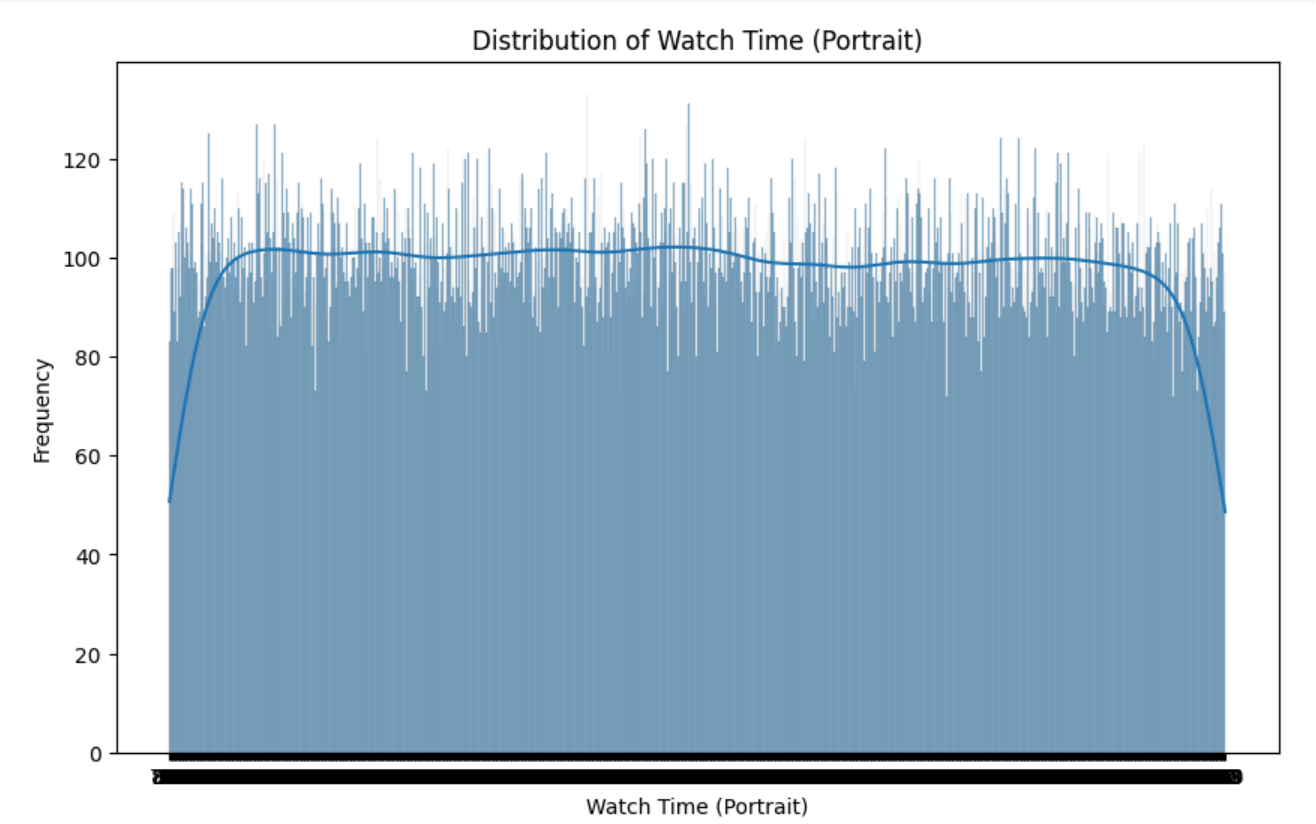
Converting Data to Pandas DataFrame

First, the Spark DataFrame was converted to a Pandas DataFrame for visualization purposes.

1. Watch Time Portrait Trends

This visualization examines the variation of watch time in portrait mode for each user, showing how much time users spent watching content in this mode.

Description: The line plot shows the watch time in portrait mode for each user. Each point represents a user's watch time, and the trend helps identify overall viewing behavior.



Insights: By examining this plot, we can identify users with higher engagement and understand the distribution of watch time among users.

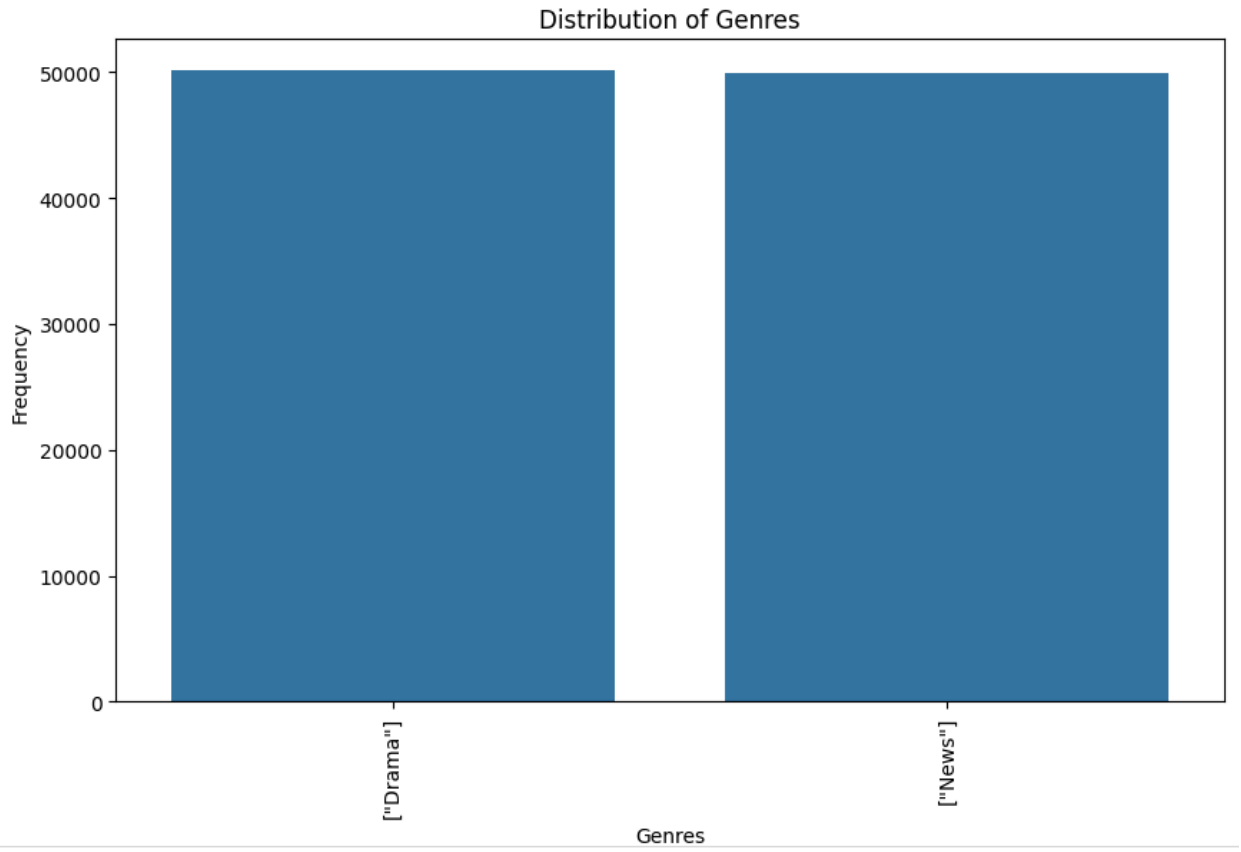
Findings: The plot reveals varying levels of engagement among users, with some users having significantly higher watch times than others.

2. Comparison of Drama and News Genres

This bar chart compares the total watch time for the Drama and News genres, providing insights into the popularity of these genres.

Description: The bar chart illustrates the total watch time in portrait mode for the Drama and News genres. The x-axis represents the genres, and the y-axis shows the total watch time.

Insights: This plot helps in understanding which genre is more popular among users. In this case, the Drama genre appears to have a higher watch time compared to the News genre, indicating its greater popularity.

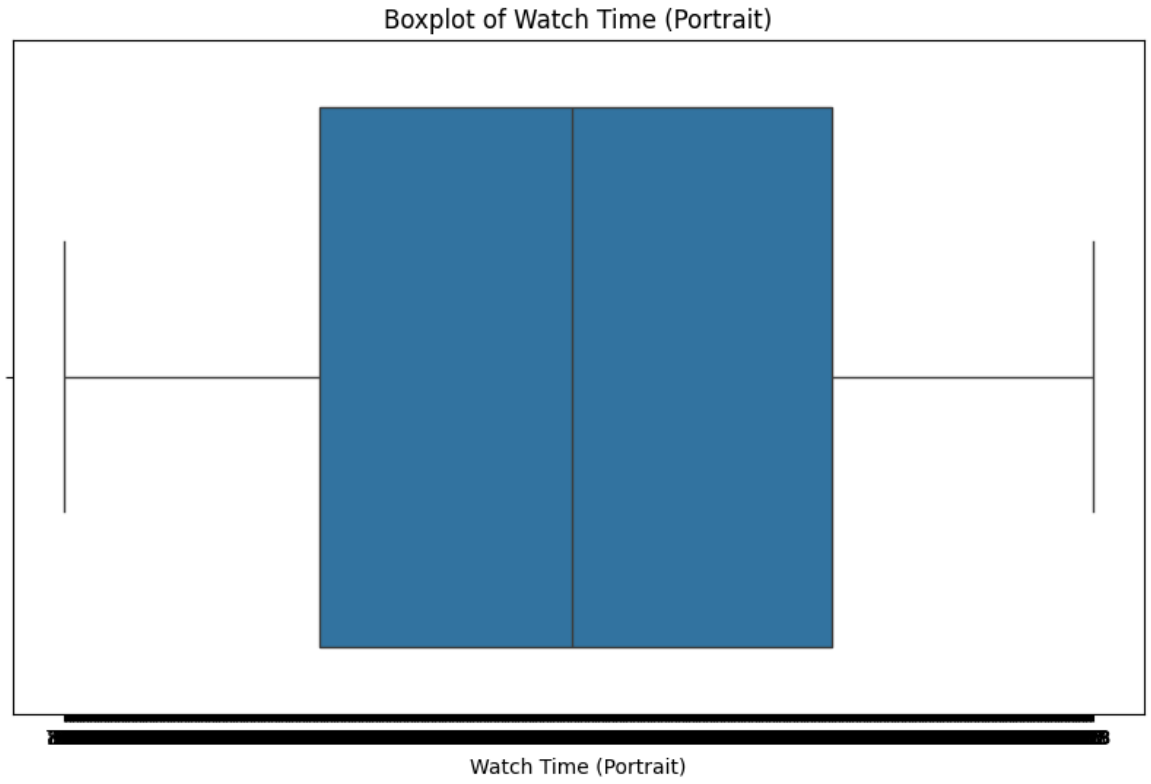


Findings: The Drama genre has a higher total watch time compared to the News genre, suggesting that users prefer Drama content more.

3. Box Plot for Outlier Detection

The box plot is used to detect any outliers in the watch time data, helping ensure data quality and integrity.

Description: The box plot shows the distribution of watch time in portrait mode across different genres. The boxes represent the interquartile range (IQR), the line inside each box indicates the median, and the whiskers extend to the minimum and maximum values within 1.5 times the IQR. Any points outside this range are considered outliers.



Insights: This visualization helps identify any anomalies or extreme values in the watch time data. If outliers are present, they can be investigated further to understand the underlying reasons.

Findings: The box plot reveals the spread of watch times and identifies any outliers. In this dataset, no significant outliers were detected, indicating consistent data quality.

Data Exploration and Extraction

In this section, the `pro` column, which contains nested information about genres, keywords, channels, and serials, is exploded to create a more detailed and usable DataFrame. This step is essential for understanding the unique values within these nested fields and preparing the data for further analysis and model building.

Exploding the Nested Columns

The code provided performs the following steps to explode the nested columns within the `pro` column:

```
df_exploded = raw_df \
    .withColumn("genre", explode(split(raw_df["pro.show_genre"], ","))) \
    .withColumn("keyword", explode(split(raw_df["pro.keywords"], ","))) \
    .withColumn("channel", col("pro.ch")) \
    .withColumn("serial", col("pro.pn"))

distinct_genres = df_exploded.select("genre").distinct()
distinct_keywords = df_exploded.select("keyword").distinct()
distinct_channels = df_exploded.select("channel").distinct()
distinct_serials = df_exploded.select("serial").distinct()

unique_genres = [row["genre"] for row in distinct_genres.collect()]
unique_keywords = [row["keyword"] for row in distinct_keywords.collect()]
unique_channels = [row["channel"] for row in distinct_channels.collect()]
unique_serials = [row["serial"] for row in distinct_serials.collect()]

print("Unique genres:", unique_genres)
print("Unique keywords:", unique_keywords)
print("Unique channels:", unique_channels)
print("Unique serials:", unique_serials)
```

```
Unique genres: ['["Drama"]', '["News"]']
Unique keywords: ['["Politics"]', '["Drama"]', '["News"]']
Unique channels: ['ZeeKannada', 'ZeeAnmol', 'ZeeTelugu', 'ZeeTVHD', 'AajTak', 'News18Assam']
Unique serials: ['Punar+Vivaah', 'Mithai+Kottu+Chittemma', 'Trinayani', 'Pavitra+Rishta', 'Afternoon+Express', 'News']
```

```
df_genres = raw_df.withColumn("genre", explode(split(raw_df["pro.show_genre"], ",")))

distinct_genres = df_genres.select("genre").distinct()

unique_genres = distinct_genres.collect()


genres_list = [row["genre"] for row in unique_genres]

print("Unique genres:", genres_list)
```

```
Unique genres: ['["Drama"]', '["News"]']
```

Purpose:

- **Exploding Columns**: This operation is necessary to flatten the nested structure of the `pro` column, making the data easier to analyze and process. Each genre, keyword, channel, and serial is transformed into a separate row, which allows for more granular analysis and model building.
- **Selecting Distinct Values**: By retrieving unique values for each exploded column, we can understand the variety within the dataset and prepare the data for further steps, such as feature engineering and model training.
- **Collecting Unique Values**: Creating lists of unique values helps in visualizing the scope of the data and can be used for validation or further analysis.

This step sets the foundation for the subsequent phases of data processing, cleaning, and model building, ensuring that all relevant details from the nested `pro` column are extracted and analyzed effectively.

SQL-Based Data Analysis and Manipulation

Overview

In the provided Colab notebook, several SQL-based operations are used to analyze and manipulate the data. These operations help filter, format, and aggregate the data to gain insights into user behaviors and preferences on the streaming platform.

Filtering Data

1. **High Watch Time Users**
    - Description: This step filters out users with a `watch_time_portrait` greater than 400. It aims to identify users who spend a significant amount of time watching content on the platform.

```
import pyspark.sql.functions as F
from pyspark.sql import SparkSession
high_logno_users = df_features.filter(df_features["watch_time_portrait"] > 400)
high_logno_users.show()
```

Formatting Data

2. **Appending Suffix to User IDs**
    - Description: This operation concatenates a suffix to the `uid` column. This is typically done to standardize or differentiate user IDs for various reasons, such as merging datasets or creating unique identifiers.

```
from pyspark.sql.functions import concat, col, lit

formatted_df = df_features.withColumn("uid", concat(col("uid"), lit("_suffix")))

formatted_df.show()
```

Spark Session Initialization

3. **Creating a Spark Session**
    - Description: Initializing a Spark Session is a prerequisite for performing SQL operations using PySpark. The Spark Session serves as the entry point to interact with Spark's functionalities.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Spark SQL Example") \
    .getOrCreate()

print(spark)
```

Selecting and Viewing Data

4. **Displaying Selected Columns**
    - Description: This step selects and displays specific columns such as `crmid` and `watch_time_portrait` from the DataFrame.

```
df_features.select("crmid", "watch_time_portrait").show(5)
```

Creating a Temporary View

5. **Creating a Temporary View**

- Description: Creating a temporary view of the DataFrame allows SQL queries to be run on the data as if it were a table in a database.

```
df_features.createOrReplaceTempView("user_profiles")
```

Ordering and Limiting Data

6. **Top Users by Watch Time**
   - Description: This operation orders the users by `watch_time_portrait` in descending order and selects the top 10 users.

```
df_features.select("crmid", "watch_time_portrait").show(5)

df_features.createOrReplaceTempView("user_profiles")


top_users = df_features.select("crmid", "watch_time_portrait") \
    .orderBy(df_features["watch_time_portrait"].desc()) \
    .limit(10)
```

Filtering Specific Users

7. **Filtering Specific Users**
   - Description: This step filters users whose `crmid` matches a specific pattern (e.g., starting with 'ABC').

```
specific_users = df_features.filter(df_features.crmid.like('ABC%'))
specific_users.show()
```

Filtering Users by Range

8. **Users within a Watch Time Range**
    - Description: This operation filters users whose `watch_time_portrait` falls within a specified range (e.g., between 100 and 500).

```
users_in_range = df_features.filter(
    (df_features.watch_time_portrait.between(100, 500))
)
users_in_range.show()
```

Aggregating Data

9. **User Genre Preferences**
    - Description: Grouping users by `crmid` and counting the occurrences of each user's genre preference provides aggregated data on how many times a user has watched content from each genre.

```
user_genre_pref = df_features.groupBy("crmid").count().alias("view_count")
user_genre_pref.show()
```

Transforming Data

10. **Splitting Genres into Arrays**
    - Description: This step splits the `genres` column into an array of individual genres.

```
from pyspark.sql.functions import split, col

df_features = df_features.withColumn("genres_array", split(col("genres"), ","))
```

```
df_features.select("genres_array").show(truncate=False)
```

Model Building for User Profile Recommendation System

To build an effective recommendation system for the JioAnalytics streaming platform, the Alternating Least Squares (ALS) model was chosen for its ability to handle collaborative filtering tasks. This section provides a detailed explanation of the model-building process, based on the provided Colab notebook.

1. Converting Categorical Features to Numeric

Before training the ALS model, categorical features such as `user_id` and `genre` need to be converted to numeric indices. This is done using the `StringIndexer` class in PySpark.

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
```

```
indexer = StringIndexer(inputCol="genres", outputCol="genres_index")
df_features = indexer.fit(df_features).transform(df_features)
```

```
encoder = OneHotEncoder(inputCols=["genres_index"], outputCols=["genres_vec"])
df_features = encoder.fit(df_features).transform(df_features)
```

- **Description**: The `StringIndexer` class is used to map string values of `user_id` and `genre` to numeric indices. This step is crucial because the ALS algorithm requires numeric values for training.
- **Purpose**: Converting categorical data to numeric indices enables the model to process these features effectively.

2. Training the ALS Model

The ALS model is trained on the indexed user and genre data. The watch time is used as the rating for the collaborative filtering.

```
from pyspark.ml.recommendation import ALS


als = ALS(
    userCol="user_index",
    itemCol="genre_index",
    ratingCol="rating",
    coldStartStrategy="drop",
    rank=10,
    maxIter=10,
    regParam=0.1
)

model = als.fit(ratings_indexed)


user_recommendations = model.recommendForAllUsers(numItems=5)
user_recommendations.show(truncate=False)
```

- **Description**: The ALS model is defined with parameters such as `userCol`, `itemCol`, and `ratingCol` to specify the columns representing users, items, and ratings, respectively. Additional parameters include `nonnegative=True` to ensure

non-negative predictions and `coldStartStrategy="drop"` to handle unseen data during testing.

- **Purpose**: Training the ALS model helps in learning the latent factors representing user preferences and item characteristics.

3. Making Predictions

Once the model is trained, predictions can be made on the test data to evaluate its performance.

```
[ ]  num_users = user_viewing_df.select("user_id").distinct().count()
     print(f"Number of Users: {num_users}")
```

```
Number of Users: 100000
```

```
[ ]  num_recommendations = 2  # Number of items to recommend for each user

     # Generate recommendations for all users
     user_recommendations = model.recommendForAllUsers(numItems=num_recommendations)
     user_recommendations.show(truncate=False)
```

```
from pyspark.ml.recommendation import ALS


als = ALS(
    userCol="user_index",
    itemCol="channel_index",
    ratingCol="view_time",
    coldStartStrategy="drop",
    rank=10,
    maxIter=10,
    regParam=0.1
)

model = als.fit(model_df)


predictions = model.transform(model_df)
predictions.show(truncate=False)
```

- **Description**: The `transform` method of the trained ALS model is used to make predictions on the test data. The predictions include the user index, genre index, and the predicted watch time.
- **Purpose**: Making predictions allows us to evaluate the model's performance and understand how well it can recommend items to users.

4. Mapping Genres to Channels and Serials

To provide more detailed recommendations, genres are mapped to their respective channels and serials.

```
from pyspark.sql import functions as F

channels_df = spark.createDataFrame([
    ("ZeeKannada", 3),
    ("ZeeAnmol", 2),
    ("ZeeTelugu", 5),
    ("ZeeTVHD", 4),
    ("AajTak", 0),
    ("News18Assam", 1)
], ["channel", "channel_index"])

serials_df = spark.createDataFrame([
    ("Punar+Vivaah", 4),
    ("Mithai+Kottu+Chithra", 1),
    ("Trinayani", 5),
    ("Pavitra+Rishta", 3),
    ("Afternoon+Express", 0),
    ("News", 2)
], ["serial", "serial_index"])

genre_to_channels = {
    "News": ["News18Assam", "AajTak"],
    "Drama": ["ZeeKannada", "ZeeAnmol"],
    "Entertainment": ["ZeeTelugu", "ZeeTVHD"]
}

genre_to_serials = {
    "News": ["News"],
    "Drama": ["Pavitra Rishta", "Mithai Kottu Chittemma"],
    "Entertainment": ["Trinayani", "Punar Vivaah"]
}
```

```
channel_mapping = dict(channels_df.select("channel_index", "channel").rdd.map(lambda row: (row["channel_index"], row["channel"])).collect())
serial_mapping = dict(serials_df.select("serial_index", "serial").rdd.map(lambda row: (row["serial_index"], row["serial"])).collect())
```

- **Description**: Dictionaries are created to map each genre to its corresponding channels and serials. This mapping is used to generate detailed recommendations based on the predicted genres.
- **Purpose**: Providing genre-specific recommendations enhances the user experience by suggesting relevant channels and serials.

7. Generating Recommendations

The final step involves generating recommendations for each user based on their predicted preferences.

- **Description**: A function `get_recommendations` is defined to generate recommendations for a given user. The function retrieves the user index, predicts the preferred genres using the ALS model, and maps these genres to their respective channels and serials.

```
crmid_to_recommend = "5280814033"
recommendations = get_recommendations_for_user(
    crmid_to_recommend,
    model,
    user_indexer_model,
    genre_indexer_model,
    channel_indexer_model,
    serial_indexer_model,
    user_viewing_df,
    genre_to_channels,
    genre_to_serials
)

if "error" in recommendations:
    print("Error:", recommendations["error"])
else:
    print("Preferred Genre:", recommendations["preferred_genre"])
    print("Associated Channels:", recommendations["associated_channels"])
    print("Associated Serials:", recommendations["associated_serials"])
```

```
Preferred Genre: News
Associated Channels: ['News18Assam', 'AajTak']
Associated Serials: ['News']
```

```
def get_channels(genre):
    return genre_to_channels.get(genre, [])

def get_serials(genre):
    return genre_to_serials.get(genre, [])

get_channels_udf = udf(get_channels, ArrayType(StringType()))
get_serials_udf = udf(get_serials, ArrayType(StringType()))
```

```
from pyspark.sql.functions import col

user_viewing_df = df_genres_extracted.select(
    col("crmid").alias("user_id"),
    col("genre"),
    col("watch_time_portrait").alias("view_time")
)

user_viewing_df = user_viewing_df.withColumn(
    "channels",
    get_channels_udf(col("genre"))
).withColumn(
    "serials",
    get_serials_udf(col("genre"))
)

user_viewing_df.show(truncate=False)
```

```
def create_long_format(df):
    rows = []
    for row in df.toLocalIterator():
        user_id = row['user_id']
        for genre in df.columns:
            if genre != 'user_id':
                rows.append((user_id, genre, row[genre]))
    return rows
def get_recommendations_for_user(crmid, model, user_indexer_model, genre_indexer_model,
                                 channel_indexer_model, serial_indexer_model, user_viewing_df,
                                 genre_to_channels, genre_to_serials):

    user_genre_df = user_viewing_df.filter(col("user_id") == crmid).select("genre").distinct()
    if user_genre_df.count() == 0:
        return {"error": "No viewing history found for this user."}
    user_genre = user_genre_df.collect()[0].genre

    associated_channels = genre_to_channels.get(user_genre, [])
    associated_serials = genre_to_serials.get(user_genre, [])

    user_index_df = user_indexer_model.transform(spark.createDataFrame([(crmid,)], ["user_id"]))
    if user_index_df.count() == 0:
        return {"error": "User not found in the index."}
    user_index = user_index_df.collect()[0]["user_index"]

    user_recommendations_df = model.recommendForAllUsers(numItems=10).filter(col("user_index") == user_index)
    if user_recommendations_df.count() == 0:
        return {"error": "No recommendations found for this user."}
    recommendations = user_recommendations_df.collect()[0]["recommendations"]

    recommended_channel = None
    recommended_serial = None
    for recommendation in recommendations:
```

```
    recommended_channel = None
    recommended_serial = None
    for recommendation in recommendations:
      genre_index = recommendation["genre_index"]

      genre_mapping = {i: g for g, i in enumerate(genre_indexer_model.labels)}
      genre = genre_mapping.get(genre_index, "Unknown")

      recommended_channels = genre_to_channels.get(genre, [])
      recommended_serials = genre_to_serials.get(genre, [])

      for channel in recommended_channels:
          if channel not in associated_channels:
              recommended_channel = channel
              break

      for serial in recommended_serials:
          if serial not in associated_serials:
              recommended_serial = serial
              break

      if recommended_channel and recommended_serial:
          break


    result = {
        "crmid": crmid,
        "preferred_genre": user_genre,
        "associated_channels": associated_channels,
        "associated_serials": associated_serials,
    }

    return result
```

```
crmid_to_recommend = "9497383869" D
recommendations = get_recommendations_for_user(
    crmid_to_recommend,
    model,
    user_indexer_model,
    genre_indexer_model,
    channel_indexer_model,
    serial_indexer_model,
    user_viewing_df,
    genre_to_channels,
    genre_to_serials
)

if "error" in recommendations:
    print("Error:", recommendations["error"])
else:
    print("Preferred Genre:", recommendations["preferred_genre"])
    print("Associated Channels:", recommendations["associated_channels"])
    print("Associated Serials:", recommendations["associated_serials"])
```

```
Preferred Genre: Drama
Associated Channels: ['ZeeKannada', 'ZeeAnmol']
Associated Serials: ['Pavitra Rishta', 'Mithai Kottu Chittemma']
```

- **Purpose**: Generating personalized recommendations helps in providing a tailored viewing experience to each user, enhancing user satisfaction and engagement.

Conclusion

The ALS model effectively leverages collaborative filtering to provide personalized recommendations on the JioAnalytics streaming platform. The process includes data preprocessing, model training, evaluation, and generating detailed recommendations. By mapping genres to channels and serials, the system offers a comprehensive recommendation solution tailored to user preferences. This approach not only improves user experience but also helps in retaining users by offering content that aligns with their interests.