Exploration of Machine Learning on Housing & Automotive Datasets

Duncan Renfrow-Symon, drenfrowsymon3

1. Introduction

In this paper we will compare the performance of 5 distinct machine learning algorithms across two datasets: housing and automotive. We will explore the effects of different hyperparameter settings of each algorithm and discuss what aspects of the dataset may lead to those effects. The housing dataset is a regression problem while the automotive dataset is a binary classification one which will allow us to observe differences in performance across these types of problems.

2. Housing Dataset

2.1 Problem Description and Dataset Overview

Real estate is an intriguing domain to apply machine learning due to its key economic importance and large volume of structured data. One of the central problems in real estate is predicting the sale price of homes based on their unique attributes.

To explore this domain, I downloaded and analyzed a dataset of 21,613 homes sold in Kings County, Washington between May 2014 and May 2015. Each home had a sale price, date of sale, and 18 other attributes describing the home itself and the average sale price of comparable homes. An interesting aspect of this dataset is that both the dependent variable, price, and several independent variables such as square feet of living area follow a gamma distribution. In fact, the median sale price of homes in the dataset was $450,000 while the highest sale price was $7.7 million.

2.2 Algorithm Analysis

The dataset was divided into an 60% training, 20% validation, and 20% test set. Each algorithm was tuned using 5-fold cross validation on the training set. Cross validation is a technique that divides the dataset up into N folds and then trains the algorithm on N-1 folds and evaluates the performance on the Nth fold. This is repeated N times with each fold being “left out” in turn. The final cross validation performance is the average across each fold. Cross-validation is useful to reduce over-fitting and maximize the data available for training. For all graphs below, unless otherwise noted, the training curve is the N-1 folds while the validation curve is the Nth fold not used in training.

The test set was only used to evaluate the final comparative performance across the 5 algorithms below. This helps ensure that data in the test set doesn’t “leak” into the choice of hyper-parameters for a given algorithm.

The algorithms below were evaluated against mean absolute error (MAE) which provides an easy, human interpretable score.

2.2.1 Decision Trees

Two common techniques for restricting decision trees during the training process are limiting their depth, i.e. the number of sequential rules applied, or requiring a certain number of training samples to fall into each leaf node. **Figure 1** below shows the effect of reducing the max_depth against the mean error with no restriction on min samples of leaf. We see, intuitively, that a small max_depth restricts the decision tree to the extent that the cross-validation error rises dramatically. However, allowing the decision tree to grow to a large depth allows it to overfit and performance also deteriorates.

In **Figure 2** we see a similar pattern although in a much smaller range with the min samples per leaf and no restriction on the max depth. One interpretation of this is that the min_samples_leaf is less restrictive in the range we specified. The training partition of our dataset contains 12,967 rows so even requiring each leaf to have at last 100 samples is only 1% of the dataset. Interestingly when we inspect the decision tree built with a min_samples_leaf = 15 (the optimal value in our range) we see the max_depth is 18 which is near our optimal value of 9 in **Figure 1**. This strongly suggests there is a strong interplay between the max_depth and min_samples_leaf for reducing the size of a decision tree and improving its ability to generalize. One aspect to note is that the min samples per leaf is particularly useful for regression problems. This is because each leaf node actually returns the average of the prices of the samples that fall within it. By increasing the number of samples required in each leaf we are in effect requiring the algorithm to average over additional samples and "smooth" the prediction values.

For housing sale prices, with their gamma distribution, the min samples per leaf is likely a more important parameter than max_depth to tune because we strongly want to avoid over-fitting on the long tail of high priced houses. As a trivial example, there are two homes in the dataset with 11 and 33 bedrooms respectively. However, the prices of these homes are both less than $650,000 and their square footage is less than 3000 feet each. This strongly suggests that these are data anomalies. However, a decision tree could learn that bedrooms > 10 have an average price of $580,000. Unfortunately, when an actual mansion comes on the market with 12 bedrooms, the predicted price could be significantly off. A minimum samples per leaf bound prevents this overfitting while a max depth restriction does not directly (although indirectly it may be likely because that specific of a rule is unlikely to offer much information gain).

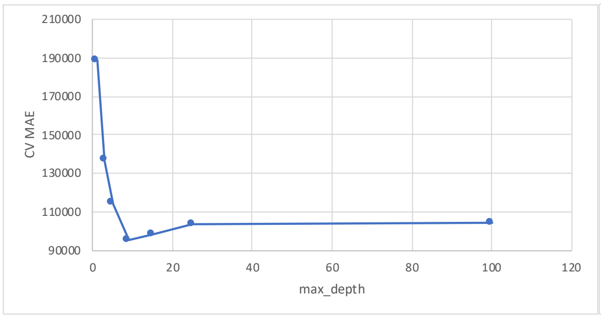


Figure 1: Average Cross Validation MAE performance against max_depth

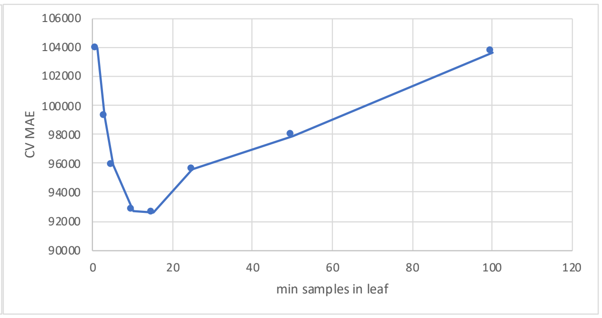


Figure 2: Average Cross Validation MAE performance against min_samples in leaf

As mentioned above we can also prune our decision tree after it has been constructed. The library scikit-learn provides a convenient measure of pruning called cost complexity pathing (ccp). At high level, the scikit-learn technique finds the least important nodes and prunes them based on a ccp_alpha threshold. Higher ccp_alpha values prune more nodes until there is only 1 node left (which is the maximum ccp_alpha_value). In **Figure 3** we demonstrate pruning various thresholds of nodes of a decision tree built with min_samples_leaf=10. We can see by the graph that modest levels of pruning improve our accuracy but additional pruning rapidly deteriorates our accuracy. Pruning with ccp_alpha is able to improve our accuracy further to an MAE of 89,119 vs. a best MAE of ~92,500 for tuning best_min_samples alone.

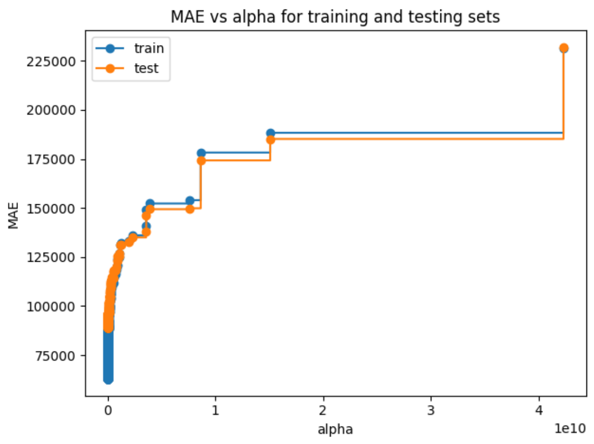
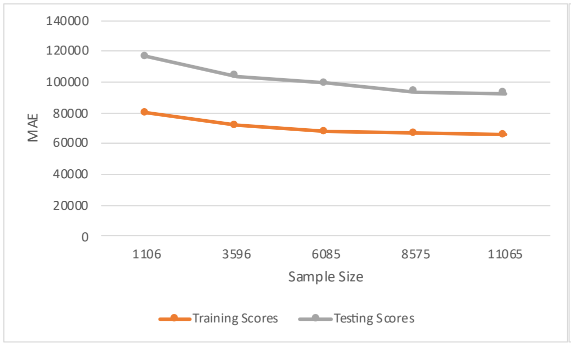


Figure 3: ccp_alpha values vs. MAE

Once we have discovered the hyperparameters that seem to work well we can show the performance of the algorithm as we include more data. In this case we will use the decision tree built with min_samples_leaf = 10. The surprising result shown in **Figure 4** is that increasing the data by approximately 10-fold only results in a 20% improvement (MAE of 116173 to 92708). This suggest that decision trees perform quite well on limited datasets and are able to create generalizable rules quickly. In **Figure 5** we can see that decision trees are quite performant, training in <0.2 seconds for all data samples, and scale linearly in training time and have roughly constant scoring time.



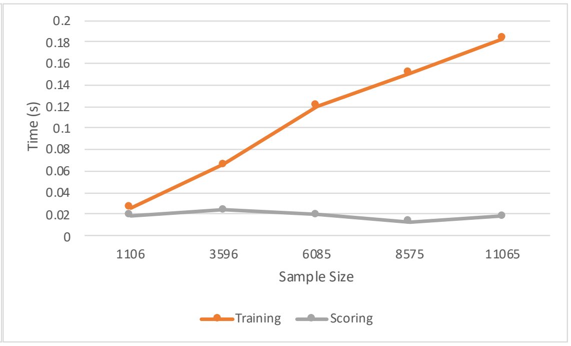


Figure 4: MAE training / testing scores vs. sample sizes

Figure 5: Scoring / Fitting times vs. sample size

2.2.2 Neural Networks

Neural networks are a challenging algorithm to tune due to their immense number of hyperparameters and potential architectures. To establish a baseline I choose an architecture with a single 64 node hidden layer with a skip connection. The skip connection is taken from the **ResNet** paper and is designed to allow information from the input layer to flow directly to the output layer if necessary.

2.2.3 Boosting

XGBoost is a popular implementation of a gradient boosted strategy which introduces several improvements such as Newton boosting (a more complicated gradient descent update) and increased regularization (to avoid overfitting) (**citation)**. Like other gradient boosted techniques, XGBoost typically relies upon decision trees as the weak learner and so the max_depth of the trees is a key hyper-parameters. As we can see in **Figure 6** the performance curve mirrors that of the above decision trees; a low max_depth leaves the algorithm with high bias while a high max_depth starts to over-fit. Interestingly the performance deterioration of smaller max_depths is much less than a single decision tree because XGBoost is able to ensemble multiple weak rules. Another interesting point to note is that the runtime performance of XGBoost deteriorates significantly with larger trees, which suggests focusing on a lower max_depth on larger datasets. See **Figure 7.**

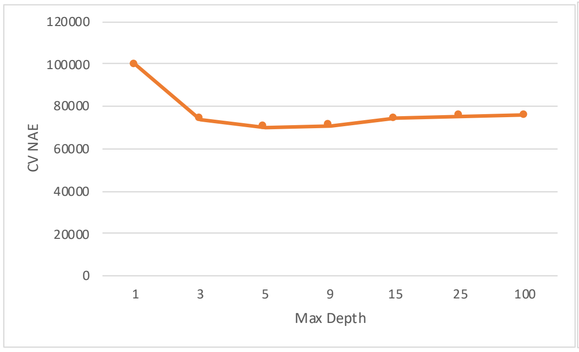


Figure 6: MAE vs. max depth

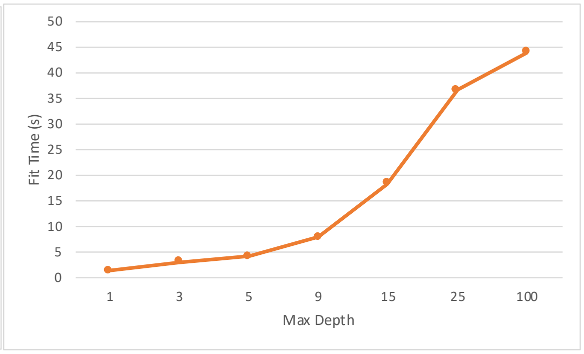


Figure 7: Fit times in seconds vs. max depth

Now let's turn our attention to the key parameters of the boosting portion of the algorithm. The most important one is learning rate, which controls how quickly to adjust our predictions based on the addition of new weak learner, i.e. the alpha in this formula **insert gradient descent formula.** Note that in actuality XGBoost uses a more complex gradient descent step but that is out of the scope of this paper. In Figure **8** we can see that varying the learning rate has a dramatic impact on model performance with a fixed max_depth of 6 (the default). Too low of a learning rate leads to significant underfitting as the weak learners are unable to correct the predictions fast enough. Too high of a learning rate seems to lead to slight overfitting as the algorithm adjust the predictions too much based on each learners results.

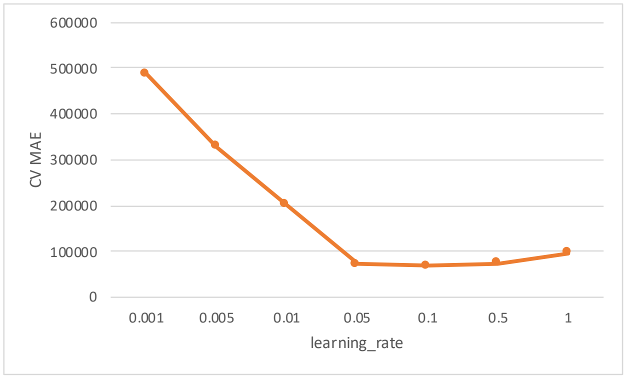


Figure 8: MAE vs. learning rate

Since gradient boosting algorithms are iterative we can see the performance over time on both the training and validation set. As Figure **9** shows there is a steep initial decline in both training & validation error and then a long plateau of incremental improvement on the validation curve. One interesting note is that this graph does seem to corroborate Professors Isbell and Littman's point that gradient boosting is resistant to overfitting. We do not see the characteristic rise in validation error which indicates overfitting even though this model is trained for a long period without substantial improvement.

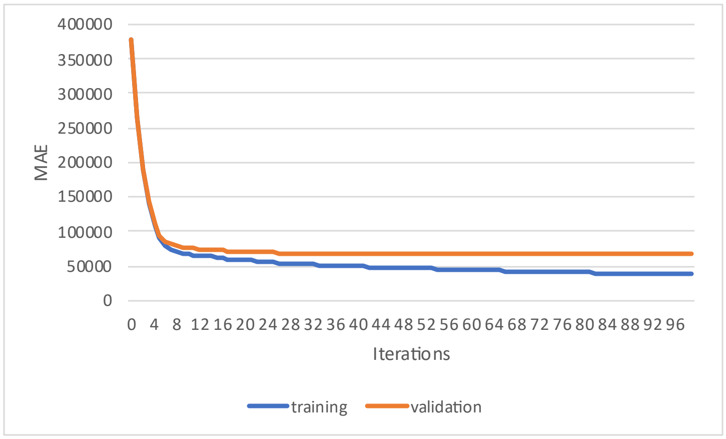


Figure 9: MAE for training / validation vs. number of iterations

In **Figure 10** we can see the performance of XGBoost across varying sample sizes. XGBoost achieved its best training set performance on the smallest sample of data, likely because of its ability to reduce bias through ensembles. In contrast, Decision trees reporting their worst training performance on the smallest sample. Yet despite this near perfect MAE on the training portion of the lowest sample of data, we see that XGBoost still achieved a lower validation score than a regular Decision tree. This is a strong indicator of the power of the boosting approach to reduce variation even on limited samples of data.

From a runtime performance standpoint we see that XGBoost appears to have a logarithmic curve with regards to fit time in **Figure 11**. This is surprising as a single decision tree demonstrated a linear curve with regards to fit times. It would be interesting to explore this curve on a larger dataset to see if this trend continues or we see a more linear curve emerge as the data processing overshadows any fixed cost algorithmic overhead.

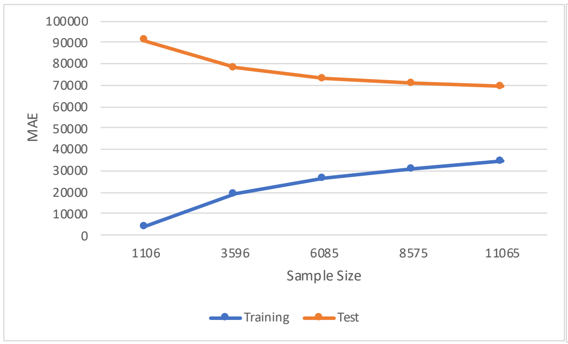


Figure 10: MAE for training / validation vs. sample sizes

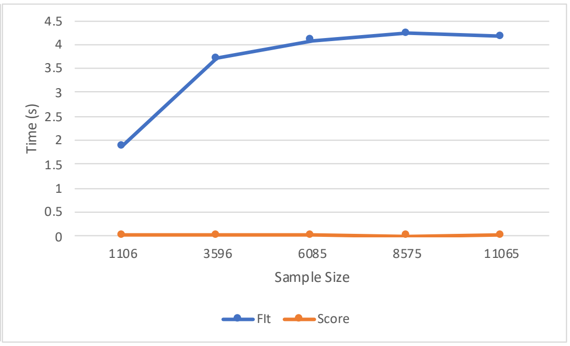


Figure 11: Fit / Score time in seconds vs. sample sizes

2.2.4 Support Vector Machines

A key detail of SVMs is their use of kernels, i.e. mathematical functions that project data points into higher dimensional spaces in order to make them linearly separable. An important optimization is the use of the "kernel trick" which enables the algorithm to calculate the required projection into higher dimensional space without actually computing the mapping.

There are a broad array of practical kernel functions and choosing the correct one is largely based on domain knowledge. Certain kernel functions allow more expressive feature interaction, e.g. the polynomial kernel, or the creation of non-linear decision boundaries, e.g. the radial basis function (RBF) kernel. In Figure **12** we see the performance of 3 different kernels. Somewhat surprisingly the complexity of the kernels seems inversely related to their performance. This may be because the dataset contains relatively simple features or perhaps the simpler kernel is better suited to dealing with noisy / incorrect data. We can explore this later idea further by taking the linear kernel and experimenting with the regularization parameter C, which controls the penalty of misclassifications. A lower C reduces the penalty for misclassifications and thus allows a more general decision boundary. Looking at Figure **13** we see a surprising result, as C increases for our linear kernel the MAE continues to decrease. This is particularly surprising given the default value of C in scikit-learn is 1.0 and yet a 100x higher value results in the best performance. One possible explanation, according to this source (https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-in-svms-with-linear-kernel) is that small values of data can tolerate very high C parameters. Many of the variables in our dataset have gamma distributions, e.g. square feet. We also scaled our dataset to unit variance which resulted in many smaller values. It will be interesting to see if this trend holds.

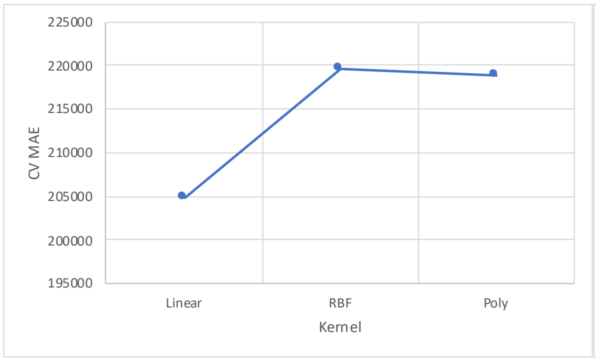


Figure 12: MAE for each kernel function

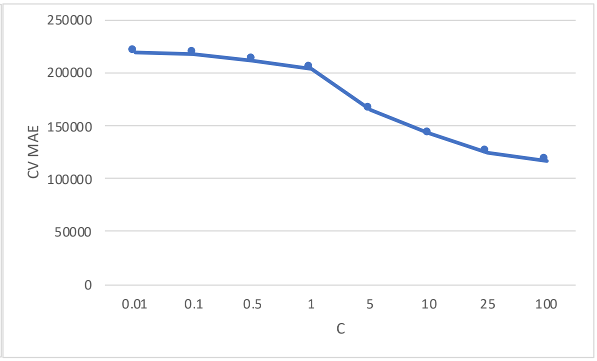


Figure 13: MAE for varying parameters of C

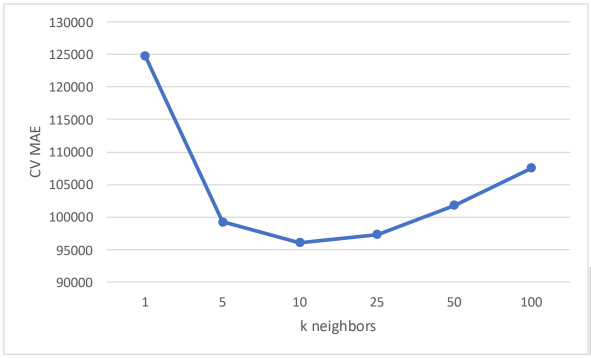
Now that we know which hyperparameters perform well we can look at the performance across various sample sizes. Unfortunately after attempting to run a learning curve over several hours we were unable to generate any results due to the length of the computation.

One possible reason is that there is a well known issue with SVMs which is their superlinear scaling with the number of data points. In fact, as Williams & Seeger point out in their Nystrom paper (**citation)** SVMs scale in O(n^3) time with regards to the training examples. One common technique is to use the Nystrom approximation introduced in the above paper to approximate the kernel and speed up runtime.

2.2.5 k-Nearest Neighbors

The most obvious, and import, hyperparameter for knn is the number of neighbors, k, to average. In **Figure 14** it is easy to see that only relying upon the closet single neighbor results in overfitting and poor CV MAE because the decision boundary is too narrow. Conversely the error increases as the value of k increases past a certain point because the decision boundary grows too large and starts to average dissimilar homes.

Quite naturally this brings up the question of how we define our nearest neighbors. As Professors Isbell and Littman noted in their course "Machine Learning" the distance metric used depends on your domain knowledge of the problem. In this case the knn algorithm defaults to Euclidean distance, i.e. sqrt(sum((x - y)^2)). However, there is evidence in the literature that for high dimensional data Manhattan distance may be preferable (https://bib.dbvis.de/uploadedFiles/155.pdf). Another interesting opinion is offered by William Raseman on his blog (https://waterprogramming.wordpress.com/2018/07/23/multivariate-distances-mahalanobis-vs-euclidean/). He demonstrates and argues that Mahalanobis distance, sqrt((x - y)' V^-1 (x - y)), handles datasets with highly correlated features better than Euclidean distance. If true, this is quite appealing for our dataset because many features in real estate are quite correlated, e.g. bedrooms, bathrooms, and square feet. Interestingly in **Table 15** we see that the Mahalanobis distance metric actually performs the worst and that Manhattan distance performs the best (albeit by a small margin). This may be a reflection of the dimensionality of our dataset as noted above.

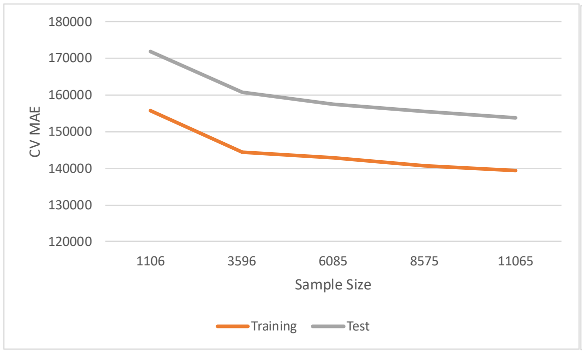


manhattan	euclidean	mahalanobis
90996.9077	96068.6106	128289.371

Figure 14: MAE vs. k neighbors

Table 15: MAE vs. distance metric

Now that we have discovered the hyperparameters that seem to work well we can show the performance of the algorithm as we include more data. In this case we will use a knn with Manhattan distance & k=10. As we can see in **Figure 16** performance continues to robustly improve as we increment the data. This is a surprising difference between knn and the other machine learning algorithms. In **Figure 17** we can observed the unique, but expected pattern of knn where the scoring time continues to increment as the sample size increases but the fit time remains constant.



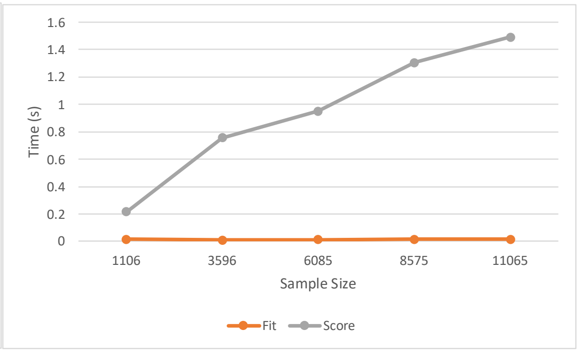


Figure 16: MAE of training / test set vs. sample size

Figure 17: Fit / Scoring time in seconds vs. sample size

3. Automotive Dataset

3.1 Problem Description and Dataset Overview

Similar to real estate, automotive data is an interesting domain to explore given the large volume of structured data and economic importance. To explore this domain I downloaded a dataset containing 72,983 vehicles recently sold at auction with 34 features.

In contrast to the prior dataset, the goal of this dataset is to predict whether or not a car recently sold will be defective. Thus, the output prediction will be a probability between [0.0, 1.0]. To measure the performance of our algorithms we will use a metric known as Log Loss, i.e. $-(y \log(p) + (1-y) \log(1-p))$ where y is the true label {0,1} and p is the probability estimate from the algorithm. Log Loss is the preferred metric for binary classification because unlike simple accuracy, i.e. % of correct labels, it exponentially penalizes the algorithm based on the magnitude of the probability estimate error.

There are ~30 additional features in this dataset pertaining to the both the auction and the vehicle sold, e.g. type of wheels, make, model, etc.

An interesting aspect of this dataset in contrast to the prior real estate one is the presence of categorical variables. In particular many of these categorical variables have high cardinalities, e.g. the variable "Model" has over 1000 unique values. This can cause issues for certain algorithms, such as k nearest neighbors, which are highly sensitive to the dimensionality of the dataset. This situation is one of the many reasons why the real-world performance of certain machine learning algorithms is highly correlated with how data elements are represented.

3.2 Algorithm Analysis

For ease of experimentation, the dataset was divided into an 30% training, 20% validation, and 60% test set. Each algorithm was tuned using 5-fold cross validation on the training set which contains 23,354 rows.

3.2.1 Decision Trees

As discussed above, we focus first on the max_depth for decision trees. Surprisingly in **Figure 18** we see that the best max_depth is 3. This suggests that the tree is overfitting on relatively simple rules. To investigate this theory we can experiment with varying levels of min_samples_leaf. As discussed in the housing dataset section on decision trees, this parameter helps prevent overfitting by forcing the tree to only create leaf nodes with a minimum level of nodes. As we see in **Figure 19**, as we increase the min_samples_leaf all the way up to 1000 samples we see improving performance. T

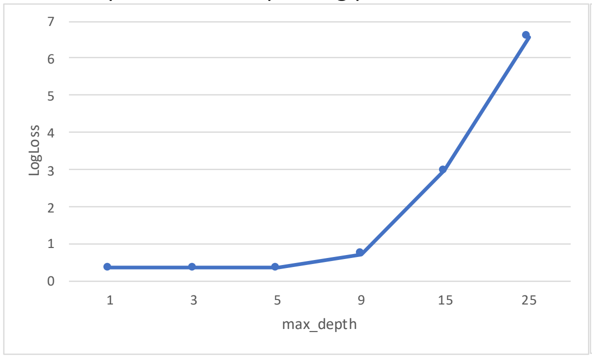


Figure 18: Logloss vs. max_depth

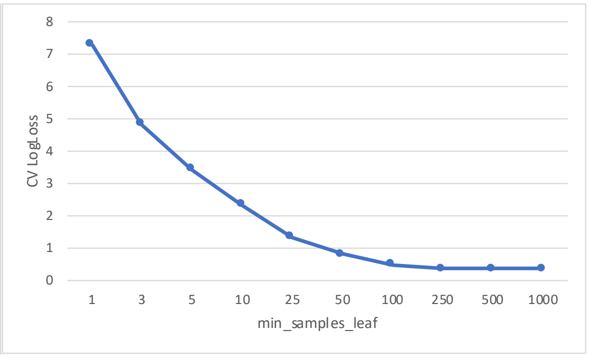


Figure 19: Logloss vs. min_samples_leaf

We can further investigate this trend by looking at post pruning. In particular, let's use a max_depth = 9 and then prune the tree using the optimal ccp_alpha parameters as discussed above in the house data section. As we can see in **Figure 20,** pruning makes an enormous difference and drops our logloss from ~0.65 all the way down to ~0.35. This suggests that there may be some relatively simple heuristics in this dataset that lead to reasonable accuracy and that going beyond those leads to overfitting.

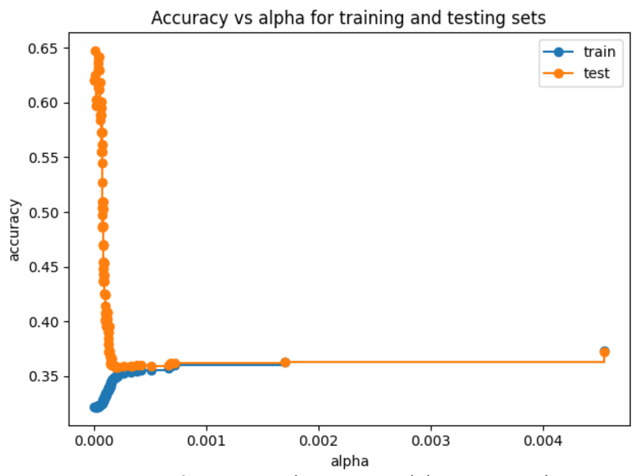


Figure 20: Logloss vs. ccp_alpha pruning value

Let's examine how well our decision tree performs across various samples. We'll use a max_depth = 5 because that appeared to be right on the cusp of overfitting on our full dataset. In **Figure X** we see evidence of overfitting with a max_depth of 5 in the smaller samples given the large gap between the training and validation datasets. However, our validation loss continues to drop as we feed it more data suggesting that a max_depth of 5 is more appropriate for the full training dataset. Although not shown here, the fit and scoring times follow the same pattern as the housing dataset decision tree.

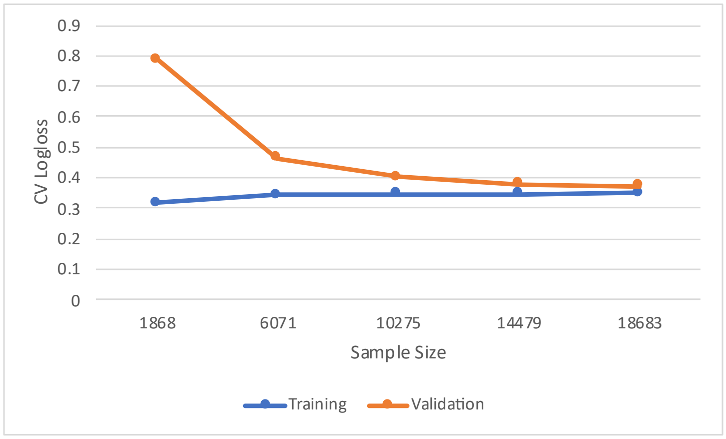


Figure 21: Logloss of training / validation vs. sample size

3.2.2 Neural Networks

Like the above housing dataset the neural network architecture was fixed at 64 hidden units with a skip connection. The first experiment I did was to explore the learning rate using a technique developed by Leslie Smith in his paper: "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates". In **Figure 22** below we can see that the appropriate learning rate appears to be somewhere between 10e-3 and 10e-2 for this problem. I then tried tuning the neural network with several different optimizers. The goal of these optimizers is to find the best gradient descent step to converge at the (hopefully) global optimum in the shortest possible time. Based on the literature I looked at 3 optimizers: Adam, AdamW, and RMSProp. Looking at **Table 23** we can see that AdamW ultimately resulted in the lowest logloss.

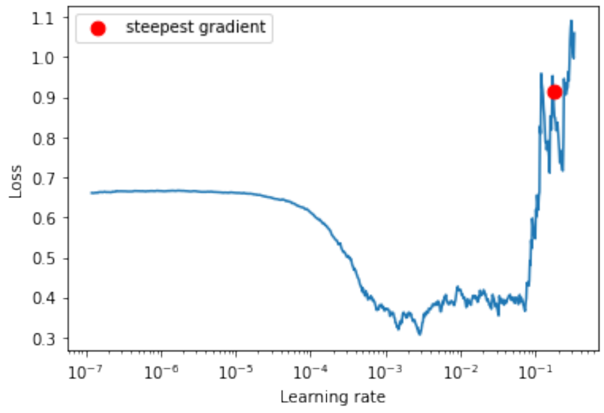
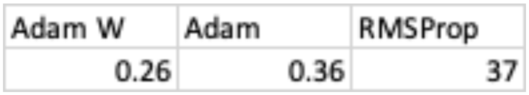


Figure 22: Logloss of training / validation vs. sample size



Adam W	Adam	RMSProp
0.26	0.36	37

Table 23: Logloss vs. optimizer

3.2.3 Boosting

As in the above housing dataset, we'll use XGBoost and first explore the min_depth parameter to see the effect on tuning the underlying weak learner Decision Trees. As shown in **Figure 24** the behavior of XGBoost matches that of the earlier decision trees with an optimal max_depth around 3. We can then use that max_depth = 3 and explore various learning rates to if it differs from our housing dataset XGBoost given the lower max depth. In this case we can see an optimal learning rate of ~0.1. This is surprisingly identical to the optimal learning rate used in the housing dataset, even though the max_depth = 9 in that model. This suggests that the learning_rate is somewhat independent of the construction of the underlying weak learner.

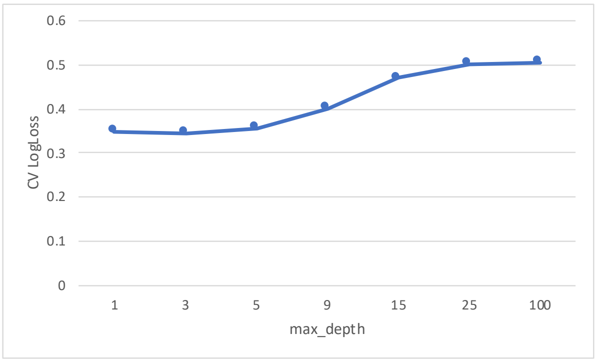


Figure 24: Logloss vs max_depth

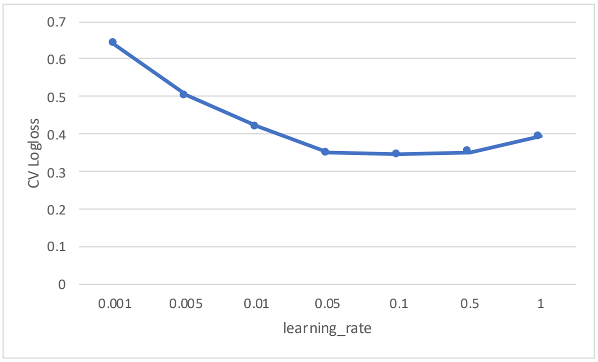


Figure 25: Logloss vs learning_rate

Figure 26 provides yet more evidence of XGBoost's resistance to overfitting. We see a tight alignment between the training and validation curves throughout most of the iterations. This matches almost exactly the curves found in the housing dataset.

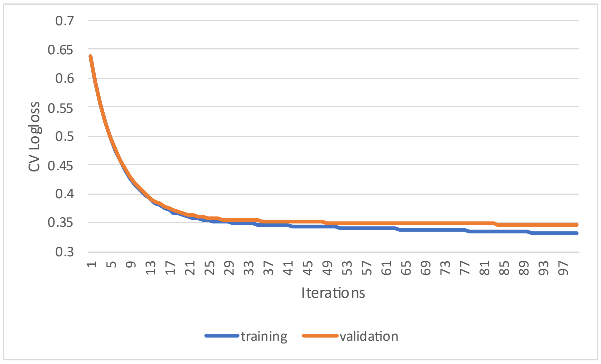
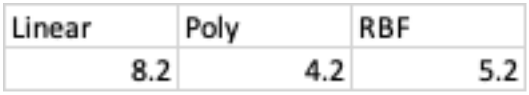


Figure 26: Logloss vs max_depth

3.2.4 Support Vector Machines

A key difference in running SVM on this dataset vs. the housing was the abrupt drop in runtime performance. Since the training datasets were roughly equal in sample size the difference lies in the categorical variables present in the automotive dataset. As we see in **Table 27** SVMs are highly sensitive to the dimensionality of the dataset. There is also substantial differences in the performance of the 3 different kernels. Based on these results we will be using ordinal encoding moving forward.



Linear	Poly	RBF
8.2	4.2	5.2

Table 27: Time (minutes) vs. kernel

3.2.5 k-Nearest Neighbors

As mentioned in the housing data section, knn is significantly impacted by the dimensionality of our dataset. In this dataset we have not only ~10 additional features but we also have high cardinality categoricals. Often these categoricals are encoded in using a technique called One Hot Encoding which creates a binary flag for each value the category takes on in the training dataset, e.g. is_Ford, is_Mazda, etc. for Car Make. Unfortunately this approach dramatically expands the dimensionality even further. In Figure **28** we can see somewhat surprisingly though that KNN, particularly with larger number of neighbors, is able to handle this large dimensionality. Given the high number of dimensions, and possible sparsity of the data, we should explore whether a weighted distance might be preferable. In **Table 29** we can see the results with n_neighbors = 25. Somewhat surprisingly we can no difference between the two types of distance weighting, suggesting that the data may not be as sparse as feared.

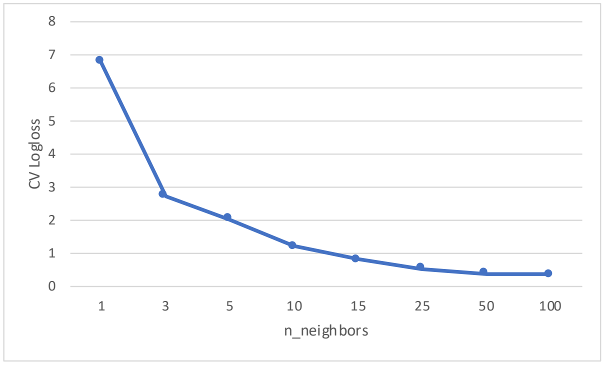
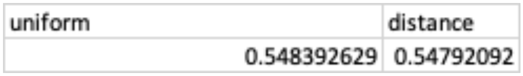


Figure 28: Logloss vs. n_neighbors



uniform	distance
0.548392629	0.54792092

Table 29: Logloss vs. distance weighting

The last area to investigate is the performance over increasing data sample sizes. Looking at **Figure 30** we see extremely surprising results. Neither the training nor validation scores improve with significantly more data. This suggests that the knn score is not in fact learning meaningful patterns in the data and instead may simply be learning an average score for all results.

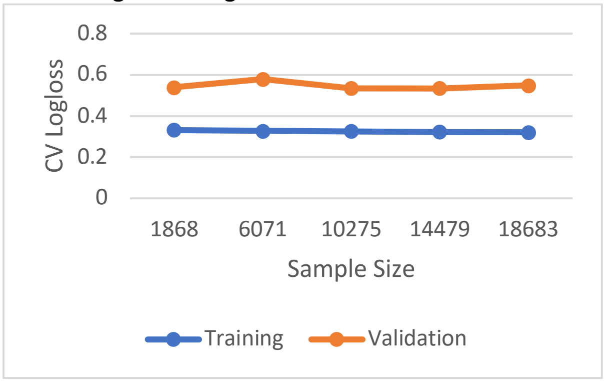


Figure 30: Logloss of training / validation vs. sample size

4. Conclusion

A key takeaway from these experiments is the broad utility and excellent performance (both in in terms of accuracy & runtime) of gradient boosted methods. It was enlightening to see that on both datasets the XGBoost gradient boosted algorithm was able to generate almost identical training & validation curves across sample sizes. This stands in sharp contrast to the decision tree algorithm by itself, which demonstrated significant gaps in accuracy between the training & testing curves across sample sizes.

Another finding during these experiments was the highly variable performance of Support Vector Machine algorithms. On the regression housing dataset the Support Vector Regressor was able to run each training fold in less than one minute. In contrast, generating each fold of the similarly sized automotive dataset took up to 8 minutes depending on the kernel function. In reading through the scikit-learn documentation one possible reason for this discrepancy is that calculating probabilities for data points, which are required for the log loss metric, is computationally expensive.

The k-Nearest Neighbors algorithm also provided several surprises. On the housing dataset the kNN algorithm delivered a surprisingly good performance, beating out the much more complex support vector machine and decision tree algorithms. But on the automotive dataset the knn algorithm displayed the extremely odd behavior of decreasing in accuracy as the sample size of training data increased.