DATA420
Assignment 2 Report

Chris McDonald

Table of Contents

Section 1: Background

This report will serve as a journey through the process of using data to create models used for both classification and recommendation. The models will utilize the Spark.ml library, of which the associated documentation proved invaluable for understanding both how the models work and the associated hyperparameters that affect a model's performance (Spark documentation is referenced in the report where relevant).
All code can be found in the accompanying Notebook and any relevant full-size plots or tables can be located in the appendices at the end of this report.

Section 2: Processing

In this section, the datasets in the **msd** directory will be explored and some transformation will also occur. To begin, I will give an overview of the directory structure, accompanied by a tree diagram and a table containing metadata from the various datasets. Then I will define some functions that will allow custom schema to be generated for datasets in the **audio** directory and for column names to be transformed into a more readable format.

Explore The Data

Using *hdfs* commands to explore the **msd** directory, I was able to generate the following tree diagram to show the directory layout:

```
msd/
├── audio/
│   ├── attributes/
│   │   ├── 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/
│       ├── msd-jmir-area-of-moments-all-v1.0.csv
│       ├── msd-jmir-lpc-all-v1.0.csv
│       ├── msd-jmir-methods-of-moments-all-v1.0.csv
│       ├── msd-jmir-mfcc-all-v1.0.csv
│       ├── msd-jmir-spectral-all-all-v1.0.csv
│       ├── msd-jmir-spectral-derivatives-all-all-v1.0.csv
│       ├── msd-marsyas-timbral-v1.0.csv
│       ├── msd-mvd-v1.0.csv
│       ├── msd-rh-v1.0.csv
│       ├── msd-rp-v1.0.csv
│       ├── msd-ssd-v1.0.csv
│       ├── msd-trh-v1.0.csv
│       └── msd-tssd-v1.0.csv
├── genre/
│   ├── msd-MAGD-genreAssignment.tsv
│   ├── msd-MASD-styleAssignment.tsv
│   └── msd-topMAGD-genreAssignment.tsv
├── main/
│   ├── analysis.csv.gz
│   └── metadata.csv.gz
└── tasteprofile/
    ├── mismatches/
    │   ├── sid_matches_manually_accepted.txt
    │   └── sid_mismatches.txt
    └── triplets.tsv
```

From here, using more *hdfs* commands, I compiled metadata about each dataset including size, format, datatypes and number of rows:

File Name	Size	Rows	Format	Data Types
audio/attributes/msd-jmir-area-of-moments-all-v1.0.attributes.csv	1.0 KB	21	CSV	String
audio/attributes/msd-jmir-lpc-all-v1.0.attributes.csv	671 B	21	CSV	String
audio/attributes/msd-jmir-methods-of-moments-all-v1.0.attributes.csv	484 B	11	CSV	String
audio/attributes/msd-jmir-mfcc-all-v1.0.attributes.csv	898 B	27	CSV	String
audio/attributes/msd-jmir-spectral-all-all-v1.0.attributes.csv	777 B	17	CSV	String
audio/attributes/msd-jmir-spectral-derivatives-all-all-v1.0.attributes.csv	777 B	17	CSV	String
audio/attributes/msd-marsyas-timbral-v1.0.attributes.csv	12.0 KB	125	CSV	String
audio/attributes/msd-mvd-v1.0.attributes.csv	9.8 KB	421	CSV	String
audio/attributes/msd-rh-v1.0.attributes.csv	1.4 KB	61	CSV	String
audio/attributes/msd-rp-v1.0.attributes.csv	34.1 KB	1,441	CSV	String
audio/attributes/msd-ssd-v1.0.attributes.csv	3.8 KB	169	CSV	String
audio/attributes/msd-trh-v1.0.attributes.csv	9.8 KB	421	CSV	String
audio/attributes/msd-tssd-v1.0.attributes.csv	27.6KB	1,177	CSV	String
audio/features/msd-jmir-area-of-moments-all-v1.0.csv	65.5 MB	994,623	CSV	Float/String
audio/features/msd-jmir-lpc-all-v1.0.csv	53.1MB	994,623	CSV	Float/String
audio/features/msd-jmir-methods-of-moments-all-v1.0.csv	35.8 MB	994,623	CSV	Float/String
audio/features/msd-jmir-mfcc-all-v1.0.csv	70.8 MB	994,623	CSV	Float/String
audio/features/msd-jmir-spectral-all-all-v1.0.csv	51.1 MB	994,623	CSV	Float/String
audio/features/msd-jmir-spectral-derivatives-all-all-v1.0.csv	51.1 MB	994,623	CSV	Float/String
audio/features/msd-marsyas-timbral-v1.0.csv	412.2 MB	995,001	CSV	Float/String
audio/features/msd-mvd-v1.0.csv	1.3 GB	994,188	CSV	Float/String
audio/features/msd-rh-v1.0.csv	240.3 MB	994,188	CSV	Float/String
audio/features/msd-rp-v1.0.csv	4.0 GB	994,188	CSV	Float/String
audio/features/msd-ssd-v1.0.csv	640.6 MB	994,188	CSV	Float/String

audio/features/msd-trh-v1.0.csv	1.4 GB	994,188	CSV	Float/String
audio/features/msd-tssd-v1.0.csv	3.9 GB	994,188	CSV	Float/String
genre/msd-MAGD-genreAssignment.tsv	11.1 MB	422,714	TSV	String
genre/msd-MASD-styleAssignment.tsv	8.4 MB	273,936	TSV	String
genre/msd-topMAGD-genreAssignment.tsv	10.6 MB	406,427	TSV	String
main/analysis.csv.gz	55.9 MB	1,000,001 (inc. header row)	GZIP	Int/Float/String
main/metadata.csv.gz	118.5 MB	1,000,001 (inc. header row)	GZIP	Int/Float/String
tasteprofile/mismatches/ sid_matches_manually_accepted.txt	89.2 KB	938	TXT	String
tasteprofile/mismatches/sid_mismatches.txt	1.9 MB	19,094	TXT	String
tasteprofile/triplets.tsv	488.4 MB	48,373,586	TSV	String

I was originally using the *hdfs dfs -ls* command for returning file sizes in bytes, but some datasets (e.g. all datasets in the audio/features directory) did not return any value for file size. So I instead used the *hadoop fs -du -h* command. This command returned a rounded file size in the relevant unit (KB/MB etc) rather than as bytes. Because of this, the file sizes may not be 100% accurate as they have been rounded.
I'm aware that the dataset downloads page (*Million Song Dataset Benchmark Downloads - IMP at IFS, UT Vienna*, 2024) also contains metadata about number of rows and file sizes, but I made sure all data collected was done through querying the actual dataset as it appears in Azure blob storage.

There are a total of 1,000,000 unique songs in this directory (as indicated by both the dataset title and the number of rows in both GZIP files located in the **main** directory). The datasets in the **features** directory all have a number of rows slightly below this, which is to be expected, since not all songs from the dataset were analysed (as detailed in the *Audio Features* section of the assignment brief).
The only dataset to feature more than a million rows is the **triplets** dataset from the **tasteprofile** directory. This is because the dataset contains not only song data, but user data and play data from various organisations. This is detailed on the dataset website (*The Echo Nest Taste Profile Subset | Million Song Dataset*, n.d.).

Preprocessing

The **audio/attributes** datasets contain both the attribute names and data types for all columns in the corresponding **audio/features** dataset. It is considered best practise to keep data and schema separate in databases for a number of reasons. The concept of 'loose-coupling' is widely used in the industry for efficiency and ensuring that any changes needing to be made to schema can be done so without disruption to the data. There are also aspects regarding data security, collaboration and flexibility (McKee, 2023).

To generate custom schema for a dataset from the **audio/features** directory, I created a function that takes the dataset's file-path as an argument. The file-path is

then transformed via string methods so the corresponding dataset from the **audio/attributes** directory can be loaded into Spark. An empty schema is then created before a *while* loop is initiated The *while* loop iterates through the number of rows in the attribute dataset, taking both the attribute name and the specified datatype. Since there are no 'real' or 'numeric' datatypes in Spark (*Data Types — PySpark 3.5.5 Documentation*, 2025), I used *DoubleType* in place of 'real' and *DecimalType(10,6)* in place of 'numeric'. Any other datatype would use *StringType* by default. The resulting *StructField* was then added to the schema.
Once the while loop has finished and the schema was returned from the function, it was used to load the dataset.

Now that the dataset was loaded into Spark with the correct schema, we could look at changing the column names, since the schema just used the default name from the **attributes** dataset, which was typically overlong and needed refining.
I created a function that takes a list of the dataset column names and the dataset file-path as arguments. From here, the column renaming method was unique to which section the dataset was from (*RPF/Marsyas/jMir*).

For *Marsyas* and *jMir* datasets, all instances of the words 'mean' and 'average' were replaced with the abbreviation **AVG** and any instances of 'standard deviation' or 'std' were replaced with **SD**.
Irrelevant words that appeared in all column names were also removed as well as specific changes depending on keywords in the file-path (e.g. the **spectral** and **spectral-derivatives** datasets had the exact same column names, so I added a **DRV** to the end of each column name in **spectral-derivatives** to differentiate it from the **spectral** dataset column names). For example, the column name '*Mean_Acc5_Mean_Mem20_Centroid_Power_powerFFT_WinHamming_HopSize512_WinSize512_Sum_AudioCh0'* was renamed '*AVG_Centroid'.*
For *RPF* datasets, they all had the exact same naming conventions for their columns. So I added the unique abbreviated code found in the dataset's file-path (*MVD,RH,RP* etc.) to the start of the column name, then added the original column integer afterwards (starting from 1 as opposed to zero). So, for example, the first column from the **ssd** dataset had its column name changed from '*component_0*' to '*SSD_1*'.
Since all **features** datasets included the track ID as the final column, this was renamed to '*Track_ID*' for consistency across all datasets.
The list of new column names was then returned by the function, where it was used to replace all the existing column names in the dataset.

Section 3: Audio Similarity

Now we begin the process of training models on our song data for classification purposes. Using audio features contained in specific datasets, we will experiment with training a number of different models in order to see how well the data can be used to predict the genre of a song, both in a binary and multiclass manner. Finally, we will explore some of the associated hyperparameters and how they can be utilised to improve model performance.

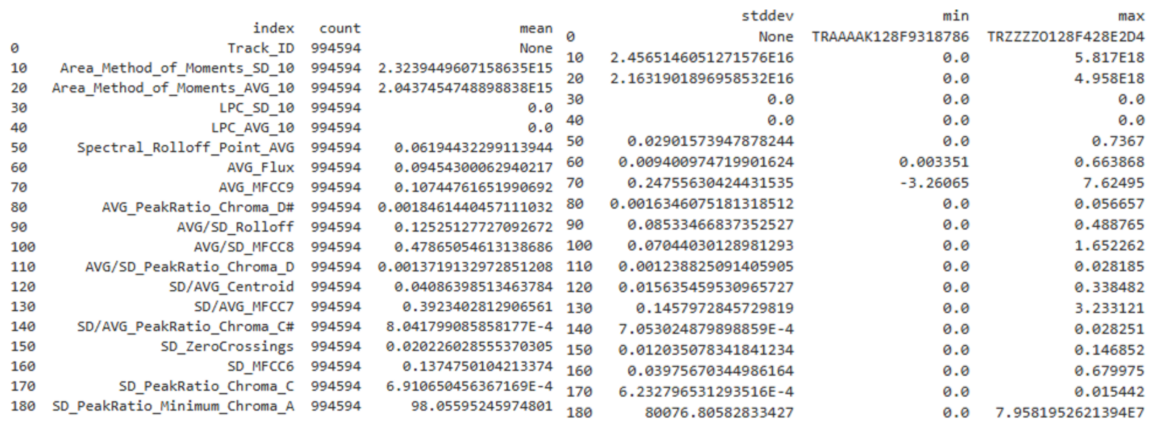
Data Preparation

The Million Songs Dataset contains many sub-datasets consisting of audio features for specific tracks, of which 4 will be used for this section.
There was little information on the Million Songs Dataset website about what each sub-dataset represented, so I utilized *ChatGPT* (OpenAI, 2025) to give a brief description of the 4 datasets, which is paraphrased below:

- **Area of Moments**: Captures distributional characteristics of audio features over time. Provide a compact representation of a song's timbral and rhythmic contents.
- **LPC (Linear Predictive Coding)**: Captures timbral characteristics of audio, useful for distinguishing between genres.
- **Spectral**: Capture the frequency content of an audio signal over time.
- **Timbral**: Utilized to extract timbral features from audio tracks.

It is clear that all 4 datasets contain features that are useful for defining a song's timbral qualities, and thus it's genre, which is what we'll be doing here.

The first step was to load the 4 relevant datasets and then inner join them based on the *Track ID*. Then all features were put into a single vector column, where descriptive statistics could then be generated.
Due to the high number of features, I have selected a systematic sample of features with their descriptive statistics:



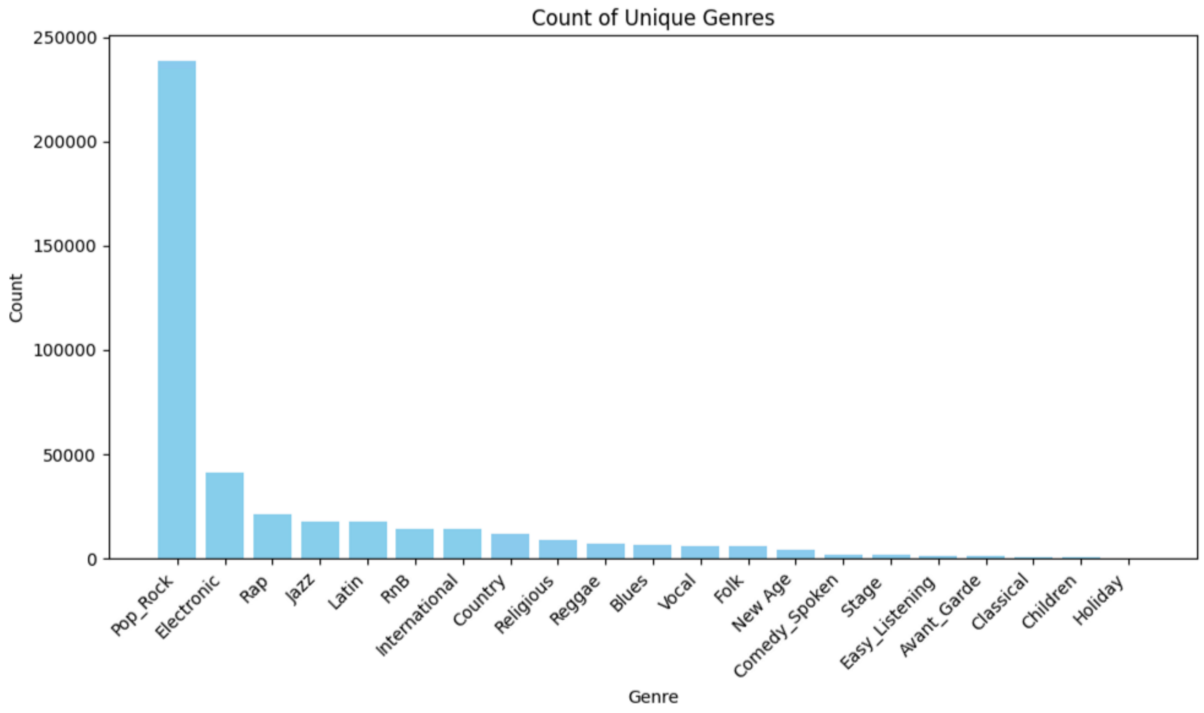
```
                               index   count                       mean
0                           Track_ID  994594                       None
10        Area_Method_of_Moments_SD_10  994594   2.3239449607158635E15
20       Area_Method_of_Moments_AVG_10  994594   2.0437454748898838E15
30                         LPC_SD_10  994594                        0.0
40                        LPC_AVG_10  994594                        0.0
50          Spectral_Rolloff_Point_AVG  994594     0.06194432299113944
60                          AVG_Flux  994594     0.09454300062940217
70                         AVG_MFCC9  994594     0.10744761651990692
80            AVG_PeakRatio_Chroma_D#  994594   0.0018461440457111032
90                   AVG/SD_Rolloff  994594     0.12525127727092672
100                    AVG/SD_MFCC8  994594     0.47865054613138686
110        AVG/SD_PeakRatio_Chroma_D  994594   0.0013719132972851208
120                 SD/AVG_Centroid  994594     0.04086398513463784
130                    SD/AVG_MFCC7  994594      0.3923402812906561
140       SD/AVG_PeakRatio_Chroma_C#  994594    8.041799085858177E-4
150              SD_ZeroCrossings  994594     0.020226028555370305
160                       SD_MFCC6  994594      0.1374750104213374
170           SD_PeakRatio_Chroma_C  994594    6.910650456367169E-4
180  SD_PeakRatio_Minimum_Chroma_A  994594       98.05595245974801
```

```
                      stddev               min                max
0                       None  TRAAAAK128F9318786  TRZZZZO128F428E2D4
10     2.4565146051271576E16               0.0           5.817E18
20     2.1631901896958532E16               0.0           4.958E18
30                       0.0               0.0                0.0
40                       0.0               0.0                0.0
50       0.02901573947878244               0.0             0.7367
60      0.009400974719901624          0.003351           0.663868
70       0.24755630424431535          -3.26065            7.62495
80     0.0016346075181318512               0.0           0.056657
90       0.08533466837352527               0.0           0.488765
100      0.07044030128981293               0.0           1.652262
110     0.001238825091405905               0.0           0.028185
120     0.015635459530965727               0.0           0.338482
130       0.145797284572981 9              0.0           3.233121
140     7.053024879898859E-4               0.0           0.028251
150     0.012035078341841234               0.0           0.146852
160      0.03975670344986164               0.0           0.679975
170     6.232796531293516E-4               0.0           0.015442
180         80076.8058283342 7             0.0  7.9581952621394E7
```

As we can see from the sample above, most features values fall between 0 and 1, with 2 even registering zeros across the board. So there is reason to believe there is strong distribution similarity between many features.
However, when looking at the full dataset (outside of the above sample of features), there are some features that have values over 1,000,000. This means that we will have to look at perhaps normalizing the data later.
I also constructed a dataframe of all correlations between features (these printouts are too large to replicate here, but they are located in the associated Notebook). It shows many groupings of features (typically from the same dataset) that are very highly correlated.

Ultimately, we have 268 unique feature pairs with a correlation greater than 0.95. There are also 30 features that have 11 unique correlations above 0.95 with other features, most of which are clustered together from the same sub-dataset.
There is an argument for removing many features from the joined dataset since they have such high correlation with other features. There are also some features that have no meaningful data associated (like the LPC features above that only registered results of zero). This would reduce computational complexity and training time, since there would be fewer features to train on. *I will engage in pre-training data processing in the next question.*

Next, the MAGD dataset was loaded in and the genre counts were plotted:



As seen, *Pop/Rock* is by far the most dominant genre in the dataset. This does pose problems for any binary or multi-class classification model we may wish to train. For example, in a binary model, if we do a *Pop/Rock* vs. *non Pop/Rock* (or any other such combination), there will be a substantial class imbalance, which would have to be mitigated by using resampling techniques or weightings. For a multi-class model, it will tend to favour the majority class (in this case, *Pop/Rock*) and minority classes will likely suffer from low recall and precision since there are fewer examples for the model to be trained on. Again, various techniques will need to be employed to ensure the model is trained fairly.

Lastly, the **genre** and **audio features** datasets were joined so each unique *Track_ID* now has its associated genre.

Binary Model

Now that the data has been loaded into Spark, we can begin the process of training models.

The 3 classification algorithms we will be using are **LogisticRegression**, **RandomForestClassifier** & **GBTClassifier**. All 3 have pros and cons, but all are suitable candidates for training our model. All information was gathered from the Spark Apache documentation (*Classification and Regression - Spark 2.4.5 Documentation*, n.d.).

- **LogisticRegression** is very easily explainable and interpretable, has a high training speed, simple hyperparameter tuning and handles high-dimensional data well. However, it struggles with non-linear feature interactions compared to tree-based algorithms.

- **RandomForestClassifier** has a high predictive accuracy, parallelized scaling and is good at handling noise and outliers. Its limitations are around interpretability, training speed, memory usage and hyperparameter tuning.

- **GBTClassifier** is considered highly accurate for binary classification models, offers flexible hyperparameter tuning and has a great ability to capture complex more subtle feature interactions. Downsides include a slow training speed compared to the previous algorithms, its inability to handle multi-class classification natively and can be prone to overfitting if not tuned properly.

As previously mentioned, I considered removing any features that showed high correlation with other features. But considering the number of features we have is not excessively large and the **LogisticRegression** algorithm has regularization of data built in, I left all features intact except for those that only had zero values, of which there were two. Since these features offered no meaningful information, I removed them from the dataset.
Normalization of the data was something else to consider. Since the **LogisticRegression** algorithm uses the dot product for calculations, it is very susceptible to features with dramatically different distributions, running the risk of the model shrinking feature coefficients unevenly. Because of this, I also created a normalized version of the data to use for the **LogisticRegression** algorithm. The other 2 algorithms do not require the data to be normalized or regularized.

I then created a new binary label to represent if a song is in the Electronic genre (**1**) or in a different genre (**0**).
This resulted in the following class balance:

Label	**Count (No. of Tracks)**
1 (Electronic)	40,662
0 (Not Electronic)	379,938

Due to the class imbalance, there will need to be some resampling conducted before training is to commence. Electronic music accounts for roughly 9.7% of the dataset, which means that any model trained without resampling will likely make biased predictions towards the majority class (**0**) to try and increase model *accuracy*. At the same time, a metric like *recall* is likely to be low due to the smaller pool of minority observations (**1**).

Before resampling occurs, the data needs to be split in training and testing sets. Stratified sampling needs to be used to ensure class balance is maintained. Because this is a binary classification, I simply used the Spark *Windows* function while hard-coding the class balance to ensure it was maintained. This resulted in an 80/20 split for training and testing data, which then had redundant columns removed and were then cached for reproducibility.
Due to the class imbalance, it was important that some resampling occurred. I could choose between oversampling, subsampling or observation weighting. Ultimately, I ran all 3 on each algorithm, as well as training a model with no resampling for comparison. Results are as follows:

Model	Resampling	Precision	Recall	Accuracy	AUROC
LogisticRegression	Normalized	0.74049	0.19157	0.91535	0.85920
	Normalized, Subsampling	0.65989	0.33137	0.91884	0.86405
	Normalized, Oversampling	0.23381	0.85971	0.71405	0.86844
	Normalized, Obs. Weighting	0.32543	0.76663	0.82380	0.86904
RandomForestClassifier	None	0.78185	0.11017	0.91100	0.81137
	Subsampling	0.70447	0.23804	0.91669	0.83383
	Oversampling	0.20389	0.87176	0.65850	0.84639
	Obs. Weighting	0.28293	0.73823	0.79380	0.84937
GBTClassifier	None	0.68086	0.31661	0.91958	0.87076
	Subsampling	0.59097	0.45408	0.91683	0.87423
	Oversampling	0.23897	0.84776	0.72428	0.87210
	Obs. Weighting	0.30393	0.76921	0.80736	0.87289

One important thing I had to think about with the model performance was the tradeoff between *Precision* and *Recall*. A lower *precision* meant more non-Electronic songs were being classified as Electronic and a lower *recall* meant more Electronic songs were not being classified as Electronic. So which one is more important? In this particular scenario, there is no overly high cost for either one, so it’s more about finding a healthy balance between the two.

Based on what we know about the class balance, we can essentially discard *accuracy* as a reliable metric. Because the class balance heavily favoured the negative class, it’s easy for the model to achieve high accuracy by just predicting a song is non-electronic the majority of the time, since accuracy is about how many predictions the model got correct. So if the majority of the data is for the negative class, guessing negative will likely result in a higher accuracy than guessing positive.

From the model performances, we can see that when no resampling occurs, *precision* outweighs *recall* fairly heavily and the *AUROC* is lower than any model that used resampling. This is expected because precision is this context essentially just

means ‘*of all the songs predicted as Electronic, how many actually were?*’. So the model, when it was unsure about what genre a song was, would just put it in the negative class because there was a higher likelihood that it would be due to the class imbalance. These models showed a high number of false and true negatives because of this, but because the number of true/false positives was much lower, it resulted in fairly good *precision* scores.
Subsampling still tended to favour *precision* over *recall*, just to a lesser extent. The subsampling I did retained just over 50% of the negative class, while retaining all of the positive class. So recall has improved now that the class balance has reduced and *AUROC* also increased for all 3 of the algorithms that used subsampling over no sampling.
Oversampling shows the biggest change so far, with *recall* now vastly outweighing *precision*. Because we have oversampled the positive class, the model sees more instances of it during training and now seemingly favours the positive class when unsure. The **LogisticRegression** and **RandomForestClassifier** algorithms have higher *AUROC* with oversampling.
Lastly, **observational weighting** was done where every instance of the negative class was weighted by 0.5 and every instance of the positive class was weighted 5. It performed similar to the oversampling models, but with a smaller gap between *precision* and *recall*. The weighted models scored the highest *AUROC* out of all **LogisticRegression** and **RandomForestClassifier** models, while the subsampling model had the highest *AUROC* for the **GBTClassifier**.
To determine which model performs the best, I would argue that the most relevant metric is the **F1 score**. This metric balance the false positives and negatives, is more informative than accuracy for unbalanced datasets like ours and penalises models that strongly favour either precision or recall. Since we’ve already established that neither precision or recall is necessarily most costly than the other, F1 is a good metric to determine a good balance.

Multiclass Model

Now that we have experimented with binary models, we can now progress to multiclass models to see how well a model can distinguish between all genres. The **LogisticRegression** algorithm will work well here since it natively supports multiclass classification. We just need to make sure to specify that the multinomial family is to be used, since that allows the model to train with a single weight vector per class, rather than training a separate binary classifier for each class.
Using a *StringIndexer*, the genre column was converted to integers that gave the following class balance:

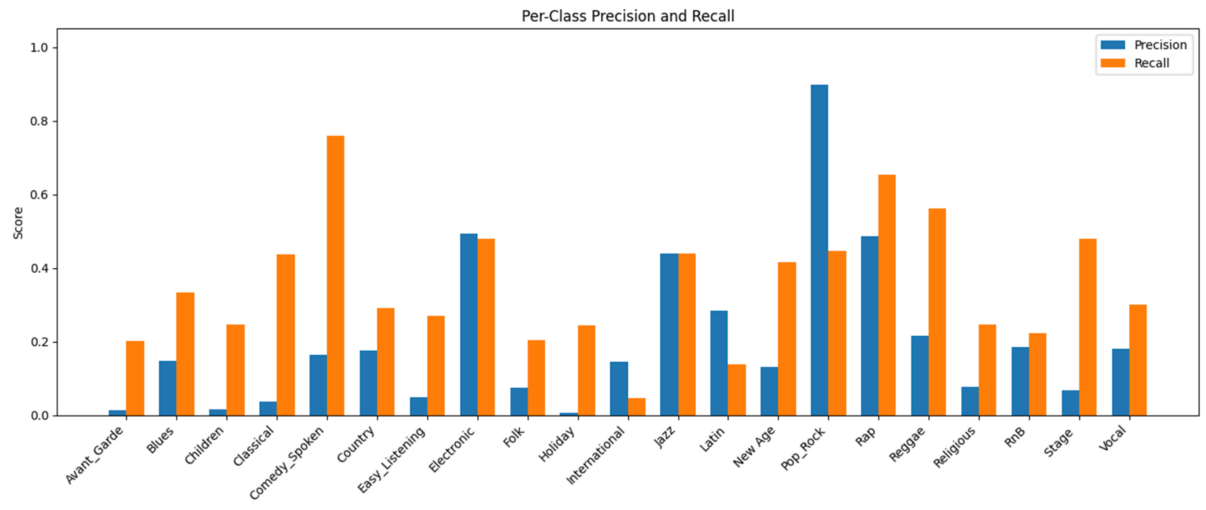
Class balance by genre		
Genre	**Count**	**Ratio**
Pop/Rock	237,641	56.50%
Electronic	40,662	9.67%
Rap	20,899	4.97%
Jazz	17,774	4.23%
Latin	17,503	4.16%

RnB	14,314	3.40%
International	14,193	3.37%
Country	11,691	2.78%
Religious	8,779	2.09%
Reggae	6,928	1.65%
Blues	6,800	1.62%
Vocal	6,182	1.47%
Folk	5,789	1.38%
New Age	4,000	0.95%
Comedy Spoken	2,067	0.49%
Stage	1,613	0.38%
Easy Listening	1,535	0.36%
Avant Garde	1,012	0.24%
Classical	555	0.13%
Children	463	0.11%
Holiday	200	0.05%

We now have an even starker difference in many class sizes compared to the binary classifier, with some classes barely being represented at all.

Before training the model, I split the dataset into training and testing sets using stratified sampling via Spark's *Window* function. Next, I chose to reweight the data due to the severe class imbalance present in the data. I reweighted by computing the inverse frequency of each class as a weight, dividing the total number of rows by the total number of classes multiplied by the number of rows for the specified class. Since the weighted model was the strongest performing **LinearRegression** model for binary classification, I elected to use it again here.
I also normalized the data since this is a **LogisticRegression** model.

For this model, the metrics I will be focusing on are *Precision* and *Recall*. Firstly, I calculated the macro-precision and macro-recall. Macro treats all classes equally and computes the average precision/recall scores across the model, which is ideal for an imbalanced dataset like ours. **The Macro-Precision was 0.6394 and the Macro-Recall was 0.4083**. So this shows that the model is still struggling in some areas to distinguish between genres.
Next, I plotted the *precision* and *recall* for each class:



Classes with fewer songs (e.g. *Holiday, Children, Avant-Garde*) have very low *precision*, meaning the model was consistently classifying other genres as belonging to these ones. This is not surprising, considering the fewer number of songs the model had to work with for the aforementioned classes. *Recall* scored higher than precision for the majority of genres, with *Pop/Rock* being the glaring exception. This likely means that many *Pop/Rock* songs were being misclassified as other genres due to their overwhelming frequency in the model's training data.
I also created a confusion matrix to see which genres were most frequently being misclassified as each other. The full confusion matrix can be found in the appendices, but here is some insights from it:

- *Pop/Rock* & *Electronic* were the most frequently misclassified genre pair, with 2,778 *Pop/Rock* songs misclassified as *Electronic* and 693 songs vice versa. These 2 genres make up over 65% of the total data, so it is unsurprising that they have the most overlap.
- *Pop/Rock* was also frequently misclassified as *RnB*, *Rap*, *Jazz* and *Reggae*, but relatively few of the songs from those genres were confused with *Pop/Rock*, suggesting some similarity in features.
- 637 *Pop/Rock* songs were misclassified as *Comedy Spoken*, but only 6 vice versa.
- *Latin* was also frequently confused with genres like *Reggae*, *RnB* and *Pop/Rock*, suggesting some similarity in features between those genres.

So the multiclass model performed significantly worse than the binary model. The multiclass model had to deal with multiple classes, some of which had very few datapoints to work with, which impacted key metrics like *precision* and *recall*.

Hyperparameter Tuning

One further aspect not really touched upon yet is hyperparameters. Each of the 3 algorithms used has a number of hyperparameters that can be configured to improve model performance. Here are some of the key ones retrieved from the Spark documents (*LogisticRegression/RandomForestClassifier/GBTClassifier — PySpark 4.0.0 Documentation*, 2025) :

Algorithm	Hyperparameter	Definition	Effect on Model
LogisticRegression	regParam	L2 (Ridge) regularization strength	Controls overfitting
	elasticNetParam	Mix between L1 (Lasoo) and L2 (Ridge) regulatization	Enables feature selection
	maxIter	Max number of training iterations	Higher values allow convergence, but increase training time
	tol	Convergence tolerance	Lower values increase precision but may slow convergence
	threshold	Decision threshold for classifying as positive	Adjusts precision/recall balance
RandomForestClassifier	numTrees	Number of decision trees	More trees increases stability/accuracy, but increases training time
GBTClassifier	maxIter	Number of trees	More trees improve fit, but too many can overfit
	stepSize	Learning rate per iteration	Lower size improves stability, but requires more iterations
RFC & GBT	maxDepth	Max depth of each tree	Controls model complexity, higher values may overfit
	maxBins	Bins for continuous feature splits	To handle categorical features, must be high
	minInstancesPerNode	Minimum examples to split a node	Reduces overfitting
	subsamplingRate	Fraction of data used to train each tree	Adds randomness to reduce overfitting

I did utilise some of these when training the **LogisticRegression** models, like *regParam* for the binary and multiclass models (*regValue=0.1*), but didn't utilize any aside from *weightsCol* for the other models. Overall, I'd say this was a sensible value

to use for a dataset with many features as it discourages reliance on any single variable and improves generalisation. Experimenting with a lower value could prove useful to reduce any potential underfitting.
My multiclass **LogisticRegression** model also utilized an *elasticNetParam* value of 0, meaning only *Ridge* regularization was applied. While this did help stabilise the model when multicollinearity was present, it did not encourage sparsity. This is arguably suboptimal due to the high feature count and the increased number of parameters in the multiclass model. With this knowledge, if I was to run the multiclass model again, I would experiment with a higher value (somewhere around 0.3-0.7), which would allow the model to discard redundant features.
For the other 2 models, it could have been worth introducing some hyperparameters to try and improve model performance. For example, increasing the number of trees for the **RandomForestClassifier** via *numTrees*. While this would be more computationally expensive, it could potentially reduce model variance.
Similarly for the **GBTClassifier**, increasing the number of boosting iterations via the *maxIter* hyperparameter could have improved overall accuracy, but at a cost for potential overfitting and increased computational complexity.

Another method to consider for improving model performance is cross validation, specifically **K-Fold Cross Validation**. This involves dividing the dataset into a predetermined number (K) of equal parts/folds. A single fold is set aside as a validation set, while the other K-1 folds are used as the training set for the model. Once trained, the model is validated using the validation set. This is then repeated K times with a different fold as the validation set for each iteration. Using this method results in a more robust evaluation of the model and is especially useful for tuning hyperparameters since you can pass multiple hyperparameter values to the *CrossValidator* and it will select the model with the best average performance, which in terms gives the optimal hyperparameters. (Fraidoon Omarzai, 2024)
If we were to engage in this for our multiclass model, you would first have to define a *ParamGridBuilder*, adding grids for each model hyperparameter you want to test and the values for that hyperparameter. Then you define a *MulticlassClassificationEvaluator*, specifying the metric you want evaluated (accuracy, weightedPrecision, weightedRecall etc.). Finally, a *CrossValidator* is initialized with the model, the *ParamGrid*, the evaluator and the number of folds you wish to use. Bear in mind that the more folds and hyperparameter values you select, the more computationally expensive it will be. (*CrossValidator — PySpark 3.5.4 Documentation*, 2025)
Doing so for our model could well have had an effect on metrics like *precision* and *recall*, possibly giving a much better balance between the two. It also helps that our **LogisticRegression** model is not too complex compared to many tree-based models, so it would potentially require fewer hyperparameters to tune, while also being quicker doing so.

Section 4: Song Recommendations

Finally, this report will tackle using Machine Learning to train a collaborative filtering model to recommend a song to a user based on the user's historic listening data. The dataset will be analysed to determine the best approach (accompanied by tables and visualizations of the dataset's distribution), then the data will be prepared before training is conducted. Finally, we will test and generate some metrics for the model to determine its performance.

Explore The Data

For this section, the **Taste Profile** dataset will be used, which is a tsv formatted data file containing **48,373,586** rows consisting of a *User ID*, *Song ID* and *Play Count*, which denotes the number of times a particular user has listened to a particular song. The dataset is roughly **488MB**, which means it will be computationally expensive to query.
The ALS algorithm (which we will be using later) utilizes iterative computation, shuffling data between nodes. Because of this, it is sensible for the dataset to be repartitioned so we reduce data shuffling and avoid bottlenecks due to incorrect partition sizing. Even though running the *repartition()* command is computationally expensive, it only has to be done once and will speed things up down the line. The dataset will be cached initially since we will be using it to generate statistics and visualizations, but once the dataset is filtered down to smaller datasets, we can release it from memory using *unpersist().* This is because such a large dataset will take up a lot of memory, which will need to be released once we no longer need to use the full dataset.
I elected to partition the data both on key columns (*User_ID* & *Song_ID*) and changing the number of partitions to reflect the number of instances and cores I was using. So I repartitioned the data to **16** partitions and also on the *User_ID* and *Song_ID* columns.

Before statistics were gathered, it was important to define what metrics I would be using to determine both song popularity and user activity. **I defined the most popular song as being the one with the highest number of total plays** (not the song with the most unique listeners) and **defined the most active user as the one with the highest number of total plays** (rather than the user who listened to the most unique songs).
Based on those definitions, here are some initial statistics:

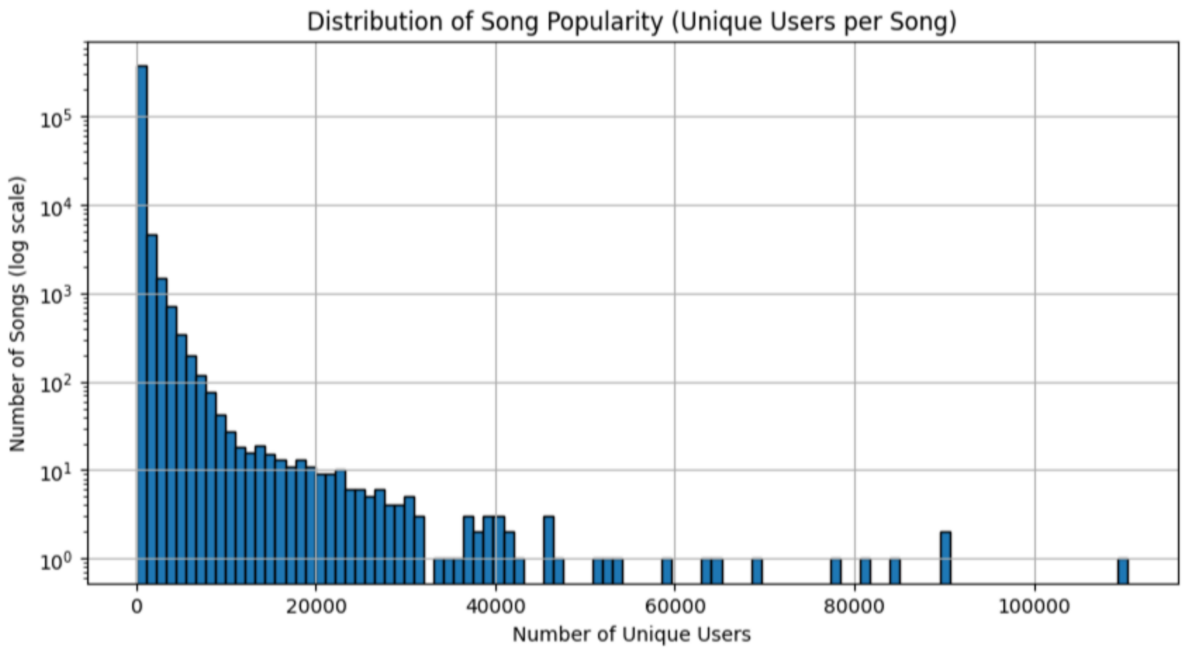
Taste Profile Dataset Statistics	
Count of Unique Songs	384,546
Count of Unique Users	1,019,318
Count of unique songs played by most active user	202
Percentage of total songs that were played by most active listener	0.05% (2dp)

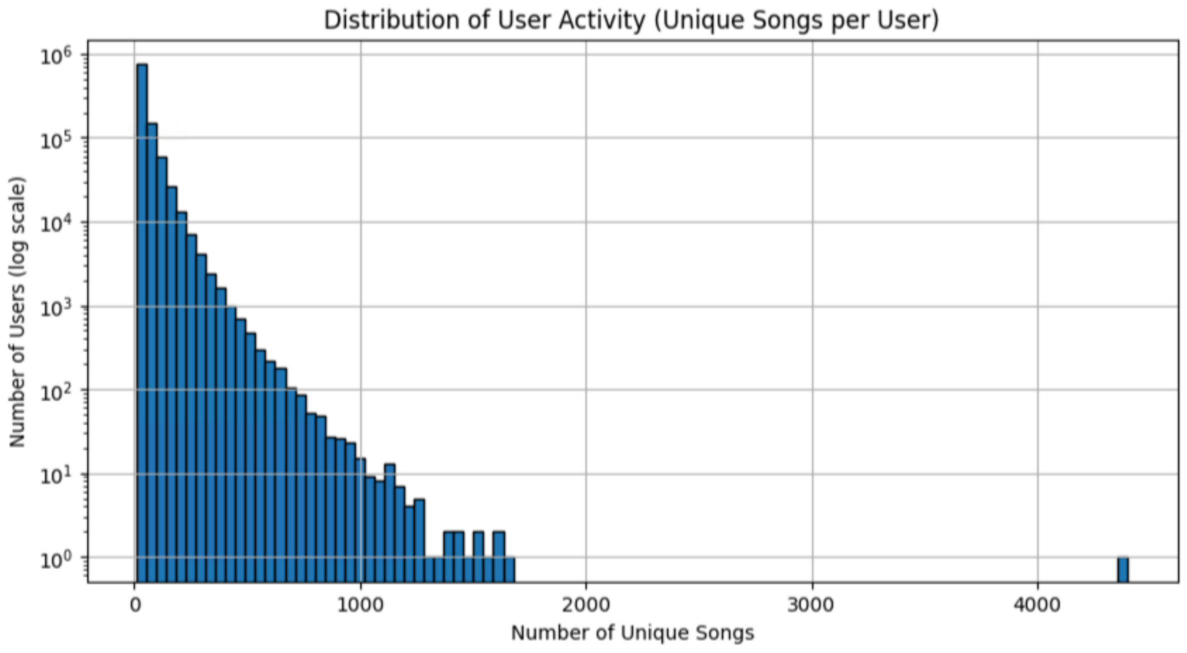
I also generated the following statistics to get an even deeper understanding of the data:

Additional Taste Profile Dataset Statistics	
Percentage of user/song combinations with >1 play	40.55%

Mean total plays for most active 10% of users	562 (rounded)
Mean total plays for least active 10% of users	15 (rounded)
Total plays across all users	138,680,243
Unique users per song (Mean)	126 (rounded)
Unique users per song (SD)	799.03 (2dp)
Unique users per song (Min)	1
Unique users per song (Max)	110,479
Unique songs per user (Mean)	47 (rounded)
Unique songs per user (SD)	57.82
Unique songs per user (Min)	10
Unique songs per user (Max)	4,400

Finally, I created 2 visualizations: one for the distribution of song popularity and one for the distribution of user activity, done using PyPlot. **NOTE: Even though I defined song popularity and user activity in terms of total number of plays, the assignment has specifically asked that these visualizations are done in terms of unique plays**.





I used the **log scale** for these visualizations due to the distribution, which is similar for both, being long-tailed. So the log scale was necessary to see the long tail of the distribution, since it shows us important information that aligns with the statistics we gathered previously.
For the **Song Popularity** visualization, we can see that the majority of songs are played by a relatively small proportion of users, while you have some evidently very popular songs getting vastly higher number of unique listeners.
The **User Activity** visualization shows a similar pattern, where most users only listen to a relatively small number of unique songs. There is a small subset clustered around the ~*1,500* song mark, then one outlying user listening to *4,000*+ unique songs.

Collaborative Filtering

Now that the data has been sufficiently explored, we can train a collaborative filtering model.
First, we need to discard rows of data that signify few interactions. This is because collaborative filtering needs density and overlapping user-item interactions, so having unpopular songs and inactive users in the data contribute little to the model training and may increase sparsity and noise.
The assignment suggests any song with less than 20 plays (taken to mean any song with less than 20 unique listeners) and any user that has played less than 20 songs should be filtered out. To do so, I first grouped the data by *song*, summed the total number of plays for each song, then filtered out any song with less than 20 overall plays. This resulting dataframe was then *inner joined* with the original dataset. Next, the filtered songs were grouped by *user* and each unique song was tallied for each user. Any user with a tally of less than 20 unique songs was also filtered out. Doing so results in the following metrics:

Dataset Filtering of unpopular songs & inactive users			
Metric	**Before**	**After**	**Reduction**
Users	1,019,318	647,808	~36%
Songs	384,546	161,173	~58%

The threshold of 20 appears to have been a suitable threshold to implement. The resulting reduction was large enough to remove noise, lower computational cost and improve model performance, while also not being so severe as to lose critical interaction information or population coverage.

From here, string indexers were used to convert the *user* and *song* identifiers into integer identifiers. This needs to happen because Spark's implementation of ALS does not accept string identifiers, only numeric types (*ALS — PySpark 3.2.1 Documentation*, n.d.). Doing this resulted in 2 new columns which identified the unique integer assigned to the row's user and song.

Splitting this dataset into *training* and *testing* sets required a different method to the classification models because every song in the test set has to have some user-song plays in the training set too. The reason for this is due to how ALS works. ALS factorizes the data into 2 different matrices, with predictions made via dot product. If a user is missing from the *training* data, no latent vector will exist for them. Thus, when it comes time to test, a prediction cannot be made and nothing will be returned. To ensure that the dataset is split in this manner as randomly as possible, a *window* was defined for each user and then ordered randomly. Each *user/song* interaction was then given a random number, then a row number. From here, the total number of interactions per user was calculated, then joined back onto each user's row. So now each row knows its position among the user's interactions and the total number of interaction for that user. Finally, a Boolean column was assigned to the data where the first 80% of rows (after random ordering) were assigned True. This allowed the data to be split randomly into *training* and *test* (80/20).

Once the ALS model was trained, it was time to use the *test* data to generate some song recommendations for users with the goal of seeing how accurate these predictions are. To do so, I first randomly selected 5 users, then used the model to generate 10 song recommendations for each of the 5 users. Then the actual songs the user played were retrieved from the *test* data, so we now had, for each of the 5 users, 10 recommendations generated by the model and a list of all songs the user actually played.
From here, I gathered the song title and artist from the **main/metadata** dataset in order to compare the relevance of the recommendations to the relevant songs. I then checked how many of the recommended songs the user had actually played. 2 users had 2 recommended songs that they had previous played, 1 had a single previously played song and the final 2 users had none.
The following is an example for one of the users, which also compares the first 10 relevant songs for the user:

Recommendation comparison for User db6a78c78c9239aba33861dae7611a6893fb27d5

Recommended songs (Artist/Song Title)	**Previously played by user**	**10 relevant songs (Artist/Song Title)**
Foreigner - Waiting For A Girl Like You	FALSE	Van Halen - One Foot Out The Door (Album Version)
Robert Johnson - I'm A Steady Rollin' Man	FALSE	1990s - Tell Me When You're Ready
Steely Dan - Reelin' In The Years	FALSE	Amy Winehouse - He Can Only Hold Her
Rihanna / Slash - ROCKSTAR 101	FALSE	The Clash - English Civil War
Eagles - Hotel California	FALSE	The Zombies - Woman
Led Zeppelin - Ten Years Gone (Album Version)	FALSE	Ashlee Simpson - Better Off
Simon Harris - Sample Track 2	FALSE	Lady GaGa - LoveGame
Tavares - Heaven Must Be Missing An Angel	**TRUE**	Rihanna / Sean Paul - Break It Off
Wolfmother - Woman	**TRUE**	The Killers - When You Were Young
Jonas Brothers - A Little Bit Longer	FALSE	Brain Damage, Just suddenly painful

As seen above, we had 2 songs recommended that the user had previously played. Aside from those 2, we can see that a Rihanna collaboration was played by the user and a different one was recommended.
While it would have been helpful to also have the genre of the song, upon checking the songs on Spotify, we can see that the relevant songs contains more Pop and Pop-adjacent music than the recommended songs, which typically lean more towards Rock. But overall, there is some level of overlap between the two and this is only based on 10 songs from the relevant song pool, so if the songs were compared against all relevant songs, we may see even more of an overlap.
So while it is difficult to make any definitive statements about the effectiveness of the model at this stage, I would argue that the recommended and relevant songs are more similar than they are dissimilar.

To further investigate the model's performance, we can generate statistics for **Precision @ K**, **NDCG** and **Mean Average Precision**. They are as follows:

Performance Metrics for ALS model (K=10)	
Precision @ K	0.12
NDCG	0.15193
Mean Average Precision	0.06122

Precision @ K is, on a technical level, "the ratio of correctly identified relevant items within the total recommended items inside the K-long list" (*Precision and Recall at K in Ranking and Recommendations*, n.d.). Essentially, it determines how many of the 10 recommended items are actually relevant. It's a simple metric to understand but one drawback is that it ignores the position of correct items and treats all successes in the top-K equally, which may be good for recommendations but not necessarily ranking of those recommendations. We achieved a P@K score of **0.12**, meaning 1.2 of the 10 recommended songs appeared in the relevant set, which is more or less what we saw based on the 5 users we generated recommendations for. This isn't a particularly good result and there is definitely room for improvement.
NDCG is somewhat similar to P@K except that it does take ranking positions into account, so it "compares rankings to an ideal order where all relevant items are at the top of the list" (*Normalized Discounted Cumulative Gain (NDCG) Explained*, n.d.). Because of this, it captures user attention behaviour because users will notice items ranked at the top more. The models NDCG score was **0.15193**, suggesting that some relevant songs were ranked highly, but a larger proportion were not.
Finally, **Mean Average Precision** is designed to measure both "the relevance of suggested items and how good the system is at placing more relevant items at the top" (*Mean Average Precision (MAP) in Ranking and Recommendations*, n.d.). It is particularly good for evaluating across multiple users, especially when there is variability in the number of relevant items. However, a consequence of this is that users with low activity tend to have a severe negative impact on the MAP score due to the lack of relevant items in the test set. Our model returned a value of **0.06122**, which indicates that that was likely a problem here.

If we wished to shift focus to the real world and undertake a similar test in the real world, we would need to conduct an **A/B test**. To do so, we would need 2 different models of a recommendation system. These models could be different in a number of ways. For example, *Model A* may have different hyperparameter values than *Model B*, or *Model A* may have been trained with implicit data and *Model B* with explicit data. The ultimate purpose is to see which model performs better by comparing key online and offline metrics.
Users would be randomly assigned to a model, which would feed recommendations to them. Over time, online metrics like **Click-Through Rate**, **Conversion Rate** and **Engagement Time** will be monitored. This will provide insight on which recommendations are interacted with by the user, if that interaction then resulted in feedback (e.g. purchase, rating etc) and which items resulted in the longest engagement times. Additionally, the historic data can be used to generate offline metrics like **P@K**, **NDCG** and **MAP**. These metrics can then be used to make a statistically significant statement about model performance, then incorporated into models for further testing.

Conclusion

This report has documented the intensive process of using Machine Learning algorithms to make decisions regarding classification and recommendation of songs. But this same process can be used for many other things like movies and online products. This report also shows the importance of understanding your data and the iterative process of tuning and adjusting parameters to create a model that works best for your data.

References

ALS — PySpark 3.2.1 documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.recommendation.ALS.html

Classification and regression - Spark 2.4.5 Documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/ml-classification-regression.html

CrossValidator — PySpark 3.5.4 documentation. (2025). Apache.org. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.tuning.CrossValidator.html

Data Types — PySpark 3.5.5 documentation. (2025). Apache.org. https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/data_types.html

The Echo Nest Taste Profile Subset | Million Song Dataset. (n.d.). Millionsongdataset.com. http://millionsongdataset.com/tasteprofile/

Fraidoon Omarzai. (2024, July 21). *Hyperparameters Tunning And Cross Validation In Depth*. Medium. https://medium.com/@fraidoonomarzai99/hyperparameters-tunning-and-cross-validation-in-depth-d0918b62d986

GBTClassifier — PySpark 3.5.3 documentation. (2024). Apache.org. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.GBTClassifier.html

LogisticRegression — PySpark 4.0.0 documentation. (2025). Apache.org. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.LogisticRegression.html#

McKee, H. (2023, April 28). *Loosely Couple Your Schema and Keep Data Separate*. DevOps.com. https://devops.com/loosely-couple-your-schema-and-keep-data-separate/

Mean Average Precision (MAP) in ranking and recommendations. (n.d.). www.evidentlyai.com. https://www.evidentlyai.com/ranking-metrics/mean-average-precision-map

Million Song Dataset Benchmark Downloads - IMP at IFS, UT Vienna. (2024). Tuwien.ac.at. https://www.ifs.tuwien.ac.at/mir/msd/download.html

Normalized Discounted Cumulative Gain (NDCG) explained. (n.d.). www.evidentlyai.com. https://www.evidentlyai.com/ranking-metrics/ndcg-metric

OpenAI. (2025). ChatGPT (V. 1.2025.133) [Large language model]. https://chat.openai.com/chat

Precision and recall at K in ranking and recommendations. (n.d.). www.evidentlyai.com. https://www.evidentlyai.com/ranking-metrics/precision-recall-at-k

RandomForestClassifier — PySpark 3.5.0 documentation. (n.d.). Spark.apache.org. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.RandomForestClassifier.html

Appendices

Multiclass Genre Classification Confusion Matrix

label	Pop_Rock	Electronic	Rap	Jazz	Latin	RnB	International	Country	Religious	Reggae	Blues	Vocal	Folk	New Age	Comedy_Spoken	Stage	Easy_Listening	Avant_Garde	Classical	Children	Holiday
Pop_Rock	21050	2784	1242	840	732	1669	310	2299	3875	993	1422	755	1902	1027	662	1190	712	2021	404	539	1101
Electronic	705	3795	712	226	74	261	53	42	147	565	112	34	83	267	118	289	94	344	61	67	84
Rap	92	340	2725	19	29	108	16	8	22	447	27	9	27	9	190	18	8	30	3	43	10
Jazz	84	153	23	1521	24	161	49	52	59	55	142	98	43	318	35	195	150	135	107	85	66
Latin	351	138	162	111	470	249	119	198	361	281	147	127	147	59	74	35	77	69	34	160	132
RnB	93	143	303	79	81	677	47	123	206	171	205	135	45	39	96	40	86	48	52	128	66
International	271	203	156	152	134	105	140	91	182	123	175	119	153	145	114	96	131	76	63	113	97
Country	205	11	11	60	32	57	39	680	241	14	164	119	173	54	81	42	87	19	42	46	162
Religious	305	33	56	30	31	137	20	111	434	37	42	70	62	49	50	42	32	45	41	32	97
Reggae	9	109	139	13	14	62	11	15	14	824	53	9	12	2	24	6	9	4	1	40	16
Blues	108	32	6	142	25	18	19	60	27	27	445	64	80	30	45	14	77	45	14	57	26
Vocal	21	5	0	107	13	28	9	71	52	17	46	331	57	29	93	34	52	15	92	64	101
Folk	62	14	2	58	19	23	22	80	48	17	56	110	235	69	68	41	67	20	33	44	70
New Age	26	31	8	83	7	2	10	3	8	2	3	9	13	327	4	115	34	38	58	5	15
Comedy_Spoken	8	1	7	3	2	7	0	13	1	9	5	4	8	0	328	1	5	3	4	3	2
Stage	4	7	4	10	0	0	2	1	4	3	6	12	9	40	7	154	13	11	26	2	8
Easy_Listening	9	4	0	24	2	4	2	10	10	2	13	18	13	20	8	28	88	12	30	4	7
Avant_Garde	14	12	3	9	1	3	0	1	2	0	6	3	7	24	3	27	14	50	6	13	5
Classical	2	1	0	0	0	0	0	2	4	0	1	6	2	14	5	13	2	2	49	1	8
Children	3	0	4	0	3	3	0	2	5	8	4	6	7	4	8	1	1	0	3	24	7
Holiday	3	1	1	3	0	1	0	3	0	1	4	2	1	1	3	3	2	1	2	3	6