Kevin McCarville
CS - 7641 - Machine Learning
1/28/2024
Assignment 1

Description of Classification Problems:

The two classification problems I selected are the MNIST handwritten digit images dataset and the GTZAN Dataset for Music Genre Classification. When looking for “interesting” datasets there were a number of things I wanted to ensure these datasets had.

- Appropriate number of features
- Appropriate number of samples
- Appropriate balance of target variable, not overly unbalanced
- Not too easy and not too difficult to solve
- Multi-class datasets
- Generically interesting in their own right

The MNIST dataset has 60,000 samples in the train set and 10,000 in the test set which is more than satisfactory. The images are 28x28 pixels that are binary 1 or 0 for 784 features. This is more than enough features and shouldn’t be any issues with the curse of dimensionality. The 10,000 test samples are split between all 10 classes (digits 0-9) evenly for 1000 each. One concern is the problem may be a little on the easy side as it's a very famous dataset which has seen a lot of approaches to it, but will be instructive to see which models do better than others. I find the dataset particularly interesting in that when you look at some of the incorrect images, and a human would not be able to tell if it were a “3” or an “8” making it a difficult question of how good a model should be able to do. It touches on the Turing Test. I also think the feature analysis will be interesting. Which pixels have the most predictive power is not obvious at all. Finally, it's an ML problem with very obvious benefits for numerous industries which always makes it more intriguing.

The GTZAN Dataset for Music genre classification is a dataset that consists of 1,000 audio tracks, each approximately 30 seconds long. I am reserving 200 for a test set leaving 800 for a train set. Genres in the dataset include blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, and rock with 100 samples each. The dataset contains 58 columns computed over multiple features that can be extracted from an audio file such as chroma features, harmony, mel-frequency cepstral coefficients (MFCCs), and spectral features among others. This should be sufficient samples and features to build models off of but may be on the low side. There are no balance concerns in terms of classes. The problem is interesting to me as similar to the MNIST images or NLP or many other ML projects, we have to convert a medium of data into numeric values to train the model, which in this case used some creative extraction techniques. I also found the problem interesting as the labels are somewhat subjective (“Is this song really reggae?”). The dataset is helpful in defining what actually is rock music (or any other genre), which is something we intrinsically know but rarely quantify.

I will analyze the model results for each of these datasets in turn, starting with the MNIST, and then compare them in the conclusion.

Preprocessing:

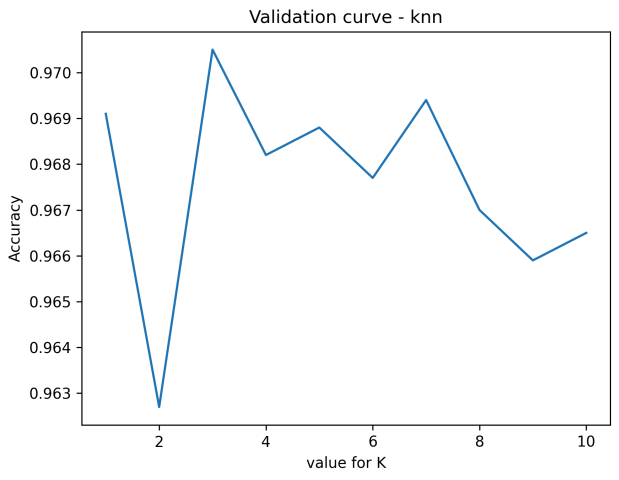
The MNIST dataset was already in binary format based on pixels thus there was minimal preprocessing necessary. For the GTZAN dataset I needed to normalize the features as the feature distributions were not on the same scale. I used a MinMax scaler for this. Other than that I dropped a couple of irrelevant columns such as “filename” and “length”. No feature engineering was done.

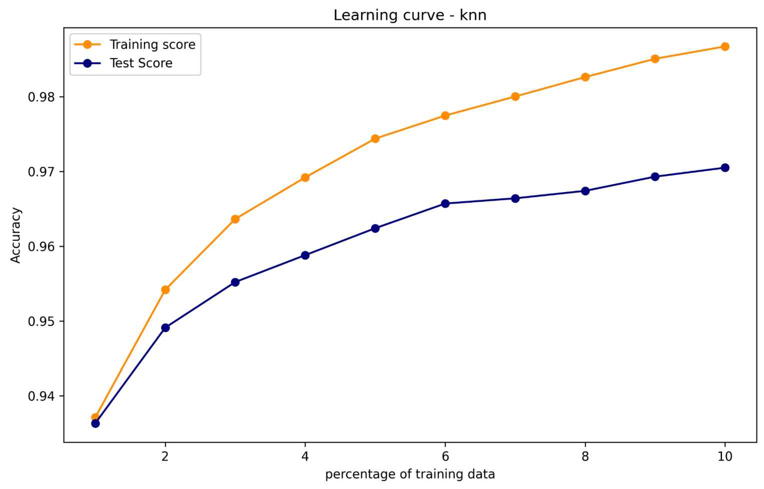
MNIST Dataset Analysis:

For each of the 5 algorithms, I will go through the validation curves, learning curves, and individual model feedback and then at the conclusion analyze the algorithms against one another.

K-Nearest Neighbor

For the KNN algorithm, I created a validation curve for the optimal value for K. The highest accuracy was with a k of 3. The accuracy varied very little amongst values but showed a clear decline in accuracy after the value of 3. The accuracy varies so little between the values (96.3 to 97.1) that there really isn't too much to read into this result. I will use 3 for k for all future learning. For the learning curve, I ran the algorithm with increments of 10% of the training data exposed to the model and monitored performance.

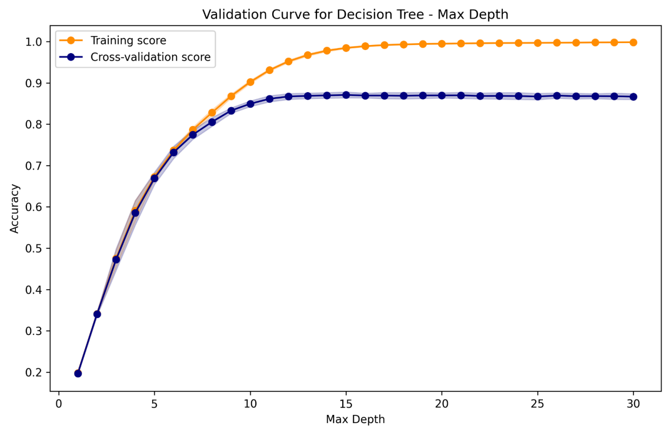


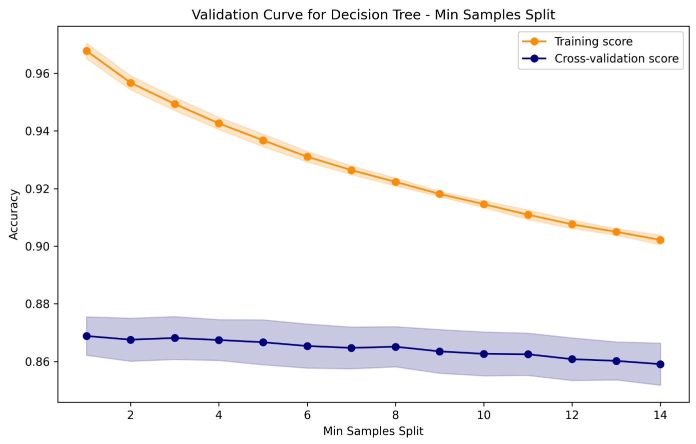


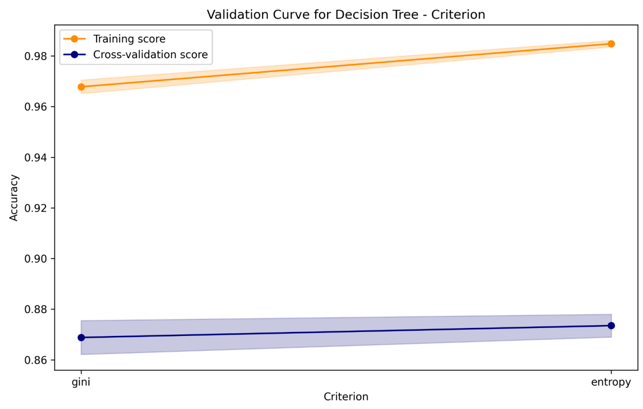
As we can see on the chart, the accuracy improved consistently as more data was shown to the model. It improved from 0.936 to 0.970. This is to be expected as the data isn't too noisy by its nature, so more data will be more likely to show the algorithm a true closer nearest neighbor. One thing of note is that the graph does not appear to level off at the end. This indicates that gathering more data should continue to improve the algorithm above that 97% accuracy level. The training accuracy overfits somewhat. The test accuracy doesn't decrease but the gap between training and test widens. With KNN there is a structural reason for this as when the model is trying to predict a training sample, it has literally seen this example before, so with a k of 3, now 1 or the 3 samples is the sample itself, which of course really boosts accuracy. The errors there will be samples that truly do look like a number they are not supposed to be. KNN overall performed well on this dataset. This is most likely due to the fact that the classes are distinctly separable in many cases. A "1" just doesn't look much like an "8". The nearest neighbors to a new sample are probably indicative of that new sample. The data also isn't too noisy. There are certainly samples that blur the lines between classes which I will analyze on future models, but it's not a big component of this dataset.

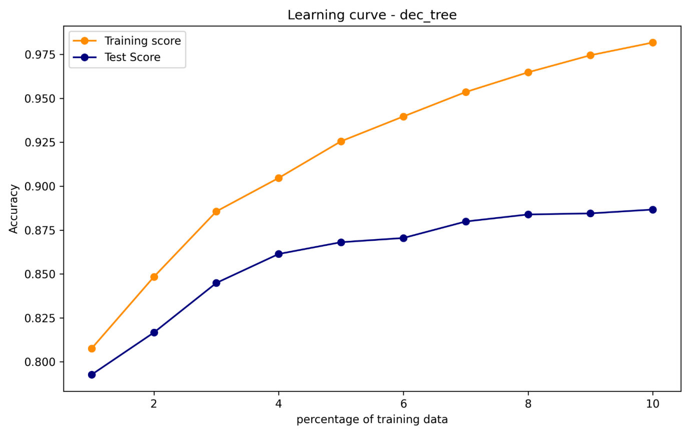
Decision Tree

I next trained a decision tree on the dataset. For the validation curves I performed hyperparameter tuning on the following variables using a 5-fold cross validation: max_depth, min_samples_to split_node, criterion. For the criterion variable I tried: entropy (information gain) and gini. The max depth was the hyperparameter with the most impact.





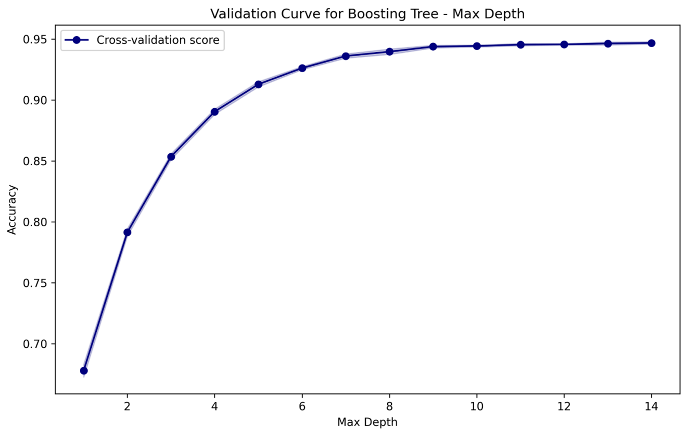


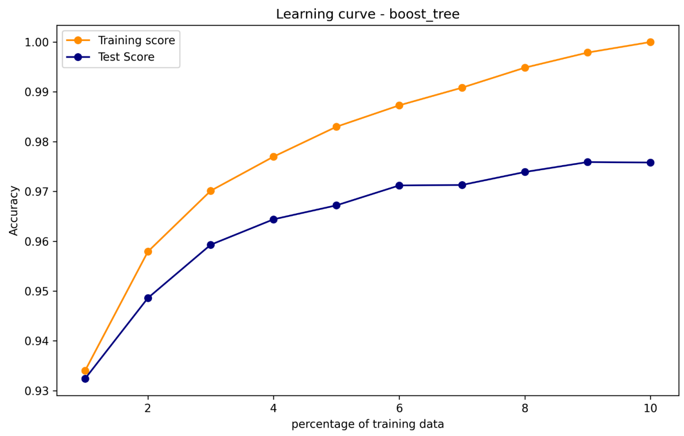


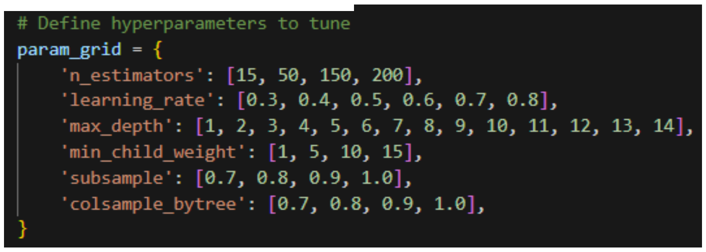
The max depth improved until a value of 13 and then leveled off. This is the value we will use for future analysis. Entropy slightly outperformed gini. Min samples to split on had minimal effect and was negative to performance. The validation curves for these metrics are presented. (Note the shaded area is the standard deviation found amongst the 5 folds. This will be the case for all future graphs as well). The first thing to note is the decision tree, similar to KNN, does pretty well with ~80% accuracy with only 10% of the data. It improves with more data but levels off around 87% test accuracy. It starts to increase overfitting at around 60% of the data, where the test-train gap begins expanding. The test accuracy here is underwhelming. Considering the nature of the data, this is to be expected as the data isn't conducive to a decision tree binary splits on features which in this case are all just singular pixels.

Boosting Tree

I next ran a boosting tree algorithm. I ran hyperparameter tuning on a handful of features using a 5-fold cross validation, which is shown below. The best results came with: n_estimators = 150, learning_rate = 0.6, max_depth = 10, min_child_weight = 5, subsample = 0.9, column sample = 1.0. The validation curve is presented for the max_depth parameter. Max Depth, similar to the decision tree, had the greatest impact on accuracy along with the number of estimators. For the learning curve, the results for the 10% increments in training data exposed are also shown below:





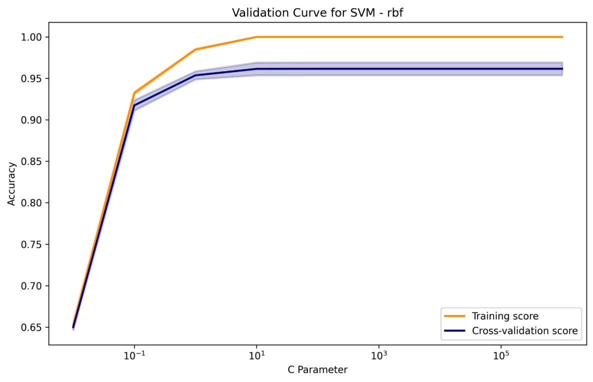


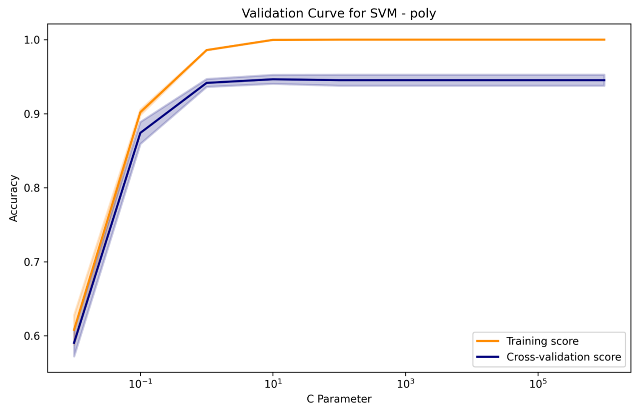
```
# Define hyperparameters to tune
param_grid = {
    'n_estimators': [15, 50, 150, 200],
    'learning_rate': [0.3, 0.4, 0.5, 0.6, 0.7, 0.8],
    'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14],
    'min_child_weight': [1, 5, 10, 15],
    'subsample': [0.7, 0.8, 0.9, 1.0],
    'colsample_bytree': [0.7, 0.8, 0.9, 1.0],
}
```

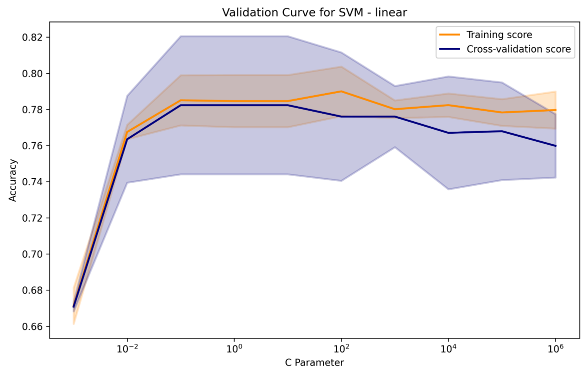
The model started at an impressive 93% accuracy and improved to over 97% where it began to level off. The training accuracy approaches 100% as some overfitting begins to set in. It’s interesting that the learning curve follows the same pattern as the decision tree for training and test accuracy but just at a higher threshold. This is also our most accurate model yet. Clearly the benefits of boosting on previous errors are improving the model’s performance.

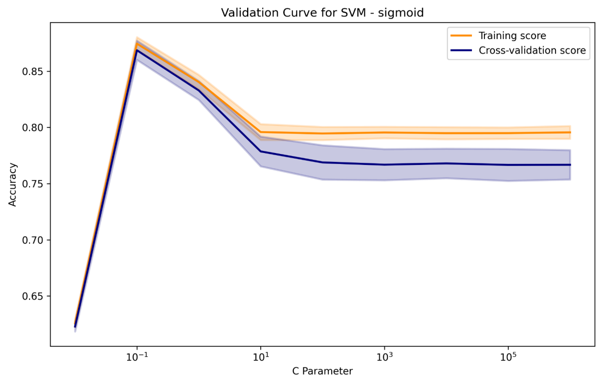
Support Vector Machine

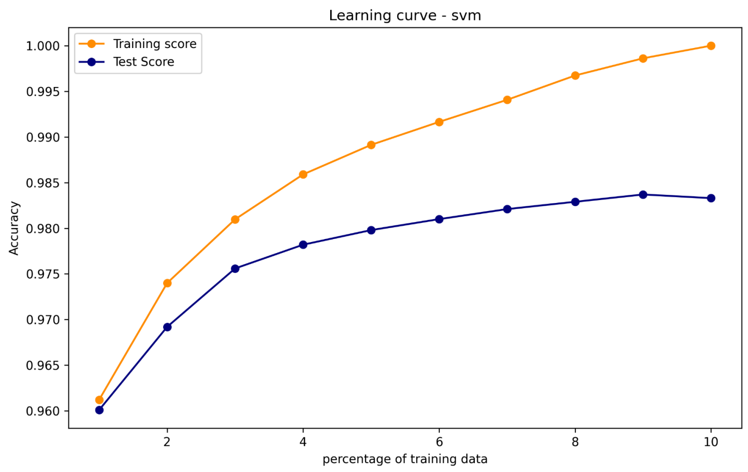
For the support vector machine model, the only hyper parameter I trained was the C (learning rate) parameter. I tried this across 4 kernel functions: RBF, poly, sigmoid and linear. RBF and C=100 were the best performers. All of the C parameters flattened out in performance around 10^2 or 10^3. The Polynomial function barely underperformed RBF. The linear function underperforming is not surprising as the data is not linearly separable. The cross validation training accuracy was close to 97% which is very good. The learning curve maintained this level of accuracy. This surprised me as I assumed the nature of the data would not lend itself to a linear solution, but this was our strongest model yet with an accuracy over 98%





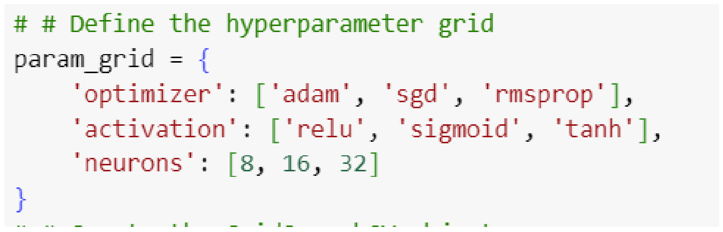




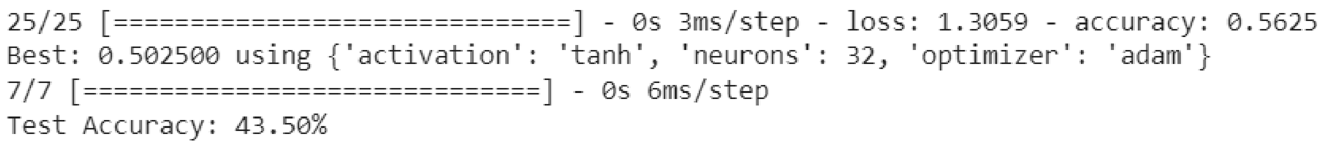


Neural Network

The final algorithm I ran was a neural network. First I ran hyper parameter tuning on the optimizer, activation function, and number of hidden neurons in a neural network with a single hidden layer. Later I expanded the different types of architectures but this was helpful to get an idea that the model does indeed perform better with more complexity. The cross validation was done with 5 folds and 20 epochs for each combination. Below is the parameter set I tried and the best results: activation = Tanh, neurons = 32, optimizer = Adam.

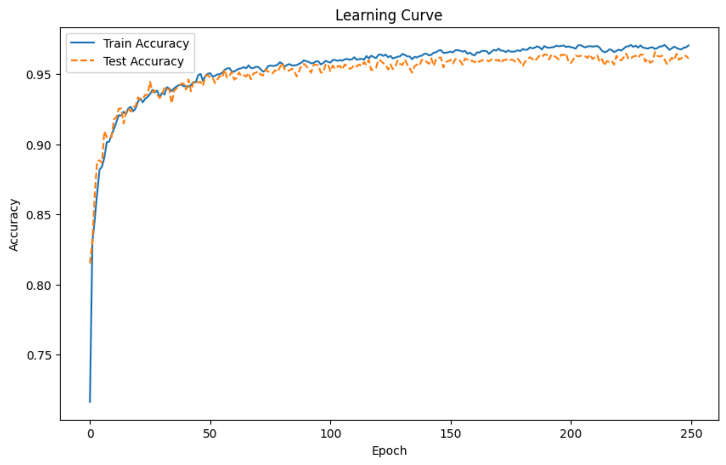


```
# # Define the hyperparameter grid
param_grid = {
    'optimizer': ['adam', 'sgd', 'rmsprop'],
    'activation': ['relu', 'sigmoid', 'tanh'],
    'neurons': [8, 16, 32]
}
```



```
25/25 [==============================] - 0s 3ms/step - loss: 1.3059 - accuracy: 0.5625
Best: 0.502500 using {'activation': 'tanh', 'neurons': 32, 'optimizer': 'adam'}
7/7 [==============================] - 0s 6ms/step
Test Accuracy: 43.50%
```

Note that the activation function used for the output layer is the softmax function as the classifier has 10 output classes and is not binary. For the neural net structure I played around with numerous architectures. Eventually I settled on a neural network with one input layer with 784 inputs for each feature and then 4 hidden layers of 128, 32, 16, 8 nodes respectively. And finally the output layer has 10 outputs, one for each class. Here is our learning curve after 250 epochs:



The model learns quickly, reaching 90% accuracy over just 10 epochs. The test accuracy continues to improve slowly but then levels off just touching 97% on par with our other models. It is interesting that there was slightly less overfitting than our other algorithms as the train accuracy paralleled the test accuracy. As for NN organization, I kept experimenting with different architectures and achieved marginal improvement. I think this will be an area to improve performance moving forward.
The neural network is the only algorithm I ran multiple epochs as the rest of the algorithms are single pass through algorithms. Their performance does not improve with an additional pass of the data. I ran multiple epochs of them in the code to prove the point, but those charts were omitted for brevity.

Conclusion

The biggest surprise in the performance of the algorithms was how well the KNN performed. It was more or less the most accurate algorithm. I was also surprised the neural network was not able to outperform, though I think that may be due to a

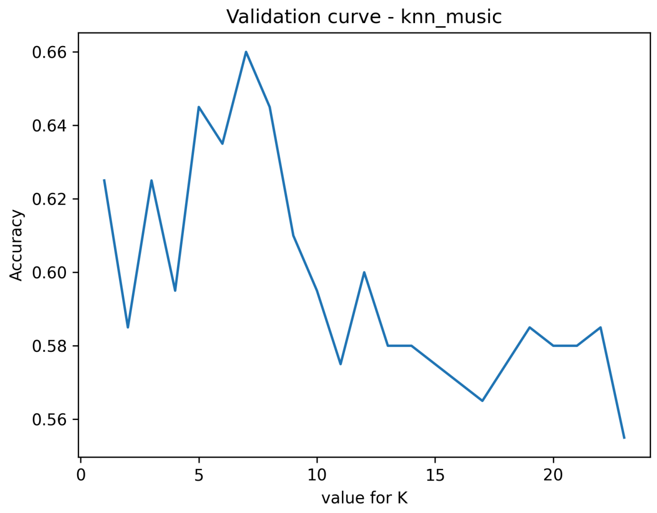
Algorithm	Test Accuracy	Train Accuracy
KNN	97.1%	98.7%
Decision Tree	98.1%	88.7%
Boosting Tree	97.6%	100.0%
SVM	98.3%	100.0%
Neural Network	96.1%	97.0%

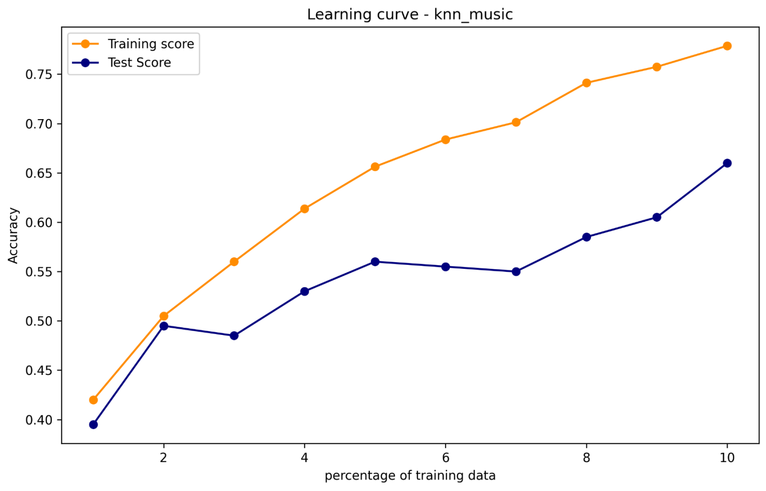
natural limit of accuracy to the dataset around 97-98%. If someone draws a 7 and says it's a 5, there isn't much the algorithm can do. Thus there will always be some error inherent in performance. The biggest changes I would make across the algorithms would be efforts to improve overfitting. Pruning of trees, ensemble methods, dropout for the neural networks could all be used. Cross validation was very helpful for parameter tuning. I think that could still be expanded to more combinations though, specifically for the neural networks. As far as run time, the SVM was by far the slowest followed by the neural network. The rest of the algorithms were not overly slow, and KNN was naturally the fastest. It's hard to put numbers on this as I often changed the amount of data or switched between CPU and GPU making the run times not comparable in terms of minutes and hours. The longest run was hyperparameter tuning for the SVM which was a few hours. The SVM performed the best. This was a little surprising as I was expecting the Boosting Tree to perform the best as they often outperform their peers, especially when the data isn't overly complex, in need of an NN model. I am here defining best based on test accuracy. Really all the algorithms performed similarly though with the exception of the decision tree which could be improved by converting that to a random forest. I do think the similarity of performance may be a symptom of the dataset as mentioned above as they all reached around a limit of 97% accuracy and thus we can't decipher which truly is better.

GTZAN Music Classification Dataset Analysis:

K-nearest Neighbor

For the GTZAN dataset I repeat the process for each algorithm that I did for the MNIST dataset, starting with the KNN. Here are the validation curve (for the value of K) and learning curve:

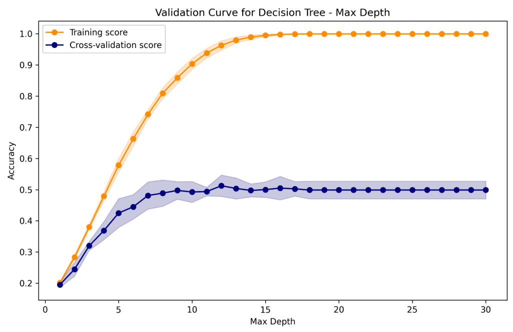


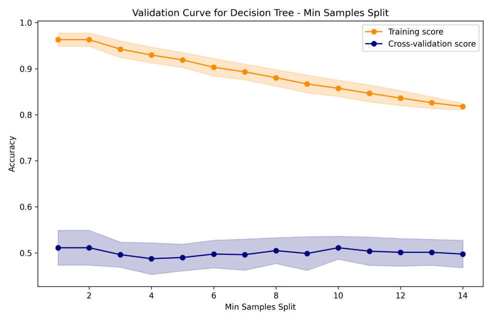


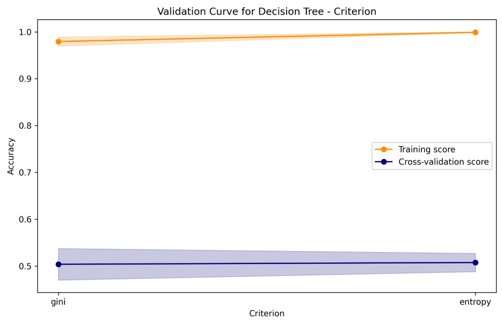
The maximum accuracy came with a K value of 7. The accuracy drops off after that. I ran with a k value equal to 7 and produced a learning curve. The learning curve achieved a final accuracy of 65%. The graph does not appear to be leveling off which indicates more data would be useful. There is not too much overfitting going on as the train-test gap widens a little from 30% of the data on, only increasing slightly, which supports our above point that the model is not fully fit to the data yet and more data would improve performance. With 10 classes, a 65% accuracy is pretty strong for this model and much stronger than a baseline of 10% for a weak learner.

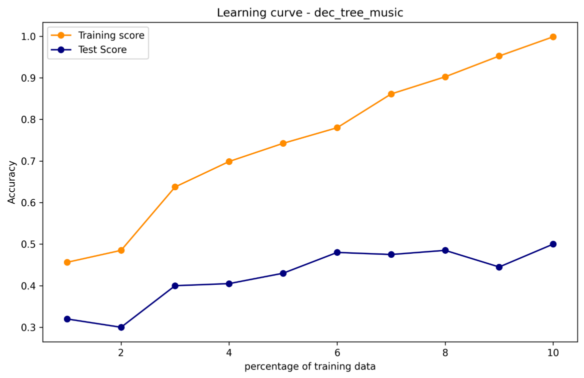
Decision Tree

The next algorithm I ran was a decision tree. For the validation curves, I tuned the same hyperparameters as before using a 5-fold cross validation: max_depth, min_samples_to split_node, criterion and the same criterion algorithms of entropy (information gain) and gini. The max depth was again, the hyperparameter with the most impact. Entropy again slightly outperformed gini and min samples to split on had minimal effect. The max depth was optimized at a value of 12. I produced the learning curve above. The decision tree performed poorly. It's accuracy barely got over the 50% threshold. Its accuracy improved with more data but its overfitting really started to take off as well. Its training accuracy approached 100% even while accuracy was down at 50%. While the test accuracy did not start to decrease, that is a massive gap between train and test.



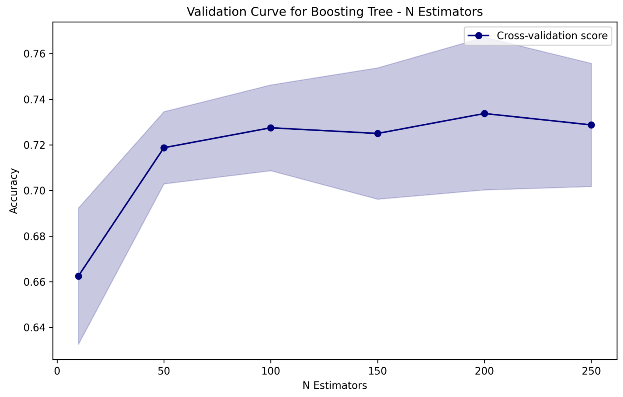


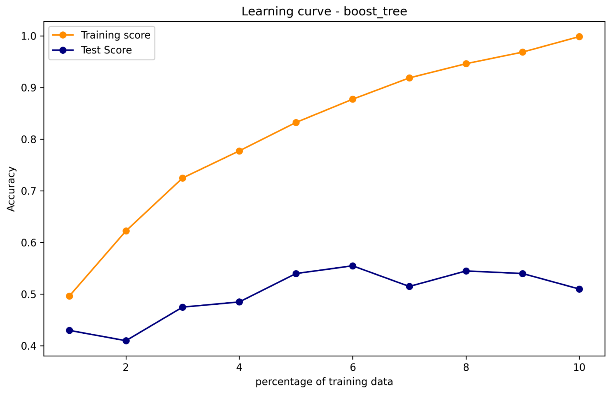


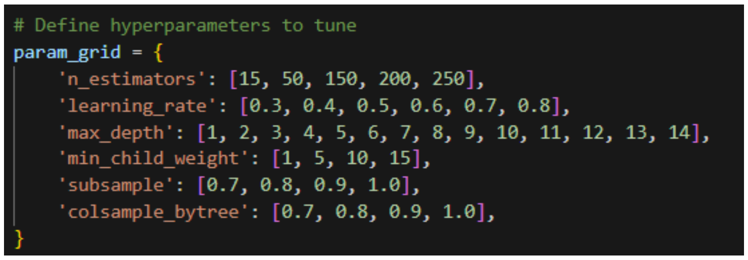


Boosting Tree

The next algorithm up was the boosting tree. I ran hyperparameter tuning on the same handful of features using a 5-fold cross validation, which again is shown below. The best results came with: n_estimators = 200, learning_rate = 0.3, max_depth = 7,







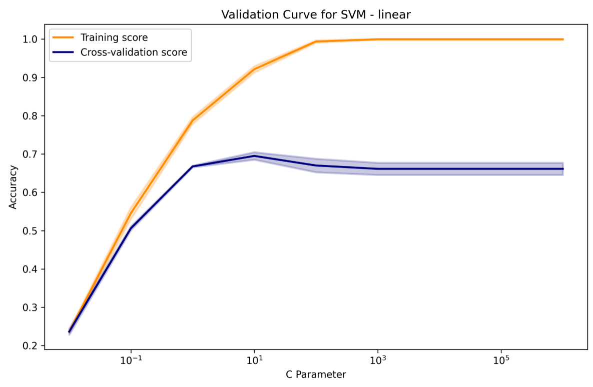
```python
# Define hyperparameters to tune
param_grid = {
    'n_estimators': [15, 50, 150, 200, 250],
    'learning_rate': [0.3, 0.4, 0.5, 0.6, 0.7, 0.8],
    'max_depth': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14],
    'min_child_weight': [1, 5, 10, 15],
    'subsample': [0.7, 0.8, 0.9, 1.0],
    'colsample_bytree': [0.7, 0.8, 0.9, 1.0],
}
```

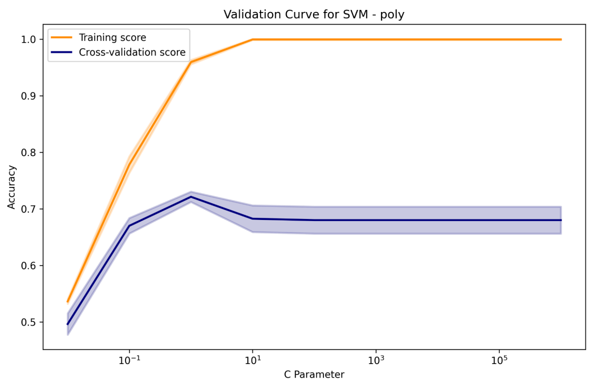
min_child_weight = 1, subsample = 0.9, column sample = 1.0. Below is the Validation curve for the n_estimators parameter, which had the best effect on improving this model's performance. For the learning curve, the results for the 10% increments in training data exposed are also shown below:

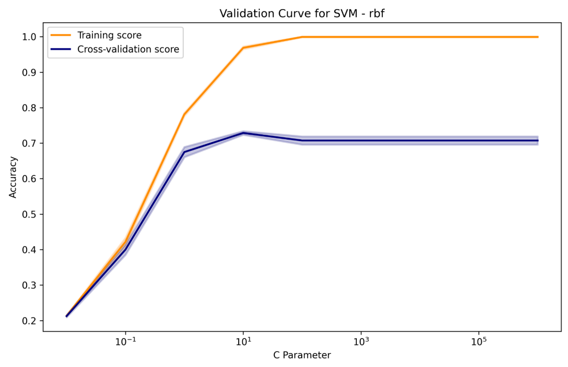
The n_estimators had a big improvement up to 100, but started to level off after that. The learning curve looks very similar to the decision tree learning curve and if anything is worse, as it begins to decline. It gets to around 50-55% accuracy at 60% of the data and levels off from there and even starts to decrease. While it is outperforming the decision tree in terms of accuracy, the test accuracy stops improving fairly early while overfitting continues to take off as training score moves towards 100%. These are pretty disappointing results for the model that was the best performer on the MNIST dataset.

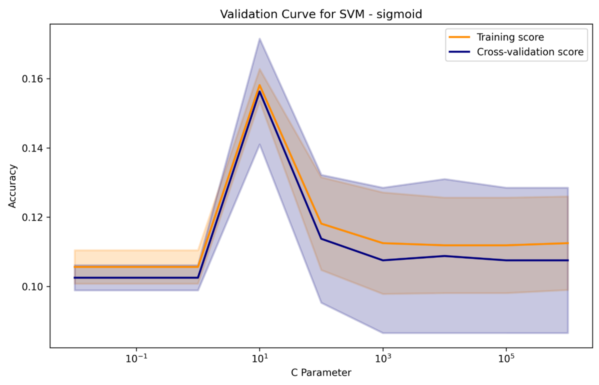
Support Vector Machine

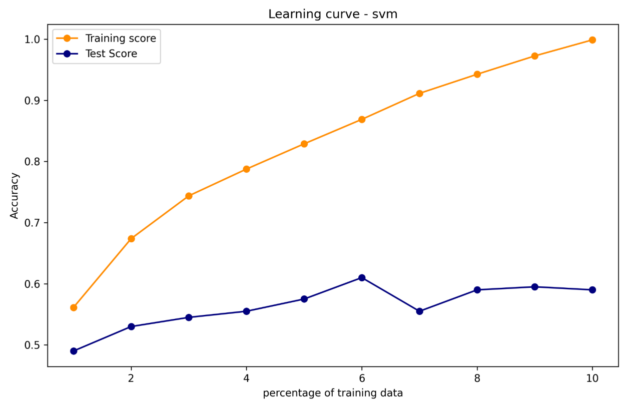
For the support vector machine model, again the only hyper parameter I tuned was the C (learning rate) parameter. I again tried this across 4 kernel functions: RBF, poly, sigmoid and linear. RBF and C=10 were the best performers. The cross validation training accuracy was close to 70% which is on the high end of what we have seen from the other algorithms. We see much more overfitting though in this step than we saw with the MNIST and the linear solution does much better relative to its peers this time. It's interesting that the three top kernel functions all have very similar performance in terms of both training and test accuracy. The learning curve, however, dipped back to 59% from 70% during the cross validation, and again we continue to see the overfitting we have seen throughout our algorithms take hold as more data is added. This is still our best model yet other than the KNN, outperforming both of our tree based algorithms.





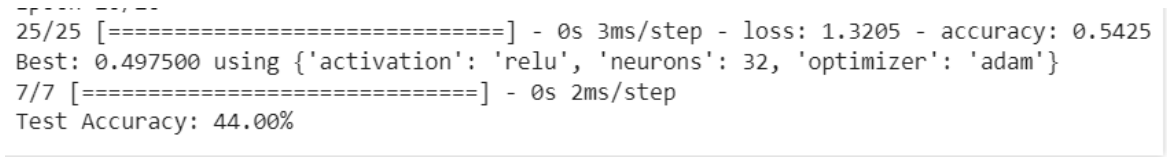


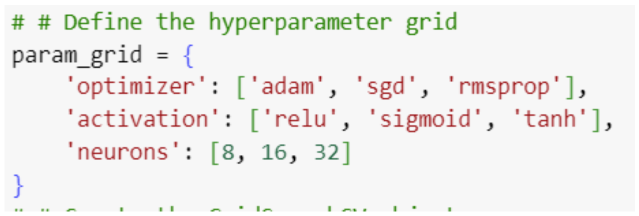


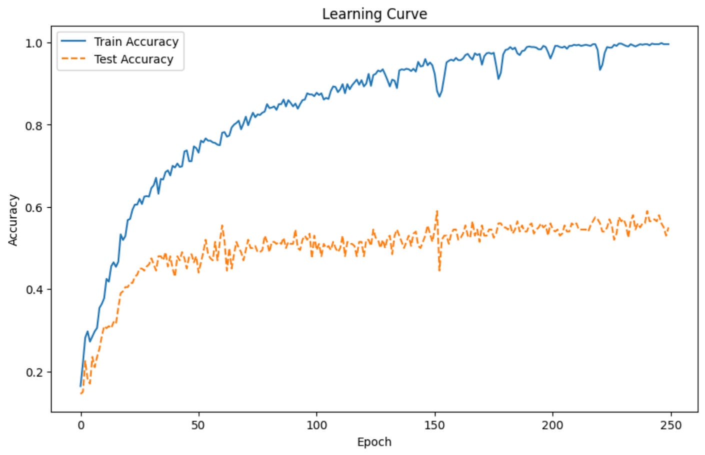


Neural Network

The final algorithm I ran was a neural network. I again ran hyper parameter tuning on the optimizer, activation function, and number of hidden neurons in a neural network with a single hidden layer. The cross validation was done with 5 folds and 20 epochs for each combination as before. Below is the parameter set I tried and the best results: activation = Relu, neurons = 32, optimizer = Adam (Again, neurons here were just informative, more structures were tried out later). The activation function used for the output layer is again the softmax function. For the NN structure, I again experimented with numerous setups and eventually I settled on a neural network with one input layer with 57 inputs for each feature and then 5 hidden layers of 128, 64, 32, 16, 8 nodes respectively. And finally the output layer has 10 outputs, one for each class. Our learning curve ran for 250 epochs. The test accuracy just touched 60%. These results were a little disappointing. The overfitting started to accelerate at around 25 epochs as the train-test accuracy gap really expanded. Training accuracy was close to 100% at about 200 epochs. The neural network outperformed all of the other algorithms except KNN but I was expecting it would have a higher test accuracy and be the best performer.


```
25/25 [==============================] - 0s 3ms/step - loss: 1.3205 - accuracy: 0.5425
Best: 0.497500 using {'activation': 'relu', 'neurons': 32, 'optimizer': 'adam'}
7/7 [==============================] - 0s 2ms/step
Test Accuracy: 44.00%
```


```
# # Define the hyperparameter grid
param_grid = {
    'optimizer': ['adam', 'sgd', 'rmsprop'],
    'activation': ['relu', 'sigmoid', 'tanh'],
    'neurons': [8, 16, 32]
}
```



Conclusion

The most noteworthy result is that our KNN outperformed all other models based on test accuracy. This is pretty surprising considering how simple the algorithm is compared to the other methods. The other algorithms produced results roughly between 50-60% while the KNN was at 65% and still improving. This may be due to it having the lowest training accuracy as it was the most resistant to overfitting. The Boosting Tree being one of the worst models was surprising as well considering it was our best performer for the MNIST. I was hopeful that the neural network would exceed the low performance of the earlier models but that did not manifest as its results were middle of the road. Again the SVM was the slowest to run and the KNN the fastest.

Algorithm	Test Accuracy	Train Accuracy
KNN	66.0%	77.8%
Decision Tree	50.0%	99.5%
Boosting Tree	51.0%	99.9%
SVM	59.0%	99.9%
Neural Network	55.5%	94.8%

The cross validation hyperparameter tuning was very slow as well for the neural network and developing the learning curve was slow as well as it continued improving through many epochs. The training times were, however, orders of magnitude quicker than for the MNIST as we had much less data in terms of rows and columns. Similar to the MNIST dataset, utilizing techniques to improve overfitting would be one of the first improvements to the models I would make. This is even more imperative with GTZAN as overfitting played a more outsized role in our results. With the Boosting Tree and the Neural Network in particular, I really think if more efforts were made to prevent overfitting and to trim down the train-test performance gap, we could work to continue to improve our test accuracy. Utilizing Dropout and ensemble techniques are the first areas I would look to explore.

Comparing Models

The two main differences in the results of the two models are that accuracy was much lower for the GTZAN than the MNIST and that its overfitting was much higher. The biggest explanation for this in my opinion is data. The MNIST has over 10x as many features and roughly 70x as many samples. Improving the quantity and quality of data in the GTZAN I think will help to close these gaps.

For future assignments, I have a plan I want to experiment with for the GTZAN. The GTZAN is broken into samples of 30 seconds each. In the future, I'd like to break each of these into 10 3-second samples thus increasing the number of samples 10-fold. There are some downsides to this as each 3-second interval may not be as representative of the label as the 30-second window as a whole but I think the benefits will far outweigh the costs here. All of the algorithms indicate that they are starved for more data. If I can increase the sample size while adding some protection against overfitting, I think there is real potential for improvement in the results. Specifically, I want to spend more time with the Boosting Trees. As we learned, Boosting Trees are particularly resistant to overfitting, which makes the results here peculiar. I am excited to see how the performance changes if I am able to feed the algorithms much more data, even if the data is slightly less rich on a per sample basis.

The other big difference in the data is that the MNIST features are all binary outcomes which should theoretically be quicker to learn versus the attributes in the GTZAN that are all numerical ranges. The GTZAN in its current state is clearly a lot noisier than the MNIST. The algorithms, with the exception of the Decision Tree, all reached test accuracies of over 96% on the MNIST which makes comparison for this study difficult as their relative performance really occurs at the margins, often differing in fractions of a percent. We saw a wider range of results across the algorithms with the GTZAN.

For the neural networks, there was a lot of tuning as the potential architectures are so vast. First with the optimization and activation functions but then the number of neurons and layers was near limitless. For the GTZAN dataset, if we are able to provide more data and control for overfitting, I think exploration here could substantially improve model performance.

Some things the models did have in common: The choice of kernel function for the SVM and criteria function for splitting for the Decision tree had minimal effect on performance in both models. They both had similar run times in terms of cardinal ranking of the algorithms. For example, they both were slowest for the SVM and then the neural network, with the KNN being the fastest.

For future assignments, I'm excited to try out these techniques to improve performance with the GTZAN. For the MNIST, I think the most interesting experiments will be around building more efficient, smaller, faster learning models and feature analysis / engineering to identify what the models are truly learning.

References:

- README file including for instructions for running the algorithms included
- https://www.kaggle.com/code/aftereffect/musicgenreclassificationfinal/notebook
- https://www.kaggle.com/datasets/andradaolteanu/gtzan-dataset-music-genre-classification/code
- All libraries used are listed in the import statements in the code. Main libraries: sklearn, keras, tensorflow, pandas, numpy, matplotlib, xgboost