Million Song Dataset Analysis using Machine Learning Models on Big Data

Stefanos Kalogerakis
Email:*skaloger@ics.forth.gr*
Computer Science Department, University Of Crete, Greece

Abstract: Music is often considered a reflection of the society and is a particularly interesting topic for researchers in order to examine the societal culture and value of each generation. For a human being it is relatively easy to determine whether a song belongs in a certain era or not, but for machines such problems are not trivial. Starting with the Million Song Dataset, a collection of audio features and metadata, performed evaluation on different classification algorithms and their ability to predict whether a song dates before or after the year 2000 and achieved a best score of 0.775 using the ROC-AUC metric.

I. INTRODUCTION

A. Problem Statement

In this project, the problem is to accurately determine whether a song dates before or after 2000. However, this is just one of the challenges concerning this implementation. The reason is that the Dataset consists files in binary .h5 format, and processing those files, especially in a distributed environment, is non-trivial. So the implementation focuses on parsing those binary files in a distributed manner with Spark, and the Machine Learning implementation is used more like a proof-of-concept.

B. Dataset

This project involves the use of a Dataset known as the **MillionSongDataset**. The Million Song Dataset is a freely available collection of audio features and metadata, including song tags, similar artists for a million popular music tracks. The raw Dataset is **280GB** and is available on AWS as a public dataset snapshot. For the purposes of that project, approximately 24GB were analyzed. Section *IV, AWS Environment Experience* contains more information regarding handling the Dataset in AWS.

The official documentation contains the available fields existing in each .h5 file. The available fields are 54 with the following data types:

- String
- Int, Float
- Array(Float)
- Array(Int)
- Array(String)
- 2D Array(Float)

II. PIPELINE AND STAGES

The overall high-level pipeline is shown in the figure 1. The formulation and the engineering part of the problem were crucial for several different reasons. In general, the divide and conquer paradigm is the best way to go when facing such multifaceted problems. Machine learning pipelines are notorious for taking a very long time to execute, let alone errors that might occur during the execution.

The implementation was built based on those principles and was divided into three main stages. Each stage is treated as a separate spark job. The first stage is the H5 file scraper, responsible for parsing and distributing the .h5 files to different workers to process. The second stage is responsible for data preprocessing, cleaning, and feature selection, whereas the third stage handles machine learning prediction.

In between those stages, the intermediate results are stored into parquet files that assist with testing and debugging as each stage is treated individually. Undoubtedly, the first stage is the most time-consuming as all the existing fields are passed into the next stage after performing light cleaning on unwanted data.

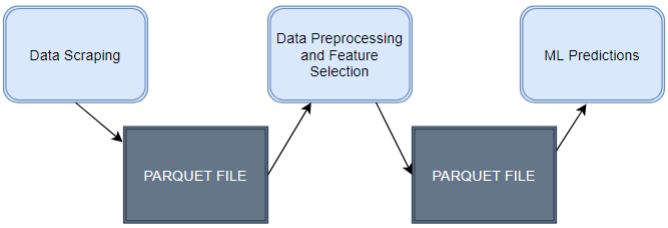

Figure 1: High-Level Pipeline Overview

In the following sections, all those stages will be analyzed in detail.

A. H5 file scraping

That is the first and most resource-demanding part of the pipeline. It is responsible for parsing and distributing the .h5 files in different spark workers to process in parallel while performing a light cleaning based on unwanted entries of the field year.

Before scrutinizing the implementation steps, It is important to note the libraries and resources used for that cause. The official MillionSongDataset site provides wrappers in different programming languages to perform analysis on those .h5 files. However, it seems that not all of them are currently supported, e.g., externally supported java libraries do not exist. The most suitable and stable choice was the python wrapper paired with the *pip tables* library.

The first objective was to clearly understand the structure of .h5 files, the existing fields, and their different types. So the first step was the creation of a .h5 file parser for a single file. The output is stored as a .csv file, a ubiquitous and human-readable format just to examine the fields. The next step is to try and parse multiple .h5 files at once. Since that initial attempt is non-distributed, the solution is trivial and simply requires a loop until all directories are parsed. This implementation was relatively simple yet very important for the data cleaning stage as it is possible to track problems in the existing fields easily.

One of the critical challenges on that project was parsing those files efficiently in a distributed manner. An important detail that influenced the final design decision is the file structure. Million Song Dataset is organized in a tree structure with thousands of directories and subdirectories. The desired .h5 files exist in the leaf nodes of that tree structure. In order to parse that structure, the main idea is to create a function that returns all the existing file paths as a list. After completing that step, using the parallelize function, Spark distributes different .h5 files to different workers. The final step requires mapping to a different function that handles the different .h5 files.

```
rdd = sparkContext.parallelize(filenames)
tr_rdd = rdd.map(lambda x: read_h5_to_tuple(x))
```

Listing 1: Pyspark code snippet that allows distributed parsing of .h5 files. **Filenames** is a complete list of paths where all the .h5 files are located for spark to parallelize. **Read_h5_to_tuple** handles each of the .h5 files individually

Handling each .h5 file can be performed in several ways. After experimenting with multiple options, including returning fields as lists, tuples, and writing every .h5 file in a different format, e.g. Avro, the best way to handle those files at that stage seems to be returning all fields as a tuple. Tuples are the closest representation to Dataframes and simply require the *createDataFrame function* with the tuple schema to produce the desired Dataframe.

This stage also includes assigning labels and a light cleaning of the data. Essentially, this process includes checking the field 'year' and assigning the corresponding label in case a song is released before 2000 or not. Cleaning involves invalid year cases where the specific song entry is omitted.

The final step requires choosing the most suitable format to store the produced results. Spark provides different format options to store and load files easily, so the most interesting aspect is the efficiency of those formats concerning processing time and file size. For that purpose, a profiling function was designed to evaluate the different formats.

	Parquet	Avro
Processing time(sec)	470.67	511.12
File Size(MB)	233,3	400,6

The results showcase that despite both Avro and parquet performing similarly for processing time, there is a considerable difference in each format compaction and consequently the final file size. CSV file format was not put to the test as it does not perform any compaction and would not certainly be the best candidate. As a result, the parquet format is utilized to store the intermediate results in the pipeline.

B. Data preprocessing and feature selection

This is the second stage which is responsible for data preprocessing, cleaning, and for feature selection. This stage changed a lot throughout the implementation, especially after experimenting with the machine learning prediction stage to improve the final models and predictions. The output of that stage is a Dataframe including the **features, labels, and weight columns** written in parquet.

In the following subsections, the Data Preprocessing and Feature Selection steps are examined.

1. Data preprocessing

As mentioned in a previous section, the Million Song Dataset contains 54 fields with different data types, including integers, strings, arrays etc. It is evident that different data types require different processing in order to produce features ready for the Machine Learning prediction stage.

For **float and integer** fields, the *MinMaxScaler* is applied to normalize those numeric features. An essential aspect of a machine learning model is to treat variables equally. It is prevalent for different variables to contain different units, which also applies in the Million Song Dataset. The MixMaxScaler function is a very common normalization step that rescales variables into a range [0,1]

For **string** fields, the *StringIndexer* is applied to convert strings into numeric vectors. Before applying the StringIndexer, however, it is vital to clean existing strings and remove the invalid ones. More specifically, the documentation states that some files contain symbols instead of strings, and it is essential to discard those files to avoid exceptions. Also, some entries may differ between different files due to accidental mistakes e.g. Add an extra space in the artist name. To avoid the previous problems, different functions apply to all the string fields with the following functionality:

- Check if a string is ASCII or not
- Remove all special characters
- Transform all to lowercase
- Remove spaces

For the **1D arrays of integers and floats**, different techniques and methods can apply. The main goal in those cases is to achieve dimensionality reduction and extract meaningful information from the arrays. The final method applied includes *sorting each array and extracting the max and min values.*

The **1D array of strings and 2D arrays** are not used in the final model for different reasons(see Feature Selection) and receive no special processing. For 2D features, the most suitable method for dimensionality reduction seems to be *PCA*. However, PCA is a very computationally expensive operation which may lead to more demanding resources. For that reason, this stage is omitted for future work.

After handling the different field types individually, use the *VectorAssembler* function that combines the selected features to a single column. The **'feature'** column is now ready to use in the machine learning models.

The final column added is the **'weight'** column. This column was added in the final steps of the machine learning analysis to investigate possible differences in the predictions after assigning weights to the different classes. This technique is used when classes are imbalanced to avoid overestimations in the majority class. The Million Song Dataset was relatively well balanced, but it was interesting to examine if there was a substantial impact on the final predictions using weights. The results showed that the difference between models with or without weights is negligible. The weight formula used to assign weights to the corresponding label classes is the following:

$$w_i = \frac{n}{n_i * C}$$

Figure 2: Weight Equation. C is the number of existing classes, n is the total number of observations and n_i is the number of observations in class i

2. Feature Selection

Feature selection is one of the most critical steps in a Machine Learning pipeline as it can significantly impact the performance of the models. At this stage, it is time to drop all the features that are not so informative.

During the initial efforts, feature selection was very naive, by simply choosing the features that seemed to be the best fit for the given problem. After researching more profound feature selection methods in later stages, two critical parameters came to the spotlight: **sparsity** and **correlation**.

Sparse features are considered those mainly containing zero values. Very sparse features add complexity without adding useful information to the model. Correlated features, on the other hand, will add noise and inaccuracy to the model, making it harder to achieve the desired outcome.

Taking those two parameters into consideration, it is time to reconsider and re-evaluate feature selection. For string data type, it is unlikely that their use can improve the final model, as most of the string fields are unique in the whole Dataset e.g. song_name, song_id. Also, in most cases, there is a correspondence between string and numerical fields. For array data type fields, higher dimensions imply more likely sparsity. Thus, only numerical features are considered in the final model, including max and min elements from 1D arrays of integers and floats.

After restricting the initial field choice, the next step is examining the correlation between those fields. Pearson's correlation method is used, and the plotted heatmap in figure 3 examines their pairwise correlation coefficient. This figure is mainly for demonstration purposes as it does not include all the desired fields, but it showcases the desired functionality.

It is obvious that features such as **'artist_familiarity'** and **'artist_hottness'** have a

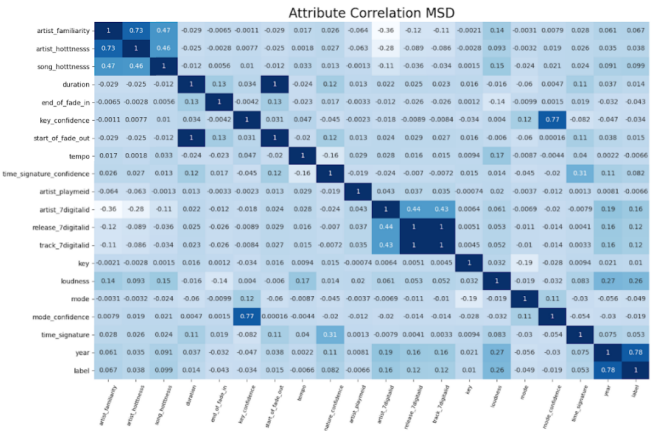

Figure 3: Pearson Correlation Heatmap

very high score of 0.73 which means they are highly correlated. So these two elements should not co-exist in the final feature selection. Following this method, after inspecting all the different fields and their correlations, **24 features** are finally selected.

III. MACHINE LEARNING PREDICTION

The final stage of the implementation is the machine learning prediction. This part was mainly considered as a proof-of-concept for the previous steps, yet it was interesting to investigate the different options that PySpark and MLlib provided.

As mentioned in previous section, the problem to solve was a binary classification problem and, more specifically, predicting whether a song was released before 2000 or not. Typically, when encountering such problems, datasets are usually heavily imbalanced. However, this is not the case in the Million Song Dataset, as both classes have close to equal distribution.

A. Picking an appropriate performance metric

Since the Dataset is relatively balanced, different performance metrics can be used to evaluate the performance models. The performance metric used in the implementation is ROC-AUC, a widespread performance metric for this class of problems.

The Receiver Operator Characteristic (ROC) curve is an evaluation metric for binary classification problems. It is a probability curve that plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings and essentially separates the 'signal' from the 'noise'. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. It quantifies the performance of a classifier to a single number and bounds it between 0 and 1. The closer the AUC is to 1, the better performance the classifier has. An essential feature of ROC-AUC is its ability to behave the same for both balanced and imbalanced classes.

B. Experimenting with models

In order to start experimenting with different models, one of the first actions is splitting the Dataset and creating a baseline model.

A very common way to split datasets is in three sets, **training, testing, and validation**. The main idea is to use the training set to train the models and map the input with the expected output, then use the model to predict the responses for the observations in the validation set, and finally use the test set to provide an unbiased evaluation. In the current implementation, the whole Dataset is split into the aforementioned sets using *60% for training, 20% for testing, and 20% for validation.*

The baseline model is the simple case that always predicts the same label. In that case, the ROC-AUC metric results in a score of **0.5**, which is expected as it is essentially random guessing.

Moving to the attempted models, this not too demanding machine learning problem allowed experimentation with almost all the suitable algorithms regarding classification provided by Spark and MLlib. More specifically, the following algorithms were utilized:

- Logistic Regression (Both with and without weights)
- Decision Tree Classifier
- Random Forest Classifier (Both with and without weights)
- Gradient Boosted Tree Classifier
- Naive Bayes
- Linear Support Vector Machine

The best results for each model are presented in the Results section, III C. Before moving to the results, It is important to address two additional aspects of the current machine learning pipeline: **hyperparameter tuning** and **k-fold cross-validation**.

In machine learning, hyperparameter tuning or optimization is the problem of choosing a set of optimal

hyperparameters for a learning algorithm. Hyperparameters are used to control the learning process. To evaluate the different models and tune their hyperparameters, both *manual Tuning* and *Grid Search* methods were used in most of the models. Manual Tuning involves experimenting with different parameters and observing the differences in scores without involving any automation in selecting parameters. Grid search, on the other hand, is used to identify the optimal hyperparameters for a model. This method uses a grid that contains sets of candidate hyperparameters, trains the model for every single set of them, and selects the best performing set of hyperparameters.

Regarding cross-validation, it is generally used to evaluate the skill of a machine learning model on unseen data. The k-fold validation procedure is as follows

- Perform random sampling to the Dataset
- Split the Dataset into k groups or folds
- For the groups:
  - Choose a group as a holdout or test data set
  - Take the remaining groups as a training data set
  - Fit a model on the training set and evaluate it on the test set
  - Retain the evaluation score and discard the model
  - Summarize the skill of the model using the sample of model evaluation scores

Figure 4: High-Level Pipeline Overview

C. Results

The following list contains the best predictions of each model on the validation set, except for Random Forest and Gradient Boosted Tree classifiers, in which case the predictions are on the test set. The reason is that the prediction scores in the validation set were very similar, so evaluate both of them in the test set. As mentioned in a previous section, the use of weights produced(as expected) negligible differences in comparison to the non-weighted models; that is why the results are not shown separately.

- Baseline, **Pred:** 0.5
- Random Forest, **Pred:** 0.755
- Linear Regression, **Pred:** 0.717
- Decision Tree, **Pred:** 0.687
- Naive Bayes, **Pred:** 0.429
- Linear Support for Vector Machines, **Pred:** 0.709
- Gradient Boosted Tree Classifier, **Pred:** 0.775

In the figure is the ROC Curve of the best model.

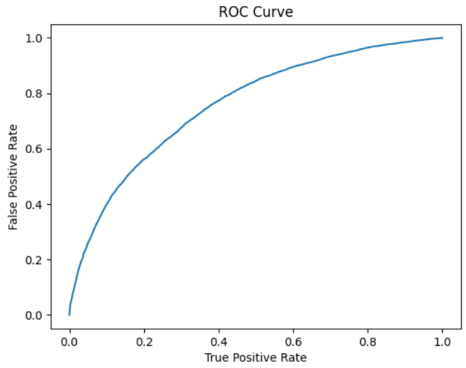

Figure 5: ROC Curve of the best model

IV. AWS ENVIRONMENT EXPERIENCE

The experience with Amazon Web Services (AWS) environment was not as smooth as expected, causing many different problems for several different reasons. AWS Setup section analyzes the services used for the purposes of that project, whereas the section Challenges with AWS highlights the different problems encountered during the deployment stage

A. AWS Setup

AWS provides plenty of services in the cloud that users can utilize to deploy applications adjusted to their needs. For this project, three AWS services were utilized. More specifically, *Amazon Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3), and Amazon Elastic MapReduce(EMR)* were used.

Before introducing the cluster setup, it is essential to note that the raw Million Song Dataset is available on AWS as a public dataset snapshot. So, to isolate the 24GB used for that project, the first step was to attach and mount the instance to an EC2 image. Then choose the desired data and copy them to an S3 bucket.

The final setup included an EMR cluster with the following features:

- Release 6.3.0
- Spark 3.1.1 version
- 1 Master Node, m5.2xlarge with 8 cores and 32GB memory
- 3 Slave Nodes, m5.2xlarge with 8 cores and 32GB memory

Additionally, to execute the code successfully, Spark must bind with python3 and not python2, which is the default setting. In order to achieve that, add the proper configuration in the Enter Configuration Settings while creating the cluster. Also, in the Bootstrap Actions section, add the script with the required dependencies to install them in all the different machines at once and avoid manual installation by hand.

B. Challenges with AWS

As mentioned at the start of this section, deployment in AWS was very time-consuming and not as easy as expected.

The first challenge was more or less expected, as python cannot create a package with dependencies like java. So the first attempts to execute code were unsuccessful because of missing dependencies. Tackling this problem was relatively easy as it required accessing all the different machines and installing the required dependencies. To make things even easier, the most efficient solution was to create a custom script with the required dependencies and attach it as a bootstrap action during cluster setup.

The second and most challenging problem was totally unexpected and was caused by inexperience with AWS Services. The problem was met during the very first stages of the Data Scraping Job. To be more precise, before distributing the .h5 files to workers fetching all the available file paths is required. However, the problem is that S3(where the Dataset is stored) is not a file system, but more like an object store, so it does not consider file paths as typical file systems would. Briefly, some of the efforts to tackle this problem were the following:

- Created new python functions using different parsing methods, in case it was a python problem.
- Utilized library boto3 natively in code, which downloads contents from s3 bucket to local node. However, this creates a new problem as the files exist only in the master node, and after distribution, workers cannot find the final files. Also, this library proved to be very inefficient and time-consuming.
- Attempted to download files in the master node. Encountered the same problem as the previous attempt.
- Used f3sf-fuse to mount the bucket to file directories individually. This method seemed to be the solution; however, it encountered the same problem as the previous two attempts despite mounting both master and slave nodes in the same directories.
- The only method that worked, which is very time-consuming and suboptimal, is downloading the whole Dataset locally in both master and slave nodes. It is almost certain that more efficient solutions exist to tackle that problem; however, time limitations enforced that kind of brute-force solution.

V. ACKNOWLEDGEMENT

- This project was implemented for the purposes of the postgraduate course Software Systems and Technologies for Big Data Applications(CS543), UoC, CSD
- I would like to express my gratitude towards teaching assistant G. Kalyvianakis and Prof. C. Kozanitis for their constant support to tackle each challenge that came up throughout the implementation

VI. REFERENCES

[1] https://machinelearningmastery.com/k-fold-cross-validation/.
[2] https://dhiraj-p-rai.medium.com/logistic-regression-in-spark-ml-8a95b5f5434c.
[3] https://elitedatascience.com/overfitting-in-machine-learning.
[4] https://dhiraj-p-rai.medium.com/logistic-regression-in-spark-ml-8a95b5f5434c.
[5] https://medium.com/@zxr.nju/how-to-evaluate-your-machine-learning-models-classification-evaluation-metrics-4670aef877ec.
[6] https://www.analyticsvidhya.com/blog/2020/10/improve-class-imbalance-class-weights/.
[7] https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/.
[8] https://medium.com/@connor.anderson_42477/hot-or-not-heatmaps-and-correlation-matrix-plots-940088fa28
[9] https://www.quora.com/What-are-the-best-ways-to-predict-datvalidation-and-test-sets.