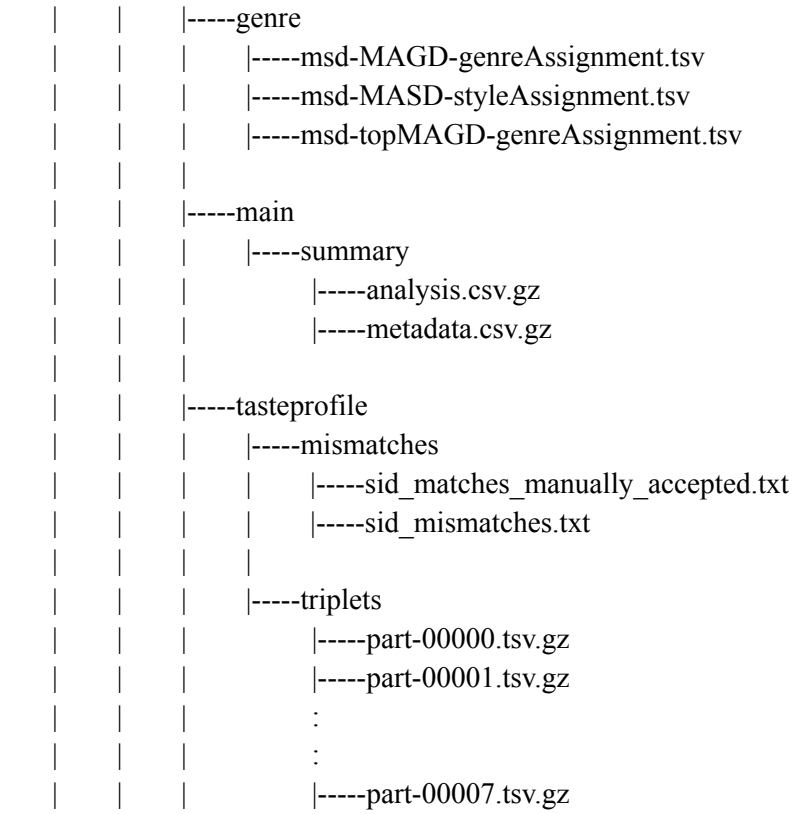
Arijit Bhattacharjee
Student ID 26649839

Data Processing

1. a) Hdfs:

```
|
|-----data
|       |-----msd
|       |       |-----audio
|       |       |       |-----attributes
|       |       |       |       |-----msd-jmir-area-of-moments-all-v1.0.attributes.csv
|       |       |       |       |-----msd-jmir-lpc-all-v1.0.attributes.csv
|       |       |       |       :
|       |       |       |       :
|       |       |       |       |-----msd-tssd-v1.0.attributes.csv
|       |       |       |
|       |       |       |
|       |       |       |-----features
|       |       |       |       |-----msd-jmir-area-of-moments-all-v1.0.csv
|       |       |       |       |       |-----part-00000.csv.gz
|       |       |       |       |       |-----part-00001.csv.gz
|       |       |       |       |       :
|       |       |       |       |       :
|       |       |       |       |       |-----part-00007.csv.gz
|       |       |       |       |
|       |       |       |       |-----msd-jmir-lpc-all-v1.0.csv
|       |       |       |       |       |-----part-00000.csv.gz
|       |       |       |       |       |-----part-00001.csv.gz
|       |       |       |       |       :
|       |       |       |       |       :
|       |       |       |       |       |-----part-00007.csv.gz
|       |       |       |       :
|       |       |       |       :
|       |       |       |       :
|       |       |       |       |-----msd-tssd-v1.0.csv
|       |       |       |               |-----part-00000.csv.gz
|       |       |       |               |-----part-00001.csv.gz
|       |       |       |               :
|       |       |       |               :
|       |       |       |               |-----part-00007.csv.gz
|       |       |       |
|       |       |       |-----statistics
|       |       |               |-----sample_properties.csv.gz
|       |       |
|       |       |
```



The directory tree represents the way the data is structured. The msd folder has four subfolders - audio, genre, main and tasteprofile.
The audio folder contains attributes, features and statistics folders. The attribute folder contains the attributes of the audio features as 13 csv files. The data types of columns/variables are strings. The features folder contains the values of 13 audio features as part files in csv.gz format. The data types of all columns/variables except the last column are numeric (loaded as string) and the last column is string. The statistics folder contains a single file in csv.gz format. The data types of columns/variables are strings and numeric (loaded as string).
The genre folder contains 3 datasets including MSD All Music Genre Dataset (MAGD) in tsv format.The data types of columns/variables are strings.
The main folder contains a subfolder summary and within that contains analysis and metadata file in csv.gz format. The data types of columns/variables are strings and numeric (loaded as string).
The tasteprofile contains mismatches and triplets folder. The mismatches folder contains 2 text files, one mismatches file and another manually accepted matches file. The triplets folder contains user song play counts in 8 separate part files in tsv.gz format. The data types of columns/variables are strings and integers.

The total size of all the data is around 13GB and the audio folder constitutes ~95% of the total size.
12.3 G 98.1 G /data/msd/audio

30.1 M 241.0 M /data/msd/genre
174.4 M 1.4 G /data/msd/main
490.4 M 3.8 G /data/msd/tasteprofile
In the audio folder, the feature dataset contributes the biggest chunk ~99%
103.0 K 824.3 K /data/msd/audio/attributes
12.2 G 97.8 G /data/msd/audio/features
40.3 M 322.1 M /data/msd/audio/statistics

b) The ideal number of partitions in our case would be 32 (partitions = 4 * 4 * 2). When we look across the dataset, the number of partitions are

metadata num partitions = 1
analysis num partitions = 1
mismatches num partitions = 1
matches_manually_accepted num partitions = 1
triplets num partitions = 8
audio_attributes num partitions = 7
audio_features num partitions = 51 (num partitions = 8 if we take only one dataset)
audio_statistics num partitions = 1
genre_assignment num partitions = 3
genre_assignment_top num partitions = 3
style_assignment num partitions = 3

We observe that the raw data has fewer partitions than the ideal number of partitions and the cluster is underutilized, repartitioning might help.
We also observed that bulk of the data is in the gzip compression format (audio features folder, triplets folder) which is not splittable and each file must be loaded in entirety by a single executor resulting in one partition per file. Considering this, repartitioning would not be of much help in our case.
So we will be proceeding without partitioning for now.

c) Dataset-wise number of rows
metadata count = 1000000
analysis count = 1000000
mismatches count = 19094
matches_manually_accepted count = 489
triplets count = 48373586
audio_attributes count = 3929
audio_features count = 12927867
- msd-jmir-area-of-moments-all-v1.0 = 994623
- msd-jmir-lpc-all-v1.0 = 994623
- msd-jmir-methods-of-moments-all-v1.0 = 994623
- msd-jmir-mfcc-all-v1.0 = 994623
- msd-jmir-spectral-all-all-v1.0 = 994623
- msd-jmir-spectral-derivatives-all-all-v1.0 = 994623

msd-marsyas-timbral-v1.0 = 995001
msd-mvd-v1.0 = 994188
msd-rh-v1.0 = 994188
msd-rp-v1.0 = 994188
msd-ssd-v1.0 = 994188
msd-trh-v1.0 = 994188
msd-tssd-v1.0 = 994188
audio_statistics count = 992865
genre_assignment count = 422714
genre_assignment_top count = 406427
style_assignment count = 273936

metadata.select("song_id").distinct().count()
Total number of unique songs is 998963
The metadata/analysis rows count is slightly higher than the number of unique songs, whereas the number of rows in individual features dataset and statistics is slightly lower than the number of unique songs. The mismatches row count is less than 2% of the number of unique songs. The genre assignment row count is ~42% of the number of unique songs. The triplet row count is almost 50 times more than the number of unique songs.

2. a) Left anti-join to remove matches manually accepted from the mismatches. Then use the result from the previous join i.e mismatches not accepted and use left anti-join to remove mismatches from triplets.
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")

b) We put all the filenames of the audio features in a list. Then we create an empty list ‘dataset’. We iterate over the filenames and append name, path and schema to the ‘dataset’. While creating the schema, we factored in that the data type of all the columns except the last column is numeric. The last column is an ID which is of string data type. All the files of the audio features follow the same pattern.

Now we can call and load any file along with it’s path and schema by calling the required index of ‘dataset’. After loading preprocessing was done to remove special characters from the ID column.

Audio Similarity

1. a) Dataset loaded - msd-jmir-area-of-moments-all-v1.0

   Descriptive statistics for each feature column is as follows:

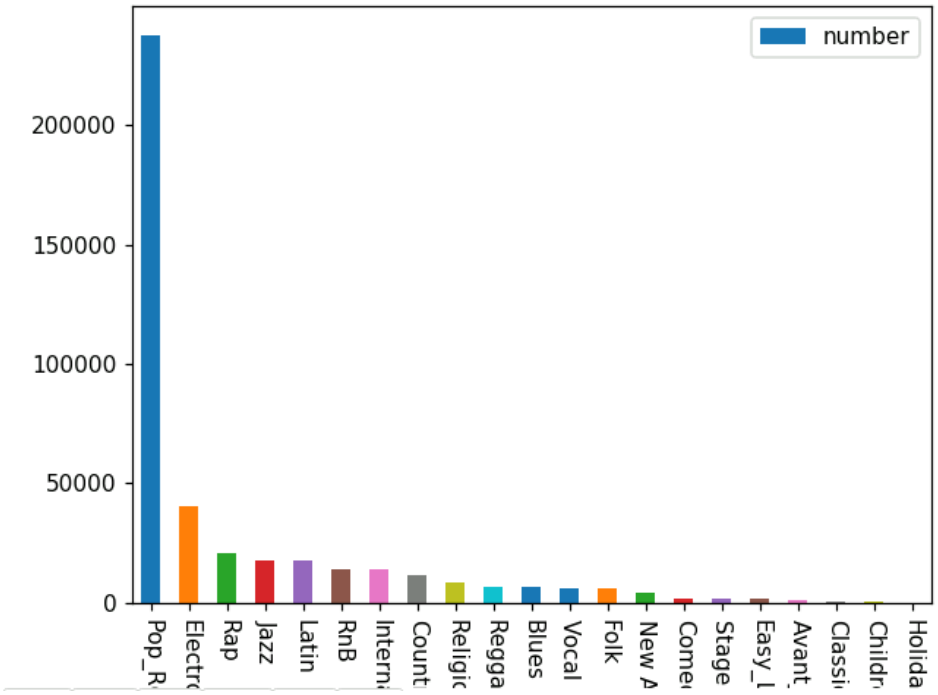
index	count	mean	stddev	min	max
F001	994623	1.2288977645373984	0.5282627649456019	0.0	9.346
F002	994623	5500.463309452325	2366.129254432958	0.0	46860.0
F003	994604	33817.8867026857	18228.646843590075	0.0	699400.0
F004	994604	1.276699261056731E8	2.396385081368778E8	0.0	7.864E9
F005	994604	7.834826067815083E8	1.5825917070672803E9	0.0	8.124E10
F006	994604	5.254830208141142E9	1.2189329483371695E10	0.0	1.453E12
F007	994604	7.770600408917489E12	5.691673392602E13	0.0	2.43E15
F008	994604	7.036546767963497E9	1.4246254033845514E10	0.0	7.46E11
F009	994604	4.722026947033176E10	1.0978759554853354E11	0.0	1.31E13
F010	994604	2.3239346743409715E15	2.456502306380948E16	0.0	5.817E18
F011	994604	3.516756849872974	1.8600935007021255	0.0	26.52
F012	994604	9476.016837603707	4088.52389790846	0.0	81350.0
F013	994604	58331.70233463418	31372.66477302157	0.0	1003000.0
F014	994604	-1.422983386983546E8	2.6710276301530725E8	-8.802E9	0.0
F015	994604	-8.72570569837153E8	1.7589081294252565E9	-9.005E10	0.0
F016	994604	-5.857219932855641E9	1.3586390808488363E10	-1.495E12	0.0
F017	994604	6.835742859356949E12	5.008474325223765E13	0.0	2.144E15
F018	994604	7.828432077183538E9	1.5832310311166527E10	0.0	8.335E11
F019	994604	5.259012157471162E10	1.2236963207338905E11	0.0	1.347E13
F020	994604	2.0437364618507848E15	2.1631793599553572E16	0.0	4.958E18

There are quite a few features that are strongly correlated (threshold considered at 0.95). For example
F002 and F012
F003 and F012
F004 and F007 and F017
F005 and F006 and F008 and F009 and F018 and F019

The below table shows all correlations (F001 is represented as 0, F002 as 1 and so on)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1.00	-0.02	0.23	0.01	0.08	0.11	0.02	0.08	0.10	0.05	0.72	-0.02	0.23	-0.01	-0.08	-0.10	0.02	0.08	0.10	0.04
-0.02	1.00	0.79	0.85	0.80	0.70	0.69	0.80	0.70	0.50	0.03	1.00	0.78	-0.85	-0.79	-0.70	0.69	0.79	0.70	0.50
0.23	0.79	1.00	0.68	0.78	0.79	0.56	0.78	0.79	0.55	0.33	0.79	0.99	-0.68	-0.78	-0.79	0.56	0.78	0.79	0.55
0.01	0.85	0.68	1.00	0.95	0.84	0.96	0.95	0.84	0.71	0.01	0.85	0.68	-1.00	-0.94	-0.83	0.96	0.94	0.83	0.70
0.08	0.80	0.78	0.95	1.00	0.96	0.91	1.00	0.96	0.85	0.09	0.79	0.78	-0.95	-1.00	-0.96	0.91	1.00	0.96	0.85
0.11	0.70	0.79	0.84	0.96	1.00	0.81	0.97	1.00	0.92	0.13	0.70	0.79	-0.84	-0.97	-1.00	0.81	0.97	1.00	0.92
0.02	0.69	0.56	0.96	0.91	0.81	1.00	0.91	0.81	0.74	0.00	0.69	0.56	-0.96	-0.91	-0.81	1.00	0.91	0.81	0.74
0.08	0.80	0.78	0.95	1.00	0.97	0.91	1.00	0.96	0.85	0.09	0.79	0.78	-0.95	-1.00	-0.96	0.91	1.00	0.96	0.85
0.10	0.70	0.79	0.84	0.96	1.00	0.81	0.96	1.00	0.92	0.13	0.70	0.79	-0.84	-0.97	-1.00	0.81	0.97	1.00	0.92
0.05	0.50	0.55	0.71	0.85	0.92	0.74	0.85	0.92	1.00	0.04	0.50	0.55	-0.71	-0.85	-0.92	0.74	0.85	0.92	1.00
0.72	0.03	0.33	0.01	0.09	0.13	0.00	0.09	0.13	0.04	1.00	0.03	0.33	-0.01	-0.09	-0.13	0.00	0.09	0.13	0.04
-0.02	1.00	0.79	0.85	0.79	0.70	0.69	0.79	0.70	0.50	0.03	1.00	0.78	-0.85	-0.79	-0.69	0.69	0.79	0.69	0.50
0.23	0.78	0.99	0.68	0.78	0.79	0.56	0.78	0.79	0.55	0.33	0.78	1.00	-0.68	-0.78	-0.79	0.56	0.78	0.79	0.55
-0.01	-0.85	-0.68	-1.00	-0.95	-0.84	-0.96	-0.95	-0.84	-0.71	-0.01	-0.85	-0.68	1.00	0.94	0.83	-0.96	-0.94	-0.83	-0.70
-0.08	-0.79	-0.78	-0.94	-1.00	-0.97	-0.91	-1.00	-0.97	-0.85	-0.09	-0.79	-0.78	0.94	1.00	0.96	-0.91	-1.00	-0.96	-0.85
-0.10	-0.70	-0.79	-0.83	-0.96	-1.00	-0.81	-0.96	-1.00	-0.92	-0.13	-0.69	-0.79	0.83	0.96	1.00	-0.81	-0.96	-1.00	-0.92
0.02	0.69	0.56	0.96	0.91	0.81	1.00	0.91	0.81	0.74	0.00	0.69	0.56	-0.96	-0.91	-0.81	1.00	0.91	0.81	0.74
0.08	0.79	0.78	0.94	1.00	0.97	0.91	1.00	0.97	0.85	0.09	0.79	0.78	-0.94	-1.00	-0.96	0.91	1.00	0.96	0.85
0.10	0.70	0.79	0.83	0.96	1.00	0.81	0.96	1.00	0.92	0.13	0.69	0.79	-0.83	-0.96	-1.00	0.81	0.96	1.00	0.92
0.04	0.50	0.55	0.70	0.85	0.92	0.74	0.85	0.92	1.00	0.04	0.50	0.55	-0.70	-0.85	-0.92	0.74	0.85	0.92	1.00

b) Matched number of songs 420604



c) Inner join genre dataset(genre_assignment) on the selected audio feature dataset(data)
songs_matched = data.join(genre_assignment, on="ID", how="inner")

Get the distribution of the genre
```
distribution = (
 songs_matched
 .groupBy(["genre"])
 .agg(
  F.countDistinct(F.col("ID")).cast(IntegerType()).alias("number"),
 )
  .sort(F.col("number").desc())
)
```

2. a) <u>Logistic regression</u>
   Logistic regression is a supervised learning algorithm that uses sigmoid function at the core of the method for binary classification. It can predict binary or multi class outcomes. It was chosen because it can be used as a good baseline to compare with the performance of other more complex algorithms. Also, it is a simple, very efficient method and doesn't require a lot of computational resources (but not very powerful) and performs well when the dataset is linearly separable. It doesn't handle multicollinearity among the independent variables. Output always lies between 0 and 1,

   Explainability - simple
   Interpretability - easy to interpret, can interpret model coefficients as indicators of feature importance
   Predictive accuracy - good
   Training speed - efficient to train/fast training
   Hyperparameter tuning - Limited. E.g. Solver, Penalty (L1, L2, elasticnet, none), regularization strength.(smaller values specify stronger regularization)
   Dimensionality - tend to overfit data with high number of predictors
   Issue with scaling - needs to be normalized/ performance is affected

   Preprocessing steps :
   Remove correlated inputs - done
   Gaussian distribution - can be explored
   Remove outliers - explored but not removed due to potential data loss
   Normalize - tried but throws a lot of NaN values while transforming the given dataset (RuntimeWarning: invalid value encountered in greater), hence skipped

   <u>Naive bayes</u>
   Naïve Bayes classifiers are a collection of algorithms based on Bayes' Theorem.
   Classifier - multinomial, gaussian, bernoulli.

All naïve Bayes classifiers assume that the value of a particular feature is independent of the value of the other features, given the class variable. Each feature makes an independent and equal contribution to the outcome
When assumption of independence holds, a Naive Bayes classifier performs better compared to other models like logistic regression and we need less training data.
It performs well in case of categorical input variables compared to numerical variables. For numerical variables, normal distribution is assumed, which is a strong assumption.
If a categorical variable has a category in the test data set, which was not observed in the training data, then the model will assign a zero probability and will be unable to make a prediction.

Explainability - simple
Interpretability - yes because of independence assumption
Predictive accuracy - good at prediction but bad estimator of probabilities
Training speed - fast, require small amount of training data
Hyperparameter tuning - limited e.g alpha, fit_prior (so preprocessing and feature selection is important)
Dimensionality - works well with large no of predictors, each distribution can be independently estimated as a one dimensional distribution
Issue with scaling - no

Preprocessing steps :
Normalize - not necessary
Remove correlation- done as the highly correlated features are voted twice in the model and it can lead to over inflating importance.
Transform if the data is not normally distributed - can be explored

GBT

GBT is an ensemble of decision trees. It is a generalised algorithm which works for any differentiable loss function. It relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. Can handle missing data. Sensitive to outliers. Tend to overfit if the number of trees is large.
Explainability - black box
Interpretability - complex black box
Predictive accuracy - high
Training speed - takes time to train
Hyperparameter tuning - Number of trees, learning rate, max depth. There is a trade-off between learning rate and number of trees.Maximum depth of each estimator limits the number of nodes of the decision trees
Dimensionality - can handle high dimensional data
Issue with scaling - not affected by scaling

Preprocessing steps:
Remove correlation - not necessary
Normalize - not necessary

Remove outliers - explored but not removed due to potential data loss

b) Conversion of genre column into binary column was done using withColumn() and F.lit().

```
songs = songs_matched.withColumn("Class",
                F.when((songs_matched.genre == "Electronic"), F.lit(1))
                .otherwise(F.lit(0)))
```

The class is imbalanced

Class	Count
0	379942
1	40662

c) Test set must have true class balance to evaluate metrics accurately. Exact stratification using Window was used to split the dataset into training and test set as random split won't guarantee the test class balance.Window function is computationally expensive but is a better option. The partition is done by class. Then partitions are ordered by rand (random order) and row number is generated. Then the split is performed where row numbers are less than or equal to proportion * class size is one split and row numbers > proportion * class size is another split. The same process is repeated for all classes. We get the training set by row number <= proportion * class size and then get the test by left anti join with input. Training and test data cached for subsequent use.

Observation weighting is preferred as a resampling technique because it doesn't create any artificial data nor does it throw out any data that could be potentially useful. Moreover, all the algorithms chosen above support observation weighting. A ratio of 1:9 was tried out.

```
training_weighted = training.withColumn("weight", F.when(F.col("label") == 0,
1.0).when(F.col("label") == 1, 9.0).otherwise(1.0))
```

d) # Logistic Regression

```
lr = LogisticRegression(featuresCol='features', labelCol='label', weightCol="weight")
lr_model = lr.fit(training_weighted)
```

#Naive Bayes

```
nb = NaiveBayes(smoothing=1.0, \
        modelType="gaussian", \
        featuresCol="features", \
        labelCol='label', weightCol="weight")
nb_model = nb.fit(training_weighted)
```

#Gradient Boosted Tree

```
gbt = GBTClassifier(labelCol='label', \
            featuresCol="features", weightCol="weight")
gbt_model = gbt.fit(training_weighted)
```

e)

		Logistic Regression	**Naive Bayes**	**GBT**
AUROC		**0.64**	**0.58**	**0.71**
Threshold = 0.5	Precision	0.14	0.26	0.16
	Recall	0.62	0.01	0.65
	Accuracy	0.59	0.9	0.65
Threshold = 0.2	Precision	0.1	0.26	0.11
	Recall	0.99	0.01	0.97
	Accuracy	0.1	0.9	0.22
Threshold = 0.6	Precision	0.19	0.26	0.22
	Recall	0.2	0.01	0.38
	Accuracy	0.85	0.9	0.82

f) The GBT classifier has the highest auroc amongst the models. Auroc tells us how much the model is capable of distinguishing between classes. Based on that, GBT would be the best performing model. Accuracy can be a misleading metric because of class imbalance, so we won't consider it.
There is a tradeoff between precision and recall while finding the optimal threshold value. The definition of optimal will change depending on the context.

The Logistic Regression didn't perform too well. One of the reasons for this could be that the data was not normalized. Better preprocessing steps and feature engineering might improve the model.
The Naive Bayes model didn't perform well. This could be attributed to the presence of numerical variables. Better preprocessing and feature selection might improve the model.
The GBT model performed better. Auto feature selection could have happened but it is difficult to interpret. Removal of outliers could improve the model. For the given dataset, no features removed performed slightly better than correlated features removed..

Because of class imbalance, training models to distinguish between positive and negative classes becomes harder thereby reducing the performance of these algorithms. Imbalance makes obtaining good metrics hard.

3. a) For Logistic Regression, the default value of the threshold hyperparameter is 0.5, this can be tweaked. Hyperparameters - elasticNetParam and regParam can also be changed (default value 0.0 used for both).
   For Naive Bayes, the hyperparameters we can try changing are - smoothing: float, modelType: str
   For GBT, the default values for the following hyperparameters are numIterations: int = 100, learningRate: float = 0.1, maxDepth: int = 3, maxBins: int = 32. We can try changing these values.

   b) We will use cross validation to tune hyperparameters of our GBT model. We will do a random grid search over possible choices of hyperparameters.
   We will do 5-fold cross validation.
   As each new parameter adds another dimension to the grid and the size of the search space is exponentially larger. We will use 8 parameter combinations, thereby training the model 40 times, which is time taking (we can reduce the grid size or increase computation resources if necessary).

   ```
   gbparamGrid = (ParamGridBuilder()
           .addGrid(gbt.maxDepth, [5, 10])
           .addGrid(gbt.maxBins, [20, 40])
           .addGrid(gbt.maxIter, [10, 20])
           .build())
   gbevaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")

   gbcv = CrossValidator(estimator = gbt,
                   estimatorParamMaps = gbparamGrid,
                   evaluator = gbevaluator,
                   numFolds = 5)
   ```

   The model performs slightly better (threshold = 0.5). Auroc has improved to 0.72 from 0.71. The precision has changed to 0.18 from 0.16 and recall has changed to 0.61 from 0.65.

4. a) For multi class classification we will consider Logistic Regression. Softmax function instead of sigmoid function is at the core of multinomial logistic regression. The softmax function squashes all values to the range [0,1] and the sum of the elements is 1.
   In multinomial logistic regression, the algorithm produces K sets of coefficients, or a matrix of dimension $K \times J$ where K is the number of outcome classes and J is the number of features.
   The weighted negative log-likelihood is minimized by using a multinomial response model, with elastic-net penalty to control for overfitting.
   The issue with the softmax function is that it blows small differences out of proportion which makes our classifier biased towards a particular class which is not desired.

   b) We converted the genre column into an integer index using StringIndexer.
   ```
   indexer = StringIndexer(inputCol="genre", outputCol="class").fit(songs_matched)
   indexed_songs = indexer.transform(songs_matched)
   ```

```
+--------------+-----+------+
|         genre|class| count|
+--------------+-----+------+
|      Pop_Rock|  0.0|237641|
|    Electronic|  1.0| 40662|
|           Rap|  2.0| 20899|
|          Jazz|  3.0| 17775|
|         Latin|  4.0| 17504|
|           RnB|  5.0| 14314|
| International|  6.0| 14194|
|       Country|  7.0| 11691|
|     Religious|  8.0|  8779|
|        Reggae|  9.0|  6928|
|         Blues| 10.0|  6801|
|         Vocal| 11.0|  6182|
|          Folk| 12.0|  5789|
|       New Age| 13.0|  4000|
| Comedy_Spoken| 14.0|  2067|
|        Stage | 15.0|  1613|
|Easy_Listening| 16.0|  1535|
|   Avant_Garde| 17.0|  1012|
|     Classical| 18.0|   555|
|      Children| 19.0|   463|
|       Holiday| 20.0|   200|
+--------------+-----+------+
```

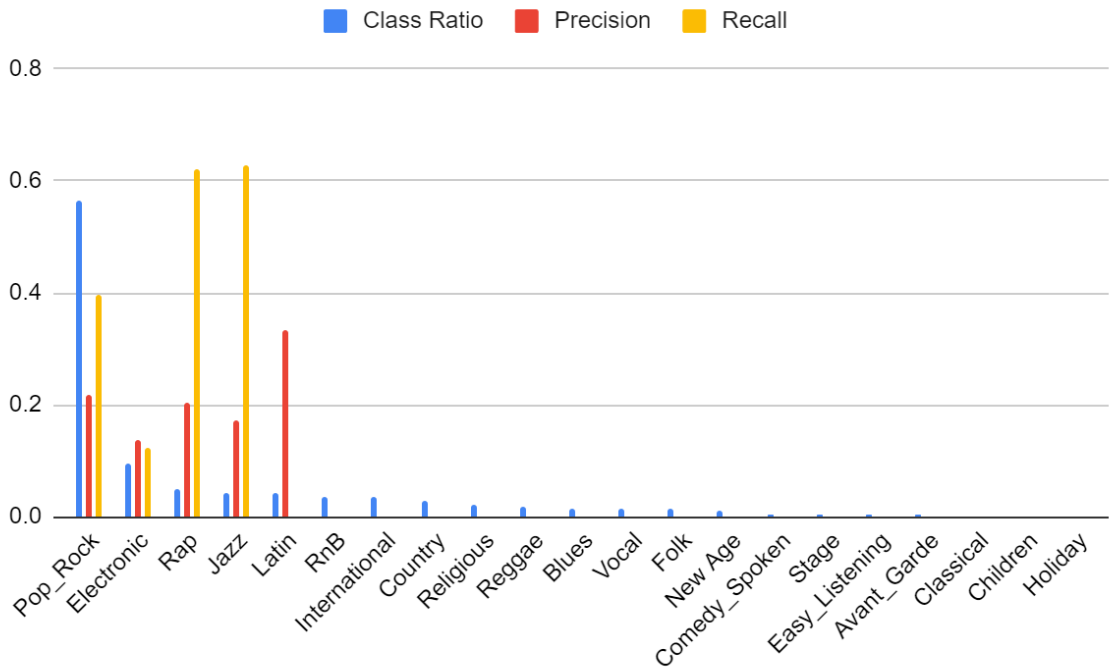
c) Exact stratification using Window was used to split the dataset into training and test set. Downsampling of Pop_rock and Electronic genre on the training set was done to handle class imbalance to an extent.
Training without downsampling was also tried but the model was significantly poor.

lr = LogisticRegression(maxIter=20, regParam=0.3, elasticNetParam=0)
lrModel = lr.fit(training_downsampled)

Genre	Label/Class	Class Ratio	Precision	Recall
Pop_Rock	0	0.565	0.218	0.398
Electronic	1	0.097	0.138	0.123
Rap	2	0.050	0.205	0.621
Jazz	3	0.042	0.173	0.628
Latin	4	0.042	0.333	0

RnB	5	0.034	0	0
International	6	0.034	0	0
Country	7	0.028	0	0
Religious	8	0.021	0	0
Reggae	9	0.016	0	0
Blues	10	0.016	0	0
Vocal	11	0.015	0	0
Folk	12	0.014	0	0
New Age	13	0.010	0	0
Comedy_Spoken	14	0.005	0	0
Stage	15	0.004	0	0
Easy_Listening	16	0.004	0	0
Avant_Garde	17	0.002	0	0
Classical	18	0.001	0	0
Children	19	0.001	0	0
Holiday	20	0.000	0	0

Inclusion of multiple genres has reduced the performance of the model. Other than the top 5 classes by proportion, all other classes are difficult to predict.



Song recommendations

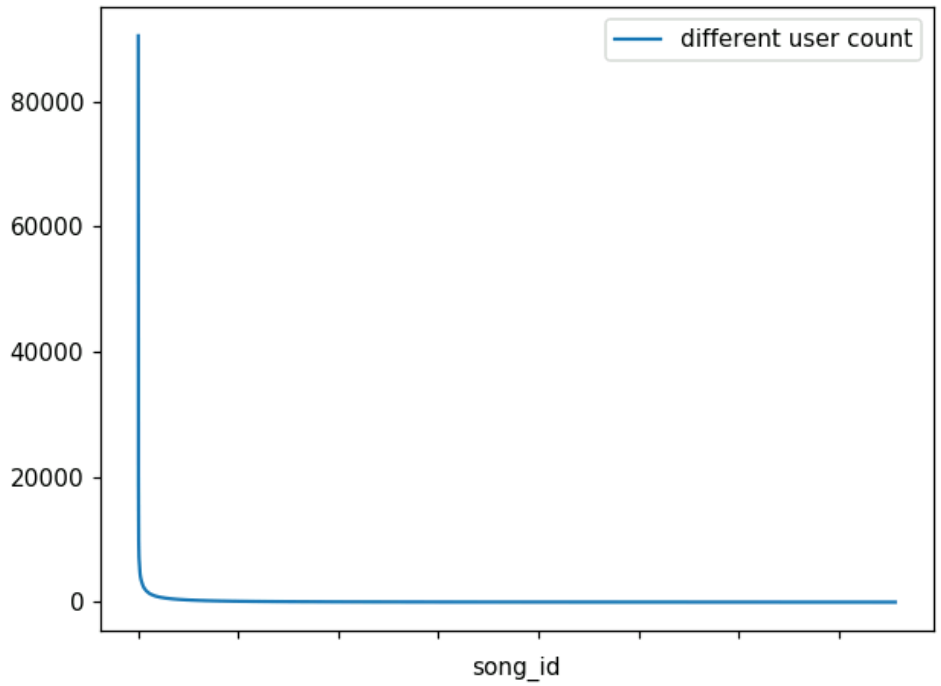
1. For this section we will consider the triplets dataset that was not mismatched.
   a) No of unique songs = 378310
   No of unique users = 1019318

   b) Most active user is defined as someone who has played most number of different songs (total no of play was also considered but as we need to recommend songs down the line, thought it would be more appropriate to go ahead with count of different songs played)

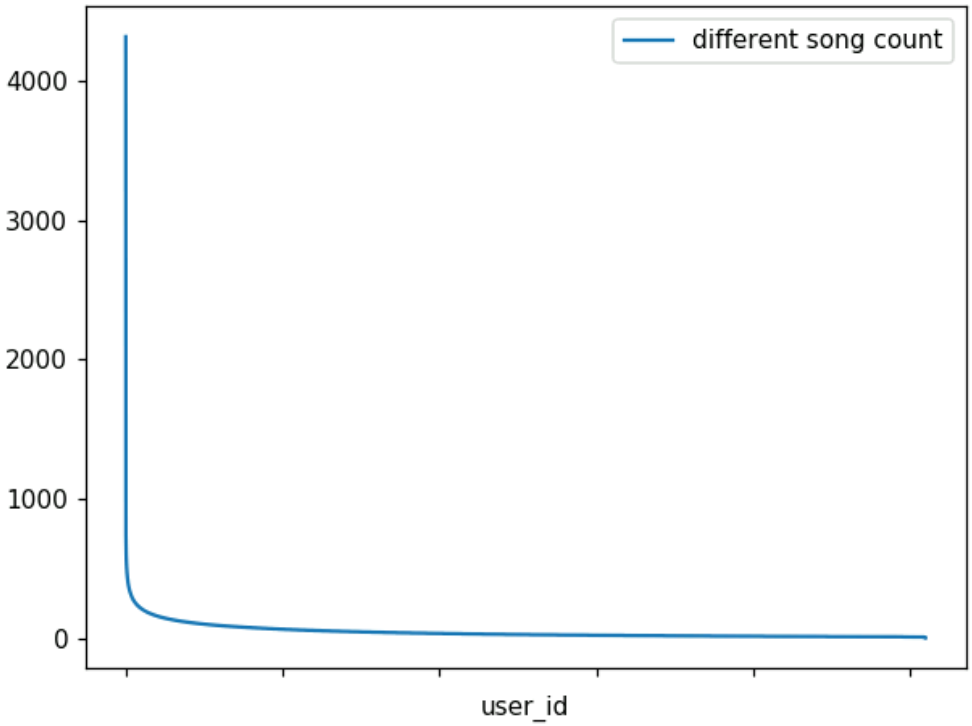
```
temp_user = (
  triplets_not_mismatched
  .groupBy(["user_id"])
  .agg(
    F.countDistinct(F.col("song_id")).cast(IntegerType()).alias("different song count"),
    F.sum(F.col("plays")).cast(IntegerType()).alias("total play count"),
  )
    .sort(F.col("different song count").desc())
)
```

   The number of different songs played by the most active user is 4316.
   It is 1.14% of the total number of unique songs in the dataset.

c) Distribution of song popularity



Distribution of user activity



The shape of the distribution is power law distribution.

d) We removed the songs played less than or equal to 50 different users. Doing so ensures we retain 1/4th of our unique songs in the dataset.
songs_retained = temp_song.select('song_id').where(F.col('different user count')>50)

We also removed users who have played less than or equal to 50 different songs. Here also we retain almost 1/4th of our initial unique users.
users_retained = temp_user.select('user_id').where(F.col('different song count')>50)

1/4th of the dataset is faster to train and we retain most of the popular songs and active users.

Then to create the clean dataset, we do an inner join of retained songs and triplets not mismatched. This is followed by another inner join of retained users with the dataset from the previous step.
songs_high = triplets_not_mismatched.join(songs_retained, on="song_id", how="inner")
songs_users_high = songs_high.join(users_retained, on="user_id", how="inner")

We also did one additional step of converting user_id and song_id from string values to corresponding numerical values using StringIndexer (as ALS doesn't accept user and item columns that are not numeric during training).
indexer = StringIndexer(inputCol="user_id", outputCol="user").fit(songs_users_high)
clean_data = indexer.transform(songs_users_high)
indexer = StringIndexer(inputCol="song_id", outputCol="song").fit(clean_data)
clean_data = indexer.transform(clean_data)

e) We do a random split of the data (30% test) and cache the training and test data set. Considering the amount of data we have this looks a good option (other methods like using window function to stratify based on song id would be slow).
training, test = clean_data.randomSplit([0.7, 0.3])

Next we check for user_id match in training and test using inner join and select the distinct matching ids
matching = training.join(test, on="user_id", how="inner")
ids = matching.select("user_id").distinct()

Then we will do an inner join of these ids on test to retain the only users in test that have some user-song plays in the training set as well.
test = test.join(ids, on="user_id", how="inner")

This is required because if there are users in the test set that we have never seen in training, that user will not exist in the P implicit preference matrix and therefore won't have recommendations.

2. a) *als = ALS(maxIter=5, regParam=0.01, userCol="user", itemCol="song", ratingCol="plays", implicitPrefs=True)*
alsModel = als.fit(training)

b) Recommendation is generated
k = 5
recommendations = alsModel.recommendForAllUsers(k)

Then, we collect and sort relevant items in order of descending relevance (plays) from the test data. After that we merge recommendations and relevant items so they can be compared.

user	recommendations	relevant
6	[5.0, 17.0, 9.0, 12.0, 21.0]	[1091.0, 1002.0, 3696.0, 1851.0, 280.0, 20969....
13	[6.0, 2.0, 39.0, 3.0, 100.0]	[513.0, 7244.0, 12008.0, 9036.0, 12500.0, 1249...
99	[0.0, 42.0, 13.0, 28.0, 124.0]	[3663.0, 505.0, 23758.0, 7139.0, 835.0, 787.0,...
106	[123.0, 61.0, 11.0, 187.0, 44.0]	[521.0, 9747.0, 24964.0, 12047.0, 34569.0, 139...
122	[149.0, 11.0, 373.0, 191.0, 198.0]	[80714.0, 35256.0, 33019.0, 78583.0, 44570.0, ...
194	[185.0, 11.0, 208.0, 70.0, 221.0]	[33936.0, 27534.0, 71619.0, 69563.0, 914.0, 87...
196	[1.0, 4.0, 13.0, 0.0, 7.0]	[4402.0, 21.0, 72165.0, 52032.0, 45033.0, 3648...
209	[70.0, 185.0, 208.0, 221.0, 225.0]	[21040.0, 20178.0, 35070.0, 18232.0, 14610.0, ...
215	[11.0, 28.0, 8.0, 39.0, 6.0]	[285.0, 9234.0, 1764.0, 15001.0, 685.0, 11609....
350	[111.0, 169.0, 189.0, 49.0, 180.0]	[264.0, 2038.0, 883.0, 711.0, 24110.0, 377.0, ...
355	[201.0, 296.0, 37.0, 13.0, 126.0]	[60190.0, 3606.0, 4558.0, 24566.0, 38513.0, 25...
364	[44.0, 185.0, 208.0, 70.0, 11.0]	[4301.0, 57394.0, 25063.0, 14540.0, 10592.0, 9...
365	[11.0, 123.0, 61.0, 0.0, 187.0]	[4394.0, 2762.0, 3339.0, 2011.0, 20760.0, 8624...
369	[187.0, 296.0, 201.0, 13.0, 198.0]	[5348.0, 16029.0, 7860.0, 49022.0, 32854.0, 23...
411	[13.0, 12.0, 9.0, 20.0, 7.0]	[44973.0, 23246.0, 68895.0, 2392.0, 27134.0, 6...

There doesn't seem to be an exact song match for each user. We will subjectively compare the effectiveness of the model by comparing the genre of the recommended and relevant songs for few users.

To achieve this we need song genre mapping. We will left join clean data and metadata to get track_7digitalid, using which we can get track_id by joining audio_statistics on clean data. Furthermore, track_id and genre mapping is in the genre_assignment. Joining on track id gives us song genre mapping on the clean data.

The song genre mapping for a couple of users are as follows-

User 6

Recommended song		Relevant song	
5	Pop_Rock	1002	Pop_Rock
17	NA	3696	Pop_Rock
9	Pop_Rock	280	Pop_Rock
12	Pop_Rock	20969	Pop_Rock
21	Pop_Rock	556	Pop_Rock

We can observe that Pop_Rock is present in both relevant and recommended items for this user.

User 106

Recommended song		Relevant song	
123	Pop_Rock	521	New_Age
61	NA	9747	Pop_Rock
11	Pop_Rock	24964	Pop_Rock
187	Pop_Rock	7809	Pop_Rock
44	NA	5813	Electronic

We can observe that Pop_Rock is present in both relevant and recommended items. Although New_Age and Electronic are present only in the relevant items but not recommended for this user. This could be because Pop_rock is higher in proportion, so other genres haven't come up in top 5 recommendations.

Genre recommendation seems quite effective for this model but the sample size is too small to be certain about it.

c) Metrics for implicit feedback

Precision @ 10: 0.03365
MAP @ 10: 0.01873
NDCG @ 10 0.04514

Relevance values themselves have no meaning, so we look at what was ranked and what is actually relevant. We first find (user, [recommended items]) and (user, [relevant items)] for each user and then compare [recommended items] and [relevant items].

Precision@K, MAP@K and NDCG@K are offline performance metrics intended to be evaluated based on measured, static, historical data only. These can be based on known interaction in a test dataset of historical data.
Precision@K is simple. Non relevant items are penalized and relevant items are rewarded. There is no penalty for position/rank.
MAP@K and NDCG@K adjust the reward/penalty per recommended item based on their position/rank.
MAP@K - There is an increasing cumulative penalty on how many are relevant with increasing denominator. There is diminishing reward with increasing position. The penalty is pretty hard.
NDCG@K extends the concept of "position matters", but it introduces possibly non binary relevance scores which we add up over the recommended items with a log penalty for rank and normalized by IDCG to get a metric between 0 and 1. The log scale softens the penalty for rank.

If we pack up these metrics into end to end formulas with consistent notation, we can compare them easily and understand how the number of users, number of recommended items and the set of relevant items for each user and binary relevance will affect the value of the metrics.

Offline metrics don't perform well when we have a sparse set of observed interactions between users and items. It also doesn't perform well when we split off only a small percentage as our test set.

A/B test can be used to compare two recommendation systems in the real world.

If we can measure future user-song plays based on our recommendations, then business metrics that can be computed based on user interaction like user churn, click through rate, revenue based on user interactions e.g ad clicks, higher subscription etc. could be useful.

References

1. https://spark.apache.org/docs/
2. https://www.sparkitecture.io/machine-learning/classification/gradient-boosted-trees
3. https://cprosenjit.medium.com/9-classification-methods-from-spark-mllib-we-should-know-c41f555c0425
4. https://medium.com/@jjw92abhi/is-logistic-regression-a-good-multi-class-classifier-ad20fecf1309#:~:text=Multinomial%20logistic%20regression%20is%20a,the%20cross%20entropy%20loss%20function.
5. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/