Table of Contents

Introduction

According to Statista until April 2022, 63% of the World's population that is about 5 billion people were active internet users. The widespread usage of the internet has brought significant changes in the music industry. Like advancement of the music, broadcast platforms are helping the people who choose to listen to online music at their convenience by recommending the songs and classifying the songs based on their genre. One such attempt has been made to classify the music genre centred using binary classification and multiclass classification, and with the help of collaborative filtering using the ALS model the song recommendation system is developed using Million Song Dataset (MSD) in Spark and reported in this report. The current report focuses on data processing, audio similarity and song recommendations.

Data processing

The Million Song Dataset (MSD) is a freely available largest research dataset. It is a collection of audio features and metadata for a million contemporary music tracks. The audio features such as loudness, beat, tempo and time signature, and metadata has song ID, track ID and artist ID along with other 51 fields. The track ID is an ID given to a particular recording of a song. Each song can have multiple track IDs. Along with its dataset, the MSD dataset is a collection of other datasets, which are contributed by organisations and the community.

Structure of the dataset

To get the directory structure of the dataset, size, data types, file formats, the number of rows and even the way they are stored in hdfs dfs commands are used.

```
hdfs dfs -du -h -v /data/msd
```

The code gives the size of each directory in the MSD dataset.

The size of the MSD dataset is 12.9 GB (Figure 1). The main directory msd has 4 sub-directories audio, genre, main, and tasteprofile. Audio consumes 12.3 GB, which is about 94.7% of the total size.

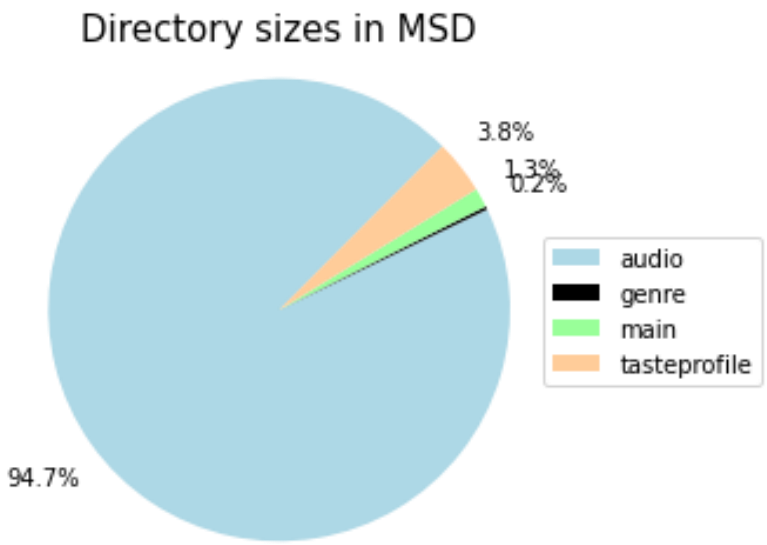

Figure 1. Distribution of the file size.

The audio directory contains audio feature sets from the Music Information Retrieval research group at the Vienna University of Technology. Attributes, features and statistics are the three subdirectories of audio. The attributes directory contains 13 comma-separated files that have the names of the attributes from the header of the ARFF. The features directory has 13 subdirectories and each directory has 8 compressed comma-separated files all containing the audio features. In the statistics directory, we have a compressed csv formatted file which has the data related to the statistics of additional tracks. The genre subdirectory in msd is the smallest one with 30.1 MB of data in it. It has three tab-separated files in a text format where the data is stored in a table structure, each record in a table is one line in the text file. The size of the main directory is 174.4 MB. There is a summary sub-directory with 2 compressed comma-separated files, which contains all the metadata for the million song dataset. In tasteprofile, the size is about 490.4 MB. It contains the user-song play counts from the Taste Profile dataset as well as logs. It has triplets as a subdirectory of mismatches which has 8 partitions of tsv formatted files.

To count the number of rows in each of the datasets, loop is used. In a specific directory, each file is first uncompressed and then counts the number of rows in it.

```
$ for i in $(hdfs dfs -find /data/msd/audio/features -name 'msd-*')
> do
>    (hdfs dfs -cat $i/* | gunzip | wc -l)
> done
# 994623
# 994623
# 994623
# 994623
# 994623
# 994623
# 995001
# 994188
# 994188
# 994188
# 994188
# 994188
# 994188
```

The above code is used to get the row count for each compressed csv format file in a subdirectory where the name starts with msd in the features directory. From the output, we can see that the total number of unique songs is between 994,188 and 994,623 which is approximately the same as the 994,960 provided in the audio features description.

To get the data types in each of these files *awk* command is used.

```
$ hdfs dfs -cat "/data/msd/audio/attributes/*" | awk -F',' '{print $2}' | sort | uniq
# NUMERIC
# real
# real
# string
# string
# STRING
```

The above code helps to get the data types for all the files in the attributes directory.

```
hdfs dfs -ls -R /data/msd > files.txt
```

The code mentioned above gives the list of files in msd directory. The output is then directed to a text file and formatted as shown below.

Name and Type		Size in bytes
msd		
|--**audio**/		12.3 GB
|--attributes/		103.0 KB
|--msd-jmir-area-of-moments-all-v1.0.attributes.csv		
|--msd-jmir-lpc-all-v1.0.attributes.csv		
|--msd-jmir-methods-of-moments-all-v1.0.attributes.csv		
|--msd-jmir-mfcc-all-v1.0.attributes.csv		
|--msd-jmir-spectral-all-all-v1.0.attributes.csv		
|--msd-jmir-spectral-derivatives-all-all-v1.0.attributes.csv		
|--msd-marsyas-timbral-v1.0.attributes.csv		
|--msd-mvd-v1.0.attributes.csv		
|--msd-rh-v1.0.attributes.csv		
|--msd-rp-v1.0.attributes.csv		
|--msd-ssd-v1.0.attributes.csv		
|--msd-trh-v1.0.attributes.csv		
|--msd-tssd-v1.0.attributes.csv		
|--features/		12.2 GB
|--msd-jmir-area-of-moments-all-v1.0.csv/		
|--part-00000.csv.gz		
|--part-00001.csv.gz		
|--part-00002.csv.gz		
|--part-00003.csv.gz		
|--part-00004.csv.gz		
|--part-00005.csv.gz		
|--part-00006.csv.gz		
|--part-00007.csv.gz		
|--msd-jmir-lpc-all-v1.0.csv/	8 partitions	
|--msd-jmir-methods-of-moments-all-v1.0.csv/	8 partitions	
|--msd-jmir-mfcc-all-v1.0.csv/	8 partitions	
|--msd-jmir-spectral-all-all-v1.0.csv/	8 partitions	
|--msd-jmir-spectral-derivatives-all-all-v1.0.csv/	8 partitions	
|--msd-marsyas-timbral-v1.0.csv/	8 partitions	
|--msd-mvd-v1.0.csv/	8 partitions	
|--msd-rh-v1.0.csv/	8 partitions	
|--msd-rp-v1.0.csv/	8 partitions	
|--msd-ssd-v1.0.csv/	8 partitions	
|--msd-trh-v1.0.csv/	8 partitions	
|--msd-tssd-v1.0.csv/	8 partitions	
|--statistics/		40.3 MB
|--sample_properties.csv.gz		
|--**genre**/		30.1 MB
|--msd-MAGD-genreAssignment.tsv		
|--msd-MASD-styleAssignment.tsv		
|--msd-topMAGD-genreAssignment.tsv		
|--**main**/		174.4 MB
|--summary/		

```
        |--analysis.csv.gz
        |--metadata.csv.gz
|--tasteprofile/                                    490.4 MB
   |--mismatches/
        |--sid_matches_manually_accepted.txt
        |--sid_mismatches.txt
        |--triplets.tsv/                        8 partitions
```

All the directories from the features have 8 partitions of compressed CSV format files. Even the triplets from the mismatches directory have 8 partitions of tsv format files. Each compressed file needs to be uncompressed on a single cluster. The maximum level of parallelism for these 8 partitions could be 8. The repartition method allows us to increase or decrease the partitions. It does a full shuffle of data which is an expensive task. Since a cluster has 8 cores, we don't have to increase the partitions. Each core runs a partition successfully and all these are processed in parallel.

Loading the dataset

To load the data, first, the datatypes for the files in tasteprofile and audio/features were obtained by using hdfs awk command as shown above in code3. Using this information, a schema is built in spark for these files to maintain consistency. The data is loaded from matches_manually_accepted and mismatches text file in mismatches subdirectory in tasteprofiles directory that has the same schema as shown below.

```
mismatches_schema = StructType([
 StructField("song_id", StringType(), True),
 StructField("song_artist", StringType(), True),
 StructField("song_title", StringType(), True),
 StructField("track_id", StringType(), True),
 StructField("track_artist", StringType(), True),
 StructField("track_title", StringType(), True)
])
```

```
with open("/data/msd/tasteprofile/mismatches/sid_matches_manually_accepted.txt", "r") as f:
 lines = f.readlines()
 sid_matches_manually_accepted = []
 for line in lines:
  if line.startswith("< ERROR: "):
   a = line[10:28]
   b = line[29:47]
   c, d = line[49:-1].split("  !=  ")
   e, f = c.split("  -  ")
   g, h = d.split("  -  ")
   sid_matches_manually_accepted.append((a, e, f, b, g, h))
matches_manually_accepted = spark.createDataFrame(sc.parallelize(sid_matches_manually_accepted,
8), schema=mismatches_schema)
matches_manually_accepted.cache()
```

The data from mismatches text file is read in the same way. These two files are filtered to remove the songs which were mismatched based on song_id. Since these are text files, readline() function and the index number specifying the range of values are used to get the data. Since the data in the triplets directory is divided into 8 partitions, they are all in compressed tsv format, read.format("csv") is used along with the delimiter as \t and decompressed. There were 48,373,586 songs in triplets. The mismatches are joined with matches_manually_accepted using left anti join. Then the triplets is joined with this new dataframe to remove the mismatched songs. After removing the songs that were mismatched we end up with 45,795,111 songs.

From hdfs awk command in the audio directory, the attributes and the features were checked and found that they have the same datatypes and attribute names. Only one schema is defined, for this as a tuple where the attribute names along with the data type are created.

```
audio_attribute_type_mapping = {
  "NUMERIC": DoubleType(),
  "real": DoubleType(),
  "string": StringType(),
  "STRING": StringType()
}
```

To read all the files in the attributes directory, a list with all the file names has been created.

```
audio_dataset_names = [
 "msd-jmir-area-of-moments-all-v1.0",
 "msd-jmir-lpc-all-v1.0",
 "msd-jmir-methods-of-moments-all-v1.0",
 :
```

for loop is used to append the tuple based on the attribute names and attributes files to infer the schema for these files.

```
for audio_dataset_name in audio_dataset_names:
 print(audio_dataset_name)

 audio_dataset_path = f"/data/msd/audio/attributes/{audio_dataset_name}.attributes.csv"
 with open(audio_dataset_path, "r") as f:
  rows = [line.strip().split(",") for line in f.readlines()]

 audio_dataset_schemas[audio_dataset_name] = StructType([
 StructField(row[0], audio_attribute_type_mapping[row[1]], True) for row in rows
 ])

 print(audio_dataset_schemas[audio_dataset_name])
```

The audioFeatures has 11 features in it. The track id is a string and all other variables are of double type.

Output:

```
        {
          'MSD_TRACKID': "'TRHFHQZ12903C9E2D5'",
          'Method_of_Moments_Overall_Average_1': 0.319,
          'Method_of_Moments_Overall_Average_2': 33.41,
          'Method_of_Moments_Overall_Average_3': 1371.0,
          'Method_of_Moments_Overall_Average_4': 64240.0,
          'Method_of_Moments_Overall_Average_5': 8398000.0,
          'Method_of_Moments_Overall_Standard_Deviation_1': 0.1545,
          'Method_of_Moments_Overall_Standard_Deviation_2': 13.11,
          'Method_of_Moments_Overall_Standard_Deviation_3': 840.0,
          'Method_of_Moments_Overall_Standard_Deviation_4': 41080.0,
          'Method_of_Moments_Overall_Standard_Deviation_5': 7108000.0
        }
```

Audio similarity

In this section, the numerical representation of a song's audio waveform is used to predict its genre. For doing this as per the requirement the smallest dataset "msd-jmir-methods-of-moments-all-v1.0" is selected from the audio/ features directory. The below output contains the 11 different features in the dataset. MSD_TRACKID is a track id and it is unique and the rest of the 10 features are of float datatype. There are 994,623 rows in the audioFeatures dataframe.

Descriptive statistics

By dropping the MSD_TRACKID, describe() is used to get the statistics of the audioFeatures. The output values in the rdd are converted to pandas dataframe rounded to two decimal places and were transposed.

Output	count	mean	std dev	min	max
Method_of_Moments_Overall_Standard_Deviation_1	994623	0.15	0.07	0	0.96
Method_of_Moments_Overall_Standard_Deviation_2	994623	10.38	3.87	0	55.42
Method_of_Moments_Overall_Standard_Deviation_3	994623	526.81	180.44	0	2919
Method_of_Moments_Overall_Standard_Deviation_4	994623	35071.98	12806.82	0	407100
Method_of_Moments_Overall_Standard_Deviation_5	994623	5297870	2089356	0	4.66E+07
Method_of_Moments_Overall_Average_1	994623	0.35	0.19	0	2.65
Method_of_Moments_Overall_Average_2	994623	27.46	8.35	0	117
Method_of_Moments_Overall_Average_3	994623	1495.81	505.89	0	5834
Method_of_Moments_Overall_Average_4	994623	143165.46	50494.28	-146300.0	452500.0
Method_of_Moments_Overall_Average_5	994623	2.40	9307340.30	0	9.47E7

The count in the above statistics shows that all the variables have 994,623 rows of data which is the same number of rows in the selected dataset from audio/features directory. This shows that we have no null values in the data. The minimum value for all the variables is zero except for Method_of_Moments_Overall_Average_4, which has a negative value of -146300. From the mean and standard deviation values, of Method_of_Moments_Overall_Average_5 we can see that the standard deviation value is much larger than the mean. This could be due to the presence of outliers in the data where both the standard deviation and mean values are affected. It is hard to get an idea of the distribution of data since the median is not in the descriptive statistics. At least by checking the correlation between the variables in the dataset, the correlation between the predictor variables can be found.

Feature importance and selection

```
corrMatrix = Correlation.corr(df_vector, vector_col,'pearson').collect()[0][0].toArray().tolist()
```

To get the correlation matrix, all the numeric columns were filtered and converted to a vector by using the VectorAssembler and transform it. The result of this vector for each track id is stored in a new column df_vector. The Correlation.corr() function along with the Pearson method is used to determine if there are any correlations between the numeric variables. The result is stored in a vector_col. To print it in a table format, the collected data is converted into an array and then to a list. By plotting these values in Python, we can see some of these predictor variables are highly correlated with each other (Figure 2).

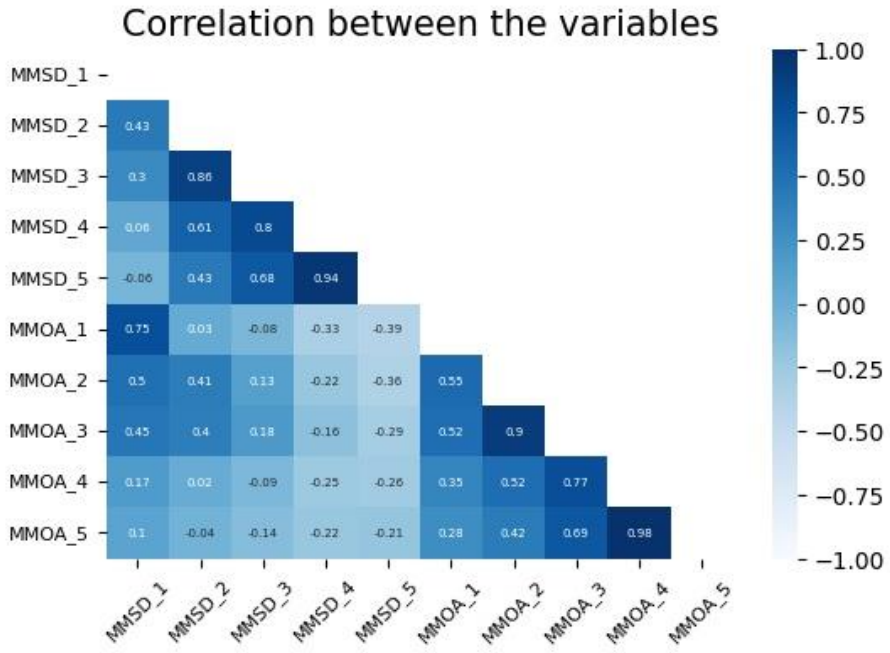

Figure 2. Correlation matrix between the predictor variables.

Three variables are highly correlated with the other variables as shown in Figure 2. MMSD_5 (Method_of_Moments_Overall_Standard_Deviation_5) has 0.94, MMOA_2 (Method_of_Moments_Overall_Average_2) has 0.9 and MMOA_4(Method_of_Moments_Overall_Average_4) has 0.98 that are highly correlated with other predictor variables.

Distribution of genres

To get the distribution of songs based on the genre type loaded, the MSD All Music Genre Dataset (MAGD) are in tsv formatted files, therefore the '/t' delimiter is used. The schema for this is defined according to the structure of the data in MAGD, which has two columns TRACK_ID and GENRE_LABEL. There was a total of 422,714 records in the dataset.

To get the distribution of genres for the songs that were matched, the GENRE_LABEL is grouped and counted. This data is then used for plotting. The plot below shows the distribution of songs in each genre (Figure 3). From the distribution, we can see that the Pop_Rock there are 238,786 songs and it has the most number of songs. The Electronic is the second genre type in the distribution has only 41,075, which has 197,711 fewer songs than the Pop_Rock. Only 200 songs are in Holiday, this has the least number of songs.

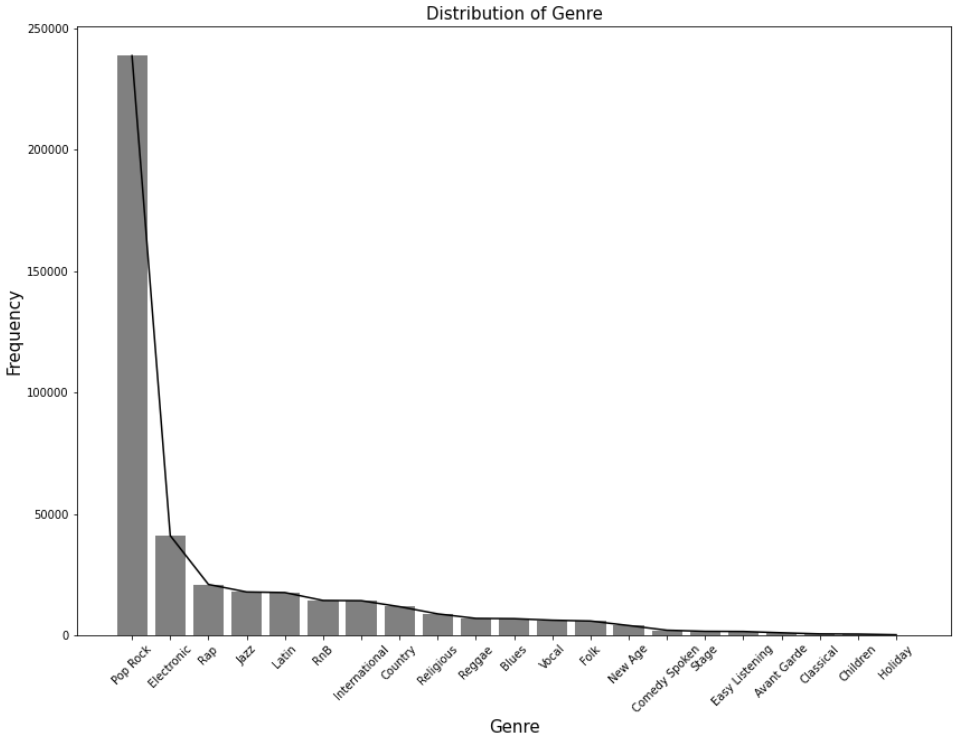

Figure 3. Distribution of genres in MAGD dataset.

Merging the datasets

'MSD_TRACKID': "'TRHFHQZ12903C9E2D5'"

The track id in MSD_TRACKID from audioFeatures dataset is in double-quotes. The data in it is replaced by using a regular expression to remove it and also the column name is changed to TRACK_ID as it has to match with the column name in the genre dataset. The two datasets were then merged using the left anti join and filtering the values where the GENRE_LABEL is not null some of the records were removed. To ensure that every song in the new dataset has a Label for it. The audioGenre dataset has all the other variables of audioFeatures and the GENRE_LABEL from the genre dataset.

```
 audioGenre.printSchema()
root
 |-- TRACK_ID: string (nullable = true)
 |-- Method_of_Moments_Overall_Standard_Deviation_1: double (nullable = true)
 |-- Method_of_Moments_Overall_Standard_Deviation_2: double (nullable = true)
 |-- Method_of_Moments_Overall_Standard_Deviation_3: double (nullable = true)
 |-- Method_of_Moments_Overall_Standard_Deviation_4: double (nullable = true)
 |-- Method_of_Moments_Overall_Standard_Deviation_5: double (nullable = true)
 |-- Method_of_Moments_Overall_Average_1: double (nullable = true)
 |-- Method_of_Moments_Overall_Average_2: double (nullable = true)
 |-- Method_of_Moments_Overall_Average_3: double (nullable = true)
 |-- Method_of_Moments_Overall_Average_4: double (nullable = true)
 |-- Method_of_Moments_Overall_Average_5: double (nullable = true)
 |-- GENRE_LABEL: string (nullable = true)
audioGenre.show()
```

Vector transformations

Using VectorAssembler, a vector containing all the selected predictor variables is created. The VectorAssembler is initiated by specifying the column containing the features and the output column containing the assembled vector.

Binary classification

The classifiers are used to classify the numerical representations of a song's audio waveform to predict its genre type for binary classification. For this purpose, the audio features dataset and MAGD dataset in MSD is used.

Classification method description

Logistic Regression, Gradient boosted tree and Random forest are the three classification methods that were chosen. Logistic Regression classifier in terms of expandability can easily be expanded to multinomial regression. The coefficients of the variables obtained from the trained model are easy to interpret the results like the feature importance. It is very efficient to train. It can easily classify between the classes when they are linearly separable this is when we end up with good accuracy. There is a chance of overfitting when there are fewer observations compared to the features. Feature scaling is required as it can handle only continuous data. A good accuracy can be obtained by scaling the features.

Gradient boosted trees (GBT) use shallow trees, which are grown sequentially. Hence, it is less parallelisable. Each tree corrects the classification error of the previous trees, which boosts in terms of its performance. It handles the non-linear data more efficiently. GBT requires no feature scaling as it can handle any type of data. GBT trains faster especially with large datasets. It controls both the variance and bias in the model. The downside of it is Spark does not support the implementation of multi-class classification. It's hard to interpret the results. They are prone to overfit the model by using a low learning rate that can be minimised.

Random forest is based on the bagging technique. These small trees are built parallel. Hence, it creates many trees on the subset of data and combines all the trees. By doing so, it reduces the problem of overfitting as in GBTs. The training is much slower when compared to GBT. It is used for both classification and regression problems but works well with non-linear data. It can control only high variance in the model. The feature scaling is not required as it can handle both continuous and categorical data. The RF method outputs the feature importance, which describes variables in the dataset that can improve the model. Hence, its output is easy to interpret.

Standardising

The Logistic regression method, which is a Gradient descent based method requires scaling the features. The GBT and DT are tree-based methods that do not require scaling the features. Feature scaling is done using StandardScaler to improve the model accuracy or F1 score.

```
:<EOF>
+-----------------+-------------------+-------------------+-----+
|         TRACK_ID|           Features|     ScaledFeatures|label|
+-----------------+-------------------+-------------------+-----+
|TRAABPK128F424CFDB|[0.1208,6.738,215...|[1.87165692771989...|    0|
|TRAACER128F4290F96|[0.2838,8.995,429...|[4.39715427224260...|    0|
|TRAADYB128F92D7E73|[0.1346,7.321,499...|[2.08547204032365...|    0|
|TRAAGRV128F93526C0|[0.1076,7.401,389...|[1.66713812435977...|    0|
|TRAAHAU128F9313A3D|[0.08485,9.031,44...|[1.31465306553835...|    0|
+-----------------+-------------------+-------------------+-----+
```

Class balance of the binary label

The GENRE_LABEL is converted into a binary column by grouping and assigning as Class 1 when the genre type is Electronic and the rest are assigned to Class 0. The class balance in Class 1 is 40,666 and in Class 0 is 379,954 (Figure 4).

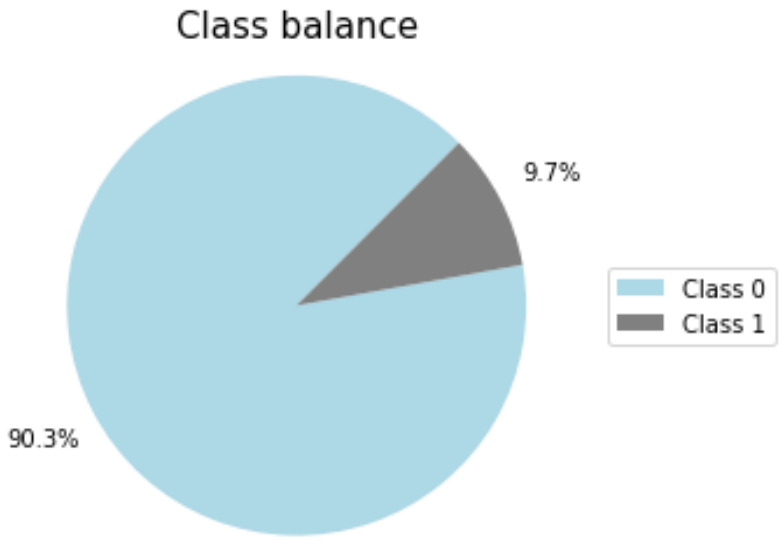

Figure 4. Class balance in genre column after grouping based on genre type.

As shown in Figure 4, approximately 90.3% of the songs are in Class 0 and only 9.7% are in Class 1. We have an issue of class imbalance approximately in the ratio of 1:10. While training the models with the class imbalance will predict the results in favour of Class 0 as they have a large number of songs when compared to Class 1.

Splitting the dataset

Stratified random sampling technique is used to ensure that the class balance is preserved even after splitting it into test and train split. Window() is defined in the helper function called temp, this helper function is used for stratified random split based on randomly selected row by checking the class count and the split to be 70%.

```
train = temp
for c in classes:
    train = train.where((col("label") != c) | (col("Row") < class_counts[c] * 0.7))
```

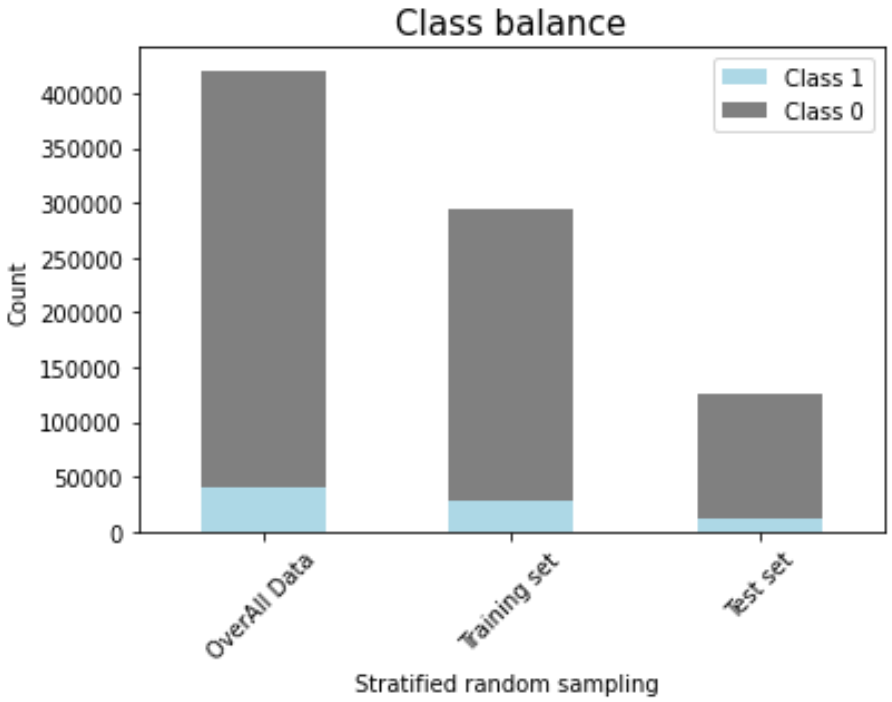

Figure 5. Data splitting strategy.

As shown in Figure 5, the data split into train and test set has the same proportion as that of the overall data. The training set has 70% of data from both the classes in it and the remaining 30% of the data in the test set. The seed is used to reproduce the same result.

Resampling methods

Resampling or rebalancing is only done after the train test split to avoid data leakage. The train data is resampled but not the test data. Down-sampling and the over-sampling techniques are used to balance the data. In down-sampling, the observations in Class 0 is the frequent class that is reduced to the size of the Class 1 by randomly selecting the data.

```
trainDownsampled = (
   trainData
   .withColumn("Random", rand())
   .where((col("label") != 0) | ((col("label") == 0) & (col("Random") < 1 * (40666 / 379954))))
)
```

Whereas in Over-sampling, the rare class that is Class 1 the observations are duplicated to create new observations to ensure that the class balance is the same. For this, a user-defined function is created to convert the data in the form of an array of array columns which is then exploded into a row using explode(). Class 1 is balanced with Class 0 in both scenarios (Figure 6).

```
randomResample_udf = udf(lambda x: randomResample(x, n, p), ArrayType(IntegerType()))
```

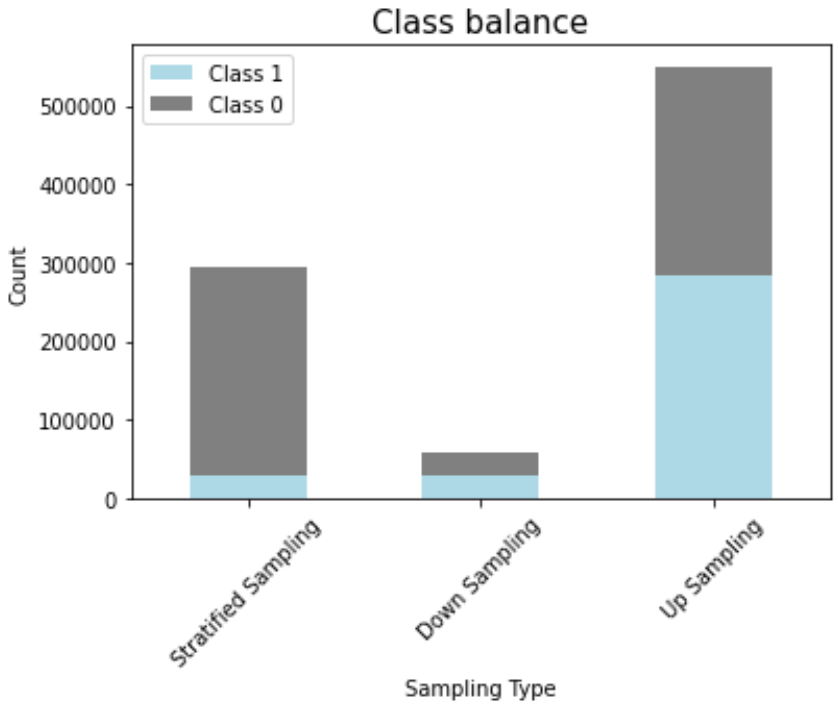

Figure 6. Rebalancing techniques.

Model evaluation and metrics

Both the test data and the train data are transformed before resampling is done. The training dataset is used to train the model using the three different classifiers. It is then tested on the unseen data or the test data.

```
lr = LogisticRegression(featuresCol='ScaledFeatures', labelCol='label')
```

A helper function is created to print the metrics like actual total, actual positive, actual negative, threshold, true positives (TP), false positives (FP), false negatives (FN), true negatives (TN), recall, precision, accuracy, F1 score and area under ROC (auROC). For class imbalance, F1 score, precision and recall are used as metrics for choosing the best model. Accuracy, precision, recall and auROC are chosen for the class balanced data. To get the metrics, the model is trained on the train data and then tested with test data; the result of the transformed test data is set as a parameter to the helper function. The threshold is set to a default value of 0.5 (Table 1). To change the threshold value we can pass it as a parameter along with the predictions.

```
lr_model = lr.fit(trainData)
predictions = lr_model.transform(testData)
predictions.cache()
print_binary_metrics(predictions,threshold = 0.1)
```

Table 1. Results for the stratified random sampling data for different threshold levels.

Model	Threshold	Recall	Precision	F1 score
Logistic regression	0.5	0.0061	0.2450	0.0118
	0.1	0.6237	0.1732	0.2711
Gradient boosting trees	0.5	0.0875	0.6301	0.1537
	0.1	0.7274	0.2089	0.3245
Random forest	0.5	0.0000	0.0000	0.0000
	0.1	0.5773	0.2140	0.3123

The results for class imbalance data at a threshold value of 0.5, the recall for all the candidate models was very low (Table 1). The candidate models when trained and tested for a threshold value of 0.1, the recall has improved. F1 score, Recall and Precision are the metrics used to evaluate the model performance. There is a trade-off between precision and recall; as the threshold decreases more false positives are predicted, decreases the precision value and increases the recall value. Even though the F1 scores for all these candidate models are very low, the GBT model is the best model for the threshold of 0.1, where 32% of the time it is predicted as Class 1.

The recall, precision, accuracy and auROC are the metrics chosen to evaluate the model in balanced data (Table 2). Gradient boosting trees has the highest accuracy of 75% and even the auROC value is 79.79%. For the balanced dataset, we can see that the recall values are in a range between 61 to 69%, which is better than the unbalanced data.

Table 2. Results for the rebalanced data.

Model	Resampling	Recall	Precision	Accuracy	auROC
Logistic regression	Down Sampling	0.6341	0.1694	0.6640	0.6976
	Up Sampling	0.6138	0.1740	0.6810	0.6977
Gradient boosting trees	Down Sampling	0.6906	0.2247	0.7398	0.7973
	Up Sampling	0.6687	0.2327	0.7546	0.7979
Random forest	Down Sampling	0.6693	0.2122	0.7278	0.7780
	Up Sampling	0.6581	0.2159	0.7359	0.7787

Hyper-parameter tuning

From the candidate models, the top-performing model is Gradient boosting trees. This is fine-tuned with the following preprocessing variations like Vectorising, Standard scaling, cross-validated for 5 folds and hyper-parameters maximum depth of the tree, maximum bins and max iterations were set to a grid with a range of values as shown in the code snippet. The ParamGridBuilder() is used to define hyper-parameters and CrossValidator() to tune each model.

```
paramGrid = (ParamGridBuilder()
        .addGrid(gbt.maxDepth, [2, 4, 6])
        .addGrid(gbt.maxBins, [20, 60])
        .addGrid(gbt.maxIter, [10, 20])
        .build())
cv = CrossValidator(estimator=gbt, estimatorParamMaps=paramGrid,
evaluator=BinaryClassificationEvaluator(), numFolds=5)
```

```
cvModel = cv.fit(trainDownsampled)
predictions = cvModel.transform(testData)
predictions.cache()
print_binary_metrics(predictions)
```

Table 3. Results for the gradient boosting model.

Type of data	Recall	Precision	Accuracy	F1 score	auROC
Stratified Random split	0.0875	0.6301	0.9068	0.1537	0.7949
Up Sampled	0.6784	0.2292	0.74834		0.7978
Down Sampled	0.6906	0.2247	0.7398		0.7973
Cross validated	0.6907	0.2325	0.7496		0.8031

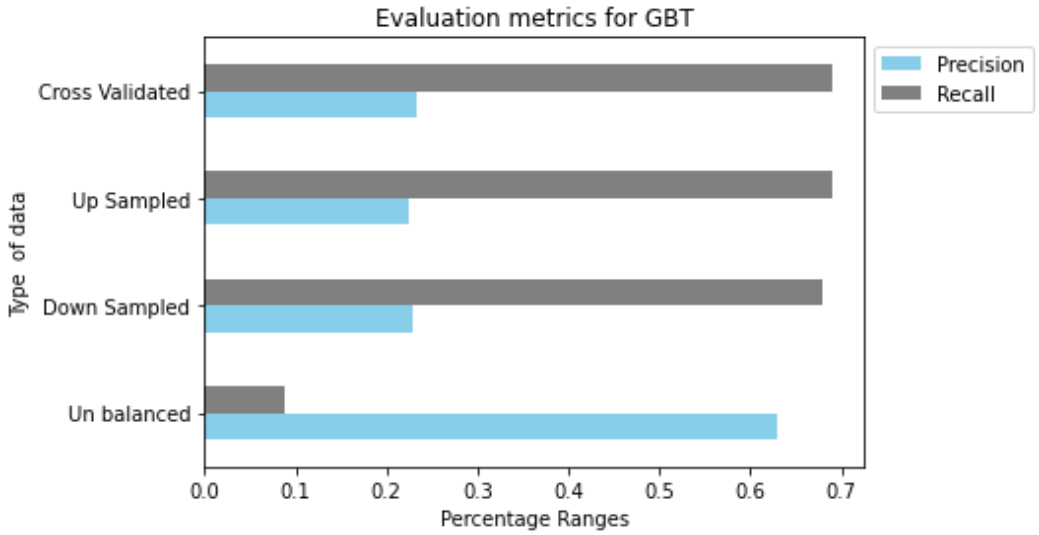

Figure 7. Precision and recall for Gradient boosting trees.

Cross validation to tune the hyper-parameters for GBT has an accuracy value of 74.96% and a value of 80% for auROC (Table 3). Class1 is predicted correctly in the unseen data 80% of the time. By fine-tuning the model there is a slight increase in the accuracy with a massive computational tradeoff when compared to the other models using only hyper-parameters on the balanced and imbalanced dataset. The precision is more in unbalanced data as the model predicts more false negatives when compared it with balanced data (Figure 7).

Multiclass classification

Even though the GBT is the best model since spark does not support the implementation of the multiclass classification problem. The random forest method is chosen to do the multiclass classification based on the genres in MSD dataset. The audioGenre dataset has all the columns from audioFeatures and the GENRE_LABEL from MAGD dataset is used to do multiclass classification.

Data pre-processing

The GENRE_LABEL in the audioGenre dataset using StringIndexer() is converted to integer index and stored the values in a new column called a label. The dataset has 21 different genres each genre is represented as a class. The Pop_Rock class is the largest and has a count of 238,786

songs and the Holiday class is the smallest with a count of 200 songs. There is a huge difference between these classes.

Splitting the dataset

As mentioned above in the binary class classification, the Window partitionBy() is used to do the stratified random split. The data from each class is split into 70/30. Where 70% of data is in the train set and 30% of data is in the test set. Then the numeric features which are not highly correlated are converted into the vector using VectorAssembler() and StandardScaler() is used to normalise the data using the pipeline for training data. It is only done after the split to avoid data leakage. The test data is transformed separately.

Since the above data has a class imbalance problem these classes have to be balanced. This is done using Random downsampling for the data that has uniform distribution and upsampling for the data that has Poisson distribution. A threshold of 50,000 is set as the upper boundary where the classes that have a count above 50,000 are downsampled and an upper threshold is 500 where the classes that have a count less than 500 are upsampled. A helper function random_resample_udf is created to identify the classes that are above and below the specified thresholds and remove or add observations based on the conditions. This data is then sent to the Sample column, which is in the form of an array of arrays. Using the explode(), the data is exploded into rows.

```
count_upper_bound = 50000
count_lower_bound = 500
training_resampled = (
   trainMCData
   .withColumn("Sample", random_resample_udf(col("label")))
   .select(
      col("ScaledFeatures"),
      col("label"),
      explode(col("Sample")).alias("Sample")
   )
   .drop("Sample"))
```

Model evaluation and metrics based on class balance

The random forest method is then trained and tested using both the balanced and imbalanced data. A user-defined function is created to print the metrics like Weighted precision, Weighted Recall, Weighted F(1) Score (Table 4), confusion matrix and precision, recall and F1 for each class.

```
:
labels = class_labels.select('label').rdd.map(lambda row : row[0]).collect()
   for label in labels:
      print("Class %s precision =  %s" % (label, metrics.precision(label)))
      print("       %s  recall =     %s" % (label, metrics.recall(label)))
      print("       %s  F1 Measure = %s" % (label, metrics.fMeasure(label, beta=1.0)))

   confusion_matrix = metrics.confusionMatrix().toArray().astype(int)
   labels = [int(l) for l in metrics.call('labels')]
   confusion_matrix = pd.DataFrame(confusion_matrix , index=labels, columns=labels)
:
```

Table 4. Results for the multiclass classification using Random forest model.

Type of data	Weighted precision	Weighted Recall	Accuracy	Weighted F(1) Score
Unbalanced	0.3699	0.5707	0.5718	0.4272
Balanced	0.5669	0.2231	0.2259	0.2240

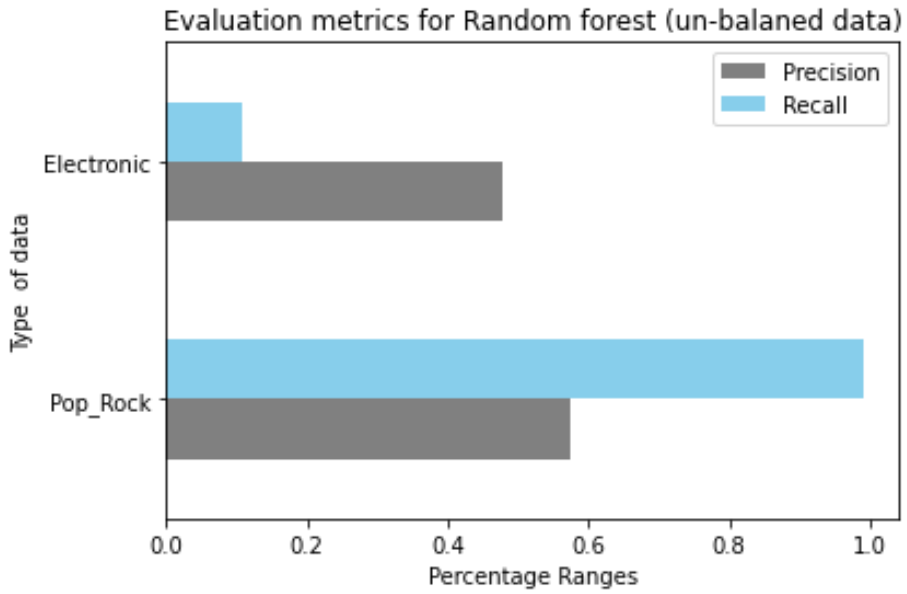

Figure 8. Precision and recall for Random forest model.

The Pop_Rock and Electronic were the two classes (Figure 8) that had the precision and recall values rest of the other classes had 0 values. The Random forest was able to predict only these classes as the dataset had class imbalance in it. Since the large portion of the data is from these two genres.

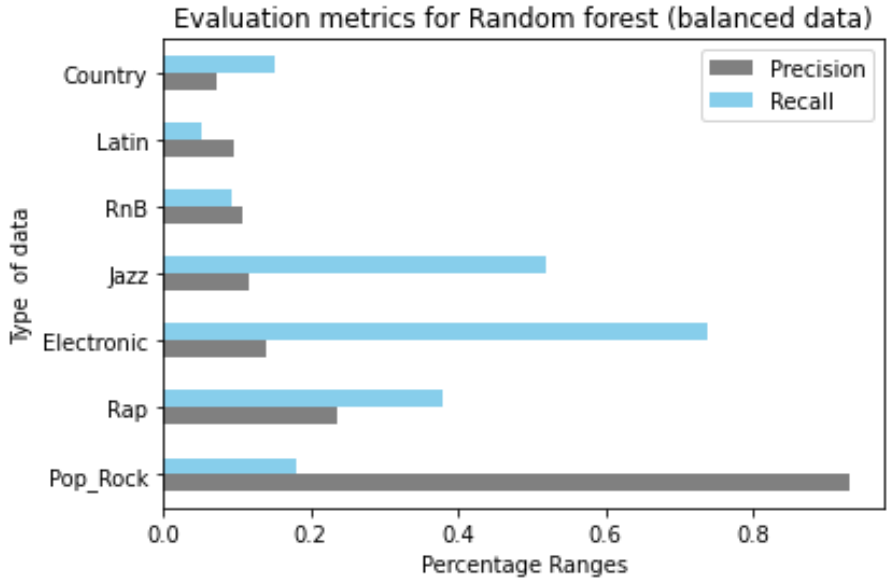

Figure 9. Precision and recall for Random forest model on balanced data.

The Pop_Rock, Electronic, Rap, Jazz, RnB, Latin and Country seven classes (Figure 9) the precision and recall values rest of the other classes had 0 values. The Random forest was able to predict only these classes even when the data in the dataset was balanced using the lower and upper threshold levels.

Song recommendations

The Taste Profile dataset is used to develop a song recommended system based on collaborative filtering. This is done by searching a large group of users and finding the subsets of users who have played similar songs and recommending them to a particular user.

Data analysis

In MSD dataset, Taste Profile has a directory called triplets which are 8 compressed tsv formatted files. In order to get the data from these files, a schema is defined that suits the

structure of the data in tsv files. Each file has 3 features or columns in it. Along with this the other two files sid_matches_manually_accepted and sid_mismatches which are in text format are read. Both these files have the same schema they have five features in each file. These are merged based on the song_id using left anti-join to get the user_id, song_id and playcounts that are not mismatched.

```
mismatches_not_accepted = mismatches.join(matches_manually_accepted, on="song_id",
how="left_anti")
triplets_not_mismatched = triplets.join(mismatches_not_accepted, on="song_id", how="left_anti")
```

There are 378,310 unique songs and 1,019,318 users in the dataset. The most active user is '093cb74eb3c517c5179ae24caf0ebec51b24d2a2' who has played 195 unique songs, which is approximately 0.05% of the total number of unique songs in the dataset.

```
:
groupBy("user_id")
  .agg(
    F.count(F.col("song_id")).alias("song_count"),
    F.sum(F.col("plays")).alias("play_count"),
  )
  .orderBy(F.col("play_count").desc())
)
user_counts.show(1, False)
```

```
+----------------------------------------------------------+----------+----------+
|user_id                                                   |song_count|play_count|
+----------------------------------------------------------+----------+----------+
|093cb74eb3c517c5179ae24caf0ebec51b24d2a2|      195    |   13074    |
+----------------------------------------------------------+----------+----------+
```

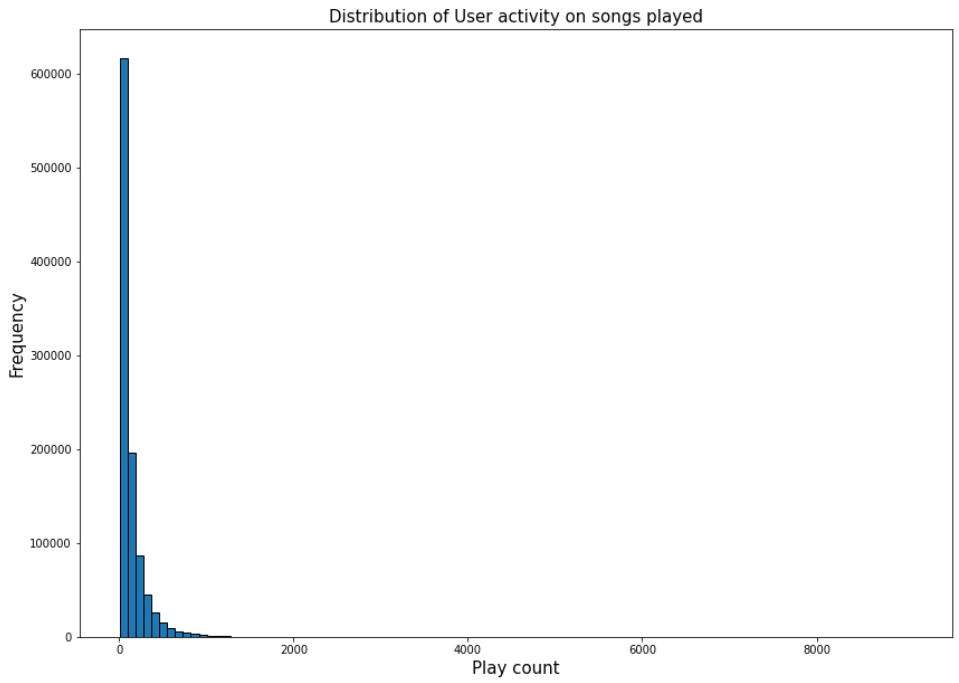

Figure 10. Distribution of user activity based on the songs played.

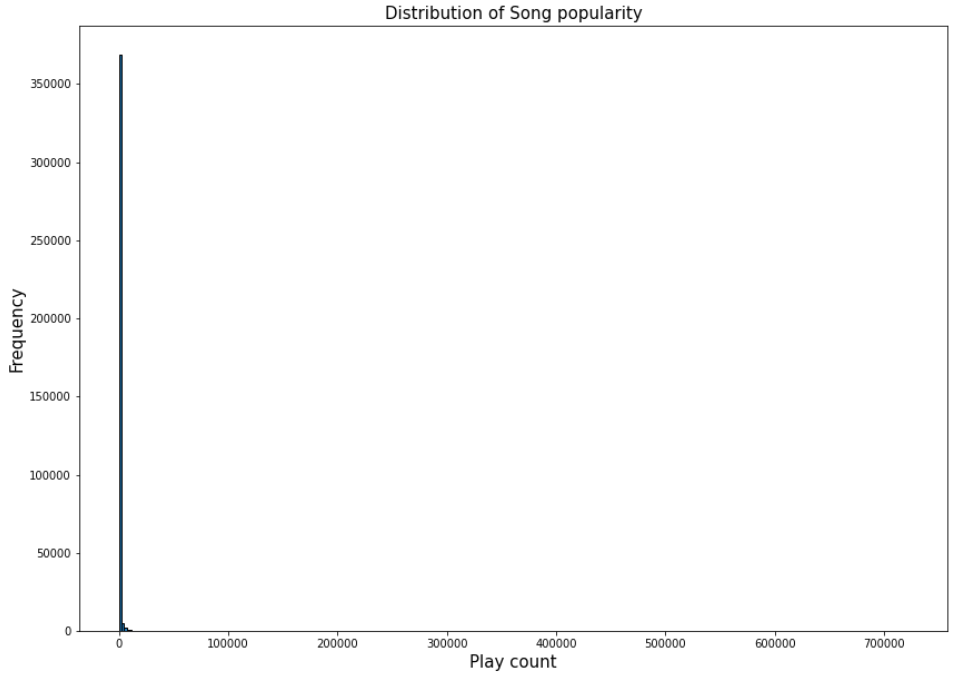

Figure 11. Distribution of song popularity.

As shown in Figures 10 and 11, the data is right-skewed; hence, they have non-Gaussian distribution. From the User activity plot the users who have played the songs only once are

many when compared to the users who have played many songs. Also from the Song popularity chart the songs that have been played once are more when compared to the song that has been played many times. Based on this distribution they are not normal and the models that are trained and tested on this data will produce biased results.

Collaborative filtering

Collaborative filtering is a technique, which is used to filter out the songs that a user might like based on the combined play history of similar users and songs. A threshold is set for the users who have played only 34 songs and songs that were played only 5 times were removed from the data set as these values will not contribute much information to the overall dataset and are not likely recommended.

```
user_song_count_threshold = 34
song_user_count_threshold = 5
```

As the used_id and song_id have characters in them they are converted to integers using StringIndexer(). The dataset is then split into training and test sets. The training set has 75% and 25% of the overall data in it. Windows partitionBy() method and randomly selecting the records from the data to split ensured that every user in the test set has some user-song play in the train set as well.

```
:
     Window
    .partitionBy("user_id")
    .orderBy("random")
 :

for k, v in counts.items():
  temp = temp.where((F.col("user_id") != k) | (F.col("row") < v * 0.75))
temp = temp.drop("id", "random", "row")
```

```
+----------------------------------------------------------------+------------------+-------+---------------+---------------+
|user_id                                                         |song_id           |plays  |user_id_encoded|song_id_encoded|
+----------------------------------------------------------------+------------------+-------+---------------+---------------+
|478f9863348edccaf57553d2507189984a6371cf|SODEOCO12A6701E922|1      |1018438.0      |147.0          |
|478f9863348edccaf57553d2507189984a6371cf|SOCPMIK12A6701E96D|1      |1018438.0      |291.0          |
|fab281059390436d2402a85a57833fe18b7ecb51|SOHYYVI12A6D4F98A6|1      |1019022.0      |9580.0         |
|21007be74bb21d508b3f1a6bfb3d3f3c59b80c73|SOAWGYF12A6310E08C|1      |1018772.0      |17781.0        |
```

ALS() method is used to train an implicit matrix factorisation model by setting the parameters implicitPrefs to True and using appropriate columns for userCol, itemCol and ratingCol.

```
als = ALS(maxIter=5, regParam=0.01, userCol="user_id_encoded", itemCol="song_id_encoded",
ratingCol="plays", implicitPrefs=True)
```

The recommendations were generated by using the top 3 users from the test set. To do this user defined functions are used.

```
recommended_songs = (
 topUsers
 .withColumn("recommended_songs", extract_songs_top_k_udf(F.col("recommendations")))
 .select("user_id_encoded", "recommended_songs")
)
recommended_songs.cache()
recommended_songs.count()
recommended_songs.show(10, 50)
```

```
# +---------------------+------------------------------------------------------------------+
# |user_id_encoded|recommended_songs                                                        |
# +---------------------+------------------------------------------------------------------+
# |               53133|                        [16, 12, 29, 20, 47, 35, 30, 5, 80]|
# |                3986|             [ 31, 6, 138, 86, 11, 141, 111, 104, 63, 16]|
# |              141443| [177, 141, 76, 76, 254, 81, 260, 38, 187, 166,197]|
# +---------------------+------------------------------------------------------------------+
```

The encoded user id and song id along with the plays are selected. Grouping by encoded user id and collecting it as a list displayed an array of encoded song id and plays the actual listed songs.

```
# +---------------------+------------------------------------------------------------------+
# |user_id_encoded| relevance                                                               |
# +---------------------+------------------------------------------------------------------+
# |                3986| [1666, 1706, 19050, 1587, 4322, 1279, 1052, 91251,|
# |               53133|   [19145, 5410, 105, 7935, 154, 6886, 12716, 6153,..|
# |              141443|[10437, 718, 2607, 12309, 336, 95146, 16559, 97755|
# +---------------------+------------------------------------------------------------------+
```

Model evaluation and metrics

The relevant songs and the recommended songs obtained from the ALS() model were merged using the inner join on encoded user id and this dataframe is converted into a rdd to evaluate the metrics.

```
combined = (
  recommended_songs.join(actualPlayed, on='user_id_encoded', how='inner')
  .rdd
  .map(lambda row: (row[1], row[2]))
)
combined.cache()
```

```
[([31, 6, 138], [[1666,1706, 19050, 1587, 4322, 1279, 1052, 91251, 4623, 5383, 13740, 1252, 76385, 329, 3617, 883, 597, 11005, 36255, 2666, 2158, 3768, 10482, 19448, 318, 3593, 7692, 223, 56, 1333, 19518, 242, 710, 259, 3255, 2163, 4206, 33949, 2011, 16372, 3101, 8206, 3003, 3444, 10457, 1453, 227, 1604, 1004, 3368, 20216, 5867, 24434, 32814, 8659, 8978, 5729, 289, 248, 1379, 1921, 2717, 19226, 20674, 326, 11104, 72630, 401, 609, 4719, 866, 858, 2473, 11456, 2031, 26948, 1825, 13366, 249, 44530, 19337, 540, 11740])]
```

The ranking metrics like Precision@k, NDCG@k and Mean Average Precision are used to compute the collaborative filtering model. The top K user recommendations are taken and compared to the set of relevant songs played by the user. The mean average precision is the score calculated by taking the mean average precision of all classes.

```
rankingMetrics = RankingMetrics(combined)
ndcgAtK = rankingMetrics.ndcgAt(10)
print(ndcgAtK)
0.029896389065203238
#print(rankingMetrics.precisionAt(10))
#0.0165226291558473
print(rankingMetrics.meanAveragePrecision)
#0.012721952570188318
```

The RankingMetrics() method is used to find precisionAt(k), ndcgAt(k) and meanAveragePrecision. The value for precision@10 is 0.0165, NDCG@10 is 0.0299 and MeanAveragePrecision (MAP) is 0.0127, which are all low values. The sparsity of the data and as these metrics are all offline with a static set of relevant songs the results are expected to be low.

Conclusions

Implemented a candidate set of models to classify music genres and a recommendation system has been developed using MSD dataset. For binary class classification, the highest test accuracy of 75% is achieved by GBT classifier with fine-tuned parameters. The Random forest method is used for multiclass classification, where the weighted F1 score for unbalanced data is 43% and the accuracy for balance data is 23%. This might be due to the Random forest predicting only those classes that have more songs. The results obtained for the collaborative filtering were also very low. Based on the results, we cannot use this recommended system for recommending songs to the users.

To improve the accuracy in binary classification or multiclass classification, we need more data to train the models. Gathering more data, especially in the genres that have low data to balance the distribution. Or we can use only some genres that are more frequently played. Using the Neural network methods can also improve the test accuracy when the data is much more balanced. Since the collaborative filtering method only extracts information from the available data, cold start problems might occur to the sparse data and lack of information that is available about the users or items. Just like the case in our data where most of the users have played the songs only once this type of data doesn't provide any information about the songs and users as well to recommend this to other user. Making recommendations to a new user when the dataset has limited data available based on users or songs, collaborative filtering both implicit or explicit cannot deal with it.

A/B testing can be an alternative method for comparing two recommendation systems in the real world. A/B testing is also known as a split test. Two different recommendation systems can be experimented to determine which of the variation can be the best system and used for recommendations in future. This can be achieved by deploying them and testing them in the real world based on the online experiences, constantly analysing and revising the systems based on analysis. Again deploying the newer version to the random users and analysing the results. A time frame can be fixed to test and it's a lot quicker, enables data-driven decisions

and easy to analyse the system as it is dealt with the real users. In the end, the results of the two systems are compared and the best one is chosen for recommending songs.

References

Machine learning applications to predict two-phase flow patterns. https://www.researchgate.net/publication/356614182_Machine_learning_applications_to_predict_two-phase_flow_patterns.

Music Reccomender System Based on Genre using Convolutional Recurrent Neural Networks https://www.sciencedirect.com/science/article/pii/S1877050919310646.

Acknowledgement

The code for this assignment is adapted from the code provided by James Williams through UC Learn.