**CS 7641 Project 1: Supervised Learning**

Yaling Wu (ywu342)

# Datasets

1. Car Evaluation Data: <https://archive.ics.uci.edu/ml/datasets/Car+Evaluation>

This data was derived from a hierarchical decision model. It was used for the evaluation of Hierarchical Induction Tool, which was proven to be able to reconstruct the original hierarchical model. So it is particularly useful for testing constructive induction and structure discovery methods. It includes attributes like buying, maintenance prices, doors, persons, size of luggage boot, and safety. The output class indicates acceptability of cars: unacc (70%), acc (22%), good (4%), vgood (3.8%). There are 1728 instances in total.

1. Tic-Tac-Toe Endgame Data: <https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame>

This dataset encodes the complete set of possible board configurations at the end of tic-tac-toe games, assuming “x” goes first. The target concept is “win for x” or positive. There are 958 instances and 9 attributes, each corresponding to one tic-tac-toe square. About 65.3% of the examples are positive.

To prepare for data processing, I randomly selected 70% of car data for training and 80% of Tic-Tac-Toe data for its training, since car data is larger than the latter.

The two datasets are interesting for this assignment because they are of completely different concepts and structures. They have given drastically different results from my experiments, as shown in details below.

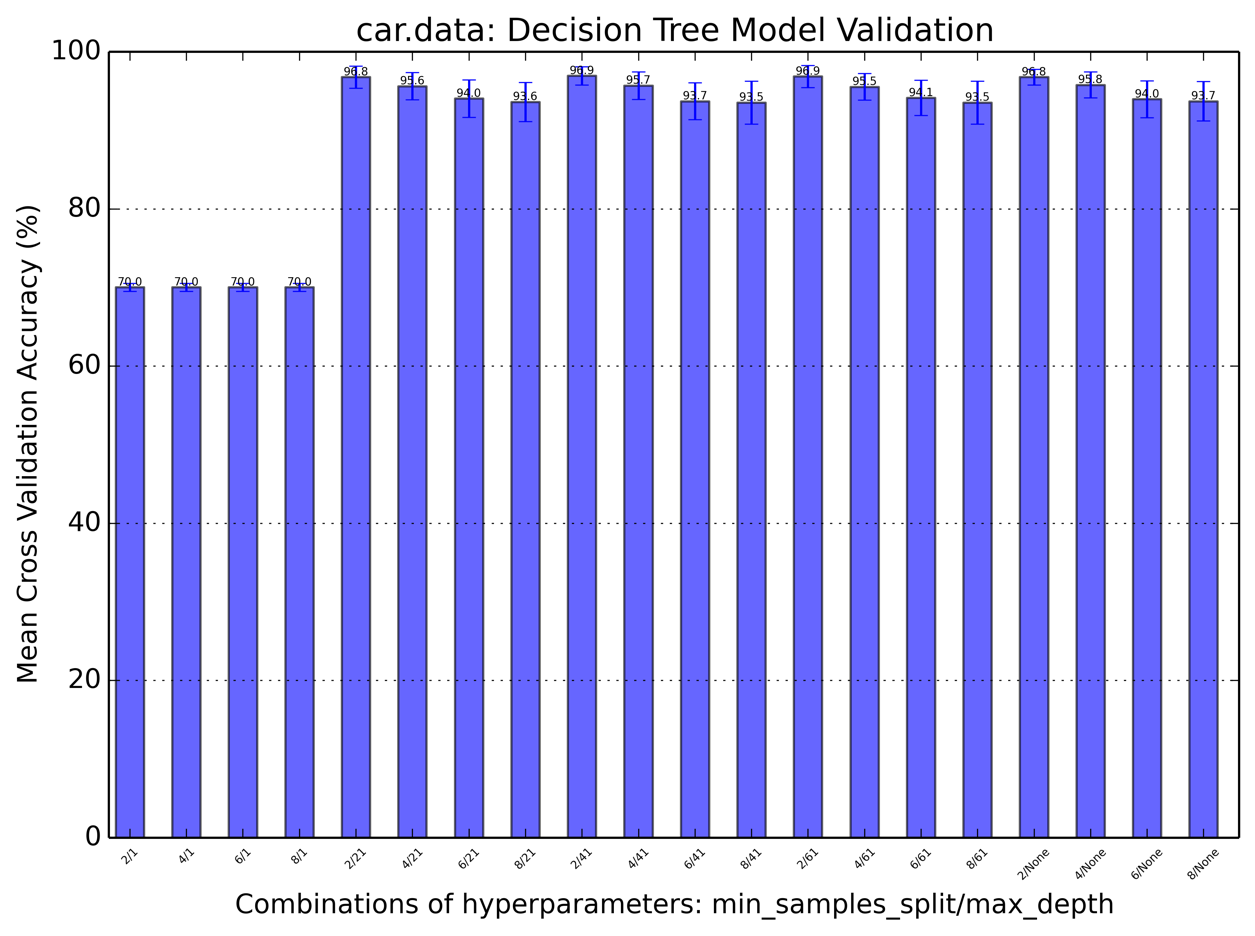
# Model Validations

The purpose of this set of experiments was to tune out the best hyperparameters for each estimator model such that I could construct a relatively good model to do the later comparisons. Two trial hyperparameters were selected for each model. Other parameters were unchanged as default values. The GridSearchCV object in python was used to perform the job of utilizing 10-folds cross validation to exhaustively search over specified parameter values. It used mean accuracy as the scoring metric to determine which set of parameter values is the best for each estimator. The corresponding objects used for estimators were DecisionTreeClassifier, SVC, AdaBoostClassifier, KNeighborsClassifier and MLPClassifier.

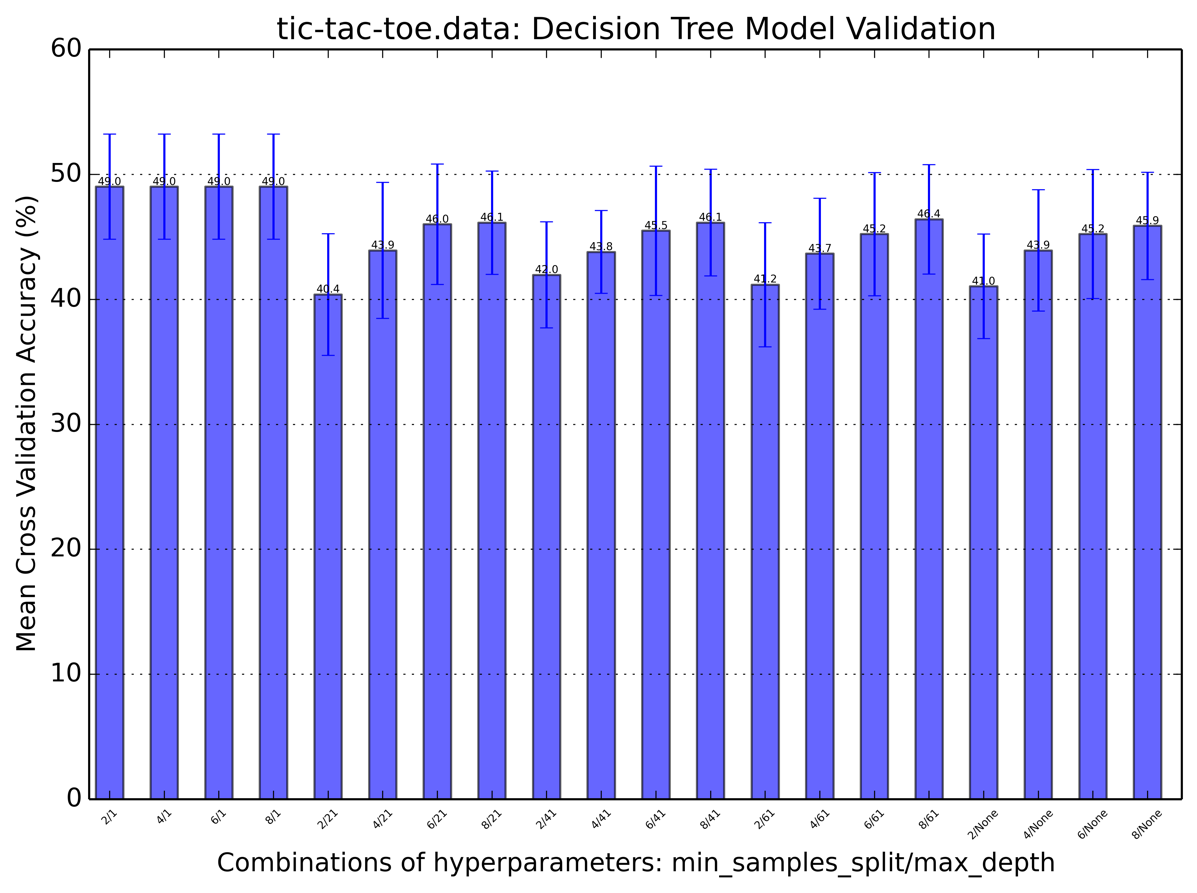
I also plotted out learning curves on the selected best parameters to find out how much we could benefit from adding more training data and whether the estimator suffers more from a variance error or a bias error.

## Decision Tree

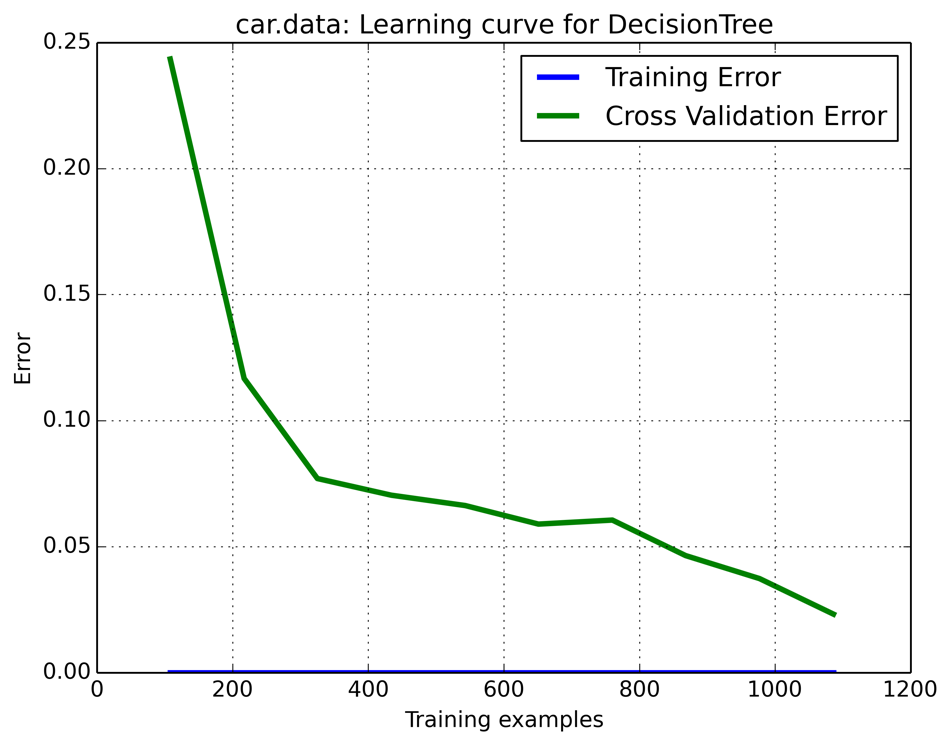
The two parameters of interest for DT were the minimum number of samples to split an internal node (min\_samples\_split) and the maximum depth of the tree (max\_depth). They are both pre-pruning factors for the tree to control the complexity of DT. The deeper a tree gets, the more closely it fits to the training data. The smaller min\_samples\_split is, the more likely the model approaches overfitting.



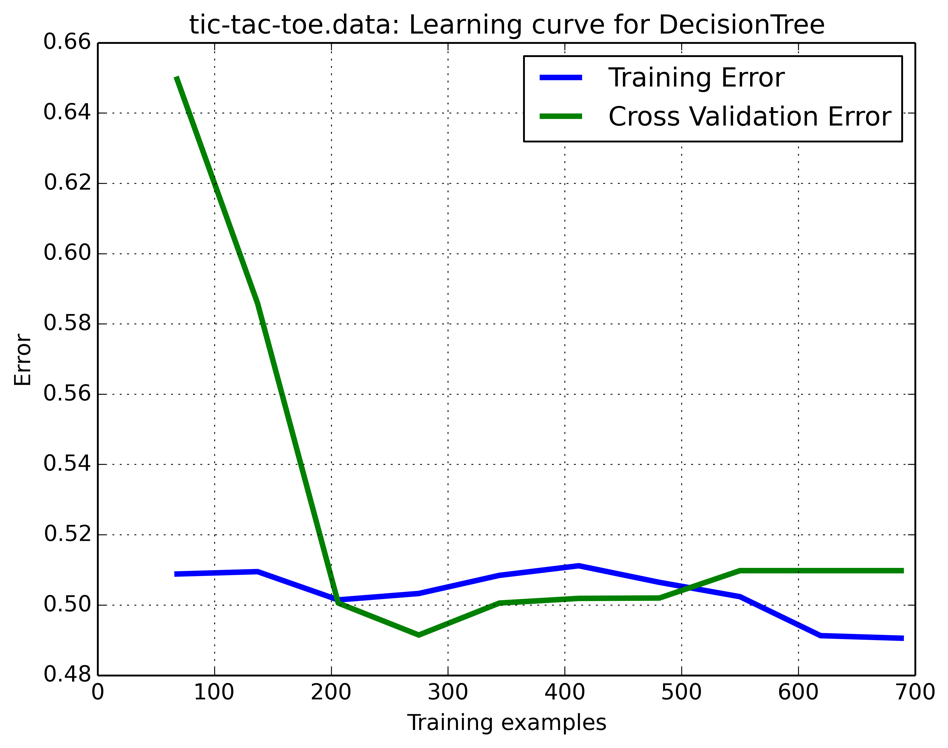
For the car data, GridSearchCV shows us DT tends to grow bigger in this range to reach a higher performance level. When the max\_depth was set to 1, the tree was constrained to a small size. This led to a highly biased model, leading to a significantly lower accuracy rate. Once the tree depth got opened up to over 21, the model reached to it higher potential. And smaller leaf node capacities gave better models. Overfitting did not seem to happen here was probably due to insufficient data to build a very complex tree. The best parameter pair here is {min\_samples\_split:2, max\_depth:41}.



All of the combinations in this experiment led to a bad score in accuracy. All of them went below 50% of accuracy. DT is generally a bad algorithm for this second dataset. It seems the tree tends to go as simple as possible. The best parameter pair here is {min\_samples\_split:2, max\_depth:1}.



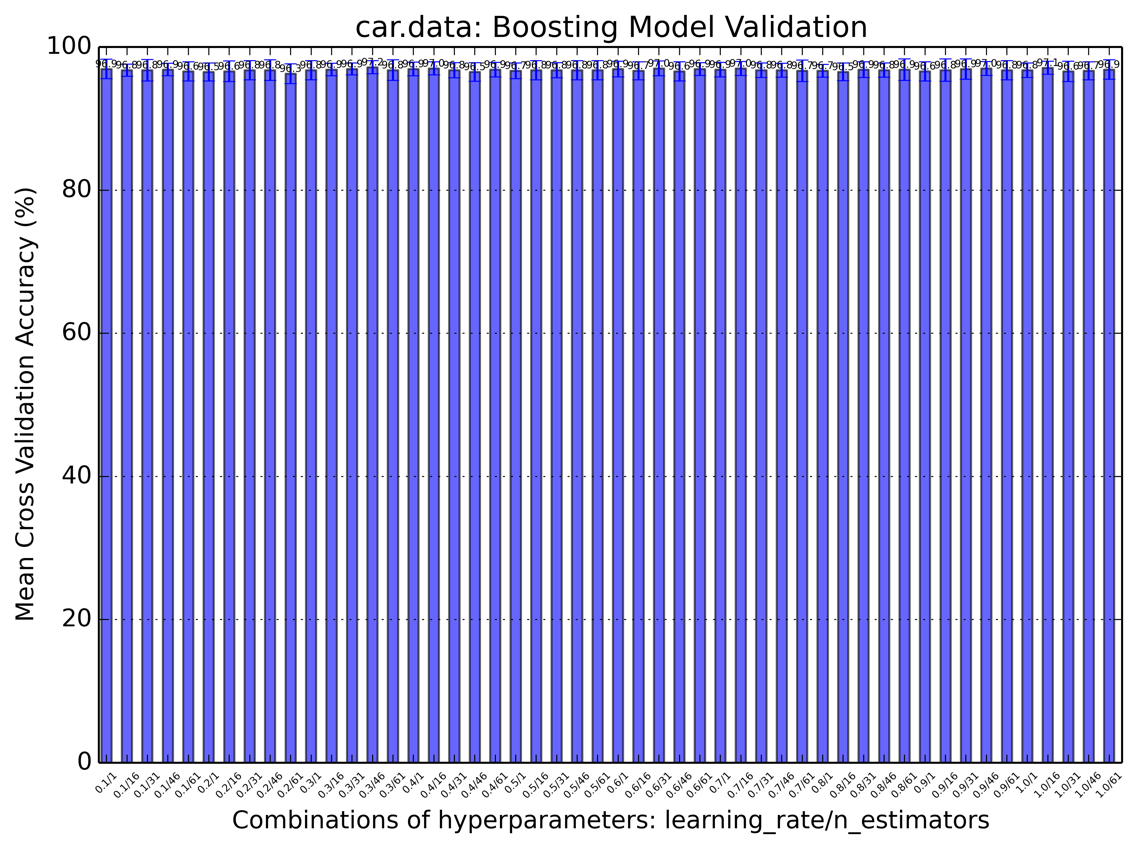
Training error stayed plat at 0 as example size grew, which indicates that DT fitted really well with the car data even when training space was low. And the cross validation error decreased gradually to a low point where it almost converged with the training error. This shows that adding more training examples will indeed help it converge. And this model is a little closer the high variance side.



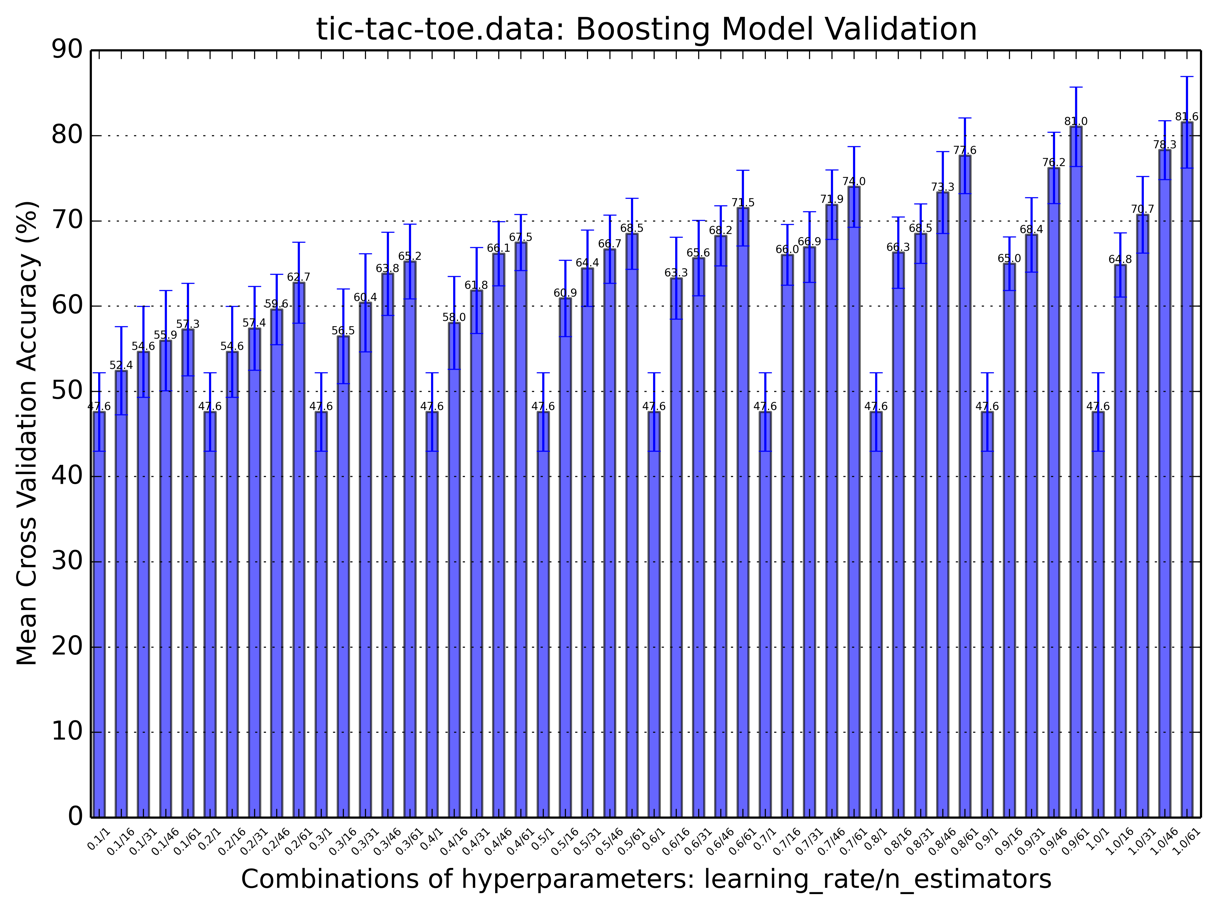
in this dataset, the training error stayed quite steadily at around 0.5 as training size increased. The Cross validation error first decreased drastically from training samples of 100 to 200, went below training error for a small range, and then fluctuated back up over size of 500. The unsteadiness here showed uncertainty or low confidence in the DT model prediction of this dataset. DT clearly could not learn too much no matter how much data is fed into it, at least not over this range of sizes. This estimator is at the high bias side.

## Boosting

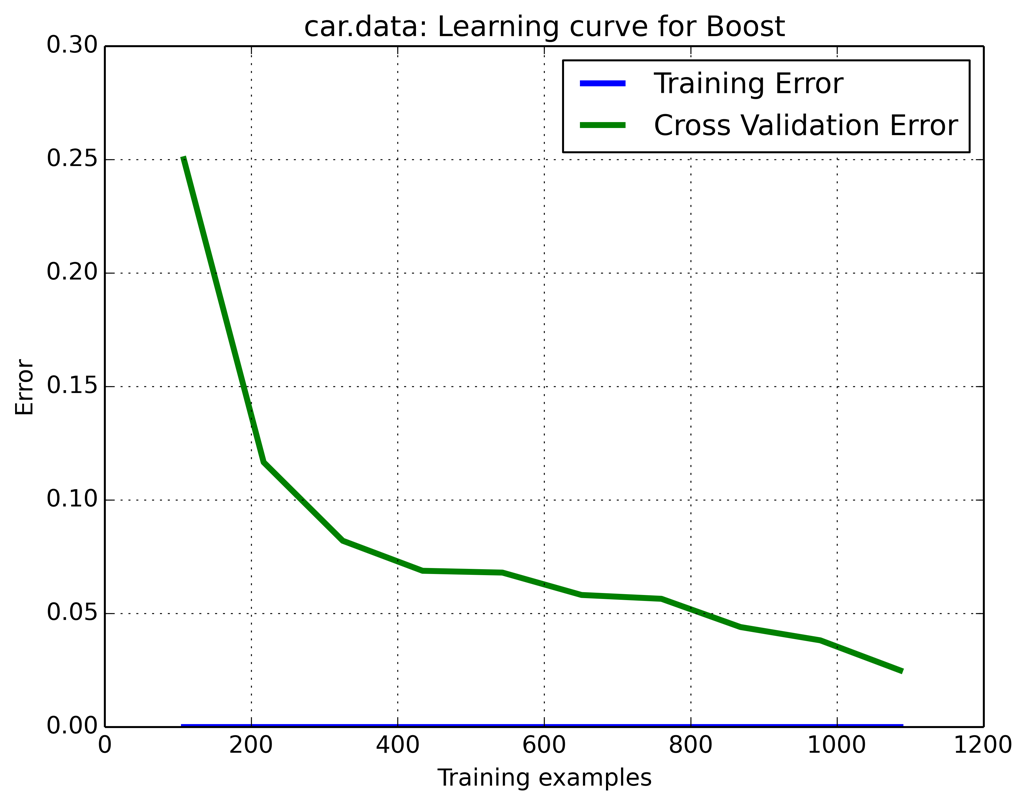
Each of the base estimator used here was tuned with the best parameters selected from above trial. The two parameters of interest for boosted Decision Trees were learning\_rate and the maximum number of base estimators (n\_estimators). The larger the n\_estimators is, the better/more complex it gets. The way adaboost does to train (i.e. increases weights of incorrectly classified instances at each iteration) can cause the model to quickly fit but it also induces overfitting. Learning rate helps shrink the contribution of each estimator to slow down the learning.



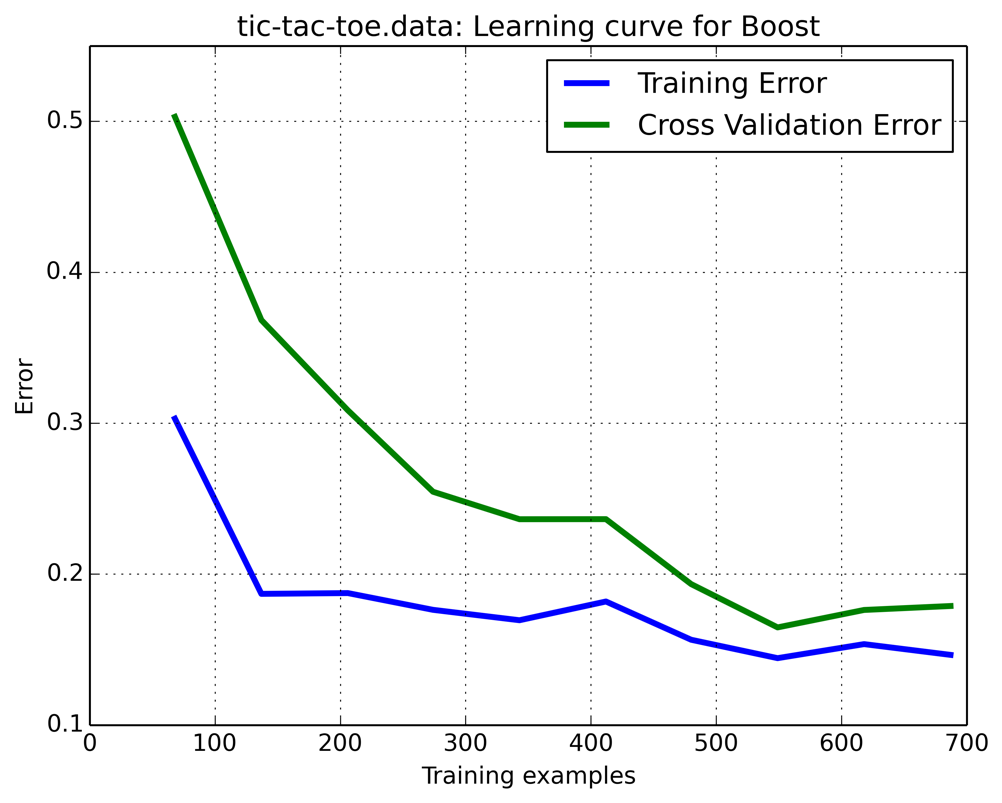
In the car data, each base estimator already performed well on the training data, the aggregated version of the trees did not seem to help much. All combinations display similar behaviors as the best DT model itself. Adding more estimators is not effective in increasing the accuracy. It means the base estimator may not have suffered too much from high variance. From the previous learning curve analysis of DT, we could tell the gap between cross validation error and training error was quite small, which also supports this assumption. The best parameter pair here is {learning\_rate:0.8, n\_estimator:51} (basically any combination).



Contrast to the first dataset, boosted DT took in effect very well on this one. DT itself performed poorly on tic-tac-toe data (below 50%). We can also see in this figure, all models with only one estimator reached around only 47.6% accuracy. But once we increased the number of estimators to above 16, accuracy increased dramatically. Learning rate helped in this case as well. Once learning rate was maximized, the overall performance reached to its higher potential. Boosting indeed corrected bias of the underlying weak learners greatly. The best parameter pair here is {learning\_rate:1.0, n\_estimator:61}



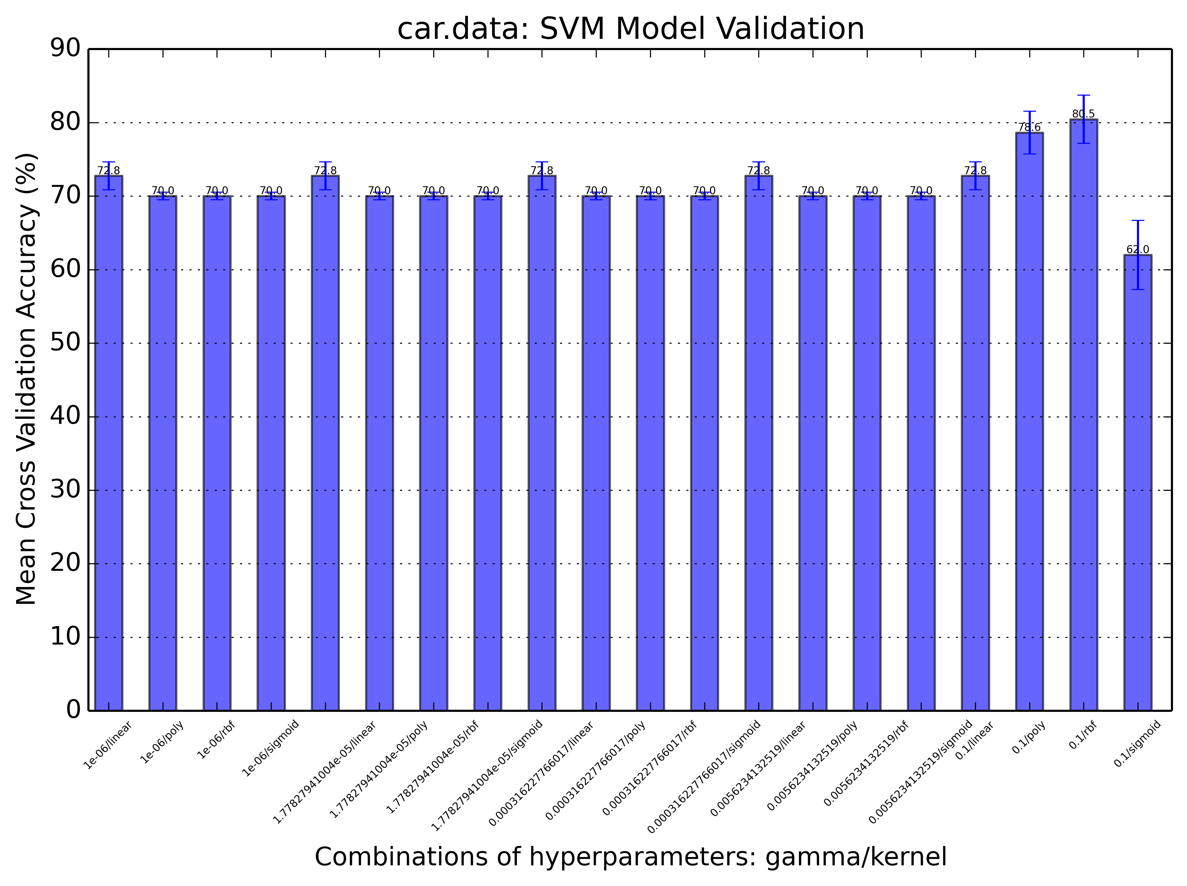
Still, similar to the DT analysis of car data before, the learning curve for boosted version left a small gap at the far right end. Training error stayed at zero. If more training data was added, it could help CV error to decrease further.



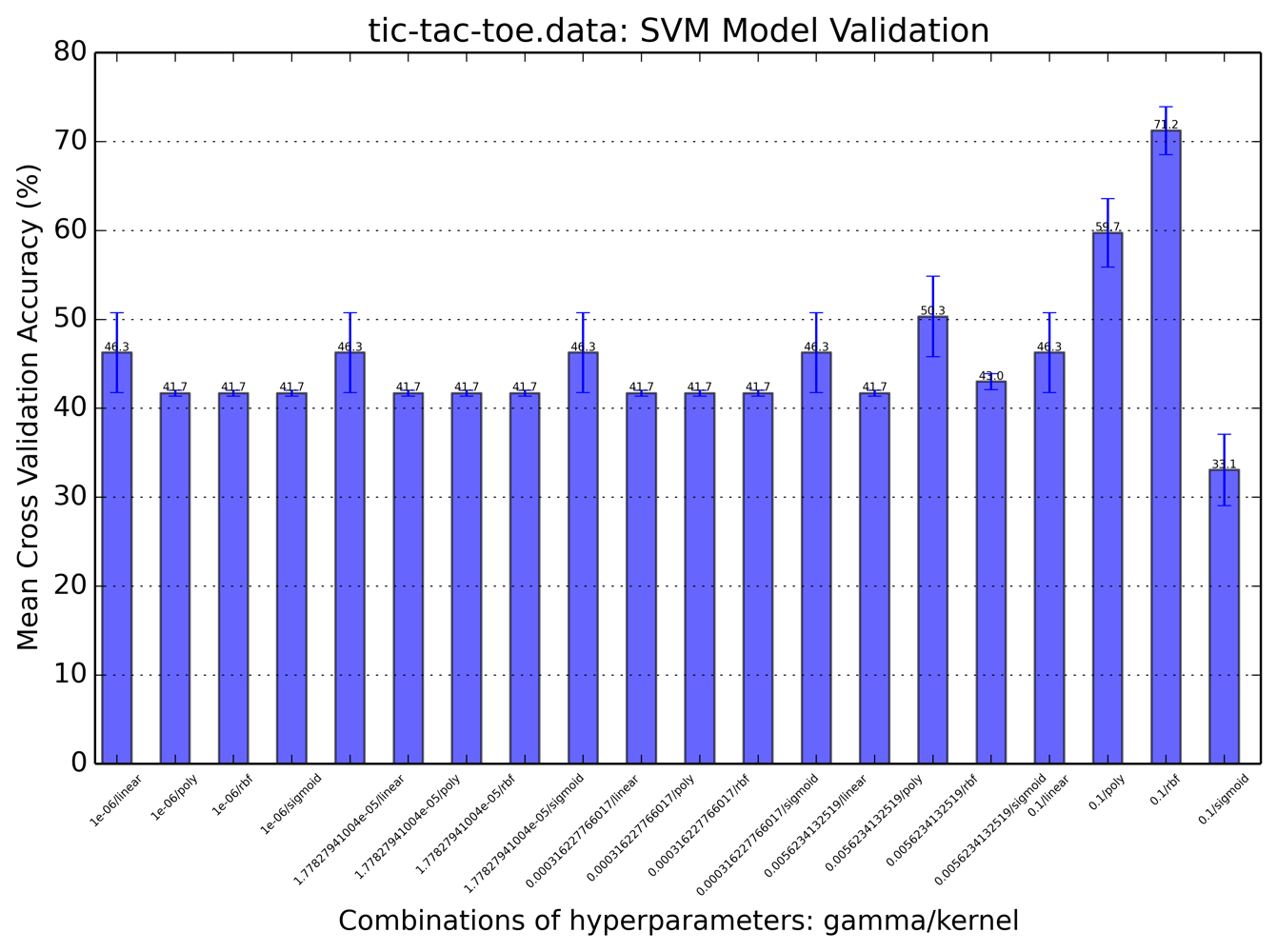
The best boosting model selected may still suffer from high bias or underfitting for this dataset. Towards the maximum training examples, gap decreased a lot, implying adding more training samples may not help much. In the future, enlarging the n\_estimator parameter may lead to a better performance to further reduce bias.

## SVM

The two parameters of interest for SVM were gamma and kernels. Kernel functions determine the way how SVM draw lines to separates data points. Large gamma itself may lead to high bias and low variance models.

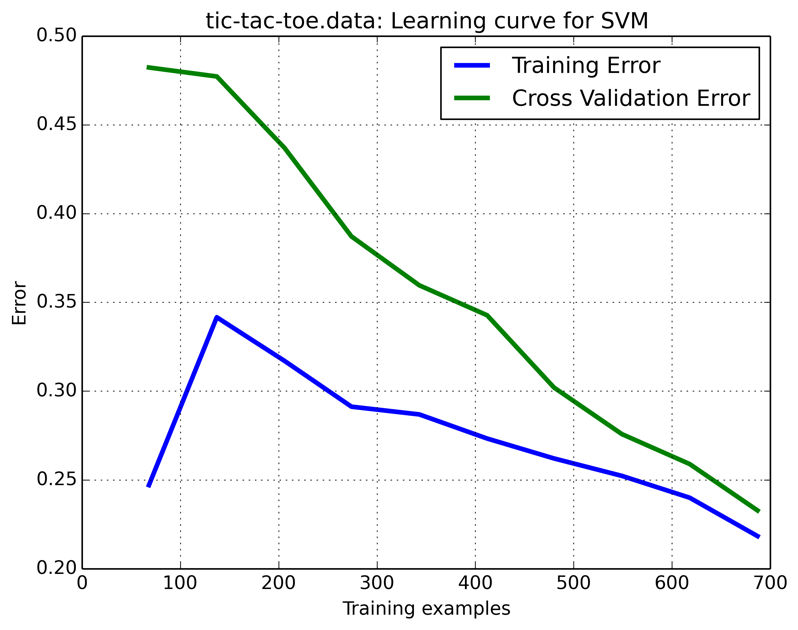
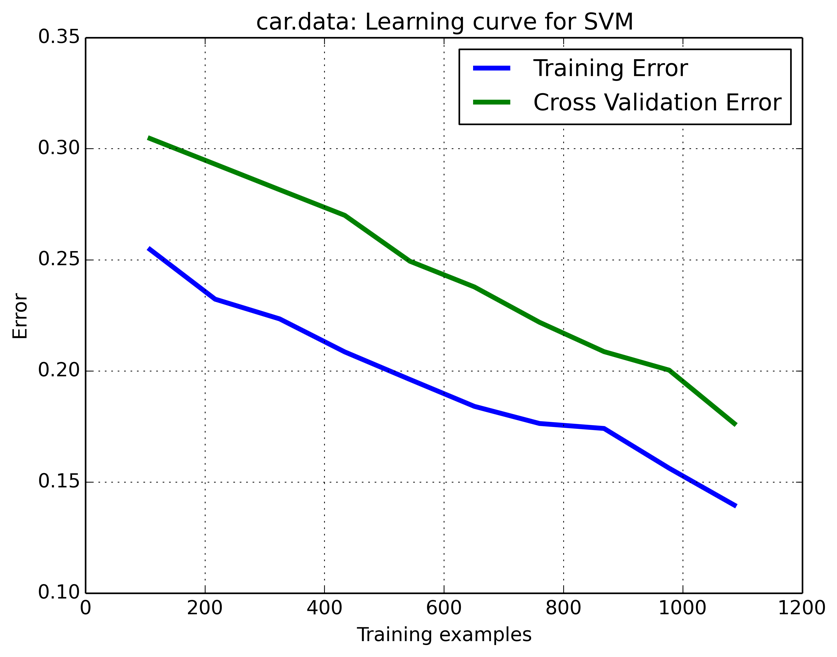


It is interesting to see that, with lower gamma values, kernel function sigmoid performed the best. When the gamma equaled to 0.1, rbf function outperformed all other functions. Overall, the performance of SVM on dataset 1 was moderately high and the differences between different kernel functions used were small.



As shown in the figure, the second dataset showed a graph that fluctuated in a similar manner as the first one, except the fact that all accuracy scores became lower. SVM did not perform well on the tic-tac-toe data may be due to the existence of outliers and noise in the data. The best parameters here were {kernel: rbf, gamma: 0.1}, too. But the differences between all kernel functions were larger than the last dataset.

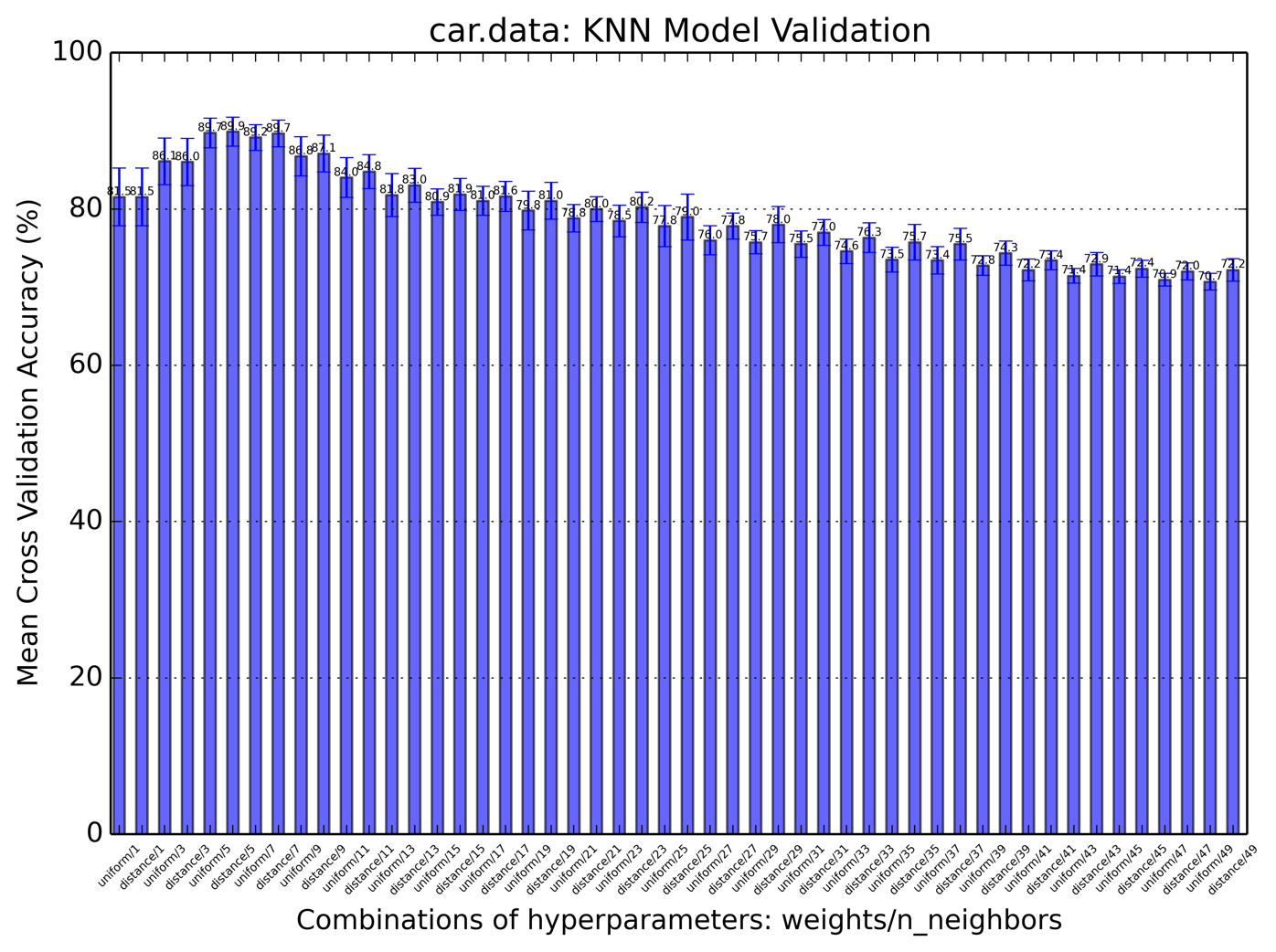
The two datasets were similar in that they both had a small number of attributes and similar number of instances.



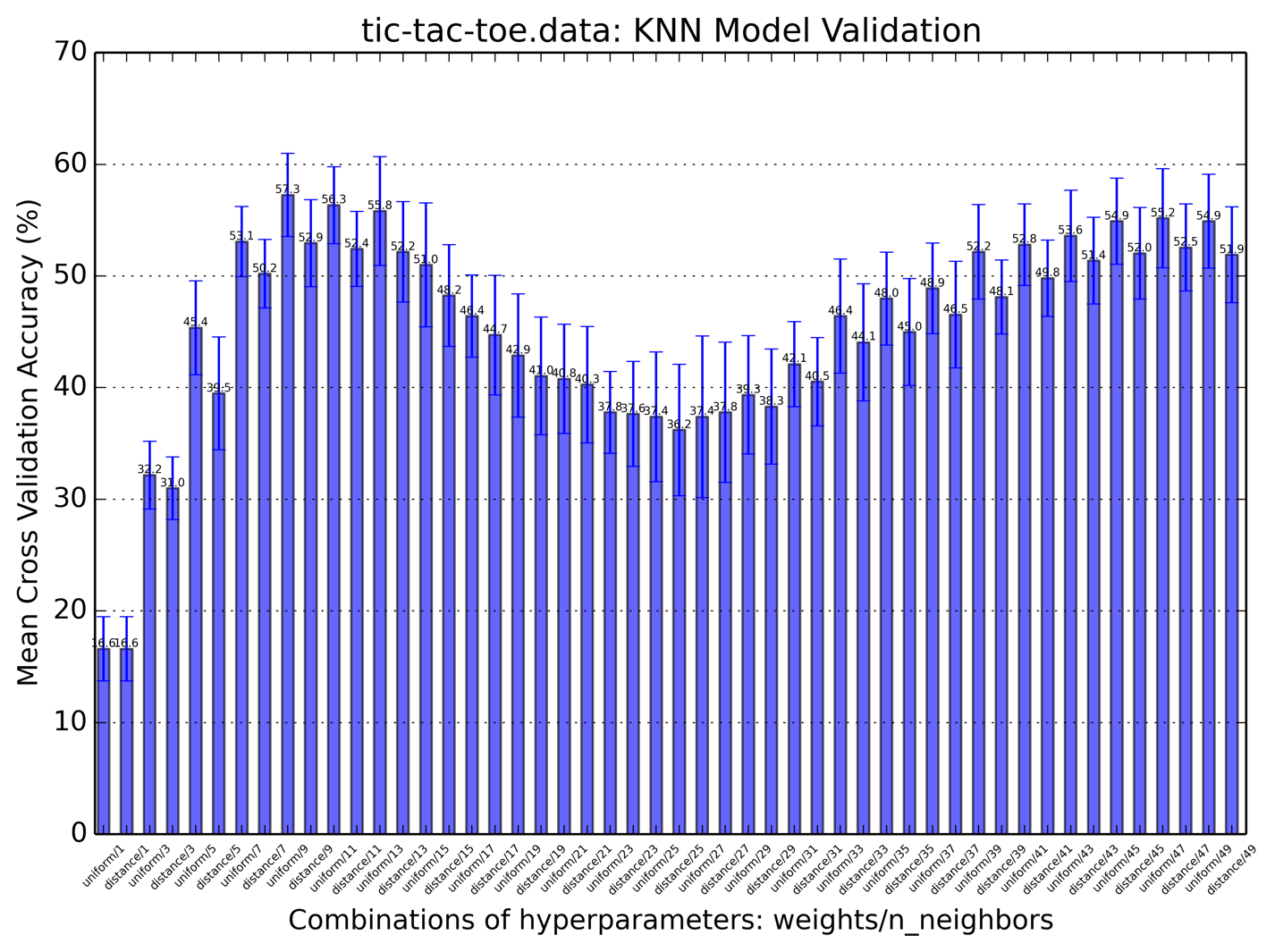
In both datasets, cross validation error and training error had the tendency to converge as the training samples increase. The second data left a smaller gap at the end. That indicates adding more training instances to it may not help as much as the first one. The car data learning curve implied a slight overfitting property of the model.

## KNN

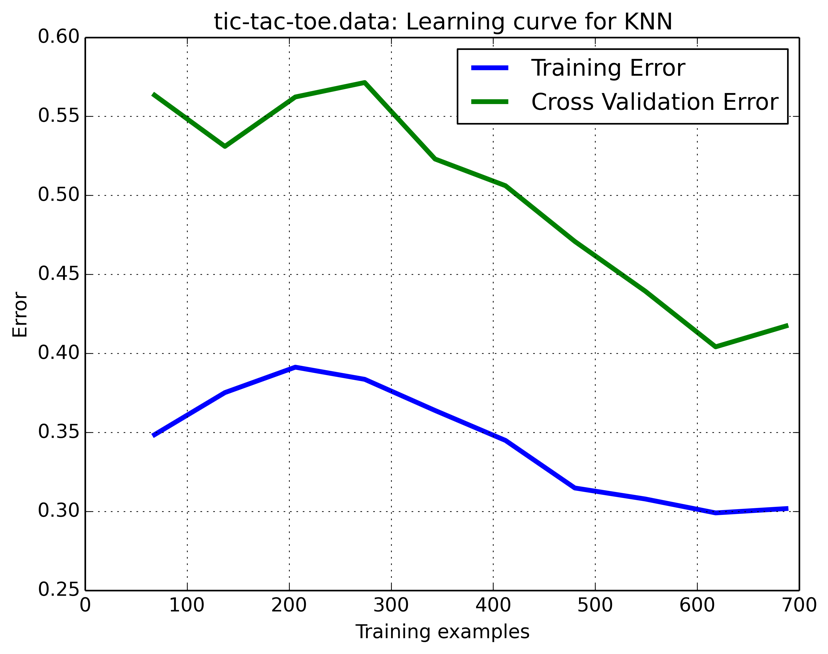
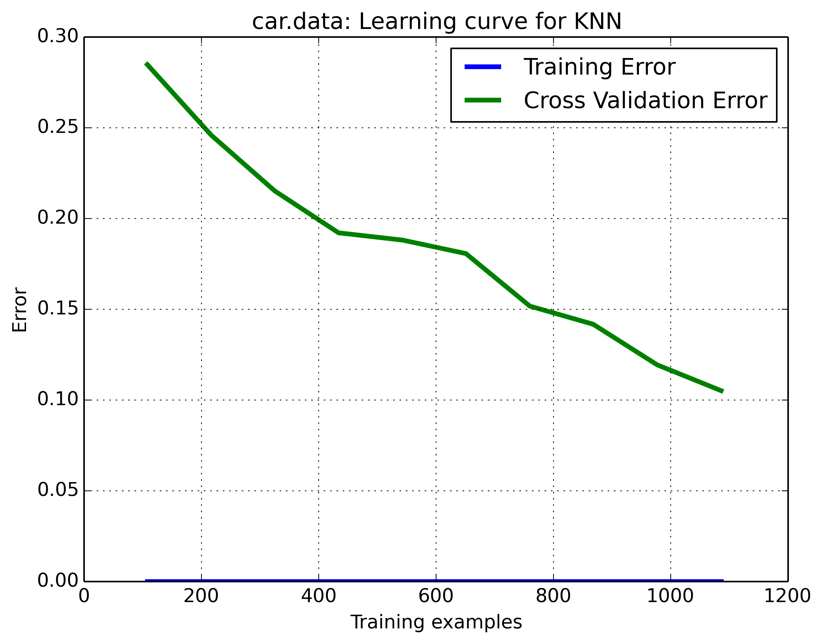
Two hyperparameters I considered for this estimator were the weight function used in prediction (weights) and the number of neighbors to use, k (n\_neighbors). The relationship between k and overfitting is inverse. A small k value often leads to overfitting models. Weights parameter determines how neighbors are weighed into the evaluation of a data point. This parameter should be adjusted based on domain knowledge.



For this data, we could see an overfitting point at around n\_neighbors of 5-7. Because the graph gradually increased to a highest point in accuracy as k value decreased to this area. And after that, smaller k induced a reduction in accuracy. The weight function did not have too much impact on the performance overall, given k value was fixed. But we can still see distance function did better in general. The best parameter pair was {weights: distance, n\_neighbors: 5}



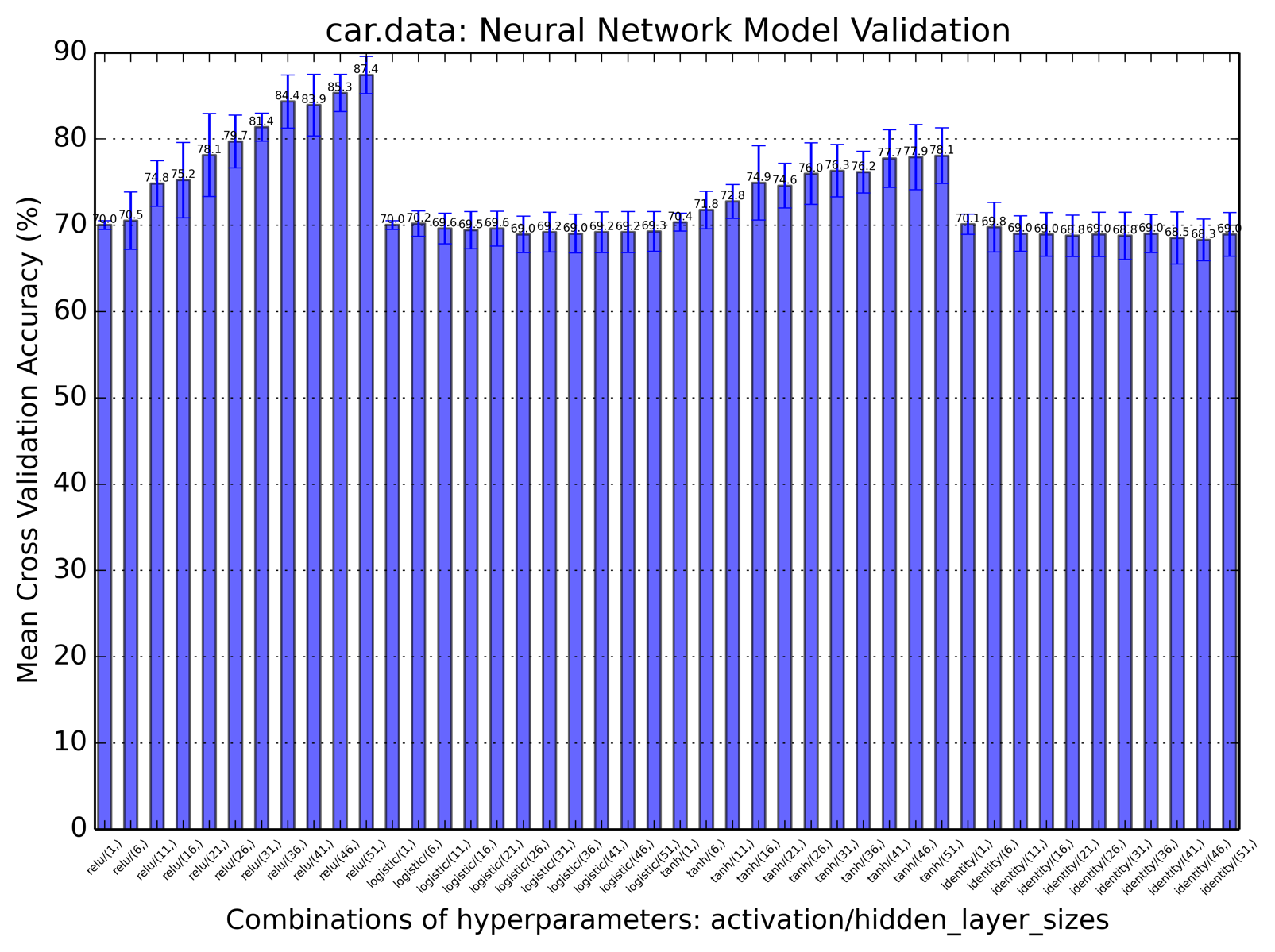
All the models for this data performed worse than last one. But the best k value was around the same area. In terms of weight function, uniform did better this time. There is also an interesting trend of the curve in this graph. There are two local hills and two valleys at two ends (low k and high k). This can be due to the fact that a seemingly drastically different configuration of the tic-tac-toe board is actually the mirror board of one, thus the two belong to the same class (e.g. win for x). So the further neighbors may contribute almost as much as near ones.



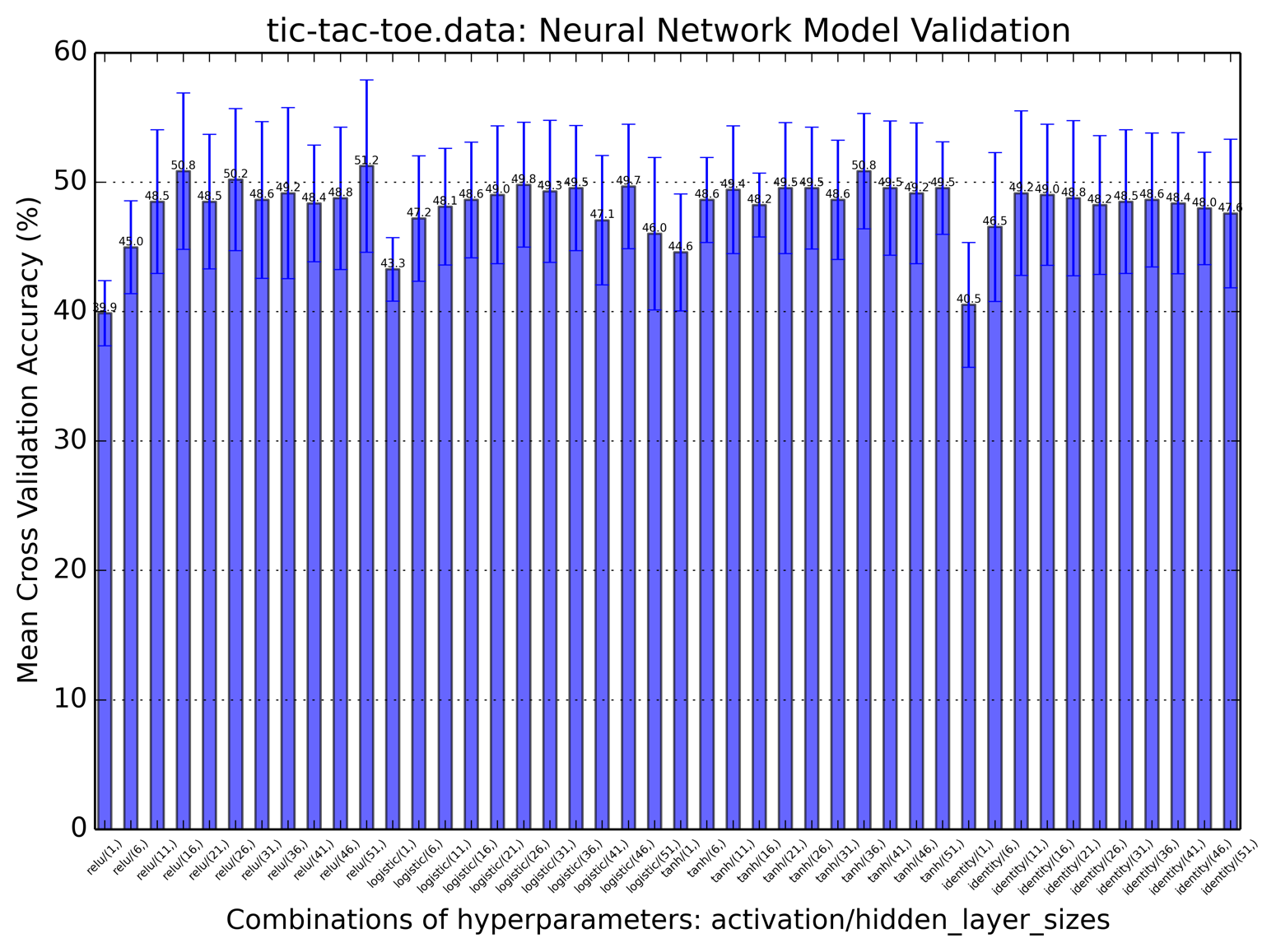
The learning curves for both datasets are similar. They both have big gaps at right end, which implies that the models are high-variance, and adding more training data can help.

## Neural Network

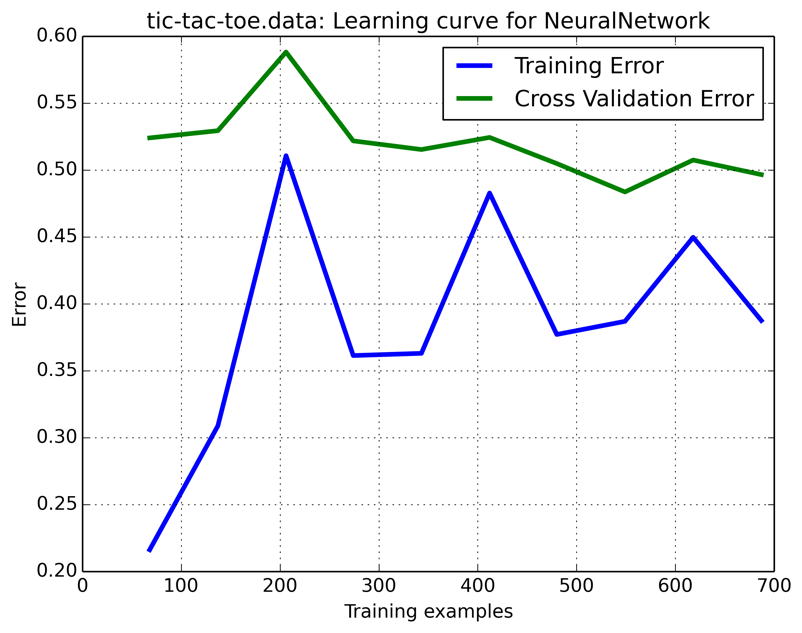
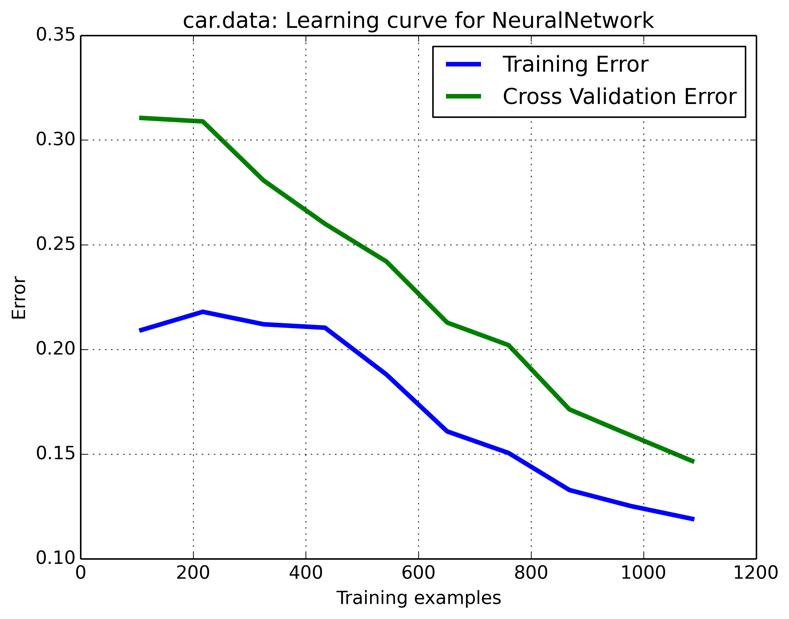
There was only one hidden layer used in the experiments. Activation function and size for the hidden layer were the two parameters I picked for Neural Network models. Increasing the number of neurons or size of the hidden layer will add complexity to the model. Different activation functions may introduce interesting performance comparisons.



The models shown here can be divided into two categories. One is that increasing complexity can induce higher performance, while the other one is that accuracy score does not change as layer size increases, given the same activation function. Relu as the activation function gave the best performance among all. And increasing layer size could actually help. More trials can be done in terms of increasing model complexity to see whether better accuracy can be achieved.



On the contrary, all activation functions showed similar behavior for this dataset. They all did equally badly in terms of accuracy. It is only that increasing hidden layer size could help in improving models. Small hidden\_layer\_sizes value brought high bias to the models. It may be helpful to try out other methods in making models more complex such as adding more hidden layers.



The learning curve for car data showed the selected model was only a little overfitted as there is a small gap between training and CV error. Adding more training examples would help boost performance.

The learning curve for tic-tac-toe data is much more interesting. It seems NN model is quite sensitive to the size of training data. Its model suffers from high bias because both training and CV errors tend to go to a high error rate.

In the future, to study more from NN models on the datasets, more layers/iterations should be used to evaluate whether the performance gets better and if it brings significant changes to the shape of learning curves.

# Algorithm Overall Comparisons

All of the estimators in this experiment were tuned using the best parameters from the model validation of each algorithm.

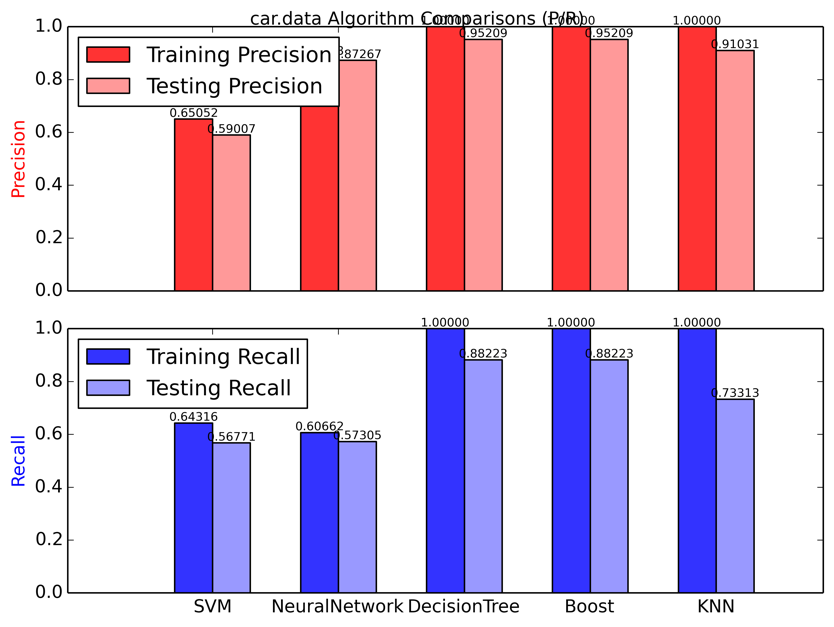
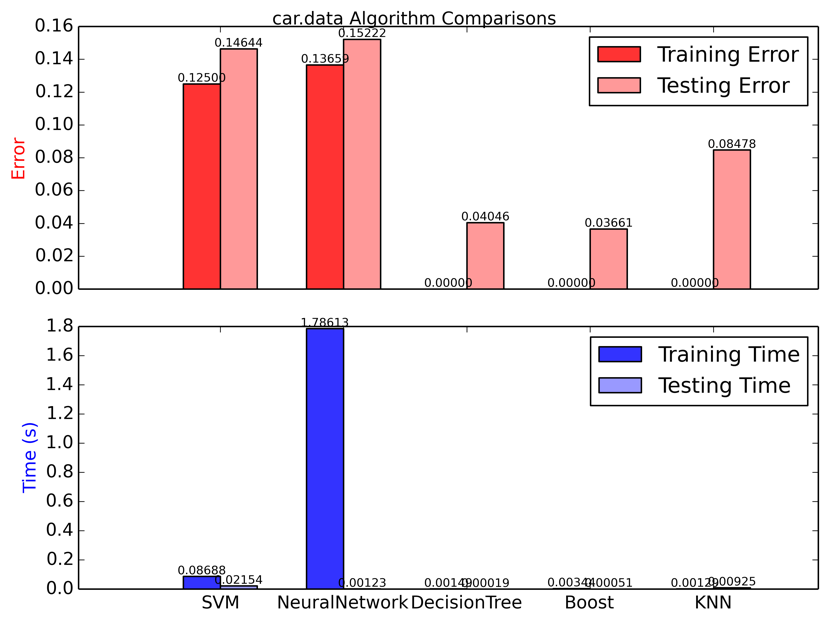


Figure 1. Comparisons for car data

The left subplot shows error rates and elapsed times of all five algorithms. The right subplot shows macro-based precision and recall scores calculated by the python metrics module. I chose to calculate precision (i.e. the probability of classifying any instance correctly) and recall (i.e. the probability of classifying a positive instance correctly) in addition to the error rates were mainly due to the unbalanced distribution of class values in the datasets. Having two more metrics would help strengthen the argument of whether an algorithm is better than another. For this car dataset, DecisionTree and Boosted DT (base DT pruned with the same hyperparameter values as DecisionTree) performed the best in terms of all three metrics. This is reasonable as the dataset has a known underlying concept structure among the attributes, due to the nature of decision trees, they are a good fit to map out the rules behind this data. Especially for the training set, DT and Boosted DT both gave 0% error rates. Boosting did not improve DT much. It shows that when base learner is good enough, boosting can only be little help. KNN was a little worse than aforementioned two. SVM was the worst of them all. It barely went over 0.5 on precision and recall. However, all the algorithms were a little overfitted to the training set since their testing results were worse.

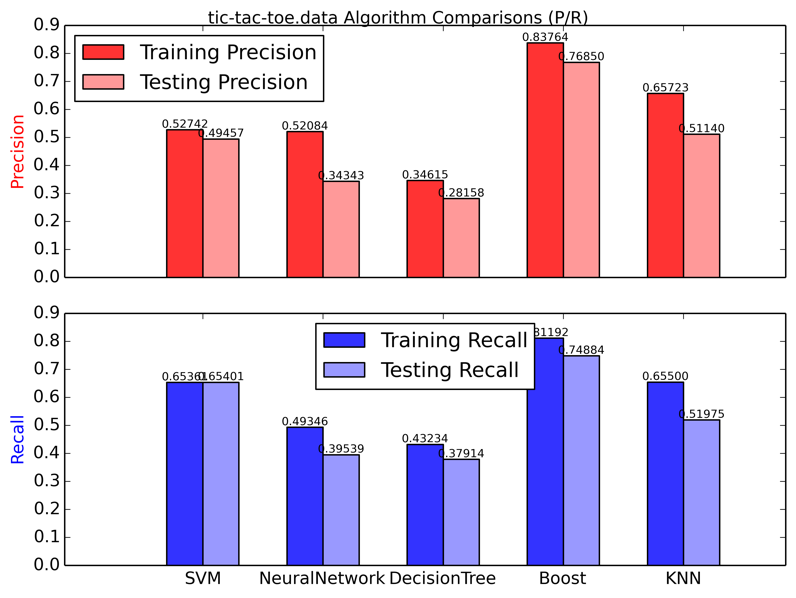
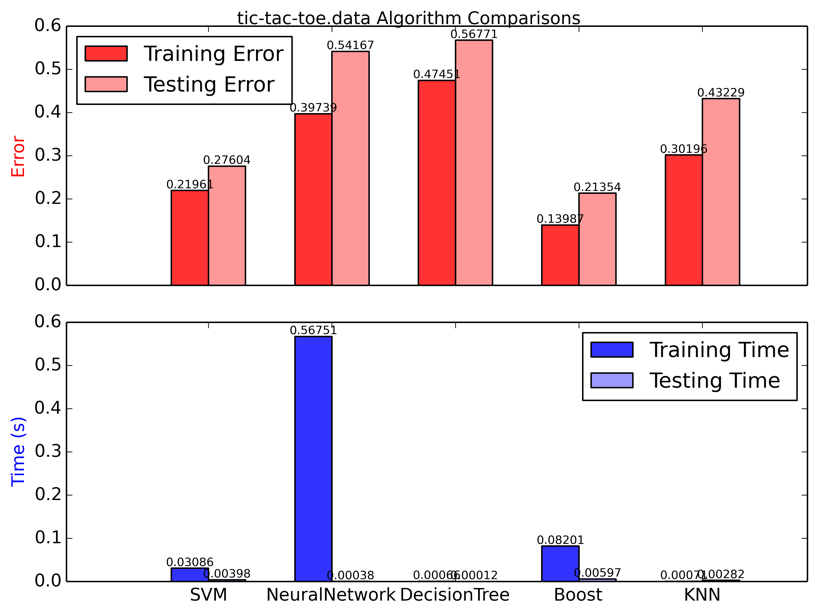


Figure 2. Comparisons for tic-tac-toe data

For this tic-tac-toe data, Boosted DT performed the best, although it took it quite a decent amount of time to fit. All the other algorithms did rather poorly on this dataset. Only SVM was ok (error rate was only ~0.28) and its training was moderately quick. The overall bad performance of this data was possibly due to small number of training instances. Interestingly, DT itself performed the worst (well below 50% of accuracy or precision) while boosted version of DT did quite well. This clearly demonstrates how much boosting can enhance the performance of a weak base estimator.

In all, none of the algorithms took extraordinarily long time to either train or predict, because the two datasets were pretty small and the number of attributes was small. DecisionTree, Boost and KNN spend little time on both training and testing generally. Comparatively, SVM and NeuralNetwork spend more time in training in general, especially NeuralNetwork. Among the five algorithms, only KNN does lazy training so it takes more time in predicting than training. KNN was not too time-consuming in my experiments was because in both datasets, best ks were about only 7, and also testing data was not too much. KNN would also use up a lot of space because it needs to store all of the data points at training time.