FIT5202 : Music Recommendation using Collaborative Filtering¶

Collaborative filtering (CF) is a technique commonly used to build personalized recommendations on the Web. Some popular websites that make use of the collaborative filtering technology include Amazon, Netflix, iTunes, IMDB, LastFM, Delicious and StumbleUpon. In collaborative filtering, algorithms are used to make automatic predictions about a user's interests by compiling preferences from several users.
In this lab, our task is to use a collaborative algorithm to recommend top artists from the given dataset. The dataset can be downloaded from Moodle.
Complete the required tasks in the tutorial. The activited is denoted as "Task" with the required instructions

Table of Contents¶

Including Libraries and Initializing Spark Context¶

In [264]:
```
#import libraries
from pyspark import SparkContext
from pyspark.ml.recommendation import ALS
from pyspark.sql import SparkSession ,Row
from pyspark.sql.functions import col,split
from pyspark.sql.types import StructType,StructField,IntegerType


appName="Collaborative Filtering with PySpark"
#initialize the spark session
spark = SparkSession.builder.appName(appName).getOrCreate()
#get sparkcontext from the sparksession
sc = spark.sparkContext
```

Alternating Least Squares DEMO ¶

Please go through the ALS Demo presented in the Lecture to understand the basic flow before starting with the lab tasks.

Use-Case : Music Recommendation ¶

The goal here is to use the data provided to create a recommendation system using collaborative filtering using the social influence data and predict artists a user might like but have not listened to.

> Consider the following example. If user A is a neighbor of user B, and they have similar musical tastes, then there is a very strong tie between them. If user B is a big fan of artist C, and has scrobbled them numerous times, then there is also a strong tie between them. Based on last.fm's data, user A has not yet listened to artist C (no link has formed between them yet), and there is a good chance that user A will also like artist C.Ref

The original dataset is available at last.fm api. The dataset provided here is a lighter version, resized for the sake of simplicity. The dataset contains three files as follows:

- **user_artist_data.txt** 3 columns: `user_id, artist_id, playcount`
- **artist_data.txt** 2 columns: `artist_id ,artist_name`
- **artist_alias.txt** 2 columns: `bad_id, good_id` [known incorrectly spelt artists and the correct artist id].

1. Lab Task: Import the two other files (user_artist_data.txt and artist_alias.txt) to create two dataframes `df_artist_alias` and `df_user_artist`

NOTE: Check the **delimiter** used in these files. `\t` may not be used for all files. </div>.

Data Loading ¶

In [266]:

```
df = spark.read.text("artist_data.txt")
split_col = split(df['value'], '\t')
df = df.withColumn('artist_id', split_col.getItem(0))
df = df.withColumn('artist_name', split_col.getItem(1))
df_artist=df.drop('value')
```

In [265]:

```
#Load user_artist_data.txt to a dataframe called df_user_artist
```

In [267]:

```
#Load artist_alias.txt to a dataframe called df_artist_alias
```

Data Preparation ¶

The `df_user_artist` contains **bad ids**, so the **bad ids** in the `user_artist_data.txt` file need to be remapped to **goodids**. The mapping of **bad_ids** to **good_ids** is in **artist_alias.txt** file. The first task is to create a dictionary of the artist_alias, so that it can be passed over a **broadcast variable**.

Broadcast makes Spark send and hold in memory just one copy for each executor in the cluster. When there are thousands of tasks, and many execute in parallel on each executor, this can save significant network traffic and memory. But you cannot directly broadcast a dataframe, it has to be converted to a list first

In [268]:

```
#After loading the artist_alias data to the dataframe, it is converted to a
dictionary to be set as a broadcast variable
artist_alias = dict(df_artist_alias.collect())
```

2. Lab Task: For the dictionary **artist_alias** which contains key value pair of badid and goodid, create a broadcast variable called **bArtistAlias**.

In [1]:

```
#Write the code below to create a broadcast variable bArtistAlias
```

```
# Hints: apply sc.broadcast on artist_alias
```

After the broadcast variable is created, a function to replace the badids by looking up the values from the broadcasted dictionary is implemented for the userArtistRDD.

In [270]:

```
from pyspark.sql.functions import udf, struct
def lookup_correct_id(artist_id):
    finalArtistID = bArtistAlias.value.get(artist_id)
    if finalArtistID is None:
        finalArtistID = artist_id
    return finalArtistID

lookup_udf = udf(lookup_correct_id, StringType())
```

3. Lab Task: Apply the udf `lookup_udf` on the column of 'artist_id' to replace the "badids" in the `df_user_artist` dataframe.

In [271]:

```
#Write your code below
```

When we want to repeteadly access a dataframe or an RDD, it is a good idea to cache them, it helps to speed up applications.

In [272]:

```
#Uncomment this to use caching
df_user_artist.cache()
```

Out[272]:

```
DataFrame[user_id: string, artist_id: string, playcount: string]
```

Data Exploration ¶

4. Lab Task: Write a query in the function below, to return top **N** artist for a user with user_id : `2062243`. You will need to join `user` and `user_artist` datasets on the common key `artist_id`. The sample output is given below.

In [321]:

```
def top_n_artists(artist,user_artist,user_id,limit):
    '''Returns top n artists liked by a particular user'''
    '''Takes artist,user_artist, user_id and limit as input'''

    df = artist.join(user_artist,artist.artist_id==user_artist.artist_id)\
            .filter(user_artist.user_id==user_id)\
            .sort(user_artist.playcount.desc())\
            .select(user_artist.user_id,user_artist.playcount,artist.artist_name)\
            .limit(limit)
    return df

top_n_artists(df_artist,df_user_artist,2062243,60).show(truncate=False)
+-------+---------+---------------------+
|user_id|playcount|artist_name          |
+-------+---------+---------------------+
|2062243|26107    |Music 205            |
|2062243|10314    |Mos Def              |
|2062243|7193     |Morrissey            |
|2062243|6652     |Modest Mouse         |
|2062243|4913     |Mouse on Mars        |
|2062243|3983     |The Movielife        |
|2062243|3658     |The Beatles          |
|2062243|3354     |Led Zeppelin         |
|2062243|2843     |Mogwai               |
|2062243|1888     |Queen                |
|2062243|1718     |Radiohead           |
|2062243|1706     |Motion City Soundtrack|
|2062243|1700     |Talib Kweli         |
|2062243|1470     |Mudvayne            |
|2062243|1359     |Kanye West          |
|2062243|1348     |Jackson Browne      |
|2062243|1257     |Bob Dylan           |
|2062243|1257     |moe.                |
|2062243|1147     |David Bowie         |
|2062243|1079     |The Killers         |
+-------+--------+---------------------+
only showing top 20 rows
```

Here we want to convert data types to integer type where required.

In [275]:

```
#Cast the data column into integer types
for col_name in df_user_artist.columns:
    df_user_artist = df_user_artist.withColumn(col_name,
df_user_artist[col_name].cast(IntegerType()))

df_artist = df_artist.withColumn('artist_id',
df_artist['artist_id'].cast(IntegerType()))
```

In [276]:

```
df_user_artist.printSchema()
root
 |-- user_id: integer (nullable = true)
 |-- artist_id: integer (nullable = true)
 |-- playcount: integer (nullable = true)
```

In [196]:

```
df_artist.printSchema()
root
 |-- artist_id: integer (nullable = true)
 |-- artist_name: string (nullable = true)
```

Train Test Split ¶

5. Lab Task: Create training and testing dataset with a 80/20 split

In [277]:

```
#Write your code here
```

Model Building [REF] ¶

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. spark.ml currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. spark.ml uses the alternating least squares (ALS) algorithm to learn these latent factors. The implementation in spark.ml has the following parameters:

- **numBlocks** is the number of blocks the users and items will be partitioned into in order to parallelize computation (defaults to 10).
- **rank** is the number of latent factors in the model (defaults to 10).
- **maxIter** is the maximum number of iterations to run (defaults to 10).
- **regParam** specifies the regularization parameter in ALS (defaults to 1.0).

- **implicitPrefs** specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data (defaults to false which means using explicit feedback).
- **alpha** is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations (defaults to 1.0).
- **nonnegative** specifies whether or not to use nonnegative constraints for least squares (defaults to false).

In [278]:
```
als = ALS(maxIter=5, implicitPrefs=True, alpha=40,userCol="user_id",
itemCol="artist_id", ratingCol="playcount",
          coldStartStrategy="drop")
```

6. Lab Task: Perform the following tasks.

- Train the model with the training set created from above.
- Then transform use the test data to get the predictions.
- Display the first 20 predictions from the results.

The predictions shown below will be just indicator of how closely a given artist will be to the user's existing preferences

In [1]:
```
#Write your code below
```

Evalutation of ALS ¶

We can evaluate ALS using RMSE (Root Mean Squared Error) using the RegressionEvaluator as shown below.

In [ ]:
```
#Write your code here
from pyspark.ml.evaluation import RegressionEvaluator

evaluator = RegressionEvaluator(metricName="rmse", labelCol="playcount",
                                predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = " + str(rmse))
```

NOTE: If you run the above code, the RMSE you will observe is very high.

For implicit data, RMSE is not a reliable score since, we don't have any reliable feedback over if items are disliked. RMSE requires knowing which items the user dislikes. Spark does not have a readily available solution for to evaluate the implicit data. The following function implements ROEM (Rank Ordering Error Metric) on the prediction data. You can refer to the details about this here.

In [280]:
```
def ROEM(predictions, userCol = "userId", itemCol = "songId", ratingCol =
"num_plays"):
    #Creates table that can be queried
    predictions.createOrReplaceTempView("predictions")

    #Sum of total number of plays of all songs
    denominator = predictions.groupBy().sum(ratingCol).collect()[0][0]

    #Calculating rankings of songs predictions by user
    spark.sql("SELECT " + userCol + " , " + ratingCol + " , PERCENT_RANK() OVER
(PARTITION BY " + userCol + " ORDER BY prediction DESC) AS rank FROM
predictions").createOrReplaceTempView("rankings")

    #Multiplies the rank of each song by the number of plays and adds the products
```

```
together
    numerator = spark.sql('SELECT SUM(' + ratingCol + ' * rank) FROM
rankings').collect()[0][0]

    performance = numerator/denominator

    return performance
```

In [ ]:

```
ROEM(predictions,'user_id','artist_id','playcount')
```

Hyperparameter tuning and cross validation ¶

Since we can't use RMSE as the evaluation metric for "implicit data", we need to manually implement the hyperparameter tuning for the ALS. This code is adapted from the following source [ref]

`alpha` is an important hyper-parameter for ALS with implicit feedback. It governs the baseline confidence in preference observations. It is a way to assign a confidence values to the `playcount`. Higher `playcount` would mean that we have higher confidence that the user likes that artist and lower `playcount` would mean the user doesn't like that much.

In [317]:

```
def ROEM_cv(df, userCol = "user_id", itemCol = "artist_id", ratingCol = "playcount",
ranks = [10], maxIters = [10], regParams = [.05], alphas = [10, 40]):

    from pyspark.sql.functions import rand
    from pyspark.ml.recommendation import ALS

    ratings_df = df.orderBy(rand()) #Shuffling to ensure randomness

    #Building train and validation test sets
    train, validate = df.randomSplit([0.8, 0.2], seed = 0)

    #Building 3 folds within the training set
    test1, test2,test3 = train.randomSplit([0.33,0.33,0.33], seed = 1)
    train1 = test2.union(test3)
    train2 = test1.union(test2)
    train3 = test1.union(test3)


    #Creating variables that will be replaced by the best model's hyperparameters for
subsequent printing
    best_validation_performance = 9999999999999
    best_rank = 0
    best_maxIter = 0
    best_regParam = 0
    best_alpha = 0
    best_model = 0
    best_predictions = 0

      #Looping through each combindation of hyperparameters to ensure all
combinations are tested.
    for r in ranks:
        for mi in maxIters:
            for rp in regParams:
                for a in alphas:
                #Create ALS model
```

```
                    als = ALS(rank = r, maxIter = mi, regParam = rp, alpha = a,
userCol=userCol, itemCol=itemCol, ratingCol=ratingCol,
                            coldStartStrategy="drop", nonnegative = True,
implicitPrefs = True)

                    #Fit model to each fold in the training set
                    model1 = als.fit(train1)
                    model2 = als.fit(train2)
                    model3 = als.fit(train3)

                    #Generating model's predictions for each fold in the test set
                    predictions1 = model1.transform(test1)
                    predictions2 = model2.transform(test2)
                    predictions3 = model3.transform(test3)

                    #Expected percentile rank error metric function
                    def ROEM(predictions, userCol = userCol, itemCol = itemCol,
ratingCol = ratingCol):
                        #Creates table that can be queried
                        predictions.createOrReplaceTempView("predictions")

                        #Sum of total number of plays of all songs
                        denominator = predictions.groupBy().sum(ratingCol).collect()
[0][0]

                        #Calculating rankings of songs predictions by user
                        spark.sql("SELECT " + userCol + " , " + ratingCol + " ,
PERCENT_RANK() OVER (PARTITION BY " + userCol + " ORDER BY prediction DESC) AS rank
FROM predictions").createOrReplaceTempView("rankings")

                        #Multiplies the rank of each song by the number of plays and
adds the products together
                        numerator = spark.sql('SELECT SUM(' + ratingCol + ' * rank)
FROM rankings').collect()[0][0]

                        performance = numerator/denominator

                        return performance

                    #Calculating expected percentile rank error metric for the model
on each fold's prediction set
                    performance1 = ROEM(predictions1)
                    performance2 = ROEM(predictions2)
                    performance3 = ROEM(predictions3)


                    #Printing the model's performance on each fold
                    print("Model Parameters: \nRank:", r,"\nMaxIter:", mi, "\
nRegParam:",rp,"\nAlpha: ",a)
                    print("Test Percent Rank Errors: ", performance1, performance2,
performance3)

                    #Validating the model's performance on the validation set
                    validation_model = als.fit(train)
                    validation_predictions = validation_model.transform(validate)
                    validation_performance = ROEM(validation_predictions)

                    #Printing model's final expected percentile ranking error metric
                    print("Validation Percent Rank Error: "), validation_performance
                    print(" ")

                    #Filling in final hyperparameters with those of the best-
performing model
                    if validation_performance < best_validation_performance:
                        best_validation_performance = validation_performance
                        best_rank = r
                        best_maxIter = mi
                        best_regParam = rp
                        best_alpha = a
                        best_model = validation_model
                        best_predictions = validation_predictions

    #Printing best model's expected percentile rank and hyperparameters
    print ("**Best Model** ")
    print ("  Percent Rank Error: ", best_validation_performance)
    print ("  Rank: ", best_rank)
    print ("  MaxIter: ", best_maxIter)
    print ("  RegParam: ", best_regParam)
    print ("  Alpha: ", best_alpha)

    return best_model, best_predictions
```

In [ ]:

```
ROEM_cv(df_user_artist)
```

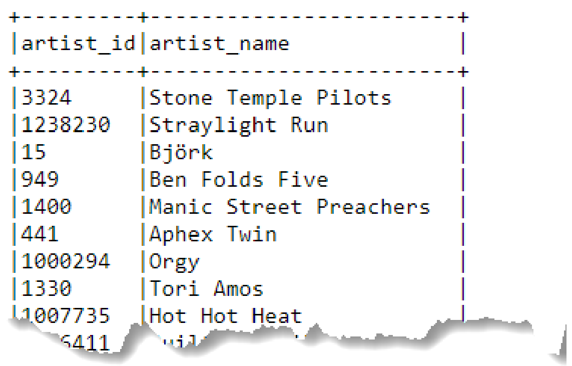
Making Predictions ¶

The k-fold validation implement might take long time to run. You can use the initial ALS model to make the predictions. Assuming you have successfully trained the model, we want to now use the model to **find top Artists recommended for each user**. We can use the ***recommendForAllUsers*** function available in the ALS model to get the list of top recommendations for each users. You can further explore the details of the API here.
The `recommendForAllUsers` only gives the list of artist_ids for the users, you can write the code to map these artist_ids back to their names.

In [ ]:

```
model.recommendForAllUsers(10).show(truncate=False)
```

7. Lab Task: Write a function to find the top **N** recommended artists for the user : **2062243**. Display `artist_id and artist_name` both. A sample output is given below.



```
+---------+------------------------+
|artist_id|artist_name             |
+---------+------------------------+
|3324     |Stone Temple Pilots     |
|1238230  |Straylight Run          |
|15       |Björk                   |
|949      |Ben Folds Five          |
|1400     |Manic Street Preachers  |
|441      |Aphex Twin              |
|1000294  |Orgy                    |
|1330     |Tori Amos               |
|1007735  |Hot Hot Heat            |
```

In [323]:

```
def recommendedArtists(als_model,user_id,limit):
    #get the recommendations
    selected_user = model.recommendForAllUsers(limit).filter(col('user_id')==user_id)

    #unnest the recommendations

    #join the top_artist dataframe with the artist master dataframe to include the
artist_name

    return final
```

In [ ]:

```
#Write your code here
recommendedArtists(model,2062243,20).show(truncate=False)
```

Congratulations on finishing this activity. See you next week.¶