CS-UA 473
Fundamentals of Machine Learning
Spring Semester 2024
Capstone Project
Classification Flavor

Name: Firas Darwish
NetID: fbd2014

Introduction:

In this report, we demonstrate the various design choices and techniques used in the process of building a music classification model based on data extracted using Spotify's API. We first describe all pre-processing decisions made (normalization, handling missing data, one-hot encoding, etc.); as these pre-processing techniques are used repeatedly across all models that we build and present, we segment this into its own introductory section. Secondly, we present the working of building a series of classification models using all the remaining features and evaluating the success of these models. Thirdly, we repeat this process, this time adding a relevant dimensionality reduction technique and extracting key insights by clustering our data in the low dimensional space. Finally, we exhibit any additional and supplementary observations that came out during this exploration.

Pre-Processing:

There are several significant pre-processing choices that we undertook that precede all upcoming analysis that we will outline and explicate here:

Dropping Irrelevant Features:
We decide to drop certain features from our dataset due to their noisy contribution to our classification or their incompatibility with our classification models.

We first drop both 'instance_id', which is the "unique Spotify ID of each song" as well as 'obtained_date', which represents "when… this information [was] obtained from Spotify". These features are simply not an attribute or property of the song they are assigned to but rather are metadata associated with Spotify's API when utilized for music data retrieval. In the case of the 'obtained_date' feature, it is likely to add noise to our classification models as it is indicative of when the designer of the dataset retrieved these songs using Spotify's API. In the case of 'instance_id', we look to the documentation associated with Spotify's API (Spotify for Developers) to find that "a Spotify ID does not clearly identify the type of resource"; using the 'instance_id' would risk injecting noise to our predictors as it is not linked to a song's attribute but is a randomly-generated and non-explanatory variable to help identify song's distinctively/uniquely.

Additionally, we decide to drop both 'artist_name' and 'track_name'. These two predictors provide rich linguistic properties of a song and could be analyzed further to extract sentiment or semantic features from the songs to help with classifying the song's genre. Using a sentiment analyzer, for example, to uncover the emotion evoked by a song's title could be an effective indicator of that music piece's genre. Nevertheless, for the models we build in this report, we avoid the use of these lingual attributes.

Encoding:
Encoding is very important, given the way in which certain features are presented in the original dataset.

The key of the song is originally reported in musical notation ('C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'B'). This data is reported as ordinal data, which is when you can "categorize and rank your data in an order, but you cannot say anything about the intervals between the rankings"

(source). To dig deeper into the key of a song, each key corresponds to a frequency range, which is depicted in Figure 1 (source), depending on the octave at which the note is played (different pitches). We see that there is certainly an underlying ranking and ordering to the key of a song, seeing as it dependent on frequency; however, there is not necessarily an 'equal interval' between each key such that we can classify this type of data as being interval. Based on the outcome of this, we decide to encode our 'key' feature using ordinal encoding where C corresponds to 0 and B corresponds to 11 (and, thus, the keys are ordered based on their frequency range they belong to). One-hot encoding this type of data takes away from this natural ordering. Instead, a type of ordinal encoding is necessary to preserve the natural order of music's keys. By assigning 0 to the key of songs with C (low frequency), 1 for those with the C# key, and 5 for those with the F key, we maintain the order of these categories while preserving the 'closeness' of the C and C# keys (frequency-wise (versus those with the F key) due to the proximity of the encodings those categories have been assigned (0 and 1 are closer to one another than they are to 5). We also communicate the information that those with the C and those with the C# keys are closer to the 'extreme' of the key feature (on the lower end of the frequency spectrum) than those in the F category. This information is principally lost if we one-hot encode ordinal data.

	0	1	2	3	4	5	6	7	8
C	16.35	32.7	65.41	130.81	261.63	523.25	1046.5	2093	4186
C#	17.32	34.65	69.3	138.59	277.18	554.37	1108.73	2217.46	4434.92
D	18.35	36.71	73.42	146.83	293.66	587.33	1174.66	2349.32	4698.63
D#	19.45	38.89	77.78	155.56	311.13	622.25	1244.51	2489	4978
E	20.6	41.2	82.41	164.81	329.63	659.25	1318.51	2637	5274
F	21.83	43.65	87.31	174.61	349.23	698.46	1396.91	2793.83	5587.65
F#	23.12	46.25	92.5	185	369.99	739.99	1479.98	2959.96	5919.91
G	24.5	49	98	196	392	783.99	1567.98	3135.96	6271.93
G#	25.96	51.91	103.83	207.65	415.3	830.61	1661.22	3322.44	6644.88
A	27.5	55	110	220	440	880	1760	3520	7040
A#	29.14	58.27	116.54	233.08	466.16	932.33	1864.66	3729.31	7458.62
B	30.87	61.74	123.47	246.94	493.88	987.77	1975.53	3951	7902.13

the values are in Hertz (Hz), the top row represents the octave (from 0 to 8)

Figure 1: Musical Key against Octave, Frequency (Hz) Correspondence

The mode of the song is originally presented as 'Major' and 'Minor'. To help clarify our results, we convert this to a one-hot encoded variable where 'Major' corresponds to 1 and 'Minor' corresponds to 0.

Furthermore, we also alter the 'music_genre' data depending on the classification technique we are utilizing. The 'music_genre' data is a type of nominal data, where "[you] can categorize your data by labelling them in mutually exclusive groups, but there is no order between the categories" (source). When using classifiers like SVM, Trees, Random Forests, and AdaBoost,

we keep the labels as they are in the original set (a string indicative of the genre of the song), as the libraries and packages we will be using accept such an output. However, when we are defining neural networks (with a loss function), we change the 'music_genre' column to contain vectors of dimensionality 10 (as we have 10 music genres, so 10 classes), where all the elements of each vector are 0 except at the index corresponding to the music genre that the song actually is (in which case the element at that index is set to 1 in the vector). This allows us to more easily compute our loss function by subtracting the 10-dimensional vector of probabilities for each class (output of our neural network) from the 10-dimensional actual label vector.

Normalization:

'popularity', 'acousticness', 'danceability', 'duration_ms', 'energy', 'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo', 'valence' are types of ratio data, where ratio data is data where "you can categorize, rank, and infer equal intervals between neighboring data points, and there is a true zero point. A true zero means there is an absence of the variable of interest. In ratio scales, zero does mean an absolute lack of the variable" (source) (for reference, loudness is measured in decibels [ratio with respect to a threshold]). It is advisable to standardize ratio data using z-score normalization, which is how we handle these features.

As the 'key' features are a type of ordinal data (and we have presented the reasoning for that in the section above), we refrain from standardizing the 'key' column in the same way. Likewise, as the 'mode' (binary variable) and 'music_genre' data are a type of nominal data, we also do not normalize these features using z-score.

Handling of Missing Data:

Through initially observing our data, we discover that there are various missing values across different rows (songs) and features that we must find a way to handle.

We first come across 5 rows (indexed using 10000, 10001, 10002, 10003, 10004) that have missing (nan) values across all features in the original dataset. This seems to be an artifact that came about in how the data was formatted or exported into a .csv file and so, accordingly, remove these 5 rows from our dataset (seeing as they are entirely undefined across all features and, therefore, do not contain any significant data we are losing by removing them).

We also came across 4980 songs that had a missing 'tempo' (the 'tempo' feature's values for these songs were '?', which is an invalid entry). Furthermore, we also came across 4939 songs that had a missing 'duration_ms' (the 'duration_ms' feature's values for these songs were -1; this is an invalid entry as duration_ms is the time of the song in milliseconds, which cannot be less than 0).

We present and rely on three different strategies for handling this set of missing values that we test and compare throughout this report. We present these techniques here:

1. Iterative Imputation

We look to established techniques for imputation that are offered by pre-existing packages; one of these techniques is iterative imputation. IterativeImputer is a function provided by sklearn that implements a "[m]ultivariate imputer that estimates each feature from all the others". The strategy it utilizes "for imputing missing values [is] by modeling each feature with missing values as a function of other features in a round-robin fashion" (source). The estimator that we use with IterativeImputer is BayesianRidge, which uses regularized linear regression to estimate the missing feature values using the other features in a round robin fashion. The benefits of using this technique is that it is simple and intuitive to implement (and is able to handle the issue of more than one feature being missing simultaneously). At the same time, the regularized linear regression estimator is, generally, fast, which aids in producing fast imputations. However, as we are using a linear model to estimate the missing features and there isn't a strong linear relationship between most features and the missing features 'tempo' and 'duration_ms', this method could disadvantageously introduce more noise into our dataset by synthetically 'asserting' a linear relationship between some of our variables and the missing 'tempo' and 'duration_ms' features.

2. Neural Network Imputation

The IterativeImputer scikit method does not provide the option of using a neural network as an estimator, so we designed our own implementation of an imputer that uses a neural network defined with PyTorch. This imputer works in a similar way as the IterativeImputer (in that it works in a round-robin fashion to estimate 'duration_ms' and then 'tempo' using all existing features); however, it uses a multi-layer perceptron (fully connected network). The network consists of 5 linear layers (7 neurons, 6 neurons, 5 neurons, 4 neurons, and 3 neurons), each followed by ReLU activation functions. This allows us to capture the nonlinear relationships present between the missing features and the remaining features. The network is trained using mini-batch gradient descent with batch_size = 128 so as to find a compromise between finding the optimal solution and doing so efficiently/without too many epochs. Furthermore, we rely on a learning rate scheduler that reduces our training rate dynamically as the validation loss begins to plateau (to allow for a faster convergence initially but a more slow descent as we approach the optimal solution). To avoid overfitting, we track our validation loss (subset of our training set, which will be discussed extensively below) to ensure it does not begin to rise as training loss plummets. Note: to ease the round-robin fashion of this implemented imputer, we simply remove songs with both 'tempo' and 'duration_ms' values missing (479 points in total). This makes up less than 1% of the data, so removing this data is rather trivial. Using such a neural network imputer provides us with another way to explore whether non-linearities between existing features can be leveraged and modelled to better predict missing feature values.

3. Removal

The third, but naïve, method to handle the missing data is simply by removing all missing values (9440 songs in total). The advantages of such a solution (avoid imputation entirely) is that we ensure we are not adding any noise or erroneously projecting a non-existing linear or non-linear relationship between some of our existing features and their missing counterparts: our imputation techniques could cast a relationship between some of our variables that isn't actually there and tamper with the data's validity. At the same time, these 9440 songs may lead to very marginal improvement in our classifiers, so we are better off working with ~40000 valid data points whose values were not synthetically predicted. On the contrary, a big disadvantage of this technique is

the amount of data we are losing. 9440 songs is approximately 20% of the data and, while 'tempo' and 'duration_ms' (or both) values are missing from these songs, the remaining features are perfectly valid and could be of great service to our classification models; therefore, by simply removing these values, we are barring ourself from valid and potentially utile data.

We explore all three of these data handling techniques and present three results for each model (for each missing data handling strategy).

Train-Test Split and Validation Set:

It is important to ensure there is absolutely no leakage between our train and test sets when we are setting up the various components of our classification pipeline. This means we need to maintain the same train and test splits in all our functions (from the ones used for imputing missing values, dimensionality reduction, clustering, setting hyperparameters, to the classification itself).

We consistently use the same random state (corresponding to N-number) in all our train-test splits. Furthermore, we keep our test set size to 10% of our original set and stratify our split using the genre column to preserve "the same proportions of examples in each class as observed in the original dataset" (source); this ensures that for each genre, 500 randomly picked songs from that genre are placed in the test set and the other 4500 songs from that genre for the training set. Maintaining this standard across all parts of our work is pertinent: even when imputing missing data, we cannot introduce points or data from our test set as that will skew the distribution of data being used to 'impute'/'generalize' what those missing values would be. For example, if building an imputer that can predict missing values using the other features of a datapoint, this imputer would have had to train on a set of data; this data ought not to be from the test set in any way so that we are not introducing examples from the test set into our imputer's learning process.

In addition, to set certain hyperparameters (such as when finding the regularization C value in linear SVMs or when setting early stopping criterion in multi-layer perceptron classifiers), we split our training set further into a smaller validation set. It is important to use the training set here to extract our validation set when setting hyperparameters such as to ensure no leakage whatsoever from our testing data.

Use of ROC instead of PR Curve:

This section deals with addressing which AUC was reported and highlighted in our analysis. It is important to distinguish the difference between the ROC and PR curves as their use when reporting data is highly dependent on the context of the data and the problem being addressed. We only address ROC curves in the report as it is more relevant to the context of the problem being developed. ROC "curves are appropriate when the observations are balanced between each class, whereas precision-recall curves are appropriate for imbalanced datasets" (source). The imbalance in our data set is quantified as: 50000 songs in total, 5000 songs for each class (music genre), so exactly 10% of our data points are for each genre. To determine what imbalance is enough to justify the use of PR curves over ROC curves, we refer to the paper, "Foundations of Imbalanced Learning" by Gary M. Weiss (source), in which he argues that there "is no

agreement, or standard, concerning the exact degree of class imbalance required for a data set to be considered truly 'imbalanced.' But most practitioners would certainly agree that a data set where the most common class is less than twice as common as the rarest class would only be marginally unbalanced, that data sets with the imbalance ratio about 10:1 would be modestly imbalanced, and data sets with imbalance ratios above 1000:1 would be extremely unbalanced". Our data set is certainly balanced; therefore, we restrict our analysis in this report to ROC curves.

At the same time, we present the ROC curve using macro-average OvR. OvR stands for One-vs-the-Rest (OvR), which is a "multiclass strategy, also known as one-vs-all, [that] consists in computing a ROC curve per each of the n_classes. In each step, a given class is regarded as the positive class and the remaining classes are regarded as the negative class as a bulk" (source). Obtaining the macro-average "requires computing the metric independently for each class and then taking the average over them, hence treating all classes equally a priori" (source). Given that the data is already balanced, we settle for a macroaverage, given that we weigh the correct classification of each music_genre equally importantly as an outcome of this model. Choosing between OvR instead of OvO (One-vs-One) becomes significant when there is significant class imbalance in the data, which is not the case for our application.

All Features:

We first build a series of classification models using all features that remain after pre-processing our data. This allows us to form a basis for how well a classifier can perform, prior to introducing an appropriate dimensionality reduction step in the next section of this report. We report the Maco-average OvR AUC of the ROC curve (as discussed in extensively in the previous section) and Accuracy for each of these models. Note that we report three values for each classification model (depending on the technique utilized to handle missing data). We describe the various classification models utilized here:

Support Vector Machine:

We begin with a linear support vector machine. We find the optimal C (regularization hyperparameter for SVM) before we produce our models. We produce a list C with values ranging from 1e-6 to 2. We iterate across the C list, producing an SVM model with each value of regularization, testing the model with 5-fold cross validation, and storing the results (note that train-test split prior to cross validation is based on the same principles outlined in the data pre-processing section of this report). We then plot the CV scores against the C value used for regularization in the SVM model and select the C value that corresponds to the highest score. We then build and test a linear SVM classifier using the data and this value of C.

Decision Tree:

We use a single decision tree using the 'gini' criterion as the function to measure the quality of a split. We keep ccp_alpha=0.0 but enforce a max_depth of 4 on the tree to ensure it does not overfit to the training data and can generalize to the train set.

Random Forest:

We use a random forest to try a bagging-based ensemble method. We set a large number of estimators to improve the performance of our random forest classifier (though that comes with an increased computational cost). To best leverage this increased number of estimators, we set the max_features as 0.5 so that half of the features are considered at each split. We set max_samples as 0.3 so that 30% of the data is sampled to train each base estimator (and ensure that bootstrap=True so that bootstramp samples are used when building trees—sampling with replacement some portion from data instead of using the entire dataset for each estimator). In addition, we set certain parameters to regularize our trees and ensure they do not overfit to the training data (by setting the minimum number of samples required to be at a leaf node and the maximum depth of the tree).

AdaBoost:

We use AdaBoost to try a boosting-based ensemble method. The estimator we use to build our boosting model on is a decision tree. We use 400 estimators to improve the performance of our boosting model (though that comes with an increased computational cost) and we use the SAMME discrete boosting algorithm.

Neural Network:

Finally, we build a multi-layer perceptron network as a classifier. The architecture of the network is 2 hidden layers, each with 12 neurons, followed by the ReLU activation function (to avoid any potential vanishing gradient issues—though the network is likely not deep enough to cause that—and because of the promising empirical results produced when using ReLU). We then use a fully connected layer to map the activations of the final hidden layer to 10 neurons (as we have 10 classes) and use softmax to assign "decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would" (source). We use the CrossEntropyLoss for this classification task as cross-entropy is a commonly used loss function for classification tasks, it is usually robust and is compatible with our gradient descent optimization algorithm. We use the SGD module as our optimizer as it works well with our defined loss function. We begin with a learning rate of 5e-2 as we will be training our model using mini-batch gradient descent and we found this learning rate worked best to help our model begin to converge to an optimal solution (without taking too much time) (this is as we use a scheduler to dynamically update this learning rate). We choose not to add a weight_decay to regularize our models during training as we have included early stopping as one criterion to prevent overfitting; we noted during testing that adding weight_decay as a second criterion for regularization lead to highly underfitting results. Furthermore, the addition of momentum = 0.5 is an important design decision that helps our gradient descent converge faster to a more optimal solution. The momentum term is computed as a moving average of the past gradients; it helps ensure that the optimizer does not get stuck in a local minimum (based on the accumulated gradients). Additionally, the use of a scheduler allows us to begin with a higher learning rate (5e-2) that helps us converge more quickly towards our solution initially but, as we begin to plateau, the learning rate is dropped by some rate so that we can more slowly and (in a controller manner) converge to a solution. This allows to us to get the best of both a high and low learning rate (depending on which sage of

training our model is in [initial vs near-convergence]). We use mini-batch gradient descent with batch size of 256 as we wish to handle the trade off between model training performance and time to converge. If we use stochastic gradient descent (practically mini-batch of 1), we will converge quickly to a solution but it is likely not to be the optimal model. If we use batch gradient descent, we will converge to an optimal solution but this will take a great deal of time to train and may not be the most time-efficient approach. We, thus, settle for mini-batch gradient descent to optimize our training time and model performance.

Classifier	**Iterative Imputer**		**Neural Network Imputer**		**No Imputation (Remove Missing Data)**	
	AUC	**Accuracy**	**AUC**	**Accuracy**	**AUC**	**Accuracy**
Support Vector Machine	0.89	0.496	0.89	0.506	0.89	0.509
Decision Tree	0.85	0.429	0.84	0.423	0.85	0.425
Random Forest	0.93	0.567	0.93	0.579	**0.93**	**0.583**
AdaBoost	0.91	0.530	0.91	0.550	0.92	0.567
Neural Network	0.84	0.506	0.90	0.549	0.89	0.526

Given these results, we see that the mdodel that maximizes macro-average OvR AUC of ROC curve and accuracy (on test data) is a Random Forest with the imputation technique being used to remove missing data, with a produced AUC and accuracy of 0.93 and 0.583, respectively. We produce the ROC curve below:

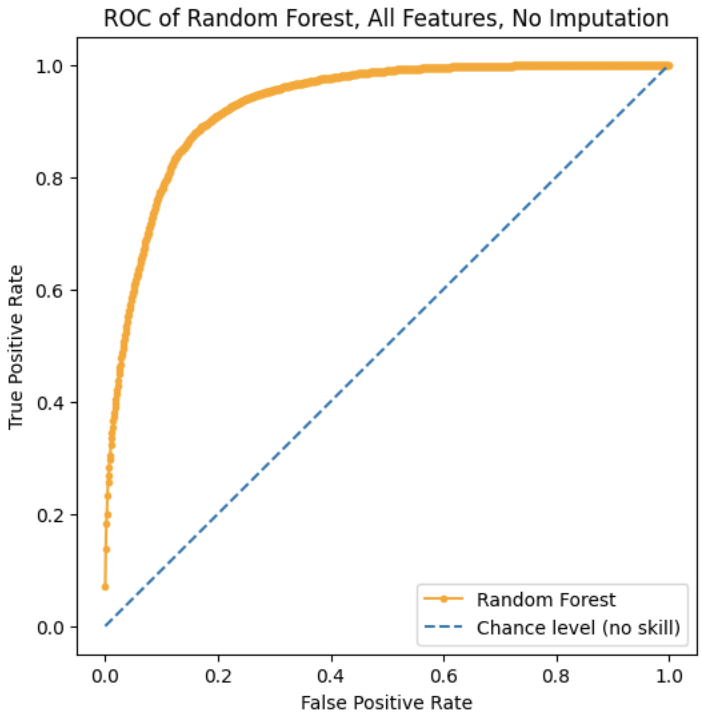


Figure 2: ROC of Random Forest, All Features, No Imputation (Remove Missing Data)

Dimensionality Reduction and Clustering:

We begin with testing out various dimensionality reduction techniques to evaluate what visualizations are produced by each and which should be used for clustering and to then proceed with attempting to build a classifier model using the lower embedded space produced by the dimensionality reduction techniques.

As presented prior, we ensure that, when fitting any of our dimensionality reduction techniques, we are doing so using the training set that we defined earlier in our pre-processing section to ensure no leakage between our training and test sets.

Principal Component Analysis (PCA):

We begin testing with PCA as a dimensionality reduction method as it is fast (closed-form solution) and will allow us to observe how the principal components along which projections of our data have the largest variance (eigenvectors of covariance matrix of our data) can visualize our data (and how the groupings formed in this lower-dimensional space compare to our original genres, which we continue to use to classify).

We continue to use our pre-processing steps of using z-score standardization as 1) PCA requires that each feature have a mean of 0 as PCA is a rotation operation and that rotation needs to be about the origin of whatever m-dimensional space the data exists in (where m is the number of features describing the data); another way to conceptualize this is thinking of how the covariance matrix of the data can be represented as $X^T X$ under the assumption that the mean of each feature is 0. 2) PCA results will be skewed if distances are not normalized as PCA is solved via the Singular Value Decomposition which approximates in the sum of squares sense; therefore, scaling the features ensures that no feature dominates the outcome. Furthermore, we drop ordinally encoded variables (like 'key' and 'mode') as they are incongruent with PCA's expectation of continuous variables. In addition, we use whiten=True as a parameter in PCA() so that the principal components "vectors are multiplied by the square root of n_samples and then divided by the singular values to ensure uncorrelated outputs with unit component-wise variances". Most pertinently, we set the number of components as 3 per the kaiser criterion as the first 3 principal components have eigenvalues above 1.0 (this is as the Kaiser rule "is to drop all components with eigenvalues under 1.0 – this being the eigenvalue equal to the information accounted for by an average single item") (source).We project the variance on the first two principal components to be able to visualize the 2D solution. Seeing as it performed best in our classification with all features, we use the no imputation (remove missing data) missing data handling technique in all our dimensionality reduction work. We show the visualization of the testing data projected on our first two principal components (based on the PCA object fitted using solely the training data), with the points labelled per the original genres they belong to:

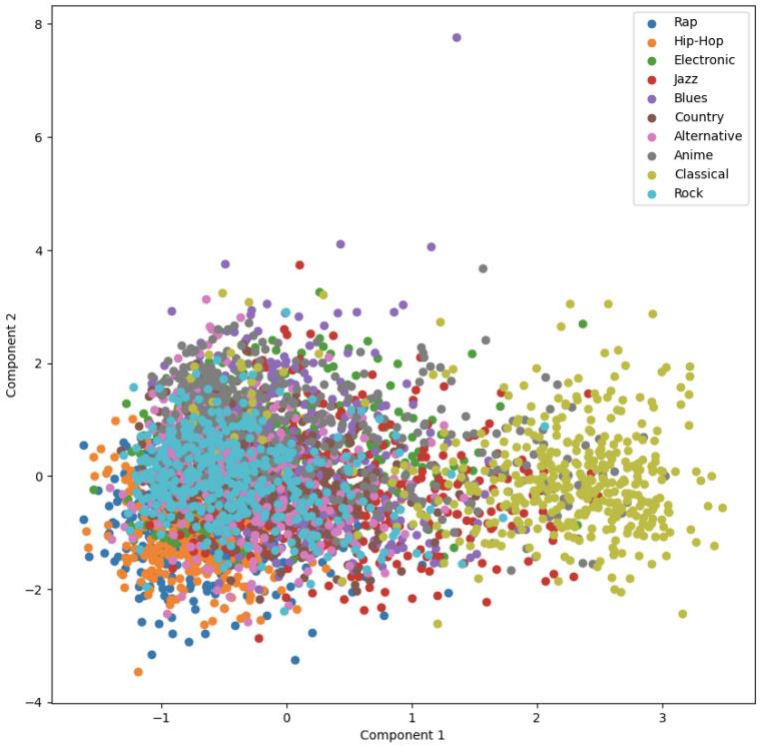


Figure 3: Testing Data Projected on First Two Principal Components

There aren't any natural groupings that emerge or surface when visually observing these two components. Even visually, it is not a trivial task to cluster the data when projected in this 2D space. It is clear there is a blob-like grouping of classical music to the right of the figure, with the remaining genres clustered in a circular-like blob to the left of the diagram. If anything, this visualization points to classical music's distinction or separation from other music genres, which can appear meshed/sound similar.

It is clear that simply finding the components along which the projection of the data is maximally varied is not sufficient to allow for insightful observation in this reduced-dimensional space. This motivates us to look towards other dimensionality reduction methods that we can use and depend on that aim to preserve the original separation provided by the music genre labels. We will approach this with the use of Linear Discriminant Analysis.

Linear Discriminant Analysis (LDA):

Linear Discriminant Analysis is a dimensionality reduction technique that is preferable to applications with labelled data. The LDA algorithm transforms the input data such that the variance of class means is maximized and the variance within classes is minimized (in other words, "maximize the distance between the means of the two classes and minimize the variation within each class" (source)). One assumption that LDA makes that affects how we organize our classes is that—in addition to requiring class labels—it assumes gaussian distribution of data within classes. To guarantee this, we iterate through the classes as well as through the various continuous features (as LDA assumes features are continuous), and produce the mean, median, mode, kurtosis, and skewness. To check whether a distribution can be classified as gaussian, we

ensure that mean = median = mode (within moderation) and that skewness is 0 and kurtosis is near 3. This is because gaussian distributions have the property that mean = median = mode, the "skewness for a normal distribution is zero, and any symmetric data should have a skewness near zero" (source), and a "standard normal distribution has kurtosis of 3 (0 with fisher definition) and is recognized as mesokurtic" (source). Through this, we discover that 'liveness', 'acousticness', 'duration_ms', and 'instrumentalness' are not normally distributed and are removed as features prior to using LDA.

We select the number of LDA components that explains 95% of the variance of the data; therefore, we select 3 LDA components as they explain ~95.0% of the data's variance. The test data projected on the first two LDA components is visualized below:

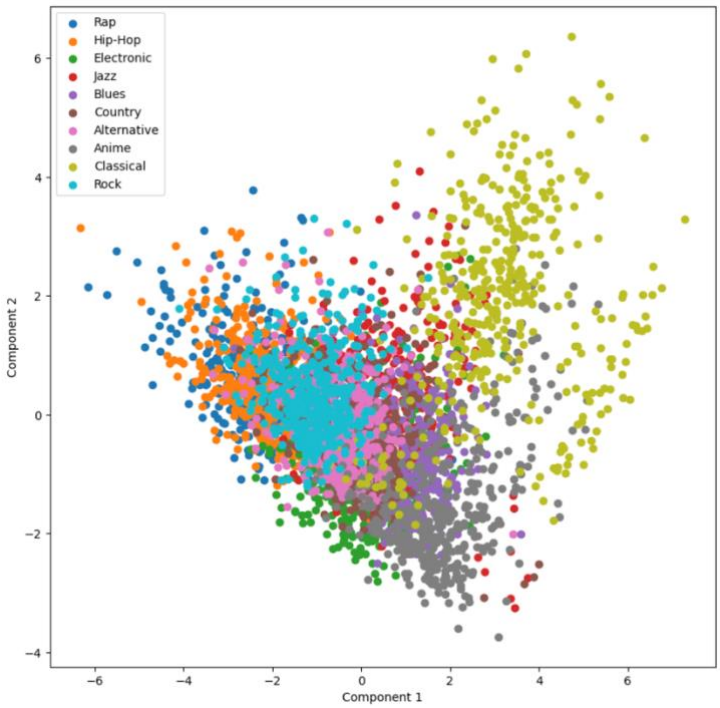


Figure 4: Testing Data Projected on First Two Components Produced by LDA

We can see that there are improved groupings (relative to the PCA solution) that seem to form out of this, whereby classical music and anime music seem to form their own distinct groupings. Most other genres seem to form into a single group, but we can still perceive that jazz music often finds itself between classical music and the larger grouping of several music genres (which is an expected outcome, given that classical music and jazz music often involve musical instruments beyond vocal invocations—though jazz music is much more likely to offer the latter (bringing it closer to music genres like rap and hip-hop, which depend greatly on the voice of an artist).

Given the computational advantage of these linear dimensionality reduction technique and LDA's greater focus on preserving class separability—which aligns more closely with our classification problem, we will continue our exploration using LDA.

From Figure 4, we see that the groupings provided by Figure 4 are generally non-blob like and have varying densities. To be able to effectively cluster the data when it's projected in this lower dimensional space, we use DBscan as a clustering algorithm.

We use the projection of the data in the lower-dimensional embedding space offered by 3 components with t-SNE, though we plot the clusters produced by DBscan in 2D which offers an incomplete but helpful visualization of what groupings are naturally forming from the data. Based on the range of values (after standardization), we set epsilon to be 0.40 and we set minimum number of values (for a cluster to be counted) as 9 to ensure we are dropping outliers and considering only sizeable groupings.

The result of this clustering is provided below:

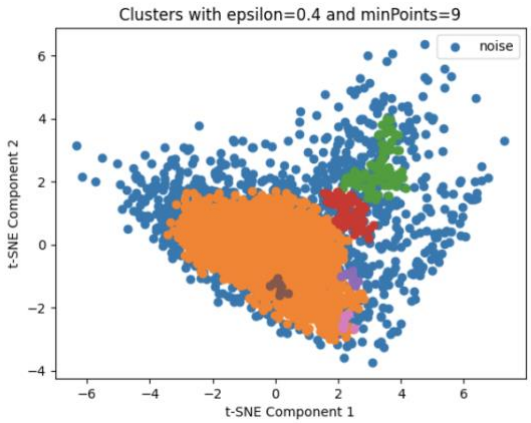
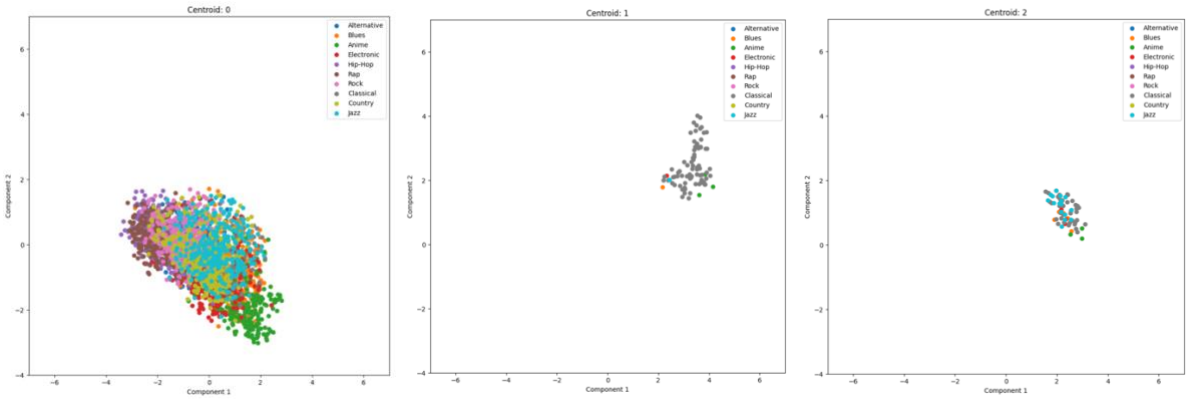


Figure 5: Clusters Provided by DBscan (color indicates cluster, not music genre)

Based on Figure 6, we can see that 6 clusters have formed while a sizeable part of the data was classified as noise. This is a consequence of our using of 3 t-SNE components to cluster (meaning there is an added dimension not visualized here that may be contributing to the placement of these points as noise.

To get a clearer image of what is being produced here, we observe each cluster individually and examine the distribution of music genres in each cluster, which is offered visually below:



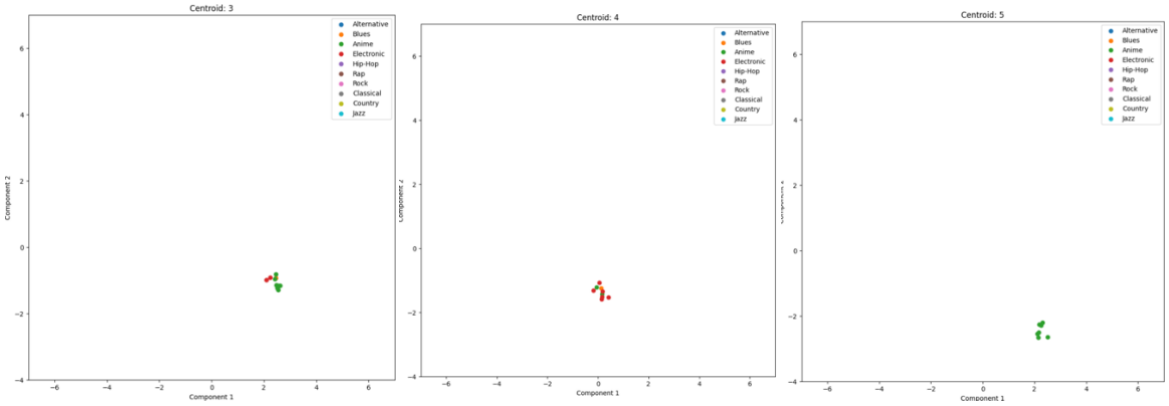


Figure 6: Distribution of Music Genres Across Clusters

We can observe that the classical music grouping we noted earlier is present in centroid 1. Centroid 0 represents a large grouping of several music genres that may be a consequence of LDA being unable to effectively separate the classes using just 3 components (95% of the variance of the data), which calls for the need for additional classification methods on top of this dimensionality reduction step (with LDA).

Classification with Reduced Dimensionality:

As a final step, we present a classification method using simply the 3 components produced with LDA (data in lower dimensional embedding space). Based on the favorable classification performance provided by Random Forests, we use the same model here (adjusting the maximum number of leaves to correspond to the reduced number of 'features' (LDA components) we are using. Similarly, we use no imputation as a technique to handling missing data as it produced the most favorable results in part 2 of this report (with all features).

Classifier	**PCA (2 Components)**		**PCA (3 Components)**		**LDA (2 Components)**		**LDA (3 Components)**	
	AUC	**Accuracy**	**AUC**	**Accuracy**	**AUC**	**Accuracy**	**AUC**	**Accuracy**
Random Forest	0.72	0.264	0.79	0.327	0.87	0.427	**0.90**	**0.508**

We can see that our results agree with the observations we drew from our earlier visualizations whereby we noted that LDA produced more natural groupings with 2-3 components and that PCA (because it was blind to the original music genre class labels) just looked to maximize variance along the principal components.

The ROC curve produced by our best performing model (when reducing dimensions with LDA on the test data and using 3 components that capture 95% of the test data's variance) is:

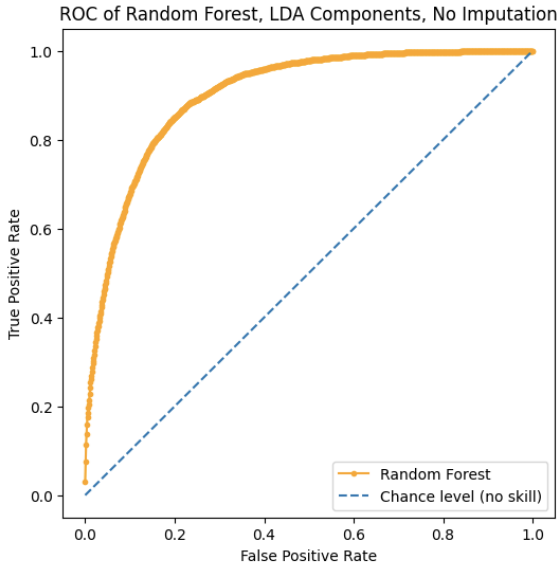


Figure 7: ROC cuve of random forest, no imputer, with test data projected on 3 LDA components

Factors Behind Classification Success:

I believe the core factors that underly the classification models we have used here and their performance is the nature in which we developed various tools to use prior to our classification. We tried out different initial imputation techniques (from using sklearn's iterative imputer class, to simply dropping the data, and all the way to developing our own round-robin, neural-network based imputer that performed better than our use of sklearn's iterative imputer class).

Furthermore, trying out different classification tools and spending time toying around with the hyperparameters/using cross validation to decide on regularization parameters played a critical role in guiding us towards the better models that we can then build off (as reflected by our use of random forest after experimenting with 5 different classification models earlier with all the features).

Non-Trivial Observations:

We also explore the use of non-linear dimensionality reduction techniques to examine what results they provide us (when casting the data in the embedding space they provide).

In addition to being computationally expensive, the non-linear dimensionality reduction techniques do not look to preserve the class separation provided by music genres and would not take that information into account the 2-dimensional embedding space provided by t-SNE produces the following, whereby it is clear that using dimensionality reduction processes that do not involve or take into account class (music genre) labels is not favorable.

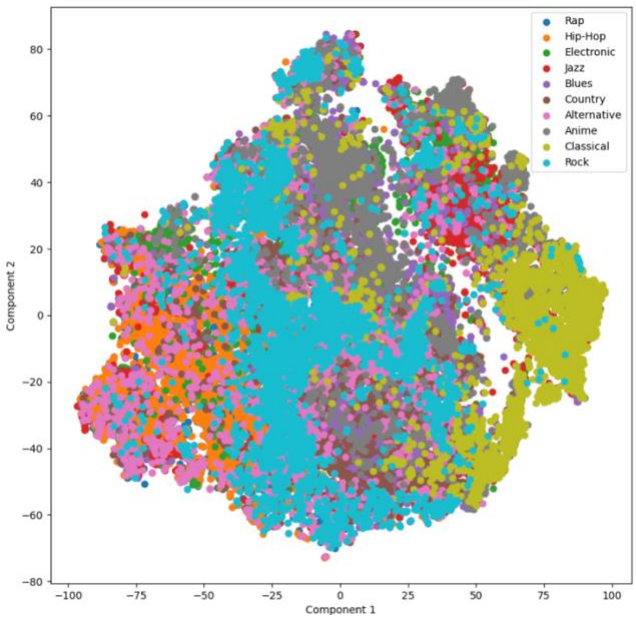


Figure 8: Data in t-SNE Lower-Dimensional Embedding Space

This motivates us to look beyond the linear and nonlinear dimensionality reduction techniques that we covered in class and potentially search for non-linear dimensionality techniques that preserve the inherent class separation provided by the data's labels (music genres).