PAC Report

Lin Zhou

In this Kaggle competition for predicting used car prices, I put in a great deal of effort in preparing the data, particularly with categorical variables. I sought out and applied various methods in an attempt to extract useful information from the features. I also experimented with multiple encoding techniques to convert qualitative categorical variables into dummy variables. For model selection, I tried using both random forest and xgboost algorithms for price prediction and employed grid search combined with 5-fold cross-validation to find the optimal hyperparameters. The results showed that data cleaning had a positive impact on the prediction effectiveness of both models, as evidenced by the reduced RMSE.

However, the performance of the xgboost algorithm was significantly inferior to that of the random forest, indicating a need for me to find a more suitable encoding method for qualitative categorical variables. Due to a lack of experience, I went through a repetitive process of data cleaning, which slowed down the progress of the project. Moreover, effective feature engineering plays a crucial role in prediction accuracy. My feature cleaning was still too rudimentary, and I did not successfully extract the most useful features. Further refinement in this area would likely reduce prediction errors.

Because of the many failures and poor results I experienced with the xgboost model, I did not use it for the final prediction. However, I have included the relevant unsuccessful code in the corresponding section below, which pertains to variable encoding and model tuning. The separate submission of code that I am providing will only encompass the portion where I employ the random forest algorithm for price prediction.

1. Data Tidying

1.1 Imputing Missing Values

First, I explored and understood the structure and summary statistics of the data for a comprehensive understanding of the dataset. Then, I have tried using KNN algorithms to predict and fill in missing values, but this complex method did not yield a higher prediction accuracy.

1.1.1 identifying missing values

The results indicate that out of 45 variables, 27 have missing values. No column in the dataset has a missing data percentage exceeding 50%, therefore, there is no need for variable deletion.

```
analysisData[analysisData == ""] <- NA
missing_values <- sapply(analysisData, function(x) sum(is.na(x))/length(x)) * 100
```

1.1.2 categorical columns

For categorical columns, the missing values were imputed with the mode, which is the most common value in each column.

1.1.3 numeric columns

For numeric columns, the strategy was to impute missing values using the median of each column.

1.1.4 boolean columns

For the variable ‘is_cpo’, missing values were filled with ‘False’ based on its meaning. Following this, Boolean variables stored as characters were converted to 1 and 0.

1.2 Variable Transformation

1.2.1 parsing

This step involves extracting representative features from key categorical variables, providing a finer granularity in our data. This enhanced level of detail lays a richer informational foundation for subsequent predictive modeling.

1. ‘engine_type’ (cylinders_configuration + cylinders_number)

The code decomposes the ‘engine_type’ variable into two more specific variables: one indicating the type of engine configuration, and the other the number of cylinders.

```
extract_cylinder_config <- function(engine_type) {
  # extract the first letter
  return(substring(engine_type, 1, 1))
}
extract_cylinder_number <- function(engine_type) {
  # extract all the numbers
  return(as.integer(gsub("[^0-9]", "", engine_type)))
}
analysisData$cylinders_configuration <- sapply(analysisData$engine_type, extract_cylinder_conf
ig)
analysisData$cylinders_number <- sapply(analysisData$engine_type, extract_cylinder_number)
```

2. ‘description’ (description_length + is_discount) and ‘major_options’ (options_count + option_is_premium)

Due to the complexity of the content in the sellers’ vehicle description variable, I extracted information on the number of characters and whether the value mentioned any discount-related keywords, integrating these into new columns. This approach can be further refined. It enables the application of techniques like Bag of Words (BoW) for vectorization or Sentiment Analysis, which are useful for assessing the emotional tendencies of the sellers.

```
analysisData$description_length <- nchar(as.character(analysisData$description))
discount_keywords <- c("off", "discount", "below", "save", "reduction", "deal", "special")
analysisData$is_discount <- sapply(analysisData$description, function(x) {
  any(sapply(discount_keywords, function(kw) grepl(kw, tolower(x), fixed = TRUE)))
})
analysisData$is_discount <- as.integer(analysisData$is_discount)
```

3. ‘power’ (power_hp + power_rpm) and ‘torque’ (torque_lbft + torque_rpm)

```
analysisData$power_hp <- as.numeric(gsub(" hp.*", "", analysisData$power))
analysisData$power_rpm <- sapply(analysisData$power, function(x) {
  rpm <- gsub(".*@ ", "", x)  # delete all content before "@"
  rpm <- gsub(" RPM", "", rpm)  # delete "RPM"
  as.numeric(gsub(",", "", rpm))  # remove commas and convert to numeric values
})
```

4. ‘transmission_display’

I discovered that the content overlap between the ‘transmission_display’ and ‘transmission’ variables was significant. Therefore, I extracted content related to speed and transformed it into a ‘transmission_speed’ variable.

5. ‘listed_date’ (listed_date + list_year + list_month)

The code decomposes the original listed_date into three separate columns, each representing a distinct temporal element.

```
analysisData <- analysisData %>%
  mutate(
    listed_date = as.Date(listed_date),
    list_year = as.integer(format(listed_date, "%Y")), # Convert to integer
    list_month = as.integer(format(listed_date, "%m")), # Convert to integer to remove leading
zero
    list_day = as.integer(format(listed_date, "%d")) # Convert to integer
  )
```

6. ‘car_age’

Inferred from the year feature, this created a new feature representing the age of the car.

```
analysisData$car_age <- 2023 - analysisData$year # Assuming the current year is 2023
```

1.2.2 grouping

1. transformations ‘interior_color’, ‘exterior_color’ and ‘trim_name’

Taking the ‘interior_color’ variable as an example, I merged highly related categories within the variable into one to reduce the number of unique color descriptions. This simplification of the analysis can enhance the performance of machine learning models by reducing dimensionality. However, this grouping approach was still rather rudimentary. If given another opportunity, I would like to explore the use of clustering algorithms to categorize the colors into distinct classes.

```
analysisData$interior_color <- tolower(analysisData$interior_color)
colors_to_standardize <- c("black", "gray", "brown", "red", "white","stone","dark","light","eb
ony","parchment","charcoal","blue","almond")
for(color in colors_to_standardize) {
  analysisData$interior_color <- ifelse(grepl(color, analysisData$interior_color), color, anal
ysisData$interior_color)
}
# replace "No Color" and "Unspecified" with "None"
analysisData$interior_color <- str_replace_all(analysisData$interior_color, "no color", "non
e")
analysisData$interior_color <- str_replace_all(analysisData$interior_color, "unspecified", "no
ne")
analysisData$interior_color <- str_replace_all(analysisData$interior_color, "\\(.*?\\)", "")
analysisData$interior_color <- str_trim(analysisData$interior_color)
```

2. ‘Other’ category

Furthermore, for the interior_color, exterior_color, trim_name, and model_name variables, I combined categories with a frequency lower than 0.01 into a single ‘Other’ category. This is particularly useful in analysis and modeling, where having too many categories and few observations can lead to overfitting or skewed results.

```
freq_threshold <- 0.01
# Group 'model_name'
model_freq <- analysisData %>% group_by(model_name) %>% summarise(Count = n()) %>%
  mutate(Frequency = Count / sum(Count))
infrequent_models <- model_freq %>% filter(Frequency < freq_threshold) %>% pull(model_name)
analysisData$model_name_grouped <- ifelse(analysisData$model_name %in% infrequent_models, 'Oth
er', as.character(analysisData$model_name))
```

The data processing approach applied to the categorical variables effectively reduced the number of qualitative variables from 27 to 12. Such a strategy successfully minimized the unique values of complex variables, streamlining the dataset for more efficient analysis. But the process of data tidying is still too rough, which is a mistake in this competition.

```
            make_name             body_type
                   46                     9
            fuel_type          transmission
                    5                     4
        wheel_system         franchise_make
                    5                    40
       listing_color cylinders_configuration
                   14                     5
   model_name_grouped      trim_name_grouped
                   22                    23
interior_color_grouped  exterior_color_grouped
                    7                    10
```

1.2.3 categorical encoding

I experimented with transforming textual categorical variables into dummy variables by employing integer encoding, target encoding, and one-hot encoding. This was done to convert the features into a format compatible with XGBoost. In practice, however, using one-hot encoding (for variables with fewer than 10 unique values) and target encoding (for other variables) didn't yield results as good as the simplest method, integer encoding. I need more time to refine and optimize these encoding strategies.

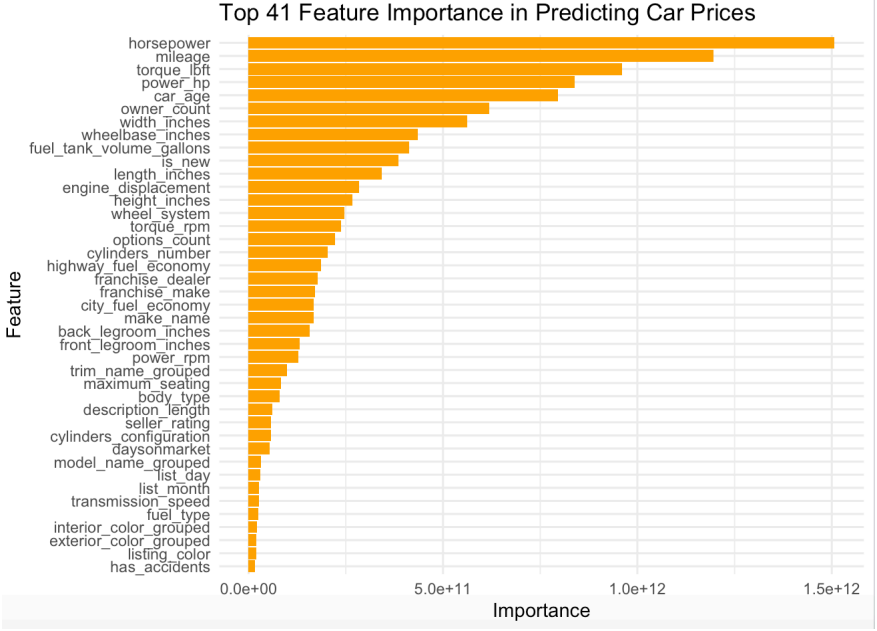
```
# apply integer encoding to the categorical variable column
analysisData[categorical_cols] <- lapply(analysisData[categorical_cols], function(x) as.intege
r(as.factor(x)))
# one hot coding
categorical_col_names <- names(analysisData)[categorical_cols]
# determine which columns need to be individually coded
one_hot_encoding_columns <- categorical_col_names[sapply(analysisData[categorical_col_names],
function(x) length(unique(x)) <= 10)]
encoded_data <- analysisData[one_hot_encoding_columns]
encode_one_hot <- function(data, columns) {
  encoded_data <- data
  for (col in columns) {
    dummies <- model.matrix(~ data[[col]] - 1)
    colnames(dummies) <- paste(col, colnames(dummies), sep = "_")
    encoded_data <- cbind(encoded_data, dummies)
  }
  return(encoded_data[, !(colnames(encoded_data) %in% columns)])
}
encoded_data <- encode_one_hot(encoded_data, categorical_col_names)
encoded_column_names <- colnames(encoded_data)
# target coding
target_encoding_columns <- categorical_col_names[sapply(analysisData[categorical_col_names], f
unction(x) length(unique(x)) > 10)]
target_data <- analysisData[c(target_encoding_columns, "price")]
# create a vtreat treatment - for continuous target variables
treatment_plan <- designTreatmentsZ(
  dframe = target_data,
  varlist = names(target_data)[names(target_data) != "price"],
  verbose = FALSE
)
target_data_treated <- prepare(treatment_plan, target_data)
analysisData[target_encoding_columns] <- target_data_treated
```

2 Feature Selection

2.1 Random Forest

It identified and selected the top 41 most influential features in predicting car prices from a total of 50 variables using the Random Forest model. These variables were used as inputs for subsequent Random Forest prediction algorithms.

```
library(ranger)
formula <- as.formula("price ~ .")
rangerModel <- ranger(
  formula,
  data = analysisData,
  importance = 'impurity',  # This computes the importance of each variable
  num.trees = 1000
)
importance <- rangerModel$variable.importance
importance_sorted <- sort(importance, decreasing = TRUE)
top_features <- names(importance_sorted)[1:41]
print(top_features)
```



2.2 XGBoost

It identified and selected the top 50 most influential features in predicting car prices from a total of 61 variables using the XGBoost model. These variables were used as inputs for subsequent XGBoost prediction algorithms.

```
data_matrix <- xgb.DMatrix(data = as.matrix(analysisData[, -which(names(analysisData) == "pric
e")]),label = analysisData$price)
set.seed(617)
xgb_model <- xgboost(
  data = data_matrix,
  nrounds = 100,
  verbose = 0
)
# extract feature importance
importance_matrix <- xgb.importance(model = xgb_model)
top_features <- importance_matrix$Feature[1:50]
print(top_features)
```

3 Model Selection, Training and Predicting

3.1 Random Forest

3.1.1 split data set

The dataset containing the top_features is split into training and testing sets, enabling the training of a model and subsequently evaluating its performance on unseen data.

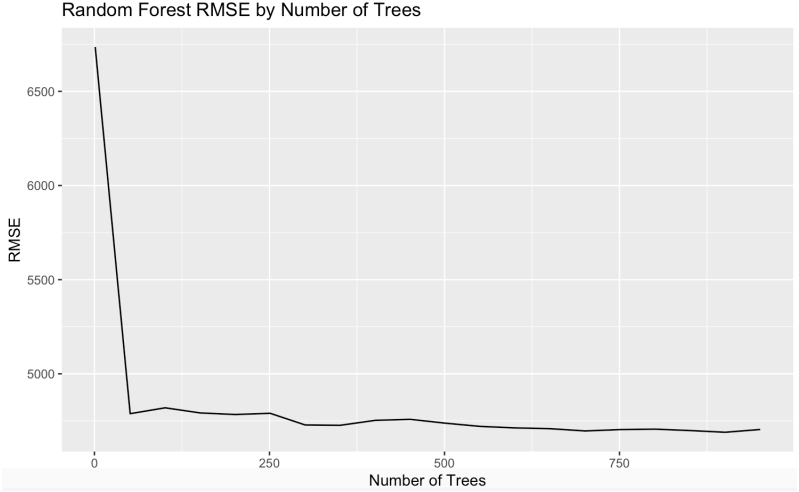
```
set.seed(123)
trainingRows <- createDataPartition(analysisData$price, p = 0.8, list = FALSE)
trainData <- analysisData[trainingRows, c(top_features, "price")]
testData <- analysisData[-trainingRows, c(top_features, "price")]
```

3.1.2 model evaluation and tuning

I utilized grid search coupled with cross-validation for hyperparameter tuning and identified the optimal parameters, which decreased the RMSE from 2300 to 1830 (public score). However, I encountered issues with insufficient computing power when attempting to conduct more fine-grained parameter adjustments. Additionally, the increase in RMSE for the private score compared to the public score suggests that there is still room for improvement in the model's generalization ability.

1. num.trees = 1000

As the number of trees increases, the error decreases until it reaches a certain point, beyond which the graph levels off, indicating an asymptotic behavior.



2. mtry = 14 and min.node.size = 5

```
library(ranger)
trControl=trainControl(method="cv",number=5)
tuneGrid = expand.grid(mtry=c(2, 4, 6, 8, 10, 14, 16),
                       splitrule = c('variance'),
                       min.node.size = c(2,5,10,15,20,25))
set.seed(617)
cvModel = train(price~.,
                data=trainData,
                method="ranger",
                num.trees=1000,
                trControl=trControl,
                tuneGrid=tuneGrid)
cv_forest_ranger = ranger(price~.,
                          data=trainData[, c(top_features, "price")],
                          num.trees = 1000,
                          mtry=cvModel$bestTune$mtry,
                          min.node.size = cvModel$bestTune$min.node.size,
                          splitrule = cvModel$bestTune$splitrule)
print(cv_forest_ranger)
# Make predictions on the test data
rfPredictions <- predict(cv_forest_ranger, trainData[, c(top_features, "price")])
rfRMSE <- sqrt(mean((rfPredictions$predictions - trainData$price)^2))
print(paste("Tuned Random Forest RMSE:", rfRMSE))
```

3.2 XGBoost

I spent a considerable amount of time coding the variables for the xgboost model and training it. However, the outcome was far from ideal, only reducing the RMSE from 3000 to 2300 (public score). The performance of the model with the optimal hyperparameters I found was even worse than the model with default parameters. Moreover, nearly all predictions suffered from severe overfitting issues. As a result, I ultimately had to abandon spending more time on xgboost and opted for random forest as my final submission.

```r
# the optimal hyperparameters
params <- list(
  objective = "reg:squarederror",
  eta = 0.1,
  max_depth = 9,
  subsample = 0.75,
  min_child_weight = 12
)
num_round <- 500
bst_model <- xgb.train(params, dtrain, num_round)
# make predictions on the test data
xgbPredictions <- predict(bst_model, dtrain)
xgbRMSE <- sqrt(mean((xgbPredictions - trainData$price)^2))
print(paste("Tuned xgboost RMSE:", xgbRMSE))

# tuning process
set.seed(617)
params_grid <- expand.grid(
  eta = c(0.01, 0.1, 0.3),
  max_depth = c(3, 6, 9),
  min_child_weight = c(1, 2, 3),
  subsample = c(0.5, 0.75, 1),
  colsample_bytree = c(0.5, 0.75, 1)
)
results <- list()
for(i in 1:nrow(params_grid)) {
  params <- as.list(params_grid[i,])
  cv_results <- xgb.cv(
    params = params,
    data = as.matrix(trainData[top_features]),
    label = trainData$price,
    nrounds = 500,
    nfold = 5,
    metrics = "rmse",
    showsd = TRUE,
    verbose = 0
  )
  results[[i]] <- cv_results
}
best_model <- NULL
best_rmse <- Inf
best_index <- NULL
for(i in seq_along(results)) {
  rmse <- min(results[[i]]$evaluation_log$test_rmse_mean)
  if(rmse < best_rmse) {
    best_rmse <- rmse
    best_model <- results[[i]]
    best_index <- i
  }
}
best_params <- params_grid[best_index, ]
best_params_list <- as.list(best_params)
names(best_params_list) <- sapply(best_params_list, as.character)

final_model <- xgboost(
  data = as.matrix(trainData[top_features]),
  label = trainData$price,
  nrounds = 500,
  params = best_params_list,
```

```
  verbose = 0
)
```