Assignment Report

Assignment No : 02
Scalable Data Science : DATA 420 19S1
University of Canterbury
College of Engineering

Author :Thiwanka Jayasiri
Student ID : 61623462
Email : gtj13@uclive.ac.nz

Million Song Data Set Analysis using Spark

The Million Song Dataset is a freely-available collection of audio features and metadata for a million contemporary popular music tracks.

Its purposes are:

- To encourage research on algorithms that scale to commercial sizes
- To provide a reference dataset for evaluating research
- As a shortcut alternative to creating a large dataset with APIs (e.g. The Echo Nest's)
- To help new researchers get started in the MIR field

The main dataset contains the song ID, the track ID, the artist ID, and 51 other fields, such as the year, title, artist tags, and various audio properties such as loudness, beat, tempo, and time signature. Note that track ID and song ID are not the same concept - the track ID corresponds to a particular recording of a song, and there may be multiple (almost identical) tracks for the same song. Tracks are the fundamental identifier, and are matched to songs. Songs are then matched to artists as well.

In this report we'll be mainly focusing on the following areas,

1) Data Processing
   i. Exploring data files
   ii. Preparing schemas
   iii. Perform JOIN operations to combine two or more data tables.

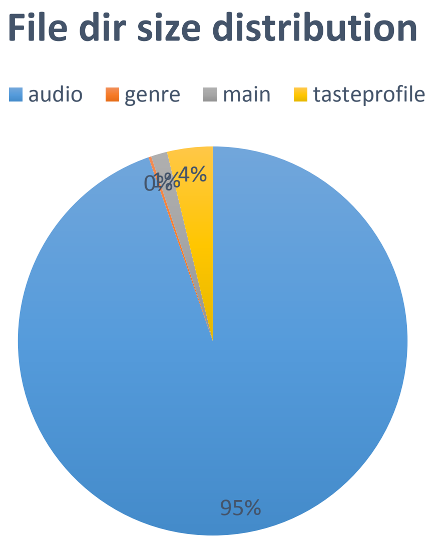
2) Audio Similarities
   i. Summaries data
   ii. Classification methods and summary
      1. Logistics Regression
      2. Random Forest
      3. Decisions Tree
   iii. Multiclass classification
3) Song Recommendations
   i. EDA
   ii. Spark.ml operations
      1. ALS
      2. Metrics
   iii. Model generated recommendations

Data Processing

'The MSD dataset consists of 4 sub-directories; *'audio', 'genre', 'main', and 'tasteprofile'*. The size of the dataset is 12.9 GB for one replica (compressed). Audio consumes 12.3 GB which is 95% of the total size. Inside audio, attributes directory contains 13 comma separated files. Where column names and data types can be used to create schema to read files under features directory.

Quick overview of the data set follows the one-line bash code,

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



Below code snippet been used to get the file directory saved to a text file and similarly could use a AWK command line to produce the same.

Code:

Refer the bash.sh file for the below file type extraction and saving on the text file format.

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

#audio attribute
For i in $(hdfs dfs -find hdfs:///data/msd/audio/attributes -name '*. *')
do
  (hdfs dfs -cat $i | wc -l)
done

#audio features
For i in $(hdfs dfs -find hdfs:///data/msd/audio/features -name '*. *')
do
  (hdfs dfs -cat $i | wc -l)
done
// ------
** Note: Similarly we can take the line count for rest of the directory files and store in the msd_filename.txt
kindly refer the code given on shell.sh file code lines from 22 to 90.
```

Snap shot of the structure of the data directory and the number of line count in each CSV file snapshot as follows,

```
name                                                        contents          file_size    line_count
-----------------------------------------------------------------------------------------------------
audio                                                                         12.3 G
|-- attributes                                                                103.0 K
    |-- msd-jmir-area-of-moments-all-v1.0.attributes.csv                                   21
    |-- msd-jmir-lpc-all-v1.0.attributes.csv                                               21
    |-- msd-jmir-methods-of-moments-all-v1.0.attributes.csv                                11
    |-- msd-jmir-mfcc-all-v1.0.attributes.csv                                              27
    |-- msd-jmir-spectral-all-all-v1.0.attributes.csv                                      17
    |-- msd-jmir-spectral-derivatives-all-all-v1.0.attributes.csv                          17
    |-- msd-marsyas-timbral-v1.0.attributes.csv                                            125
    |-- msd-mvd-v1.0.attributes.csv                                                        421
    |-- msd-rh-v1.0.attributes.csv                                                         61
    |-- msd-rp-v1.0.attributes.csv                                                         1441
    |-- msd-ssd-v1.0.attributes.csv                                                        169
    |-- msd-trh-v1.0.attributes.csv                                                        421
    |-- msd-tssd-v1.0.attributes.csv                                                       1177
|-- features                                                                  12.2 G
    |-- msd-jmir-area-of-moments-all-v1.0.csv               (8 parts-csv.gz)               994623
    |-- msd-jmir-lpc-all-v1.0.csv                           (8 parts-csv.gz)               994623
    |-- msd-jmir-methods-of-moments-all-v1.0.csv            (8 parts-csv.gz)               994623
    |-- msd-jmir-mfcc-all-v1.0.csv                          (8 parts-csv.gz)               994623
    |-- msd-jmir-spectral-all-all-v1.0.csv                  (8 parts-csv.gz)               994623
    |-- msd-jmir-spectral-derivatives-all-all-v1.0.csv      (8 parts-csv.gz)               994623
    |-- msd-marsyas-timbral-v1.0.csv                        (8 parts-csv.gz)               995001
    |-- msd-mvd-v1.0.csv                                    (8 parts-csv.gz)               994188
    |-- msd-rh-v1.0.csv                                     (8 parts-csv.gz)               994188
    |-- msd-rp-v1.0.csv                                     (8 parts-csv.gz)               994188
    |-- msd-ssd-v1.0.csv                                    (8 parts-csv.gz)               994188
    |-- msd-trh-v1.0.csv                                    (8 parts-csv.gz)               994188
    |-- msd-tssd-v1.0.csv                                   (8 parts-csv.gz)               994188
|-- statistics                                                                40.3 M
    |-- sample_properties.csv.gz                                                           992866
genre                                                                         30.1 M
|-- msd-MAGD-genreAssignment.tsv                                                           422714
|-- msd-MASD-styleAssignment.tsv                                                           273936
|-- msd-topMAGD-genreAssignment.tsv                                                        406427
main                                                                          174.4 M
|-- summary
    |-- analysis.csv.gz                                                                    1000001
    |-- metadata.csv.gz                                                                    1000001
tasteprofile                                                                  490.4 M
|-- mismatches
    |-- sid_matches_manually_accepted.txt                                                  938
    |-- sid_mismatches.txt                                                                 19094
|-- triplets.tsv                                            (8 parts-csv.gz)               48373586
```

Both features and triplets are split into 8 partitions of gzip CSV format. Each compressed file needs to be uncompressed on a single cluster, so, the maximum level of parallelism is expected to be 8.

To achieve a higher level of parallelism, repartition method can be used. This allows partitioning by either by the number of partitions or column names. From the feature's directory, the total number of unique songs is between 994,188 and 994,623 which is approximately the same number as information from the Assignment Description.

Taste Profile.

It's prudent to observe the mismatches files in the Taste Profile data set and which comes in a dump text styles, Song ID and Track ID can be extracted using 'substr' function,

```
Code
mismatches = (
    spark.read.text(root + 'tasteprofile/mismatches/sid_mismatches.txt')
    . select(
              F.trim(F.col('value').substr(9,
      18)).alias('song_id').cast(StringType()),
              F.trim(F.col('value').substr(28,
      18)).alias('track_id').cast(StringType())
          )
      )
```

```
mismatches.show(3, False)
mistmatches.count()
```

```
Output

+------------------+------------------+
|song_id           |track_id          |
+------------------+------------------+
|SOUMNSI12AB0182807|TRMMGKQ128F9325E10|
|SOCMRBE12AB018C546|TRMMREB12903CEB1B1|
|SOLPHZY12AC468ABA8|TRMMBOC12903CEB46E|
+------------------+------------------+

Count
         19094
```

We can perform a left anti join in SQL to get the mismatches of the triplets, Anti join is a form of reverse logic instead of returning the matches it would check and return the those rows from the left side of the predicate for which there is no match on the right. 2,578,486 records were removed.

```
Code

triplets = (
    spark.read.format('com.databricks.spark.csv')
    .option('delimiter', '\t')
    .option('header', 'false')
    .option('inferSchema', 'true')
    .schema(
        StructType([
            StructField('user_id', StringType()),
            StructField('song_id', StringType()),
            StructField('playcount', IntegerType())
        ])
    )
    .load(root + 'tasteprofile/triplets.tsv/*')
)

triplets.show(3, False)
triplets.count()


#Remove the mistmatches from triplet.

triplets = (
    triplets
    .join(
        mismatches
        . select('song_id'),
        on='song_id',
        how='left_anti'
    )
)


# count after removed mismatches
triplets.count()
```

```
#Triplets dataframe show

+----------------------------------------+------------------+---------+
|user_id                                 |song_id           |playcount|
+----------------------------------------+------------------+---------+
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQEFDN12AB017C52B|1        |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQOIUJ12A6701DAA7|2        |
|f1bfc2a4597a3642f232e7a4e5d5ab2a99cf80e5|SOQOKKD12A6701F92E|4        |
+----------------------------------------+------------------+---------+
#Count
48373586

#Count after the mismatches were removed.
45795100
```

```
** Note: Reference code, please refer to data_processing.py
```

Define schemas for audio features.

To read audio features it’s ideal to build function. Where it could take filename as input, creates directory paths for attributes and features. Attributes file is read row by row, column name and data type are extracted and transformed into StructField. The schema is created and used to read features file.

msd-jmir-lpc-all-v1.0, random selected and was read. Since the number of column and column name are long, json style printing was used for ease of reading.

```
Code

# Function to read the features in a dataframe
def read_features(filename):
    # create path
    path_attr = root + 'audio/attributes/' + filename + ‘. attributes.csv'
    path_feat = root + 'audio/features/' + filename + '.csv/*'

    # create schema from attributes
    dtypes = {
        'real': DoubleType(),
        'numeric': DoubleType(),
        'string': StringType()
    }
    schema = StructType()
    for row in spark.read.csv(path_attr).collect():
        colname = row[0].lower()
        if colname in ['msd_trackid', 'track_id', 'instancename']:
            colname = 'track_id'
        dtype = row[1].lower()
        schema.add(StructField(colname, dtypes[dtype], True))

    # read features
    df = (
        spark.read.format('com.databricks.spark.csv')
        .option('header', 'false')
        .option('inferSchema', 'true')
        .schema(schema)
        .load(path_feat)
        .withColumn('track_id', F.regexp_replace('track_id', "'", ''))
    )
    return df

# Loading small feature data set.
features = read_features(‘msd-jmir-lpc-all-v1.0’)

# Print the feature heads
print(json.dumps(features.head().asDict(), indent=2))
```

```
Output

{
  "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,
  "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,
  "track_id": "TRHFHQZ12903C9E2D5"
}
```

Some basic statistics of the 10 files are as follows,

```
Code: print(features.drop('track_id').describe().toPandas().transpose())
```

```
Output

                                                 0                    1                    2          3         4
summary                                      count                 mean               stddev        min       max
method_of_moments_overall_standard_deviation_1  994623   0.1549817600174655  0.06646213086143041        0.0     0.959
method_of_moments_overall_standard_deviation_2  994623   10.384550576951835   3.8680013938747018        0.0     55.42
method_of_moments_overall_standard_deviation_3  994623    526.8139724398112    180.4377549977511        0.0    2919.0
method_of_moments_overall_standard_deviation_4  994623    35071.97543290272   12806.816272955532        0.0  407100.0
method_of_moments_overall_standard_deviation_5  994623    5297870.369577217   2089356.4364557962        0.0   4.657E7
method_of_moments_overall_average_1             994623   0.3508444432531261   0.1855795683438387        0.0     2.647
method_of_moments_overall_average_2             994623    27.46386798784021    8.352648595163698        0.0     117.0
method_of_moments_overall_average_3             994623   1495.8091812075486   505.89376391902437        0.0    5834.0
method_of_moments_overall_average_4             994623   143165.46163257837   50494.276171032136  -146300.0  452500.0
method_of_moments_overall_average_5             994623  2.396783048473542E7    9307340.299219608        0.0   9.477E7
```

We can further explore the data set by looking at the Audi Similarities.

Audio Similarity

Code Reference File Name: audiosimilarity_gtj13.py

Focus of this section is to use numerical representations of a song’s audio waveform to predict its genre. Which would enable online music streaming service to offer a unique service to their customers. As the first step, can use describe () function to produce descriptive statistics, then convert it to a Pandas data frame and transpose column to row to accommodate long feature names.The snapshot of descriptive statistics are as follows,
Feature count: 994623

Selecting a small file to process the data: msd-jmir-methods-of-moments-all-v1.0

To obtain the correlation matrix, first the data was converted to RDD and the used the .corr() function.

Correlation matrix for the above file selected.

```
      0     1     2     3     4     5     6     7     8     9
0  1.00  0.43  0.30  0.06 -0.06  0.75  0.50  0.45  0.17  0.10
1  0.43  1.00  0.86  0.61  0.43  0.03  0.41  0.40  0.02 -0.04
2  0.30  0.86  1.00  0.80  0.68 -0.08  0.13  0.18 -0.09 -0.14
3  0.06  0.61  0.80  1.00  0.94 -0.33 -0.22 -0.16 -0.25 -0.22
4 -0.06  0.43  0.68  0.94  1.00 -0.39 -0.36 -0.29 -0.26 -0.21
5  0.75  0.03 -0.08 -0.33 -0.39  1.00  0.55  0.52  0.35  0.28
6  0.50  0.41  0.13 -0.22 -0.36  0.55  1.00  0.90  0.52  0.42
7  0.45  0.40  0.18 -0.16 -0.29  0.52  0.90  1.00  0.77  0.69
8  0.17  0.02 -0.09 -0.25 -0.26  0.35  0.52  0.77  1.00  0.98
9  0.10 -0.04 -0.14 -0.22 -0.21  0.28  0.42  0.69  0.98  1.00
```

Following code snippet was used to define the schema of the Genre.

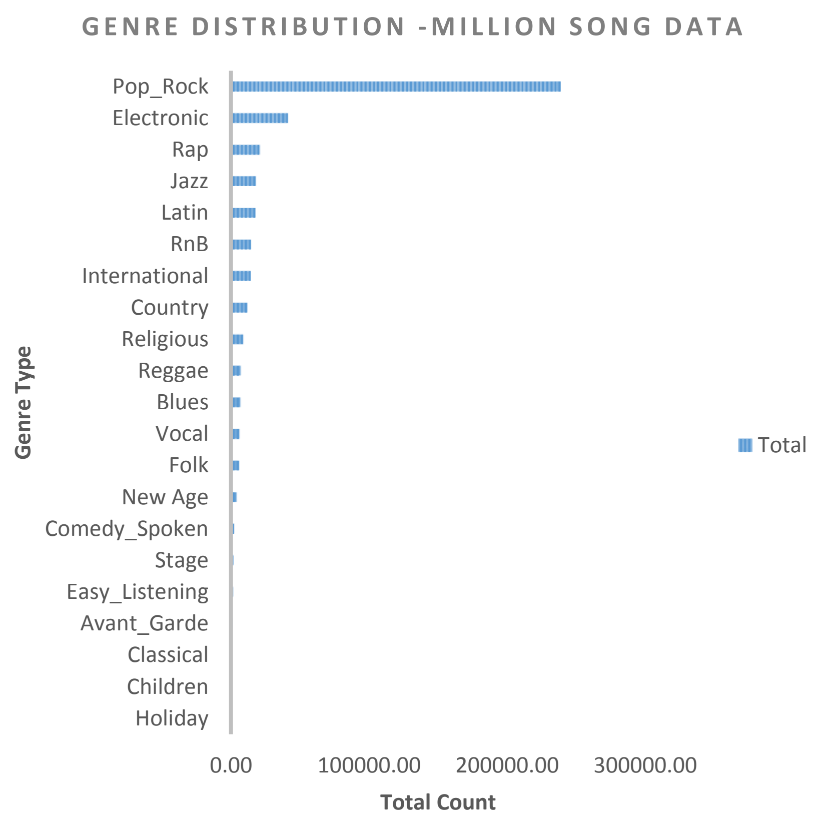
```
Genre1 = (
    spark.read.format('com.databricks.spark.csv')
    .option('delimiter', '\t')
    .option('header', 'false')
    .option('inferSchema', 'true')
    .schema(
        StructType([
            StructField('track_id', StringType()),
            StructField('genre', StringType())
        ])
    )
    .load(root + 'genre/msd-MAGD-genreAssignment.tsv')
)
```

Genre Distribution

It's prudent to look at the Genre distribution and we can read using .csv format and write using tab separated value file. To visualize the distribution of genres, song count for each genre were calculated and exported. Distribution of the genre was plotted using Microsoft excel, and saved the file as tab delimited and used text to column option to trim the data fields.

Summary as follows,

Row Labels	Sum of count
Holiday	200.00
Children	477.00
Classical	556.00
Avant_Garde	1014.00
Easy_Listening	1545.00
Stage	1614.00
Comedy_Spoken	2067.00
New Age	4010.00
Folk	5865.00
Vocal	6195.00
Blues	6836.00
Reggae	6946.00
Religious	8814.00
Country	11772.00
International	14242.00
RnB	14335.00
Latin	17590.00
Jazz	17836.00
Rap	20939.00
Electronic	41075.00
Pop_Rock	238786.00
Grand Total	422714.00



Have performed a 'left_anti' joined into features dataset. Due to the missing genre 574,003 record were removed from list. Once the features been joined with the Genre we can get the following input (Code reference, audio similarity.py: code line 134 to 195.

```
+------------------+--------+
|track_id          |genre   |
+------------------+--------+
|TRAAAAK128F9318786|Pop_Rock|
|TRAAAAV128F421A322|Pop_Rock|
|TRAAAAW128F429D538|Rap     |
+------------------+--------+
```

Once we gropy by the 'genre' we can receive the following summary output for the first 3 values,

```
+----------+------+
|genre     |count |
+----------+------+
|Pop_Rock  |238786|
|Electronic|41075 |
|Rap       |20939 |
+----------+------+
```

Binary Classification

We can first test out the classification problem using the Binary Classification. The models been selected considering the explainability, interpretability, predictive accuracy, training speed, hyperparameter tuning, dimensionality, and issues with scaling. Under binary classification I've selected following 3 algorithms[1] , our problem type is 'Binary Classification' I've selected the below from the documentations.

1. Logistics Regression 2. Random Forests 3. Decision Tree Classifier

Setting up the data for scalar and vector conversion been coded from line 202 to 256 of the reference notebook.

'VectorAssembler' was used to combine all explanatory columns into a single vector column and normalized by 'StandardScaler' command.

```
+------------------+-------------+----------------------------+-----------------------------+
|          track_id|        genre|                    features|                   scfeatures|
+------------------+-------------+----------------------------+-----------------------------+
|TRAAABD128F429CF47|     Pop_Rock|[0.1308,9.587,459.9,27280.0...|[2.022118802771498,2.624321...|
|TRAAADT12903CCC339|Easy_Listening|[0.08392,7.541,423.7,36060....|[1.2973716355396339,2.06425...|
|TRAAAEF128F4273421|     Pop_Rock|[0.1199,9.381,474.5,26990.0...|[1.8536089025405398,2.56793...|
+------------------+-------------+----------------------------+-----------------------------+
only showing top 3 rows
```

Data Splitting

The dataset was randomSplit into the training set (70%) and test set (30%).

```
df=dataset.withColumn('label',F.when(F.col('genre')=='Electronic',1.0).otherwise(0.0))
```

Class Balance

```
#Splitting
training, test = df.randomSplit([0.7, 0.3])

#Class balance
class_0 = training.filter(F.col('label') == 0.0).count()
class_1 = training.filter(F.col('label') == 1.0).count()

#Class Balance split,

In [42]: print(class_0)
266167

In [43]: print(class_1)
28395
```

Selected the Genre = 'Electronic' and labelled as 1 and we have the class balance as Class_0 = 266,167 and Class_1 = 28,395 which is in the ratio of 1 to 10. This could lead to many biased results basis of the selected methods, E.g. Random Forest.

We can use the sampleBy()[2][3] function On the training set down sample , which is sampling stratification method,

> *"It is common for a large population to consist of various-sized subpopulations (strata), for example, a training set with many more positive instances than negatives. To sample such populations, it is advantageous to sample each stratum independently to reduce the total*

[1] https://spark.apache.org/docs/2.3.0/mllib-classification-regression.html
[2] https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/stratified_sampling_example.py
[3] SamplebyKey() function doesn't support in the Python currently.

variance or to represent small but important strata. This sampling design is called stratified sampling.[4]"

```
training_bal = training.sampleBy('label', fractions, seed=1)
```

Model Selection (a.k.a hyperparameter tuning)

Based on the spark.mltuning the following format could be used to build cross validator models for each classification algorithm been chosen, e.g. Logistic regression (Code reference, File: audio similarity.py, Line 234 to 275), as a default value we can use number of folds as 5, this could be 10 as well. In this example we can proceed with 5 folds cross validation to avoid longer iteration time.

Sample:

```
crossval = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=BinaryClassificationEvaluator(),
                          numFolds=2)[5]
```

Comparatively with the sample code provided, there is a slight change using the 'ParamGridBuilder[6]' utility has used to build a grid of parameters around the default value of each parameter. The selected algorithms were defined using default parameter settings. Available parameters and default value for each parameter were observed using 'explainParams()[7]'. Although we can get an idea of using the explainParams () it's prudent analyse few real time cases and set the parameters properly.

```
Linear_reg_cv = CrossValidator(
    estimator=lr,
    estimatorParamMaps=(
        ParamGridBuilder()
        .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
        .addGrid(lr.regParam, [0.0, 0.25, 0.5])
        .build()
    ),
    evaluator=MulticlassClassificationEvaluator(metricName='f1'),
    numFolds=5[8]
```

Selection of the Evaluator: 'F1'-score was used as evaluator, both Precision and Recall are important for the model evaluation. Following are the obtained results using, TP, FP, TN and FN. Used the binary classification (BinaryClassificationEvaluator) function to determine are under ROC.

```
    tp|     fp|     tn|     fn| precision|    recall|        f1|       auc|algorithm
   349|    379| 113471|  11837|     0.479|     0.029|     0.054|     0.763|logistic regression (normal)
  8769|  35608|  78242|   3417|     0.198|     0.720|     0.310|     0.769|logistic regression (balance)
  8769|  35608|  78242|   3417|     0.198|     0.720|     0.310|     0.769|logistic regression (cv)
     0|      0| 113850|  12186|     0.000|     0.000|     0.000|     0.754|random forest (normal)
  8476|  31917|  81933|   3710|     0.210|     0.696|     0.322|     0.781|random forest (balance)
  8476|  31917|  81933|   3710|     0.210|     0.696|     0.322|     0.781|random forest (cv)
   279|    195| 113655|  11907|     0.589|     0.023|     0.044|     0.372|decision tree (normal)
  8206|  31478|  82372|   3980|     0.207|     0.673|     0.316|     0.674|decision tree (balance)
  8366|  27506|  86344|   3820|     0.233|     0.687|     0.348|     0.586|decision tree (cv)
```

Above is the evaluation matrix evaluated and have similar performances and out of which the decision tree algorithm behaves poorly. Out of all three Random Forest has performed well.

The evaluation metrics shows similar output among 3 algorithms, F1-scores were generally poor. A high number of False Positive could be observed. On imbalanced dataset, the models predicted negative class

[4] https://databricks.com/blog/2014/08/27/statistics-functionality-in-spark.html
[5] https://spark.apache.org/docs/latest/ml-tuning.html
[6] https://spark.apache.org/docs/2.1.0/api/scala/index.html#org.apache.spark.ml.tuning.ParamGridBuilder
[7] https://spark.apache.org/docs/2.0.0/api/python/pyspark.ml.html
[8] https://docs.databricks.com/spark/latest/mllib/binary-classification-mllib-pipelines.html

most of the time, thus it produced terrible Recall scores. Hyperparameters tuning added up small improvement with massive computational time trade-off.

For the Multiclass Classification I've selected Random Forest [9]as it's supports the Multiclass, Following are the results. (Code reference: audio_similarity_gtj13.py, Line reference: 408 to 471)

```
   accuracy| precision|    recall|        f1|algorithm
      0.571|     0.394|     0.571|     0.430|random forest (normal)
      0.269|     0.566|     0.269|     0.329|random forest (balance)
      0.578|     0.401|     0.578|     0.448|random forest (one-vs-rest)
      0.575|     0.397|     0.575|     0.449|random forest (cv)
```

One-vs-One and One-vs-All

One-vs-One: Here, we can pick 2 classes at a time, and train a two-class-classifier using samples from the selected two classes only (other samples are ignored in this step). We can repeat this for all the two class combinations and end up with N (N-1)/2 classifiers. While testing, you do voting among these classifiers. Thus, an imbalanced class issue is lesser but the computational burden is high.

The difference between the one vs. one and the one vs. all multiclass classification strategies is the number of learning classifiers. One vs. all, train one classifier per class. The total number of learning classifiers is N. For each training, the model will treat class 'I' as positive and the rest as negative. Thus, an imbalanced class issue could be expected.

To perform multiclass classification, the genre column was converted into an integer index that represents the genre consistently using StringIndexer[10][11] and Pipeline function. The dataset contains 21 classes. The smallest class has a count of 200, the largest is nearly has a count of 24,000 and the median is 6,800. Thus, to study impact of class balancing on multiclass classification, larger classes were down-sampled to 5,000 using sampleBy () function.

Random Forest was chosen to perform multiclass classification. Evaluation metrics such as Accuracy, Precision, Recall, and F1-score were determined using MulticlassClassificationEvaluator.F1-score shows that the imbalanced class performed better than the balanced class. One vs. all produced somewhat better scores than One vs. One.

[9] https://towardsdatascience.com/multi-class-text-classification-with-pyspark-7d78d022ed35
[10] https://spark.apache.org/docs/2.1.0/ml-features.html
[11] https://stackoverflow.com/questions/36942233/apply-stringindexer-to-several-columns-in-a-pyspark-dataframe

Song recommendations

On triplets data set after removing the mismatches there are 1,019,318 unique users and 378,309 unique songs.

Most Popular Songs – Top #10

Highest popular song of the list: SOAOSDF12A58A779F1

```
[Row(user_id='093cb74eb3c517c5179ae24caf0ebec51b24d2a2', song_id='SOAOSDF12A58A779F1', playcount=9667),
 Row(user_id='c11dea7d1f4d227b98c5f2a79561bf76884fcf10', song_id='SOZTEZR12A8C14204B', playcount=3534),
 Row(user_id='d8e6fa08d73821f305b9a3af1cf1e0a704473d82', song_id='SOBONKR12A58A7A7E0', playcount=3532),
 Row(user_id='1854daf178674bbac9a8ed3d481f95b76676b414', song_id='SOVLAWN12A81C234AB', playcount=2948),
 Row(user_id='69807196f964e5b063af898fd1cb098fab2e6a1f', song_id='SOVQQJO12AB0182328', playcount=2381),
 Row(user_id='a263000355e6a46de29ec637820771ac7620369f', song_id='SONSTND12AB018516E', playcount=2368),
 Row(user_id='6b36f65d2eb5579a8b9ed5b4731a7e13b8760722', song_id='SOGDNAW12A6D4F6804', playcount=2165),
 Row(user_id='0d0f80a34807aab31a3521424d456d30bf2c93d9', song_id='SOFRRFT12A8C140C5C', playcount=1890),
 Row(user_id='944cdf52364f45b0edd1c972b5a73d3a86b09c6a', song_id='SOKGULH12A6D4F70BB', playcount=1862),
 Row(user_id='b881152f1394e3c2bbdb9cc853016432ba184c6c', song_id='SOJLWPI12A6D4F93BC', playcount=1739)]
```

Followed by the song IDs, SOZTEZR12A8C14204B & SOBONKR12A58A7A7E0, which have an organic growth compared to the top song ID. It's seems that the user '093cb74eb3c517c5179ae24caf0ebec51b24d2a2' continuously playing this song over a period of time.

User Activity

User ID '093cb74eb3c517c5179ae24caf0ebec51b24d2a2' is the most active user, he has played the songs 13,074 times. 195 songs were played by him, which is 0.05% of the total number of unique songs in the dataset. Song ID SOAOSDF12A58A779F1 has the highest number of play count by this user.

```
Most active user

+------------------+----------------------------------------+---------+
|song_id           |user_id                                 |playcount|
+------------------+----------------------------------------+---------+
|SOAOSDF12A58A779F1|093cb74eb3c517c5179ae24caf0ebec51b24d2a2|9667     |
|SOXFYXJ12A67AD87D1|093cb74eb3c517c5179ae24caf0ebec51b24d2a2|494      |
|SOVBGJV12A6701C53F|093cb74eb3c517c5179ae24caf0ebec51b24d2a2|243      |
+------------------+----------------------------------------+---------+
count : 195
```

Play count for each song ID and the song count for user_id being summarized, Quartiles were set and calculated using approxQuantile() function. The huge difference between 3rd quartile and max value and density plots shows massively right-skewed distribution of both song popularity and user activity.

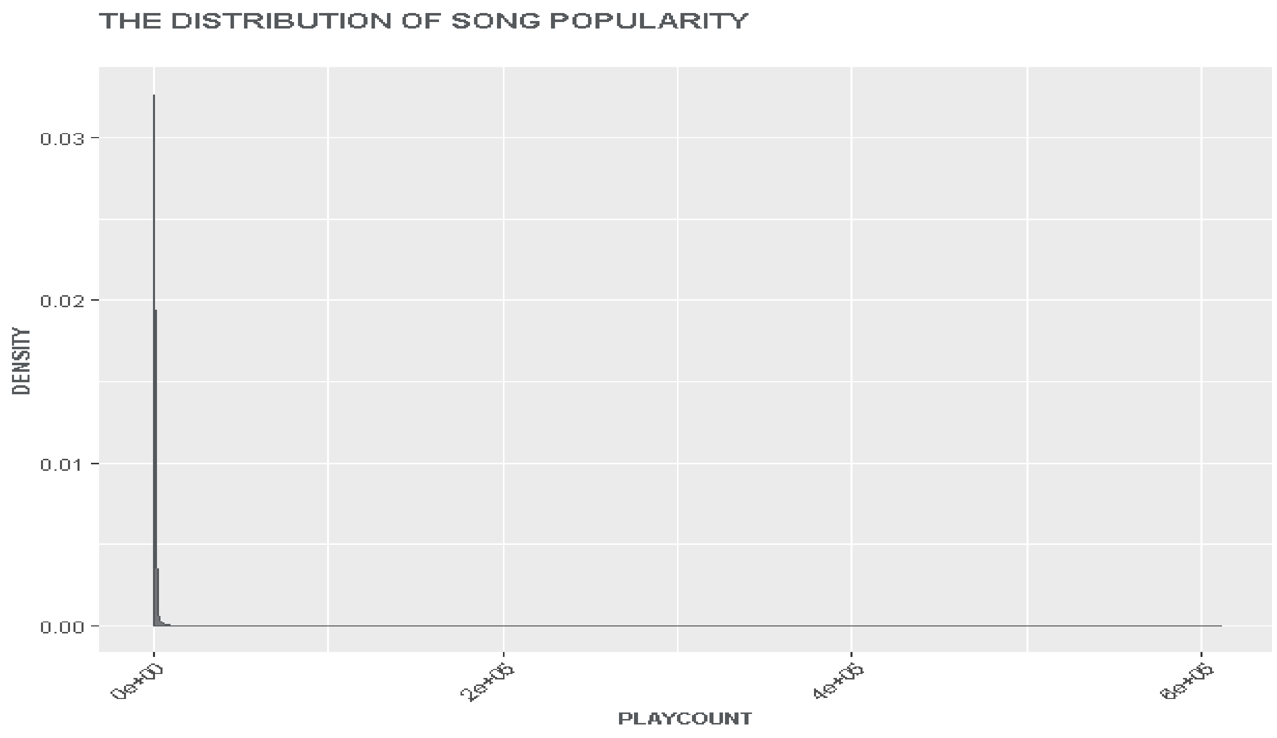
Song Popularity

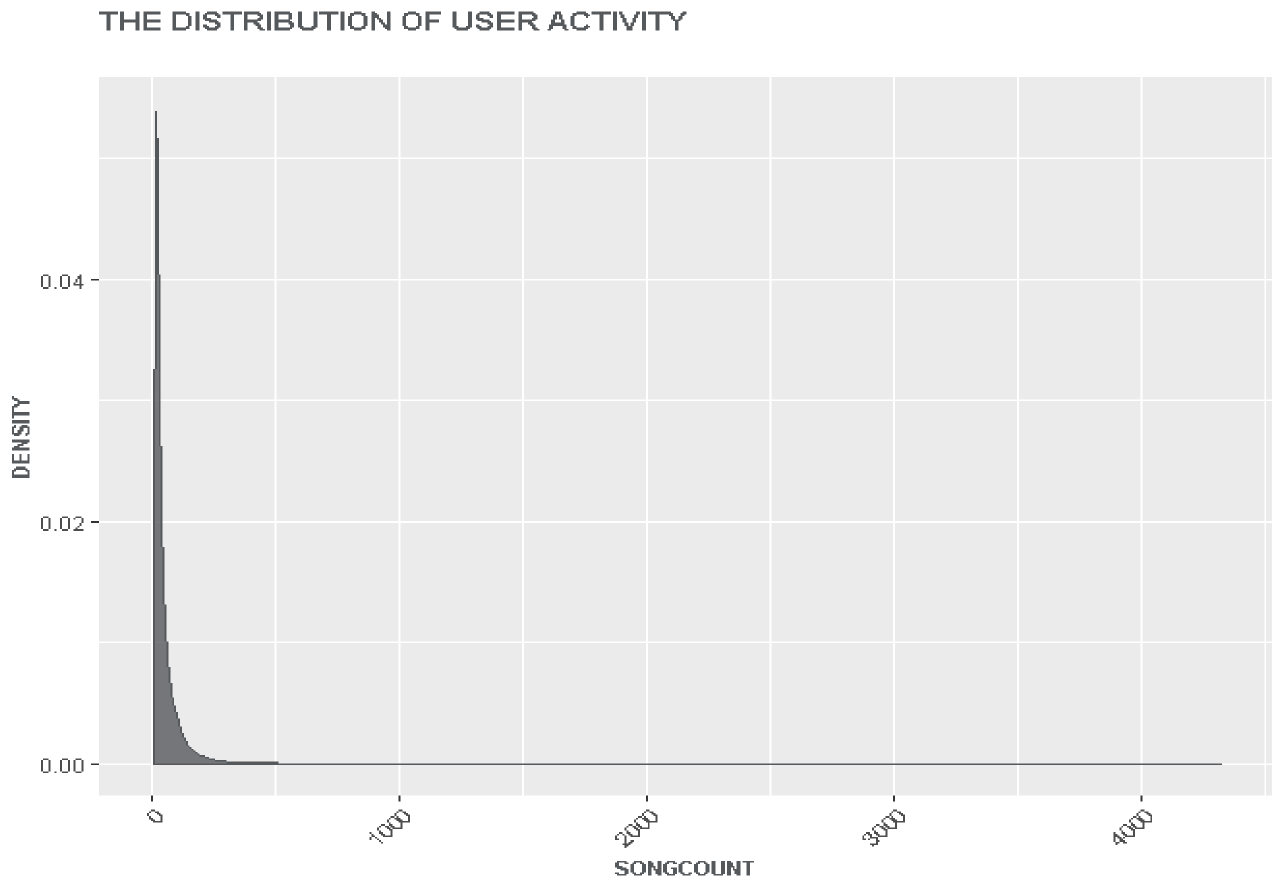
```
song_popularity.approxQuantile('playcount', [0.0, 0.25, 0.5, 0.75, 1.0], 0.0)
= [1.0, 8.0, 31.0, 130.0, 726885.0]
```

User activity

```
user_activity.approxQuantile('songcount', [0.0, 0.25, 0.5, 0.75, 1.0], 0.0)
= [3.0, 15.0, 26.0, 53.0, 4316.0]
```

Distribution Graphs





Both the graphs are highly skewed.

Songs which have been played less than ‘n’ times and users who have listened to fewer than ‘s’ songs were considered as inactive and removed from the dataset due to less information contribution, as if we were process the whole set it would take considerable computational time . 1st quartiles, which were obtained when finding the distribution of song popularity and user activity, were used as N and M values. 10% of user-song pairs was removed.

User ID and song ID were converted into integer index using StringIndexer(). For ease of use, columns were renamed into user, item, and rating which are default column names for ALS model.

```
Code:
dataset = (
    dataset
    .withColumn('user', F.col('user').cast(IntegerType()))
    .withColumn('item', F.col('item').cast(IntegerType()))
    .withColumn('rating', F.col('playcount').cast(IntegerType()))
    .select(['user', 'item', 'rating'])
)

Output :

#Transformed data set

+------+------+------+
|user  |item  |rating|
+------+------+------+
|298837|314027|1     |
|298837|27959 |2     |
|298837|25826 |4     |
+------+------+------+
only showing top 3 rows
```

To make sure that every user in the test set has some user-song plays in the training set as well, sampleBy() method was used for training and test splitting. Instead of using collectAsMap to produce dictionary of fraction which could computationally expensive .The list of unique user ID was collected using groupBy and collect_list. Then, the dictionary of 30% fractions was created by iterating through the list of unique user ID. The test set was sampled out and the training set was obtained by left_anti joining dataset with the test set. The number of observation of the training set and test set were 29,968,400 and 12,849,760 respectively.

In order to select users to validate a recommendation system, the test set was sorted by rating and filtered out the top 3 users.

```
u = [525941, 519321, 208949]
users = sqlContext.createDataFrame(
    [(525941,), (519321,), (208949,)],
    StructType([StructField('user', IntegerType())])
)
```

Using Alternative Least Square s[12] to train implicit matrix factorization which is a class of collaborative filtering algorithm which we used for recommender systems.User ID and song ID were converted into integer index using StringIndexer(). For ease of use, columns were renamed into user, item, and rating respectively which are default column names for ALS model and picks the names instantly.

[12] https://spark.apache.org/docs/2.2.0/mllib-collaborative-filtering.html

```
Code :

recommends = model.recommendForUserSubset(users, 10)

recommends.printSchema()

Output:

  root
   |-- user_index: integer (nullable = false)
   |-- recommendations: array (nullable = true)
   |    |-- element: struct (containsNull = true)
   |    |    |-- song_index: integer (nullable = true)
   |    |    |-- rating: float (nullable = true)
```

UDF (User define function) was used to extract recommended songs (recommends) from the array, Str for the song recommendation as follows,

```
#UDF for the recommend

def recommend(recommendations):
    items = []
    for item, rating in recommendations:
        items.append(item)
    return items

udf_recommend  =  F.udf(lambda  recommendations:  recommend(recommendations), ArrayType(IntegerType()))
```

Testing the model

By using the above User define function we receive the following results,

```
recommends = (
    recommends
    .withColumn('recommends', udf_recommend(recommends.recommendations))
    .select(
        F.col('user').cast(IntegerType()),
        F.col('recommends')
    )
)

recommends.show(3, False)
```

Output

```
  +-------+--------------------------------------------------------------------------------+
  |user   |recommends                                                                      |
  +-------+--------------------------------------------------------------------------------+
  |525941|[241640, 134941, 18502, 103829, 248908, 263679, 155493, 125402, 179474, 85901]|
  |208949|[241640, 215904, 208705, 251809, 183896, 45731, 277120, 286985, 258726, 60703]|
  |519321|[209307, 107204, 127445, 80147, 166446, 196345, 69627, 149664, 217693, 163124]|
  +-------+--------------------------------------------------------------------------------+
```

Actual listened songs (ground_truths) for each user were collected from the test set using groupBy and collect_list and following is the sample code snippet.

```
ground_truths = (
    test
    .filter(F.col('user').isin(u))
    .orderBy('rating', ascending=False)
    .groupBy('user')
    .agg(F.collect_list('item').alias('ground_truths'))
)
```

We could then join both ground_truth and recommend results in to a tuple and convert it to a RDD to calculate the ranking metrics, Results are as follows,

Results for the ground truth,

```
+------+------------------------------------------------------------------------------------------------------------------------------------+
|user  |ground_truths                                                                                                                       |
+------+------------------------------------------------------------------------------------------------------------------------------------+
|525941|[51266, 23962, 27235, 2993]                                                                                                         |
|208949|[0, 2642, 4800, 25028, 21565, 29, 120303, 284034, 103178, 5256, 8175, 405, 1559, 1276, 612, 32810, 36614, 21961, 41137, 40332, 2925]|
|519321|[3251, 31284, 128667, 85783, 86341, 30302, 2452, 16794, 2742]                                                                       |
+------+------------------------------------------------------------------------------------------------------------------------------------+
```

We can measure the effectiveness of the recommendation system by the precision of how the algorithm recommend the songs to the User, The suggested ranking metrics are an indication of different sort of precision we can take in to account, in this exercise we have used NDGC, MAP and Precision.

To compare the recommends and ground truth together we can join the recommends and ground truth table on User, can use __getatrr__ can be used to faster processing without cache the object results. Usually the above magic method being used to catch reference attribute in the object that don't exist in the object. Once we use this and when we use parallelize() function on the data frame (usually the DF are parallelized in the background) we can avoid some of the cluster memory issues.

Comparative results are as follows,

```
compare = recommends.join(ground_truths, on='user', how='left')
compare = [(r.__getattr__('recommends'), r.__getattr__('ground_truths')) for r
in compare.collect()]
compare = sc.parallelize(compare)

# print metrics
metrics = RankingMetrics(compare)
print(metrics.precisionAt(5))
  # 0.06666666666666667
print(metrics.ndcgAt(10))
  # 0.06506334166535027
print(metrics.meanAveragePrecision)
  # 0.02777777777777776
```

The results we received are not satisfactory. But we could try out with a different user set and may be with another stratified data sample to improve the results.

There are other sorts of Metrics as well within the same paradigm DCG, IDCG and Liftindex which we have not considered in here.

Advantages, Limitations & Improvements

The Collaborative filter method is extracting the information of existing user and an item. In the event of a cold start problem; where the new and existing data available data is not yet current the collaborative filtering method is not an effective one to suggest recommendation and it won't be able to predict precisely. What happens at this stage model will go by the rank and provide more biased based on the popularity which may or may not relevant to the user preferences.

In most of the present day recommender systems both implicit and explicit rating are taking into consideration when recommending items, or content. When dealing with Collaborative Filtering something to take in to consideration could be the normalised approach, focusing some of the unusual patterns (unusual taste leads to poor recommendations) and finding what causes them and take it out from the model. It's better to have control chart for data pipeline to observe the peaks make autonomous normalisation effects to the data set.

Improvement summary

- Time delay of correcting measures
- Requirements , features and development for every system
- Users moods are not important which leads in to inaccuracy problems

Other recommendation systems

1) Content based recommendation
    a. Without user's evaluation or ratings.
    b. Uses machine language to acquire information.
    c. Using an algorithmic approach: Decision trees, neural networks and vector based methods.
2) Knowledge based recommendation
    a. Based on demand and preferences of the user.
    b. Predications are merely by function and feature objects.

User preferences could profiled (taste profile) using many methods, based on the, colours, gender, topics and using their social media profile. Taking demographic based profiling would be an ideal method to distinguish to use when making recommendation to a cold start problem.

Usage of metadata

Sample Meta data attributes of the MSD and these Meta data attribute differ by the platform,

```
{
    "analyzer_version": null,
    "artist_7digitalid": "4069",
    "artist_familiarity": "0.6498221002008776",
    "artist_hotttnesss": "0.3940318927141434",
    "artist_id": "ARYZTJS1187B98C555",
    "artist_latitude": null,
    "artist_location": null,
    "artist_longitude": null,
    "artist_mbid": "357ff05d-848a-44cf-b608-cb34b5701ae5",
    "artist_name": "Faster Pussy cat",
    "artist_playmeid": "44895",
    "genre": null,
    "idx_artist_terms": "0",
    "idx_similar_artists": "0",
    "release": "Monster Ballads X-Mas",
    "release_7digitalid": "633681",
    "song_hotttnesss": "0.5428987432910862",
    "song_id": "SOQMMHC12AB0180CB8",
    "title": "Silent Night",
    "track_7digitalid": "7032331"
  }
```

And the Meta data provided in the MSD summary could give more detailed preferences and grill down to most of the sensitive findings of the song set. There are many I've noticed that some of the research been carried out on the relationship between artist hotness and song popularity , these are very sensitive approaches and more the explicit information more relationships can be taken in to account and new patterns could be performed. Out of these possible outcomes, we can use recommender system to produce some commercially viable products,

1) Type of song videos (content writers) nudity + or –
2) Lyrics (topic modelling for songs)
3) Next hit song list
4) Recommendation for artist on their content (highly out of control but can provide insights)
5) If we capture comments from 'Youtube' on the songs and fuse it with the negative and positive comments that could also be a good selling point.

To achieve above we can do many clustering and filtering, the best way of performing these sort high dimensional data could be handled using different parsing and clustering methods while securing the differential privacy.