Car Price Predictor

Jacob Szlachetka

Introduction

In this project we propose a regression model that predicts car price from a set of relevant features. Data set *Car_price_cleaned.csv* is referred, where predictors are standardized and quadratic basis functions are applied. Firstly, we load the data set in R and observe the variables.

```
car_price = read.csv("C:/Users/speed/Desktop/SL/R Projects/Car_price_cleaned.csv")
attach(car_price) # attach variables (predictors and response)
dim(car_price) # sample size of 205, 43 variables
```

```
## [1] 205  43
```

```
sample(names(car_price), 10) # variable names (limit 10)
```

```
##  [1] "power_to_weight_ratio" "brand"                 "symboling"
##  [4] "citympg_squared"       "peakrpm_squared"       "wheelbase"
##  [7] "enginelocation"        "enginesize_squared"    "enginesize"
## [10] "wheelbase_squared"
```

The sample size is 205, and we have 40 predictors to consider (excluding from the 43 variables the response *price*, as well as *car_ID* and *CarName*). We may observe that there are some quite complicated predictors, so the goal in this case could be to build a model that is relatively simple, in the sense that there are fewer features. This allows for a more interpretable model with the application of using a few intuitive and important predictors such as horsepower to predict car price.

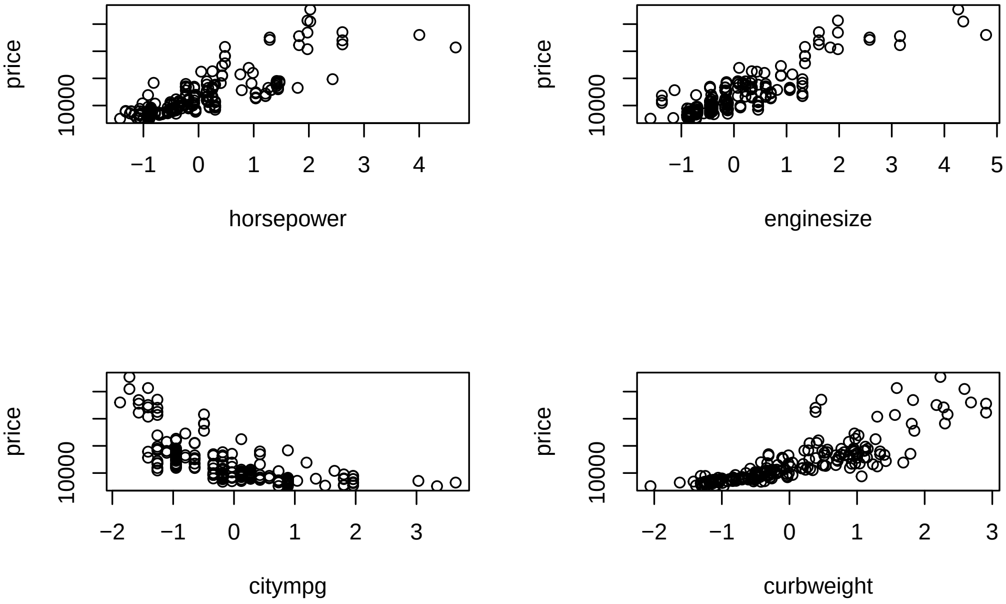
Alternatively, we could use a more flexible model, one that increases the complexity by considering many predictors in an attempt to maximize prediction accuracy. Regularization is applied in this study to models of higher dimension, in order to assess whether it pays off to sacrifice interpretability for accuracy.

In total, seven methods (linear model, general additive model, ridge, lasso, bagging, random forest, boosting) are tested and compared through the test error. We discuss the results after the test errors are generated, and argue for the best model to select for this specific task.

Selecting Predictors

As mentioned previously, it would be great for only a handful of predictors to well explain the response, and four predictors have exhibited some initial good signs at fulfilling this role. Individual scatterplots are provided below.

```
par(mfrow=c(2,2))
plot(horsepower, price)
plot(enginesize, price)
plot(citympg, price)
plot(curbweight, price)
```



There are some clear patterns between the predictors and the response, which may be well modeled through a general additive model (GAM). In addition, these four features are in fact intuitive, and all other options were ultimately not considered due to many of them representing complicated quantitative variables or overly specific qualitative variables with many levels.

Training and Test Sets

A split of 70/30 is applied for the training and test sets. K-fold cross-validation is used in the training set to tune parameters.

```r
# Import libraries
library(splines)
library(ncvreg)
library(interp)
library(gam)
library(glmnet)
library(tree)
library(randomForest)
library(gbm)

# Train and Test Set Split
set.seed(13) # save results
n = length(price)
split = 0.7 # 70/30 Split
ii = sample(1:n, floor(split * n))
y_test = car_price[-ii,]$price
```

Particularly, for the Ridge and LASSO methods the tuning parameter lambda is first solved explicitly, with the selected value for lambda then being chosen to build the corresponding model. We form a subset of the data that removes columns *car_ID* and *CarName* in order to hold every predictor for the regularization process.

```r
# Regularization Setup
x = model.matrix(price ~ . -car_ID -CarName, car_price)[,-1] # remove ID & name
x_train = x[ii,]
y_train = car_price[ii,]$price
grid = 10^seq(10,-2,length = 100) # lambda candidates
cv_lambda_r = cv.glmnet(x_train, y_train, alpha=0, lambda=grid, data=car_price)
cv_lambda_l = cv.glmnet(x_train, y_train, alpha=1, lambda=grid, data=car_price)
```

Training the Models

Now that the data is ready to be used for training, we first create four models (linear regression, general additive model, LASSO and Ridge penalties) that assume linear form.

```r
linear_model = lm(price ~ horsepower + enginesize + citympg + curbweight,
                  subset = ((1:n)[ii]), data=car_price)
gam = gam(price ~ s(horsepower,4) + s(enginesize,4) + s(citympg,4) + curbweight
          + horsepower:enginesize:curbweight, data = car_price,
          subset = ((1:n)[ii]))
ridge_model = glmnet(x_train, y_train, alpha=0, lambda = cv_lambda_r$lambda.min)
lasso_model = glmnet(x_train, y_train, alpha=1, lambda = cv_lambda_l$lambda.min)
```

The linear model provides a straightforward fit of a hyperplane, and is included as a benchmark for comparison with other methods. The expectation is for our more flexible models to achieve a lower test error.

On the other hand, the general additive model introduces great flexibility. For this GAM, smoothing splines with four degrees of freedom are applied on all predictors but *curbweight*, as this feature did not benefit greatly from any non-parametric effects. An interaction is also added to account for dependence.

The Ridge and LASSO penalties on least squares regression aim to trade in a higher bias for a lower variance, and instead of implementing the penalty on just the four selected predictors, it is applied on all predictors from the data set. We proceed with this approach, as it helps to indicate if there are more important predictors from the data set that we should include for training when comparing the results to the linear model (at least for this linear assumption).

Next, we train the remaining three models that are based on tree ensemble methods.

```r
# Tree Ensemble Methods
B = 500 # average over 500 trees
m = 4 # number of predictors
boost_train = car_price[ii, c("price", "horsepower", "enginesize",
                              "citympg", "curbweight")] # boosting training set

bagging = randomForest(price ~ horsepower + enginesize + citympg + curbweight,
                       subset=((1:n)[ii]), data=car_price, mtry = m, ntree = B)
random_forest = randomForest(price ~ horsepower + enginesize + citympg +
                             curbweight, subset=((1:n)[ii]), data=car_price,
                             mtry = sqrt(m), ntree = B)
boosting = gbm(price ~ horsepower + enginesize + citympg + curbweight,
               data = boost_train, distribution = 'gaussian', n.trees = 5000,
               interaction.depth = 6, shrinkage = 0.01, cv.folds = 5)
bi = gbm.perf(boosting, method = "cv") # number of iterations chosen by cv
```

These methods (as well as the GAM) provide a non-parametric approach to the problem, yet there is still room for interpretability through variable importance measures that will be addressed very shortly.

Model Summaries

Before finally calculating predictions and test errors, model summaries are displayed to note a few key points.

```
# Summary of Models
summary(linear_model)
```

```
##
## Call:
## lm(formula = price ~ horsepower + enginesize + citympg + curbweight,
##     data = car_price, subset = ((1:n)[ii]))
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9387.8 -1746.8   242.4  1474.4 13049.1
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 13447.34     292.94  45.905  < 2e-16 ***
## horsepower   2260.32     627.80   3.600 0.000442 ***
## enginesize   3814.15     768.87   4.961 2.03e-06 ***
## citympg       -53.55     541.83  -0.099 0.921414
## curbweight   1728.01     717.36   2.409 0.017322 *
## ---
## Signif. codes:  0 ’***’ 0.001 ’**’ 0.01 ’*’ 0.05 ’.’ 0.1 ’ ’ 1
##
## Residual standard error: 3484 on 138 degrees of freedom
## Multiple R-squared:  0.8166, Adjusted R-squared:  0.8113
## F-statistic: 153.6 on 4 and 138 DF,  p-value: < 2.2e-16
```

```
summary(gam)
```

```
##
## Call: gam(formula = price ~ s(horsepower, 4) + s(enginesize, 4) + s(citympg,
##     4) + curbweight + horsepower:enginesize:curbweight, data = car_price,
##     subset = ((1:n)[ii]))
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -8554.56 -1104.49   -96.55   897.84 11135.75
##
## (Dispersion Parameter for gaussian family taken to be 7283372)
##
##     Null Deviance: 9132629139 on 142 degrees of freedom
## Residual Deviance: 932272405 on 128.0001 degrees of freedom
## AIC: 2681.528
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##                                    Df     Sum Sq    Mean Sq F value    Pr(>F)
## s(horsepower, 4)                    1 5973227987 5973227987 820.118 < 2.2e-16
## s(enginesize, 4)                    1 1089818114 1089818114 149.631 < 2.2e-16
## s(citympg, 4)                       1  506886693  506886693  69.595 9.998e-14
## curbweight                          1  330989522  330989522  45.444 4.836e-10
## curbweight:horsepower:enginesize    1  913238278  913238278 125.387 < 2.2e-16
## Residuals                         128  932272405    7283372
##
## s(horsepower, 4)                 ***
## s(enginesize, 4)                 ***
## s(citympg, 4)                    ***
## curbweight                       ***
## curbweight:horsepower:enginesize ***
## Residuals
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##                                  Npar Df Npar F     Pr(F)
## (Intercept)
## s(horsepower, 4)                       3 14.217 4.711e-08 ***
## s(enginesize, 4)                       3 15.743 8.974e-09 ***
## s(citympg, 4)                          3  6.941 0.0002295 ***
## curbweight
## curbweight:horsepower:enginesize
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

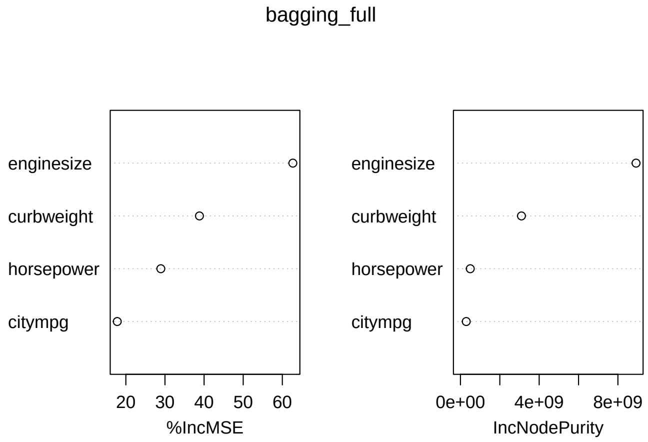
```
# coef(ridge_model) omitted from output, just shrinkage on full linear model
coef(lasso_model) # included in output here as it applies model selection
```

```
## 41 x 1 sparse Matrix of class "dgCMatrix"
##                                     s0
## (Intercept)              -1.286872e+04
## symboling                -2.881821e+02
## fueltype                  .
## aspiration                .
## doornumber               -3.366464e+02
## carbody                  -7.526280e+02
## drivewheel                6.897649e+02
## enginelocation            1.156196e+04
## wheelbase                 2.373057e+02
## carlength                 .
## carwidth                  .
## carheight                 4.501762e+02
## curbweight                4.531755e+02
## enginetype                .
## cylindernumber            4.602641e+02
## enginesize                .
## fuelsystem                2.856920e+02
## boreratio                 .
## stroke                   -5.441277e+01
## compressionratio          .
## horsepower                .
## peakrpm                      .
## citympg                    -1.701521e+02
## highwaympg                   .
## brand                      -1.716571e+02
## model                       7.158490e+00
## power_to_weight_ratio        .
## wheelbase_squared            .
## carlength_squared            .
## carwidth_squared            4.831132e+00
## carheight_squared            .
## curbweight_squared          3.518161e-04
## enginesize_squared          2.379828e-01
## boreratio_squared            .
## stroke_squared             -3.988921e+02
## compressionratio_squared  4.268447e+00
## horsepower_squared          7.320086e-02
## peakrpm_squared             1.040772e-04
## citympg_squared              .
## highwaympg_squared           .
## log_enginesize               .
```

```
bagging_full = randomForest(price ~ horsepower + enginesize + citympg +
                              curbweight, data = car_price, mtry = m,
                            importance = TRUE) # for variable importance
importance(bagging_full) # output numerical values
```

```
##              %IncMSE IncNodePurity
## horsepower 28.93992     498928574
## enginesize 62.69020    8921792640
## citympg    17.79842     294474807
## curbweight 38.81086    3100290353
```

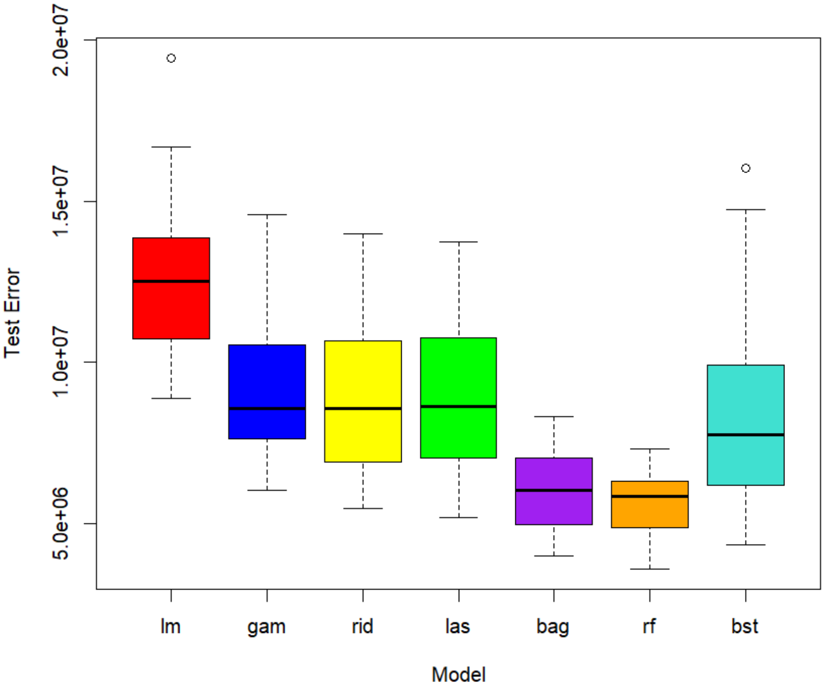
```
varImpPlot(bagging_full) # plot
```



According to the above outputs everything seems to be in order, with just a couple drawbacks. The LASSO penalty allows for model selection, and in this case it sets many coefficients to zero, which could in fact imply that the predictors we selected are a good enough choice (we later explore this possibility with the test error comparison). However, is quick to notice that *citympg* looks to be a redundant predictor, as its p-value is very high for the linear model output, and it does not increase the MSE much when removing it from the trees. In the conclusion we explain the verdict regarding this predictor, but for now it is included in all of the following test error calculations.

Generating the Test Errors

To compare test errors in a fair manner, thirty simulations are conducted and summarized through boxplots. This simultaneously presents the median test error values for each method while also showing the spread, how variable each issue of test error is.



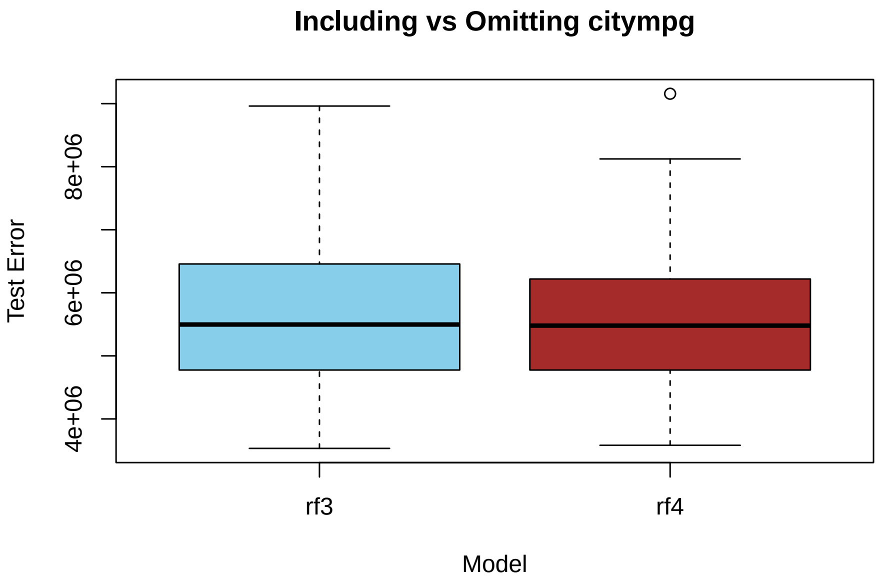
As expected, the linear model performs the worst due to its strict linear assumption that limits it from being able to accurately model the true relationship between the predictors and the response. The Ridge and LASSO penalties that take into account all features also perform as expected, yielding a lower test error than the linear model. However, the GAM that only contains four of the predictor performs on a similar level, hence it looks like it was indeed the right idea to leave out most of the features in preference of a simpler model.

Boosting led to a result that is highly variable, but performs slightly better than the linear methods. Bagging and the random forest instead provide the best results from this entire set. Bagging is simply a special case of random forests, where we do not randomly select a subset of predictors to split on for each node when growing each tree. Random forests provided the lowest test errors, it yielded the lowest median test error of 5840510. It appears that it was impactful to decorrelate the trees by allowing for unique splits.

Selecting the Model

Out of the models covered in this study, the random forest proves to be the right choice, since it tends to have lower test errors. Furthermore when looking back at the model summaries, it was obvious that *citympg* was not that useful. With more experimentation on the side it was revealed that random forests omitting this predictor yield similar results.

```r
MSE_random_forest_4 = rep(0,30)
MSE_random_forest_3 = rep(0,30)

for(i in (1:30)){
  ii = sample(1:n, floor(split * n))
  y_test = car_price[-ii,]$price

  random_forest_4 = randomForest(price ~ horsepower + enginesize + citympg +
                                curbweight, subset=((1:n)[ii]), data=car_price,
                                mtry = sqrt(m), ntree = B)
  random_forest_3 = randomForest(price ~ horsepower + enginesize +
                                curbweight, subset=((1:n)[ii]), data=car_price,
                                mtry = sqrt(m), ntree = B)

  y_hat_random_forest_4 = predict(random_forest_4, newdata = car_price[-ii,])
  y_hat_random_forest_3 = predict(random_forest_3, newdata = car_price[-ii,])

  # Test MSE
  MSE_random_forest_4[i] = mean((y_test - y_hat_random_forest_4)^2)
  MSE_random_forest_3[i] = mean((y_test - y_hat_random_forest_3)^2)
}

boxplot(MSE_random_forest_3, MSE_random_forest_4, names = c('rf3','rf4'),
        col = c('skyblue', 'brown'), xlab = "Model", ylab = "Test Error",
        main = "Including vs Omitting citympg")
```



```r
summary(MSE_random_forest_3)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## 3532639 4818628 5497034 5759130 6374269 8961546
```

```r
summary(MSE_random_forest_4)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## 3582071 4775657 5479651 5711898 6155628 9156807
```

The boxplots look identical with close medians and variances, so this implies that it is rational to drop *citympg*, and rather train a random forest with the other three predictors.

To see how accurate these predictions really are, we look at the residual values for a random forest with the three features.

```r
random_forest = randomForest(price ~ horsepower + enginesize +
                             curbweight, subset = ((1:n)[ii]), data=car_price,
                             mtry = sqrt(3), ntree = B)

y_hat_random_forest = predict(random_forest, newdata = car_price[-ii,])
print(y_test - y_hat_random_forest) # residuals
```

```
##           5           6          10          11          13          15 
##  1588.72362  3525.45880 -1036.71560  2703.38467  4805.84993  5526.13980 
##          18          19          32          33          35          37 
##    51.68295  -991.96963   329.68472  -705.01693    63.24315   158.53267 
##          38          40          45          59          60          63 
##  -938.12170  -209.50821  2306.04550  3867.77957  -822.22567   934.39117 
##          65          67          77          78          84          90 
##  1674.82667  6387.48410 -1058.67276  -509.94298   633.93775 -1082.61322 
##          93          99         104         107         109         110 
##  -277.68983   859.62756 -2455.56670  1166.38847 -4378.87170 -3393.96520 
##         112         113         116         120         121         126 
##   386.94913  -295.63600  1568.91067  -424.92274  -378.89926  6714.98173 
##         128         129         135         136         140         141 
##  3100.98200  6111.19588  2123.42440   797.69060  -688.46778  -864.32654 
##         143         144         145         147         149         150 
##   -87.93096  -955.32253  -998.65480 -1218.09091 -2286.13330 -1019.24997 
##         151         154         155         159         162         167 
## -1367.15571 -1026.40890  -324.85653  -161.01710   545.63448  -437.45250 
##         169         170         172         174         181         189 
##  -547.02110 -2490.21183 -1346.47494  -716.23349  -855.07493  -755.38444 
##         192         202 
##  -817.27197  1363.75338
```

```r
# % of correct predictions within 2k range
print(sum(abs((y_test - y_hat_random_forest)) < 2000)/((n - length(ii))))
```

```
## [1] 0.7419355
```

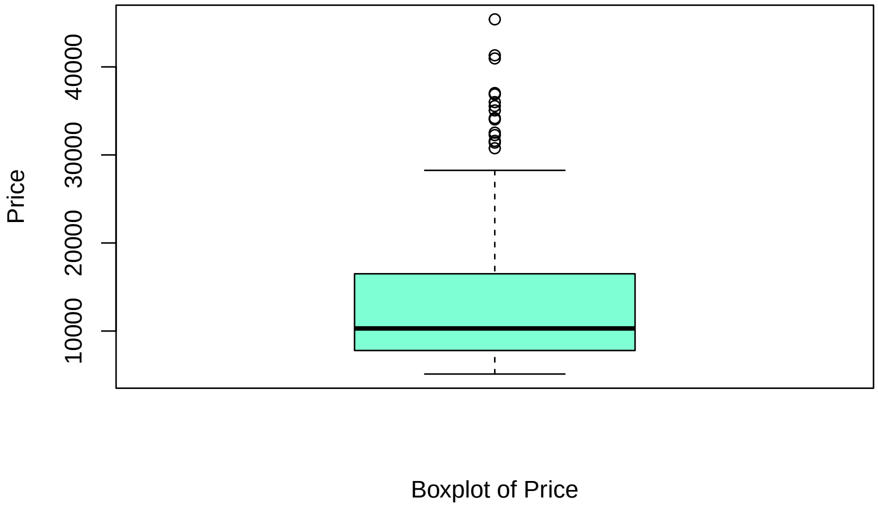
```
# % of correct predictions within 5k range
print(sum(abs((y_test - y_hat_random_forest)) < 5000)/((n - length(ii))))
```

```
## [1] 0.9354839
```

This random forest did a great job considering it is not the most complex model, it predicted the correct car price within a 2k range 74% of the time, and 94% of the time for the 5k range.

One caveat to note is that there are indeed a couple large residuals over the 5k mark, and upon further analysis this was due to the presence of some outliers and high leverage points in the data. These data points correspond to cars that are highly priced, it can be hypothesized first as well through a view of the distribution of price.

```
# response is right skewed, outliers have a high price
boxplot(price, xlab = "Boxplot of Price", ylab = "Price", col = "aquamarine")
```



These problematic observations could have been omitted from training, but in that case we would lose some valuable information, and it seems to be the case that some important predictors are missing from the data set. Additionally, more data could help reduce the variance of our predictions, since this is a smaller data set of 205 cars.

Conclusion

In the end, a random forest has been selected as the model for predicting car price. The model is trained on all 205 observations, so that it can generalize better on observations outside of the entire data set.

This model is not complex, it consists of three predictors, therefore it is not too difficult for one to use the three values of horsepower, engine size, and curb weight to reasonably predict what price the car would have. Inference was carried out, as it was clear that engine size was the most key feature, having the greatest impact on the pricing of cars. Henceforth, this model may be optimized by simply gathering more data which keeps its structure intact, or by examining and adding any important factors not included in the data set.