Music Mood Classifier

EECE5645 Parallel Processing for Data Analytics
Final Project
Fall 2023

Gabriel Jentis
Sanjay Suman Senthil Geetha
Saurav Pallipadi Krishna
Sana Taghipour Anvari

Introduction and Problem Definition

Music is an artform that exists in almost every culture across the world, and one that contains a lot of passion and emotion. Spotify has a metric to quantify the emotion and mood of a song called valence. Valence, as defined by Spotify, is "a measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry)" [1]. From this it can be gathered that a song with a valence greater than 0.5 is of a positive mood and less than 0.5, a negative mood. In this project, a dataset from Kaggle [2] containing over 1 million songs as pulled from Spotify is used to classify whether a song is more of a positive or negative mood. The goals of this project are to generate binary classifiers for this task where positive songs are given a label of 1, and negative songs a label of 0 or -1.

Data Preparation and Preprocessing

There are two types of data present in this dataset: float metrics and data that is boolean or string that serves best to be binarized. Table 1 shows the features of the dataset that were chosen to be used, and which of the aforementioned categories they fell into:

Feature	Data Type or Binarize
Artist id	Binarize (String)
explicit	Boolean (binarize)
danceability	float
energy	float
key	Binarize (int)
loudness	float
mode	binarize
speechiness	float
accosticness	float
instrumentalness	Float
liveness	float
valence	Float→ to ±1 (Label)

Tempo (bpm)	float
duration_ms	int
time_signature	binarize
year	binarize

Table 1: Chosen features from the dataset

This list of 15 features as chosen from the dataset were handled in two separate ways. If a feature was set to be binarized, then for each nonbinary value of that feature, a unique binary feature would be created for training the classifiers. This would cause each song that was used as a datapoint to have the part of the binary feature associated with itself as a feature, and not the numerous other features derived from the binarized feature associated with it. For the data that was not binarized, there was a wide range of values that could fit for each of these features. Not using a way to normalize this data would skew the results of the classifiers. To address this, for each feature containing a float or int, the z-score was calculated for that feature for each song and used as opposed to the actual datapoint. The z-score was calculated with the following formula:

$$z = \frac{x-\mu}{\sigma}$$

This allowed the values of these features of the dataset to be more evenly distributed, when they could have been very skewed to a single extreme. For the generation of these classifiers, a subset of 50,000 of the 1 million songs in the original dataset were taken. In this subset the breakdown between the labels is as follows (Table 2):

Positive (1)	20700
Negative (0 or -1)	29300

Table 2: Breakdown of number of songs in each label class

When the full dataset is created with these 50,000 songs, it is found that while the dataset is sparse, there are 8,416 total features.

Classification Approaches

To perform this classification three approaches were explored:

1. Logistic regression
2. Decision Tree
3. Random Forest

Logistic regression

With logistic regression, the goal is to minimize the loss function:

$$L(\beta) = \sum_{i=1}^{n} log(1 + e^{-y_i \beta^T x_i}) + \lambda ||\beta||_2^2$$

This loss function was minimized using gradient descent, where the gradient of the loss is:

$$\nabla L(\beta) = \sum_{i=1}^{n} \frac{-y_i x_i}{(1+e^{-y_i \beta^T x_i})} + 2\lambda\beta$$

On each descent iteration, backtracking line search was used to determine the next values of β. This continued until either the gradient reached a specified threshold, by default $1x10^{-20}$, or after a specified number of iterations. This approach used k-fold cross validation to get a better understanding of the accuracy, precision, and recall metrics for each set. A number of folds is specified at run, and the program maintains the value of β across folds, both to provide a better starting beta than the zero vector, and to maintain knowledge of features that may not be present in the current training folds. The value of the regularization parameter λ was found through successive trials to find which led to the greatest accuracy when averaging the accuracy on each the test folds. This value for lambda was then used for a longer final training.

Decision Tree

Decision tree algorithm is a supervised machine learning technique in which data is continuously partitioned. A decision tree is a flowchart-like tree structure consisting of nodes and is used for making decisions. Decision tree consists of the root node which stands for the complete dataset, Branch or internal nodes, which are basically the decision points and represent test on attributes, and Leaf nodes, which represent the outcome and final prediction. The basic algorithm for decision tree is that first, the whole training set would be considered as the root, find and choose the best attribute using Attribute Selection Measures (ASM) for splitting, then make the selected attribute a decision node and partition the dataset into subsets, start building tree by repeating this process recursively until:

1. No remaining attributes are left.
2. No more instances are left.
3. All the subset of training dataset belong to the same feature value.

Attribute selection measure or splitting rules provides a rank to each attribute in the given dataset, then selects the best score attribute as the splitting attribute. Gini Index and information Gain are two attribute selection measures used to select from n features in a dataset. Gini Index is used to measure how often a randomly chosen element would be incorrectly predicted, thus the attributes with lower Gini index are preferred and it is defined as:

$$Gini(D) = 1 - \sum_{i=1}^{m} P_i$$

Where P_i is the probability of an element being classified to a particular class. Entropy, used in another method in making decision in decision trees and that is basically measuring of disorder and impurity, and it is defined as:

$$E = - \sum_{i=1}^{m} P_i \log_2 (P_i)$$

Where P_i is again the probability of an element being classified to a particular class. Information Gain computes the differences between entropy before the split and the average entropy after the split.

$$Gain = E_{parent} - E_{children}$$

Information gain helps us to determine how good the split is, and the more entropy removed, the better information gain would be.

In our decision tree python code, we used the default splitting criterion in Spark's MLlib API which is the Gini impurity. Here we visualized the decision tree model that we have trained over the "500" first rows of our dataset:

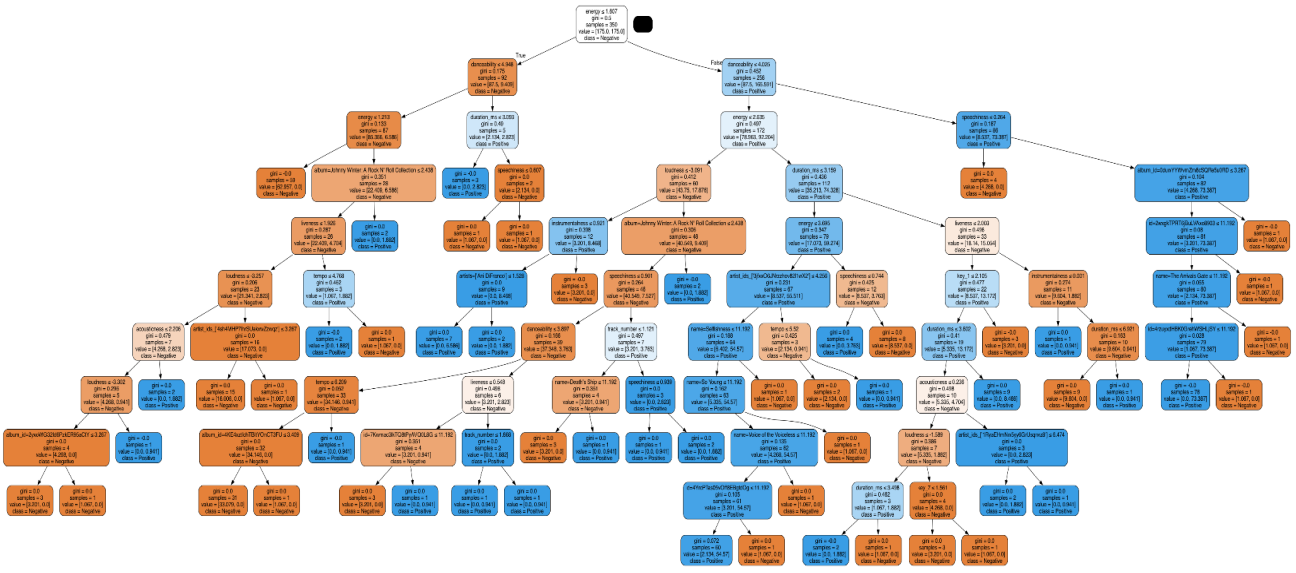

Figure 1: Representation of the Decision Tree model implemented on 500 first rows of our Dataset

Random Forest

Random Forest is an ensemble learning method which combines predictions from multiple machine learning models to make more accurate predictions when compared to a single model. It is usually constructed by combining a multitude of decision trees at training time and providing

the class that has the majority vote. Each tree present in the random forest is built using a sample drawn with replacement from the training set. In a classification setting like our project, the equation of the prediction of a random forest is given by Y = mode{y1,y2,...yT} where y1,y2…yT are predictions made by each of the T decision trees and mode corresponds to the most frequent label in the set. A Random Forest model helps in generalizing the model and smoothing out the predictions because it takes into account the results of all decision trees as opposed to one decision tree which can be prone to overfitting.

To train the Random Forest model in our code, we used RandomForestClassifier[3] which is a part of Apache Spark's MLlib library. It is a popular algorithm used for classification tasks, leveraging the power of a Random Forest in a distributed computing environment. It takes in two mandatory parameters featuresCol and labelsCol where featuresCol is fed with the list of all the features present in the dataset and labelsCol is provided with the label we are trying to predict (valence in this case). It has other optional parameters that may serve useful in fine-tuning the model like numTrees and maxDepth where numTrees determines the number of trees that will constitute the forest and maxDepth determines the depth of each tree in the forest.

The SparseVector representation of the dataset utilized for Logistic Regression and Decision Tree had to be modified to use the DataFrame based Mllib module so the preprocessed sparse dataset was converted into a dataframe and the missing features in each row were populated with 0's to maintain uniformity.

Classifier Results And Performance

Logistic Regression

To observe the results of the logistic regression classifier, the first goal was to find an optimal value for the regularization parameter λ. This was done by performing multiple runs of the program with different values of beta. To observe the impact of the change in this parameter, the number of folds was held constant at 4 and the maximum number of iterations for each fold was held constant at 20. The average cross validation accuracies, precisions, and recalls were then plotted to determine an optimal value for λ (Figure 2).

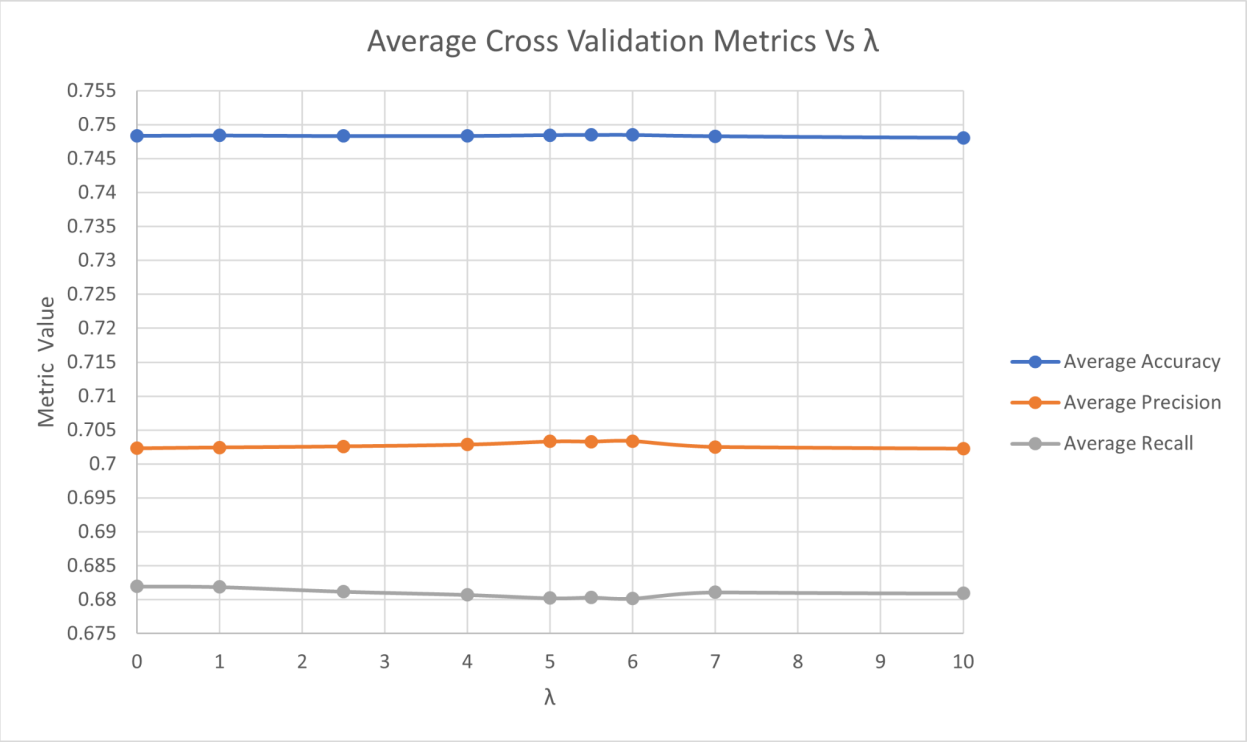

Figure 2: Average Cross Validation Metrics for 4 Folds and 20 Iterations per Fold

As can be seen from this plot, the average cross validation accuracy, precision, and recall are fairly consistent for λ between 0 and 10. There was also a trial with $\lambda = 100$, which is not on this plot for axes display sakes, but its average accuracy was 0.745, which is worse than the runs where $\lambda \leq 10$. Based on this analysis, it was determined that $\lambda = 6$ would be the best value to choose as it resulted in the highest accuracy and precision of all tested values for λ. With $\lambda = 6$, another 4-fold run was performed with a maximum of 50 iterations per fold to get a final classifier. The results of this run were:

Average Accuracy	Average Precision	Average Recall
0.7521	0.7067	0.6878

Table 3: Results of 4 Fold, 50 Iteration Logistic Regression Train

Using the feature vector, β, resulting from this train, the classifier was tested over the entire 50,000 song dataset to get a final set of metrics. The resulting metrics from this run were:

Accuracy	Precision	Recall
0.7603	0.7203	0.6881

Table 4: Results of Feature Vector From 50 Iteration Train Applied on Entire Dataset.

Overall, this accuracy in the range of 0.76 is fairly moderate performance for this classifier. It is far from a perfect classifier, but it performs much better than randomly assigning a label to a datapoint. This classifier also tends toward a slightly higher precision as compared to recall. While there could be more improvements to this classifier, the results do show that this classifier was able to generate meaningful results for the given dataset.

The performance of this method was explored to determine the impact parallelism had with this classifier, the number of partitions used during training was changed through multiple runs. To test the impact, the training was run with different numbers of partitions, 4 folds, a regularization parameter of $\lambda = 0$, and a maximum of 20 iterations per training fold. To test the impact of parallelism, the average time per training iteration across all folds was taken (Figure 3):

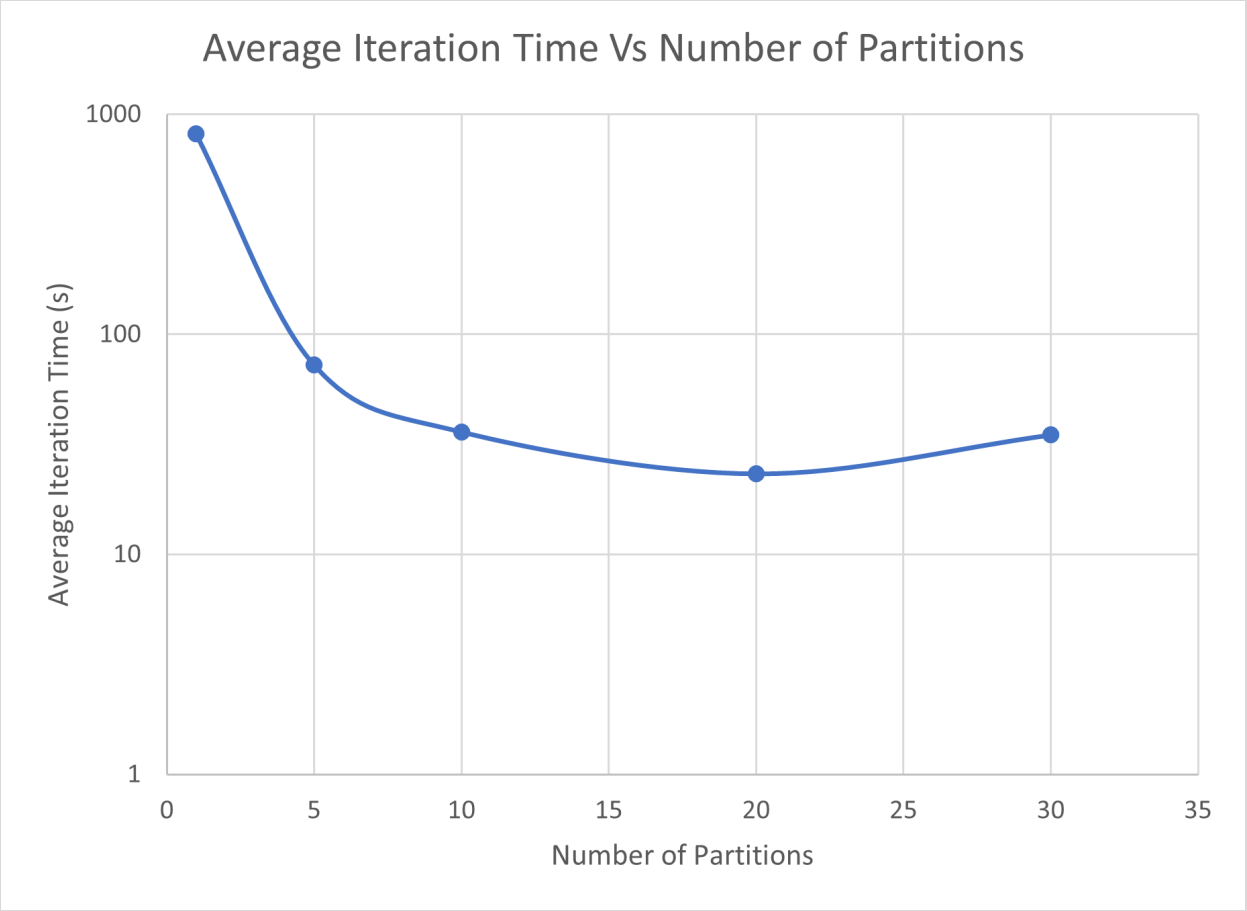

Figure 3: Average Time per Iteration for Logistic Regression Classifier Based on Number of Partitions

As can be seen in the above figure, parallelism significantly decreased the time per iteration as opposed to no parallelism. The plot itself uses a logarithmic y-axis because the average time per iteration with a single partition, or no parallelism, took over 800 seconds per iteration. Increasing to just 5 partitions reduced that timing to around 70 seconds per iteration. The best number of partitions found was 20 partitions, which led to each iteration taking on average 23 seconds. When the number of partitions increased to 30, the time per iteration increased, which can be

explained as 30 partitions surpassed the number of CPUs on the node in the Courses partition on the Discovery Cluster. Overall, parallelism had a significant impact on improving the run time of the training of the logistic regression classifier.

Decision Tree

For performing DecisionTreeClassifier on our dataset, First, we ensured that the data is in the correct format, by following the logic defined in the data preprocessing section. Then, the LabeledPoint class, which is part of MLlib's RDD-based API, is used to represent a data point with features and a label, then convert dataframe into RDDs in which features are represented as a SparseVector. Lastly, the code handles the training of the Decision Tree Classifier and evaluates its performance, as well as implementing k-fold cross-validation with 4 folds.

The differences in how well this Decision Tree algorithm works when run in parallel versus when it's not, are clear at various steps of the process. Utilizing Spark's MLlib API , the parallelization of the Decision Tree algorithm is effectively achieved. This technique involves simultaneously training multiple subtrees, with the number of trees being trained in parallel tailored periodically to accommodate memory availability. Examining the differences at different stages of DecisionTreeClassifier, with (N = 5) and without parallelism is summarized in the below table.

Event	Without Parallelism (s)	With Parallelism (s)
Data Preprocessing/ normalizing	26.73976755142212	24.325247049331665
Calculating AUC Score	1.4092652797698975	0.7534608840942383
Training Fold 1	2.7555816173553467	2.381197214126587
Testing Fold 1	0.05280017852783203	0.03803610801696777
Training Fold 2	1.8166618347167969	1.0642571449279785
Testing Fold 2	0.03316164016723633	0.0235898494720459
Training Fold 3	1.6940052509307861	1.05751371383667
Testing Fold 3	0.04472064971923828	0.0300295352935791
Training Fold 4	1.7145450115203857	1.0498180389404297
Testing Fold 4	0.03363800048828125	0.0235650539398193

Table 5: Time difference at different stages of of the program of a decision tree algorithm with and without parallelism

The comparison reveals significant time savings with parallelization. The average training time of the Decision Tree algorithm between 4 folds is 1.6 times faster with parallelism, and the average testing time between 4 folds is 1.55 times faster with parallelism.

Furthermore, here is how the Total execution time change by choosing different number of partitions:

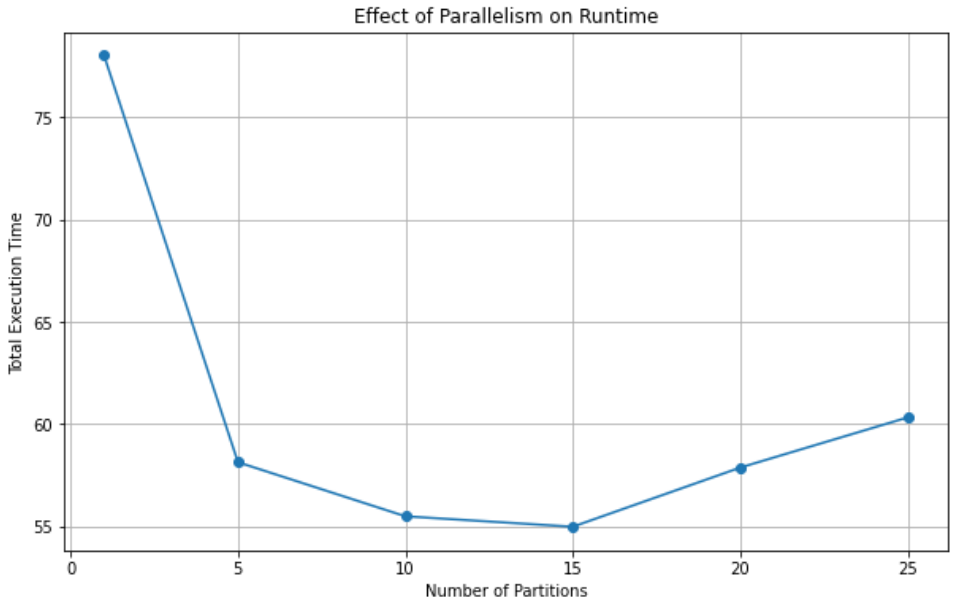

Figure 4:Result of parallelism over runtime by choosing different number of partitions.

The parameters maxDepth (the maximum depth of the tree, number of levels of decision nodes), and maxBins (maximum number of bins used for splitting features) are important hyperparameters in the configuration of decision tree algorithms, and in determining the model's performance and complexity. In our decision tree code, these parameters can be passed to the classifier from the user. Here we see how accuracy changes by changing these parameters:

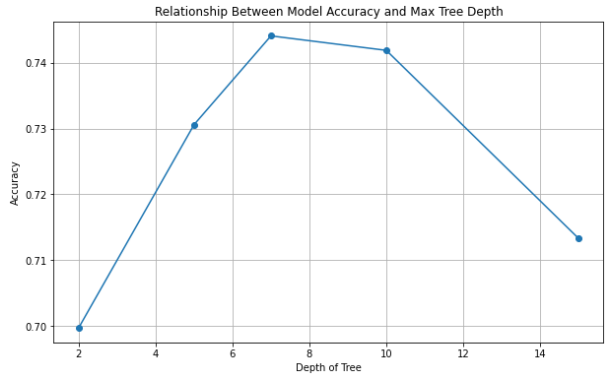

Figure 5: Effect of Maximum tree depth on Accuracy

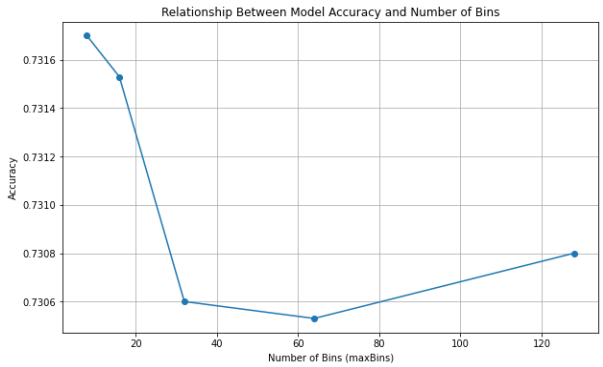

Figure 6: Effect of maximum number of bins on accuracy

As we can see from the results, maxDepth and maxBins are crucial for controlling the decision tree's complexity and fitting ability, and they should be carefully tuned to achieve the best performance on the specific dataset that we are working with. A higher maxBins value allows the algorithm to consider more split points and has the potential to gain more precise trees, but it also increases computational complexity. Also, setting maxDepth too high can lead to overfitting.

To evaluate the performance and efficiency of this decision tree classifier, we measured the mean accuracy, recall, precision, and the validation score over the folds:

Accuracy	Precision	Recall
0.7441747508888981	0.6949649043353607	0.6805885468761852

Table 6: Result of 4 fold Decision Tree Classifier

Cross-Validation Scores: [0.7491036505867015, 0.739998402938593, 0.7389714649074001, 0.7486254851228978]

AUC Scores: [0.7272964256302552, 0.7181106810546442, 0.7214883371962649, 0.7262018305439503]

The accuracy, precision and recall suggest that the model is performing reasonably well on making accurate predictions. If we consider the cross validation score, the numbers indicate that the Decision Tree model's performance is relatively consistent across different subsets of the data, plus, consistency is a positive sign and suggests that this model is not overfitting to the training data. The AUC scores range from approximately 0.718 to 0.727, indicating that this model has a moderate ability to distinguish between classes. In summary, This Decision Tree model's performance is decent, but there is potential for enhancement, especially in terms of recall and discrimination ability.

Random Forest

To measure the effectiveness of the Random Forest model, there were two hyperparameters that proved to be significant in terms of raising the accuracy of the predictions. Increasing the number of trees present in the random forest model up to a point allows us to take into account more trees which results in generalization of the model. Tuning the maximum tree depth parameter of each tree allows us to adjust the complexity of the tree and the random forest model will be able to make better decisions with higher depth until it leads to overfitting.

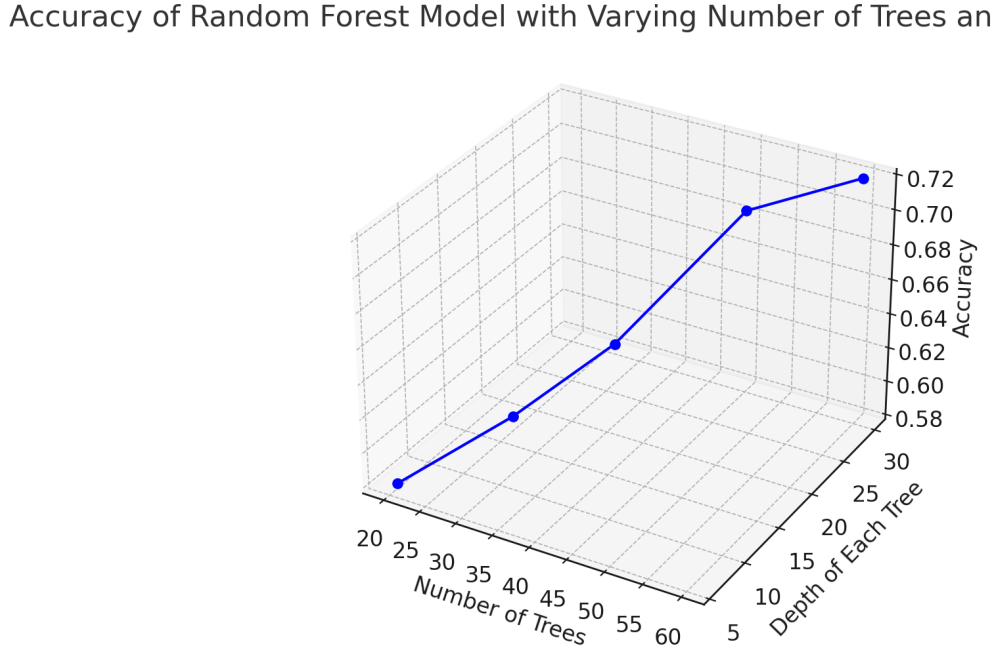

Figure 7: 3-D plot visualizing the accuracy as opposed to Number of trees and Depth of each tree

As seen in Figure 7 we varied the number of trees in the random forest along with the depth of each tree to measure the performance of the random forest model. We started by setting the number of trees to 20 and maximum depth to 6. The number of trees was incremented in multiples of 10 and the maximum depth was incremented in multiples of 6 which resulted in a steady increase of model accuracy at each increment. A 1% increase (to 72%) was seen in the final increment with the number of trees set to 60 and maximum depth set to 30.

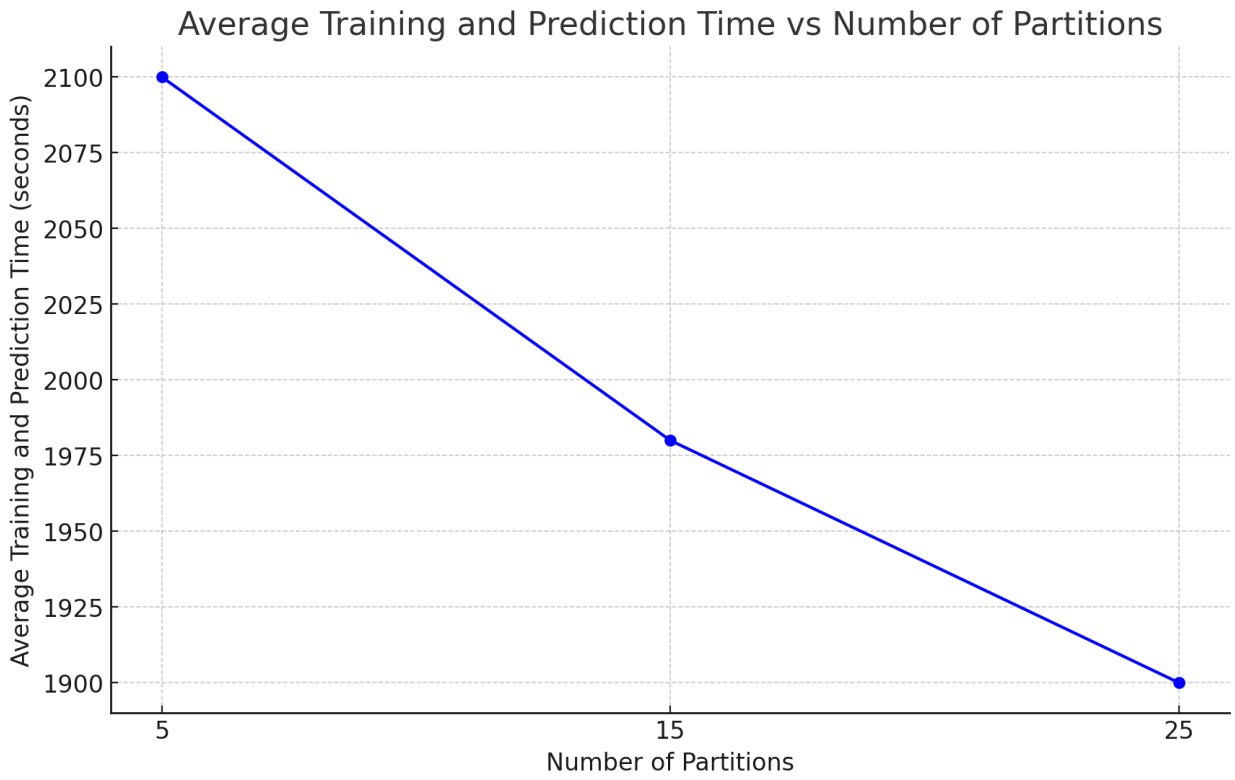

Figure 8: Average Time per run for Random Forest Classifier Based on Number of Partitions

Once the best hyperparameters were established, we measured the impact of parallelism on the random forest model. MLlib utilizes Spark's core feature of distributed data processing. When an ML algorithm is applied to a DataFrame, the data is processed in parallel across the cluster. The DataFrame is divided into partitions, and each partition is processed by different nodes in the cluster. As the number of partitions increased the time taken to finish the run gradually decreased (Figure 8). Although, because of the volume of the dense matrix that MLlib's data frame method requires, the change in time taken to finish a run across the number of partitions isn't substantial.

Conclusion

In this project, three methods were used to classify the mood of a song as defined by its valence from the Kaggle dataset of Spotify songs. Throughout the development and experimentation of these methods, the accuracy of the classification was tracked, and the final accuracies per method were:

Logistic Regression	Decision Tree	Random Forest
0.7603	0.74417	0.72

Table 7: Accuracies of The Three Classifiers Built For The Dataset

From this it can be seen that logistic regression performed the best of the three classifiers developed from the perspective of classifier accuracy. There is plenty of future work that can be performed with this project. As mentioned earlier, a subset of only 50,000 of the over 1 million songs available in the Kaggle dataset were used. Future experimentation with more time and machine power could explore using all the songs, and potentially extracting more possible features directly from Spotify's API. There is also work that can be done in improving the implementations of these classifiers, both in performing finer tuning of the parameters and optimizing the algorithms themselves.

References

[1] https://developer.spotify.com/documentation/web-api/reference/get-audio-features
[2] https://www.kaggle.com/datasets/rodolfofigueroa/spotify-12m-songs
[3]https://spark.apache.org/docs/3.1.3/api/python/reference/api/pyspark.ml.classification.RandomForestClassifier

Group Member Tasks And Other Notes

Gabriel Jentis wrote the majority of the base script used for data preprocessing. He also did all implementation and testing of the Logistic Regression classifier approach. The Logistic Regression code is based off of the code written in Homework 3, but includes some changes to support the cross validation done as well as maintaining the β vector across folds.

Saurav Pallipadi Krishna revamped the sparse dataset into a full rank dataframe to use it in MLlib's Random Forest Classifier built for dataframes. He also trained and tested the Random Forest classifier using different hyperparameters and multiple combinations of memory and number of executors.

Sanjay Suman Senthil Geetha also worked on building the Random Forest Classifier model. Utilized the full-rank data frame to be used on the model and experimented it with different parameters and configurations to optimize the model's performance. Collaborated on integrating the model into the overall project architecture.

Sana Taghipour Anvari Preprocessed the dataset for Decision Tree Classification, wrote the code required to apply the dataset to the Decision Tree using MLlib's Decision Tree Classifier, also handled the implementation of both training and testing phases for the Decision Tree Classifier.