Application of various Machine Learning methods on Price Optimization

Jack Hanlon, Valeria Frolova, Jeremy Kormish
University of Victoria, Canada

Abstract

Price optimization is a critical component of retail sales and is needed to adapt current and new product prices to maximize profits. We experimented with three machine learning algorithms to calculate the prices of similar items based on their condition, category, brand and whether shipping was included. The algorithms were logistic regression, a boosting method, and a neural network. Due to computational constraints our results contained overfitting most likely due to an insufficient amount of data. However we were still able to see that XGBoost performed the best under such conditions and should be used in the event data is lacking.

1 Introduction

When a new product for a store is set to be sold, finding a good initial price point can be a challenge. In real life, there are several dynamic factors that may affect the price. This includes the current state of stock market, price on raw materials or any other real-life situation that affects the final product, one way or the other. Given our problem in particular, we assume these dynamic factors are frozen to let us focus on a more internal set of price-dependent variables, such as item brand, description, quality or shipping. Such a dataset is extracted from a Japanese online shopping app, Mercari, published and retrieved from Kaggle. Except from the price label, most part of our original dataset is qualitative and preprocessing was an integral part of the entire project.
Both external and internal factors are essential for survival of the business, and letting algorithms to keep a sharp watch after any changes is indeed a great opportunity for modern business planning.

2 Related Work

In the paper, “Machine Learning Methods to Perform Pricing Optimization” by Spedicato et al [1], the authors explore machine learning solutions to price optimization. The authors claim that the price optimization modelling scene is “one currently dominated by frameworks based on Generalised Linear Models (GLMs)” and that research using other methods is scarce. This incentivises the paper to experiment using methods that differ from GLMs such as Boosting methods and neural networks. Using these and other methods, the authors found that machine learning models could provide a high degree of accuracy, specifically noting boosted models excellent performance.

In another paper, “Demand prediction and price optimization for semi-luxury supermarket segment” Qu et al [2], the authors apply a machine learning technique to optimize the prices of semi-luxury items. These items are described as items that are expensive and vary in demand. Some factors taken into account in the experiments were price, holidays, discounts, and inventory. These, and other, factors were combined with “A regression tree/random forest-based machine learning algorithm… to predict weekly demand”.

Finally, “Analytics for an Online Retailer: Demand Forecasting and Price Optimization” by Ferreira et al [3] is yet another paper in which the authors used machine learning to optimize the prices of first exposure items of an online retailer. The two main concerns for these items were they either sold out quickly, which suggested a potential price increase could have been made, or they sold less than 25% of their items suggesting the price was too high. Unlike other studies, the retailer implemented the machine learning tool to test the optimizations which led to positive results. The authors found that their solution resulted in a 9.7% increase in revenue for first exposure items which led to the adoption of the tool by the retailer for daily use.

3 Methods of Optimization

XGBoost, Logistic Regression and Neural Networks were chosen to solve our multi-classification problem. While XGBoost is a separate library, Logistic Regression and Neural Networks can be used freely through *sklearn*.
Each of these methods act in a different way. While boosting may take the form of decision trees, Neural Networks recognize patterns through perceptrons and updates at each level. Logistic Regression relies on a binary regression and probability estimates, given some dependent variable. In order to predict or amplify the price, the algorithms are taking into account several different factors, or variables, that may affect the final outcome.

Before these methods could be performed, some preprocessing needed to take place. This began with first taking a subset of the original dataset of 100,000 rows after filtering out rows that did not include a brand entry. This ensured we could get effective use out of any column we desired. Next we removed the price id and item description columns. The id column was not necessary, while the description was deemed to introduce too much complexity. Vectorization of the text columns came next. While attempting to vectorize the data a few things became clear, firstly “,” characters in the text needed to be replaced with ““ for proper data extraction, as well as “/” characters replaced with “ “ to properly vectorize the categories. We also found that vectorizing had a very high memory cost which meant the dataset needed to be cut even further, thus only the first 5000 entries were taken. Even with 1/20th the size, including the item name column produced a significantly larger dataset, than one with the column removed resulting in both being kept in case computational constraints manifested later on with training. Once the data was vectorized, the data was split into a training and a test set using a 80/20 split.

In between these methods, we are using Grid Search where possible, by taking a dictionary of parameters and fitting the algorithm using every unique combination as a configuration. The search’s default value uses 5-fold cross validation and can output various scores corresponding to the performance of configuration. Through these scores the best configuration can be chosen for further experiments.

3.1 XGBoost

'Extreme' or X stands for an enhanced, faster version of Gradient Boosting algorithm along with improved efficiency. Boosting is a technique where an algorithm combines weak learners into a set to make a better prediction on our model. Strictly speaking, it works iteratively by taking outcomes of a previous instance and weighting them – the algorithm learns its own mistakes and combines them to make a final decision.

Generally, boosting technique is an ensemble, or collection of learning algorithms, in the form of a decision tree. This is why one of the parameters of our classifier is *depth,* which is one of the regularization parameters. Another regularization parameter is the number of *estimators*, which represents weak learners. We want to know how big the tree should be, and how many of these trees there better be, in total. However, amplifying gamma parameters and learning rate may also affect our model.

3.1.1 Procedure

The usage of this method is not significantly different from any others contained within the Sklearn library. It is, however, a standalone 'xgboost' module with a different API and set of parameters. We are familiar with some of them, such as gamma-parameter, learning rate, etc. It was decided, however, to set constants for XGB's Classifier and Regressor, such as the tree method being '*gbtree*', number of used processor's cores as *n_jobs=4*.

Firstly, we calculate using XGBoost Classifier. The machine was left turned on for 15 hours with a larger set of parameters to be optimized using Grid Search. Since it takes time, resources and does not report score at each iteration, it was decided to suspend Grid Search at this point and use a reduced number of parameters through looping. Summarised result [Table 1] [Figure 1.1] shows relationships between analysed gamma γ and accuracy score. This option allowed us to form test accuracy results within 3 hours.

eta \ γ	**0.0**	**0.2**	**0.4**	**0.7**	**0.9**	**1.3**
0.1 - 0.7	0.52	0.53	0.58	0.58	0.60	0.60

*Table 1. XGBoost test accuracy with respect to γ and **eta***

Generally, XGB Classifier takes a much larger amount of time for calculating train or test scores. Regressor calculation is almost instant. Now, recall how regularization parameters *estimator* and *depth* play a major role in our optimization. Hence, we picked these parameters and saw which one causes less or no overfitting giving the best desired accuracy [Table 2]. This time, the constants are optimized and default gamma=1.3 and eta=0.3, respectively, were applied. Train accuracy for both Classifier and Regressor was calculated using *5-fold* Cross Validation, while test accuracy was predicted using XGB library and compared to a given test set.

Depth	**1**	**2**	**3**	**4**	**5**
Test	0.67	0.69	0.61	0.63	0.60

Train	0.85	0.80	0.82	0.80	0.81

Table 2. Dependency of depth and accuracy in XGBoost Classifier

Similarly, the data has been computed for XGBoost Regressor, using the same constants gamma=1.3, eta=0.3. The number of core threads has been reduced to 1 due XGB's *predict* limitations.

Estimators	**1**	**2**	**3**	**4**	**5**
Test	0.55	0.63	0.65	0.78	0.85
Train	0.57	0.67	0.72	0.80	0.82

Table 3. Dependency of n_estimators and accuracy in XGBoost Regressor

Further experiment was conducted using XGB's internal method, CV, with the same γ and *eta* as before. The greatest advantage of using *xgb.cv* is that we are able to plot RMSLE at each of these iterations [Figure 1.4], with corresponding train and test RMSLE mean and standard error, *std*. The amount of time we spend training boosted models depends on the number of iterations in XGB, and the machine it is being run on. Fortunately, with *num_boost_round=1800* the machine provided output within several minutes. Entire dataset from the *xgb.cv* output has been converted and refined in Excel to display plots properly. [Figure 1.4].

3.1.2 Analysis

Evidently, *learning rate* had no effect on accuracy in our case, with gamma, estimators and tree depth causing more impact. This is why learning rate axis was excluded from the plot [Figure 1.1]. XGB Classifier showed best test results at *tree depth=2,* and no overfitting was observed. Given relative test and train accuracy, it is visible how *tree depth=2* is the best parameter choice. [Figure 1.2][Table 2].

XGB Regressor has a linear relationship with the number of estimators, with more estimators providing better accuracy. However, there is clear overfitting since test accuracy falls beyond training score boundaries. We might conclude our *number of estimators*, being a discrete value, should not exceed 4. [Figure 1.3] [Table 3].

Finally, our 5-fold cross-validation under Cross Validation RMSLE metrics shows error fluctuations. There are two groups: train and test standard errors, and their corresponding RMSLE. Clearly there is overfitting reported by RMSLE and test error keeps increasing with iterations, while train error slowly decaying. Standard Error (*std*) shows familiar patterns and no report on overfitting. It is the case of an 'elbow method' applied on the graph: the plot shows our best choice in number of iterations could be around 200. Due to overfitting, we cannot choose larger values, yet too small may also be erroneous.

3.2 Logistic Regression ML

This method is commonly used for problems relating to finance, such as Insurance, Economics and of course Price Optimization. Logistic regression uses a sigmoid function to fit a 0 or 1 response to an S- curve, allowing for outliers to not fall under or above the line as is the case

for Linear Regression. In this case, because there are multiple features a multi-class implementation is used.

3.2.1 Procedure

Scikit learn is a free to use machine learning library for python that was used for the implementation of Logistic Regression.

Firstly, a basic Logistic Regression implementation was run on the dataset. The regularization parameter was varied exponentially in both directions from the default value of 1. This was done to see the effects of L2 regularization on overfitting to increase accuracy on the test set.

Reg. Param.	**16**	**8**	**4**	**2**	**1**	**0.5**	**0.2**	**0.1**	**0.04**	**0.08**
Test	0.55	0.56	0.57	0.57	0.57	0.58	0.58	0.59	0.59	0.57
Train	1.0	0.99	0.99	0.99	0.96	0.86	0.70	0.66	0.64	0.62

Table 4. Variation of Regularization Parameter and accuracy score

Next, Grid Search cv was performed to optimize all hyperparameters of the regression. This operation had a much longer runtime than the basic testing above, resulting in an hour long training time with a Nvidia GPU. The accuracy on the training data found through this method was 0.59, which is the same as the regularization above. This shows that there may be a flaw in the model being used for learning for this problem, as attempting Grid Search did not improve much upon basic testing.

Lastly, Logistic Regression was run again but this time with a Balanced weight association for each feature as there is considerably more data in the vectorized features than others.

Reg. Param.	**16**	**8**	**4**	**2**	**1**	**0.5**	**0.2**	**0.1**	**0.04**	**0.08**
Test	0.50	0.47	0.42	0.35	0.26	0.17	0.08	0.03	0.01	0.01
Train	0.99	0.96	0.90	0.79	0.61	0.44	0.237	0.15	0.09	0.07

There was a negative correlation between decreasing regularization and balancing the features.

3.2.2 Analysis

It is apparent from the results that this dataset is difficult to classify, despite the simplifications made to the Kaggle data during pre-processing. Experimentally, the concept of balancing feature weights was not effective at increasing accuracy on the test set. The common route to improving accuracy of Logistic Regression ML is:

1. Feature Scaling and/or Normalization
2. Class Imbalance
3. Log-loss optimization
4. Hyperparameter tuning (Grid Search)

Feature scaling was implemented in the regularization c parameter test, and the Grid Search. The best accuracies were found for a l2 regularization of parameter range of (0.04 - 0.1) and from the Grid Search. There was clear overfitting in the balanced method as there was significant improvement in accuracy when regularization was increased. As the accuracy only reached 0.5 the classifier is analogous to a coin flip and is thus not an effective method for classifying this dataset. In the first method there is also evidence of over-regularization in which as the c parameter increases the accuracy starts to get better and then worse.

3.3 Neural Network ML

As stated in [1] "Neural networks are generally used for detecting recurring patterns and regularities" which seems to be a good fit for price optimization as utilizing sales patterns for similar products is the main area of concern for our experiments. Neural networks consist of an input layer, one or more hidden layers and output layers. These layers are composed of neurons (i.e. nodes) in which calculations are performed. The calculations consist of weights and an activation function which together determine if and by how much a given signal should affect the overall prediction. Neural networks can be used for both classification and regression problems and have several hyper parameters that can significantly affect the algorithms outcome. Some of these parameters are as follows [1]: number of *hidden layers* - the more, the deeper; *activation function* - decides if neurons output should be "activated" (i.e. is it is useful); *size* of hidden layers - number of neurons within each hidden layer; *regularization parameter* - generally used to prevent overfitting; *solver* - used for weight optimization; and maximum *number of iterations* - passes of the training set.

Of these parameters the last four were chosen to be experimented with, using default values assigned to the rest. Even though the reference article used suggested a deep learning (more than one hidden layer) approach, it was decided due to time and computational limitations that the use of a single layer would be conducted in all experiments.

Procedure 3.3.1

The initial experiments for testing the neural net's effectiveness on the preprocessed data, consisted of using a grid search function (GridSearchCV) from the sklearn library, in order to test a range of different parameters. Furthermore the classification and regression techniques were also performed using the sklearn library, utilizing the MLPClassifier and MLPRegressor functions as the neural network models.

The parameters used in the grid search consisted of the number of nodes in the hidden layer, the solver used by the algorithm, and the regularization (alpha). These values were limited due to computational constraints as a first attempt at a grid search (approximately 3800 total

models would have been trained) was estimated to have taken over two days to have finished, which for a project of this scope is not viable. In this method, Grid Search has been applied with default preset *cv=5*. Along with the Grid Search a comparison of classification neural nets vs. regression neural nets was performed at the same time using the same parameter configurations.

Once the initial hyper parameter tuning with the grid search was complete, further tuning was done by trying different max iteration parameter values. Finally a sanity check was performed for the number of hidden layer nodes as there were multiple configurations with close scores originally and the change in max iterations may have had an effect on the optimal number of nodes. Also some more precise numbers were tested to ensure optimality of the number of nodes.

Once the final parameter configuration was determined, the neural net was run 25 times to analyze the consistency and values of the training and test root mean squared logarithmic error (RMSLE) to provide comparison results for the other algorithm experiments. The RMSLE was computed by once again using the sklearn library, by taking the square root of the mean_squared_log_error function. Finally a last experiment was run using the final parameter configuration to plot the predicted values vs. the actual values to show the differences in the value ranges.

3.3.2 Analysis

As stated previously computational constraints played a large factor in the generation of experiments. This became apparent early on in the preprocessing stage where the training and test data used had to be substantially cut to allow for reasonable work to be done. This was further realized when attempting a grid search with what seemed to be a reasonable range of parameters which, after running the experiment for 3 hours, resulted in the calculation of an estimated finish time of more than two days. This first incomplete experiment included tests for different learning rates, activation functions, solvers, regularization, and size of hidden layers. The revised search utilized a grid only consisting of two solvers, a range of three regularization parameters, and a range of three hidden layer sizes. This reduced the overall number of model fitting to 90, from roughly 3800 and allowed for optimization to be completed within roughly one and a half hours [Figures 3.1, 3.2, 3.3, 3.4].

It is important to note that both methods' best configurations for using the *lbfgs* solver performed worse than either of *adam* solver. Furthermore it can generally be seen that a lower number of hidden layer neurons provided a higher score as well as the alpha value of 0.1. While the values of 0.1 and 1.0e-7 were almost identical for the *adam* regression experiments, the value of 0.1 was chosen as it was more consistent throughout other experiments and more tuning was to be performed.

Next the maximum iterations tuning can be observed [Figure 3.5], which shows that as the max iterations increased the model began to over fit. Also the number of negative values in the test predictions was plotted as they needed to be removed in order to perform a RMSLE calculation and intuitively a higher number of negative values implies a higher error (unless the algorithm could predict desire to get rid of something accurately which is highly unlikely for this project).

Following the max iterations data, the sanity check for the hidden layer size seen below [Figure 3.6], supported the previous observation that significant increases in size tended to

perform worse. After this, the final tuning of the hidden layer size can be observed [Figure 3.7]. It was determined that due to the fluctuations seen around size of 10-17 that a value greater than that should be chosen. Furthermore, while score did marginally improve with small increases so did the number of negative numbers, thus 25 was chosen as it was a reasonable ratio of size vs. negative values and was already used in previous tests.

The results for the experiment to be compared with the alternative algorithms using RMSLE can be seen further below [Figure 3.8, 3.9]. Figure 3.8 shows the RMSLE error for the test and training data where some fluctuations between repeated experiments can be seen, however the results for test error all fall within the range of 0.775 - 0.825. Notably the training error was consistently better hinting that there still may have been some overfitting happening which could be caused by some ignored parameters or issues within the dataset. Finally Figure 3.9 shows a plot of a single test with the same parameters which displays graphs of the distribution of the predicted values and the actual values (left two and right two graphs respectively) for the training data (top two graphs) and the test data (bottom two graphs). These graphs show that the model could not predict values for the high sale items most likely due to an insufficient amount of data. A more accurate model for the other items may have been generated if these high sale items had been removed in the preprocessing step. Overall the RMSLE achieved was better than anticipated even after considering the added error from removal of the negative values

4 Score Analysis, Combined

When the three methods and their sub-optimizations are compared, all three methodologies classify to a non-trivial accuracy on the dataset. The optimal accuracies from the RMSLE error for XGBoost, Logistic Regression and Neural Nets were 0.85, 0.59 and 0.82 respectively. There were significant sources of error due to the pre-processing and reduction of dataset size.

XGBoost	Logistic Regression	Neural Networks
0.85	0.59	0.82

5 Conclusion

To conclude, XGBoost was the most effective method of classifying the data and effectively optimizing price for the products on the Mercari App Platform. XGBoost is good for tabular data with a small number of variables, whereas neural nets are good for images or data with larger numbers of variables. In this case, XGBoost works well with the nature of product data. From a business perspective, the company, Mercari, could utilize XGBoost to create an automatic price optimization system for their software that would be accurate approximately 85 % of the time. This would be highly beneficial as with large traffic through the app it would be next to impossible to manually observe and price those products effectively.

On a final note, it is relevant to note the computational power in our personal computers was not efficient enough to compute the full dataset as with even a reduced dataset size, the neural

net had an estimated training time of 48 hours. It would be interesting to experiment with the full dataset and a longer training time in the future.

6 References

[1] Spedicato, Giorgio & Dutang, Christophe & Petrini, Leonardo. (2018). Machine Learning Methods to Perform Pricing Optimization. A Comparison with Standard GLMs. 12. 69-89.

[2] Qu, Tianliang & Zhang, Jianghua & Chan, Felix & Srivastava, R.S. & Tiwari, Manoj & Park, Woo Yong. (2017). Demand Prediction and Price Optimization for Semi-Luxury Supermarket Segment. Computers & Industrial Engineering. 113. 10.1016/j.cie.2017.09.004.

[3] Kris Johnson Ferreira, Bin Hong Alex Lee, and David Simchi-Levi (2016). Analytics for an Online Retailer: Demand Forecasting and Price Optimization. Manufacturing & Service Operations Management 18:1, 69-88

Appendix

XGBoost

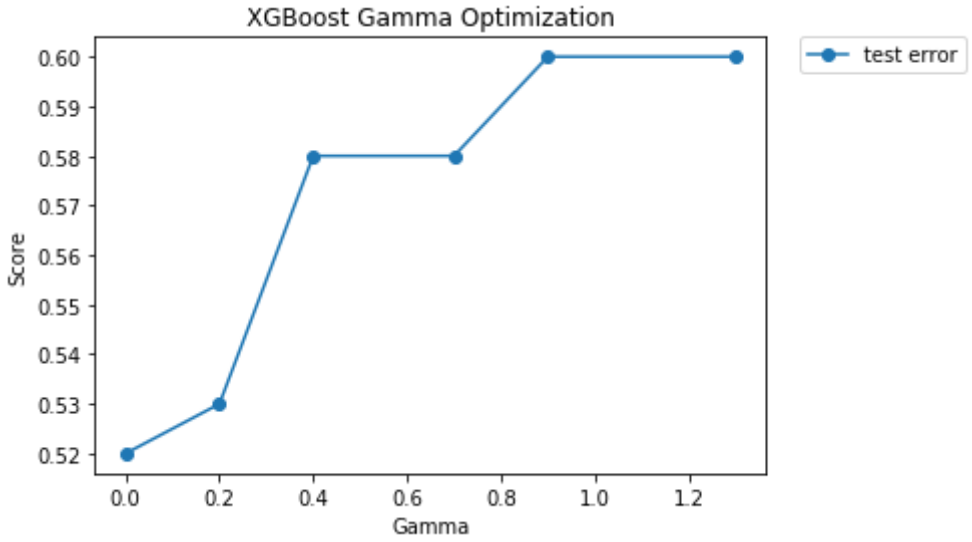

Figure 1.1 Observing test accuracy w.r.t. Gamma parameter

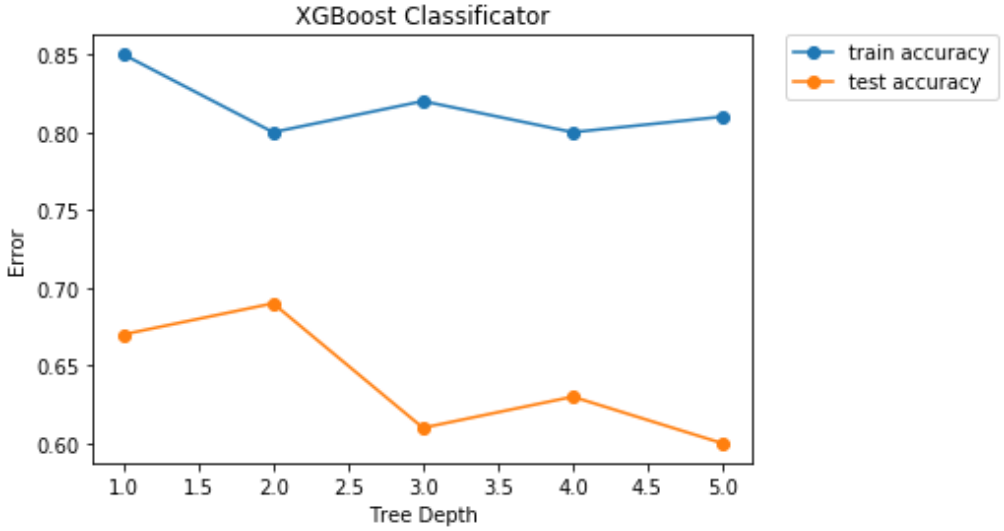

Figure 1.2 XGBoost Classificator accuracy w.r.t. Tree Depth

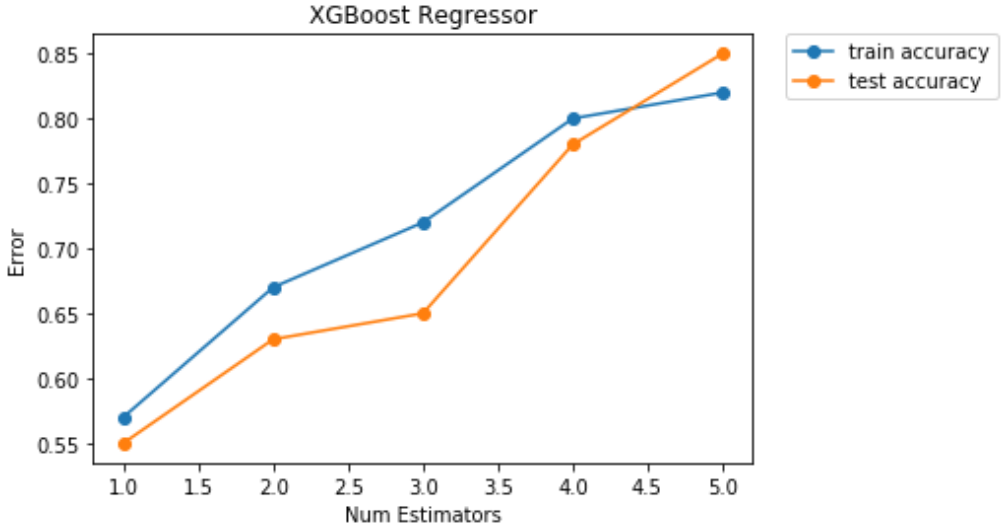

Figure 1.3 XGBoost Regressor accuracy w.r.t. Number of Estimators

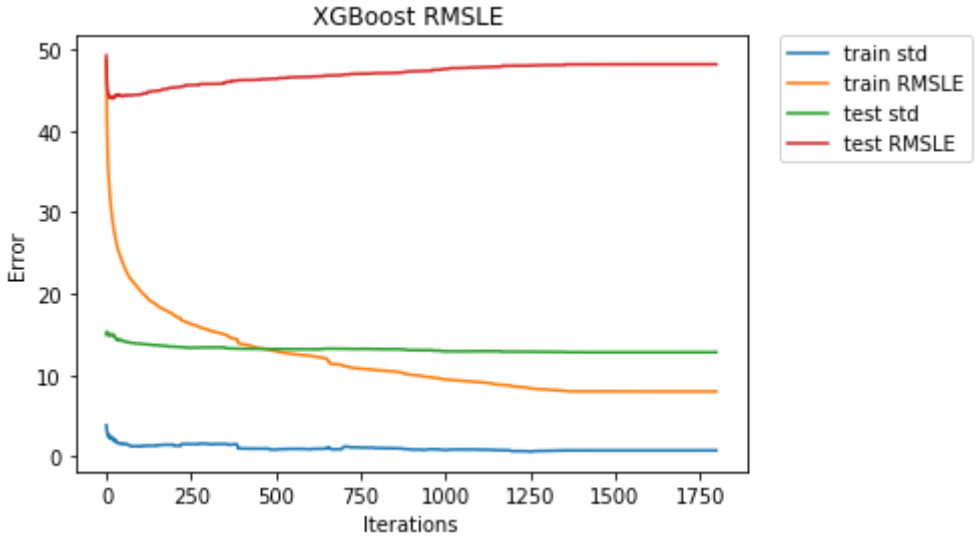

Figure 1.3 XGBoost Root MSE Cross-Validated accuracy w.r.t. Number of Iterations

Logistic Regression

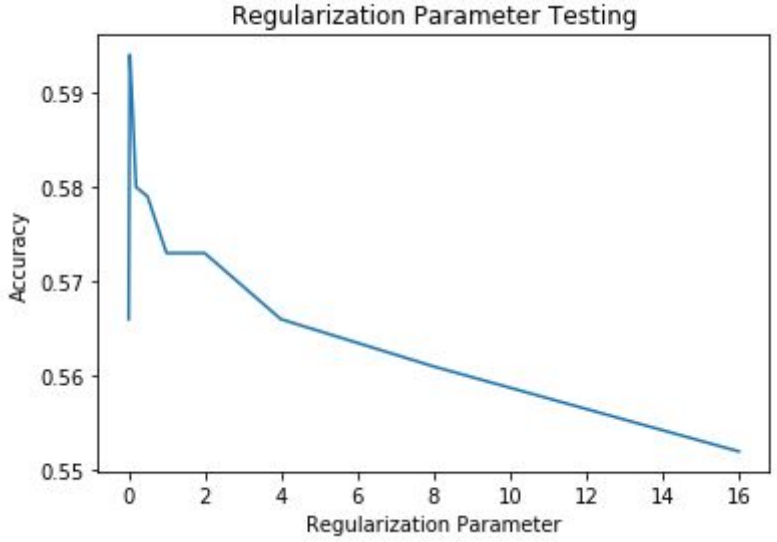

Figure 2.1: Logistic Regression accuracy vs regularization parameter

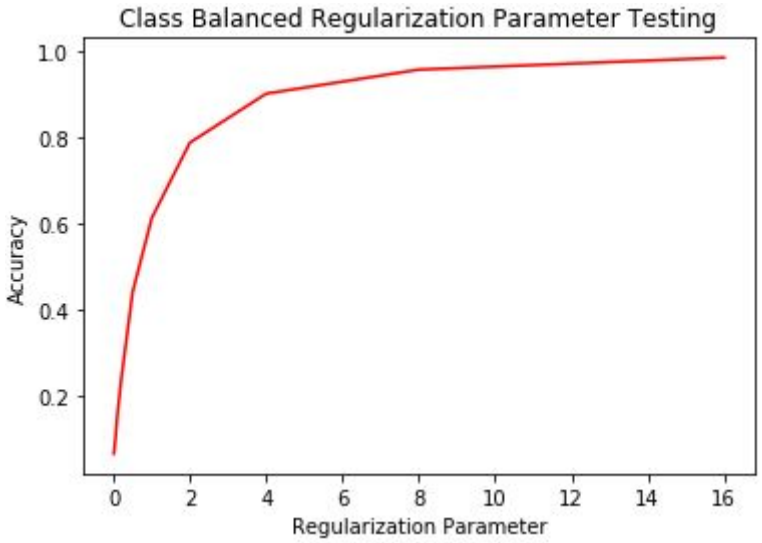

Figure 2.2: Class Balanced Logistic Regression accuracy vs regularization parameter

Neural Networks

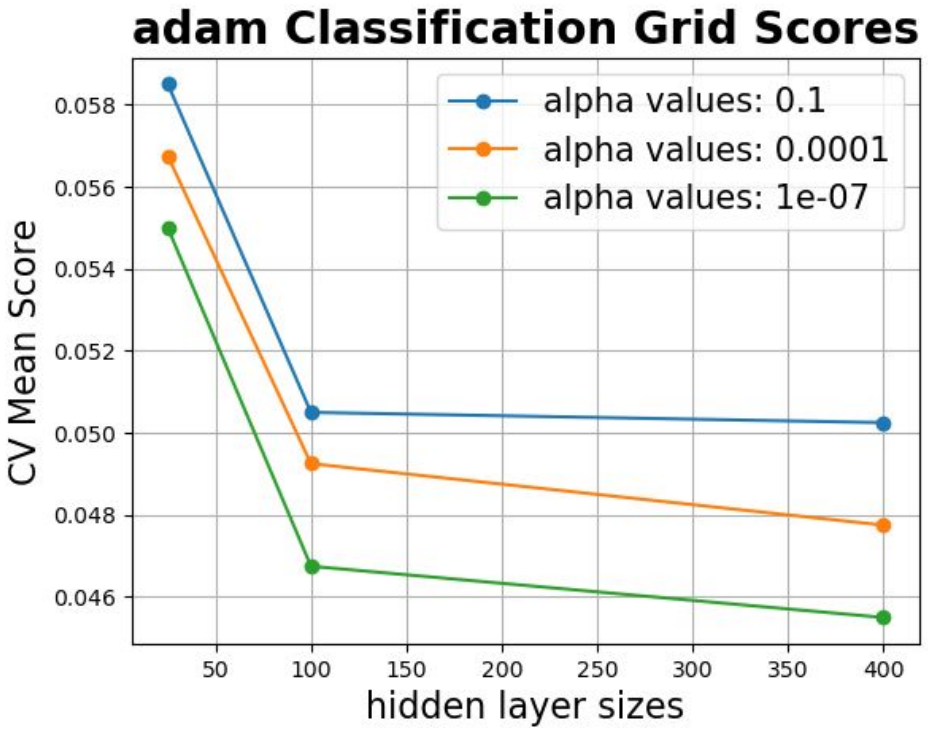

Figure 3.1: Classification neural network grid search results for adam solver

Figure 3.2: Regression neural network grid search results for adam solver

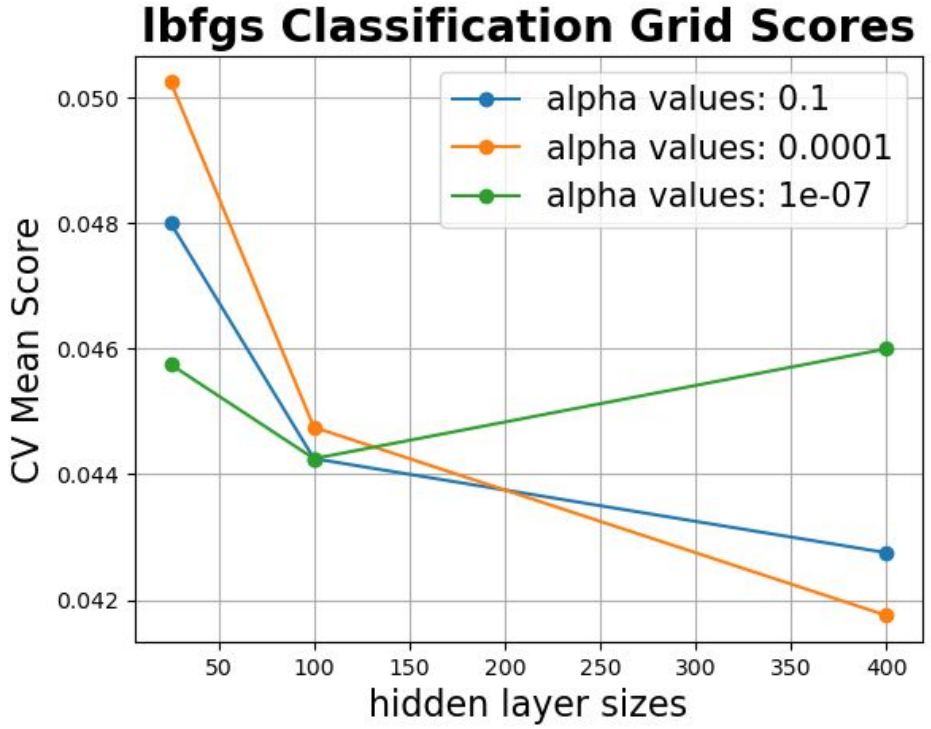

Figure 3.3: Classification neural network grid search results for lbfgs solver

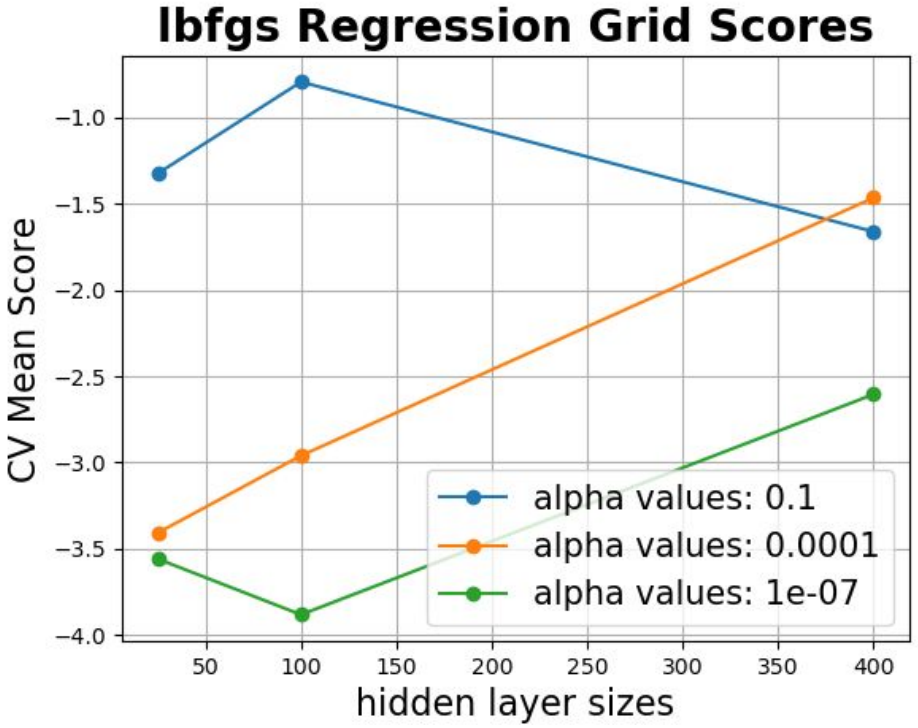

Figure 3.4: Regression neural network grid search results for lbfgs solver

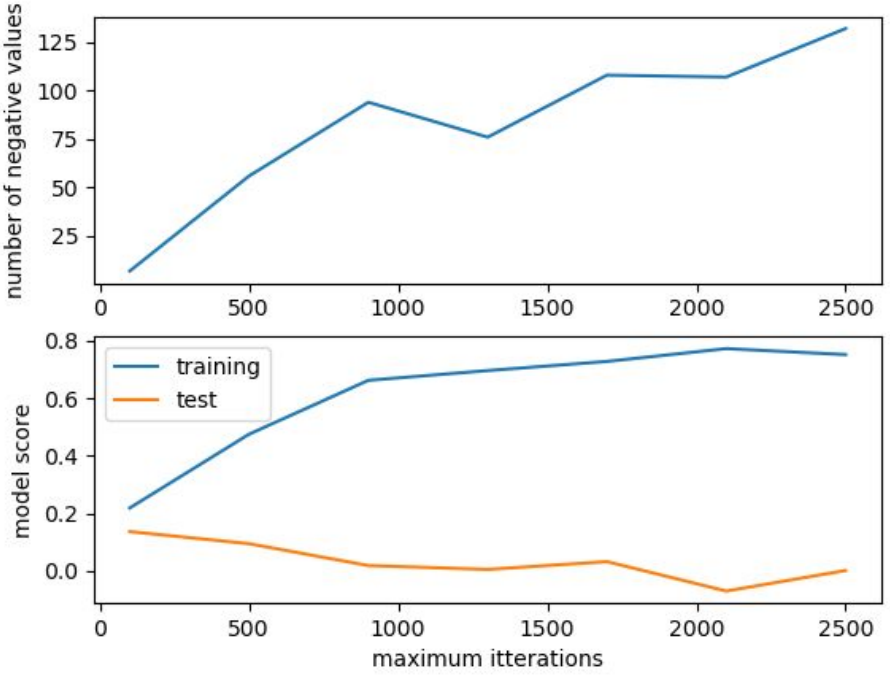

Figure 3.5: Regression neural network max iteration tuning using alpha = 0.1, hidden layer size = 25, and adam solver

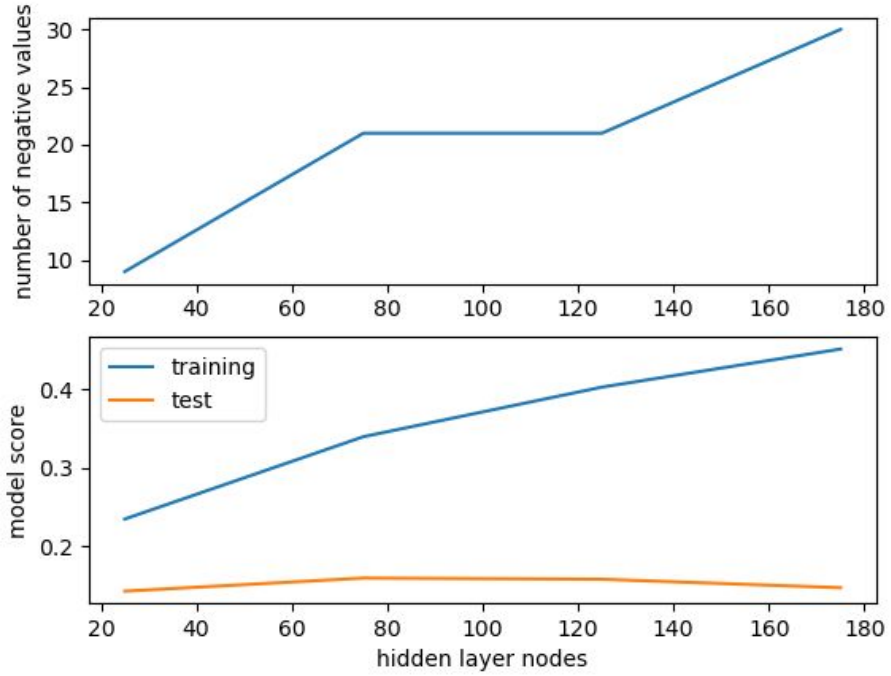

Figure 3.6: Regression neural network hidden layer size sanity check using alpha = 0.1, max iterations = 100, and adam solver

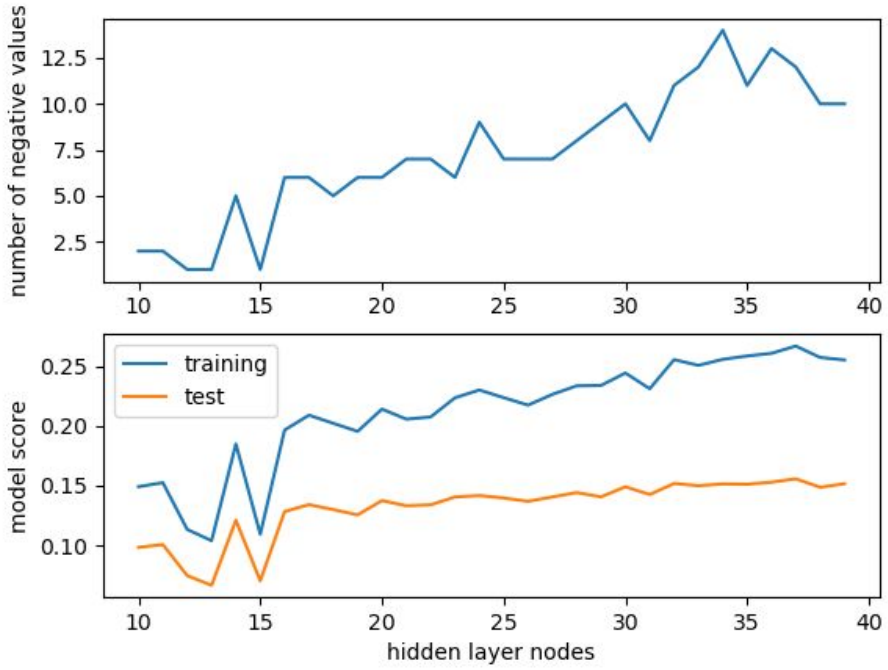

Figure 3.7: Regression neural network hidden layer size fine tuning using alpha = 0.1, max iterations = 100, and adam solver

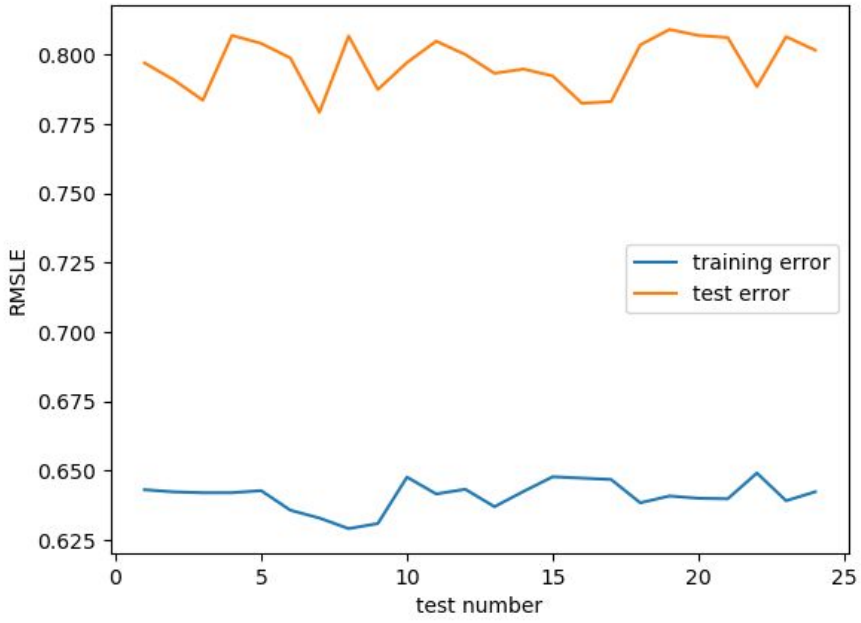

Figure 3.8: Regression neural network root mean squared logarithmic error test using alpha = 0.1, hidden layer size = 25, max iterations = 100, and adam solver

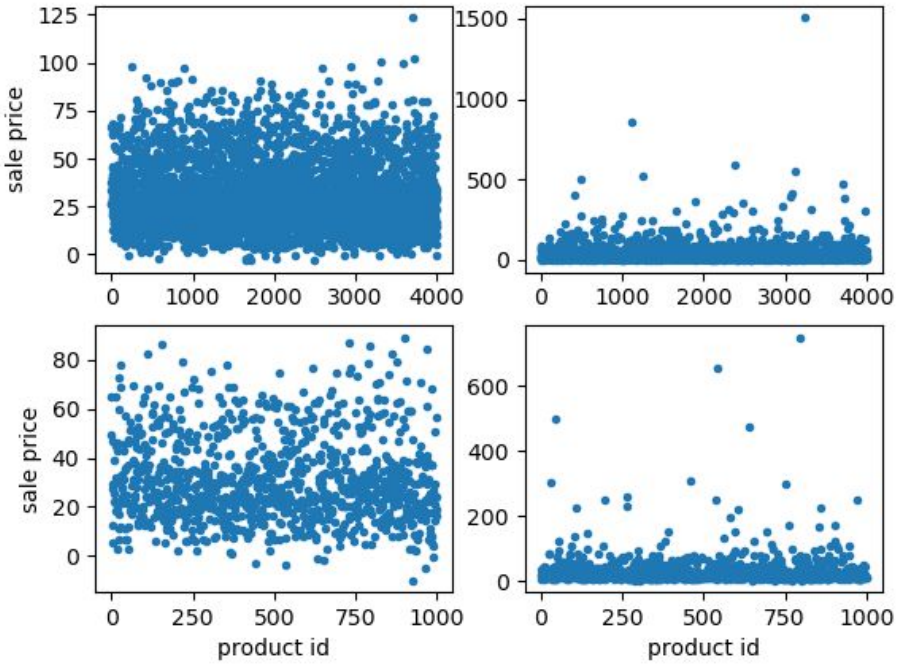

Figure 3.9: Regression neural network scatter plot of predicted and actual sale prices using alpha = 0.1, max iterations = 100, and adam solver. Top left = predicted training, top right = actual training, bottom left = predicted test, bottom right = actual test.