recommender

January 14, 2016

1 Music Recommender System using Apache Spark and Python

Estimated time: 8hrs

1.1 Description

For this project, you are to create a recommender system that will recommend new musical artists to a user based on their listening history. Suggesting different songs or musical artists to a user is important to many music streaming services, such as Pandora and Spotify. In addition, this type of recommender system could also be used as a means of suggesting TV shows or movies to a user (e.g., Netflix).

To create this system you will be using Spark and the collaborative filtering technique. The instructions for completing this project will be laid out entirely in this file. You will have to implement any missing code as well as answer any questions.

Submission Instructions: * Add all of your updates to this IPython file and do not clear any of the output you get from running your code. * Upload this file onto moodle.

1.2 Datasets

You will be using some publicly available song data from audioscrobbler, which can be found here. However, we modified the original data files so that the code will run in a reasonable time on a single machine. The reduced data files have been suffixed with `_small.txt` and contains only the information relevant to the top 50 most prolific users (highest artist play counts).

The original data file `user_artist_data.txt` contained about 141,000 unique users, and 1.6 million unique artists. About 24.2 million users' plays of artists are recorded, along with their count.

Note that when plays are scribbled, the client application submits the name of the artist being played. This name could be misspelled or nonstandard, and this may only be detected later. For example, "The Smiths", "Smiths, The", and "the smiths" may appear as distinct artist IDs in the data set, even though they clearly refer to the same artist. So, the data set includes `artist_alias.txt`, which maps artist IDs that are known misspellings or variants to the canonical ID of that artist.

The `artist_data.txt` file then provides a map from the canonical artist ID to the name of the artist.

1.3 Necessary Package Imports

```
In [1]: from pyspark.mllib.recommendation import *
        import random
        from operator import *
        import pyspark
        import operator
        #from pyspark import SparkContext
        #sc = SparkContext()
```

1.4 Loading data

Load the three datasets into RDDs and name them `artistData`, `artistAlias`, and `userArtistData`. View the README, or the files themselves, to see how this data is formated. Some of the files have tab delimeters while some have space delimiters. Make sure that your `userArtistData` RDD contains only the canonical artist IDs.

```
In [2]: def mapArtistData(line):
            s = line.split("\t")
            #artistid artist_name
            return (int(s[0]), s[1])

        def mapArtistAlias(line):
            s=line.split()
            #badid, goodid
            return (int(s[0]), int(s[1]))

        def mapUserArtistData(line):
            s=line.split()
            #given - userid artistid playcount
            return (int(s[0]), int(s[1]), int(s[2]))

        artistData = sc.textFile('artist_data_small.txt').map(mapArtistData)
        artistAlias = sc.textFile('artist_alias_small.txt').map(mapArtistAlias)
        userArtistData = sc.textFile('user_artist_data_small.txt').map(mapUserArtistData)

        # artistData.collect()
        # artistAlias.collect()
        # userArtistData.collect()
```

1.5 Data Exploration

In the blank below, write some code that with find the users' total play counts. Find the three users with the highest number of total play counts (sum of all counters) and print the user ID, the total play count, and the mean play count (average number of times a user played an artist). Your output should look as follows:

```
User 1059637 has a total play count of 674412 and a mean play count of 1878.
User 2064012 has a total play count of 548427 and a mean play count of 9455.
User 2069337 has a total play count of 393515 and a mean play count of 1519.
```

```
In [3]: explore = userArtistData.map(lambda x: (x[0], (x[1], x[2]))).groupByKey().mapValues(lambda d: di
        .map(lambda u: (sum(u[1].values()), u[0], len(u[1].values())))\
        .map(lambda t: (t[0], t[1], t[0]//t[2]))
        explore.collect()
        top = explore.takeOrdered(3, lambda x: -x[0])

        for entry in top:
            print("User " + str(entry[1]) + " has a total play count of " + str(entry[0]) + " and a mean
```

```
User 1059637 has a total play count of 674412 and a mean play count of 1878.
User 2064012 has a total play count of 548427 and a mean play count of 9455.
User 2069337 has a total play count of 393515 and a mean play count of 1519.
```

Splitting Data for Testing Use the randomSplit function to divide the data (`userArtistData`) into: * A training set, `trainData`, that will be used to train the model. This set should constitute 40% of the data.

* A validation set, `validationData`, used to perform parameter tuning. This set should constitute 40% of the data. * A test set, `testData`, used for a final evaluation of the model. This set should constitute 20% of the data.

Use a random seed value of 13. Since these datasets will be repeatedly used you will probably want to persist them in memory using the cache function.

In addition, print out the first 3 elements of each set as well as their sizes; if you created these sets correctly, your output should look as follows:

```
[(1059637, 1000049, 1), (1059637, 1000056, 1), (1059637, 1000113, 5)]
[(1059637, 1000010, 238), (1059637, 1000062, 11), (1059637, 1000112, 423)]
[(1059637, 1000094, 1), (1059637, 1000130, 19129), (1059637, 1000139, 4)]
19817
19633
10031
```

```
In [4]: datasize = userArtistData.count()
        trainData, validationData, testData = userArtistData.randomSplit([0.4*datasize, 0.4*datasize, 0

        trainData.cache()
        validationData.cache()
        testData.cache()
```

```
Out[4]: PythonRDD[15] at RDD at PythonRDD.scala:43
```

```
In [5]: print(trainData.take(3))
        print(validationData.take(3))
        print(testData.take(3))
        print(trainData.count())
        print(validationData.count())
        print(testData.count())

        artistIDMap={}

        for artist in artistAlias.collect():
            badid = artist[0]
            goodid = artist[1]
            artistIDMap[badid] = goodid

        def cano(uad):
            s=[]
            for u in uad.collect():
                userID = u[0]
                artistID = u[1]
                playcount = u[2]
                if artistID in artistIDMap.keys():
                    t = (userID, artistIDMap[artistID], playcount)
                else: t = (userID, artistID, playcount)
                s.append(t)
            return s

        canonical = sc.parallelize(cano(userArtistData))\
        .map(lambda z: ((z[0], z[1]), z[2])).reduceByKey(lambda a,b: a+b).map(lambda y: (y[0][0], y[0][

        # canonical.cache()
        # canonical.collect()
```

```
[(1059637, 1000049, 1), (1059637, 1000056, 1), (1059637, 1000113, 5)]
[(1059637, 1000010, 238), (1059637, 1000062, 11), (1059637, 1000112, 423)]
[(1059637, 1000094, 1), (1059637, 1000130, 19129), (1059637, 1000139, 4)]
19817
19633
10031
```

1.6 The Recommender Model

For this project, we will train the model with implicit feedback. You can read more information about this from the collaborative filtering page: http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html. The function you will be using has a few tunable parameters that will affect how the model is built. Therefore, to get the best model, we will do a small parameter sweep and choose the model that performs the best on the validation set

Therefore, we must first devise a way to evaluate models. Once we have a method for evaluation, we can run a parameter sweep, evaluate each combination of parameters on the validation data, and choose the optimal set of parameters. The parameters then can be used to make predictions on the test data.

1.6.1 Model Evaluation

Although there may be several ways to evaluate a model, we will use a simple method here. Suppose we have a model and some dataset of true artist plays for a set of users. This model can be used to predict the top X artist recommendations for a user and these recommendations can be compared the artists that the user actually listened to (here, X will be the number of artists in the dataset of true artist plays). Then, the fraction of overlap between the top X predictions of the model and the X artists that the user actually listened to can be calculated. This process can be repeated for all users and an average value returned.

For example, suppose a model predicted [1,2,4,8] as the top X=4 artists for a user. Suppose, that user actually listened to the artists [1,3,7,8]. Then, for this user, the model would have a score of 2/4=0.5. To get the overall score, this would be performed for all users, with the average returned.

NOTE: when using the model to predict the top-X artists for a user, do not include the artists listed with that user in the training data.

Name your function `modelEval` and have it take a model (the output of ALS.trainImplicit) and a dataset as input. For parameter tuning, the dataset parameter should be set to the validation data (`validationData`). After parameter tuning, the model can be evaluated on the test data (`testData`).

In [6]:
```
'''
X would be the number of artists the user listens to in the testData when testData is passed as
After prediction you select these top X predicted artists and find its intersection with the ar
If out of these suppose Y artists match the answer would be Y/X for user1.
You calculate this for all the other users and return the average value.


1) Find all artists from list of all artists in the UserArtistData
2) Subtract the artists the user listens to in the training set for each user
3) Do the prediction in this newly generated user, artists pairs
4) Find the number of artists that the user listens to in the validationData, this would be you
5) Find the score for each user and return the average back.


allArtists = all the artists in the entire dataset
Iterate through the users and for each user
nonTrainArtists = allArtists - the artists that were in the training set for the current user
trueArtists = the artists in the dataset passed to modeleval for the current user
X = len(trueArtists)
use the model to predict all the ratings on nonTrainArtists
```

```
predictResult = top X sorted by highest rating from the prediction
Compare predictResult to trueArtists.

# Correct logic; but doesn't work. Empty RDD returned by predictAll()
def modelEval(model, ds):
    totalArtists = userArtistData.map(lambda u: u[1]).distinct()
    totalArtistsTrain = trainData.map(lambda u: u[1]).distinct()
    predictOnArtists = totalArtists.subtract(totalArtistsTrain)
    usersInDS = ds.map(lambda u: (u[0], u[1])).groupByKey().map(lambda u:u[0])
    artistsInDS = ds.map(lambda x: (x[0], x[1])).groupByKey().map(lambda y: (y[0], list(y[1])))
    usersAndArtists = usersInDS.cartesian(predictOnArtists)

    predictedResults = model.predictAll(usersAndArtists).map(lambda r: (r[0], r[1], r[2]))\
    .sortBy(lambda s:(s[0], -s[2])).map(lambda x: (x[0], x[1]))\
    .groupByKey().map(lambda t: (t[0], list(t[1])))

    userXValue = ds.map(lambda m: (m[0], m[1])).groupByKey().map(lambda n: (n[0], len(n[1])))

    topArtists = userXValue.join(predictedResults).map(lambda a: (a[0], a[1][1][:a[1][0]]))

    intersection = topArtists.join(artistsInDS).map(lambda p: (p[0], set(p[1][0]).intersection(
    intersectionCount = intersection.map(lambda r: (r[0], len(r[1])))

    probability = intersectionCount.join(userXValue).map(lambda s: (s[0], s[1][0]/float(s[1][1]
    score = 0
    for p in probability.collect():
        score += p[1]
    score = score/float(probability.count())

    return score
'''

def modelEval(model, ds):
    totalArtists = userArtistData.map(lambda u: u[1]).distinct()
    usersInDS = ds.map(lambda u: (u[0], u[1])).groupByKey().map(lambda u:u[0])

    usersIntoTotalArtists = usersInDS.cartesian(totalArtists)
    usersArtistsTrain = trainData.map(lambda u: (u[0], u[1])).distinct()
    predictOn = usersIntoTotalArtists.subtract(usersArtistsTrain)

    predictedResults = model.predictAll(predictOn)\
    .sortBy(lambda r: (r.user, -r.rating)).map(lambda u: (u[0], u[1]))\
    .groupByKey().map(lambda v: (v[0], list(v[1])))

    artistsInDS = ds.map(lambda x: (x[0], x[1])).groupByKey().map(lambda y: (y[0], list(y[1])))
    userXValue = ds.map(lambda m: (m[0], m[1])).groupByKey().map(lambda n: (n[0], len(n[1])))
    topArtists = userXValue.join(predictedResults).map(lambda a: (a[0], a[1][1][:a[1][0]]))

    intersection = topArtists.join(artistsInDS).map(lambda p: (p[0], set(p[1][0]).intersection(
    intersectionCount = intersection.map(lambda r: (r[0], len(r[1])))

    probability = intersectionCount.join(userXValue).map(lambda s: (s[0], s[1][0]/float(s[1][1]

    score = 0
```

```
        for p in probability.collect():
            score += p[1]
        score = score/float(probability.count())

        return score
```

1.6.2 Model Construction

Now we can build the best model possibly using the validation set of data and the `modelEval` function. Although, there are a few parameters we could optimize, for the sake of time, we will just try a few different values for the rank parameter (leave everything else at its default value, **except make `seed=345`**). Loop through the values [2, 10, 20] and figure out which one produces the highest scored based on your model evaluation function.

Note: this procedure may take several minutes to run.

For each rank value, print out the output of the `modelEval` function for that model. Your output should look as follows:

```
The model score for rank 2 is 0.090431
The model score for rank 10 is 0.095294
The model score for rank 20 is 0.090248
```

```
In [7]: #The training is done on the trainData and the scoring is done on the validationData(=dataSet)

        ranks = [2, 10, 20]
        for r in ranks:
            model = ALS.trainImplicit(trainData, rank=int(r), seed=345)
            score = modelEval(model, validationData)
            print("The model score for rank " + str(r) + " is " + str(score))
```

```
The model score for rank 2 is 0.0904308588871
The model score for rank 10 is 0.0952938950541
The model score for rank 20 is 0.0902586630954
```

Now, using the bestModel, we will check the results over the test data. Your result should be ~`0.0507`.

```
In [8]: bestModel = ALS.trainImplicit(trainData, rank=10, seed=345)
        result = modelEval(bestModel, testData)
        print(result)
```

```
0.0507314301519
```

1.7 Trying Some Artist Recommendations

Using the best model above, predict the top 5 artists for user `1059637` using the recommendProducts function. Map the results (integer IDs) into the real artist name using `artistAlias`. Print the results. The output should look as follows:

```
Artist 0: Brand New
Artist 1: Taking Back Sunday
Artist 2: Evanescence
Artist 3: Elliott Smith
Artist 4: blink-182
```

```
In [9]: artistTop5 = bestModel.recommendProducts(1059637,5)
        #artistTop5 = sc.parallelize(artistTop5).map(lambda r: r.product)
        #artistTop5IDs = artistTop5.collect()
```

```
        count = 0

        # For Canonical Artist IDs
        artistTop5IDs = sc.parallelize(cano(sc.parallelize(artistTop5).map(lambda r: (r[0], r[1], r[2]))
        .map(lambda z: ((z[0], z[1]), z[2])).reduceByKey(lambda a,b: a+b).map(lambda y: (y[0][0], y[0][
        .map(lambda r: r[1]).collect()


        for artistID in artistData.collect():
            if artistID[0] in artistTop5IDs:
                print("Artist %d: " %count + artistID[1])
                count+=1
```

```
Artist 0: Taking Back Sunday
Artist 1: Brand New
Artist 2: Elliott Smith
Artist 3: Evanescence
Artist 4: blink-182
```

```
In [ ]:
```