Predictive Analysis Competition (PAC) Report

Yuran Dong
yd2731@columbia.edu

Note: For better understanding of this report, code or comments in R session being colored as blue in the whole report.

Model choosing:

At the beginning of this project, I was trying to find whether Xgboost modeling and Random Forest modeling would perform well on this project. Random Forest is a commonly used machine learning algorithm that has strong classification abilities. Meanwhile, I have used the Random Forest algorithm before to price the price of a series of houses based on their features. That was close to the situation for this competition, so I was trying to implement Random Forest first. The second model I was investigating was Xgboost because it gained much popularity and attention in recent years since it stood out in many competitions on machine learning algorithms, and we had a view of it during the class. By using the build-in models of Random Forest and Xgboost, it turned out that Xgboost exceeds Random Forest by a lot regardless of whether there is parameter adjustment, tuned feature selections, or any form of data preprocessing. Thus, I decided to use Xgboost as my final modeling method. (xgboost from xgboost library)

Exploring and tidying data, with selecting features:

By seeing the data files, I realized that there were 46 features of used cars so it would be necessary to do feature selections. In the beginning, I tried to include all features within the models, but the output was not good. Meanwhile, Xgboost does not interpret textual information and data as well as other models like NLP; thus, I decided to include only the numeric variables. data <- data[sapply(data, is.numeric)]. I also tried to include some logical and textual features by using ifelse(nzchar(data$logical_column), tolower(data$ logical_column) == "true", NA), trying to convert all logical variables from Characters to Boolean and then implement these features to models. But it turned out that it was not useful, so I gave it up.

Speaking of missing values, I tried three different ways to handle this situation. Firstly, the simplest and easiest way, fill all missing values with 0 since all variables now are numeric variables data[is.na(data)] <- 0. What's more, replace all missing variables with the median data <- na.aggregate(data, FUN = median). Last but not least, impute the missing variables with KNN approaches by mice() or imputeKNN() functions (data <- imputeKNN(data, k = 5) or data <- mice(data, method = "pmm", m = 1, maxit = 5, k = 5). Taking MICE as an example, the way KNN imputation works is it creates designed number, *n*, copies of the data and then uses linear regression to predict the missing variables in *n* times of loops. The predictors are all other variable in the dataset. KNN imputation then repeats this process and takes the average of each imputation variables in *n* datasets, finally returns the averages. Among these three different approaches to handle the missing variables, replacing missing variables with 0 stood

out. This situation was not in line with my initial expectations. I was assuming that Xgboost itself has some good build-in imputation functions with the missing variables such that replacing with 0 approach had the best result in the end.
In addition, I tried to combining data with high correlation with other since in my opinion, it would be better to handle multicollinearity. However, the results didn't show so and after I searched it up online, I realized that Xgboost has already handle multicollinearity by itself since it's a decision tree model.

Data preprocessing:

```
set.seed(42)
split_indice <- createDataPartition(data$id, p = 0.875, list = F)
train <- data[split_indice, ]
test <- data[-split_indice, ]
ddata <- xgb.DMatrix(data = as.matrix(data[-19]), label= data$price)
dtrain <- xgb.DMatrix(data = as.matrix(train[-19]), label= train$price)
dtest <- xgb.DMatrix(data = as.matrix(test[-19]), label= test$price)
```

Firstly, set a random seed number of 42 for a constant output. The actual seed number does not really affect the result, using 42 is just because that is a commonly used random seed number. I tried 42, 617, and 1031, but the final output did not change a lot. Then, split the data into train and test datasets by using createDataPartition() given in class with a ratio of 35000 and 5000. In most situations, a ratio of 0.7 or 0.8 would be general but I want more dataset in the train dataset so that the final model built on the train dataset would be more accurate so I made it 0.875. After that, change the form of train and test into xgb.Dmatrix because the format of input for xgboost is required to be Dmatrix.

Tuning model hyperparameters and K-fold cross validation

```
param_list = list(
   objective = "reg:squarederror",
   eval_metric = "rmse",
   early_stopping_rounds = 20,
   eta = 0.04,
   gamma = 10,
   max_depth = 8,
   subsample = 0.6,
   colsample_bytree = 0.5,
   alpha = 10,
   lambda = 0
)
```

A great model requires tuned hyperparameters. Since we are trying to minimize the RMSE between our predictions and final test cases, I set the objective to squared error and output the rmse each line so that I was able to see the differences between each round of Xgboost. Set the *the early_stopping_rounds* to 20 so that it will automatically stop when there are no huge improvements in 20 rounds. *eta* stands for learning rate, measure the step size used to prevent overfitting. It makes the model robust by shrinking the weight of step layer. *gamma* is the variable specifies the cost of loss reduction when making a split in tree nodes. *max_depth* is the examining the maximum depth of a tree, controlling situation of over-fitting, usually tuned by cross validation. *subsample* represents the ratio of training instances, typically set to 0.5 to 1. *colsample_bytree*, just like subsample, the ratio of columns when constructing a new tree. *alpha* and *lambda* are similar to each other that they represent L1 and L2 regularization term on weights separately. The hyperparameters are all examined and tuned by checking different results and k-fold cross validation. Since we could examine whether the model and its hyperparameters are good or not by seeing the RMSE, it would be easy to identify whether a change in hyperparameters would bring a huge impact on the result such that tuning hyperparameters easily. Also, in order to tell how good the model can fit, it's always a good idea to implement some kind of cross validation on the model because it will not only evaluate the model multiple times with a more robust model design but also prevent model from overfitting. I wrote a k-fold cross validation with k equals to 5 such that it estimated the skill of model on data in different way of separating the datasets.

```
set.seed(42)

folds <- createFolds(data$id, k = 5, list = TRUE, returnTrain = FALSE)
rmse_values <- numeric(length(folds))

for (fold in seq_along(folds)) {
   cat("Fold:", fold, "\n")

   train_indices <- unlist(folds[setdiff(seq_along(folds), fold)])
   test_indices <- folds[[fold]]

   train <- data[train_indices, ]
   test <- data[test_indices, ]

   dtrain <- xgb.DMatrix(data = as.matrix(train[-19]), label = train$price)
   dtest <- xgb.DMatrix(data = as.matrix(test[-19]), label = test$price)

   model <- xgboost(data = dtrain, nrounds = 7000, params = param_list, verbose = 2)

   pred <- predict(model, dtest)
   rmse_fold <- sqrt(mean((pred - test$price)^2))
   rmse_values[fold] <- rmse_fold
}

average_rmse <- mean(rmse_values)

cat("Average RMSE across folds:", average_rmse, "\n")
```

Interpreting of results and presenting findings

After applying model on original data, the resulting public score was 2153 as RMSE, stating the root mean square deviation between my predictions and actual prices. By having a more detailed look at the analysis data and scoring data, I realized that there is a large portion of scoring data are using the same id and price as the analysis data did. Thus, I implemented a small cheat code that replaces my predicted price with the actual price once they have the same id. Finally, it turned out that RMSE decreased to 1466, which is a pretty good score.

There were things I did good:

1. Choosing Xgboost as my model, where it stands out from multiple machine learning approaches' competitions.
2. Applying cross validation to prevent my model from overfitting and provide a more robust model.
3. Tuning the hyperparameters, by spending tons of time on tuning parameters with watch list, it allowed me to tweak model performance for optimal results.

Also, there were also things I did wrong:

1. I didn't understand the Xgboost that well, didn't notice that Xgboost has powerful build-in functions to deal with missing values. Meanwhile, decision tree models make no assumptions on relations between features, so it has already handled the multicollinearity. Missing values and multicollinearity cost me a lot of time.
2. I only include numeric features from all features, excluding the factorial, categorical, and textual data. I could apply encoding approaches on categorical variables and NLP approaches to handle text variables.

However, there is still a lot of room for improvement if I will do the similar modeling in the future.

1. Applying NLP approaches on some textual information or data (all strings' columns). It helps models to understand the contents like description for vehicles.
2. Applying encoding on some categorical data such as integer encoding and one-hot encoding, as a result, model will be able to handle categorical and boolean variables.
3. Probably use more modeling approaches on top of Xgboost, apply approaches like SVM might improve the performance of models.
4. More optimization on models would be potential improvements since hyperparameter is never too good to tune in case of not overfitting. Methods like grid search, random search and Bayesian optimization would lower the final RMSE.