Yahoo Music Recommendation

EE627 Final Project
Group Name:你吃披萨吗
Group Member: Jiahao Lu
Wei Hong
Kai Li
Instuctor: Prof. Wang
Date: 05/07/2020

I. Summary of the methods and Performance

1. Organize Data

(1) Data pre-processing

In order to process the data, we have to read all the data and put them together in one file or some data structure. At first, I planned to put them in some dictionary. Dictionary is in a good performance for us to edit and view our data in a short time. For this project, we build three dictionaries.

The first one is training data dictionary from trainitem2.txt. In this dictionary, keys are userID and values are lists. In every list, the odd index of value is itemID and the even index of data is score of the previous item. Like (userID, [itemID1, score1, itemID2, score2, itemID3, score3….]).

```
file_name_train='trainItem2.txt'
fDict_train= open(file_name_train, 'r')
#output_file = 'output1.txt'
#fOut = open(output_file, 'w')

dict_train = {}
i = 0
list_train = []


for line in fDict_train:
    if '|' in line:
        if i != 0:
            dict_train[userID] = list_train
            #print(dict_train)
            #if i == 4:
                #break
        arr_userID=line.strip().split('|')
        arr_userID = [int(x) for x in arr_userID]
        userID = arr_userID[0]
        #print(userID)
        i += 1
        list_train = []
    else:
        arr_trackID=line.strip().split('\t')
        #print(arr_trackID)
        arr_trackID = [int(x) for x in arr_trackID]
        itemID = arr_trackID[0]
        itemPoint = arr_trackID[1]
        #print(itemID)
        #print(itemPoint)

        list_train.append(itemID)
        list_train.append(itemPoint)
        #print(list_train)

dict_train[userID] = list_train
```

The second one is dictionary for track data from trackData2.txt. This dictionary contains all the relationship of all tracks and their album, artist, genres. Keys in this dictionary is trackID and values are albumID, artistID and genres.

```
file_name_track='trackData2.txt'
fDict_track= open(file_name_track, 'r')

dict_track = {}
a = 0
list_track = []


for line in fDict_track:
    arr_all_line=line.strip().split('|')
    trackID = arr_all_line[0]
    trackID = int(trackID)
    #print(trackID)
    del arr_all_line[0]
    arr_all_line_change = ['0' if i == 'None' else i for i in arr_all_line]
    arr_all_line_change = [int(x) for x in arr_all_line_change]
    #print(arr_all_line)
    dict_track[trackID] = arr_all_line_change
    #print(dict_track)
    a += 1
    #if a == 4:
        #break

#print(dict_track)
```

The third one is a big list from testItem2.txt. In this list, the odd indexes are userID and the even indexes are small lists. In these small lists, odd indexes are trackID and even indexes are scores for the previous track. Like [userID1, [trackID1, score1-1, trackID1-2, score1-2…. trackID1-6, score1-6], userID2, [trackID2-1, score2-1, trackID2-2, score2-2…. trackID2-6, score2-6] ….].

```
file_name_test='testItem2.txt'
fList_test= open(file_name_test, 'r')

j = 0
b = 0
list_test = []
list_test_track = []


for line in fList_test:
    if '|' in line:
        if j != 0:
            list_test.append(list_test_track)
            list_test_track = []
            #print(list_test)
        #if b == 4:
            #break
        arr_test_userID=line.strip().split('|')
        arr_test_userID = [int(x) for x in arr_test_userID]
        test_userID = arr_test_userID[0]
        list_test.append(test_userID)
        b += 1
        j += 1
        # #print(userID)
        # i += 1
        # list_train = []
    else:
        arr_test_track_ID=line.strip().split('\t')
        arr_test_track_ID = [int(x) for x in arr_test_track_ID]
        #print(arr_trackID)
        track_test_ID = arr_test_track_ID[0]
        #print(track_test_ID)
        list_test_track.append(track_test_ID)
        list_test_track.append(0)
        #print(list_test_track)

        #list_train.append(itemID)
        #list_train.append(itemPoint)
#list_test.append(list_test_track)
#print(list_test)

output_file= 'output4.txt'                  # 输出文件
fOut = open(output_file, 'w')           # 输出文件写入状态
```

Now, all the data structures have been built. We can easily get every data we want from these three dictionaries and lists but not from those files every time we run our code. Also,

it's a good way to write all these data in our data structure into one file and only read this file at the begin of our code.

(2) Method: Simply Change Weight
The simplest way to score all the track is just change the weight of all the feature we already have. We tried a lot of combination of weight. In the beginning, we limited the sum of all the proportions to 1. The best result we get in Kaggle is 0.86679. The combination is
Predict = Track + Album * 0.6 + Artist * 0.35 + Genre * 0.05.

By accident we find that we may get better score when the sum of these weight is not 1. Then we tried some new combination then we get 0.86841 in Kaggle with the following combination.
Predict = Track + Album 0.912 + Artist * 0.591 + Genre * 0.05.

2. Logistic Regression

Beside album, artist and genre score, we get some other numeric feature like the max, min and mean of genre. In order to build our logistic regression model by spark pipeline, we have to transform all our data type from string to integer. Then we just call the function in pipeline to build our logic regression model. Using this model to get a new feature column with those numeric features. After that we can preview some output of feature.

In [9]:
```
import pandas as pd
pd.DataFrame(df1.take(5), columns=df1.columns).transpose()
numeric_features = ['Userid','trackid','label','albumscore','artistscore','genreamax','genreamin','genreamean']

df1.select(numeric_features).describe().toPandas().transpose()
```

Out[9]:

	0	1	2	3	4
summary	count	mean	stddev	min	max
Userid	120000	224372.7714	14155.612277030812	199810	249010
trackid	120000	147871.92235833334	85421.1993529344	1	296099
label	120000	0.0	0.0	0	0
albumscore	120000	23.685075	38.767846279190096	0	100
artistscore	120000	35.016175	42.73300263592536	0	100
genreamax	120000	35.471716666666666	40.80419698479321	0	100
genreamin	120000	32.74715833333333	39.44356509895082	0	100
genreamean	120000	34.149823609307354	39.70187105524825	0.0	100.0

Then get the probability as the basis for scoring. The last step is scoring every element with the probability we get. After uploading our output of logic regression, we get 0.83256 in Kaggle.

```
In [11]: from pyspark.ml import Pipeline
         pipeline=Pipeline(stages=[assembler1])
         model1=pipeline.fit(df1)
         df1=model1.transform(df1)

In [12]: predictions1 = lrModel.transform(df1)
         predictions1.show(10)
```

```
+------+-----+-------+----------+-----------+---------+---------+----------+--------------------+--------------------
+--------------------+----------+
|Userid|label|trackid|albumscore|artistscore|genreamax|genreamin|genreamean|            features|       rawPrediction
|         probability|prediction|
+------+-----+-------+----------+-----------+---------+---------+----------+--------------------+--------------------
+--------------------+----------+
|199810|    0| 208019|         0|          0|        0|        0|         0|(7,[0,1],[199810....|[6.63749142170894...
|[0.99869140452593...|       0.0|
|199810|    0|  74139|         0|          0|       80|       80|        80|[199810.0,74139.0...|[3.92866452036579...
|[0.98070951849845...|       0.0|
|199810|    0|   9903|         0|          0|        0|        0|         0|(7,[0,1],[199810....|[6.31163706225367...
|[0.99818822903157...|       0.0|
|199810|    0| 242681|         0|          0|        0|        0|         0|(7,[0,1],[199810....|[6.69450228306005...
|[0.99876383225790...|       0.0|
|199810|    0|  18515|         0|         70|        0|        0|         0|(7,[0,1,3],[19981...|[3.47091791060977...
|[0.96984887167018...|       0.0|
|199810|    0| 105760|         0|         90|       80|       80|        80|[199810.0,105760....|[0.31010867154557...
|[0.57691178640196...|       0.0|
|199812|    0| 276940|         0|          0|        0|        0|         0|(7,[0,1],[199812....|[6.75091328850732...
|[0.99883155610018...|       0.0|
|199812|    0| 142408|       100|        100|       80|       80|        80|[199812.0,142408....|[-0.0373915314027...
|[0.49065320612404...|       1.0|
```

3. Decision Tree

(1) Principle of Decision Tree:

Decision tree model has influenced a wide area of machine learning, covering both classification and regression. As the name goes, it uses a tree-like model of decisions. Though it is a commonly used tool in data mining for deriving a strategy to reach a particular goal, it is also widely used in machine learning.

(2) Formula:

1. Entropy: a measure of the degree of "chaos".

The more ordered the system, the lower the entropy. The more chaotic or dispersed the system, the higher the entropy.

If the classification of event A is (A1, A2, ..., An), the probability of each part occurring is (p1, p2, ..., pn), then the information entropy is defined as the formula is as follows:

$$Ent(A)=-\sum_{k=1}^{n} p_k log_2 p_k$$

2. Information gain: The difference in entropy before and after the data set is divided by a certain feature.

The information gain formula is as follows:

D: for the sample set

Ent (D): overall entropy

a: Discrete attributes

v: is a possible value node in the a attribute

D ^ v: The vth branch node contains all the samples in D that take the value a ^ v on the attribute a

$$Gain(D,a)=Ent(D)-\sum_{v=1}^{v} \frac{D_v}{D} Ent(D_v)$$

3. Gini value Gini (D): Two samples are randomly selected from the data set D, and the probability that the category labels are inconsistent, so Gini (D) The smaller the value, the higher the purity of data set D. Gini formula is as follows:

$$Gini(D)=1-k=1\sum|y|pk2$$

Gini gain formula is as follows:

$$Gini(D,a)=Gini(D)-v=1\sum v|D||Dv|Gini(Dv)$$

(3) Discussions:

To generate a decision tree, we need to choose the largest information gain, the largest the information gain ratio, and the smallest Gini index as the optimal feature. Starting from the root node, a decision tree is generated recursively. It is equivalent to using information gain or other criteria to continuously select locally optimal features, or to split the training set into a subset that can be basically correctly divided.

(4) Code:

```
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.types as types
df=df.withColumn('label',df['label'].cast(types.IntegerType()))
df=df.withColumn('Userid',df['Userid'].cast(types.IntegerType()))
df=df.withColumn('trackid',df['trackid'].cast(types.IntegerType()))
df=df.withColumn('albumscore',df['albumscore'].cast(types.IntegerType()))
df=df.withColumn('artistscore',df['artistscore'].cast(types.IntegerType()))
df=df.withColumn('genreamax',df['genreamax'].cast(types.IntegerType()))
df=df.withColumn('genreamin',df['genreamin'].cast(types.IntegerType()))
df=df.withColumn('genreamean',df['genreamean'].cast(types.IntegerType()))

assembler=VectorAssembler(inputCols=['Userid','trackid','label','artistscore','genreamax','genreamin','genreamean'],
```

```
from pyspark.ml import Pipeline
pipeline=Pipeline(stages=[assembler])
model=pipeline.fit(df)
df=model.transform(df)
```

```
train,test=df.randomSplit([0.95,0.05])
```

```
handleInvalid = "keep"
print("Training Dataset Count: " + str(train.count()))
print("Test Dataset Count: " + str(test.count()))
```

```
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol = 'features', labelCol = 'label',maxDepth = 3,)
dtModel = dt.fit(train)
```

For this code, we get the decision tree model by calling the function in DecisionTreeClassifier. The output of this code results in a final total correct rate of 0.83597. From the score on Kaggle, we learn that this score is good but not very high. Therefore, we think that decision tree model is a good but not perfect model. It needs to be improved.

4. Gradient-Boosted Tree

(1) Principles of Gradient-Boosted Tree:
Gradient-Boosted Tree model is a model that using the steepest descent approximation method, that is, the value of the negative gradient of the loss function in the current model is used as the approximate value of the residual of the lifting tree algorithm in the regression problem, and a regression tree is fitted.

(2) Formula:

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

(3) Discussions:
Gradient Boosted Tree algorithm is a general learning algorithm. In addition to Decision Tree model, other models can also be used as base learners. The idea of the Gradient Boosted Tree algorithm is to adjust the model so that the value of the loss function continuously decreases, and then add up the various models as the final prediction model. The Gradient Boosted Decision Tree is based on the Decision Tree model as the learner.

(4) Code:

```
from pyspark.ml import Pipeline
pipeline=Pipeline(stages=[assembler])
model=pipeline.fit(df)
df=model.transform(df)
```

```
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(featuresCol = 'features',maxIter=10)
gbtModel = gbt.fit(df)
```

For this code, we get the decision tree model by calling the function in GBT Classifier. The output of this code results in a final total correct rate of 0.80075. The results of this model is worse than other models, so it should be also improved or even using other models.

5. Random Forest Classifier

(1) Principles of Random Forest

Random forest is a classification algorithm proposed by Leo Breiman, which uses a self-serve resampling technique to generate new training decision trees from the original training sample set N with n randomly selected samples put back into the ground repeatedly, and then m decision trees are generated to form the random forest according to the above steps, and the classification results of the new data are determined by the fraction formed by how many votes of the classification trees. It is a refinement of the decision tree algorithm, combining multiple decision trees together, with each tree relying on an independently drawn sample for its establishment. The classification power of a single tree may be small, but after randomly generating many decision trees, a test sample can statistically select the most likely classification from the classification results of each tree.

(2) Process of Random Forest

First, n samples were selected from the sample set with put-back random sampling. K features are then randomly selected from all features, and a decision tree is built using these features for the selected sample. Repeat the above two steps m times, i.e. generate m decision trees to form a random forest. Then for new data, a decision is made on each tree, and a final vote is taken to confirm which category is assigned.

(3) Formula of Random Forest

Classification accuracy, which is the probability that the positive and negative samples will be correctly classified, respectively.

$$Accuracy = \frac{TP + TN}{P + N}$$

Accuracy, which is the degree of truthfulness in the case where the classification results in a positive sample.

$$Precision = \frac{TP}{TP + FP}$$

Where P (Positive Sample): Number of positive samples. N (Negative Sample): Number of negative samples. TP (True Positive): Number of positive samples correctly predicted. FP (False Positive): Number of negative samples predicted to be positive. TN (True Negative): Number of negative samples correctly predicted.

(4) Code:

```
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.types as types
df1=df1.withColumn('label',df1['label'].cast(types.IntegerType()))
df1=df1.withColumn('Userid',df1['Userid'].cast(types.IntegerType()))
df1=df1.withColumn('trackid',df1['trackid'].cast(types.IntegerType()))
df1=df1.withColumn('albumscore',df1['albumscore'].cast(types.IntegerType()))
df1=df1.withColumn('artistscore',df1['artistscore'].cast(types.IntegerType()))
df1=df1.withColumn('genreamax',df1['genreamax'].cast(types.IntegerType()))
df1=df1.withColumn('genreamin',df1['genreamin'].cast(types.IntegerType()))
df1=df1.withColumn('genreamean',df1['genreamean'].cast(types.IntegerType()))

assembler1=VectorAssembler(inputCols=['Userid','trackid','label','artistscore','genreamax','genreamin','genreamean'],outputCol='features')
```

```
from pyspark.ml import Pipeline
pipeline=Pipeline(stages=[assembler1])
model1=pipeline.fit(df1)
df1=model1.transform(df1)
```

```
predictions1 = rfModel.transform(df1)
predictions1.show(10)
```

For this code we perform random forest calculations based on the original decision tree structure. The nature of the random forest is also the multiple decision tree. The output of this code results in a final total correct rate of 0.84449.

For this we can add more parameters to the raw data. to improve the overall structure. The code is as follows.

```
from pyspark.ml.feature import VectorAssembler
import pyspark.sql.types as types
#df=df.withColumn('label',df['label'].cast(types.IntegerType()))
df=df.withColumn('Userid',df['Userid'].cast(types.IntegerType()))
df=df.withColumn('trackid',df['trackid'].cast(types.IntegerType()))
df=df.withColumn('albumscore',df['albumscore'].cast(types.IntegerType()))
df=df.withColumn('trackMax',df['trackMax'].cast(types.IntegerType()))
df=df.withColumn('trackMin',df['trackMin'].cast(types.IntegerType()))
df=df.withColumn('trackMean',df['trackMean'].cast(types.IntegerType()))
df=df.withColumn('artistscore',df['artistscore'].cast(types.IntegerType()))
df=df.withColumn('albumMax',df['albumMax'].cast(types.IntegerType()))
df=df.withColumn('albumMin',df['albumMin'].cast(types.IntegerType()))
df=df.withColumn('albumMean',df['albumMean'].cast(types.IntegerType()))
df=df.withColumn('genreamax',df['genreamax'].cast(types.IntegerType()))
df=df.withColumn('genreamin',df['genreamin'].cast(types.IntegerType()))
df=df.withColumn('genreamean',df['genreamean'].cast(types.IntegerType()))

assembler1=VectorAssembler(inputCols=['Userid','trackid','albumscore','trackMax','trackMin','trackMean','artistscore','albumMax','albumMin
```

```
from pyspark.ml import Pipeline
pipeline=Pipeline(stages=[assembler1])
model1=pipeline.fit(df1)
df1=model1.transform(df1)
```

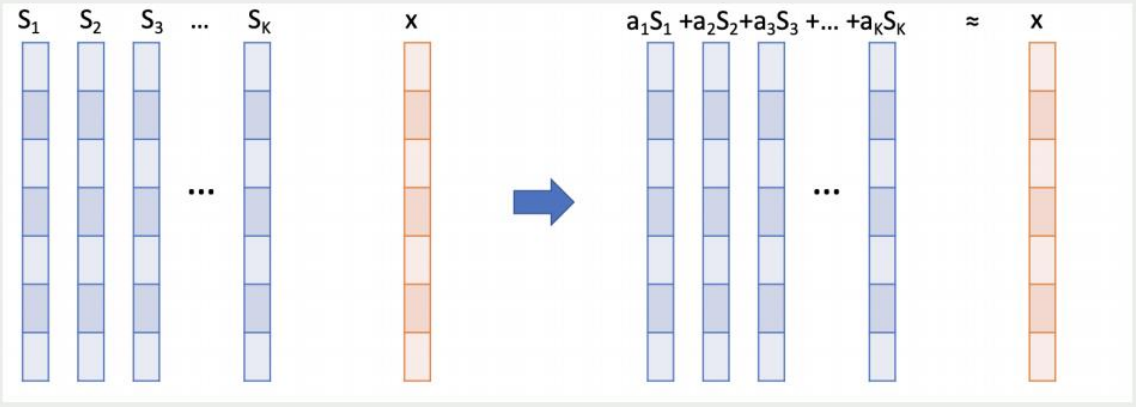
The results of the new document structure have also improved. The overall correct rate was achieved 0.86699. It shows that such an improvement is feasible.

6. Ensembling

(1)Principles of Random Ensembling

We can build an optimized linear combination of all the predictions to approximate the ground truth solution. That is, given submitted solution vectors **s**1*;* **s**2*; · · · ;* **s***K*, we look for a set of weights to combine a new vector which is close to the **x** as much as we can.

$$a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K \approx \mathbf{x}$$



This ensembling question can be converted to a typical optimization problem, i.e., a minimum distance linear fitting problem.

$$a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K \approx \mathbf{x}$$

$$\Rightarrow \qquad \arg\min_{a_1,a_2,\cdots,a_K} \|\mathbf{x} - (a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K)\|^2$$

The minimum distance optimization problem has a classic solution: Least-Squares(LS) solution.

$$\arg\min_{\mathbf{a}} \|\mathbf{x} - \mathbf{S}\mathbf{a}\|^2 \qquad \Rightarrow \qquad \mathbf{a}_{\mathrm{LS}} = \left(\mathbf{S}^T\mathbf{S}\right)^{-1}\mathbf{S}^T\mathbf{x}$$

The way to find a_{LS} is using the set {1,-1}, the multiplication products “1" indicate the cases where the prediction and the truth matching, while the multiplication products “-1" indicate the cases where the prediction and the truth not matching.

So the S^TX= number of succeed predictions × 1 + number of failed predictions × (-1). We can calculate that $S^TX = N(2Pi - 1)$, where *Pi* is the correct rate.

$$\mathbf{S}^T\mathbf{x} = \begin{bmatrix} \mathbf{s}_1^T\mathbf{x} \\ \mathbf{s}_2^T\mathbf{x} \\ \vdots \\ \mathbf{s}_K^T\mathbf{x} \end{bmatrix} = \begin{bmatrix} N(2P_1 - 1) \\ N(2P_2 - 1) \\ \vdots \\ N(2P_K - 1) \end{bmatrix}$$

We have known how to calculate S^TX , so we can use the formula $S_{ensemble}$ to find the prediction for the result.

$$\mathbf{s}_{\text{ensemble}} = a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K = \mathbf{S} \cdot \mathbf{a}_{\text{LS}} = \mathbf{S}\left(\mathbf{S}^T\mathbf{S}\right)^{-1}\mathbf{S}^T\mathbf{x}$$

(2)Code:

```
M = xlsread('Newinput.xls');
N = 120000;
P1 = 0.80075;
P2 = 0.83256;
P3 = 0.85442;
P4 = 0.74292;
STX = [N*(2*P1-1);N*(2*P2-1);N*(2*P3-1);N*(2*P4-1)];
INVTM = inv(M'*M);
M*INVTM*STX
```

The input for this code is a file that has already integrated the results four times. The new output can be calculated by following the above formula after the file is used as input. The final output can be obtained by arranging the new output by size. The final correct rate is 0.86841. The overall improvement in correctness isn't much.

II. conclusion

In this final project we try different algorithms including change the weight of the items ratings we get to obtain a highest accuracy of recommendation. Combining the above-mentioned methods and their corresponding results, the highest rate of correctness is 0.86841. The Weighting method and Ensembling both can achieve this rate.

We used different methods to get this result. Through this project, we found that different methods mean different thoughts. We cannot easily judge which one is better because the scores are very close. However, we did our best to get the results we wanted. After repeated attempts, we found that the weighting method can show a higher accuracy rate than RF, GBT and other methods.

We learned a lot from this project, like different methods about data mining, teamwork, real market analysis skill, etc. We realized that what really matters is the combination of multiple ways to solve one problem. Each one would have his own approach. Different approach means different advantages. The comprehensive result should conclude from all the possible approach, and it would be the best of best.