Anthony Tong

Atong30@gatech.edu

Problem Definition

Dataset 1 – Shopper's Intent Classification

The first classification problem I chose to analyze was to try to predict whether a customer would complete a purchase or not using features gathered from their browser session data. I chose to tackle this problem for several reasons. From an analysis standpoint, the sample size (n=12,330) and the number of features – 10 numerical features and 8 categorical features- are sufficiently large that I will probably be able to have well fitted machine learning algorithms that will allow me to do meaningful analysis. Additional, plotting scatterplots of several of the features together shows that there is quite a bit of noise in the data. It will be interesting to see how each algorithm performs to this noise.

From a practical standpoint, I believe that this problem is interesting because an accurate model can be used in a corporate setting to enhance purchase rates. For example, if certain page views correlate with higher purchase rates, they can try to direct user to those pages more often. One thing to note with this problem is that it is slightly skewed, with 84.5% of the samples being negative. This means that the predictions from the supervised learning models will have to perform significantly better to be of any value.

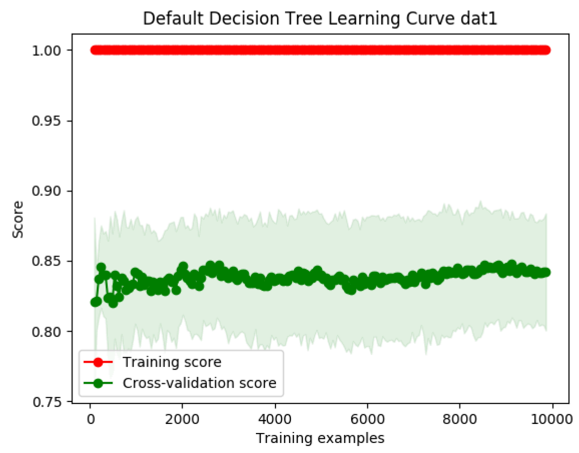
Dataset 2 – Vehicle Silhouette Classification

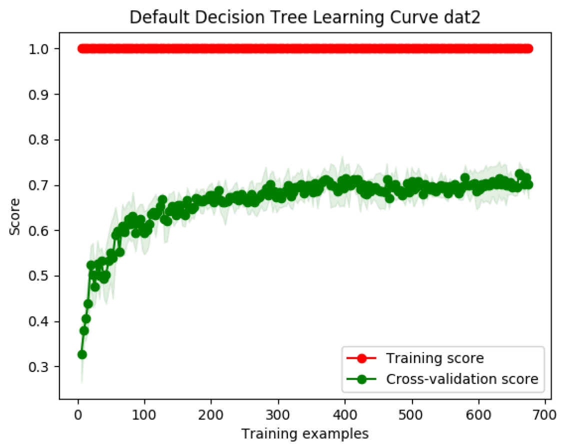
The second problem I will be analyzed will be to train machine learning models to determine what type of vehicle is in an image based on a number of features derived from its silhouette. From an implementation standpoint, I chose this dataset for several reasons. The smaller sample size (n=846) and the fact that the classification is non-binary (4 choices of car: double decker bus, Cheverolet van, Saab 9000, and Opel Manta 400) will allow me to contrast results and performance of the algorithms on dataset 1. In addition, this dataset appears to be significantly less noisy and contains only numerical features. It will be interesting to see how the performance of each machine learning algorithm will vary between the two datasets.

I also find this problem interesting because I have very little background knowledge in computer vision. Studying this dataset may open new doors for me to apply this same technique to identify different types of objects, such as types of fruits and buildings, based on their respective silhouettes in images.

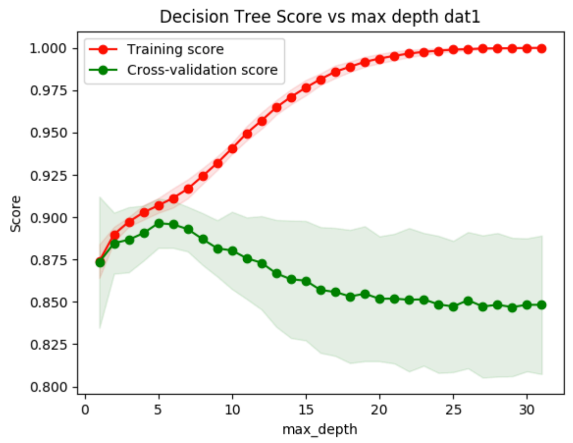
Decision Tree Analysis

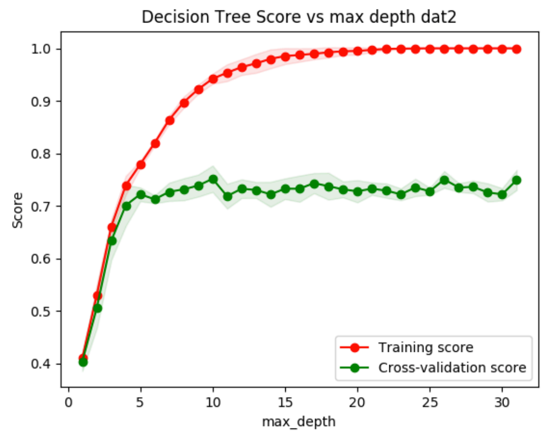
The learning plots below was generated using scikit-learn's default decision tree classifier. The large gap between the training set score and the cross-validation score as well as the high score for the training set indicates that the decision tree is severely overfitting. Also, the cross-validation score does not appear to increase with larger amounts of training examples, so increasing the size of the training set will not help. This is to be expected since the decision tree is allowed to grow without any sort of limit or pruning.





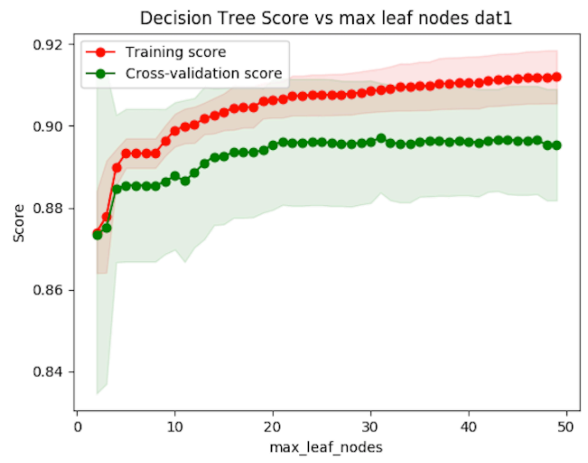
In an attempt to improve performance, I varied several hyper parameters such as splitting criterion, maximum tree depth, and maximum number of leaf nodes. Limiting maximum tree depth and the maximum number of leaf nodes are both methods of pre-pruning the decision tree to decrease complexity. Given that the tree appears to be severely overfitting, I expect these two parameters to both increase performance greatly. Based on empirical data, it looks like using entropy over Gini impurity as the splitting criterion has better performance across the board for my two data sets. Varying maximum depth for both sets of data yields the following plots.

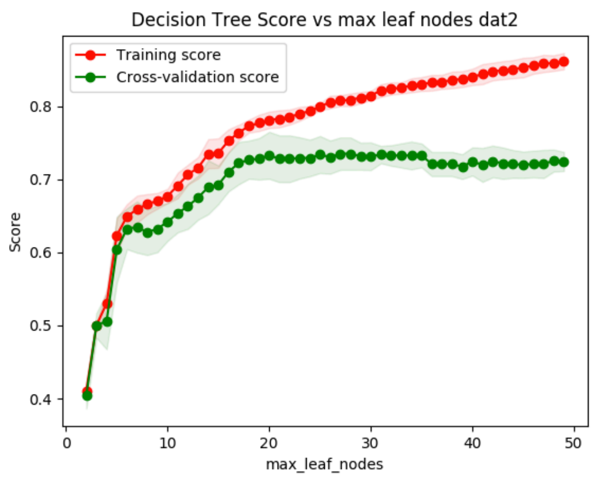




Looking at these charts, it looks like the decision tree performs relatively well on both sets of data until maximum depth exceeds 5, when it starts to over fit.

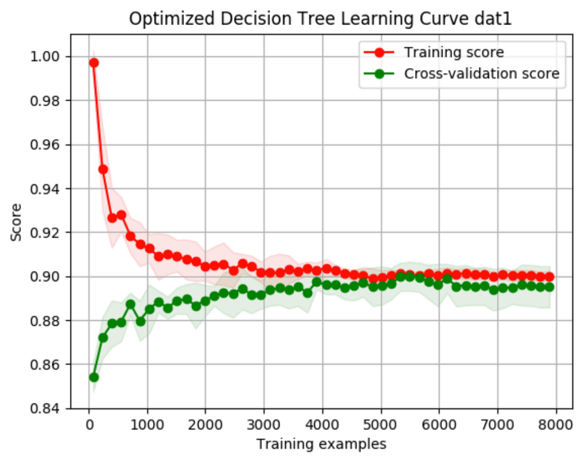
The charts below show the accuracy of the decision tree when varying the maximum number of leaf nodes. It can be seen from these charts that limiting leaf nodes is a much less aggressive form of pruning than limit maximum depth.

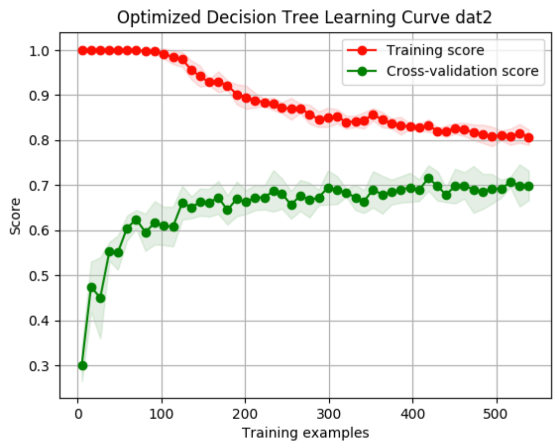




Running a grid search over both hyper parameters show that the following parameters perform the best on each data set:

Dataset	Hyperparameter Values	Train Set accuracy	CV accuracy	Test set accuracy	Training time(s)	Prediction time(s)
Dataset 1	{'criterion': 'entropy', 'max_depth': 8, 'max_leaf_nodes': 11, 'splitter': 'best'}	0.902	0.896	0.892	0.328	0.00283
Dataset 2	{'criterion': 'entropy', 'max_depth': 10, 'max_leaf_nodes': 23, 'splitter': 'best'}	0.797	0.719	0.741	0.00494	0.000376



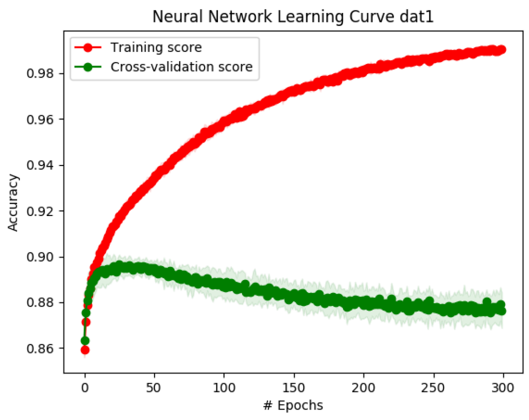


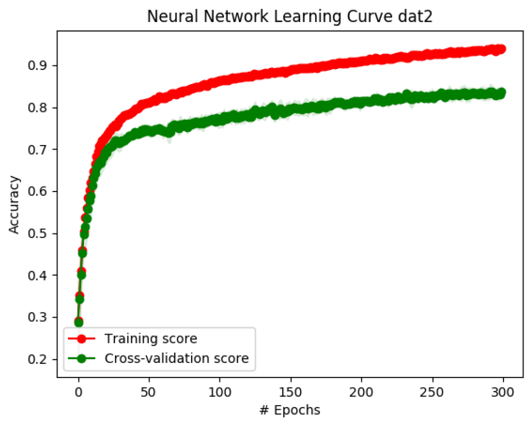
Based small difference in performance between the train, CV, and Test results, it looks like the decision tree fits dataset 1 very nicely. I suspect that the large number of sample really helped to limit overfitting once we limited the number of leaf nodes and tree depth. Dataset 2 also performed well, although there is a larger difference between training set accuracy and CV and test set accuracies. I predict that using a large number of samples would achieve even better performance for dataset 2. Regarding runtimes, the decision tree both trained and predicted very quickly on both datasets. Dataset 1 took longer as expected due to having a larger dataset.

Given more time, I would like to explore weighting classes for splits to target the imbalanced dataset 1. Also, manually pruning would also allow me to better fit my decision tree to the data, but scikit learn does not support this. In the future, I will explore other libraries to better tune my decision tree algorithms.

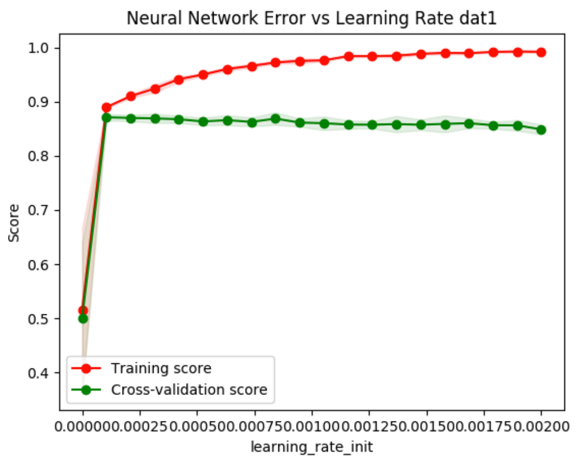
Neural Network Analysis

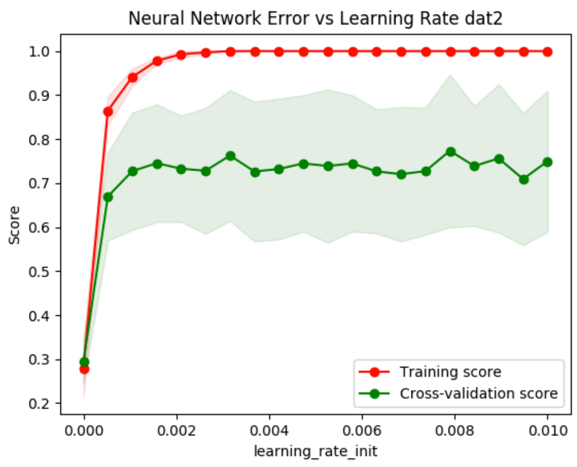
In my neural network analysis, I also chose to start by using scikit's MLPClassifier with default settings to get a general understanding of how it fits my data. Plotting the initial learning curve of the MLPClassifier versus number of epochs using 1 layer of 100 nodes yields the following two plots.



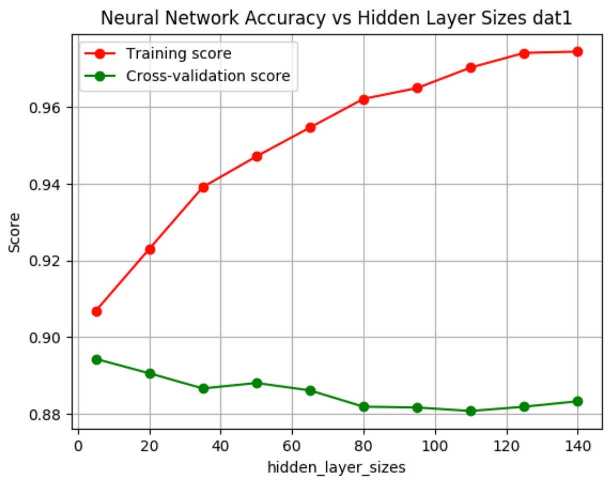


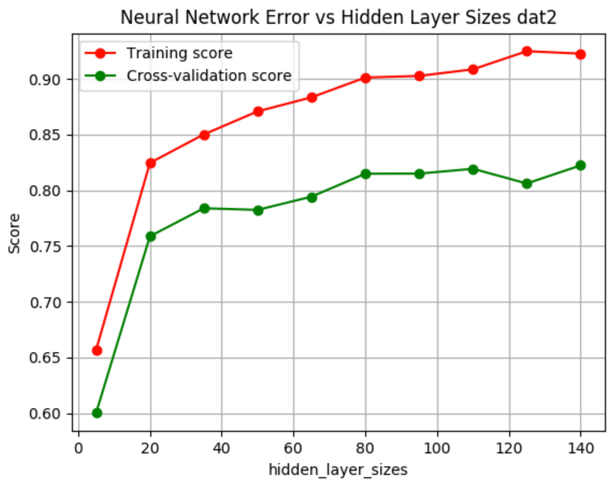
Based on the above plots, it can be seen that the neural network starts to overfit dataset 1 at roughly 25 epochs. Dataset 2 performs better on this algorithm, but still experiences high variance at higher epochs. To help reduce variance, I tried different values of learning rate. As seen in the plots below, however, changing the learning rate does not seem to impact performance significantly.





Next, I looked into varying the number of nodes and layers in the neural network. Through trial and error, I found that increasing the number of hidden layers increased the training time greatly without offering any significant increases in performance, so I chose to use only one hidden layer. Varying the number of nodes in the hidden layer yields the following plots:





Consistent with its learning curve, it can be seen that dataset1 is very sensitive to overfitting in neural networks. It looks like good performance can be observed with a hidden layer size of 5. Dataset 2, on the other hand, performs well with a hidden layer size of roughly 140. Due to time constraints, I did not explore larger hidden layer sizes, but it looks like it may perform even better using a larger hidden layer.

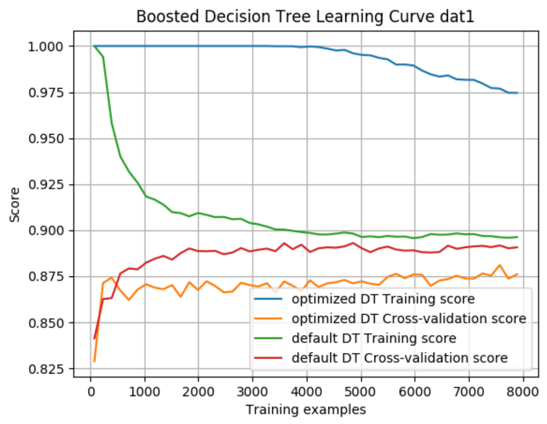
The table below summarizes the best parameters obtained from a grid search of the activation function, alpha values, hidden layer sizes, and initial learning rate. Similarly to the decision tree, dataset 1 achieves roughly 90% accuracy on all 3 train, cv, and test set accuracy. This indicates that the model generalizes very well. For dataset 2, the neural network achieves significantly better accuracy on CV and test sets than the decision tree. However, there is high variance between the training set and the test set. Regarding training times, the neural network took significantly longer than the decision tree. This is to be expected due to the large amount of calculations done during back propagation.

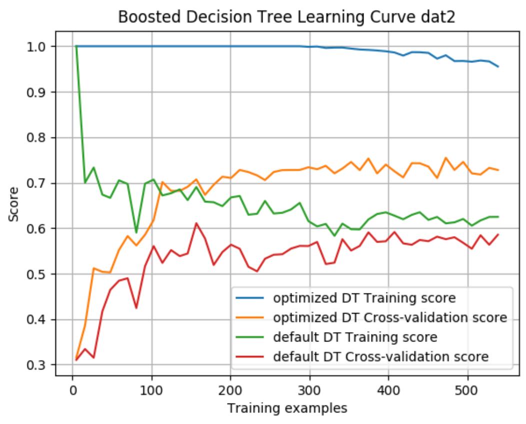
Dataset	Hyperparameter Values	Train Set accuracy	CV accuracy	Test set accuracy	Training time(s)	Prediction time(s)
Dataset 1	{'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (5,), 'learning_rate_init': 0.001}	0.902	0.896	0.909	4.512	0.00224
Dataset 2	{'activation': 'relu', 'alpha': 0.001, 'hidden_layer_sizes': (140,), 'learning_rate_init': 0.00333}	0.989	0.851	0.835	1.147	0.00197

I do not think I can improve the performance of the neural network much on dataset 1 even if given more time. However, I feel that dataset 2 can still be improved given the high variance. I would probably explore even larger hidden layers and other solvers to improve performance. Getting more data would likely also help.

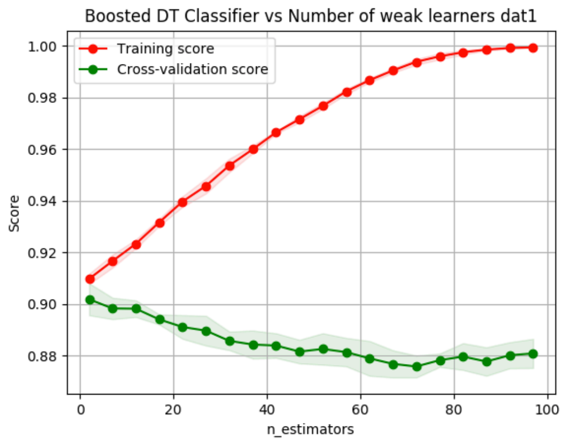
Boosting Analysis

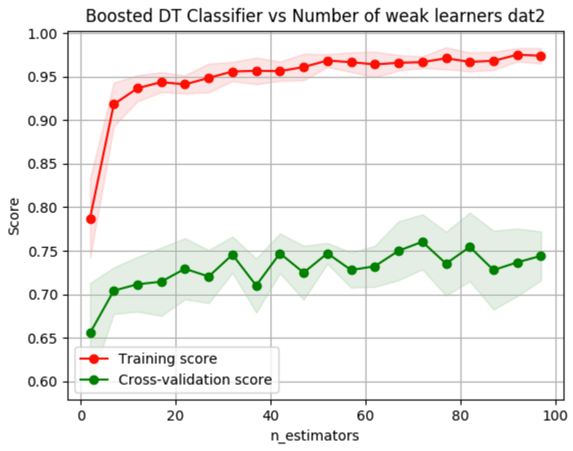
The plots below show the learning curves of the AdaBoostClassifier on both datasets. I tested out the AdaBoostClassifier with both the default base learner, a decision tree with max depth=1, and the decision trees using the optimized parameters from my previous experiments. The large variance between the training data and the cross validation data for the AdaBoostClassifier using optimized DT parameters indicate that they are overfitting. The decision trees are too complex for use within a boosting algorithm. Interestingly, the optimized DT AdaBoostClassifier still has better accuracy than the default AdaBoostClassifier despite the overfitting.



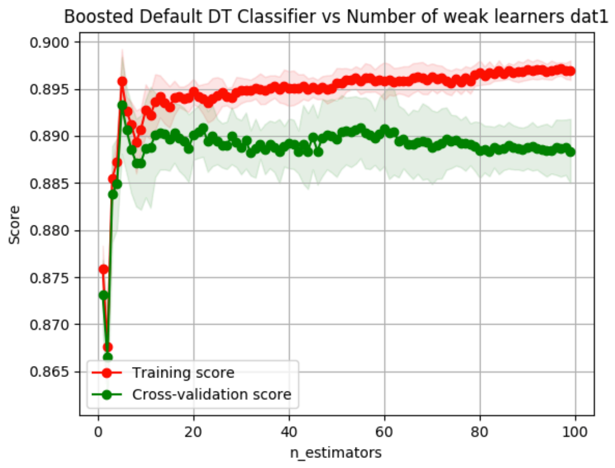


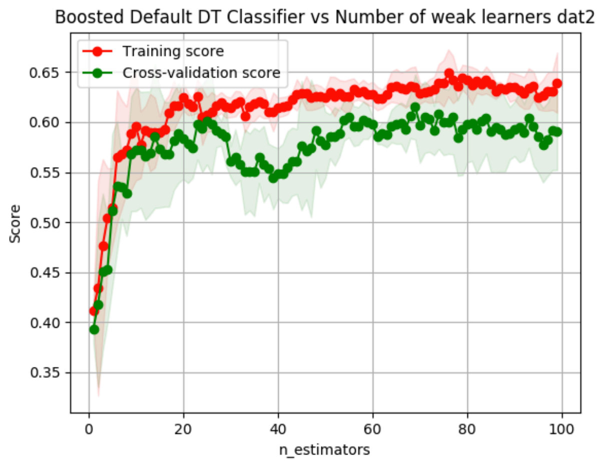
Next, I chose to vary the number of weak learners to improve performance. The two plots below show the accuracy of the AdaBoostClassifier using the optimized DT Learners as base with varying number of estimators. Varying the number of estimators does not seem to increase performance on the cross validation set. On data set 1, especially, there appears to be severe overfitting as the number of estimators increase.



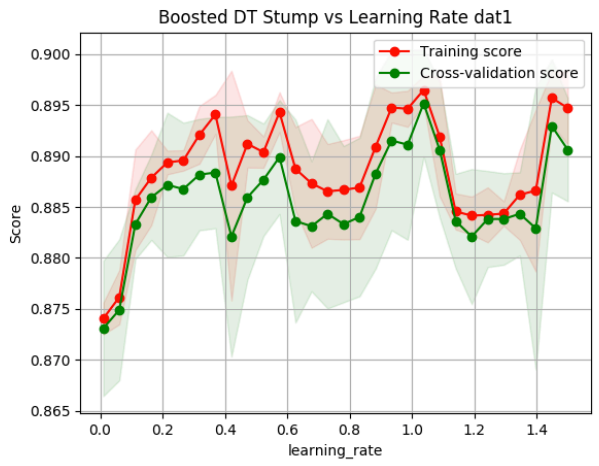


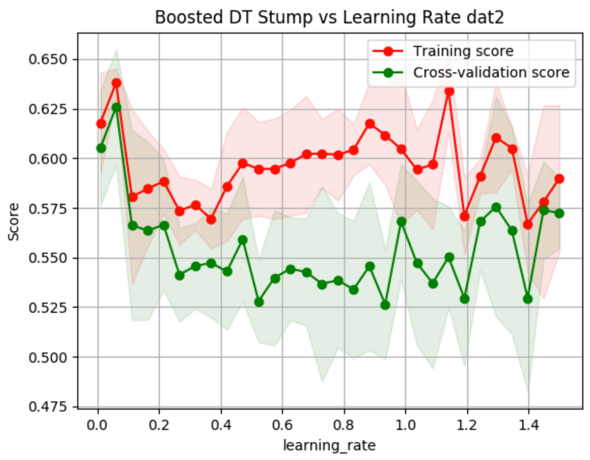
Using the decision tree stumps as the base estimator, on the other hand, yields the following two plots. The decision tree stump appears to be much less prone to overfitting.



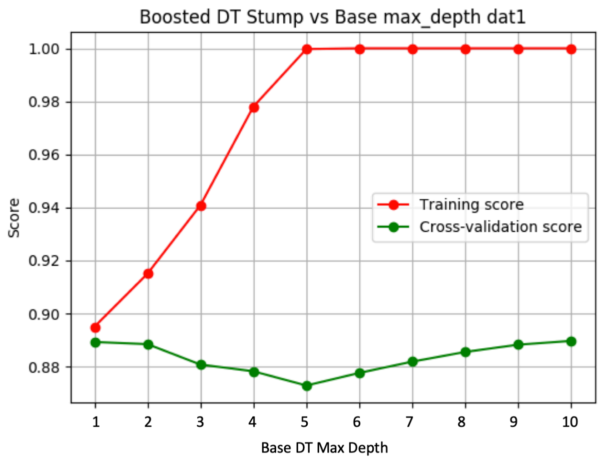


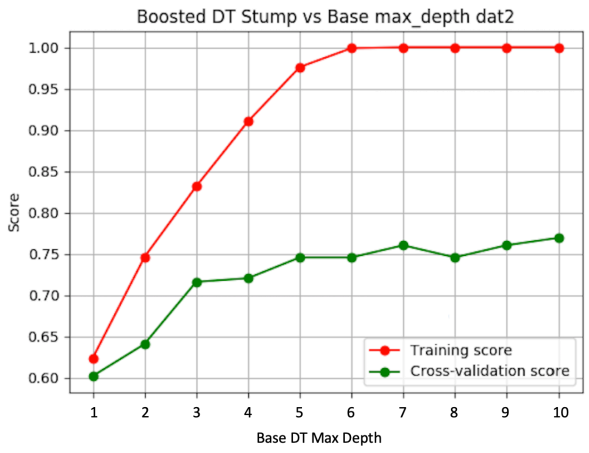
Based on the above charts, it can be seen that an n-estimator value of 5 and 23 are optimal respectively for each set of data. Next, I tried to alter the learning rate of the algorithm to further improve accuracy. It appears that controlling the learning rate can improve performance based on the plots below, although the trend is not obvious.





Compared to a regular decision tree, however, the accuracy of the Adaboost learner is significantly lower for dataset2. The Adaboost learner may be underfitting due to the base estimator being too simple. The two plots below show the performance of the AdaBoost learner using decision trees as base learners using various max depth values. Increasing the max depth of the base learner does not help dataset one, but rather causes it to overfit quickly. A slightly more complex decision tree seems to increase performance for dataset 2. Both of these results are consistent with earlier findings. Dataset 1 was already fitted nicely with a max depth of 1, so increasing the complexity caused it to overfit. Dataset 2 was underfitted, so increasing the complexity of the base estimator improved its performance.





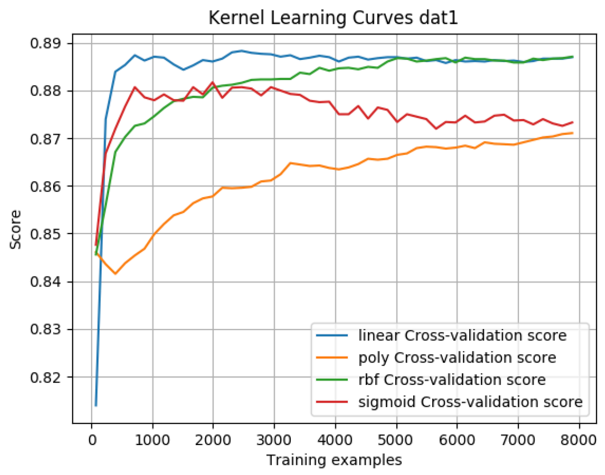
Grid search yields the following best parameters. Compared to decision tree and neural network, the adaboost learner performs slightly better on data set 1. In addition, it has low variance between the 3 sets. Similar to neural networks, however, the adaboost learner also suffers from overfitting on data set 2. Training time for this learner is longer than a typical decision tree because many decision trees must be generated. Also, the prediction time is even longer than the neural network because each of the decision tree must be traversed to generate a result.

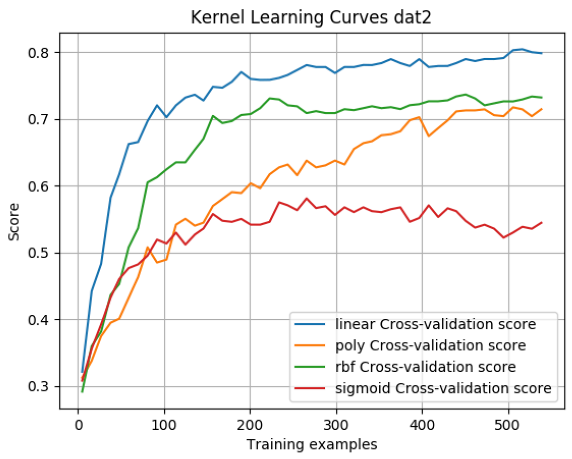
Dataset	Hyperparameter Values	Train Set accuracy	CV accuracy	Test set accuracy	Training time(s)	Prediction time(s)
Dataset 1	{'base_estimator_max_depth': 4, 'learning_rate': 0.01, 'n_estimators': 86}	0.911	0.904	0.913	3.085	0.213
Dataset 2	{'base_estimator_max_depth': 6, 'learning_rate': 0.837, 'n_estimators': 81}	1.0	0.838	0.758	0.389	0.0251

Given more time, I was likely, again, focus on trying to improve results for dataset 2. Due to limited computing resources, the grid search was only done with a 3-fold cross validation. Performing a grid search with 5-fold or 10-fold cross validation would likely eliminate much of the overfitting seen here.

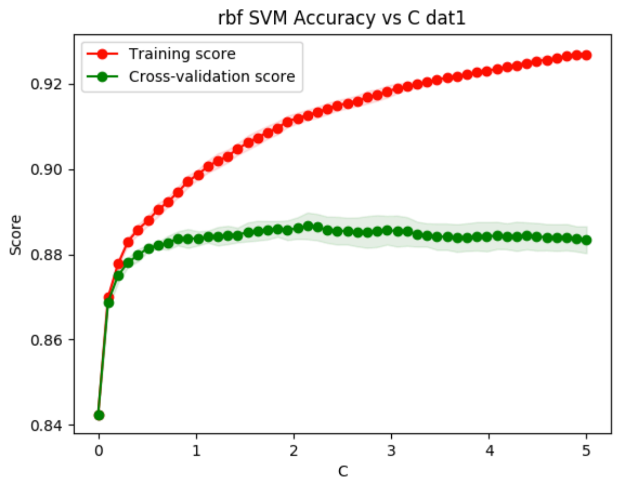
Support Vector Machine Analysis

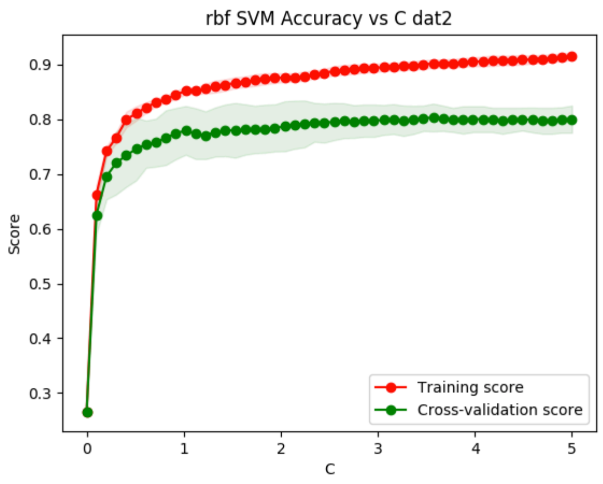
For my SVM analysis, I chose to use scikit-learn SVM library's SVC to analyze my data. The two most important parameters for SVM's are the kernel function and the penalty parameter C. Since I have no domain knowledge of either online shopper's intents or computer vision, I will be choosing the kernel function using experimental data. The following plots were generated using scikit-learn's supported kernel functions: linear, polynomial, radial basis function(rbf), and sigmoid.





For both datasets, the linear kernel function appears to perform best, with rbf a close second. This is somewhat surprising for the shopper's intent dataset because I did not see any indication of linear separability while constructing scatterplots using various features. I selected 'linear' and 'rbf' kernel functions to explore further. To further optimize the SVM learners, I varied the penalty parameter C. For the linear SVM learners, varying the values of C did not affect performance. This is likely because linear kernel functions do not allow for very complex models to develop. Varying C for the rbf svm learner yields the following two plots. Based on these two plots, it looks like higher values of C causes high variance. A value of $C < 1$ appears to be a good choice for both datasets.



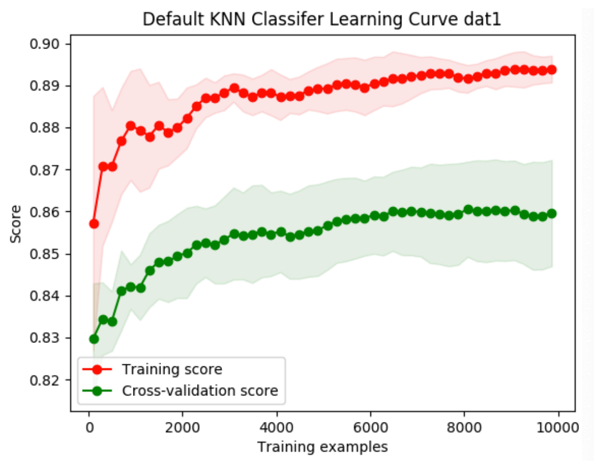


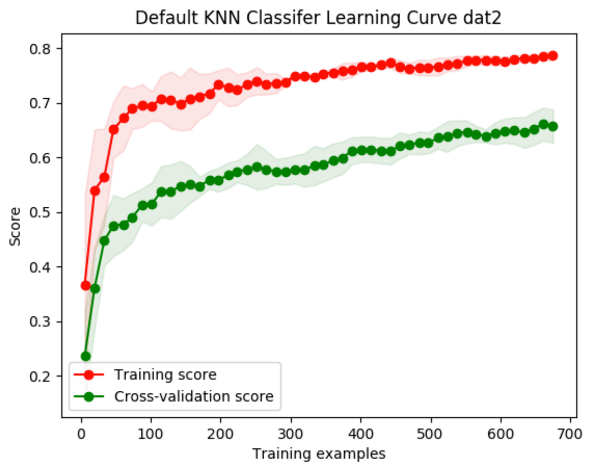
Performing grid search for both kernel functions yield the following results. Despite trying to tune the rbf kernel function for better performance, the linear kernel achieves better accuracy and lower variance. It looks like both SVM classifiers using the rbf kernel are overfitting. If given more time, I predict that varying the kernel efficient, gamma, and also limiting the max iteration would help lower the amount of overfitting. Relative to other learners, the SVM performs above average on dataset 2. However, it is one of the worst performers on dataset 1. Also, the training and prediction times are both very long for dataset 1 because it needs to calculate distances for the large number of samples.

Dataset	Hyperparameter Values	Train Set accuracy	CV accuracy	Test set accuracy	Training time(s)	Prediction time(s)
Dataset 1 – rbf kernel	{'C': 1.0527105263157894, 'gamma': 0.200001, 'kernel': 'rbf'}	0.944	0.851	0.850	9.906	4.849
Dataset 1 – linear kernel	{'C': 0.789557894736842, 'kernel': 'linear'}	0.886	0.887	0.883	12.59	1.045
Dataset 2 – rbf kernel	{'C': 2.105321052631579, 'gamma': 0.200001, 'kernel': 'rbf'}	0.945	0.754	0.714	0.0187	0.0106
Dataset 2 – linear kernel	{'C': 2.631626315789474, 'kernel': 'linear'}	0.838	0.800	0.8	0.0189	0.00412

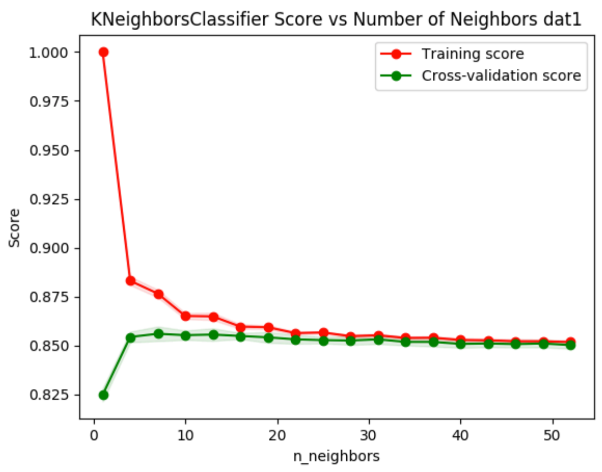
K-Nearest Neighbors Analysis

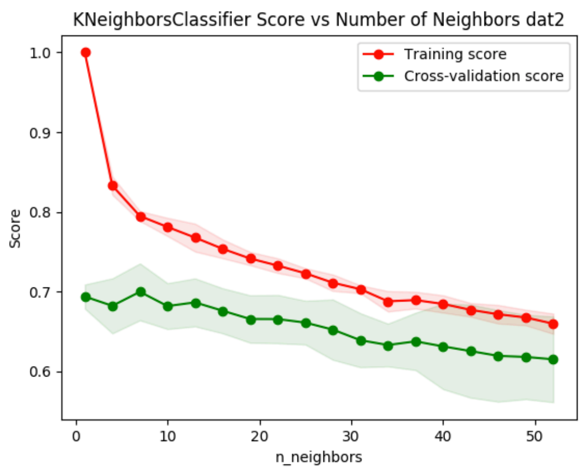
For my K-nearest neighbors analysis, I used scikit-learn’s KNeighborsClassifier learner. Plotting the learning curves using the KNeighborsClassifier with default settings (k=5) yields the following two plots. It appears to perform poorly both in terms of overall accuracy and having high variance. This indicates that the KNN learner is probably underfitting.



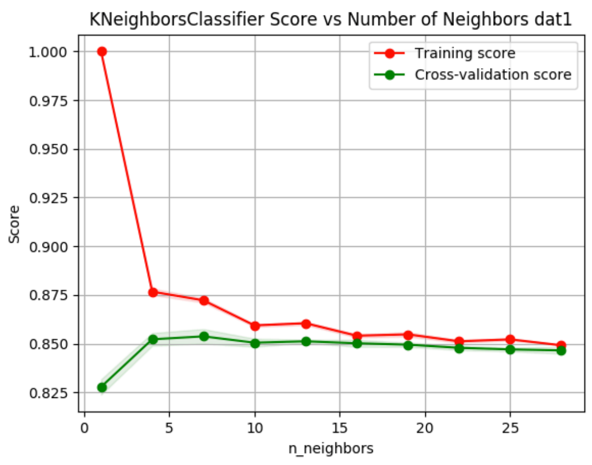


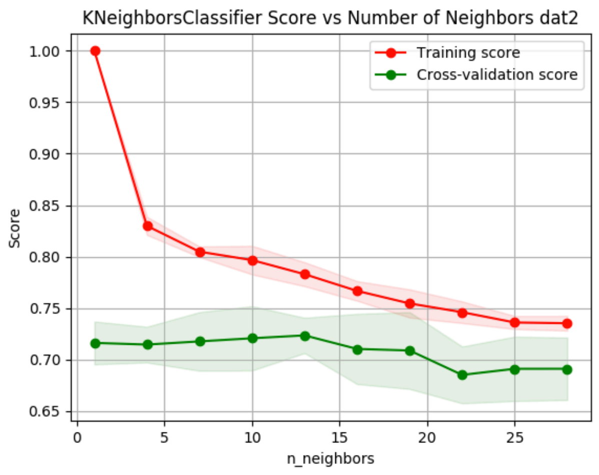
To improve performance, I looked into varying the number of neighbors used to classify samples. Unfortunately, it looks like increasing the number of neighbors actually decreases performance in dataset 2 and does nothing for dataset 1.





I also performed the same complexity analysis using Manhattan distance for the following two plots rather than the Euclidean distance used for the knn learner above. There is almost no difference in performance dataset 1, but there is ~5% increase in accuracy for dataset 2. Manhattan distance tend to perform better with higher number of features, so this may explain the performance increase.





Performing grid search on n_neighbors, p (1 = Manhattan distance, 2 =Euclidean distance), and weights yield the following results. This learner performed especially poorly on dataset 1. Given that 84.7% of the population is negative, this accuracy means that it only does 1.1% better than predicting all of the samples to be negative. I suspect that knn does poorly with this dataset due to the large number of categorical features that creates a lot of noise in distance calculation. This dataset does decently on dataset 2 using Manhattan distance. In order to improve results, I would explore the various algorithms used to calculate distance. Also, I would consider further preprocessing the feature data of dataset 1 to eliminate noisy features and creating a more balanced dataset.

Dataset	Hyperparameter Values	Train Set accuracy	CV accuracy	Test set accuracy	Training time(s)	Prediction time(s)
Dataset 1	{'n_neighbors': 10, 'p': 2, 'weights': 'uniform'}	0.867	0.866	0.856	0.274	11.1
Dataset 2	{'n_neighbors': 7, 'p': 1, 'weights': 'uniform'}	0.786	0.724	0.717	0.00146	0.0473

Works Cited:

Dataset obtained from:

Sakar, C.O., Polat, S.O., Katircioglu, M. et al. Neural Comput & Applic (2018). https://doi.org/10.1007/s00521-018-3523-0

Turing Institute Research Memorandum TIRM-87-018 "Vehicle Recognition Using Rule Based Methods" by Siebert,JP (March 1987)

Algorithms from:

Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.