# Homework Writeup

## //--Preparing the Data--

val rawUserArtistData = //reading user\_artist\_data into rawUserArtistData

spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/user\_artist\_data.txt")

rawUserArtistData.take(5).foreach(println) //take the top 5 and println each

/\*

rawUserArtistData: org.apache.spark.sql.Dataset[String] = [value: string]

1000002 1 55

1000002 1000006 33

1000002 1000007 8

1000002 1000009 144

1000002 1000010 314

\*/

val userArtistDF = rawUserArtistData.map { line => //for each line of rawUserArtistData

val Array(user, artist, \_\*) = line.split(' ') //create an array with user, artist, and unspecified by splitting line at space

(user.toInt, artist.toInt) //convert user and artist id to int

}.toDF("user", "artist") //save as a dataframe, with those two names respectively as columns

userArtistDF.agg( //do the following aggregations

min("user"), max("user"), min("artist"), max("artist")).show()

/\*

userArtistDF: org.apache.spark.sql.DataFrame = [user: int, artist: int]

+---------+---------+-----------+-----------+

|min(user)|max(user)|min(artist)|max(artist)|

+---------+---------+-----------+-----------+

| 90| 2443548| 1| 10794401| //values are small enough to leave as ints! (since max int value = 2147483647)

+---------+---------+-----------+-----------+

\*/

val rawArtistData = spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/artist\_data.txt")

/\* This will error!

rawArtistData.map { line =>

val (id, name) = line.span(\_ != '\T') //split line at the very first tab

(id.toInt, name.trim) //convert id to int and trim name

}.count() //this breaks when we call count, since some lines are corrupted (no tab or newline character)

\*/

val artistByID = rawArtistData.flatMap { line => //use flatMap instead since it "flattens" collection or results into one dataset!

val (id, name) = line.span(\_ != '\t') //split line at the very first tab

if (name.isEmpty) {

None //return none if name empty

} else {

try { //try to convert!!!

Some((id.toInt, name.trim)) //using the option class, we can use some to simplify instead returning empty element

} catch { //catch le error

case \_: NumberFormatException => None //if can't convert due to corruption, return none

}

}

}.toDF("id", "name") //save column names as

/\*

artistByID: org.apache.spark.sql.DataFrame = [id: int, name: string]

[1134999,06Crazy Life]

[6821360,Pang Nakarin]

[10113088,Terfel, Bartoli- Mozart: Don]

[10151459,The Flaming Sidebur]

[6826647,Bodenstandig 3000]

\*/

val rawArtistAlias = spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/artist\_alias.txt")

val artistAlias = rawArtistAlias.flatMap { line => //use flatMap instead since it "flattens" collection or results into one dataset!

val Array(artist, alias) = line.split('\t') //split line at the very first tab

if (artist.isEmpty) { //return none if artist id missing

None

} else {

Some((artist.toInt, alias.toInt)) //try to make le value

}

}.collect().toMap //collect all since it doesnt specify and convert to map

artistAlias.head

/\*

artistAlias: scala.collection.immutable.Map[Int,Int] = Map(1208690 -> 1003926, 2012757 -> 4569, 6949139 -> 1085752, 1109727 -> 1239120, 6772751 -> 1244705, 2070533 -> 1021544, 1157679 -> 2194, 9969617 -> 5630, 2034496 -> 1116214, 6764342 -> 40, 1272489 -> 1278238, 2108744 -> 1009267, 10349857 -> 1000052, 2145319 -> 1020463, 2126338 -> 2717, 10165456 -> 1001169, 6779368 -> 1239506, 10278137 -> 1001523, 9939075 -> 1329390, 2037201 -> 1274155, 1248585 -> 2885, 1106945 -> 1399, 6811322 -> 1019016, 9978396 -> 1784, 6676961 -> 1086433, 2117821 -> 2611, 6863616 -> 1277013, 6895480 -> 1000993, 6831632 -> 1246136, 1001719 -> 1009727, 10135633 -> 4250, 7029291 -> 1034635, 6967939 -> 1002734, 6864694 -> 1017311, 1237279 -> 1029752, 6793956 -> 1283231, 1208609 -> 1000699, 6693428 -> 1100258, 685174...res66: (Int, Int) = (1208690,1003926)

\*/

artistByID.filter($"id" isin (1208690, 1003926)).show() //filter through id and show these two values

/\*

+-------+----------------+

| id| name|

+-------+----------------+

|1208690|Collective Souls|

|1003926| Collective Soul|

+-------+----------------+

\*/

This section starts off by grabbing information from text and putting them within our variables. We found aggregations of the user\_artist\_data, wrote a try case for artist\_data due to corrupt lines, and learned how alias worked in relation.

## //--Building a First Model--

import org.apache.spark.sql.\_

import org.apache.spark.broadcast.\_

def buildCounts(

rawUserArtistData: Dataset[String], //input is a datset of strings

bArtistAlias: Broadcast[Map[Int,Int]]): DataFrame = { //return a dataframe

rawUserArtistData.map { line => //for each line of rawUserArtistData

val Array(userID, artistID, count) = line.split(' ').map(\_.toInt) //split each line at the space and map em to int

val finalArtistID =

bArtistAlias.value.getOrElse(artistID, artistID) //Get artist’s alias if it exists, otherwise get original artist.

(userID, finalArtistID, count) //return the following tupple

}.toDF("user", "artist", "count") //name the following columns as such

}

val bArtistAlias = spark.sparkContext.broadcast(artistAlias) //Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

//If we just use artistAlias directly, it takes up 15 mg of storage, so instead use broadcast so it hold in memory a copy for each executor

val trainData = buildCounts(rawUserArtistData, bArtistAlias) //THIS IS THE RESULTING DATASET

trainData.cache() //suggest to spark to store and keep in memory in cluster

trainData.take(5).foreach(println)

/\*

import org.apache.spark.sql.\_

import org.apache.spark.broadcast.\_

buildCounts: (rawUserArtistData: org.apache.spark.sql.Dataset[String], bArtistAlias: org.apache.spark.broadcast.Broadcast[Map[Int,Int]])org.apache.spark.sql.DataFrame

bArtistAlias: org.apache.spark.broadcast.Broadcast[scala.collection.immutable.Map[Int,Int]] = Broadcast(43)

trainData: org.apache.spark.sql.DataFrame = [user: int, artist: int ... 1 more field]

res71: trainData.type = [user: int, artist: int ... 1 more field]

[1000002,1,55]

[1000002,1000006,33]

[1000002,1000007,8]

[1000002,1000009,144]

[1000002,1000010,314]

\*/

import org.apache.spark.ml.recommendation.\_

import scala.util.Random

val model = new ALS(). //construct model as an ALSModel

setSeed(Random.nextLong()). //use a random seed

setImplicitPrefs(true).

setRank(10).

setRegParam(0.01).

setAlpha(1.0).

setMaxIter(5).

setUserCol("user").

setItemCol("artist").

setRatingCol("count").

setPredictionCol("prediction").

fit(trainData)

/\*

21/09/09 22:37:43 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS

21/09/09 22:37:43 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS

21/09/09 22:37:43 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeSystemLAPACK

21/09/09 22:37:43 WARN LAPACK: Failed to load implementation from: com.github.fommil.netlib.NativeRefLAPACK

model: org.apache.spark.ml.recommendation.ALSModel = als\_9a0e5c0e9737

\*/

model.userFactors.show(1, truncate = false)

/\*

+---+---------------------------------------------------------------------------------------------------------------------------+

|id |features |

+---+---------------------------------------------------------------------------------------------------------------------------+

|90 |[1.0062273, -0.055548448, -0.53881216, -0.08377492, -0.29344943, -0.10351685, -0.5451834, 0.25785673, 0.6842964, 0.7220503]|

+---+---------------------------------------------------------------------------------------------------------------------------+

only showing top 1 row

vs

scala> model.userFactors.show(1, truncate = true)

+---+--------------------+

| id| features|

+---+--------------------+

| 90|[1.0062273, -0.05...|

+---+--------------------+

only showing top 1 row

\*/

This section we created a function called buildCounts that matches the alias artists to real artists (using getOrElse). We also used broadcast to make read only variables, and made our first ALS model with pretty bad parameters.

## //--Spot Checking Recommendations

val userID = 2093760 //given by the textbook to work with

val existingArtistIDS = trainData.

filter($"user" === userID). //filter lines where user is 2093760 (WHY 3 EQUALS SIGNS)

select("artist").as[Int].collect() //collect all artist the user listens to as ints

artistByID.filter($"id" isin (existingArtistIDS: \_\*)).show() //filter those artists out!

/\*

userID: Int = 2093760

existingArtistIDS: Array[Int] = Array(1180, 1255340, 378, 813, 942)

+-------+---------------+

| id| name|

+-------+---------------+

| 1180| David Gray|

| 378| Blackalicious|

| 813| Jurassic 5|

|1255340|The Saw Doctors|

| 942| Xzibit|

+-------+---------------+

\*/

def makeRecommendations(model: ALSModel, userID: Int, howMany: Int): DataFrame = //the inputs and outputs

{

val toRecommend = model.itemFactors.

select($"id".as("artist")).

withColumn("user", lit(userID)) //select all artist ids and pair with target user id

model.transform(toRecommend).

select("artist", "prediction").

orderBy($"prediction".desc).

limit(howMany) //score all artists, with top score descending, limiting amount to howMany

}

//makeRecommendations: (model: org.apache.spark.ml.recommendation.ALSModel, userID: Int, howMany: Int)org.apache.spark.sql.DataFrame

val topRecommendations = makeRecommendations(model, userID, 5)

topRecommendations.show()

/\*

+-------+-----------+

| artist| prediction|

+-------+-----------+

| 2814|0.026949896|

|1001819|0.026780745|

|1300642|0.025676597|

|1007614|0.025646321|

| 4605|0.025559124|

+-------+-----------+

\*/

val recommendedArtistIDs =

topRecommendations.select("artist").as[Int].collect() //from recommendedAristIDS steal the column artist and save

artistByID.filter($"id" isin (recommendedArtistIDs:\_\*)).show()

/\*

+-------+----------+

| id| name|

+-------+----------+

| 2814| 50 Cent|

| 4605|Snoop Dogg|

|1007614| Jay-Z| //not great recommendation, seems to be all popular general hiphop artists

|1001819| 2Pac|

|1300642| The Game|

+-------+----------+

\*/

This section we grabbed the plays of a user, found their recommendation ids, and converted those ids into artist names.

## //--Evaluating Recommendation Quality--

/\*

The system thinks only the songs that user listens to are good, and all else are bad. So what if we exclude some of the artist plays?

That way, this held out data can be thought of some the best recommendations to the user. We can basically say, 1 is a perfect rec, 0.5

is like a randomly ranking artist, and 0 is worst score. This scoring system relates to a infro retrievel concept called ROC

(receiver operating characteristics) curve. The metric mentioned ealier relates to the ROC's AUC (area under curve). AUC can be read

as the "randomly chosen good rec ranks above randomly chosen bad rec." When using AUC for recomendors, we compute AUC for each user and

find the mean. Another evlaution we can use (within RankingMetrics) is precision, recall, and mean average precision (MAP).

Data is typically divided into 3 sets for machine learning: aining, cross-validation (CV), and test sets

\*/

## //--Computing AUC--

//PROVIDED BY SOURCE CODE

import scala.collection.mutable.ArrayBuffer

def areaUnderCurve(

positiveData: DataFrame,

bAllArtistIDs: Broadcast[Array[Int]],

predictFunction: (DataFrame => DataFrame)): Double = {

// What this actually computes is AUC, per user. The result is actually something

// that might be called "mean AUC".

// Take held-out data as the "positive".

// Make predictions for each of them, including a numeric score

val positivePredictions = predictFunction(positiveData.select("user", "artist")).

withColumnRenamed("prediction", "positivePrediction")

// BinaryClassificationMetrics.areaUnderROC is not used here since there are really lots of

// small AUC problems, and it would be inefficient, when a direct computation is available.

// Create a set of "negative" products for each user. These are randomly chosen

// from among all of the other artists, excluding those that are "positive" for the user.

val negativeData = positiveData.select("user", "artist").as[(Int,Int)].

groupByKey { case (user, \_) => user }.

flatMapGroups { case (userID, userIDAndPosArtistIDs) =>

val random = new Random()

val posItemIDSet = userIDAndPosArtistIDs.map { case (\_, artist) => artist }.toSet

val negative = new ArrayBuffer[Int]()

val allArtistIDs = bAllArtistIDs.value

var i = 0

// Make at most one pass over all artists to avoid an infinite loop.

// Also stop when number of negative equals positive set size

while (i < allArtistIDs.length && negative.size < posItemIDSet.size) {

val artistID = allArtistIDs(random.nextInt(allArtistIDs.length))

// Only add new distinct IDs

if (!posItemIDSet.contains(artistID)) {

negative += artistID

}

i += 1

}

// Return the set with user ID added back

negative.map(artistID => (userID, artistID))

}.toDF("user", "artist")

// Make predictions on the rest:

val negativePredictions = predictFunction(negativeData).

withColumnRenamed("prediction", "negativePrediction")

// Join positive predictions to negative predictions by user, only.

// This will result in a row for every possible pairing of positive and negative

// predictions within each user.

val joinedPredictions = positivePredictions.join(negativePredictions, "user").

select("user", "positivePrediction", "negativePrediction").cache()

// Count the number of pairs per user

val allCounts = joinedPredictions.

groupBy("user").agg(count(lit("1")).as("total")).

select("user", "total")

// Count the number of correctly ordered pairs per user

val correctCounts = joinedPredictions.

filter($"positivePrediction" > $"negativePrediction").

groupBy("user").agg(count("user").as("correct")).

select("user", "correct")

// Combine these, compute their ratio, and average over all users

val meanAUC = allCounts.join(correctCounts, "user").

select($"user", ($"correct" / $"total").as("auc")).

agg(mean("auc")).

as[Double].first()

joinedPredictions.unpersist()

meanAUC

}

//areaUnderCurve: (positiveData: org.apache.spark.sql.DataFrame, bAllArtistIDs: org.apache.spark.broadcast.Broadcast[Array[Int]],

//predictFunction: org.apache.spark.sql.DataFrame => org.apache.spark.sql.DataFrame)Double

val allData = buildCounts(rawUserArtistData, bArtistAlias) //create an array with alias fixed

val Array(trainData, cvData) = allData.randomSplit(Array(0.9, 0.1)) //split the array, 90% into trainData and 10% cvData

trainData.cache() //store to memory

cvData.cache() //store to memory

/\*

allData: org.apache.spark.sql.DataFrame = [user: int, artist: int ... 1 more field]

trainData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [user: int, artist: int ... 1 more field]

cvData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [user: int, artist: int ... 1 more field]

res32: trainData.type = [user: int, artist: int ... 1 more field]

res33: cvData.type = [user: int, artist: int ... 1 more field]

scala> allData.take(5).foreach(println)

[1000002,1,55]

[1000002,1000006,33]

[1000002,1000007,8]

[1000002,1000009,144]

[1000002,1000010,314]

\*/

val allArtistIDs = allData.select("artist").as[Int].distinct().collect() //collect all distinct artists ids

val bAllArtistIDs = spark.sparkContext.broadcast(allArtistIDs) //create a read only

val model = new ALS().

setSeed(Random.nextLong()).

setImplicitPrefs(true).

setRank(10).setRegParam(0.01).setAlpha(1.0).setMaxIter(5).

setUserCol("user").setItemCol("artist").

setRatingCol("count").setPredictionCol("prediction").

fit(trainData)

areaUnderCurve(cvData, bAllArtistIDs, model.transform)

/\*

model: org.apache.spark.ml.recommendation.ALSModel = als\_c65315b0e83d

res38: Double = 0.912093562911161 //AYE WE GOT WAY HIGHER THAN BOOK

\*/

def predictMostListened(train: DataFrame)(allData: DataFrame) = { //model of most listened artists

val listenCounts = train.

groupBy("artist").

agg(sum("count").as("prediction")).

select("artist", "prediction")

allData.

join(listenCounts, Seq("artist"), "left\_outer").

select("user", "artist", "prediction")

}

areaUnderCurve(cvData, bAllArtistIDs, predictMostListened(trainData))

/\*

predictMostListened: (train: org.apache.spark.sql.DataFrame)(allData: org.apache.spark.sql.DataFrame)org.apache.spark.sql.DataFrame

res40: Double = 0.8857297321607271 //HAHAHA NONPERSONALIZED IS LESSS

\*/"""Clearly, the model needs some tuning. Can it be made better?""" //HAHAHAHA NOPE, IM BETTER THAN U

This section grabbed code from the textbook, explored the metric of area under the curve, learned how to split the data into two groups: test and training data, and how to test the resulting new model on the test data.

## //--Hyperparameter Selection

/\* THIS IS TAKEN DIRECTLY FROM THE TEXTBOOK [NO MISTAKES IN MIS UNDERSTANDING]

setRank(10)

The number of latent factors in the model, or equivalently, the number of columns k in the user-feature and product-feature matrices.

In nontrivial cases, this is also their rank.

setMaxIter(5)

The number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.

setRegParam(0.01)

A standard overfitting parameter, also usually called lambda. Higher values resist overfitting, but values that are too high hurt the

factorization’s accuracy.

setAlpha(1.0)

Controls the relative weight of observed versus unobserved user-product interactions in the factorization.

\*/

val evaluations =

for (rank <- Seq(5, 30);

regParam <- Seq(4.0, 0.0001);

alpha <- Seq(1.0, 40.0)) //triple nested loop (different value for each)

yield {

val model = new ALS().

setSeed(Random.nextLong()).

setImplicitPrefs(true).

setRank(rank).setRegParam(regParam). //use test values instead!

setAlpha(alpha).setMaxIter(20).

setUserCol("user").setItemCol("artist").

setRatingCol("count").setPredictionCol("prediction").

fit(trainData)

val auc = areaUnderCurve(cvData, bAllArtistIDs, model.transform)

model.userFactors.unpersist() //Free up model resources immediately

model.itemFactors.unpersist() //Free up model resources immediately

(auc, (rank, regParam, alpha))

}

evaluations.sorted.reverse.foreach(println) //have it printed descending order

/\*

evaluations: Seq[(Double, (Int, Double, Double))] = List((0.9201206231476234,(5,4.0,1.0)), (0.9191436348025518,(5,4.0,40.0)), (0.9132892036807542,(5,1.0E-4,1.0)), (0.9188247842998887,(5,1.0E-4,40.0)), (0.9225502446940053,(30,4.0,1.0)), (0.9234664132404989,(30,4.0,40.0)), (0.9082654521690375,(30,1.0E-4,1.0)), (0.9222690272202381,(30,1.0E-4,40.0)))

(0.9234664132404989,(30,4.0,40.0))

(0.9225502446940053,(30,4.0,1.0))

(0.9222690272202381,(30,1.0E-4,40.0))

(0.9201206231476234,(5,4.0,1.0))

(0.9191436348025518,(5,4.0,40.0))

(0.9188247842998887,(5,1.0E-4,40.0))

(0.9132892036807542,(5,1.0E-4,1.0))

(0.9082654521690375,(30,1.0E-4,1.0))

\*/

//when alpha was 40, did significantly better (model is better of focusing on what the user listened to that not)

//higher regparam, because model may be overfitting

This section took the longest by far because of the triple for loop that tested different values for 3 different parameters. The resulting best parameters were mentioned above!

## //--Making Recommendations

//Testing with best set of Hyperparameters!

val allData = buildCounts(rawUserArtistData, bArtistAlias) //create an array with alias fixed

val Array(trainData2, cvData) = allData.randomSplit(Array(0.9, 0.1)) //split the array, 90% into trainData and 10% cvData

trainData2.cache() //store to memory

val model2 = new ALS().

setSeed(Random.nextLong()).

setImplicitPrefs(true).

setRank(30).setRegParam(4.00).setAlpha(40.0).setMaxIter(5). //using our new best values!

setUserCol("user").setItemCol("artist").

setRatingCol("count").setPredictionCol("prediction").

fit(trainData2)

val topRecommendations2 = makeRecommendations(model2, userID, 5)

val recommendedArtistIDs2 = topRecommendations2.select("artist").as[Int].collect()

artistByID.filter($"id" isin (recommendedArtistIDs2:\_\*)).show()

/\*

+-------+-----------+

| id| name| //WAY BETTER SUGGESTIONS~!

+-------+-----------+

|1034635| [unknown]|

| 930| Eminem|

|1270639|The Killers|

| 4267| Green Day|

|1001412| blink-182|

+-------+-----------+

\*/

val someUsers = allData.select("user").as[Int].distinct().take(100) //snatch 100 distinct users

val someRecommendations = someUsers.map(userID => (userID, makeRecommendations(model, userID, 5))) //for each user make a recommendation

someRecommendations.foreach { case (userID, recsDF) =>

val recommendedArtists = recsDF.select("artist").as[Int].collect() //grab list of artists as recommendedArtists

println(s"$userID -> ${recommendedArtists.mkString(", ")}") //print with user id followed by recommnededArtists

}

/\*

1000190 -> 1003853, 6694932, 1006134, 1006411, 1244362

1001043 -> 1274, 1854, 4267, 1034635, 1205

1001129 -> 1034635, 121, 352, 1307, 189

1001139 -> 15, 2, 313, 1000113, 352

1002431 -> 4267, 3327, 976, 1000139, 979

1002605 -> 1205, 1000113, 1034635, 1274, 1275996

1004666 -> 1034635, 733, 1003084, 1295531, 1247272

1005158 -> 4267, 1001412, 1026440, 1854, 5409

1005439 -> 1000481, 2003588, 4468, 1000183, 1004162

1005697 -> 2814, 1001819, 930, 1300642, 1003249

1005853 -> 1001907, 1233770, 1001531, 1193, 352

1007007 -> 1177, 1002061, 1000113, 4267, 1205

1007847 -> 1000113, 979, 1205, 1000139, 976

1008081 -> 1003965, 1004077, 1235066, 1011262, 1005975

1008233 -> 1300642, 1117311, 1084205, 1008984, 1003249

1008804 -> 1298659, 1105902, 478, 1285410, 1006371

...

\*/

/\*

By swapping the fields of the user and artist, we can actually see the best user for artist

rawArtistData.map { line =>

val (id, name) = line.span(\_ != '\t')

(name.trim, id.int)

}

\*/

With the new and improved ALS model, we gave a new recommendation to the aforementioned user and even created a method of recommending to multiple users at once.

# Extension Writeup

## Question: If the recommender gives a different recommendation based on the used training data, why not use a cross-validation method to recommend to users?

//--My Extension Part 1--

import org.apache.spark.sql.\_

import org.apache.spark.broadcast.\_

import org.apache.spark.ml.recommendation.\_

import scala.util.Random

import scala.collection.mutable.ArrayBuffer

def buildCounts(

rawUserArtistData: Dataset[String], //input is a datset of strings

bArtistAlias: Broadcast[Map[Int,Int]]): DataFrame = { //return a dataframe

rawUserArtistData.map { line => //for each line of rawUserArtistData

val Array(userID, artistID, count) = line.split(' ').map(\_.toInt) //split each line at the space and map em to int

val finalArtistID =

bArtistAlias.value.getOrElse(artistID, artistID) //Get artist’s alias if it exists, otherwise get original artist.

(userID, finalArtistID, count) //return the following tupple

}.toDF("user", "artist", "count") //name the following columns as such

}

def makeRecommendations(model: ALSModel, userID: Int, howMany: Int): DataFrame = //the inputs and outputs

{

val toRecommend = model.itemFactors.

select($"id".as("artist")).

withColumn("user", lit(userID)) //select all artist ids and pair with target user id

model.transform(toRecommend).

select("artist", "prediction").

orderBy($"prediction".desc).

limit(howMany) //score all artists, with top score descending, limiting amount to howMany

}

val rawUserArtistData = //reading user\_artist\_data into rawUserArtistData

spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/user\_artist\_data.txt")

val userArtistDF = rawUserArtistData.map { line => //for each line of rawUserArtistData

val Array(user, artist, \_\*) = line.split(' ') //create an array with user, artist, and unspecified by splitting line at space

(user.toInt, artist.toInt) //convert user and artist id to int

}.toDF("user", "artist") //save as a dataframe, with those two names respectively as columns

val rawArtistData = spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/artist\_data.txt")

val artistByID = rawArtistData.flatMap { line => //use flatMap instead since it "flattens" collection or results into one dataset!

val (id, name) = line.span(\_ != '\t') //split line at the very first tab

if (name.isEmpty) {

None //return none if name empty

} else {

try { //try to convert!!!

Some((id.toInt, name.trim)) //using the option class, we can use some to simplify instead returning empty element

} catch { //catch le error

case \_: NumberFormatException => None //if can't convert due to corruption, return none

}

}

}.toDF("id", "name") //save column names as

val rawArtistAlias = spark.read.textFile("/proj/cse398-498/course/AAS\_CH3/profiledata\_06-May-2005/artist\_alias.txt")

val artistAlias = rawArtistAlias.flatMap { line => //use flatMap instead since it "flattens" collection or results into one dataset!

val Array(artist, alias) = line.split('\t') //split line at the very first tab

if (artist.isEmpty) { //return none if artist id missing

None

} else {

Some((artist.toInt, alias.toInt)) //try to make le value

}

}.collect().toMap //collect all since it doesnt specify and convert to map

val bArtistAlias = spark.sparkContext.broadcast(artistAlias)

//Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

val allData = buildCounts(rawUserArtistData, bArtistAlias) //create an array with alias fixed

Up till this point, the code has predominantly been focused on just initializing all the text files we will be working with, the functions we may need to use, and modifying/creating arrays. Below here, I create an ArrayBuffer to hold a list of recommended artist ids. The reason its here is because I am running a loop below (11 times) to see basically cross-validate the model by giving it a different 90% training code. This way, after it generates 5 new recommendations per run, I can compile a list to see which one shows up the most often, then those would be the “best matches.”

var foo = new ArrayBuffer[Int]()

val start = Calendar.getInstance()

val sh = start.get(Calendar.HOUR)

val sm = start.get(Calendar.MINUTE)

val ss = start.get(Calendar.SECOND)

printf("Start Time: %02d:%02d:%02d\n", sh, sm, ss);

for(i <- 0 to 10) //loop 11 times since unknown will most likely arrive multiple times

{

val Array(trainData2, cvData) = allData.randomSplit(Array(0.9, 0.1)) //split the array, 90% into trainData and 10% cvData

trainData2.cache() //store to memory

val model2 = new ALS().

setSeed(Random.nextLong()).

setImplicitPrefs(true).

setRank(30).setRegParam(4.00).setAlpha(40.0).setMaxIter(5).

setUserCol("user").setItemCol("artist").

setRatingCol("count").setPredictionCol("prediction").

fit(trainData2)

val userID = 2093760

val topRecommendations2 = makeRecommendations(model2, userID, 5)

val recommendedArtistIDs2 = topRecommendations2.select("artist").as[Int].collect()

artistByID.filter($"id" isin (recommendedArtistIDs2:\_\*)).show()

recommendedArtistIDs2.foreach(foo += \_)

}

/\*

+-------+---------------+ //run 1 results

| id| name|

+-------+---------------+

|1034635| [unknown]|

| 1784|Black Eyed Peas|

| 1786| Bob Marley|

| 930| Eminem|

|1001048| Avril Lavigne|

+-------+---------------+

+-------+------------+ //run 2 results

| id| name|

+-------+------------+

|1034635| [unknown]|

| 930| Eminem|

| 1205| U2|

| 250| Outkast|

| 121|Beastie Boys|

+-------+------------+

+-------+------------+ //run 3 results

| id| name|

+-------+------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 930| Eminem|

| 4267| Green Day|

|1058104|Gwen Stefani|

+-------+------------+

+-------+---------------+ //run 4 results

| id| name|

+-------+---------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 1784|Black Eyed Peas|

| 930| Eminem|

| 250| Outkast|

+-------+---------------+

+-------+---------+ //run 5 results

| id| name|

+-------+---------+

| 2814| 50 Cent|

|1034635|[unknown]|

| 930| Eminem|

| 250| Outkast|

| 4267|Green Day|

+-------+---------+

+-------+------------+ //run 6 results

| id| name|

+-------+------------+

|1034635| [unknown]|

| 930| Eminem|

| 4267| Green Day|

| 121|Beastie Boys|

|1001412| blink-182|

+-------+------------+

+-------+------------+ //run 7 results

| id| name|

+-------+------------+

|1034635| [unknown]|

| 1786| Bob Marley|

| 930| Eminem|

| 4267| Green Day|

| 121|Beastie Boys|

+-------+------------+

+-------+------------+ //run 8 results

| id| name|

+-------+------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 930| Eminem|

| 250| Outkast|

|1058104|Gwen Stefani|

+-------+------------+

+-------+------------+ //run 9 results

| id| name|

+-------+------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 930| Eminem|

| 250| Outkast|

| 121|Beastie Boys|

+-------+------------+

+-------+------------+ //run 10 results

| id| name|

+-------+------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 930| Eminem|

| 1854| Linkin Park|

|1058104|Gwen Stefani|

+-------+------------+

+-------+------------+ //run 11 results

| id| name|

+-------+------------+

| 2814| 50 Cent|

|1034635| [unknown]|

| 930| Eminem|

| 250| Outkast|

|1058104|Gwen Stefani|

+-------+------------+

\*/

val end = Calendar.getInstance()

val eh = end.get(Calendar.HOUR)

val em = end.get(Calendar.MINUTE)

val es = end.get(Calendar.SECOND)

printf("End Time: %02d:%02d:%02d\n", eh, em, es);

val dh = eh-sh

val dm = em-sm

val ds = es-ss

printf("Total Time: %02d:%02d:%02d\n", dh, dm, ds);

import scala.collection.immutable.ListMap

val hoo = foo.groupBy(identity).mapValues(\_.size) //convert the array to a map, containing each instance and occurances

/\*

(1205,1)

(930,11)

(2814,7)

(121,4)

(1001048,1)

(1058104,4)

(1034635,11)

(1784,2)

(1786,2)

(4267,4)

(250,6)

(1001412,1)

(1854,1)

\*/

Here I group foo by identity(which is a generic input) and map the values by size, which in turns creates a Map[Int,Int] with the first value representing the artist id and the second value representing the number of times suggested.

val goo = ListMap(hoo.toSeq.sortWith(\_.\_2 > \_.\_2):\_\*) //flip the list over (sort by value)

/\*

(930,11)

(1034635,11)

(2814,7)

(250,6)

(121,4)

(1058104,4)

(4267,4)

(1784,2)

(1786,2)

(1205,1)

(1001048,1)

(1001412,1)

(1854,1)

\*/

Now we convert hoo to a sequence so we may sort it descending based on the value (Map[Key, Value])

var topRecs = new ArrayBuffer[Int]()

goo.foreach(topRecs += \_.\_1) //add each key(artistID) to find artist names

/\*930

1034635

2814

250

121

1058104

4267

1784

1786

1205

1001048

1001412

1854

\*/

artistByID.filter($"id" isin (topRecs:\_\*)).show() //show names of artists from topRecs

/\*

+-------+---------------+

| id| name|

+-------+---------------+

| 2814| 50 Cent| //3

|1034635| [unknown]| //2

| 1784|Black Eyed Peas| //8

| 1786| Bob Marley| //9

| 930| Eminem| //1

|1001048| Avril Lavigne| //11

| 1205| U2| //10

| 250| Outkast| //4

| 1854| Linkin Park| //13

| 4267| Green Day| //7

|1058104| Gwen Stefani| //6

| 121| Beastie Boys| //5

|1001412| blink-182| //12

+-------+---------------+

\*/

Since we now know what artists are recommended the most, I separate the values of just the Artists so we can match the accompanying ID with the artist name. Eminem ended up being the most recommended, which makes sense given the time frame and general popularity. It is after suggestion 4 when it actually suggests not based on as much popularity, but actual relation to artist (since more popular artists will be recommended more).

While this method gave us significantly better results than before, the amount of time amassed to calculate this was way too long and impractical. By using the Java Util calendar method, we can calculate the time taken to run (18 minutes and 28 seconds)! This method is useful if needed to do this weekly, but as the textbook mentions, but lacks real time capabilities. To be able to do something of this sort would require using something on part with NoSQL