Performance Comparison of Traditional Machine Learning Algorithm and Neural Network on Music Genre Dataset

Kevin Fernandez*, Xirong Cao*, Sean Patten*

Computer and Information Science Department

Fordham University, New York, NY

{jfernandez108, xcao50, spatten2}@fordham.edu

ABSTRACT

The goal of this project is to apply and compare three popular classifiers for music genre classification using the GTZAN Dataset **[1]**. The classifiers are Bayes Classifier, Support Vector Machine (SVM), and Neural Network (NN). The project investigates how traditional machine learning models fare against NNs, which are widely used in machine learning and deep learning. The project also implements enhancements for each classifier: feature selection for Bayes Classifier, kernelization for SVM, and randomization for NN. The project helps to reinforce the understanding of machine learning algorithms and explore possible ways to improve current machine learning methods. The steps of the project include preprocessing the dataset, dividing it into training and testing sets, building the classifiers, measuring their performance, and comparing them using true accuracy as the metric. The results show that NN is the best classifier. However, it also takes longer to run than SVM, which is the second-best classifier, especially for larger datasets. Bayes Classifier, although less accurate, is useful for fast feature selection checks. Future work could involve using Bayes Nets, experimenting with different training/testing splits, and comparing with other datasets.

INTRODUCTION

We have undertaken a comparative study and implementation of three methods of classification applied to a musical genre recognition problem using the GTZAN Dataset. The methods are Bayes Classifier, Support Vector Machine, and Neural Network. The latter has become ubiquitous in modern machine learning and deep learning applications, but its merits and limitations can only be assessed by contrasting it with the more established techniques. We have not only implemented the basic versions of these methods, but also introduced some refinements to enhance their performance. For the Bayes Classifier, we have improved the feature selection process; for the Support Vector Machine, we have used a more suitable kernel function; and for the Neural Network, we have utilized randomization in the back propagation process. Our project serves both an educational and a scientific purpose: it reinforces our knowledge of the fundamental principles of machine learning, and it contributes to the ongoing research on the best methods for machine learning by exploring our techniques in the field of musical genre recognition. Neural Networks are a relatively recent innovation in machine learning, and their superiority over other methods is not yet fully established. Our project aims to shed some light on this question by comparing them with their predecessors.

METHODOLOGY

Before starting the implementation process, the irrelevant "filename" feature is removed, and the dataset is divided into two sets: one with 58 features and the other with ten classes representing music genres. These sets are further split into 40% training and 60% testing data, using a randomness state of 42 to ensure consistent analysis of our implementation and improvement models. The three main classifiers we will construct are the Bayes Classifier, Support Vector Machine, and Neural Network. Each classifier has its own definition: Bayes Classifier is a probabilistic model that assigns the most likely class label based on conditional probabilities of features given class labels; Support Vector Machine is a powerful algorithm that separates and classifies data by finding an optimal hyperplane with the maximum margin between classes; Neural Network is a computational model consisting of interconnected artificial neurons that can learn and predict complex data patterns. Following the implementation of the main classifiers,

each classifier will undergo specific improvements. The performance of all models will be compared using true accuracy as the evaluation metric, and the runtimes will be executed in Google Colaboratory. Ultimately, the best model for each classifier will be compared for final selection which will represent the overall best model for our dataset. Throughout the paper, we will provide detailed explanations for our implementation decisions and improvements, while thoroughly evaluating the performance of each model.

BAYES CLASSIFIER

Introduction

The Bayesian classifier, commonly known as the Bayes classifier, is a probabilistic classification technique that employs Bayes' theorem. It is used in machine learning for pattern recognition and classification tasks. The Bayes classifier evaluates the likelihood of each class given the input data and chooses the class with the highest probability. During training, the classifier learns the class and conditional probabilities of each feature with respect to each class from the training data. Once the Bayes classifier model is trained, it can classify unclassified data into the most probable category. Some examples of applications of the Bayes classifier include atmospheric state classification, spam filtering, and mood classification based on actions. We will now test the effectiveness of the Bayes classifier using our music genre classification dataset.

Implementation

We will create three Bayes classifiers using the Gaussian, Uniform, and Rayleigh distributions. We will structure the Bayes classifiers code into a class format to enable comprehensive documentation and testing. The first Bayes classifier focuses on data with a Gaussian distribution, consisting of three parts as functions. The "init" function defines the class or prior probabilities, feature probabilities, and classes. The "fit" function takes in X and Y-training data and calculates the priors for each class by dividing occurrences per class by the overall occurrences. This function also separates the data by class to calculate and store the mean and variance for each feature that shares the same class. The "predict" function takes in the X-testing data and classifies it. Before classifying, it multiplies the trained priors from the "fit" function with the respective Gaussian likelihood function value of the data in question from the X-testing data. We can mathematically represent the Gaussian likelihood function **[2]** as "(1 / np.sqrt(2 * np.pi * variance)) * np.exp(-(X_test - mean)**2 / (2 * variance))" in Python and $(1/\sqrt{2\pi\sigma^2}) * exp(-(x-\mu))^2/(2\sigma^2))$ as seen in Appendix A, where π is the mean, σ² is the variance from the training data, and X is the data in question. The function chooses the maximum posterior out of all the posteriors between classes and classifies the data accordingly. After executing the Gaussian Bayes classifier, we developed an accuracy function called "accuracy_model" to evaluate the model's performance by mapping our predicted classes from our testing data to their actual classes by calculating the ratio of correct predictions to total occurrences. Our accuracy function showed a 51.76% accuracy rate.

Our second Bayes classifier will assume the dataset has a Uniform distribution. This classifier will recycle some parts of the Gaussian Bayes classifier code like the "int" function, but the "fit" and "predict" functions will be modified. The "fit" portion of the code will now include getting the maximum and minimum value for each feature per class from the training data that will be requested as input into an array form. In addition, the Uniform likelihood function **[3]** will be based on the formula seen in Appendix A or written in code as "np.where((min <= X_test) & (X_test <= max), 1 / (max - min), 0)" in the code submission. The

idea behind the function is if the value of a feature from the testing data falls within the range of its maximum and minimum values based on our training, then the likelihood is one divided by that range. If it does not, then the likelihood is zero. The mathematics assumes that the data points are equally likely to fall anywhere in each range. Then, the likelihood computation gets multiplied by its respective prior and classified based on the maximum posterior. After implementing the modifications and reusing the accuracy function, we got an accuracy score of 29.67%.

The third Bayes classifier was developed using the Rayleigh distribution. For this classifier, we reused the "init" and "fit" functions from the Bayes classifier Gaussian distribution model. However, we modified the "predict" function to incorporate the Rayleigh likelihood formula. The formula **[4]**, which is found in Appendix A or $(x/\sigma^2) * exp(- (x^2)/(2 * \sigma^2)$ where sigma or σ is also calculated as "sigma = np.sqrt(np.mean(A_class**2, axis = 0) / 2)" in Python. The sigma focuses on each class from the training data separately and then gets calculated as previously stated to represent the variability of the distribution. The likelihood will take the testing data and our calculated sigma and then be multiplied by the prior to be stored as a posterior. The function ends by choosing the maximum posterior and classifies it accordingly. The Rayleigh distribution resulted in an accuracy score of 23.00%.

Out of the three distribution models we tested, the Gaussian Bayes classifier performed the best with an accuracy score of 51.67%, while the Rayleigh model scored the lowest at 23.00%. These scores align with our expectations since the Rayleigh distribution is best suited for continuous non-negative valued random variables, while the audio waveform dataset we used contains multiple peaks and features with negative values. Therefore, the Gaussian assumption was a better choice since audio waveforms are often symmetric. We also compared our model against a pre-made Gaussian Bayes classifier from the "sklearn" package, which achieved an accuracy score of 39.33%, 12.34% less than our constructed Gaussian Bayes classifier. Despite the relatively better performance of our Gaussian model than the others, none of the models performed exceptionally well. But based on our results, we will use the Gaussian Bayes classifier for further analysis.

Traditional Greedy Feature Selection

We will now focus on developing a feature selection technique known as the traditional Greedy approach **[5]**. Its purpose is to identify and utilize the most relevant features for a given model where the features are selected sequentially based on their performance. This technique has the potential to be very complex in terms of criteria, which will be seen throughout this section.

Our goal with traditional Greedy feature selection is to sequentially add the best-performing features to an initially empty set until there is no further improvement in the model's performance. The "init" function initializes the model and four splits of training and testing data and includes lists to store the selected and remaining features for analysis. The selected features are the ones that are the best while the remaining features will start as all the features that are waiting for consideration. The "accuracy_fs" function calculates the accuracy between predicted and true classification. It is used in the "fit" function to calculate the accuracy of the model using specific sets of features we input. In the first iteration of "fit", we examine one feature at a time from the remaining feature list and reconstruct our training and testing data only containing that feature. It will calculate the accuracy of the model after the new training and testing data goes through our Bayes classifier using the "accuracy_fs" function. The current accuracy and feature will be compared to a previously stored best accuracy with its

corresponding feature. If the current feature has higher accuracy than the previously stored one, the current one and its accuracy replace the stored ones to be compared to the other remaining features. If a feature does not have a better accuracy or gets replaced, then it will return as a remaining feature to be considered again after the best-selected feature has been selected from that iteration. After considering all features, the feature with the best accuracy will get stored as one of the best-selected features and be used along with another remaining feature for the next round. Once a feature is selected as best, it will be permanently removed from the remaining features for consideration. The next iterations will reconstruct our training and testing data using all the best-selected features stored with one new remaining feature. However, we need our code to stop when no remaining features improve the model. So, we implemented a flag called "improvement" to ensure not all the features will be selected. The flag works by setting the "improvement" as "False" if no remaining features improve the accuracy, which keeps track of the accuracy between iterations and stops the feature selection process.

Our objective was to evaluate the performance of our Bayes classifier with Gaussian distribution through feature selection. The traditional Greedy feature selection method was used to select the best features, resulting in an overall accuracy of 55.33% with ten ordered features. This accuracy was better than using all features as shown in Appendix B. We also implemented a modified version of the Greedy feature selection method with the Akaike Information Criterion (AIC) **[6]** to see if we received better results. The AIC takes the log-likelihood to adjust the number of parameters used in the model but with a penalty term that discourages having additional features. Those values get subtracted to get the AIC score. Unlike the previous Greedy feature selection function, we had to address a bug by vectorizing and mapping our Y-testing classifications to be used as an array in "aic" method which is under the basic "NumPy" package. The formula to get the AIC score is $2k - 2ln(L)$, where k is the penalty term and the number of estimated parameters while L is the maximum likelihood function value of the model. The incorporation of residuals and the sum of squared errors are needed in AIC to consider the goodness of fit of the model and the amount of information lost due to the number of features used. The "aic" function was used to find the lowest AIC score instead of the "accuracy_fs" function. We kept the "accuracy_fs" to calculate the accuracy of this model at the end. The method will stop when the AIC score is infinite or very large meaning either the sum of squared errors is very small, or k is too large. This indicates there is a risk of overfitting the data, so the feature will not be added. The AIC method returned only three features with an accuracy score of 35.17% as shown in Appendix B. This is because AIC is a criterion that balances model complexity and goodness of fit, which may not necessarily lead to the highest accuracy. However, due to its computational cost and lower accuracy, we will focus on improving the Greedy feature selection method without AIC in the next section. Normalizing the data could have changed our performance which should be implemented in future work. We can see that the Greedy feature selection is needed to select the best features and remove irrelevant or redundant ones. This reduces the number of dimensions which can lead to a reduction in overfitting and an increase in the generalization ability and accuracy of the model if correctly implemented.

Improvements & Hyperparameters

Our first improvement involves making our traditional Greedy feature selection incorporate a randomness aspect with a hyperparameter to prevent the model from overfitting. We want this model to keep selecting the best feature in terms of accuracy one at a time, just like the traditional one, but after "K" steps/iterations with a default of 3, we will randomly remove a previously selected feature and add it back to the remaining features for consideration only using

the "random" package. We will also randomly add a previously removed feature to the remaining features for consideration to get picked or not again. The “init” function will start with the same variables as the traditional one but will include the "K" step hyperparameter and a list to keep track of removed, remaining, and selected features, as well as an improvement flag to make the function stop if the remaining features do not add any improvements. Our “fit” function will do the same methods and calculations as the traditional for the model's accuracy between training and testing data of our selection of features, but we will now track the number of steps. Once we reach the "K” step or a multiple of it, the function triggers a new set of operations. If the removed features list is not empty, we select a randomly removed feature from previous iterations and remove it from the selected features list if it is in there too. This step is required because if the previously removed feature is again the best feature to select at a certain iteration the feature would need to be moved back from the removed features list to the selected features list so that it can be considered for future iterations on the remaining features list. If the removed features list is empty, we will move on to randomly remove a selected feature, which will be added to the removed and remaining features list. Another crucial component of this code is our three stopping conditions. The first condition is when the number of selected features is equal to the number of remaining features to ensure that all available features have been considered and evaluated for performance (written as “if len(self.selected_features) == len(self.remaining_features): break” in the code). The second stopping condition is when there are no more features to evaluate in the remaining feature list (written as "if i >= len(self.remaining_features): break" to prevent an infinite loop). The third condition is when none of the remaining features improve our performance using the “improvement” flag, as we did with the traditional Greedy feature selection. So, the “improvement” flag stops the feature selection when the remaining features, or during the "K” step, do not increase performance.

We have tested five different values of "K," and Appendix C shows their corresponding accuracies, number of features, ordered features, and run-time results. Please note that the results are subject to change due to the randomness component. Based on our experiments, we can conclude that this model outperformed the traditional feature selection method, and increasing the value of "K" resulted in the selection of more features. The randomness of this method allowed us to explore a wider range of feature subsets, which can adapt to different data sets and hyperparameters that the traditional does not. This can lead to improved accuracy and generalization performance compared to traditional methods.

The second improvement on our traditional Greedy feature selection will select the best feature pair or triple at a time with a hyperparameter called “num_initial” that will influence the initial number of random features we will look at from the training dataset. The hyperparameter is important because we will be looking at all possible ways to form features in pairs and triples, so the runtime can be extremely long, especially for our dataset with 58 initial features. So, we will want to take a percentage of the initial features, or the first iteration will have 25,611 combinations to model and measure its accuracy. We will continue with the “init” function asking for the model, splits of training and testing data, and creating a list for selected and remaining features like the traditional Greedy feature selection. Now, there is a hyperparameter called “num_initial” with a default of .4 (40%) that wants a value between zero and one representing the percentage of randomly selected features we will look at from the initial training dataset, which will be all the initial features placed in the remaining list for consideration. However, to make that possible we will need to define a new function called “initials_functions” that does with the help of the “random” package. Before we analyze the “fit” function, we must

create another function called “all_possibilities” that acts like a generator and returns all possible combinations of n elements from an array, which gives us the flexibility to make any number of sets like two or three at once. The “accuracy_fs” function will continue to calculate the accuracy of the model with given inputs. Now, the “init” function will start by tracking a new list of feature sets that have been selected so far. We want the function to stop if there are fewer than two remaining features since the one remaining feature can’t be paired up with anything. Our flag “improvement” is on standby again to stop the function when there is no further improvement like in the other constructed feature selections. We can now create all the possible feature sets of two using our “all_possibilities” function. However, we must make sure to not add repeated feature sets by skipping them if they already appear in the selected feature sets. Then, it will select the corresponding features from the training and testing data to make new sets that will be used as inputs to fit the model. The “accuracy_fs” function will calculate the accuracy of the model on the previous new testing data. If the accuracy is better than the previous stored best accuracy, the best feature set, best accuracy, and “improvement” will be updated like what the previous feature selection models have done. If no more pairs of features can improve the model by using a second “improvement” flag, then it will start creating triple combinations of features until it exhausts all the possibilities and calculates the accuracies as it did for the pairs. This loop will stop when the third “improvement” flag goes up when the accuracies did not get better. It will add the best pair or triple feature set from the iteration to our main selected features list for output and to a selected feature set that will be used for the next iteration with new sets of features for consideration. Also, the best feature set will be individually removed from the remaining feature list for consideration.

Similarly, to the random Greedy feature selection, we also tested the impact of the "num_initial" hyperparameter on the pairs/triples feature selection. Appendix D presents the results obtained with different values for this hyperparameter, including their corresponding accuracies, number of features, ordered features, and runtimes. The results showed that the value of "num_initial" significantly affects the runtime of the model due to a large number of possibilities to calculate. Additionally, there was a direct relationship between the number of features returned and the value of "num_initial." However, the accuracy of this model was lower than our traditional and random Greedy feature selection, despite longer runtimes, especially with a larger “num_initial.” One potential issue might be that the smaller features selected from the training set in pairs/triples could lead to overfitting, and randomly selecting the initial features could weaken our model even further. As a result, we do not recommend using this type of Greedy feature selection.

Conclusion

We refrained from discussing the specific features chosen for each model to assess their overall similarities. Despite variations in the selected features and their respective importance rankings, the results displayed striking similarities. Nearly all of the feature selection models, except for the Greedy feature selection with AIC, produced a consistent set of features, which were chroma_stft_mean, chroma_stft_var, rms_mean, rms_var, spectral_centroid_mean, spectral_centroid_var, spectral_bandwidth_mean, spectral_bandwidth_var, mfcc2_mean, and mfcc2_var in some sort of variation or missing a few features at times. This outcome serves to confirm two key points: firstly, the feature selection and classifiers are functioning consistently and accurately, as they consistently yield similar results even when introducing randomization or combinations; and secondly, the shared features that were selected are crucial for achieving optimal performance. In conclusion, the Greedy feature with random improvement benefited our

Gaussian Bayes classifier model the most by increasing the overall accuracy without any huge computation costs. The revelation is our model needs to evaluate more possible scenarios with new features being tested one at a time for better performance. This model, chosen from the Bayes Classification realm, will be considered the leading contender for evaluation alongside the top models from other classifiers. While our implementation and improvements on the Bayes Classifiers have shown less than promising results, some limitations could be addressed in future research. Firstly, we could explore the use of different probability distributions beyond the three we used in our study. For instance, the log-normal and gamma distributions might be suitable for our audio waveform dataset. Additionally, all our Greedy feature selection models focused on forward selection, so there is a potential benefit of trying backward elimination, where we remove features one by one, to gain further insights or improvements into our dataset. Another area of exploration that we could have pursued is selecting a representative data point from each class and running it under three different probability distributions instead of making the whole dataset go through the same distribution calculations. These changes could potentially lead to more robust and accurate models for audio classification.

SUPPORT VECTOR MACHINE

Introduction

Support Vector Machine(SVM) is a powerful supervised machine learning algorithm used for classification and regression tasks, which aims to find the optimal separating hyperplane between classes while maximizing the margin between the nearest data points. Mathematically, it solves a constrained optimization problem to find the optimal hyperplane parameters. SVM is commonly employed in binary classification problems but can also be extended to handle multi-class classification tasks using techniques like one-vs-one or one-vs-all. It is effective at classifying linearly separable data. For non-linearly separable data, it leverages the kernel trick to project the data into a higher-dimensional space where the classes become linearly separable. In our dataset GTZAN Dataset - Music Genre Classification[1] classification task, we are going to first test the classification performance SVM algorithm on this dataset, and later test different improvements compared to the baseline.

Implementation & Improvement

The GTZAN dataset is a famous benchmark for music genre classification tasks, containing 1,000 audio samples evenly distributed across 10 music genres. The first task in the SVM section is to perform Principal Component Analysis(PCA) to reduce feature dimension because intuitively that could reduce the computation burden. PCA is performed using a prebuilt Python package called scikit-learn. By only keeping 95% of the original variance, we use this as the starting point to build up a linear SVM classifier as a baseline.

I. Linear SVM + Regularization

The first implementation of SVM is gradient descent optimization based on hinge loss. The hinge loss function used in LinearSVM measures the classification error of the model. It is defined as:

$$L(w, b) = \max(0, 1 - y_i * (w * x_i + b))$$

Hinge loss finds the optimal weight and bias by assigning a penalty to the dataset point. It assigns 0 penalty to correctly classified points, assigns ($0 <$ penalty < 1) to correctly classified by too close to the decision boundary, and assigns a penalty greater than 1 to misclassified points. Generated accuracy based on pure hinge loss, we then create the baseline for the rest of the improvement analysis. Adding regularization to Linear SVM is the first improvement choice. In

our implementation, we use gradient descent as the optimization method and experiment with three different slack variables in the regularization term - L1, L2, and a customized slack variable w^2. In Python code, Initialize the weight and bias to zero at the beginning for each class, and then start the iteration loop, repeating the value by feeding each data point into the hinge loss function, to check if hinge loss evaluates greater or equal to 1. We simply add the regularization terms to both the weight and bias after each iteration if the optimization constraint equation is violated, and only update the weight with the regularization term if the constraint equation is met.

II. One-vs-One method

Our dataset is a multi-class dataset, even though the vanilla SVM is a binary classifier, it could perform normally on multi-class datasets with some modification. There are two ways to implement SVM for multi-class classification tasks, One-Vs-Rest (OvR) and One-Vs-One (OvO) methods. In our previous linear SVM implementation, we used the One-Vs-Rest method, creating a binary SVM classifier between data in one class and the data in the rest of the class. The second improvement is changing from the OvR method to the OvO method in the linear SVM. Unlike OvR, OvO creates a binary classifier for every pair of classes. The difference between the two methods in Python implementation lies in the 'fit' method and the 'predict' method in the LinearSVM class. In the 'fit' method, instead of iterating over each class, we iterate over each pair of classes for binary classification in a single round. In the 'predict' method, instead of calculating the scores for each class, we pass the sample through all the binary classifiers. Each classifier will predict either one class or the other class from the pair of classes it was trained on. Count the number of times each class is predicted across all the classifiers.The final predicted class for the sample is the one with the highest vote count. In other words, the class predicted the most times by the binary classifiers is the class for one classification.

III. SVM Kernelization (dual problem)

Compared to Linear SVM implementation, we can confidently assume our dataset has a greater chance of being non-linearly separable since there are 10 genres. Therefore, our next improvement is by implementing the kernelization support vector machine. We achieve the optimization by solving the "dual" problem in the support vector machine in the kernel SVM class. The dual problem in Support Vector Machines (SVM) is an alternative formulation of the SVM optimization problem, which can be solved more efficiently in certain cases, especially when using kernel functions. The primal SVM problem aims to minimize the following objective function by minimizing 1/2 * ||w||^2 + C * Σξ_i with a constraint of hinge loss illustrated above. The dual problem of SVM can be formulated using Lagrange multipliers (α_i) to maximize the following equation:

$$\Sigma\ \alpha\_i - ½ * W, \text{ where } W(\alpha) = \Sigma_i\ \alpha_i - 1/2\ \Sigma_i\ \Sigma\square\ \alpha_i\ \alpha\square\ y_i\ y\square\ K(x_i, x\square),$$
$$\text{subject to } 0 <= \alpha\_i <= C \text{ and } \Sigma\ \alpha\_i * y\_i = 0$$

And the popular method for solving the dual problem in SVM is the SMO algorithm, developed by John C. Platt in his paper *Sequential Minimal Optimization* [7]. The SMO algorithm works by breaking down the dual optimization problem into smaller subproblems that involve finding the optimal values for only two Lagrange multipliers (α_i and α_j) at a time. Selecting two multipliers, α_i and α_j, based on whether they are violating the constraint formula introduced above. The algorithm iteratively updates the multipliers by Optimizing α_i and α_j while keeping all the other multipliers fixed, as well as updating the threshold (bias) b. The SMO repeats those two steps until the algorithm converges. This is usually determined by a predefined tolerance or

maximum number of iterations. In Python implementation, we initialize the Lagrange multipliers (alpha) to zeros and the bias term b to 0. These values will be updated during the running of the algorithm, and then we set up the Gram matrix K, which is computed using the selected kernel function (linear or RBF). This matrix stores the inner product between each pair of input data points, which is used in updating the alphas and bias terms. We start with the outer loop representing the number of iterations through the whole dataset, and we create an inner loop for each training example for the algorithm to check if the error Ei violates the constraints. If the constraints are violated, it proceeds to update alphas and the bias term. During the inner iteration, the algorithm will randomly select an index j not equal to i, and compute the error Ej. At the same time, it will also calculate the lower and upper bounds L and H for alpha_j based on the current values of alphas and the class label y. These bounds ensure that the updated alphas satisfy the constraints of the optimization problem. Next, if the lower bound and upper bound are equal, then skip the current iteration. Also if the second derivative of the objective function with respect to alpha_j is non-negative, skip the current iteration as well. We then update alpha_j using the formula:

$$\alpha\_j\_new = \alpha\_j\_old + (y\_j * (E\_i - E\_j) / eta)$$

This formula is derived from the objective function of the dual problem in SVM. The update aims to minimize the objective function while satisfying the KKT conditions[11] and the constraint $\sum(\alpha\_i * y\_i) = 0$. By updating α_j using this formula, the algorithm can adjust the decision boundary to better separate the classes and maximize the margin. Keeping the updated alpha_j within the range [L, H], we then check if the change in alpha_j is significant. If the change in alpha_j is too small (less than the tolerance), skip the current iteration. After this, we update alpha_i using the formula provided, calculating the new bias terms b1 and b2 using the updated alphas, incrementing the number of changed alphas in this pass. If no alphas were updated, increment the number of passes. Otherwise, reset the number of passes. In the end, after training is complete, we find the support vectors (data points for which alpha > 0), their corresponding alphas, and labels. In the OvO method, the algorithm will create a binary classifier for each class pair, and each optimal alpha value will be stored for each feature (Appendix F). After implementing the SMO algorithm, we then experiment with it using different kernel choices, including the linear kernel, polynomial kernel, radial basis function kernel, and Laplace radial basis function kernel.

Discussion

I. Comparing regularization in Linear SVM

In the Linear SVM part, some regularization improves the accuracy while others don't improve the accuracy compared to the pure hinge loss baseline with an accuracy of 0.6 (Appendix G). Especially for customized slack variable w^2, this slack variable makes the calculation overflow and generates 0.1 accuracy in the end. There are several reasons why adding regularizations doesn't improve accuracy or even decrease accuracy. The first

consideration is the imbalance of classes in the dataset. However, our dataset evenly split the 1000 data points into 10 genres, which is a perfectly balanced dataset. The second reason we can consider is because of the incorrect regularization parameter constructed. The customized w^2 is an incorrect slack variable because it decreases the accuracy score to 0.1. On the other hand, L1 regularization decreases the baseline accuracy down to 0.54, while L2 regularization slightly improves the accuracy from 0.6 to 0.62(Appendix G). It might be because of the different properties between the two regularizations. L1 regularization promotes sparsity, which means it encourages some feature weights to become exactly zero. This can be helpful when the dataset has a large number of irrelevant or redundant features. On the other hand, L2 regularization promotes smoothness, which means it tries to distribute the weights evenly among features. In our dataset, it could have feature information distributed across multiple features, because the dataset is evenly distributed, and each genre from a human-evaluated perspective is not that relevant to each other, meaning they could have important distinguished audio properties among each other. L2 regularization might be more suitable as it avoids completely discarding any feature.

II. PCA as feature selection & OvO method

In Appendix G we can see adding principal component analysis for feature selection doesn't decrease the accuracy performance for Linear SVM. It could be the same reason we mentioned above, that each class is evenly distributed, and each class has its unique audio properties. Therefore, removing some of their features decreases this kind of uniqueness, and hurts the performance. Another discovery is that One-vs-One performs better than One-vs-Rest because it breaks down the multi-class problem into several binary classifiers with high and low entropy.

III. Comparing Kernel in Dual Problem

Since our dataset has 10 genres, we assume it's non-linearly separable at the beginning. And based on the accuracy performance(Appendix H), can then now conclude that it is linearly separable to some degree, but it is more prone to have the property of non-linearly separable because kernelization yields better accuracy performance. The ranking is like the following: polynomial kernel < linear kernel < Laplace RBF < RBF. Linear Kernel assumes the data is linearly separable, and this is consistent with previous Linear SVM implementations because they reach about the same accuracy of 0.625. Our dataset is linearly separable to some degree. The polynomial kernel performs the worst, which has around 0.2 accuracy. It could be the inherent data structure problem because we can see even though we use a kernel like Radial Basis Function, its highest accuracy is still around 0.72 accuracy. Therefore, it is possible that we can't define the dataset as simply linearly separable or non-linearly separable. After 5-fold validation of testing regularization parameter C from 0.1 to 100 with an interval of 10 (Appendix I). We can see when C has reached 20, the accuracy reaches the highest. When C is greater than 20, accuracy decreases to around 0.72 and doesn't change a lot. I didn't discuss the runtime for different improvements because they both don't take a long time to train, and all of the different implementations take up to around 15 seconds to train, but they are implemented in the code.

Conclusion

In our experiment, we compared two split ratios on the dataset, which train:test = 4:6 and 8:2. Testing 4:6 is just consistent with our Bayes Classifier, we want to see that the basic tendency for improvement doesn't change greater because of a different set up for training and validating. However, for classifiers like support vector machines, a greater ratio for the training set could better train the model for generalization because it yields significantly higher accuracy

compared with fewer data for training. In terms of feature selection, applying PCA performs worse than not applying, even though I didn't test removing PCA on kernelization SVM, but It should be consistent because the reason causing PCA to perform worse is more likely to be that each feature adds up to the uniqueness of each class. Based on our result, we can see that multiple class datasets usually are more likely to be non-linearly separable, and choosing the right model to do the classification is important. Although our model reaches about the same accuracy as the SVM in the scikit-learn library, we suggest that for future tasks, using a prebuilt classifier could save more training time especially if you are dealing with a huge dataset.

NEURAL NETWORK

Introduction

Neural Networks are a machine learning technique that are designed to mimic the structure of biological neurons which underpin the cognitive abilities of humans. The core computational unit is therefore the single artificial neuron, which selectively fires based on the input it receives. In basic implementations, this is composed of weights, which set the strength of an input by multiplying the weight by it, and a bias, which is a constant number added to the sum of the products of the weights and the inputs. The final number is then analyzed by an activation function, which determines if the number is suitable for activating the neuron, usually by having it be above a certain threshold. Neural networks are normally paired with backpropagation, an algorithm to set the weights and biases based on the derivative of the error function, which compares the difference between the network's predictions and actual data and gives the difference. The derivative is then used to tell how to modify the weights and biases in the direction of decreasing loss. The amount of change is proportional to the learning rate, a variable that tells how much change in the values of the weights and biases to make in the direction of the derivative. Modifications, as will be discussed, can change how this is done but all the approaches used will still calculate the derivative and so this broad outline remains valid. Neural networks have been known in principle from the start of digital computers, but their large computational requirements have made them unsuitable for most practical tasks until the last decade, in which they have shown remarkable results in many diverse applications. We will now test their ability to classify the music dataset in this paper.

Implementation

There are four main implementations to discuss. The first is the prebuilt Keras neural network, keras6040.py. The second is the neural network from scratch with no alterations beyond the normal scope of neural networks, scratchNoRandomizationTimed6040.py. The third is the neural network from scratch modified to have randomization, RandomizationVersion1Timed6040.py. The fourth is the neural network with randomization of local changes in the weights and biases in a way similar to simulated annealing, randomizedAnnealingWIthBackprop6040.py. In each the implementation was originally started with a split of 80 percent testing and 20 percent training, but to synchronize the results created new versions split to 60-40, and all such changed results are labeled 6040. The results of the programs are copied into .txt files named after the .py that generated them."It is more cumbersome to navigate the large numbers of .py files rather than putting them all in one file, but this became a practical necessity due to the long training times of each.

The prebuilt Keras neural network utilizes a gradient descent optimizer called Adam[8]. It is comparatively memory efficient compared to regular backpropagation and uses moments. Adam stands for ADAptive Moment estimation, which might be considered a case of overfitting the acronym. Adam computes the gradient of the objective function on mini-batches of data and uses them to estimate the first and second moments of the gradient.[9] A moment is a variable used to predict the gradient, and in Adam includes the mean and variance. These moments are then used to scale the gradient and adjust the learning rate for each parameter. Adam is impressively fast at training but gets criticism for not translating that speed into better performance over time compared to other methods.[10] However, in the more limited resource and time and data-constrained problem in this paper, that is not so much of an issue as would be the case when these things are in more abundance.

Before usage, the data is normalized. When the network is run without the normalization, the results converge to a single class of music every iteration, essentially meaning the neural network could not even begin to traverse the space of answers correctly and defaulted to labeling every piece of music a single genre. Normalization fixed this by reducing the range of feature values. 58 features are given as the input variables, which are taken from the dataset. The label and filename are removed. The model is composed of 4 layers of 128, 64, 32, and 10 layers each, the concept being that the network should be filtering the data through decreasingly complicated "tests" that each layer represents. It should be noted that there is no feature extraction necessary by the neural network as we are starting with features pre extracted from the raw data, and to have, like in autoencoders, a small middle layer in the neural network to force feature extraction would in this case be getting features from features, which is wasteful. The model is trained with a batch size of 32 to speed up calculations. Although the parameter epochs are given as 1, the entire training is contained in a for loop of 1000 epochs. The reason for this is to be able to get intermediate results from the training after each epoch without relying on any of the prebuilt reports. The reason 1000 was chosen is that originally it was to be 10000 to match what was done in the other implementations, but for Keras, this takes far longer and over 55 minutes of runtime, so it was lowered to 1000. However since Keras finds numbers that it does not break out after long periods of training time, this still shows what Keras would result in if given more time as even in the longer time the results were the same as after the first 1000 iterations. The model optimizes based on its training data, then is evaluated based on the test data.

The accuracies are how well the neural network performs on the test data by how many correct classifications over total classifications. The loss is how well the network performs on the error function from the training data alone. Each is given in intervals of 100 to reduce the size of the output and also to get a less cherry picked version of results. So if an accuracy is said in this paper for a neural network to be 73, that was the accuracy that just happened to be on the interval of 100 epochs that was the highest shown in all the iterations, not the best accuracy ever obtained in the test set. This reduces the chance of outliers which are likely unstable configurations if they cannot last over 100 instances. It is common to see the neural networks overfit on the train data and so it is important to pick the accuracy from before this occurred rather than its final one, which is what is done in this paper.

The runtime of the 6040 version is about 519.54 seconds in google collab when using 1000 iterations. For the split of 80 and 20, Keras can often result in accuracies of between 73 and 75 at best and then usually converges to lower accuracies as it likely overfits the testing data. There are some situations where it shows a surprising performance of 78, but these situations are not due to any change in the code but just running the same code many times which will

sometimes outperform compared to other times. This cannot be replicated. A few times it did not degrade and maintained a roughly 75 percent test accuracy to the end, showing the possibility of either a good neural configuration or somehow a better training testing split than normal. For 60-40, it does slightly worse, around 72 at its best, but shows the same overfitting and characteristics as the 80-20. In the testing of various splits beyond the mentioned, the more data was given to training, the better the network performed. See Figure K.

The neural network from scratch was made to resemble keras as closely as possible. However, it differs significantly in optimization due to the inability to make a program that resembles the Adam optimizer in such a short time. It instead uses a pure derivative-based gradient descent method and utilizes a static learning rate without decay. The neural layers, the number of neurons, are kept the same. The number of epochs is set to 10000 because the program tends to show long periods of no improvement before this period of time. This clearly showed a worse performance than Keras, getting in an 80-20 split with a best of around 71 percent accuracy. It also showed a long period of the first 1000 or so epochs showing no improvement on the test set, but which afterward improves rapidly. This is likely due to a static learning rate, but because this does not impact the final accuracy of the result, it was not a priority to change. However, this is a clear thing to optimize if the algorithm were to be deployed in a real-world use case. It is interesting to note that this code seems "resistant" to different outcomes based on different training sizes, which other code does not exhibit, but as the performance is worse this is not useful. In the 60-40 it still shows an accuracy that is very similar at 70.25. The runtime is 618.10 seconds in google collab. See Figure L.

Improvement

The improvement of the neural network from scratch with randomization added a new component at the beginning of the neural network labeled randomization. In it, a period of time was used to generate random numbers to fill the neural network, which was then evaluated against the train data set. Instead of being back propagated, this process is repeated and the best-performing models on the random numbers are saved and used when the random numbers are finished. The assumption is that this process allows the model to start with a series of weights and biases suited for the problem better than wholly random numbers would tend to be, without having them subject to issues of overfitting and falling into local optimums because they are not back-progoated, which is assumed to cause this. This process is printed out with the evaluation by the test dataset to see how the lowering loss on the training dataset compares with the testing dataset and shows the result that the longer randomization occurs, the worse the random numbers tend to do for the testing dataset. This can be explained by the fact that the better the random numbers do on the training dataset, the more likely they are to start to overfit for just that. Therefore having too few random numbers would result in the degraded performance of the original implementation by being no different from the underperforming original, but too many would have bad performance as well, meaning there must be a middle point in which the random numbers tend to do better. In testing, it is found that this tends to be between 100 to 1000 random numbers in the dataset, but the nature of randomness means that the program must be run multiple times to ensure that a better representation is going to be obtained even within this band of possibilities. The best result for 60-40 was 72.25, matching Keras with a google collab runtime of 618.45 seconds. On 80 20 it was a .75, once again matching Keras. See Figure M. Of additional note is that unlike other results, there was a tendency for high test accuracies to be sustained for longer periods of time, and only degrading slowly by overfitting, suggesting that configurations from best performing random numbers tend to generalize better and overfit less.

The final implementation was to combine the randomization of weights at the beginning, using the 100 iterations found to work well, with probabilistic local randomization of each layer separately. What this means is that random numbers were at random times added to each layer's existing weights. If the change was bad, backpropagation should remove it, and it should also break local optimums during runtime while still allowing global improvement in the algorithm. In other words, while it is expected for backpropagation to converge, the randomization of layers prevents this and always preserves a chance for greater best performance the longer the program runs. The exact number and method of this was derived from testing, however due to time constraints it is highly likely that the configuration currently used is not necessarily the best. It has an independent 50 percent chance of randomizing any layer every 100 epochs, which should lead to half of the layers on average being changed to a random amount. The changes are based on adding the random numbers instead of multiplying them, which should result in more stable and small changes compared to multiplication. The 50 percent chance was initially modified to drop lower as time went on, but it found that it seemed to show better results if this was kept static in the limited testing. The biases being added to zeros is unneeded, but preserved for the sake of being able to be modified to other values in future testing. This approach is more prone to being unable to "hill climb" to a local optimum minimal loss, but that is considered a benefit as maximizing the loss function of the train data is clearly shown in the data to reduce the accuracy on the test dataset likely from overfitting. Like the other improvement, its reliance on probability makes it necessary to run it multiple times to definitely see the better results mentioned, but quickly shows them as this is done. This process could perhaps be simply construed as a single program which has multiple copies of the same code and of which the best configuration of weights and biases is used, rather than one program that doesn't work everytime but does upon multiple times.

Although this worked astonishingly well once, showing a 78 percent classification accuracy, this could not be repeated and would not consistently show such good results, instead going between 73 and 75. On one rare occurrence Keras showed the same result, but replicating this proved similarly impossible. Although the previous randomization approach in this paper does not always show results similar to what their best was, once run multiple times the best results shown in the data recurred in all upon a few runs, whereas the 78s were not seen again no matter how many retries. This hints at intermediate randomization as being potentially useful in addition to initialization randomization, but the inability to reliably generate these results means that the current approach would have to be run an infeasible number of times to try and get them, which would be too computationally demanding to realistically use. Unfortunately, there is not enough time to investigate this further. At 80-20, it shows performance equal to Keras at 75. In the 60-40 splits, this approach resulted in 72 percent accuracy, also equal to keras, with a runtime of 799.25 seconds on Google Colab. See Figure N.

Conclusion

In the 60-40 split, Keras performed at 72, the basic implementation at 71, the randomized implementation at 72, and the simulated annealing-based approach at 72. Both the improvements have been shown to be just as good as Keras is, but have not convincingly demonstrated themselves to be better in final accuracy. However, the comparison with the 80-20 split data shows that the basic implementation is far worse than this data would suggest in comparison and that at least Keras and the annealing approach have some chance of being substantially better from the 78s witnessed. This makes these two approaches the most attractive in comparison with the others, but Keras has a big advantage in the speed of training. Although the simulated

annealing randomization approach may have a higher chance of being substantially better by chance, most applications and teams don't have the luxury of being able to try many times hoping for better results and so Keras would be recommended for the average use case. However, for research and absolute performance, the better performance of the randomization approach and the potential of the annealing combined with this would probably be able to lead to better results if they were developed further. If accuracy and not runtime were the primary concern, the best choice would be simulated annealing combined with randomization because it performs as well as all the other techniques but likely has a better chance of having substantially better results due to being able to break out of local optimums over time due to the randomization of layers which Keras with the Adam optimizer cannot do and the randomization in the beginning only cannot do either after the program starts as it cannot randomize at any other time. e. Although the annealing approach could have been made more stable or consistent in results, after testing the version with the highest average performance was kept, which is the one discussed, and it should be expected to replicate its ability to generate very good configurations with higher probability than other approaches based on this evidence or just as well as the others, making its average results likely to be higher than the other techniques. However, the relative improvement from the basic implementation made by both improvements, which share the same code except for the alterations, raise the possibility that the same techniques might show further improvements on any or many neural network optimizer techniques. If the "bad" original implementation in this paper was turned into one as good as keras using Adam optimizer, perhaps a better implementation could be turned into an even greater model. The size of the dataset and significant improvement upon expanding the dataset from 60-40 to 80-20 and results implies that larger datasets and more diverse use cases will better show the benefits and limitations of these new approaches and would be useful to investigate. Methods that improve the probability of the significantly better results seen only sometimes in the training further would be of great importance, as would be alterations to improve the time it takes to achieve this. This would also allow for a better verdict as to whether the performance of the improvements in accuracy is only just equal to Keras using the Adam optimizer, which is proven, or better, which is still uncertain.

CONCLUSION

The project compared three classifiers for music genre classification and found that the neural network (NN) was the best classifier, likely because it could learn a general representation of the data and relate it to the final labels with that general model. Figure Z shows that the NN had higher accuracy than the other two classifiers. However, NNs also had longer running times and might not be suitable for larger datasets without significant amounts of computational power and time. The randomization alone did not significantly impact the neural network training times, but simulated annealing done randomly on each layer caused a large increase in runtime, pointing out the possibility that this method could have problems scaling. The generalized model of the data and their relationships to the labels we are ultimately trying to predict is likely more suitable to NN because it lends itself to a computable model of relationships between data and labels more so than it does to being expressed from inherently random processes in bayes and data separable by distance in SVM. The relationship between features and labels is not inherently random or independent, which explains the difficulty of Bayes in predicting the data to some degree. SVMs categorize data by some metric of distance, but relationships between objects that are far away according to some metrics of distance can be relevant in others ways, and the neural

network with its interconnected neurons can better represent this. Having more initial neurons in the first layer than there were features enabled the possibility of having multiple representations of the features in different ways, though due to the opaque nature of the neural network it is unclear if this mattered or not.

The support vector machine (SVM) classifier was the second-best classifier, which had similar accuracy to the NN but with faster computation times, especially for larger datasets. SVMs were flexible in fitting models of different dataset structures and were relatively insensitive to feature relationships. However, the performance of SVMs could be sensitive to the shape of the dataset, and the training and testing data split could greatly affect their performance. This faster computation time can easily become important in many situations, as neural networks could have exponential growth of neurons and computation time if the original features get far larger whereas SVMs can handle this increase in features without nearly the same penalty.

The naive Bayes classifier was the worst classifier of the three, but it still performed well compared to the other models, especially for quick cross-validation and feature selection validations. However, the classifier assumes independence between features, which is an important drawback because features from music data are not really independent of each other. Moreover, Bayes relied on estimating probabilities and making assumptions based on training data, so its performance could suffer if the training data was small as was the case in the dataset. In runtime, Bayes took ¼ of the time of the SVM, meaning it is highly suitable for quick cross-validation in terms of feature selection validations. The high "Performance vs Time" ratio also makes it suitable to be used on far less powerful devices which often have a choice of using an easy-to-compute machine learning model or none at all.

Future work could include using Bayes Nets and comparing them to our previous Bayes and NN models, experimenting with different training and testing splits, experimenting with different hyperparameters of the implementations, and comparing them on another dataset. By improving and evaluating our models further, we can build more accurate and efficient classifiers for both this project and machine learning in general.

APPENDIX

Figure A) - Bayes Classifiers

Distribution	Formula	Accuracy	Runtime
Gaussian	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$	51.67%	0.006
Uniform	$f(x) = \begin{cases} \frac{1}{b-a} & \text{for } a \le x \le b, \\ 0 & \text{for } x < a \text{ or } x > b. \end{cases}$	29.67%	0.0046
Rayleigh	$f(x;\sigma) = \frac{x}{\sigma^2} e^{-x^2/(2\sigma^2)}, \quad x \ge 0,$	23.00%	0.0059
Premade Gaussian	$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$	39.33%	0.0063

Figure B) - Traditional Greedy Feature Selections with Gaussian Bayes Classifier

Type	Number of Features	Features	Accuracy	Runtime
Without AIC	9	[6, 26, 0, 1, 14, 2, 49, 55, 51]	55.33%	0.8344
With AIC	3	[5, 40, 2]	35.17%	0.3837

Figure C) - Greedy Feature Selection Random with Gaussian Bayes Classifier

K	Number of Features	Features	Accuracy	Runtime
3	18	[2, 49, 47, 23, 55, 26, 16, 38, 1, 30, 3, 51, 0, 34, 27, 28, 14, 6]	58.67%	4.1007
4	20	[6, 1, 14, 49, 2, 42, 3, 30, 27, 36, 47, 16, 32, 28, 23, 51, 34, 0, 45, 10]	59.50%	3.3471
7	25	[6, 26, 0, 49, 55, 51, 16, 42, 32, 3, 2, 27, 38, 47, 36, 39, 37, 30, 23,	60.33%	3.9502

		13, 24, 1, 14, 15, 48]		
10	26	[6, 14, 2, 49, 55, 51, 16, 42, 32, 3, 27, 38, 39, 47, 36, 37, 1, 30, 23, 13, 24, 15, 48, 40, 34, 0]	59.00%	2.9512
20	28	[6, 26, 0, 14, 2, 49, 55, 51, 16, 42, 32, 3, 27, 38, 39, 47, 36, 37, 30, 1, 23, 13, 24, 15, 48, 40, 34, 17]	60.17%	2.6342

Figure D) - Greedy Feature Selection Pairs or Triples with Gaussian Bayes Classifier

Num_Initial	Number of Features	Features	Accuracy	Runtime
.3	17	[19, 1, 3, 32, 14, 55, 31, 26, 49, 54, 38, 45, 9, 7, 50, 22, 20]	46.83%	2.4933
.4	21	[0, 1, 17, 30, 36, 6, 54, 14, 37, 51, 42, 55, 40, 45, 31, 9, 25, 44, 19, 50, 48]	48.33%	7.2984
.6	34	[17, 49, 0, 1, 27, 30, 21, 42, 5, 45, 55, 34, 51, 53, 47, 33, 48, 2, 11, 56, 10, 29, 37, 7, 19, 35, 50, 52, 25, 13, 12, 6, 8, 46]	43.17%	45.5737
.8	45	[15, 1, 8, 51, 23, 28, 21, 32, 16, 17, 27, 14, 10, 45, 36, 55, 39, 3,	50.17%	128.6867

		24, 53, 26, 56, 49, 9, 2, 22, 42, 7, 31, 11, 41, 33, 40, 18, 4, 25, 52, 29, 44, 35, 46, 6, 47, 5, 50]		
1	56	[0, 26, 6, 23, 1, 2, 21, 49, 16, 30, 28, 55, 32, 53, 54, 39, 45, 3, 14, 38, 24, 48, 31, 9, 15, 37, 13, 43, 56, 34, 47, 29, 36, 42, 51, 10, 41, 25, 40, 27, 4, 8, 22, 52, 19, 33, 11, 12, 18, 7, 17, 44, 5, 20, 46, 35]	52.17%	356.0134

Figure F) - Binary Classifier Pair for Each of the Genre Using One-Vs-One SVM

Alpha for Binary SVM classifier (8, 9)

```
[0.3738384  0.20018296 0.27684861 1.21042828 5.3557466  0.09888868
 1.91060296 2.57979686 0.77736169 0.1862714  2.45377125 0.58988299
 3.69160343 1.11688435 1.00997871 1.16222019 0.66607507 0.38304928
 0.89914645 1.48523014 0.47618427 2.29633126 0.52000839 0.46935546
 0.31727736 0.30238796 1.34418775 2.33189171 1.25531162 0.3803284
 2.2132091  0.19112954 0.14706149 0.19901027 0.47544247 0.33671706
 0.98326229 0.84900271 1.16330699 0.64597183 1.17269242 1.9001523
 1.4775367  0.41271704 0.55638129 0.19134911 0.38328739 3.78090809
 1.36247215 2.26280772 0.01510773 0.77572039 1.04297092 3.20828343
 0.57793411 0.0572376  0.61556022 0.80746306 1.61729198 0.61899295
 1.34938655 1.77670629 0.18740413 4.05552745 1.98743662 2.99531328
 1.76195173 0.13807441 0.31403648 0.37367164 0.02552752 2.16867865
 1.04769247 0.58844743 0.66664231 0.78753222 1.94876232 0.46191633
 0.7320111  3.47469645 1.33218982 1.03660702 0.42338345 1.39133139
 1.13445481 1.20547779]
```

```
(0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9)
(1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8) (1,9)
(2,3) (2,4) (2,5) (2,6) (2,7) (2,8) (2,9)
(3,4) (3,5) (3,6) (3,7) (3,8) (3,9)
(4,5) (4,6) (4,7) (4,8) (4,9)
(5,6) (5,7) (5,8) (5,9)
(6,7) (6,8) (6,9)
(7,8) (7,9)
(8,9)
```

Figure G) - Baseline Performance, PCA, and Slack Variable Choice on Linear SVM Model

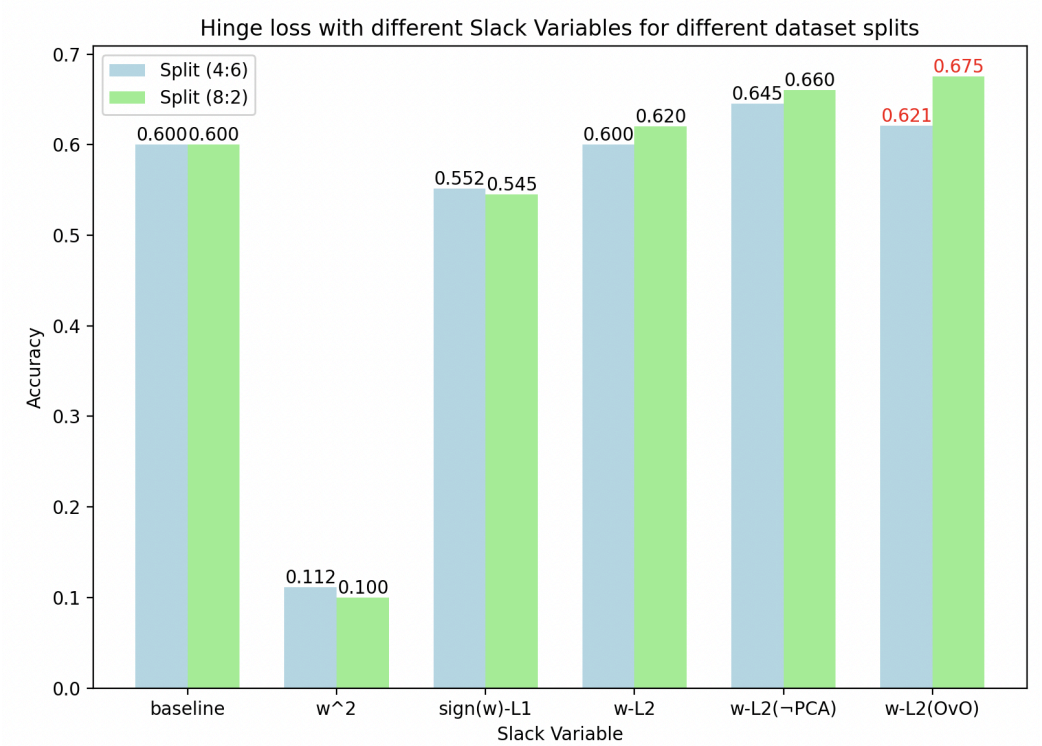

Figure H) - Difference Performance on Kernelization SVM With SMO Algorithm

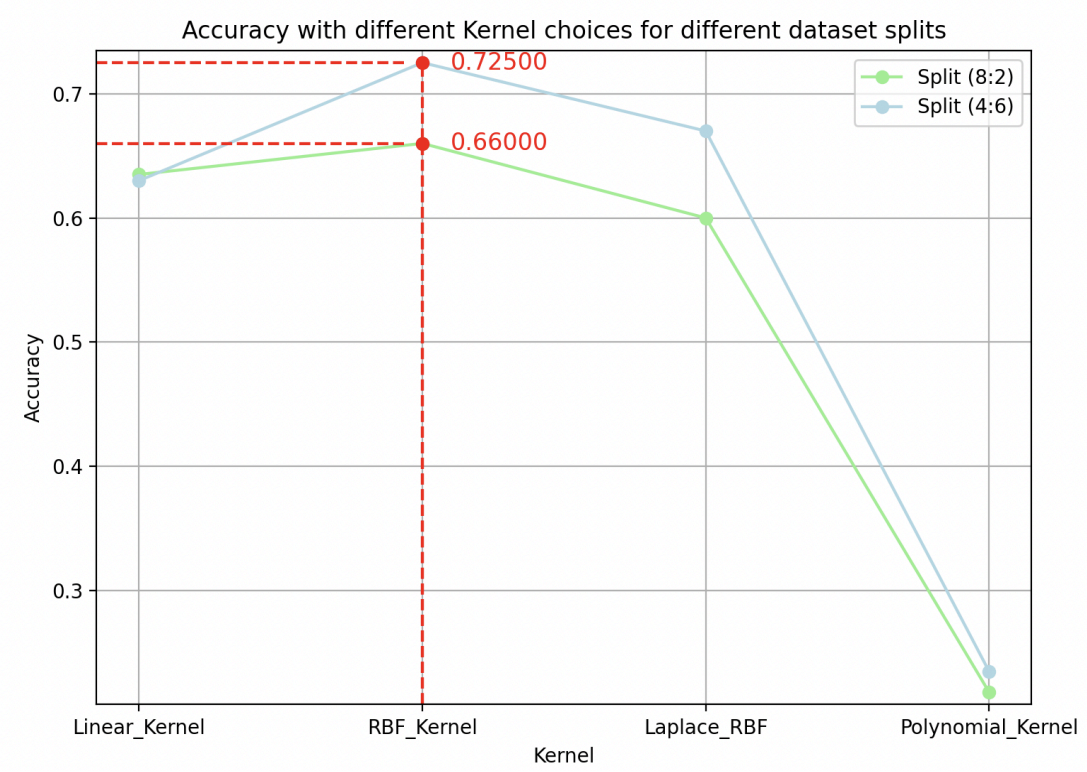

Figure I) - Accuracy Performance With Different Regularization C Parameter Value Under Radial Basis Function Kernel

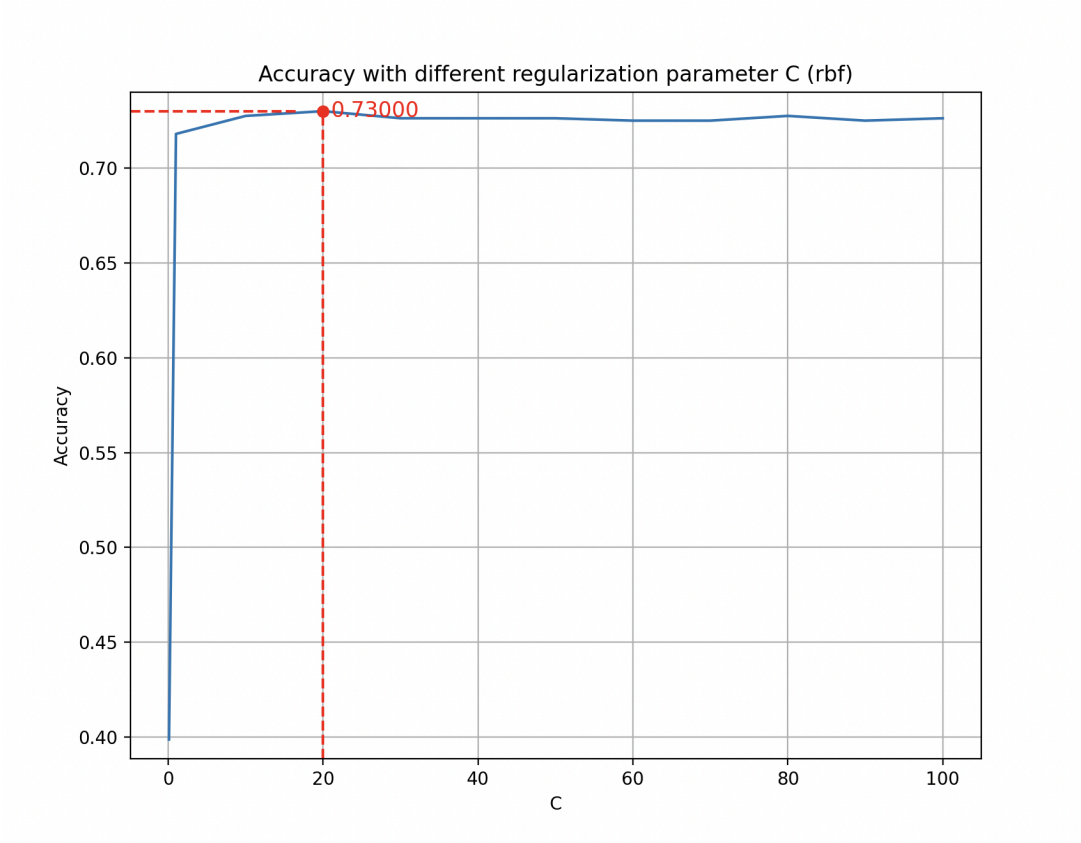

Figure J) - Five Different Amounts of Randomization

Note how the worst performance is around .71, which is what the original implementation without randomization trended towards. This is for an 8:2 split.

Randomization Amount	None	10	100	1000	10000
Best Test Accuracy	71	71.5	74.5	72	70

Source file: #Code and results of the 5 different randomization amounts, scroll down for results.txt

Figure K) - Text Files for Keras

```
13/13 [==============================] - 0s 834us/step
13/13 [==============================] - 0s 2ms/step - loss: 2.3430 - accuracy:
0.7500
Test accuracy: 0.75
13/13 [==============================] - 0s 833us/step
```

Source:
kerasResults.txt

```
13/13 [==============================] - 0s 2ms/step
13/13 [==============================] - 0s 2ms/step - loss: 1.2162 - accuracy: 0.7250
Test accuracy: 0.7250000238418579
```

Source:
kerasResults6040.txt

Figure L) - Text Files for scratch implementation

```
Test Accuracy: 0.71
Epoch 20400, Loss: 0.016971649721136357
```

Source:
scratchNoRandomizationResults.txt

```
Epoch 17000, Loss: 0.1362986512769347 7
Test Accuracy: 0.7025
```

Source:
scratchNoRandomizationResults6040.txt

Figure M) - Text Files for Randomization Approach

```
Test Accuracy: 0.7225
Epoch 16600, Loss: 0.13461914012156445
```

Source:
RandomizationVersion1Timed6040.txt

```
Epoch 18700, Loss: 0.029400243733884928
Test Accuracy: 0.75
Epoch 18800, Loss: 0.0261707430216565
Test Accuracy: 0.75
Epoch 18900, Loss: 0.023414162994564797
Test Accuracy: 0.75
Epoch 19000, Loss: 0.02114845011197213
Test Accuracy: 0.75
Epoch 19100, Loss: 0.0191601554604074
```

Source:
RandomizationVersion1Timed.txt

Figure N) - Text Files for Randomization Combined With Random Layer Simulated Annealing

```
Test Accuracy: 0.72
Epoch 23600, Loss: 0.0031839588588426 02
Test Accuracy: 0.72
Epoch 23700, Loss: 0.0030488902895037017
Test Accuracy: 0.7175
Epoch 23800, Loss: 0.0029895882762532534
Test Accuracy: 0.7225
Epoch 23900, Loss: 0.002933895230355451
Test Accuracy: 0.72
Epoch 24000, Loss: 0.0027780734695951204
```

Source:
randomizedAnnealingWithBackprop6040.txt

The special run:

```
Test Accuracy: 0.77
Epoch 9100, Loss: 0.025122022904386507
Test Accuracy: 0.77
Epoch 9200, Loss: 0.02159096959030047
Test Accuracy: 0.775
Epoch 9300, Loss: 0.018868945828721877
Test Accuracy: 0.78
Epoch 9400, Loss: 0.01676764286724802
Test Accuracy: 0.78
Epoch 9500, Loss: 0.01507720073391945
Test Accuracy: 0.785
Epoch 9600, Loss: 0.01338983002506135
Test Accuracy: 0.78
Epoch 9700, Loss: 0.012078230029806165
Test Accuracy: 0.78
Epoch 9800, Loss: 0.011130302344636531
Test Accuracy: 0.78
Epoch 9900, Loss: 0.01008200441175982
Test Accuracy: 0.78
Epoch 10000, Loss: 0.009143316996220159
Test Accuracy: 0.78
Epoch 10100, Loss: 0.00848616035894549
Test Accuracy: 0.775
Epoch 10200, Loss: 0.00786237201362752
```

Source:
randomizedAnnealingWithBackprop.txt

The best of the second run:

```
Test Accuracy: 0.745
Epoch 26400, Loss: 0.0023312528786986 94
Test Accuracy: 0.75
Epoch 26500, Loss: 0.0023224165912979853
Test Accuracy: 0.745
Epoch 26600, Loss: 0.0023354852879084475
```

Source:
randomizedAnnealingWithBackpropSecondResult.txt

Figure Z) - Leading Contenders From Each Classifier (60-40 Split)

Model	Accuracy	Runtime(seconds)
Gaussian Bayes Classifier With Random Greedy Feature Selection	60.33%	3.9502
Support Vector Machine With SMO algorithm and Radial Basis Function Kernel	68%	12.888
Neural Network With Randomization and Simulated Annealing	72%	799.25

REFERENCES

1. https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification
2. https://en.wikipedia.org/wiki/Normal_distribution
3. https://en.wikipedia.org/wiki/Continuous_uniform_distribution
4. https://en.wikipedia.org/wiki/Rayleigh_distribution
5. https://docs.rapidminer.com/latest/studio/operators/modeling/optimization/feature_selection/optimize_selection.html
6. https://en.wikipedia.org/wiki/Akaike_information_criterion
7. John C. Platt, Sequential Minimal Optimization https://www.microsoft.com/en-us/research/uploads/prod/1998/04/sequential-minimal-optimization.pdf
8. [1412.6980] Adam: A Method for Stochastic Optimization (arxiv.org)
9. Overview of various Optimizers in Neural Networks | by Satyam Kumar | Towards Data Science
10. Adam — latest trends in deep learning optimization. | by Vitaly Bushaev | Towards Data Science
11. https://en.wikipedia.org/wiki/Karush%E2%80%93Kuhn%E2%80%93Tucker_conditions