Saavn- Case Study

Problem Statement

Build a system that keeps the users updated based on their music preferences. Suppose, a new track of an artist is released. Now, my responsibility would be to push the notification about this song to the appropriate set of audience. But the challenge is to push a song notification to its interested and relevant audience only.

Objective

The task is to build a model which give users an update about the new songs launched in the segment of their music preferences.

The data provided contains multiple files:

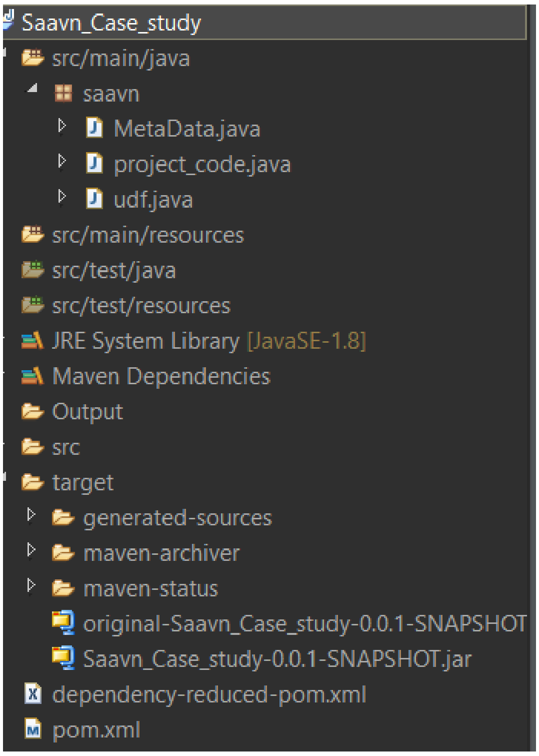
- User Click-Stream Activity which contain "User ID", "Timestamp", "Song ID" and "Date"
- MetaData which contain Attributes "Song ID" and "Artist ID"
- Notification Clicks which contain Attributes "Notification ID", "User ID", and "Date"
- Notification Artists which contain Attributes "Notification ID" and "Artist ID"

Approach

Data preparation with Collaborative filtering which is a technique to build personalised recommendation on the web.

For Clustering: K-means clustering was used, which is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining.

Model Building

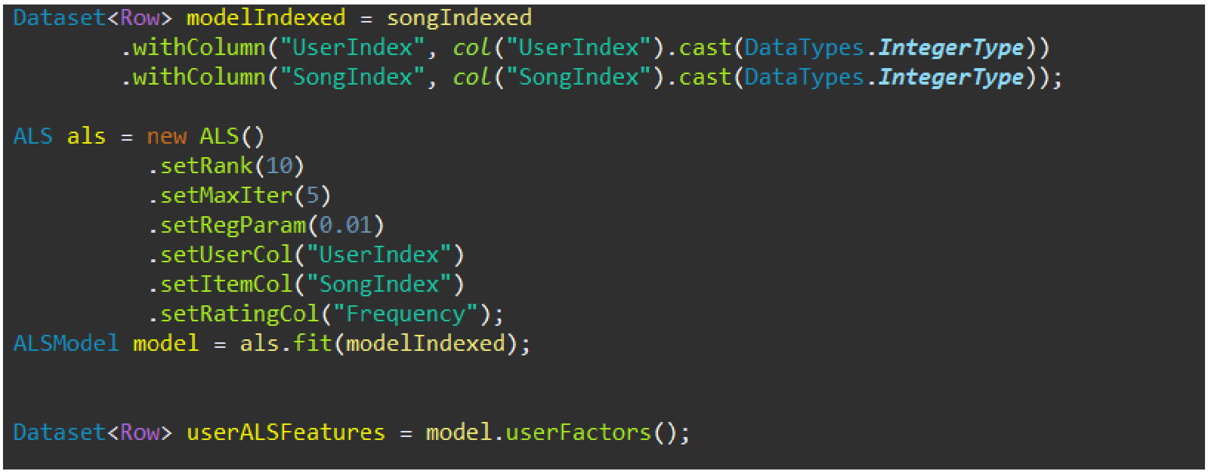


```
Saavn_Case_study
  src/main/java
    saavn
      MetaData.java
      project_code.java
      udf.java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  Output
  src
  target
    generated-sources
    maven-archiver
    maven-status
    original-Saavn_Case_study-0.0.1-SNAPSHOT
    Saavn_Case_study-0.0.1-SNAPSHOT.jar
  dependency-reduced-pom.xml
  pom.xml
```

- project_code is the primary class containing main function.
- This includes Spark session that contains the access key and the secret key for s3 bucket in the configuration.

Data preparation for model building:

With the help of read function of spark data file we read the used click stream CSV. Timestamp, date column and null values are dropped. Later on, we take the count of songs grouped by used ID and song ID. By using the string indexer, we may convert the user ID and song ID column to integer as ALS does not accept string value. After that user ID and Song ID columns are dropped.



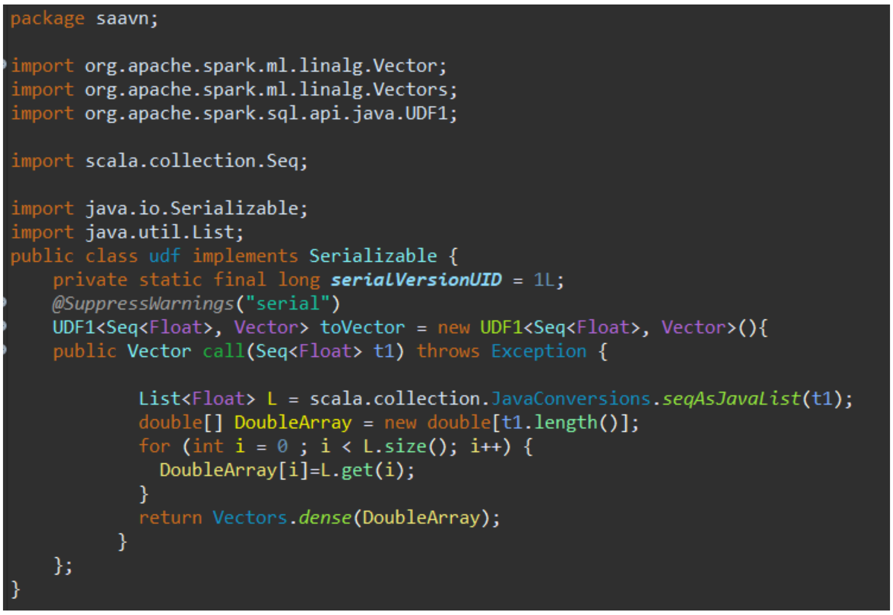
```
Dataset<Row> modelIndexed = songIndexed
        .withColumn("UserIndex", col("UserIndex").cast(DataTypes.IntegerType))
        .withColumn("SongIndex", col("SongIndex").cast(DataTypes.IntegerType));

ALS als = new ALS()
          .setRank(10)
          .setMaxIter(5)
          .setRegParam(0.01)
          .setUserCol("UserIndex")
          .setItemCol("SongIndex")
          .setRatingCol("Frequency");
ALSModel model = als.fit(modelIndexed);


Dataset<Row> userALSFeatures = model.userFactors();
```

*Collaborative Filtering: Using ALS (*Alternating Least Squares*)*

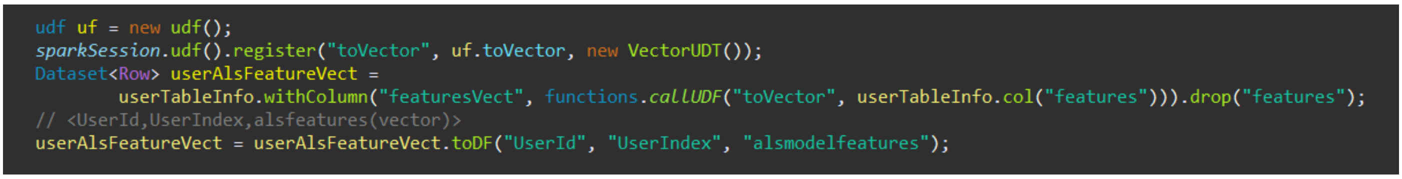
Parameters used are:

- **Rank**: Here taking rank as 10 to begin for more accuracy.
- **Max Iter**: number of iterations of ALS to run.
- **RegParam**: the regularization parameter.
- Using the user factors function, we get the feature array and the ID for the K-means algorithm.

UDF Creation and Registration for Converting feature array to vector:



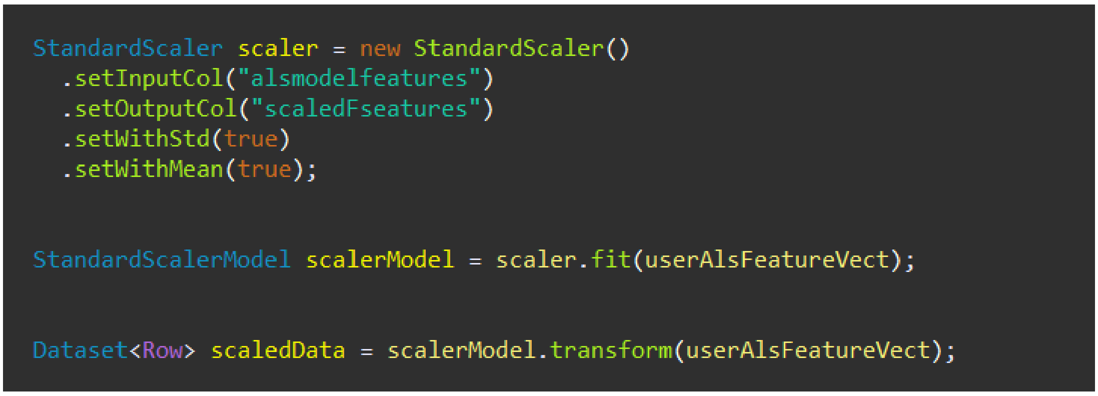
```
package saavn;

import org.apache.spark.ml.linalg.Vector;
import org.apache.spark.ml.linalg.Vectors;
import org.apache.spark.sql.api.java.UDF1;

import scala.collection.Seq;

import java.io.Serializable;
import java.util.List;
public class udf implements Serializable {
    private static final long serialVersionUID = 1L;
    @SuppressWarnings("serial")
    UDF1<Seq<Float>, Vector> toVector = new UDF1<Seq<Float>, Vector>(){
    public Vector call(Seq<Float> t1) throws Exception {

            List<Float> L = scala.collection.JavaConversions.seqAsJavaList(t1);
            double[] DoubleArray = new double[t1.length()];
            for (int i = 0 ; i < L.size(); i++) {
              DoubleArray[i]=L.get(i);
            }
            return Vectors.dense(DoubleArray);
          }
    };
}
```



```
udf uf = new udf();
sparkSession.udf().register("toVector", uf.toVector, new VectorUDT());
Dataset<Row> userAlsFeatureVect =
        userTableInfo.withColumn("featuresVect", functions.callUDF("toVector", userTableInfo.col("features"))).drop("features");
// <UserId,UserIndex,alsfeatures(vector)>
userAlsFeatureVect = userAlsFeatureVect.toDF("UserId", "UserIndex", "alsmodelfeatures");
```

Scaling data:

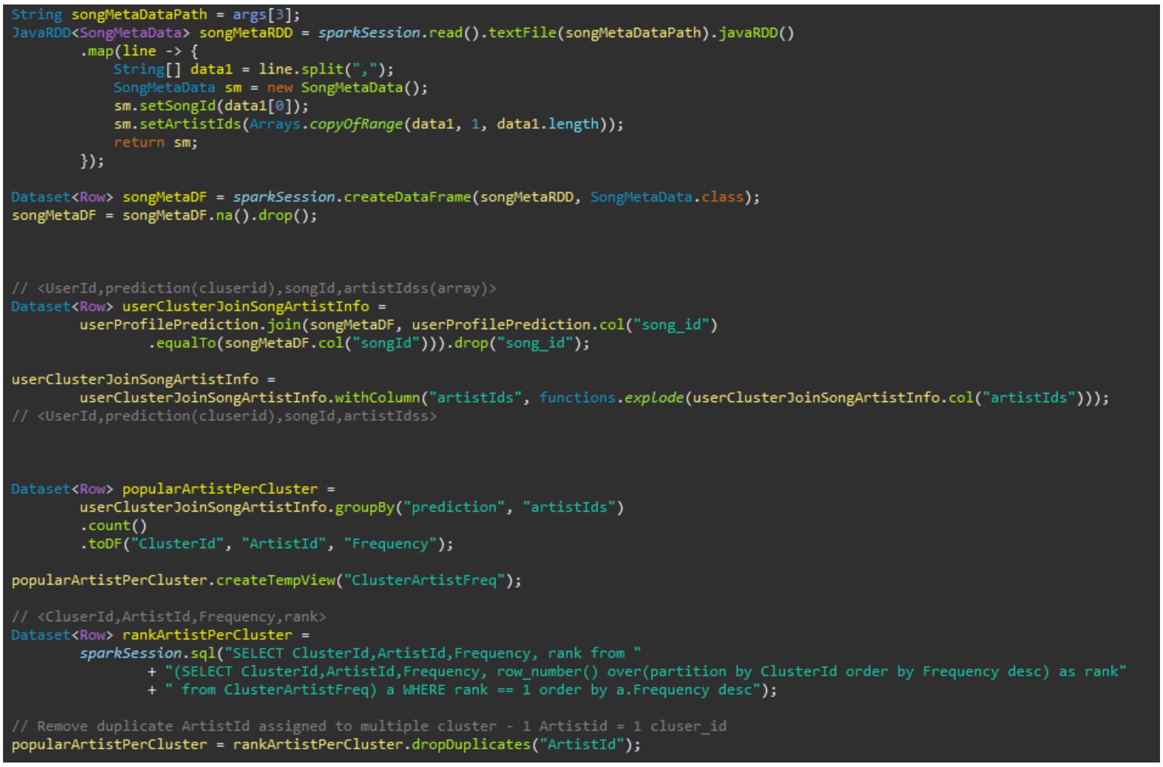
Standard scaler function is used to scale the data to be in a proper range.



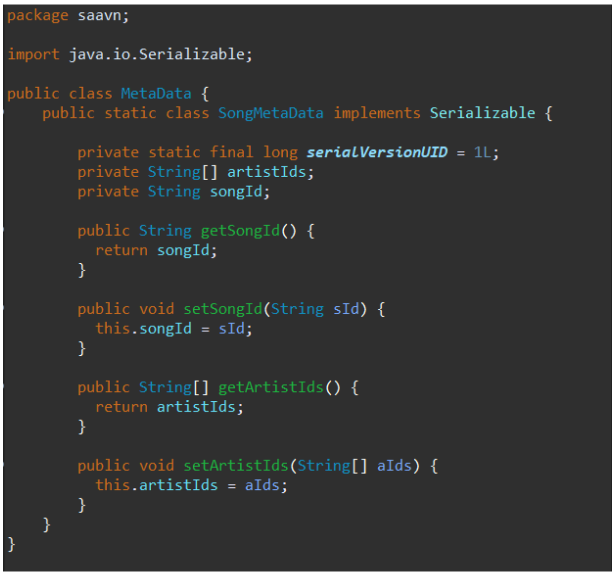
```
StandardScaler scaler = new StandardScaler()
  .setInputCol("alsmodelfeatures")
  .setOutputCol("scaledFseatures")
  .setWithStd(true)
  .setWithMean(true);


StandardScalerModel scalerModel = scaler.fit(userAlsFeatureVect);


Dataset<Row> scaledData = scalerModel.transform(userAlsFeatureVect);
```

Reading song metadata file and creating data frame with song ID and array of artist ID:

- joined the prediction table, user click stream table and song metadata.
- In order to generate the popular artist per cluster, explored function is used to merge the clusters and spark sql function is used to get the popular artist per cluster.

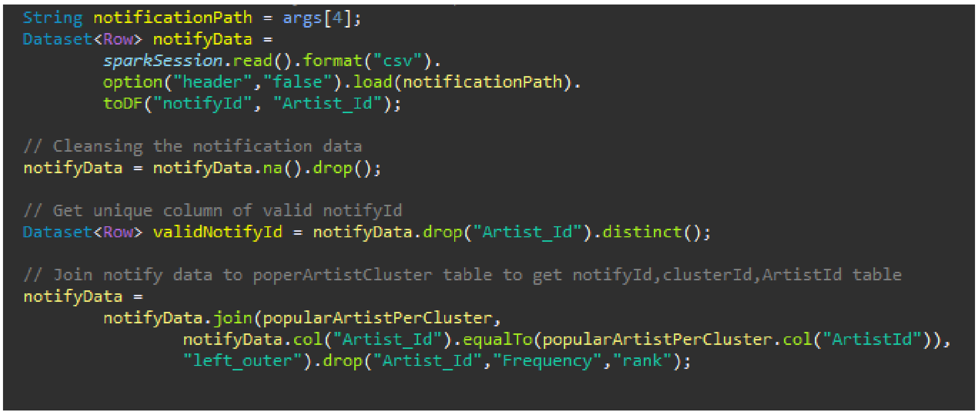


```
String songMetaDataPath = args[3];
JavaRDD<SongMetaData> songMetaRDD = sparkSession.read().textFile(songMetaDataPath).javaRDD()
        .map(line -> {
            String[] data1 = line.split(",");
            SongMetaData sm = new SongMetaData();
            sm.setSongId(data1[0]);
            sm.setArtistIds(Arrays.copyOfRange(data1, 1, data1.length));
            return sm;
        });

Dataset<Row> songMetaDF = sparkSession.createDataFrame(songMetaRDD, SongMetaData.class);
songMetaDF = songMetaDF.na().drop();



// <UserId,prediction(cluserid),songId,artistIdss(array)>
Dataset<Row> userClusterJoinSongArtistInfo =
        userProfilePrediction.join(songMetaDF, userProfilePrediction.col("song_id")
                .equalTo(songMetaDF.col("songId"))).drop("song_id");

userClusterJoinSongArtistInfo =
        userClusterJoinSongArtistInfo.withColumn("artistIds", functions.explode(userClusterJoinSongArtistInfo.col("artistIds")));
// <UserId,prediction(cluserid),songId,artistIdss>



Dataset<Row> popularArtistPerCluster =
        userClusterJoinSongArtistInfo.groupBy("prediction", "artistIds")
        .count()
        .toDF("ClusterId", "ArtistId", "Frequency");

popularArtistPerCluster.createTempView("ClusterArtistFreq");

// <CluserId,ArtistId,Frequency,rank>
Dataset<Row> rankArtistPerCluster =
        sparkSession.sql("SELECT ClusterId,ArtistId,Frequency, rank from "
                + "(SELECT ClusterId,ArtistId,Frequency, row_number() over(partition by ClusterId order by Frequency desc) as rank"
                + " from ClusterArtistFreq) a WHERE rank == 1 order by a.Frequency desc");

// Remove duplicate ArtistId assigned to multiple cluster - 1 Artistid = 1 cluser_id
popularArtistPerCluster = rankArtistPerCluster.dropDuplicates("ArtistId");
```



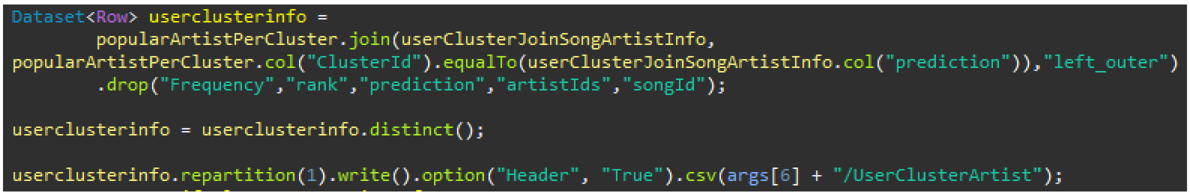
```
package saavn;

import java.io.Serializable;

public class MetaData {
    public static class SongMetaData implements Serializable {

        private static final long serialVersionUID = 1L;
        private String[] artistIds;
        private String songId;

        public String getSongId() {
          return songId;
        }

        public void setSongId(String sId) {
          this.songId = sId;
        }

        public String[] getArtistIds() {
          return artistIds;
        }

        public void setArtistIds(String[] aIds) {
          this.artistIds = aIds;
        }
    }
}
```

Reading notification CSV: After reading notification CSV, join them with the popular artist table.



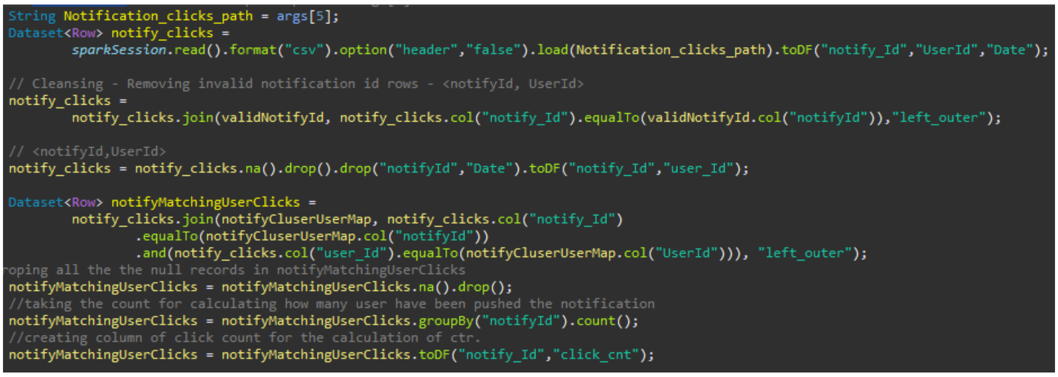
```
String notificationPath = args[4];
Dataset<Row> notifyData =
        sparkSession.read().format("csv").
        option("header","false").load(notificationPath).
        toDF("notifyId", "Artist_Id");

// Cleansing the notification data
notifyData = notifyData.na().drop();

// Get unique column of valid notifyId
Dataset<Row> validNotifyId = notifyData.drop("Artist_Id").distinct();

// Join notify data to poperArtistCluster table to get notifyId,clusterId,ArtistId table
notifyData =
        notifyData.join(popularArtistPerCluster,
                notifyData.col("Artist_Id").equalTo(popularArtistPerCluster.col("ArtistId")),
                "left_outer").drop("Artist_Id","Frequency","rank");
```

Intermediate result generation: this output file contain the UserID, its associated ClusterID, and the popular artist that you have recognised for that cluster.



```
Dataset<Row> userclusterinfo =
        popularArtistPerCluster.join(userClusterJoinSongArtistInfo,
popularArtistPerCluster.col("ClusterId").equalTo(userClusterJoinSongArtistInfo.col("prediction")),"left_outer")
        .drop("Frequency","rank","prediction","artistIds","songId");

userclusterinfo = userclusterinfo.distinct();

userclusterinfo.repartition(1).write().option("Header", "True").csv(args[6] + "/UserClusterArtist");
```

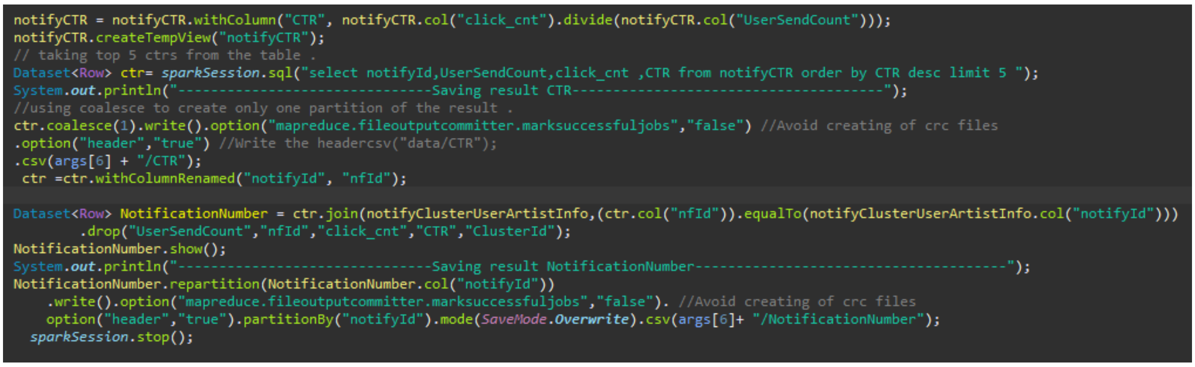
Reading notification clicks: After reading the notification clicks, we count the number of users grouped by notification ID and user ID as a click count.



```
String Notification_clicks_path = args[5];
Dataset<Row> notify_clicks =
        sparkSession.read().format("csv").option("header","false").load(Notification_clicks_path).toDF("notify_Id","UserId","Date");

// Cleansing - Removing invalid notification id rows - <notifyId, UserId>
notify_clicks =
        notify_clicks.join(validNotifyId, notify_clicks.col("notify_Id").equalTo(validNotifyId.col("notifyId")),"left_outer");

// <notifyId,UserId>
notify_clicks = notify_clicks.na().drop().drop("notifyId","Date").toDF("notify_Id","user_Id");

Dataset<Row> notifyMatchingUserClicks =
        notify_clicks.join(notifyCluserUserMap, notify_clicks.col("notify_Id")
                .equalTo(notifyCluserUserMap.col("notifyId"))
                .and(notify_clicks.col("user_Id").equalTo(notifyCluserUserMap.col("UserId"))), "left_outer");
roping all the the null records in notifyMatchingUserClicks
notifyMatchingUserClicks = notifyMatchingUserClicks.na().drop();
//taking the count for calculating how many user have been pushed the notification
notifyMatchingUserClicks = notifyMatchingUserClicks.groupBy("notifyId").count();
//creating column of click count for the calculation of ctr.
notifyMatchingUserClicks = notifyMatchingUserClicks.toDF("notify_Id","click_cnt");
```

For model evaluation, we have to find the CTR (click through rate) with respect to the notification sent to the users as per the model made.

- The result of top 5 CTR in the form of CSV are saved by using write and CSV function.
- In order to get the notification number, we need to perform broadcast join on the table CTR and popular artist table.

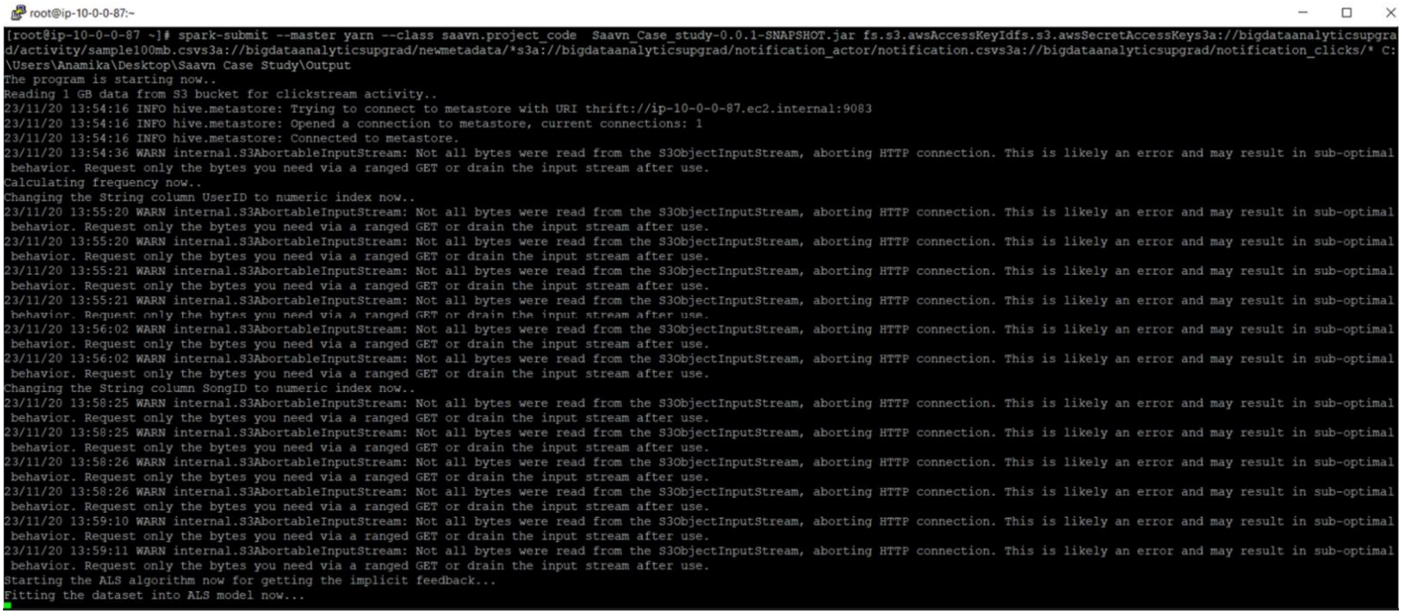


```
notifyCTR = notifyCTR.withColumn("CTR", notifyCTR.col("click_cnt").divide(notifyCTR.col("UserSendCount")));
notifyCTR.createTempView("notifyCTR");
// taking top 5 ctrs from the table .
Dataset<Row> ctr= sparkSession.sql("select notifyId,UserSendCount,click_cnt ,CTR from notifyCTR order by CTR desc limit 5 ");
System.out.println("-------------------------------Saving result CTR--------------------------------------");
//using coalesce to create only one partition of the result .
ctr.coalesce(1).write().option("mapreduce.fileoutputcommitter.marksuccessfuljobs","false") //Avoid creating of crc files
.option("header","true") //Write the headercsv("data/CTR");
.csv(args[6] + "/CTR");
 ctr =ctr.withColumnRenamed("notifyId", "nfId");

Dataset<Row> NotificationNumber = ctr.join(notifyClusterUserArtistInfo,(ctr.col("nfId")).equalTo(notifyClusterUserArtistInfo.col("notifyId")))
        .drop("UserSendCount","nfId","click_cnt","CTR","ClusterId");
NotificationNumber.show();
System.out.println("-------------------------------Saving result NotificationNumber--------------------------------------");
NotificationNumber.repartition(NotificationNumber.col("notifyId"))
    .write().option("mapreduce.fileoutputcommitter.marksuccessfuljobs","false"). //Avoid creating of crc files
    option("header","true").partitionBy("notifyId").mode(SaveMode.Overwrite).csv(args[6]+ "/NotificationNumber");
  sparkSession.stop();
```

Code Execution

Command to execute the Jar file:

spark-submit --master yarn --class saavn.project_code Saavn_Case_study-0.0.1-SNAPSHOT.jar fs.s3.awsAccessKeyIdfs.s3.awsSecretAccessKeys3a://bigdataanalyticsupgrad/activity/sample100mb.csvs3a://bigdataanalyticsupgrad/newmetadata/*s3a://bigdataanalyticsupgrad/notification_actor/notification.csvs3a://bigdataanalyticsupgrad/notification_clicks/* C:\Users\Anamika\Desktop\Saavn Case Study\Output



```
[root@ip-10-0-0-87 ~]# spark-submit --master yarn --class saavn.project_code  Saavn_Case_study-0.0.1-SNAPSHOT.jar fs.s3.awsAccessKeyIdfs.s3.awsSecretAccessKeys3a://bigdataanalyticsupgra
d/activity/sample100mb.csvs3a://bigdataanalyticsupgrad/newmetadata/*s3a://bigdataanalyticsupgrad/notification_actor/notification.csvs3a://bigdataanalyticsupgrad/notification_clicks/* C:
\Users\Anamika\Desktop\Saavn Case Study\Output
The program is starting now..
Reading 1 GB data from S3 bucket for clickstream activity..
23/11/20 13:54:16 INFO hive.metastore: Trying to connect to metastore with URI thrift://ip-10-0-0-87.ec2.internal:9083
23/11/20 13:54:16 INFO hive.metastore: Opened a connection to metastore, current connections: 1
23/11/20 13:54:16 INFO hive.metastore: Connected to metastore.
23/11/20 13:54:36 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
Calculating frequency now..
Changing the String column UserID to numeric index now..
23/11/20 13:55:20 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:55:20 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:55:21 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:55:21 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:56:02 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:56:02 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
Changing the String column SongID to numeric index now..
23/11/20 13:58:25 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:58:25 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:58:26 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:58:26 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:59:10 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
23/11/20 13:59:11 WARN internal.S3AbortableInputStream: Not all bytes were read from the S3ObjectInputStream, aborting HTTP connection. This is likely an error and may result in sub-optimal
 behavior. Request only the bytes you need via a ranged GET or drain the input stream after use.
Starting the ALS algorithm now for getting the implicit feedback...
Fitting the dataset into ALS model now...
```