Project by UPGrad
Read me File

Summary

Overview

My project was to build a model which give users an update about the new songs launched in the segment of their music preferences.
The data given to me 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"

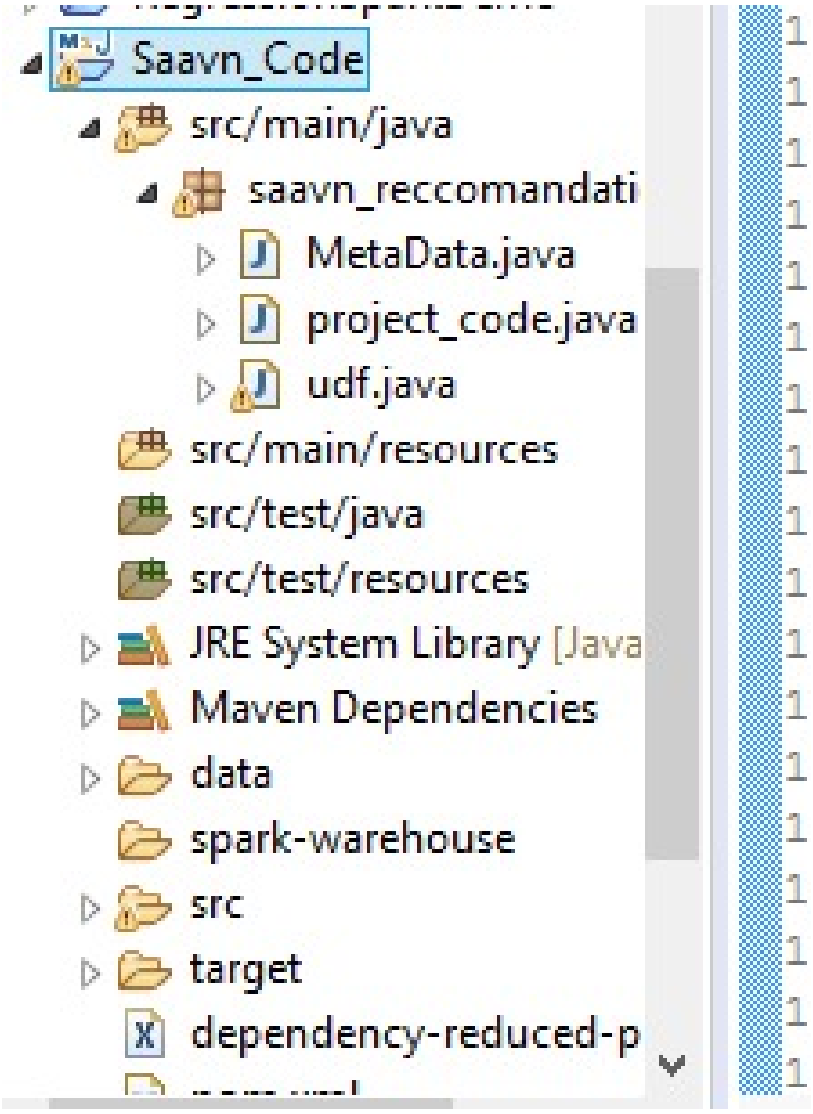
Approach:
Data preparation with **Collaborative filtering** which is a technique to build personalised recommendation on the web.

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.

Program snapshot and the command to run the code

Sharing the details of the Spark program:
Created the spark program with the main file as project code, UDF to convert feature array to vector. Metadata java to read the metadata file.

Command to execute the Jar file:
spark2-submit --master yarn --deploy-mode client --class saavn_reccomandation_system.project_code /home/ec2-user/Saavn_code-0.0.1-SNAPSHOT.jar AKIA5YBLCD6MVW7RSYYR jO89KuMtyiXxX4iBfzwX5sjSGvFsUdwJpfpj2ir/ s3a://bigdataanalyticsupgrad/activity/sample100mb.csv s3a://bigdataanalyticsupgrad/newmetadata/* s3a://bigdataanalyticsupgrad/notification_actor/notification.csv s3a://bigdataanalyticsupgrad/notification_clicks/* s3a://saavnrecommendatiosys/output3



Project classes, data preparation & model building

Project Code Class: This is the first primary class which includes-

- Creation of spark session: Spark session contains the access key and the secret key for the S3 bucket in the configuration. Due to the performing broadcast join, I have added an additional configuration **("spark.sql.broadcastTimeout", "36000")** for getting the notification number CSVs.
  It includes master (yarn) to make the execution engine as yarn.

```
public class project_code {


    public static SparkSession sparkSession;

    public static void main( String[] args ) throws AnalysisException
    {

        Logger.getLogger("org").setLevel(Level.OFF);
        Logger.getLogger("akka").setLevel(Level.OFF);

    if (args.length != 7) {
            System.out.println(
                "Usage: spark2-submit --master yarn --class saavn_reccomandation_system.project_code  SaavnAnalytics-0.0.1-SNAPSHOT.jar "
                        + "fs.s3.awsAccessKeyId"                                //args[0]
                        + "fs.s3.awsSecretAccessKey"                            //args[1]
                        + "s3a://bigdataanalyticsupgrad/activity/sample100mb.csv"   //args[2]
                        + "s3a://bigdataanalyticsupgrad/newmetadata/*"          //args[3]
                        + "s3a://bigdataanalyticsupgrad/notification_actor/notification.csv" //args[4]
                        + "s3a://bigdataanalyticsupgrad/notification_clicks/*"  //args[5]
                        + "output_dir_path"); //arg[6]
            return;
        }

        // setting up connection with spark
        sparkSession = SparkSession.builder()
                .config("fs.s3.awsAccessKeyId", args[0])
                .config("fs.s3.awsSecretAccessKey", args[1])
                .config("spark.sql.broadcastTimeout", "36000")
                .appName("RecomendationSystem")
                .master("yarn")
                .getOrCreate();
```

Data preparation for model building: In this step-

- Reading the used click stream CSV with the help of read function of spark data file.
- Dropped timestamp and date column.
- Dropped all the null values.
- Further, taking the count of songs grouped by used ID and song ID.
- By using string indexer, I have converted the user ID and song ID column to integer as ALS does not accept **STRING VALUES.**
- After this, dropped user ID and Song ID columns

```
    userProfile = userProfile.drop("TimeStamp", "Date");
    userProfile = userProfile.na().drop();

    //Get the song frequency per user by group by operation
    Dataset<Row> userRatings =
            userProfile.groupBy("UserId", "SongId")
            .count()
            .toDF("UserId", "SongId", "Frequency");

    //Create UserIndex(Integer Type) for string UserId column to use in ALS model
    StringIndexer indexer = new StringIndexer()
            .setInputCol("UserId")
            .setOutputCol("UserIndex");

    //Table columns - UserId, SongId, Frequency, UserIndex
    Dataset<Row> userIndexed = indexer.fit(userRatings).transform(userRatings);

    //Create SongIndex(Integer Type) for string SongId column to use in ALS model
    indexer.setInputCol("SongId").setOutputCol("SongIndex");

    //Table columns - UserId, SongId, Frequency, UserIndex
    Dataset<Row> songIndexed =
            indexer.fit(userIndexed).transform(userIndexed);

    //Cast UserIndex, SongIndex to Interger Type to use in ALS model
    // <UserId,UserIndex,SongId,SongIndex,Frequency>
    Dataset<Row> modelIndexed = songIndexed
            .withColumn("UserIndex", col("UserIndex").cast(DataTypes.IntegerType))
            .withColumn("SongIndex", col("SongIndex").cast(DataTypes.IntegerType));
```

Project classes, data preparation & model building

Collaborative Filtering: Using ALS - I'm creating implicit features of the users.
The parameters used here are:

- **Ranks:** which refers the presumed latent or hidden factors. To drive the data, I have to guess more underlying factors. The more we use the better result we get up to the point. Hence, I have taken rank as 10 to begin with.
- **Max Iterations**: which is the number of iterations of ALS to run.
- **RegParam** : which specifies the regularization parameter
- Using the user factors function, I have got the feature array and the ID for the K-means algorithm.

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

// Get the userFactors from ALS model to use it in kmeans
Dataset<Row> userALSFeatures = model.userFactors();
```

UDF Creation and Registration for Converting feature array to vector:

```
package saavn_reccomandation_system;
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;
    UDF1<Seq<Float>, Vector> toVector = new UDF1<Seq<Float>, V
    public Vector call(Seq<Float> t1) throws Exception {

            List<Float> L = scala.collection.JavaConversions.s
            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");
```

Project classes, data preparation & model building

Scaling data: Using the standard scaler function, have scaled the data to be in a proper range.

```
//Scale the alsfeatures before giving to kmeans
StandardScaler scaler = new StandardScaler()
  .setInputCol("alsmodelfeatures")
  .setOutputCol("scaledFseatures")
  .setWithStd(true)
  .setWithMean(true);

// Compute summary statistics by fitting the StandardScaler
StandardScalerModel scalerModel = scaler.fit(userAlsFeatureVect);

// Normalize each feature to have unit standard deviation.
Dataset<Row> scaledData = scalerModel.transform(userAlsFeatureVect);
```

Training K-means: Initializing an array of integers to assign them as K value each time and generate multiple K-means models. First, create a K-means object and then generate the K-means model using the .fit method on the scaled data.
Now for the model, we use the .**computeCost method** to compute the **Within Set Sum of Square Error (WSSSE).** Assume that I'm creating seven clusters, now the WSSSE is nothing but the sum of SSE of each cluster formed.

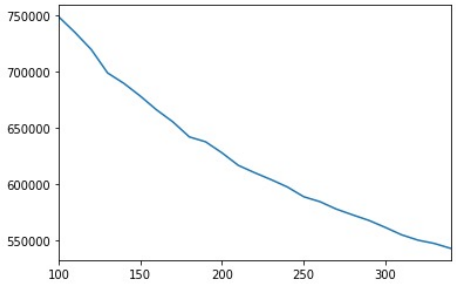
```
    // Trains a k-means model, given array of k's and scaled and non scaled data
//List<Integer> numClusters = Arrays.asList(180,200,230,240,260,280,300);
/*List<Integer> numClusters = Arrays.asList(7);
for (Integer k : numClusters) {
    KMeans kmeans = new KMeans().setK(k).setSeed(1L);
    KMeansModel modelk = kmeans.fit(scaledData);

    //Within Set Sum of Square Error (WESSE).
    double WSSSE = modelk.computeCost(scaledData);
    System.out.println("WSSSE = " + WSSSE);

    //s the results

    Vector[] centers = modelk.clusterCenters();
    System.out.println("Cluster Centers for k: " + k + " ");
    for (Vector center: centers) {
      System.out.println(center);
    }

}*/
```



Project classes, data preparation & model building

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.
- to generate the popular artist per cluster, used explored function to merge the clusters and have used the spark sql function 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_reccomandation_system;

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;
        }
    }
}
```

Project classes

Reading notification CSV: After reading notification CSV, joining 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");
```

Generating the intermediate result: 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, counting 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");
droping 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");
```

Model Evaluation

For model evaluation, I had to find the CTR (click through rate) with respect to the notification sent to the users as per the model made.
-- Saved the result of top 5 CTR in the form of CSV by using **Write & CSV function**.
-- to get the notification number performed 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");
performing broadcast join from ctr to notifyClusterUserArtistInfo to get only top 5 notification <Userid, artistid>
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();
```

References

To create UDF, have refer to this link:
https://stackoverflow.com/questions/52927303/convert-array-to-densevector-in-spark-dataframe-using-java

For broadcast join:
https://stackoverflow.com/questions/41123846/why-does-join-fail-with-java-util-concurrent-timeoutexception-futures-timed-ou

For K-means:
https://learn.upgrad.com/v/course/331/session/33965/segment/180225

Implementation of collaborative filtering:
https://spark.apache.org/docs/2.2.0/ml-collaborative-filtering.html