Predicting House Prices Through Advanced Regression Techniques Based on the Kaggle House Prices Competition

Kelsey Sterner
kns56028@uga.edu

Quentin Proels
qap74567@uga.edu

Natalie Mayer
nrm20749@uga.edu

Jan Schiller
jan.schiller@uga.edu

School of Computing at the University of Georgia

Abstract

The housing market is a complex and continually changing system based on various factors that can be hard to predict or measure (e.g., economic stability, supply/demand, shifting aesthetics, location, etc.). Accurately predicting housing prices is a crucial tool for buyers, sellers, policy makers, and real estate agents seeking to make informed decisions. The dataset we are using to apply and test different methods is a set containing instances of houses sold in Ames, Iowa. Many different methods have been historically applied and tested in this type of regression problem. We will be focusing on simple models and their efficacy as a baseline versus more sophisticated methods such as neural networks, forests, and boosting. We report the results of multiple experiments conducted on our dataset and how each led to the next in our iterative process to improve our margin of error.

1 Introduction

There is a massive store of data collected and displayed regarding the housing market in almost every satellite accessible area due to sites like Zillow, Realtor, Redfin, Trulia, etc. that display various details for every property listed. This gives plenty of data and potential features for analysts and the above-mentioned interested parties to select for data mining purposes to tackle the house price regression problem. Most of the problem arguably comes from the sheer number of features. Feature engineering is a key component to minimizing errors and selecting the most informative attributes to train on. These features are not universally applicable to all geographical areas.

In our experiments, we will be focusing on minimizing error for the specific location of the dataset, Ames, Iowa, while keeping in mind how our feature engineering techniques and encoding can be generalized to other similar datasets.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

2 Examining the Data

Since our problem is proposed through a Kaggle competition, we are examining the datasets provided through Kaggle. This consists of a Train.csv file (79 attributes including target and 1460 instances), a Test.csv (78 attributes and 1459 instances), a Data_description.txt (to describe attributes), and a Sample_submission.csv (we won't be looking at this much for our techniques).
The general statistics of our dataset is as follows:
Mean — $180,921.20
STD — $79442.50
Q1 (25%) — $129,975
Q2 (50%) — $163,000
Q3 (75%) — $214,000
Max — $755,000

Looking at our general statistics, we quickly notice that our mean is noticeably higher than our Q2 middle of the range value. This is an early indicator of significant outliers that we will explore in our first iterations of our experiments.

3 Preparing the Dataset

In our first iteration of feature selection, our goal was to simplify the dataset by reducing the number of features while preserving those most relevant to predicting housing prices. The dataset contains 79 features, but not all are equally useful for predicting housing

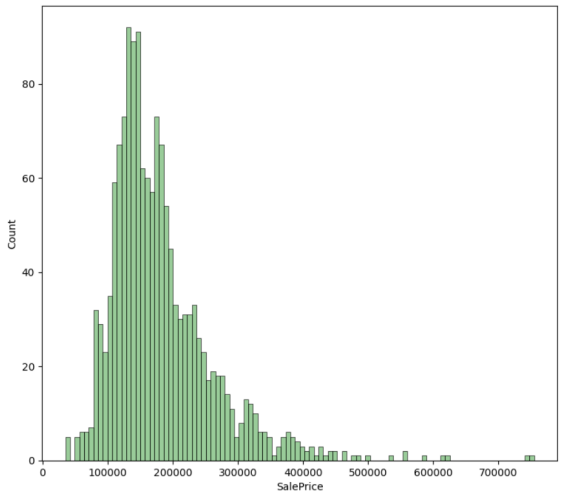

Figure 1: Data Distribution

prices. Some add noise or overlap with others, which can reduce the overall performance of the model.

In this section, we describe our preprocessing steps, how we handled missing values and categorical data, and the techniques we used for selecting informative features.

3.1 Preprocessing and Cleaning

Before selecting features, we first cleaned and prepared the data to ensure it was ready for modeling. This included handling missing values and encoding categorical variables. These steps make sure the data is in a format that learning models can understand, especially the models that need all input to be numeric and don't handle missing values well.

3.1.1 Missing Values

To address missing values, we chose to retain categorical features with sparse data by filling their missing entries with the placeholder value "None". This allowed us to preserve potentially useful information without dropping entire columns.

For numerical features like "MasVnrArea" and "GarageYrBlt", missing values were filled with 0, while "LotFrontage" was imputed using the median to reduce the influence of outliers. Finally, the single missing value in "Electrical" was filled using the mode to maintain its distribution.

3.1.2 Encoding Categorical Features

Before converting the categorical data into numeric values, we made two copies of the dataset. One was designated for tree-based models, where we used label encoding, and the other for linear models, where we applied one-hot encoding.

3.1.3 Label Encoding

For tree-based models like decision trees, random forests, and gradient boosting, we used label encoding to convert variables into integer values. This method assigns a unique numeric code to each category in a column. Tree-based models handle this type of encoding well because they make splits based on category values rather than relying on any assumed order in the values.

3.1.4 One-Hot Encoding

For linear-based models such as linear regression, logistic regression, and SVMs, we applied One-Hot Encoding. This method turns each categorical variable into a group of binary columns, one for every unique category. It helps avoid implying any kind of order among categories that don't actually have one. While this approach can increase the number of features significantly, it tends to perform better with models that are sensitive to the way categorical values are encoded.

3.2 Feature Selection

After encoding and cleaning the dataset, the next step was selecting the most relevant features to include in our models. With 79 features available, it was important to identify and retain only those that had the most predictive value for housing prices. This helps reduce noise, improve training speed, and increase generalization to unseen data. We explored multiple strategies to ensure a well-rounded selection process.

3.2.1 Correlation Matrix and Heatmap

To start, we examined the relationships between numerical features using a correlation matrix. Features that are strongly correlated (with correlation coefficients above 0.8 or below -0.8) can introduce redundancy and multicollinearity, especially in linear models, which can hurt performance and interpretability. We used a heatmap to visualize these relationships and flagged highly correlated feature pairs.

In particular, when two features were strongly correlated with each other but not necessarily with the target variable (SalePrice), we considered removing one of them. However, features that showed a strong correlation with SalePrice were retained or given priority. This step acted as an initial filter before applying more targeted selection techniques.

3.2.2 Stepwise Feature Selection

Next, we applied forward stepwise selection using SequentialFeatureSelector from scikit-learn. This method starts with no features and incrementally adds one feature at a time, choosing the one that improves

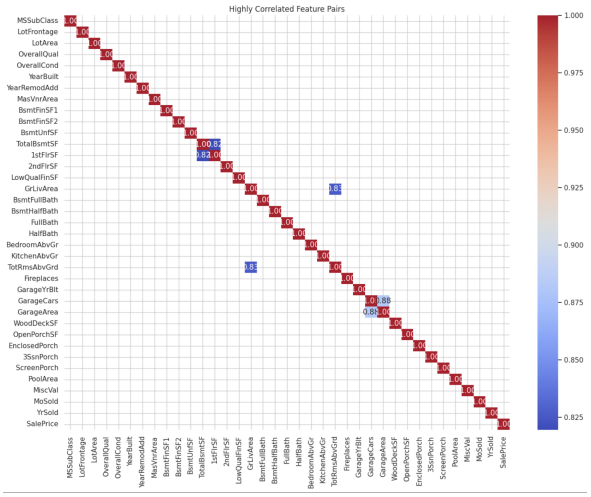

Figure 2: Heatmap

model performance the most at each step. We used a linear regression model to evaluate the predictive power of each feature and selected the top 20 based on this iterative process.

3.2.3 Feature Importance From Random Forest

In addition to the stepwise selection, we also evaluated feature importance using a Random Forest Regressor. This tree-based method assesses how much each feature helps reduce prediction error across an ensemble of decision trees. Since decision trees are great at capturing non-linear relationships and interactions between features, this method often identifies important variables that might not stand out as much in linear models.

After comparing the results from both methods, we combined the selected features to form a final set of predictors. This mixed approach made sure that the final feature list captured both statistical importance and what worked best for the model. We saved the resulting dataset and used it to train our final tree-based models.

4 Simple Model Examination

The first experiments we conducted were seeing how effective a basic Linear Regression Model and basic Support Vector Regression Model would perform as a baseline.

4.1 Linear Regression Model

For the implementation of our linear regression model, we use the sklearn package and measure the error through RMSE, MSE, and R-squared, focusing primarily on RMSE.

After fitting our model to a randomly selected 80% training, 20% testing standard split (1026 instances used in training, 434 instances used in testing), the model performance can be seen in Figure 2 and statistics in Table 1.

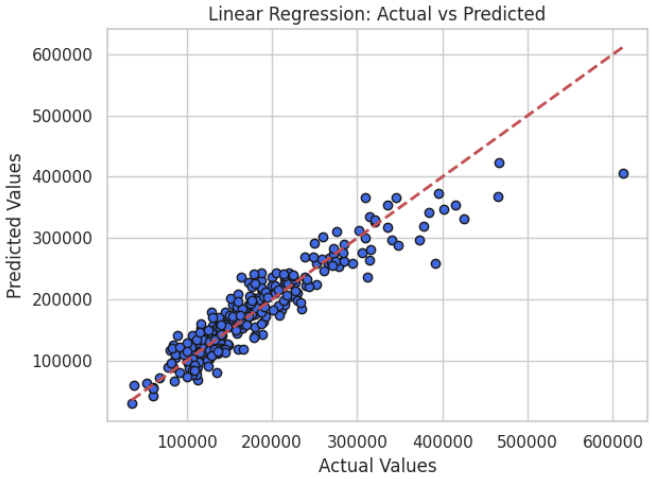

Figure 3: Linear Regression Model

Table 1: Linear Regression Statistics

MSE	880446450.8044447
RMSE	29672.317921
R-squared	0.8623652268232884

The results of our linear regression model yielded around $29,672.32 as our RMSE, with some clear visual influence as the values of the actual instances increase above $300,000. This makes sense logically because there is generally a cap around what a "high end" or expensive house could be, especially somewhere more rural like Ames, Iowa. This makes a linear regression model less ideal since historical housing market trends and house prices do not only move linearly.

4.2 Support Vector Regression

Our next experiment uses a support vector regression model. We chose SVR because it is more robust in terms of addressing outliers. Additionally, SVR incorporates kernels, which can improve performance for non-linear relationships.

For our implementation we used sklearn's SVR model with a StandardScalar package since SVR can be sensitive to feature scaling. We tried 3 different kernels: linear, poly, and rbf. RBF performed the best which is the model shown in Figure 3 that yielded the statistics in Table 2.

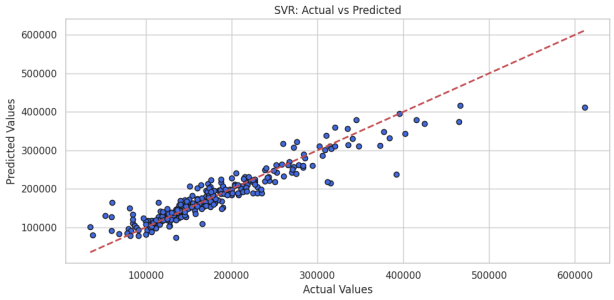

Figure 4: SVR Model

Table 2: SVR Statistics

MSE	793656608.9970845
RMSE	28171.9117029
R-squared	0.8759325484704875

We can see from the table that there is a marginal improvement to the RMSE from $29672.32 to $28171.91. However, the improvement is small compared to the change in computational complexity from a linear regression model to support vector regression model.

5 Tree-based Models

Tree-based algorithms are commonly used for classification but also for regression tasks. Supervised machine learning models construct decision trees to recursively partition the feature space into regions, enabling hierarchical representation of the input variables. Each internal node represents a decision for a feature and each leaf node represents a prediction. Their ability to handle non-linear relationships and handle different data types make them effective for our housing price prediction [Ban20]. To ensure a consistent comparison across the different tree-based models, the dataset was split into training and test sets using an 80/20 split with train_test_split with a fixed random_state. This split is applied to both versions of the dataset: one with the all features and another with only the selected features (see Section 5). In the following, we successively compare tree-based methods including Decision Trees, Random Forests, Gradient Boosting and XGBoost. Starting with the simplest model and then introducing bagging and boosting techniques. All implementations are carried out using the scikit-learn library, except for XGBoost, which is implemented using the xgboost package.

5.1 Decision Tree - CART Algorithm

The Classification and Regression Tree (CART) algorithm is used for both classification and regression tasks. An advantage is that a generated tree is interpretable and can capture non-linear patterns. At each node the algorithm selects the feature and threshold that minimizes the root mean squared error (RMSE). Each leaf node holds a constant prediction value showing the average target value. In Figure 5 the scatter plot compares the actual versus predicted house prices for the decision tree model using all features (color blue) and selected features (color orange). Both versions tend to under-predict higher prices but also show that feature selection is better than using all features. With all features the model achieved an RMSE of 45159.90 and R^2 of 0.734, whereas after applying feature selection the RMSE improved to 44033.18 and R^2 increased to 0.747.

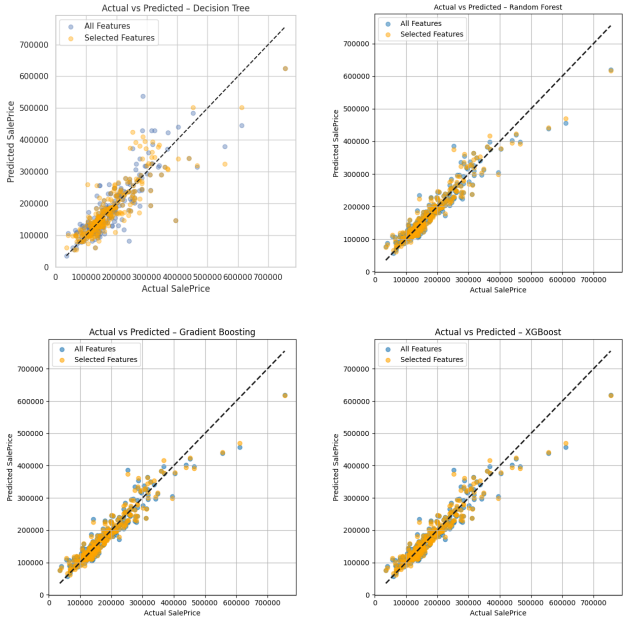

Figure 5: Comparison of Different Tree-Models

5.2 Random Forest

Random Forest is an ensemble learning method that combines multiple decision trees using bootstrap aggregation (bagging). Every tree is trained on a randomly sampled subset of the data and a random subset of features. To get the final prediction the algorithm averages the output of all individual trees, which reduces variance and improves robustness, while being less prone to overfitting than CART.

Comparing the scatter plots Random Forest produces more accurate and stable predictions across the full price range than the CART Model. Including all features the model achieved an RMSE of 28968.07 and

R^2 of 0.891 and RMSE of 28474.28 and R^2 of 0.894 with selected features, showing the positive impact of feature selection in this ensemble method.

5.3 Gradient Boosting Regressor

Gradient Boosting is an ensemble method that builds decision trees sequentially, where each new tree is trained on the residual errors made by the previous ensemble. Compared to bagging it reduces bias by focusing on difficult to predict instances. Using all features the model reached an RMSE of 28979.51 and R^2 of 0.891, whereas after applying feature selection the RMSE improved to 26822.27 and R^2 to 0.906, showing that it benefits from a reduced feature set.

5.4 XGBoost

Extreme Gradient Boosting (XGBoost) is a sophisticated variant of gradient boosting that includes features such as L1/L2 regularization, parallel processing and handling of missing values. These features make it faster and more robust. A similiar pattern can be seen here, as using feature selection the RMSE with 26108.69 and R^2 with 0.911 improved compared to using all features (RMSE: 26757.87, R^2: 0.907).

5.5 Comparison

Comparing the results of all four tree-based models (see Table 3) shows a clear improvement in performance as model complexity increases. The CART model showed the weakest performance. Ensemble methods like Random Forest and Gradient Boosting reduce prediction error and achieve similar R^2 values of 0.891. XGBoost achieved the best overall results with an RSME of 26,108.69 and R^2 of 0.911 after applying the feature selection. Using the introduced feature selection algorithm shows across all four models a consistent improvement confirming that reducing irrelevant and redundant input variables can help improve generalization and overfitting.

Model	**RSME**	R^2
CART (All Features)	45159.90	0.734
CART (Selected Features)	44033.18	0.747
Random Forest (All)	28968.07	0.891
Random Forest (Selected)	28474.28	0.894
GBT (All)	28979.51	0.891
GBT (Selected)	26822.27	0.906
XGBoost (All)	26757.87	0.907
XGBoost (Selected)	26108.69	0.911

Table 3: Tree-Models performance for all and selected features.

5.6 XGBoost - Hyperparameter Tuning

To improve the performance of the XGBoost Model hyperparameter tuning is applied using a RandomizedSearchCV and 10-fold cross-validation. In the first stage a broad parameter grid is defined including ranges of the number of estimators (n_estimators), learning rate (learning_rate), tree depth (max_depth), and regularization terms (reg_alpha, reg_lambda), as well as sub sampling and feature sampling ratios. We iteratively adjusted and narrowed the subsequent stages towards the most promising combinations. The best result was obtained using the following parameters:

```
n_estimators: 1000
learning_rate: 0.02
max_depth: 4
subsample: 0.7
colsample_bytree: 0.5
gamma: 0.5
reg_alpha: 0
reg_lambda: 0.5
```

With the tuned hyperparameter the model achieved an RMSE of 24812.25 and an R^2 score of 0.920 and outperforms the default configurations (RMSE: 26108.69, R^2: 0.911). These results show that conducting hyperparameter tuning increases the predictive accuracy and reduces the error.

6 Deep Neural Network

6.1 Overview

Deep Neural Networks (DNN) can efficiently handle high-dimensional data such as the dataset being worked with allowing for all of the parameters to be looked at to create the predictions. DNN models formulate complex relationships between complex patterns from the inputted features. If the dataset used is large in both dimensions, it allows the DNN to efficiently improve upon itself and provide fitting output.

6.2 Model Architecture

The deep neural network model was implemented using TensorFlow and Keras. The final model was structured as follows:

- **Input Layer:** 287 Features after encoding and preprocessing
- **Hidden Layer:** Three connected layers of 128 neurons
- **Activation:** ReLU for all hidden layers

- **Regularization:** Dropout of .2 after each hidden layer
- **Output Layer:** The output is sent to a single neuron with linear activation to output predicted house price

The model was compiled using the Adam optimizer using the mean squared error as the loss function.

6.3 Training

The model was trained using many hyperparameter combinations. The following was a good combination of time needed to train-to-performance ratio:

- Epochs: 100
- Batch Size: 128
- Learning Rate: 0.01

The training was performed on scaled numerical features and one hot encoded categorical features. All NaN values were removed during preprocessing for the model to run.

6.4 Experimenting and Tuning

The following experiments were performed to optimize model performance:

6.4.1 Baseline

The initial basic model used comprised of:

- 2 hidden layers: 128 followed by 64 neurons
- Learning rate: 0.001
- Epochs: 50
- Batch size: 32

Result: MSE = 1,260,793,088, RMSE ≈ 35,500 (not good at all)

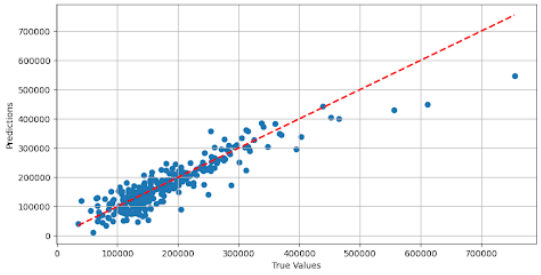

Figure 6: Baseline Model Predictions

6.4.2 Batch Size

Different batch sizes were first tested with the following results:

- Batch size 32: MSE = 991,104,000
- Batch size 64: MSE = 875,392,512
- Batch size 128: MSE = 817,047,616
- Batch size 256: MSE = 32,362,987,520 (unstable learning at this size)

Batch size 128 will be used

6.4.3 Epochs

With batch size 128, these epoch counts were tested:

- 75 epochs: MSE = 749,735,872
- 100 epochs: MSE = 685,382,528
- 200 epochs: MSE = 644,712,512

Since the jump from 100 to 200 epochs drastically increases training time, 100 epochs will be used

6.4.4 Learning Rate

With batch size 128 and 100 epochs:

- Learning rate 0.001: MSE = 685,382,528
- Learning rate 0.01: MSE = 564,026,816
- Learning rate 0.1: MSE = 783,871,168

.01 is the obvious choice among these.

6.4.5 Hidden Layer Configuration

Various combinations of neurons and depth of layers were tested:

- (128, 64): Significantly worse performance
- (128, 128): MSE = 478,299,360 (strong improvement)
- (128, 128, 128): Similar MSE but faster convergence

Due to the faster convergence, the three layers of 128 will be used.

6.5 Final Model

The best model found:

- 3 layers of 128 neurons
- 3 dropout layers at 20%
- Batch size: 128, Epochs: 100, Learning rate: 0.01

Final Result: MSE = 478,299,360, RMSE ≈ 21,868 This result improved RMSE by roughly 40% from the baseline.

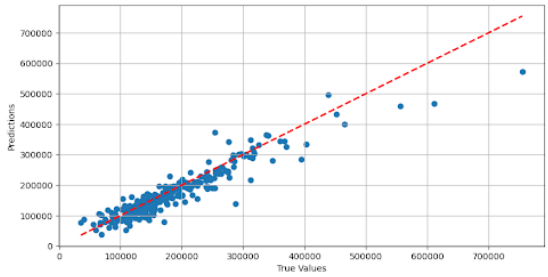

Figure 7: Final Model Predictions

6.6 DNN discussion

The deep neural network was significantly improved through hyperparameter tuning. Increasing depth, learning rate, and batch size provided large improvements in RMSE. However, it can also be observed that more is not always better. Deeper models and fast learning rates tend to lead to divergence or overfitting completely ruining the performance of the model. The final Model provides competitive performance in one package and can be possibly used in ensemble to further improve predictions.

7 Splitting the Dataset

After our status presentation, through the recommendation of Dr. Rasheed, we decided to try classifying the data into 3 categories and then using our regression models on the classified instances. This is because the outliers seen earlier had significant impact on the model predictions

7.1 Gradient Boosting Regressor

We examined one of our better performing models using a rudimentary split of the data into thirds. Our results (see Figure 8 and Figure 9) were promising especially in the lower ranges where we see more consistency with a RMSE of $13,190 for the lower range ($60,000 to $140,000) and $11,270 for the medium range ($140,000 to $190,000). That being said, there is clear training set bias in this examination which is why we followed this experiment with a KNN implementation to explore realistic performance.

The High Price Range ends up performing significantly worse when isolated. Based on Figure 10, it appears predictions are only viable up to around $350,000 and more hyper parameter tuning and data examination would be required to get more effective results in a more volatile price range.

7.2 Gradient Boosting Regressor and KNN

The process of this experiment starts by splitting the range of the SalePrice into thirds and labeling each instance in the training set into one of the low, medium, or high categories. Then the dataset is 80/20 split into training and testing.

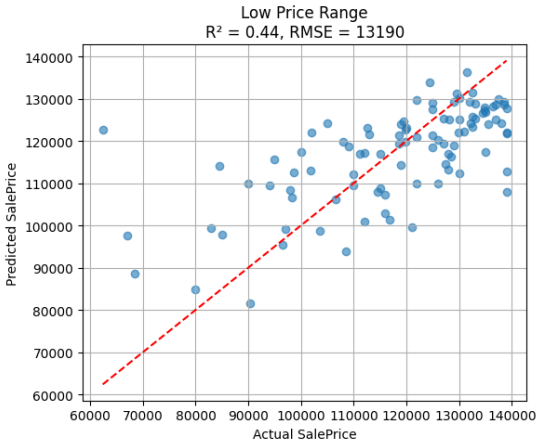

Figure 8: Low Price Range

Figure 9: Medium Price Range

For the K-nearest neighbor classifier we used sklearn's KNeighborsClassifier with a default of 5 neighbor input. We then fit the KNN on the training instances and the bins and ran the predictions.

Following our KNN we create 3 GradientBoostingRegressor models, one for each bin (low, medium, high). The models are each fitted and trained on each set of bins.

Our results can be seen in Figure 11, Figure 12, and Figure 13.

Our experiment yielded a significant improvement in RMSE for low price range instances with $19803.34 and a comparable RMSE for the medium price range at $26292.66.

The high price range still performs the worst at $34612.85 due to significant outliers and our other

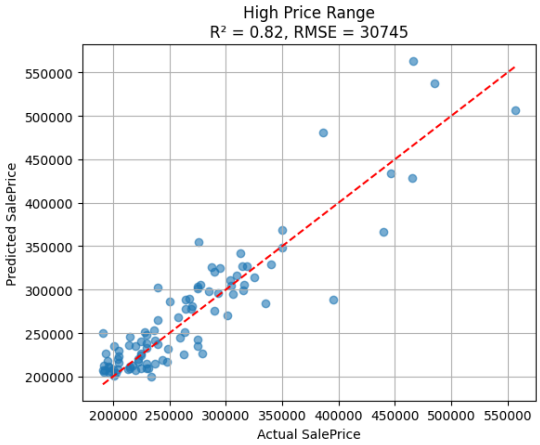

Figure 10: High Price Range

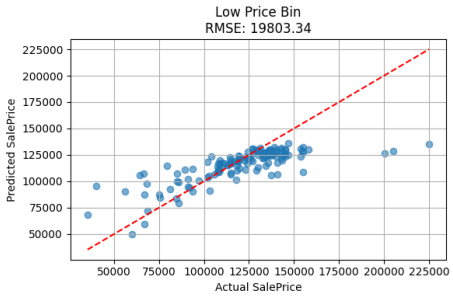

Figure 11: Low Price Bin

models should be opted for in those cases. Using a more sophisticated binning process could likely improve RMSE for future work.

8 Conclusion

Based on our experiments, our deep neural network performs the best with the drawback of high computational needs. Following our all encompassing predictions, XGBoost with hyperparameter tuning produces the lowest RMSE. And finally, a Gradient Boosting Regressor with a KNN bin process produces the lowest RMSE for "low" price ranges.

For market application, a deep neural network should be used with proper tuning given enough training instances for effective results. Additionally, in future experiments, XGBoost should be used in combination with a KNN classifier for binning yields the best average results for median market ranges. Higher price ranges where instance density begins to drop off should be examined on a case-by-case basis when possible since the predictions become extremely volatile.

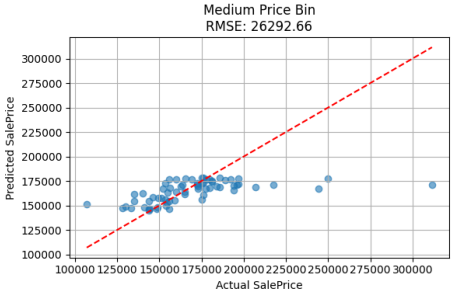

Figure 12: Medium Price Bin

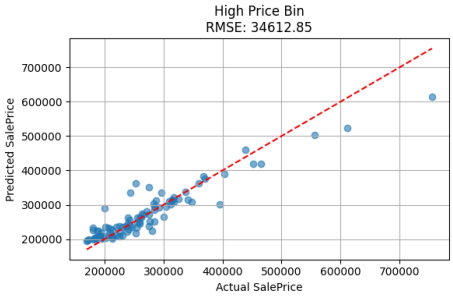

Figure 13: High Price Bin

9 Quick Note on Individual Contributions

Kelsey implemented the LGM, SVR, Gradient Boosting Regressor (GBR), GBR with KNN, setup the template formatting for the paper, wrote full sections (and created associated figures) for Abstract, Introduction, Examining the Data, Simple Model Examination, Splitting the Dataset, Conclusion. Also created all associated slides with those sections for progress and final presentations.

Natalie handled the cleaning and pre-processing of the data as well as encoding categorical values with one-hot and label encoding. She performed Feature Selection on the dataset with Forward Selection and Feature Importance via Random Forest. She wrote the following sections in the paper: Feature Selection Iteration 1, Preprocessing the Dataset, Missing Values, Encoding Categorical Features, Label Encoding, One-Hot Encoding, Feature Selection, Correlation Matrix and Heatmap, Stepwise Feature Selection, and Feature Importance From Random Forest. Natalie also created the associated slides in the progress and final presentations.

Jan implemented Decision Tree (CART), Random Forest, Gradient Boosting, and XGBoost and evaluated all with and without feature selection. He created graphs for each with actual vs. predicted and feature importance graphs. He performed hyperparam-

eter tuning for XGBoost using multi-stage RandomizedSearchCV. He wrote the corresponding sections in the paper: Tree-Based Models, CART, Random Forest, Gradient Boosting, XGBoost, Model Comparison, and Hyperparameter Tuning. He also created the associated progress and final presentation slides.

Quentin implemented everything to do with the deep neural network (DNN). This included setup, DNN preprocessing, tuning, and testing. Quentin also wrote the section for the DNN in the research paper and did all slides associated with it and corresponding figures.

9.1 Main notebook testing and demos for reference

Main group notebook:
https://colab.research.google.com/drive/1kmrWk5u3jyYnGizrGx4kiZ8X80etV79D?usp=sharing
GBR-Testing notebook:
https://colab.research.google.com/drive/1J853wQHyJJQNLHne1nWrmvqc0HEkEKGS?usp=sharing

References

[Ban20] Prateek Banerjee. Tree based machine learning algorithms explained. *Medium*, 2020. **https://medium.com/analytics-vidhya/tree-based-machine-learning-algorithms-explained-b50937d3cf8e**, Accessed: May 5, 2025.

[1] Gusthema. Ranik. *House Prices Prediction Using Tfdf Notebook*, 2023.