CS 4641 Machine Learning
Name Redacted — Section B Final Report

1 Introduction to Dataset

1.1 Music Features from Kaggle (← link)

Every year I look forward to Spotify's end of year wrap up. As a consumer of sounds, I love seeing which "conventional categories," AKA genres, of music I sink thousands of hours in. This past year a new one made itself to the top of my list: Cpop. After some research, that stands for Chinese Pop. I've literally never heard of songs being labeled in this kind of genre before which led me to wonder how can someone categorize all the various kinds of music in the world.

TODO: Feature description. When I found this dataset, I was already interested in seeing if machine learning could analyze the sound waves of a piece of music and determine it's genre. However, that seems a too big of a computational task so I decided to use the features that are provided by this dataset. This dataset contains **1,000 datapoints** with **30 features per datapoint**. To prove that the features describe a piece pretty well, here are some of the features:

- tempo: speed at which a passage of music is played
- beats: basic unit of time. Think about it as the rhythm you would tap your foot to when listening to a piece
- chroma_stft: The Short Time Fourier Transform can be used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes overtime (kind of like those visualizers that jump up and down)
- 20 mel-frequency ceptral coefficients that make up a mel-frequency cepstrum: representation of the short term power spectrum of a sound

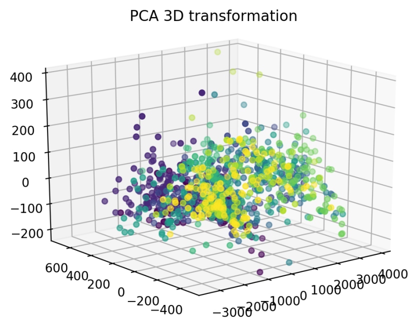
1.2 Supervised Learning Problem

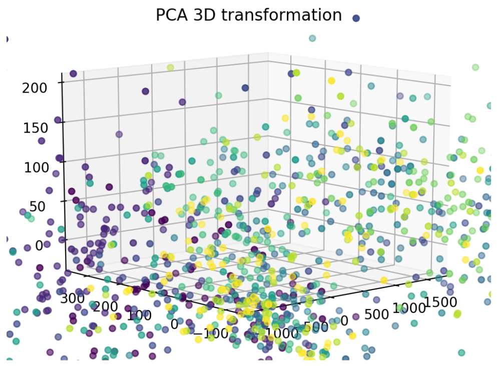
The underlying supervised learning problem is to **classify the genre** of pieces of music using the features of it's waveform. This dataset is **not trivial**. At first glance, there is no feature that determines the genre of a song trivially. To prove the non-triviality of this dataset, I tried to fit the centered data using sklearn.svm.LinearSVC (with a one-vs-the-rest multiclass scheme) 100 times and averaged the results.

- avg testing accuracy: 0.62
- avg training accuracy: 0.71

The training accuracy is not high enough to prove that the dataset is linearly separable. But to further research the triviality of the dataset, I did a few more experiements:

- 3D PCA Plot





  The graph on the right is a zoomed in version of the left graph. It is clear that there are many data points mixed together and the dataset is at least not linearly separable in 3 dimensions.

- Linear Regression: I one-hot-encoded the labels and tried to fit a linear regression. The testing and training accuracy was 0.26 and 0.31 respectively. This serves as more evidence that the dataset is non-trivial.

After these additional experiements, I have convinced myself that my **dataset is non-trivial**.

1.3 Performance Metrics

I will be scoring each classfier using cross validation on the testing dataset to obtain k independent scores. These scores represent the accuracy at which the classifier sucessfully classified each data point. I will also be further exploring results with a **confusion matrix** in a later section. I'll be using the cross validated scores to create a confidence interval that will help in comparing the classifiers to each other.

2 Description of Algorithms

2.1 Random Forests with Bagging

A random forest consists of a large number of individual decision trees that form an ensemble. An ensemble bascially uses multiple algorithms to obtain more accurate predictions than any one individual of the ensemble. In a random forest, the ensemble is made up of relatively uncorrelated decision trees. Bagging (Bootstrap Aggregation) is a method that allows each decision tree in the random forest to randomly sample the dataset with replacement. Since decision trees are very sensitive the training data, this greatly increases the variablility of the decision trees. Here are the hyperparameters I will optimize:

- n_estimators: [100, 200, 300, 400, 500]
  Number of trees in the forest. Affects the size of the random forest.
- max_depth: [100, 200, 300, 400, 500]
  The maximum depth of each decision tree. affects the complexity of each tree.
- max_features: ['sqrt', 'log2', None]
  The max size of the random subsets of features to consider when splitting a node.

2.2 Support Vector Machines with a non-linear kernel

SVMs attempts to separate datapoints by its labels by finding the best hyperplane using the large margin principle. For non-linear data, it can use the kernel trick to map the data to a higher dimension in hopes of finding a more suitable hyperplane. However, SVMs are inherently binary classifiers so it is not suitable for my dataset that has 10 unique labels. I will be using the **one-versus-all** classification technique to train my multiclass SVM. Here are the hyperparameters I will optimize:

- C: [1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03, 1.e+04]
  Regularization paramenter represents how much I want to avoid misclassifying each training example. Large C will choose the a plane that correctly classifies the most training examples. Small C will choose a plane with a larger margin even if that plane wrongly classifies more training examples
- kernel: ['poly', 'rbf', 'sigmoid']
  The kernel type to use when mapping the data to a new dimension space. This will affect the separablility of the data after it is mapped to a new dimension.
- gamma: [1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03]
  Kernel coefficient for rbf, poly, and sigmoid.

2.3 Neural Network

A neural net is basically a bunch of algorithms, designed and modeled loosely like a human brain, that can **recognize patterns**. They all start with an input layer, which can be thought of an array of inputs such as numbers, rgb values, words, etc. The input layer is **densely connected** to the first hidden layer and the first hidden layer to the next hidden layer. There can be many hidden layers and each layer is made up of some amount of neurons. Each neuron's job is to compute a **weighted average** of its input, and this sum is passed through a non-linear activation function such as a sigmoid or ReLU. Eventually the last hidden layer will connect to an output layer responsible for outputting the prediction. The entire neural net is trained using a technique called **back propagation**, which calculates a gradient descent to nudge the weights for a perceptron towards its theoretical optimal value. Here are the hyperparameters I will optimize for my nerual net:

- hidden_layer_sizes: [(10, 10, 10), (20, 20, 20), (10, 10), (20, 20)]
  A tuple of length (number of layers - 2) where the element at position i represents the number of neurons in hidden layer i.
- activation: ['identity', 'logistic', 'tanh', 'relu']
  The activation function for the hidden layer.
- solver: ['lbfgs', 'sgd', 'adam']
  The solver for weight optimization for each neuron
- apha: [0.0001, 0.001]
  The L2 regularization penalty parameter. This penalizes larger weight values to favor smaller values in the weight matrix. This in turn decreases the effect of the activation function, resulting in a less complex function. We want a less complex function to avoid overfitting
- learning_rate_init: [0.01, 0.001]
  For my purposes, this will be the learning rate for the entirety of the training because I will be using a constant learning rate for all trials to limit the complexity of this GridSearch.
- max_iter: [500, 1000] $\rightarrow$ [2000, 3000]
  The limit on the number of iterations of training. The solver iterates until convergence or when it has reach the max number of iterations.

3 Tuning Hyperparameters

I will be using **Nested k-Folds Cross Validation** for hyperparameter tuning. In Nested k-Folds Cross Validation, I will split data randomly into two halves, X and Y. Then I will perform k-Folds Cross Validation on X to tune for optimal hyperparameters. I'll be using sklearn's **GridSearchCV** for hyperparameter tuning. Then I will take the hyper parameters tuned from X and score it using k-folds cross validation on Y. Then I will switch X and Y and repeat the process. I chose k=10 because my data has 10 different labels and this will give me 20 data points to calculate a confidence interval from. Here is a short step by step guide to my process:

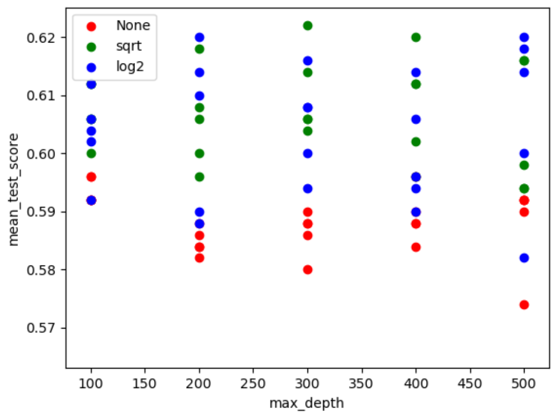
1. Split dataset into X and Y
2. 10 folds cross validation on X and tune hyperparameters
3. Use hyperparameters from 2. to create classfier C
4. Split Y into 10 folds. For each fold i: train C on all other folds and score on current fold i. This gives 10 scores
5. Switch X and Y and repeat steps 1 to 4. This gives 20 scores in total
6. Use the 20 scores to calculate confidence interval for the classifier

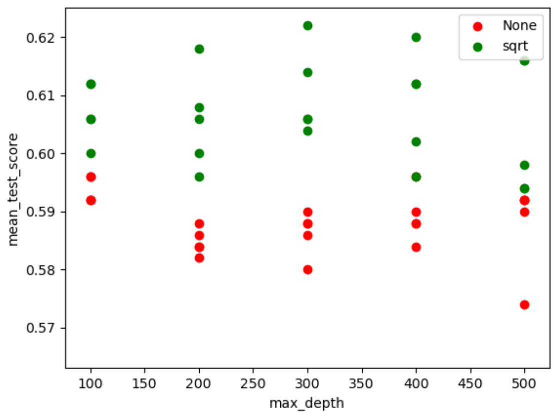
3.1 Tuning Random Forests

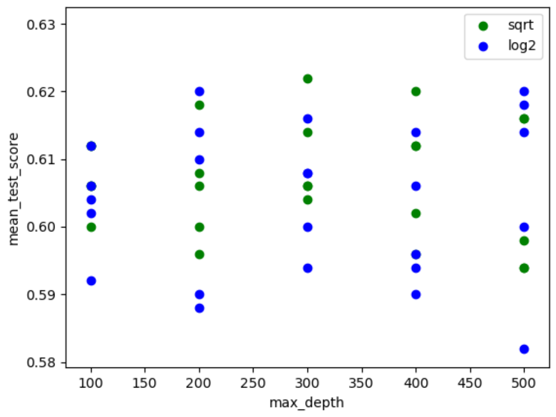
Here is the set of hyperparameters that I tuned for Random Forests:

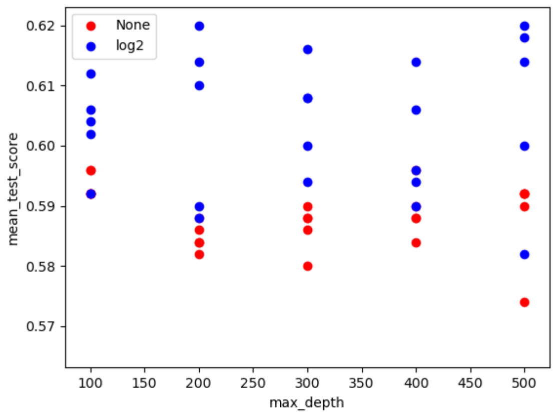
- n_estimators: [100, 200, 300, 400, 500]
- max_depth: [100, 200, 300, 400, 500]
- max_features: ['sqrt', 'log2', None]

Each GridSearchCV on a split half the size of the data with these hyperparameters took about 184 seconds so the whole experiement took **368s** to run.
Let us analyze this first batch of graphs where the x-axis is max_depth, the y-axis is mean_test_score, and the colors correspond to max_features:



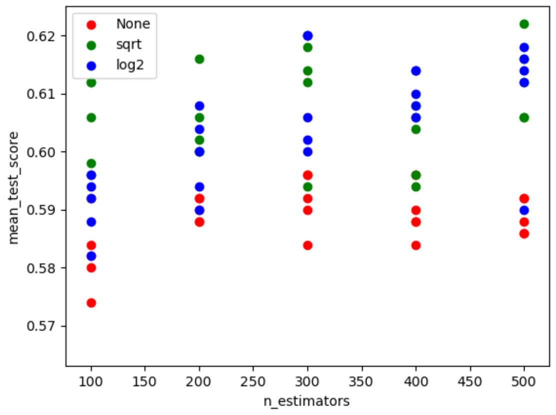


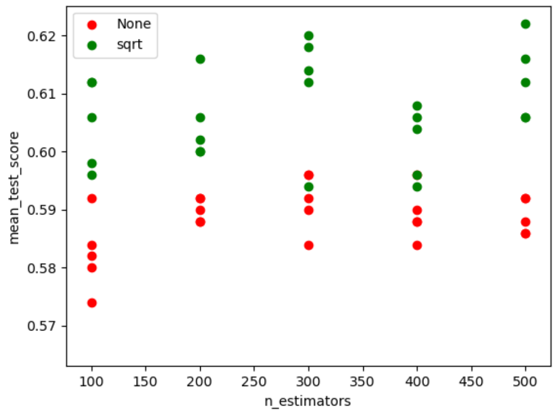


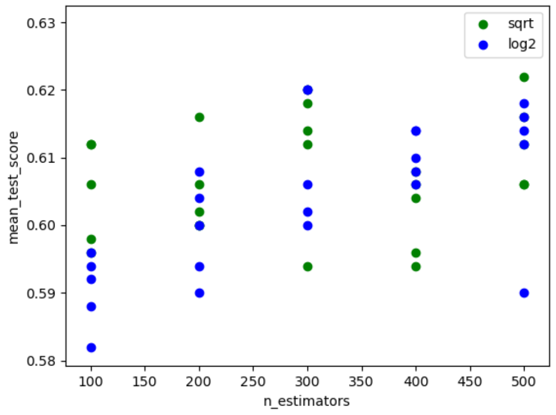


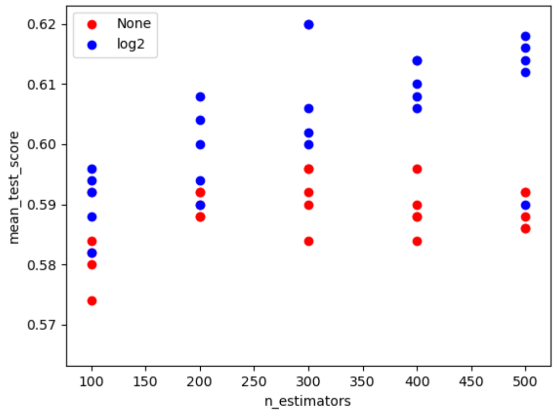
Looking at these graphs, we can see that green and blue points consistently out-perform the red points. When we compare green and blue, it seems that both are evenly distributed. Both colors tend to have higher scores around 300 max_depth.

Let us look at the second batch of graphs where the x-axis is n_estimators, the y-axis is mean_test_score, and the colors correspond to max_features:









Looking at these graphs, we see again that the green and blue points out-perform the red points. The blue and green points also seem to be evenly matched as well. The blue point with the best score has around 300 n_estimators but there seems to be an upward trend in score as n_estimators increase for blue points. The green point with the highest score has around 500 n_estimators. The distribution of green points does not have any noticeable trend.
Based on these graphs, there is no clear "best" set of hyperparameters. If we were to choose, we should choose either sqrt or log2 for the max_features, 300 for max_depth, and 300 or 500 for n_estimators. The best set of hyperparameters that GridSearchCV returned was ['max_depth': 300, 'max_features': 'sqrt', 'n_estimators': 500]. This is expected based off of our observations of the distributions of the graphs.

3.2 Tuning Non-Linear SMVs

Here is the set of hyperparameters that I tuned for Non-Linear SVMs:

- C: [1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03, 1.e+04]
- kernel: ['poly', 'rbf', 'sigmoid']
- gamma: [1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03]

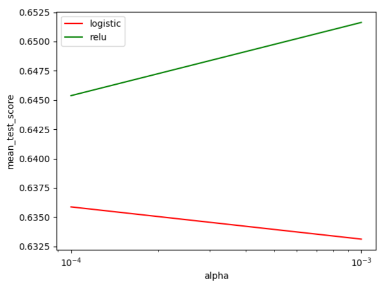
Each GridSearchCV on a split half the size of the data usually took 7 seconds to run. However, sometimes it would take significantly longer to converge. My results took **14s** to complete.

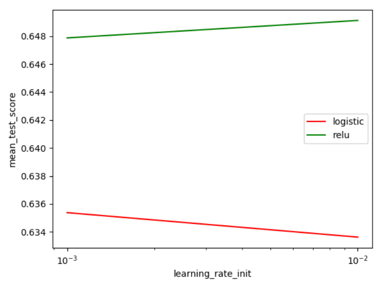
3.3 Tuning Neural Nets

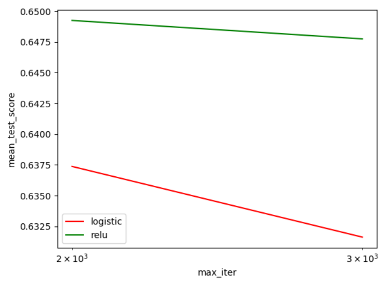
Here is the set of hyperparameters that I tuned for MLP:

- hidden_layer_sizes: [(10, 10, 10), (20, 20, 20), (10, 10), (20, 20)]
- activation: ['logistic', 'relu']
- apha: [0.0001, 0.001]
- learning_rate_init: [0.01, 0.001]
- max_iter: [500, 1000] → [2000, 3000]
  None of my MLPs would converge within 1000 iterations so I increased the max_iter

Each GridSearchCV on a split half the size of the data took 950 - 1000 seconds to run. This first round of my experiment took **2008s** to complete.
These combinations of hyperparameters took a long time to run but did not give me too much data. Here are some graphs showing the effects of each hyperparameter:







Each graph was made this way: I took the average mean_test_score of each activation function at each x-value of (alpha, learning_rate_init, max_iter). Since I only had two different values for each of these, there is a straight line connecting the two averages to show a very rough estimate of a trend. From these graphs, we can see that higher scores are associated with:

- higher alpha value
- higher learning rate for relu, lower for logistic
- lower max iterations. This is probably because a higher alpha reduces **overfitting**

Looking at the MLP_out.txt, I printed the best params of the two grid searches I did. I noticed that 3 hidden layers did better with a higher learning rate while 2 hidden layers did better with a lower learning rate. From these results, it is clear that using relu activation for the neurons performed a lot better than using logistic. This makes sense because the logistic function Sklearn's MLP uses is the sigmoid function. The sigmoid function is good for binary classification but relu is used in almost every other type of problem. It is hard to conclude anything about the learning rate and it seems we do better with lower max_iterations. So I took these results to run another experiment in which I only vary alpha and learning rate. Here are the hyperparameters:

- hidden_layer_sizes: [(100, 100, 100)]
- activation: ['relu']
- apha: [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1]
- learning_rate_init: [0.001, 0.005, 0.01, 0.05, 0.1]
- max_iter: 1500

1
2
3