Final Project Group 6

House Prices: Advanced Regression Techniques

Predict Sales Prices and Practice Feature Engineering, Random Forest, and Gradient Boosting

Member:

Sabina Yang

Yang Han Deng

Yinfeng Cong

1. Introduction

Ask a home buyer to describe their dream house, and they probably will begin with something other than the height of the basement ceiling or the proximity to an east-west railroad. But it has been proved that much more influences price negotiations than the number of bedrooms or a white-picket fence.

With the dataset provided by Kaggle competitions consisting of 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, we aim to predict the final price of each home.

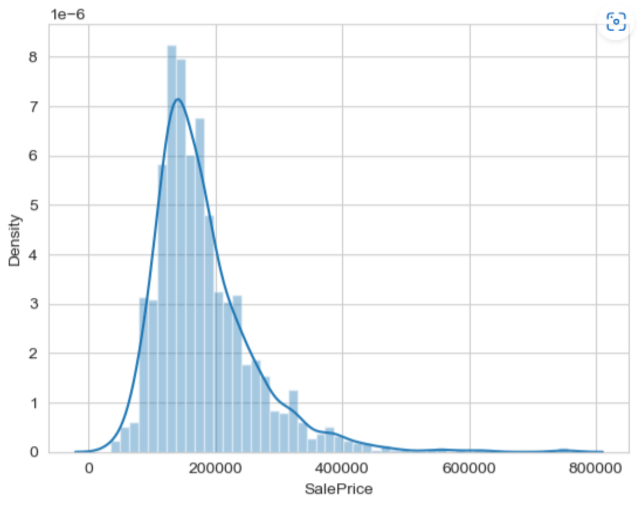
2. Methodology

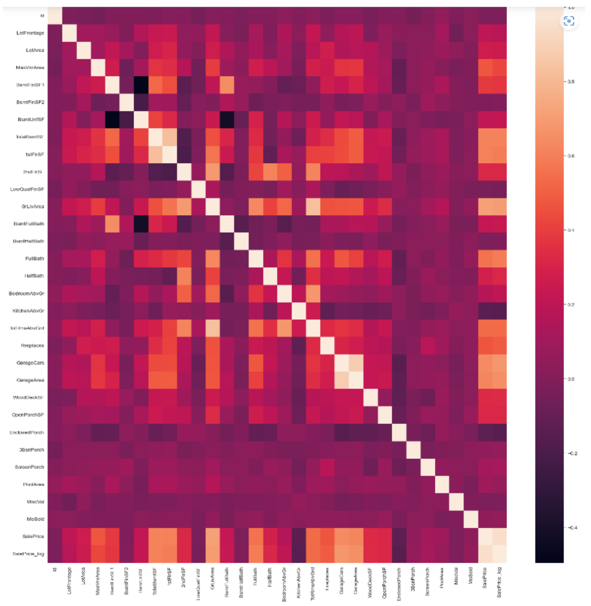
2.1 Data Exploration

We used the pandas' function to analyze the training data and testing data. The training data has 1460 records and 81 features, which include feature[0] = "Id" and feature[80] = SalePrice, the price we want to predict. The testing data also has 1459 records and 80 features, which excludes SalePrice. The training dataset has 6965 null values, which means we have to handle it in the preprocessing section.

Next, we used seaborn and matplotlib to visualize the dataset. I visualized SalePrice with distplot, the average price is about 200,000.

Then, I used a heatmap to check the relationship between features. Brighter colors represent more common values, and darker colors represent less common values. We can see that the SalePrice is more related to "GarageCars", "GarageArea", "ground living area", and "Total square feet of basement area". We can see that the price is more related to the size of the house.





2.2 Data Preprocessing and Feature Design

Because there were null values, in preprocessing we have to replace them with other values. We set those values as '0' or 'No XXX.' For example, the null value in the feature "Fence" will be replaced with "No Fence," and numerical features such as "GarageArea" will be set as 0.

Next, we have some categorical features with numbers such as "Years", and "Ratings"(10 for Very Excellent and 1 for Very Poor). We turned those features into strings to avoid numeric mix-ups.

For those categorical features, we want to convert the training and test data to one hot encoded numeric data. We utilized sklearn's SklearnOneHotEncoder to encode those categorical features.

2.3 Model Exploration

We decided to use Decision Tree as our model since it took less time to train the model compared to other regression models. Then, we tried to use it alone and combine it with the Random Forest technique and the Gradient Boosting technique to see which approach generated the most desired result. We used 5-fold cross-validation, so the data was divided into 5 equally-sized folds. The model was trained on 4 folds and validated on the remaining 1 fold.

For Decision Tree, we use DecisionTreeRegressor, a tool in the Scikit-Learn library. The following is our choice of hyperparameters:

1. “Criterion”: we set to “mse” which was used to evaluate the quality of a split.
2. “Random state”: we set it to 0 for the purpose of reproducibility.
3. “Max_depth”: we tried on these values [5, 10, 30] that control the complexity of the model.
4. “Max_feature”: we tried on these values [0.1, 0.3, 0.7] that also control the complexity of the model since they specify the number of features we would consider when we split on a node.

For Random Forest, we RandomForestRegressor, a tool in the Scikit-Learn library. The following is our choice of hyperparameters:

1. “N_jobs”: we set it to -1, so we can use all the available CPU to parallel the training and testing process.
2. “Random state”: we set it to 0 for the purpose of reproducibility.
3. “Bootstrap”: we set it to true. Each tree is built on a random sample of training data with replacement, reducing the effect of overfitting.
4. “N_estimators”: we tried on these values [200, 300, 400, 500, 600] which tell the number of trees we want to include in the bag.
5. “Max_feature”: we tried on these values [0.1, 0.3, 0.6] that control the complexity of the model since they specify the number of features we would consider when we split on a node.

For Gradient Boosting, we use GradientBoostingRegressor, a tool in the Scikit-Learn library. The following is our choice of hyperparameters:

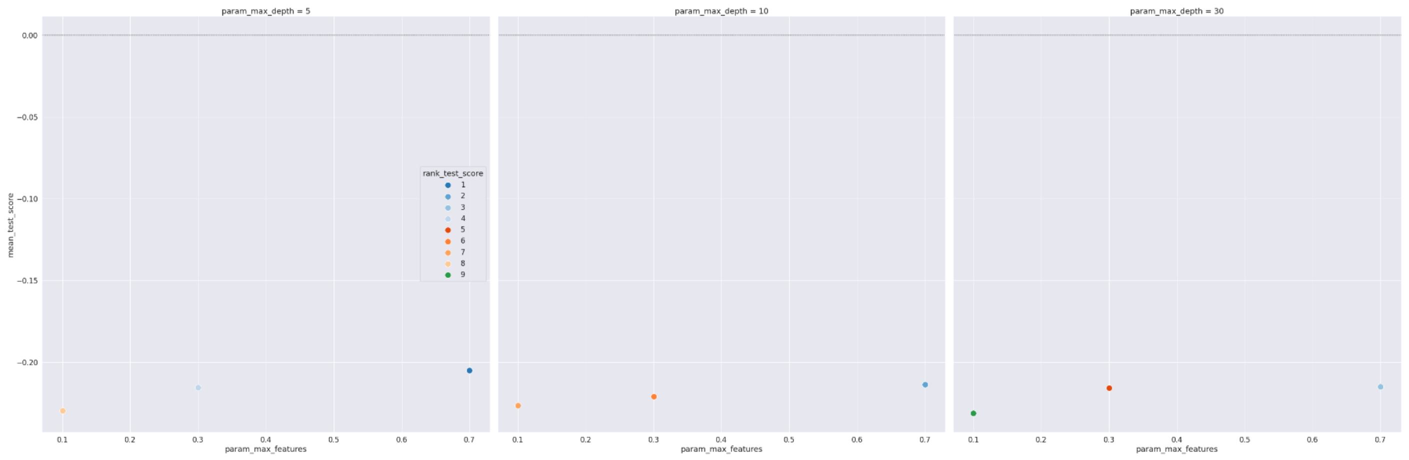
1. Random state”: we set it to 0 for the purpose of reproducibility.
2. “Max_feature”: we set it to “sqrt” which only considers the square root number of features in the split.
3. “N_estimators”: we tried on these values [300, 500, 1000] which tell the number of trees we want to use in making the prediction.
4. “Max_depth”: we tried on these values [5, 10, 30] that control the complexity of the model.
5. “Learning_rate”: we tried on these values [0.01, 0.05, 0.1] that control our efficiency in fine-tuning the model and whether we can find the minimum loss value.

2.4 Performance Validation

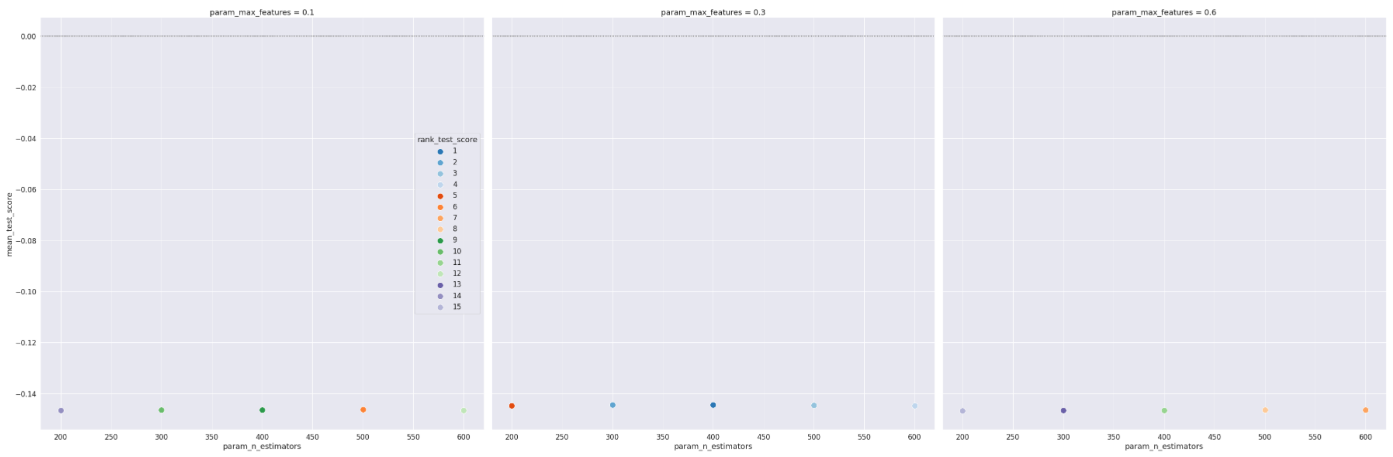
We need to know, for each of the three kinds of models, which combination of the hyperparameters would produce the model with the highest testing score of that kind. Then, among these 3 kinds of models, we will select the kind that has the best testing score. In order to do this, we used 5-fold cross-validation. The reason that we chose cross-validation as our validation method is that it can give a better estimate of the generality quality of the trained model. Rather than evaluating the model only on a single set of validation data, it evaluated the model on different sets of new and unseen data.

Here are the plots that show the predicting performance under different combinations of the hyperparameters in 3 kinds of models:

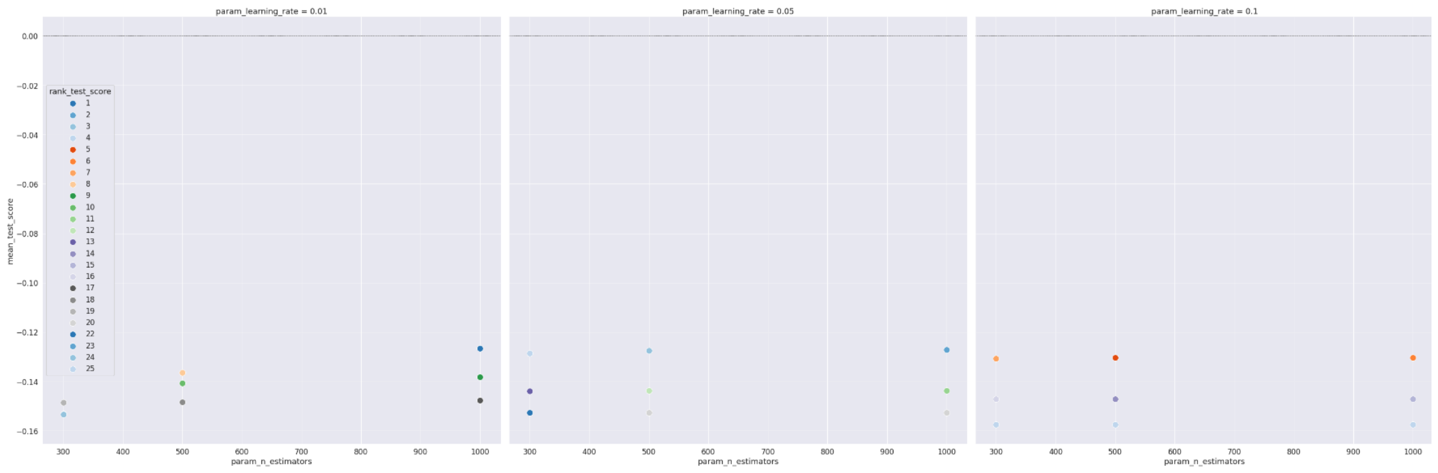
- For Decision Tree:

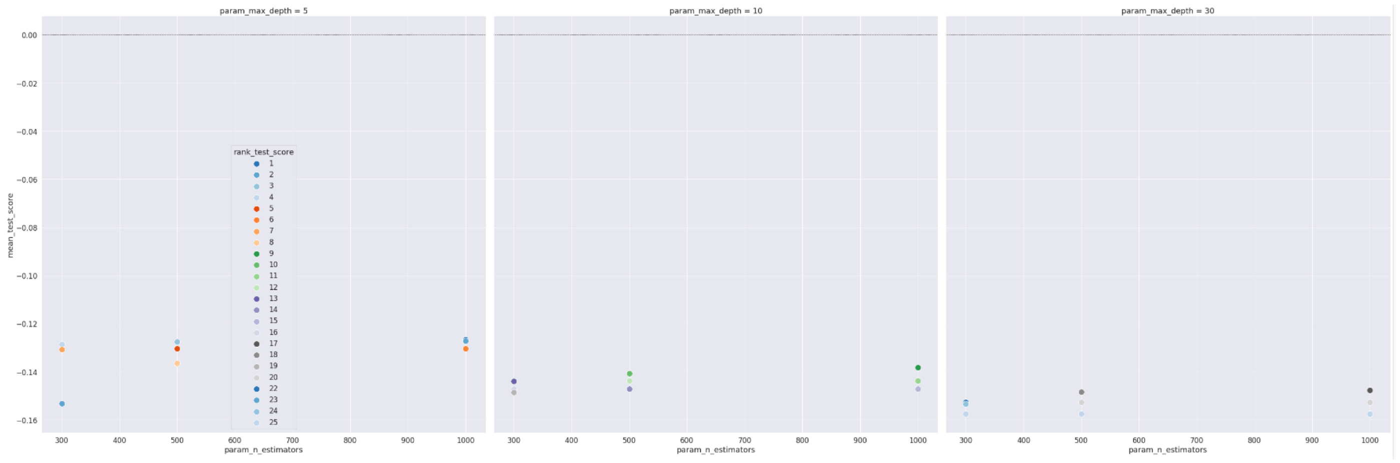


- For Random Forest:



- For Gradient Boosting:



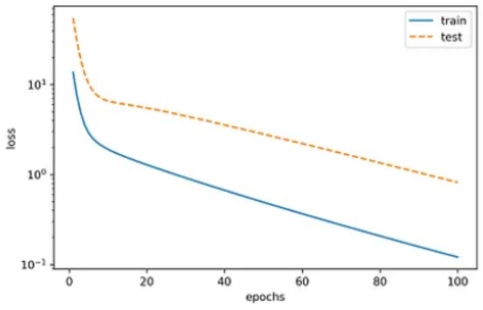


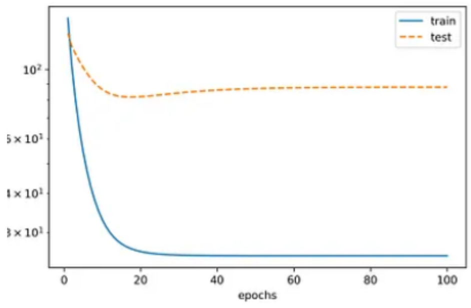
As we can see from the above, the Gradient Boosting model with the hyperparameter of “n_estimator” = 1000, “max_depth” = 5, and “learning_rate” = 0.01 shows the best performance on validation data. Finally, we adopted this parameterized model to solve our predicting house prices problem.

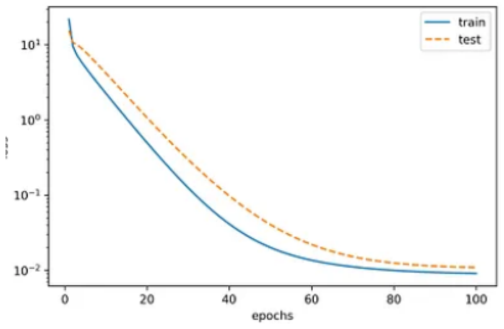
2.5 Adaption to Underfitting and Overfitting

To adapt the underfitting and overfitting problem in our project, we first need to make sure what would cause the problem of underfitting and overfitting in our project. As we have learned and know in the lecture, the problem of underfitting and overfitting would be affected by the model's capacity. As we tried to test the proper ratio between the training data and the test data. we setted three hyperparameters (learning_rate = 0.01 batch_size = 10 epochs = 100) to try testing if the model here is overfitting or underfitting.

First of all, we have created the iterator for training the models. Then we picked the training data size to be 80%







of the data, and test data is the other 20% of the data. After we have utilized the training iterator. We have found that the model would become overfitting based on our chosen data. (the 1st figure)

Then we have chosen to change the ratio of the data to solve the problem of overfitting. We picked 40% of the data as the training data, and 60% percent as the test data (I know the ratio here might be too small, and it might have the problem of underfitting) Here is the graph of this situation.
(We can clearly see that it would be underfitting based on this ratio we have picked in the 2nd figure)

After trying several times, we think we have finally picked the ratio of data that would be the best solution for the model. The size of the test data would be (1459, 80), and the size of the training data would be (1460, 81).

As the 3rd figure above shows, there would not be the problem of underfitting or overfitting anymore. This means that we have finished our goal of adaptation to underfitting and overfitting.

3. Result & Conclusion

Based on the result we have found during the project, it is clear that gradient booster is the best method to use here. For this reason that we have changed the parameters of the function many times to get the best results we want. However, even though we have found that the NRMSE of the gradient booster is much better than the other two methods, the time it will take would also be much more significant.

The method of the decision tree is one of the simplest methods we can think of. As we predicted before the operation, this method would take the least amount of time and its performance might be the worst of the three methods. Indeed, the NRMSE of it is -0.205 and it just took less than 2 seconds.

For the method of random forest, we did not train much time on the data, and the parameters here were easily picked. As it is mentioned above, the NRMSE of it is -0.144. This would be a much better method compared to the decision tree. Also, it took much less time (75 seconds) compared to the gradient booster.

As a result, based on the project we have done. We think a gradient booster would be the best method here if we only want the most precise result. However, for a huge company, the method of random forest might be the best choice. (The error of it is relatively small, and it took much less time compared to the gradient booster)

4. Individual Contribution

Sabina Yang: Introduction, model exploration, performance validation
Yang Han Deng: Data Exploration, Data Preprocessing, and Feature Design
Yinfeng Cong: adaption to underfitting and overfitting, result, conclusion