DATA420 Assignment 2

Jing Wu
29696576

Data Processing

Q1

I used –ls -R to get the subdirectory of the msd directory and used –du to check the file size. The data is structured according to the directory tree below:

```
                                                                        Size
/data/msd/                                                              12.9G
├─ audio                                                                12.3G
│   ├─ attributes                                                       103.0K
│   │   ├─ msd-jmir-area-of-moments-all-v1.0.attributes.csv             1.0K
│   │   ├─ msd-jmir-lpc-all-v1.0.attributes.csv                        671B
│   │   ├─ msd-jmir-methods-of-moments-all-v1.0.attributes.csv         484B
│   │   ├─ msd-jmir-mfcc-all-v1.0.attributes.csv                       898B
│   │   ├─ msd-jmir-spectral-all-all-v1.0.attributes.csv               777B
│   │   ├─ msd-jmir-spectral-derivatives-all-all-v1.0.attributes.csv   777B
│   │   ├─ msd-marsyas-timbral-v1.0.attributes.csv                     12.0K
│   │   ├─ msd-mvd-v1.0.attributes.csv                                 9.8K
│   │   ├─ msd-rh-v1.0.attributes.csv                                  1.4K
│   │   ├─ msd-rp-v1.0.attributes.csv                                  34.1K
│   │   ├─ msd-ssd-v1.0.attributes.csv                                 3.8K
│   │   ├─ msd-trh-v1.0.attributes.csv                                 9.8K
│   │   └─ msd-tssd-v1.0.attributes.csv                                27.6K
│   ├─ features                                                         12.2G
│   │   ├─ msd-jmir-area-of-moments-all-v1.0.csv                       65.5M
│   │   ├─ msd-jmir-lpc-all-v1.0.csv                                   53.1M
│   │   ├─ msd-jmir-methods-of-moments-all-v1.0.csv                    35.8M
│   │   ├─ msd-jmir-mfcc-all-v1.0.csv                                  70.8M
│   │   ├─ msd-jmir-spectral-all-all-v1.0.csv                          51.1M
│   │   ├─ msd-jmir-spectral-derivatives-all-all-v1.0.csv              51.1M
│   │   ├─ msd-marsyas-timbral-v1.0. csv                               412.2M
│   │   ├─ msd-mvd-v1.0.csv                                            1.3G
│   │   ├─ msd-rh-v1.0.csv                                             240.3M
│   │   ├─ msd-rp-v1.0.csv                                             4.0G
│   │   ├─ msd-ssd-v1.0.csv                                            640.6M
│   │   ├─ msd-trh-v1.0.csv                                            1.4G
│   │   └─ msd-tssd-v1.0.csv                                           3.9G
│   └─ statistics                                                       40.3M
│       └─ sample_properties.csv.gz                                    40.3M
├─ genre                                                                30.1M
│   ├─ msd-MAGD-genreAssignment.tsv                                    11.1M
│   ├─ msd-MASD-styleAssignment.tsv                                    8.4M
│   └─ msd-topMAGD-genreAssignment.tsv                                 10.6M
├─ main                                                                 174.4M
│   └─ summary                                                          174.4M
│       ├─ analysis.csv.gz                                             55.9M
│       └─ metadata.csv.gz                                             118.5M
└─ tasteprofile                                                         490.4M
    ├─ mismatches                                                       2.0M
    │   ├─ sid_matches_manually_accepted.txt                           89.2K
    │   └─ sid_mismatches.txt                                          1.9M
    └─ triplets.tsv                                                     448.4M
```

The Million Song Dataset (MSD) was 12.9G. There were four parts of this data, which were audio, genre, main and tasteprofile.

The audio directory in this dataset was the largest part which is 12.3G. There were 13 csv files in the attributes directory of audio directory, each of which defined the schema of the corresponding csv file in features directory of audio directory.

I used for loop in Unix shell to display the first three lines to check the structure and data type of each file in the attributes directory and features directory.

Each csv file in the attributes directory had the same structure. They all consisted of two columns, the first one was the column name and the second was the data type of this column.

I can get the data type of each csv file in the features directory from the corresponding csv file in the attributes directory. I used for loop to group the data type of each csv file in the attributes directory to get the data type count of the corresponding csv file in the features directory. The example results are provided below:

```
/data/msd/audio/attributes/msd-jmir-area-of-moments-all-v1.0.attributes.csv
     20 real
      1 string
/data/msd/audio/attributes/msd-jmir-lpc-all-v1.0.attributes.csv
     20 real
      1 string
……
/data/msd/audio/attributes/msd-tssd-v1.0.attributes.csv
   1176 NUMERIC
      1 STRING
```

There were three tsv files in the genre directory. Each tsv file contained two columns, the first column was the track ID, and the second was the corresponding genre label.

Since the files in main directory were csv.gz, we can decompress with the gunzip function, then use head -n3 to view the first three lines.

Each csv file in the features directory was divided into 8 partitions and stored in a gzip compressed csv format, and triplets.tsv was divided into 8 partitions and stored in a gzip compressed tsv format. So, the expected level of parallelism should be 8.

The repartition method can be used to either increase or decrease the number of partitions and the repartition method makes new partitions and evenly distributes the data in the new partitions. If there is a cluster with 8 cores, we don’t need this method since 8 patitions can be processed parallel. If there is a cluster with 12 cores, the repartition method can be useful to increase the level of parallelism.

In order to count the rows in the each of the dataset, I got the directory of each csv file by using awk method in Unix first, and then used for loop to count the number of rows in each csv file in the features directory.

The result is provided below:

```
/data/msd/audio/features/msd-jmir-area-of-moments-all-v1.0.csv
994623
/data/msd/audio/features/msd-jmir-lpc-all-v1.0.csv
994623
/data/msd/audio/features/msd-jmir-methods-of-moments-all-v1.0.csv
994623
/data/msd/audio/features/msd-jmir-mfcc-all-v1.0.csv
994623
/data/msd/audio/features/msd-jmir-spectral-all-all-v1.0.csv
994623
/data/msd/audio/features/msd-jmir-spectral-derivatives-all-all-v1.0.csv
994623
/data/msd/audio/features/msd-marsyas-timbral-v1.0.csv
995001
/data/msd/audio/features/msd-mvd-v1.0.csv
994188
/data/msd/audio/features/msd-rh-v1.0.csv
994188
/data/msd/audio/features/msd-rp-v1.0.csv
994188
/data/msd/audio/features/msd-ssd-v1.0.csv
994188
/data/msd/audio/features/msd-trh-v1.0.csv
994188
/data/msd/audio/features/msd-tssd-v1.0.csv
994188
```

As indicated in the assignment of assignment2, there were a total of 994, 960 unique songs in the sample. Each feature dataset contained nearly the same number of rows.

Data Processing
Q2

There were two txt files in the mismatches directory, one of which was all the mismatches without check, the other one was the manually checked matches. Since they had the same structure to store data, I defined a shared schema to load these two files and loaded the mismatches and matches data separately. Each useful line in sid_matches_manually_accepted.txt started with "< ERROR: ", while the lines in sid_mismatches.txt started with "ERROR: ". The schema of the mismatches is provided below:

```
root
 |-- SONG_ID: string (nullable = true)
 |-- TRACK_ID: string (nullable = true)
 |-- SONG_ARTIST: string (nullable = true)
 |-- SONG_TITLE: string (nullable = true)
 |-- TRACK_ARTIST: string (nullable = true)
 |-- SONG_TITLE: string (nullable = true)
```

I defined the schema of triplets and loaded the triplets data. Since the format of triplet was tsv file, when I loaded this file I set delimiter to tab ("\t"). The number of triplets was 48373586. The examples of triplets table were provided below:

```
+----------------------------------------+------------------+-----+
|USER_ID                                 |SONG_ID           |COUNT|
+----------------------------------------+------------------+-----+
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQEFDN12AB017C52B|1    |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQOIUJ12A6701DAA7|2    |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQOKKD12A6701F92E|4    |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOSDVHO12AB01882C7|1    |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOSKICX12A6701F932|1    |
+----------------------------------------+------------------+-----+
```

In order to get all the matches in the Taste Profile dataset, I filtered the mismatches to remove the manually checked matches first by using left-anti join method to join mismatches with manually checked matches. And then I using the left-anti join method to join triplets and the filtered mismatches. The number of matched triplets was 45795111.

In the Question 1, we have already viewed the data type of each csv file in the feature directory through the corresponding csv file in the attributes directory. It can be seen that the attributes types were defined as four types in total, "string", "STRING", "NUMERIC" and "real".

I created a function called create_schema(filename), the input is the filename in the attributes directory. We can use this function with the filename in the attributes directory to create the corresponding schema, which was named as filename + "_schema" by using the locals() method.

- The first step of this function was to load the data from the csv file in the attributes directory by using the input filename.

- And then to create a new column which was the corresponding data type in pyspark format (“string” and “STRING” ------“ StringType()”, “NUMERIC” and “real” ------ “DoubleType()”) based on the original data type.
- The last step is to create the StructureField list by using the for loop to append the tuple of (attribute name, eval(data type)) based on the loaded attributes file and create the schema for the loaded attributes file. We need to use eval() to evaluate the data type (for example, “ StringType()”) as a Python expression.

Audio Similarity
Q1

The datasets I picked was msd-jmir-mfcc-all-v1.0.csv, which is 70.8M. There are total 27 columns in this dataset, the only one string type column was the track ID and the remaining columns were numeric.

I loaded this dataset by using the schema defined in Data Processing Q2.

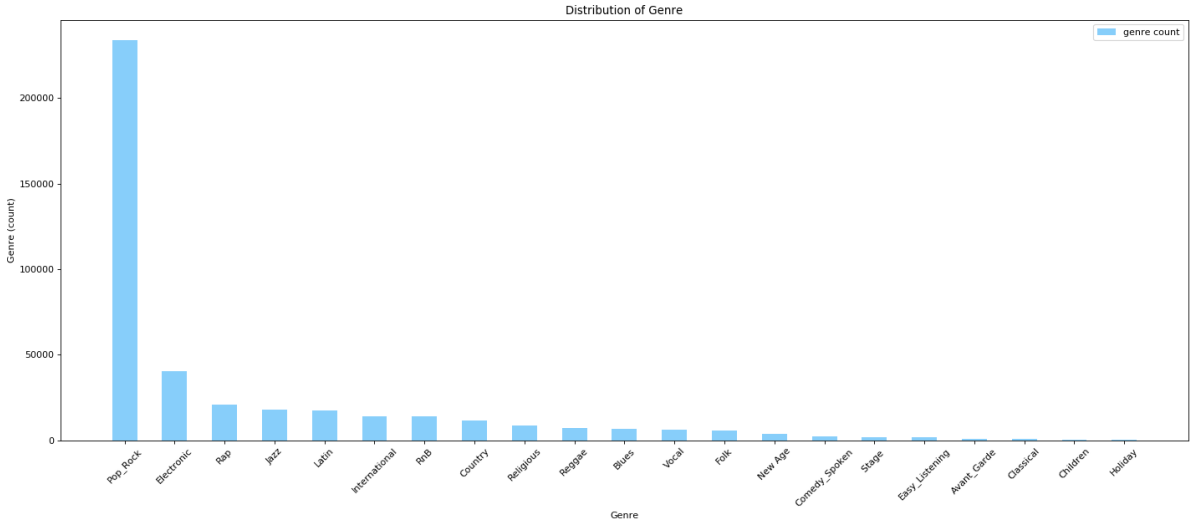
I used describe() method to produce the descriptive statistics. Since there were 26 numeric columns so there were 26 columns in the result rdd which was too wide, I used toPandas() to convert the rdd to pandas DataFrame and used pandas transpose() methods to transpose it to make it easy to read. The descriptive statistics result is provided below:

summary	count	mean	stddev	min	max
MFCC_Overall_Standard_Deviation_1	994623	46.242498594945175	23.145602482988075	0.0	544.0
MFCC_Overall_Standard_Deviation_2	994623	5.611131172112441	1.889936426496876	0.0	51.19
MFCC_Overall_Standard_Deviation_3	994623	4.304985814350773	1.2052843850324944	0.0	24.48
MFCC_Overall_Standard_Deviation_4	994623	3.226507721414047	0.7316868534116313	0.0	20.28
MFCC_Overall_Standard_Deviation_5	994623	2.685735150266994	0.5861501761953776	0.0	21.02
MFCC_Overall_Standard_Deviation_6	994623	2.4386808561489195	0.50160715471356	0.0	33.46
MFCC_Overall_Standard_Deviation_7	994623	2.2412334862214225	0.42181309487313284	0.0	19.15
MFCC_Overall_Standard_Deviation_8	994623	1.9406361865024098	0.3481757007129871	0.0	23.39
MFCC_Overall_Standard_Deviation_9	994623	1.8686131949018876	0.3319301127778366	0.0	23.08
MFCC_Overall_Standard_Deviation_10	994623	1.825155394964726	0.3161322442104294	0.0	26.12
MFCC_Overall_Standard_Deviation_11	994623	1.6955911254565832	0.31978956305968814	0.0	37.27
MFCC_Overall_Standard_Deviation_12	994623	1.5781773833070414	0.2807041487616942	0.0	25.74
MFCC_Overall_Standard_Deviation_13	994623	1.439864644249128	0.25329645461762473	0.0	19.38
MFCC_Overall_Average_1	994623	-115.59257219066833	43.36804507302751	-1150.0	-50.81
MFCC_Overall_Average_2	994623	11.392127466813102	6.760063128281962	-115.7	62.3
MFCC_Overall_Average_3	994623	-0.5534998088467622	4.036609641037462	-37.68	58.51
MFCC_Overall_Average_4	994623	4.066574771745199	2.4289482754121634	-33.08	36.08
MFCC_Overall_Average_5	994623	-0.2978992831513065	1.7483692371223167	-19.74	49.66
MFCC_Overall_Average_6	994623	0.9740160068880853	1.5423332699267618	-16.39	78.78
MFCC_Overall_Average_7	994623	-0.7772938419444773	1.2125057142924198	-14.68	42.52
MFCC_Overall_Average_8	994623	1.065497412938001	1.069868514512282	-15.24	55.6
MFCC_Overall_Average_9	994623	-0.6177456971876134	0.959355776314872	-38.85	53.53
MFCC_Overall_Average_10	994623	0.9894734842247073	0.7971012920853822	-7.523	54.79
MFCC_Overall_Average_11	994623	-0.6663714301789341	0.7505053318672916	-11.13	85.05
MFCC_Overall_Average_12	994623	0.2864940668087436	0.6325169315977884	-7.127	59.06
MFCC_Overall_Average_13	994623	-0.5510901588250338	0.5658620831688016	-24.18	28.42

In order to get the correlation between each two features, I computed the correlation matrix by using pyspark.mllib.stat.Statistics.corr() method. We can see from the result that there were features correlated. For instance, the correlation between the column MFCC_Overall_Standard_Deviation_9 and the column MFCC_Overall_Standard_Deviation_8 was 0.809384.

The MSD All Music Genre Dataset (MAGD) had two columns, track ID and genre label. There were total 422713 records in the dataset. I filtered the MAGD dataset to remove the unaccepted mismatch to get the matched song genres. There were 415350 records that were matched.

In order to visualize the distribution of genres, I grouped the genres and counted how many songs of each genre. I used the matplotlib.pyplot to plot the distribution of genres in the pyspark shell and save it as png format to local. The plot is provided below:



We can see from the plot that Pop_Rock was the most.

The track ID in the audio feature dataset were all with single quotes, so the single quotes should be removed before merging the genres dataset and the audio feature dataset.

Audio Similarity
Q2

I selected Logistic regression (lr), Random forest (rf) and Gradient-boosted tree (gbt) to perform the binary classification.

The computational cost of logistic regression is not high, it is easy to understand and realize, but it is easy to under fit the model, the classification accuracy is not high.

Random forests can process very high-dimensional data without the need for feature selection. But after training, what feature are more important can be given by random forests.

Gradient-boosted tree in pyspark does only yet support binary classification with strong predictive power but difficult to calculate in parallel.

I created a new column which was the converted genre label by using the withColumn() method with the pyspark.sql.functions.when() method. When the genre label is “Electronic”, the converted label is “1”, when the genre label is not “Electronic”, the converted label is “0”.

I grouped the converted genre label to compute the count of “1” and “0” labels. The ratio of electronics to others was about 1:9.3. The count of each class is provided below:

```
+-------------+------+
|Convert_label| count|
+-------------+------+
|            1| 40028|
|            0|373265|
+-------------+------+
```

Since the proportion of Electronic in the dataset was only about 10%, I chose stratified sampling to split the dataset to get the training data and test data. I used the sampleBy() method to perform stratified sampling based on the binary labels, randomly extracting 70% Electronic and 70% Others as training set, and the rest as test set. The balance of training set is provided below, the ratio of electronics to others in the training set is also about 1:9.3.

```
+-------------+------+
|Convert_label| count|
+-------------+------+
|            1| 27955|
|            0|260932|
+-------------+------+
```

Most machine learning algorithms work best when the amount of data for each class in the sample is roughly equal. This is because machine learning algorithms are designed to minimize errors. Since the probability of data belonging to most classes is very high in the unbalanced data set, the algorithm is more likely to classify new observations into most classes. So, I chose to balance the data of Class 1 and Class 0 before fitting models. I have tried both oversampling and subsampling on the training data.

I performed oversampling method by three steps:

1. Separate the data of class1 and class0 in the training set
2. Use the takeSample() method on class1 data, set withReplacement to true, and extract samples that match the number of class0.

3. Union the data from the second step with the data from class0 to get a balanced training set.

I performed subsampling method by using the sampleBy() method. Since the ratio of Class 1 and Class 0 was 1:9.3, the sampling ratio of Class 1 was set to 1, and the sampling ratio of Class 0 was set to 0.11.

Before training the model, I performed a series of pre-processing on the dataset. Considering that there are 26 features in the dataset, and some features are highly correlated, I use the VectorAssembler() method to combine all columns into a single vector column, and then use the StandardScaler() method to normalize each feature to have unit standard deviation, and finally use PCA() method to project vectors to a Low-dimensional space.

After combining VectorAssembler(), StandardScaler(), and PCA(), I used the pipeline() method to build the pipeline model using training data. Transform the training data and test data separately using the built pipeline model.

I used the defult parameter values to train the three classification algorithms on the transformed training set.

I created a function which could count the true-positive, true-negative, false-positive and false-negative of the confusion matrix and calculate the recall, precision and accuracy based on the prediction of the model performed on the test set.

The recall, precision and accuracy of each model fitted on the subsampling data are provided below:

	recall	precision	accuracy
lrModel	0.7420690797647643	0.30952874516307355	0.8143256756105011
rfModel	0.735028576161683	0.32318449996358073	0.8249039435397006
gbtModel	0.7745382257930921	0.3070835112147384	0.808514058807453

The recall, precision and accuracy of each model fitted on the oversampling data are provided below:

	recall	precision	accuracy
lrModel	0.744802451751843	0.3094926688235699	0.8139719949198592
rfModel	0.7420690797647643	0.3244251312692377	0.8250084401073903
gbtModel	0.7750352025180154	0.3137932190885006	0.8136906580068486

I created a grid of parameters for each of the three models by using the ParamGridBuilder() method. I chose the numeric parameters for tuning. Based on the default value, a value greater than the default value and a value less than the default value are added in the grid for each parameter. Then I used the CrossValidator() method to tune each model.

The recall, precision and accuracy of each subsampling model after cross-validation are provided below:

	recall	precision	accuracy
lrModel	0.7178828791518264	0.32628091706509055	0.8287703165442182
rfModel	0.764018885115547	0.32635154259835836	0.8240518946031542
gbtModel	0.7771887683260167	0.30735718029350106	0.8084095622397633

The recall, precision and accuracy of each oversampling model after cross-validation are provided below:

	recall	precision	accuracy
lrModel	0.7236809409425992	0.3261169795826957	0.8280629551629343
rfModel	0.756812722604158	0.3325084610065869	0.8289632332845682
gbtModel	0.735939700157376	0.3348660159047224	0.8325161165860168

We can see from the results above that the accuracy of these three models of balanced class data were between 80% and 83%. The recall values of these three models were between 73% and 78%, the precision values were between 33% to 33%. The random forest model had the highest accuracy but the recall was the lowest.

After tuning the parameters using cross validation of the oversampling models, the recall of the lr model and gbt model drop, but the precision increased, overall accuracy has increased. With the cross validation, the recall of the rf model and the precision both increased, the overall accuracy also increased. Cross validation optimizes model performance, but it was very slight.

I fitted the model using unbalanced data and used cross validation to tune hyper parameters. The recall, precision and accuracy of each model fitted on the unbalanced data are provided below:

	recall	precision	accuracy
lrModel	0.8858610121759297	0.3082487894858197	0.7959985852772374
rfModel	0.4817361053590657	0.7543450064850843	0.9344806520585823
gbtModel	0.5195063364532427	0.7071823204419889	0.9324952172724789

The rfModel and the gbtModel of the unbalanced class had a relatively low recall value compared to the balanced class, while the accuracy is as high as 90%.

So, when the dataset was less balanced, the algorithm's accuracy was higher but the recall value was lower.

Audio Similarity
Q3

In the one-vs-all strategy, assuming there are n categories, then n binomial classifiers are created, each classifier classifying one of the categories and the remaining categories. When making predictions, the n binomial classifiers are used for classification, and the probability that the data belongs to the current class is obtained, and one of the categories with the highest probability is selected as the final prediction result.

In the one-vs-one strategy, if there are also n categories, a two-class classifier is created for the two-two categories, and k=n*(n-1)/2 classifiers are obtained. When classifying new data, the k classifiers are sequentially used for classification, and each classification is equivalent to one vote. Which class is classified is equivalent to which class is voted for. After classifying using all k classifiers, it is equivalent to k-voting, and the class with the most votes is selected as the final classification result.

Pyspark has a OneVsRest() method that implements the one-vs-all strategy. As for the one-vs-one strategy, each category can be modeled using the binary classification method in Q2.

I think the one-vs-one strategy will perform better because of the imbalance between the classes in this case.

I converted the genre labels into integer index by using StringIndexer() method and created the new pipeline for multiple class data pre-processing. And I used the randomSplit() method to spalit the dataset into training set and test set, randomly extracting 70% as training set, and the rest as test set.

I chose the random forest algorithm with default parameter values to train the multiple-label model, since it performed better than logistic regression in the binary case and the Gradient-boosted tree is only deal with binary classification.

With the unbalanced data, the accuracy, weightedPrecision, weightedRecall and weightedFalsePositiveRate for multiple classification rf model with default parameter values and the tuned hyperparameters are provided below:

	accuracy	weightedPrecision	weightedRecall	weightedFPRate
rfModel(Default)	0.57303589	0.39676769	0.57303589	0.54513360
rfModel(cv)	0.58091985	0.41725995	0.58091985	0.50005497

We can see that the accuracy of the multi-classification model was only about 58%, which was much lower than the binary classification model. Only five classes had non-zero recall and precision, and the recall and precision for remaining classes are zero.

With the balanced data, each class has the recall and precision values. The larger the amount of data, the higher the recall and precision. The accuracy, weightedPrecision, weightedRecall and weightedFalsePositiveRate for balances training set of rf model with default parameter values and the tuned hyperparameters are provided below:

	accuracy	weightedPrecision	weightedRecall	weightedFPRate
rfModel(Default)	0.34815690	0.55068621	0.34815690	0.07693994
rfModel(cv)	0.34587428	0.56922805	0.34587428	0.06647974

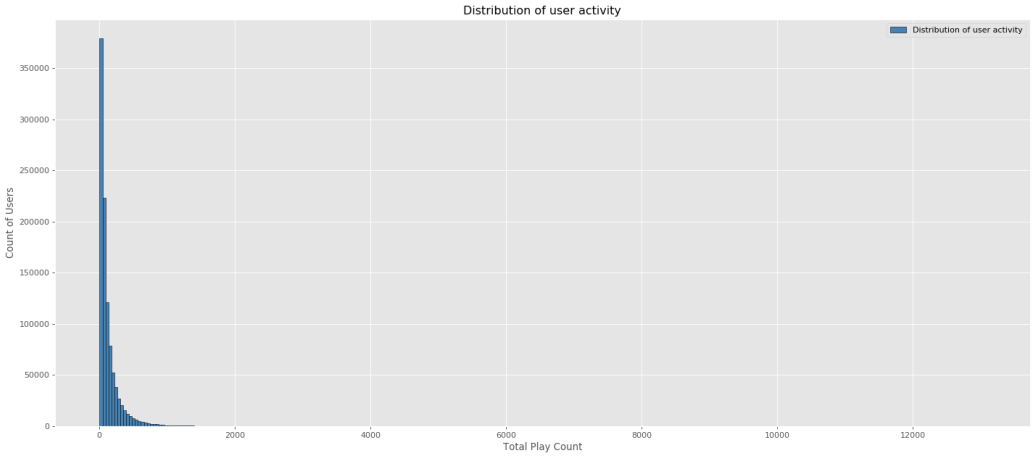
Song Recommendations
Q1

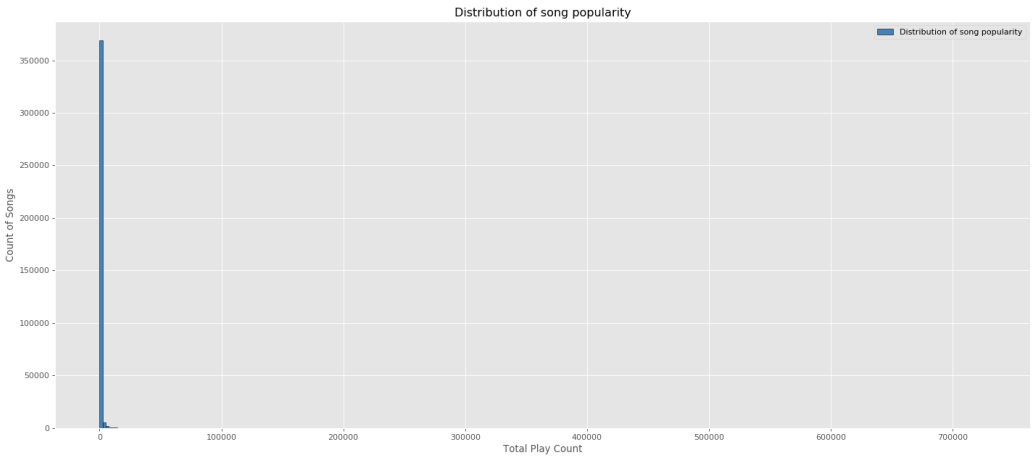
There were 378310 unique songs and 1019318 users in the Taste Profile dataset. I selected each of the user ID column and song ID column, and drop the duplicates to get the count.

The most active user played 195 unique songs, which is just 0.051545% of the total number of the unique songs. I grouped the user ID and sum the played count to get the descending order of the total number of songs that each user played. The first one was the most active user who had 13074 total plays. I filtered the song played by the most active user by the user ID. The top five lines of the count of songs that each user played is provided below.

USER_ID	TOTAL_COUNT
093cb74eb3c517c5179ae24caf0ebec51b24d2a2	13074
119b7c88d58d0c6eb051365c103da5caf817bea6	9104
3fa44653315697f42410a30cb766a4eb102080bb	8025
a2679496cd0af9779a92a13ff7c6af5c81ea8c7b	6506
d7d2d888ae04d16e994d6964214a1de81392ee04	6190

I grouped song ID and sum the played count to get the total number of each song were played. And I used pyspark to plot the distribution of song popularity and the distribution of the user activity and save the images to local directory.





The distribution plot of song popularity showed that the song popularity distribution was right-skewed, which was a result of a lower boundary in the dataset, as well as the user activity.

I converted the dataframe of song popularity and user activity into pandas dataframe and used the pandas quantile() method to get the 1st quantile (N=32) value of user activity, and also used the same method to get the 1st quantile (M=8) of song popularity.

I used the 1st quantile values to filter the users and songs which were less than 32 and 8. There were 254739 users who played songs less than 32 and there are 90750 songs that were played less than 8.

I used left-anti join to remove the user ID which had less than 32 plays and the song ID which had less than 8 plays in the Taste Profile dataset. The filtered dataset had 41978747 lines.

I created a for loop to do the sampling. There are three steps in the loop:

1. I randomly selected 30% data from the whole dataset and used the user ID I selected to do the stratified sampling, which selected 25% songs of each user played as test set.
2. I computed the totel number of the total plays in the test set.
3. If the total number is less than 20% of the total plays, redo the first two steps until the test set contained more than 20% of the plays.

If there is no data of the user in the training set, it is impossible to create a matrix for calculations when using the model to generate recommendations.

Song Recommendations
Q2

Before I trained the model, I used the StringIndexer() method to encode the user ID column to a column of label indices, as well as the song ID column. And then, I split the encoded dataset into training data and test data by using the function I created in Q1. Finally, I used ALS() method to train the implicit matrix model with the parameter implicitPrefs is True.

I selected the encoded user ID column and remove the duplicates to get the first five users and used recommendForUserSubset() method to generate recommendations for each selected users.

In order to compute the performance metrics, I used the RankingMetrics() method in pyspark. The input within the RankingMetrics() method was an RDD of (predicted ranking, ground truth set) pairs. There were four steps getting the RDD of (predicted ranking, ground truth set) pairs.

1. Use the ALSModel fitted above to fit the test set to get the prediction
2. Use the pyspark.sql.Window.partitionBy() method to seperatly sort the songs by the prediction column for each user to get the prediction songs list.
3. Use the same method in step two to get the actual played songs list for each user.
4. Join the prediction songs list and the actual played songs list on the user index and select the the prediction songs list column and the actual played songs list column to get the dataframe of (predicted ranking, ground truth set) pairs.

When using the RankingMetrics() method, I converted the dataframe of (predicted ranking, ground truth set) pairs into rdd.

The matrics created by the RankingMetrics() method contains the precision, ndcg and MAP. The precision at 5 can be computed by using precisionAt(5) method, which was 0.915. The NDCG at 10 can be computed by using ndcgAt(10), which was 0.820. The MAP can be computed by using meanAveragePrecision method, which was 0.251.

We can also use *Mean Average Recall* (MAR) to measure the performance of the model. MAR gives insight into how well the recommender is able to recall all the items the user has rated positively in the test set.

Also, we can compute the *coverage. Coverage* is the percent of items in the training data the model is able to recommend on a test set.

Song Recommendations
Q3

Since the model in Q2 is built based on existing users and items, if there is a lack of information about the users or items, a cold start problem will occur in the recommendation system. Once there is relatively little information, such as new users and new items which contain very little information, which leads to the inability to recommend user recommendations. Providing recommendations in the case of new users is actually very difficult, there is very little historical information about available users, and for new items, there is usually no rating and other relevant data, so collaborative filtering does not provide useful recommendations in new situations.

In order to solve the above problem of cold start, we can use Non-personalized recommendation system. This kind of recommendation system is non-personalized and will recommend the same items to everyone. They rely on user evaluation data and generate recommendations like "selling", "most popular", "hot trends" and so on.

When the user has a record of browsing, purchasing, etc., use Content-based recommendation system according to their behaviour. In this type of algorithm, items are first represented by their attributes (attributes can be the title of the song, genre, author, etc.). The user profile is then built by user rating these attributes (user's favourite type or author). Ratings can be obtained by taking the user's implicit actions, such as reading, purchasing, and clicking. This model then predicts the user's rating of the attribute to generate a recommendation.

When the user has just logged into the system, the Non-personalized recommendation system can be used to recommend the user. After the user has some relevant information about the item, the Content-based recommendation system can be used to recommend the items.

Metadata contains detailed information about the song, such as artist similarity, popularity, geographic information, song genre, and more. The songs can be classified by using clustering, and the content-based recommendation system described above can be used for recommendation. For example, according to artist similarity, songs can be clustered on the attribute of artist similarity.