Part_2_Homework

Paola Carolina Suarez & Taylor Stone

2024-10-24

Part 2 Homework

Programmming in R

Universidad Carlos III de Madrid

For the development of the part 2 of the homework , we will use the data set "diamonds" provided by R, and also additional packages such as h2O and Caret to do testing and predictions of the variables.

```
library(ggplot2)
data(diamonds)
```

I.Using the variables of your dataset, apply the library caret or/and H2O for analyzing

```
#Load necessary packages

if (!require(h2o)) install.packages("h2o")
```

```
## Loading required package: h2o
```

```
##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit https://docs.h2o.ai
##
## ----------------------------------------------------------------------
```

```
##
## Attaching package: 'h2o'
```

```
## The following objects are masked from 'package:stats':
##
##     cor, sd, var

## The following objects are masked from 'package:base':
##
##     %*%, %in%, &&, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc
```

```r
library(h2o)
h2o.init()
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         9 hours 44 minutes
##     H2O cluster timezone:       Europe/Paris
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.44.0.3
##     H2O cluster version age:    10 months and 11 days
##     H2O cluster name:           H2O_started_from_R_cream_nhj536
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   7.20 GB
##     H2O cluster total cores:    16
##     H2O cluster allowed cores:  16
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.4.1 (2024-06-14 ucrt)

## Warning in h2o.clusterInfo():
## Your H2O cluster version is (10 months and 11 days) old. There may be a newer version available.
## Please download and install the latest version from: https://h2o-release.s3.amazonaws.com/h2o/latest_
```

```r
#caret package

if (!require(caret)) install.packages("caret")
```

```
## Loading required package: caret

## Loading required package: lattice
```

```r
if (!require(caretEnsemble)) install.packages("caretEnsemble")
```

```
## Loading required package: caretEnsemble
```

```r
if (!require(e1071)) install.packages("e1071")
```

```
## Loading required package: e1071
```

```r
if (!require(randomForest)) install.packages("randomForest")
```

```
## Loading required package: randomForest

## randomForest 4.7-1.2

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
if (!require(gbm)) install.packages("gbm")
```

```
## Loading required package: gbm

## Loaded gbm 2.2.2

## This version of gbm is no longer under development. Consider transitioning to gbm3, https://github.c
```

```r
library(caret)
library(caretEnsemble)
library(e1071)
library(randomForest)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following object is masked from 'package:randomForest':
##
##     combine

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

To create a dichotomous, or binary, variable from the "cut" variable, we group the original categories of "cut" into two broad classifications. The "cut" variable consists of multiple categories that describe the quality of the cut, such as Fair, Good, Very Good, Premium, and Ideal. The "Premium" category will include the "Premium" and "Very Good" values from the original "cut" variable and were grouped together to represent a higher quality of cut. The "Otherwise" group will include the remaining values—"Fair," "Good," and "Ideal." This group is defined to capture any quality that is not specifically Premium or Very Good.

```
#Categories of variable cut
unique(diamonds$cut)
```

```
## [1] Ideal     Premium   Good      Very Good Fair
## Levels: Fair < Good < Very Good < Premium < Ideal
```

```
#create a dichotomy variable
diamonds <- diamonds %>%
  mutate(cut_dichotomy = ifelse(cut %in% c("Premium", "Very Good"), 1, 0))


head(diamonds)
```

```
## # A tibble: 6 x 11
##   carat cut       color clarity depth table price     x     y     z cut_dichotomy
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>         <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43             0
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31             1
## 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31             0
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63             1
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75             0
## 6  0.24 Very Go~ J     VVS2     62.8    57   336  3.94  3.96  2.48             1
```

This following section prepares the diamonds dataset to be compatible with the H2O package by standardizing variable types. The transformation ensures that all categorical data is treated as unordered, which simplifies the compatibility with H2O's algorithms. Additionally, the code specifies that the variable cut_dichotomy should remain a factor to ensure it is correctly interpreted as categorical in further analysis. The code str(diamonds) is to confirm that these adjustments have been successfully applied.

```
#identify ordered factor variables existence
str(diamonds)
```

```
## tibble [53,940 x 11] (S3: tbl_df/tbl/data.frame)
##  $ carat        : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut          : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color        : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity      : Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth        : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table        : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
##  $ price        : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
##  $ x            : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y            : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z            : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
##  $ cut_dichotomy: num [1:53940] 0 1 0 1 0 1 1 1 0 1 ...
```

```r
#Convert ordered factor variables to a regular factor

diamonds[] <- lapply(diamonds, function(x) {
  if (is.ordered(x)) {
    return(as.factor(as.character(x)))
  } else {
    return(x)
  }
})


#maintain cut_dichotomy as a factor
diamonds <- diamonds %>%
  mutate(cut_dichotomy = factor(cut_dichotomy))
```

Converting the data into an H2O frame:

```r
#convert data to h2o frmae

diamonds.hex = as.h2o(diamonds)
```

```
##   |                                                                      |
```

```r
#describe
h2o.describe(diamonds.hex)
```

```
##            Label Type Missing Zeros PosInf NegInf   Min     Max         Mean
## 1          carat real       0     0      0      0   0.2    5.01    0.7979397
## 2            cut enum       0  1610      0      0   0.0    4.00           NA
## 3          color enum       0  6775      0      0   0.0    6.00           NA
## 4        clarity enum       0   741      0      0   0.0    7.00           NA
## 5          depth real       0     0      0      0  43.0   79.00   61.7494049
## 6          table real       0     0      0      0  43.0   95.00   57.4571839
## 7          price  int       0     0      0      0 326.0 18823.00 3932.7997219
## 8              x real       0     8      0      0   0.0   10.74    5.7311572
## 9              y real       0     7      0      0   0.0   58.90    5.7345260
## 10             z real       0    20      0      0   0.0   31.80    3.5387338
## 11 cut_dichotomy enum       0 28067      0      0   0.0    1.00    0.4796626
##           Sigma Cardinality
## 1     0.4740112          NA
## 2            NA           5
## 3            NA           7
## 4            NA           8
## 5     1.4326213          NA
## 6     2.2344906          NA
## 7  3989.4397381          NA
## 8     1.1217607          NA
## 9     1.1421347          NA
## 10    0.7056988          NA
## 11    0.4995908           2
```

This analysis checks to see how well the variables carat, depth, table, x, y, and z predict the binary outcome, cut_dichotomy, in diamonds. Using a binomial logistic regression model, with H2O, we find a low R^2 value of 26%, indicating the model does not explain much of the variance in the dichotomous variable. Due to this result, this model lacks predictive power for diamond cut quality, suggesting that additional or different predictors may be needed.

```r
model <- h2o.glm(
  x = c("carat", "depth", "table", "x", "y", "z"),
  y = "cut_dichotomy",
  training_frame = diamonds.hex,
  family = "binomial"
)
```

```
##   |                                                                              |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: glm
## Model ID:  GLM_model_R_1730373408053_34
## GLM Model: summary
##     family  link                                regularization
## 1 binomial logit Elastic Net (alpha = 0.5, lambda = 3.95E-4 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
## 1                          6                           4                    3
##         training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm coefficients
##       names coefficients standardized_coefficients
## 1 Intercept   -18.991090                 -0.065584
## 2     carat     0.279300                  0.132392
## 3     depth    -0.078472                 -0.112421
## 4     table     0.413701                  0.924411
## 5         x     0.000000                  0.000000
## 6         y    -0.038686                 -0.044184
## 7         z     0.000000                  0.000000
##
## H2OBinomialMetrics: glm
## ** Reported on training data. **
##
## MSE:  0.1990696
## RMSE:  0.4461721
## LogLoss:  0.6037643
## Mean Per-Class Error:  0.2239983
## AUC:  0.7787932
## AUCPR:  0.6863396
## Gini:  0.5575864
## R^2:  0.2024022
## Residual Deviance:  65134.09
## AIC:  65144.09
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
```

```
##             0     1    Error          Rate
## 0       22899  5168 0.184131   =5168/28067
## 1        6827 19046 0.263866   =6827/25873
## Totals 29726 24214 0.222377  =11995/53940
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold        value idx
## 1                       max f1  0.492962     0.760517 209
## 2                       max f2  0.206632     0.822834 357
## 3                 max f0point5  0.501328     0.777899 205
## 4                 max accuracy  0.495170     0.777846 208
## 5                max precision  0.524480     0.795294 192
## 6                   max recall  0.002634     1.000000 399
## 7              max specificity  0.995837     0.999252   0
## 8             max absolute_mcc  0.495170     0.554978 208
## 9   max min_per_class_accuracy  0.452778     0.760145 227
## 10 max mean_per_class_accuracy  0.495170     0.776131 208
## 11                     max tns  0.995837 28046.000000   0
## 12                     max fns  0.995837 25873.000000   0
## 13                     max fps  0.002634 28067.000000 399
## 14                     max tps  0.002634 25873.000000 399
## 15                     max tnr  0.995837     0.999252   0
## 16                     max fnr  0.995837     1.000000   0
## 17                     max fpr  0.002634     1.000000 399
## 18                     max tpr  0.002634     1.000000 399
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

In this portion, we modify the model to analyze how the numerical predictors– carat, depth, table, x, y, and z– relate to the different categories of the categorical variable color. The R^2 value from this model is significantly higher, with a value of 0.762, indicating the model explains about 76.2% of the variance in predicting the color variable. Although this fit is much stronger than the latter, there is still almost a quarter of the variance left unexplained.

The confusion matrix gives insight to how well the model classifies each color category. The matrix shows the counts of correctly and incorrectly printed classes for each actual color category, giving us a numerical visualization of where the model performs well and where it doesn't. For instance, the model appears to have high misclassification rates across most categories, particularly D and J with error values of 1.0000. The overall mean per-class error is approximately 0.825, indicating the model struggles to make accurate classifications for each color.

```r
model <- h2o.glm(
  x = c("carat", "depth", "table", "x", "y", "z"),
  y = "color",
  training_frame = diamonds.hex,
  family = "multinomial"    #we change the type model to use a categorical response
)
```

```
##   |                                                                            |
```

```r
print(model)
```

```
## Model Details:
```

```
## ==============
##
## H2OMultinomialModel: glm
## Model ID:  GLM_model_R_1730373408053_36
## GLM Model: summary
##        family        link                                regularization
## 1 multinomial multinomial Elastic Net (alpha = 0.5, lambda = 1.073E-4 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
## 1                         49                          40                    4
##        training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm multinomial coefficients
##       names coefs_class_0 coefs_class_1 coefs_class_2 coefs_class_3
## 1 Intercept     -1.610123     -0.190562     -0.011650      2.378624
## 2     carat     -1.097219     -0.752121     -0.401003      0.875942
## 3     depth     -0.024092     -0.047050     -0.035265     -0.014008
## 4     table      0.024826      0.040595      0.011687     -0.023835
## 5         x      0.034100     -0.199767     -0.016007     -0.098586
## 6         y     -0.001354      0.013406     -0.000367     -0.095130
## 7         z      0.084353      0.209325      0.076660     -0.341490
##   coefs_class_4 coefs_class_5 coefs_class_6 std_coefs_class_0 std_coefs_class_1
## 1     -2.474602     -1.561732     -7.179610         -2.060666         -1.690825
## 2      2.004367      2.605553      2.315328         -0.520094         -0.356514
## 3      0.036324      0.020059      0.040938         -0.034514         -0.067405
## 4      0.009121      0.003326      0.036758          0.055473          0.090709
## 5     -0.339915     -0.526160     -0.409649          0.038252         -0.224090
## 6      0.000000     -0.019535     -0.008395         -0.001546          0.015311
## 7     -0.494135     -0.314877      0.000000          0.059528          0.147720
##   std_coefs_class_2 std_coefs_class_3 std_coefs_class_4 std_coefs_class_5
## 1         -1.660315         -1.475942         -1.804909         -2.294718
## 2         -0.190080          0.415206          0.950093          1.235062
## 3         -0.050522         -0.020069          0.052038          0.028737
## 4          0.026114         -0.053260          0.020381          0.007433
## 5         -0.017956         -0.110590         -0.381303         -0.590225
## 6         -0.000419         -0.108652          0.000000         -0.022312
## 7          0.054099         -0.240989         -0.348710         -0.222209
##   std_coefs_class_6
## 1         -3.088154
## 2          1.097492
## 3          0.058648
## 4          0.082135
## 5         -0.459529
## 6         -0.009589
## 7          0.000000
##
## H2OMultinomialMetrics: glm
## ** Reported on training data. **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("diamonds_sid_923e_1")`
## MSE: (Extract with `h2o.mse`) 0.6877922
```

```
## RMSE: (Extract with `h2o.rmse`) 0.8293324
## Logloss: (Extract with `h2o.logloss`) 1.827182
## Mean Per-Class Error: 0.8254343
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Null Deviance: (Extract with `h2o.nulldeviance`) 202494.1
## Residual Deviance: (Extract with `h2o.residual_deviance`) 197116.4
## R^2: (Extract with `h2o.r2`) 0.7623143
## AIC: (Extract with `h2o.aic`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##        D     E   F     G    H    I J  Error             Rate
## D      0  2651  80  3690  306   48 0 1.0000 =   6,775 / 6,775
## E      0  3979  83  5173  484   77 1 0.5939 =   5,818 / 9,797
## F      0  3197  95  5509  616  124 1 0.9900 =   9,447 / 9,542
## G      0  3487  89  6556  912  248 0 0.4194 =  4,736 / 11,292
## H      0  2015  75  4693 1062  459 0 0.8721 =   7,242 / 8,304
## I      0  1101  49  2816  928  528 0 0.9026 =   4,894 / 5,422
## J      0   453  32  1362  570  391 0 1.0000 =   2,808 / 2,808
## Totals 0 16883 503 29799 4878 1875 2 0.7735 = 41,720 / 53,940
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-7 Hit Ratios:
##   k hit_ratio
## 1 1  0.226548
## 2 2  0.436911
## 3 3  0.614868
## 4 4  0.766351
## 5 5  0.881498
## 6 6  0.960586
## 7 7  1.000000
```

Taking into account the previous results, we want to simplify our model by including less variables to see if there is an improvement for the relation for the color variable.

Despite reducing the number of variables, the model's performance continues to lack the strength and accuracy needed to make confident predictions regarding diamond color. The R^2 value for the relaxed model is slightly lower than the latter, which is intuitive as we are removing variables that could have prediction relationships, however small they may be. This outcome suggests that the physical dimensions alone may not solely represent the factors influencing color, highlighting the need for additional or alternative variables that could better capture the relationships within.

```
model <- h2o.glm(
  x = c("x", "y", "z"),
  y = "color",
  training_frame = diamonds.hex,
  family = "multinomial"   #we change the type model to use a categorical response
)
```

```
##   |                                                                              |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: glm
## Model ID:  GLM_model_R_1730373408053_37
## GLM Model: summary
##        family        link                                 regularization
## 1 multinomial multinomial Elastic Net (alpha = 0.5, lambda = 1.035E-4 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
## 1                         28                          18                    3
##        training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm multinomial coefficients
##       names coefs_class_0 coefs_class_1 coefs_class_2 coefs_class_3
## 1 Intercept     -0.253302      0.148633     -0.905387     -1.065570
## 2         x     -0.072925     -0.212686     -0.033464     -0.027812
## 3         y     -0.084968      0.037546      0.018297     -0.024872
## 4         z     -0.257128     -0.238022     -0.194550     -0.039144
##   coefs_class_4 coefs_class_5 coefs_class_6 std_coefs_class_0 std_coefs_class_1
## 1     -2.836138     -4.388078     -6.534062         -2.068403         -1.697289
## 2      0.000000      0.283977      0.353049         -0.081804         -0.238583
## 3      0.000000      0.013262      0.000000         -0.097044          0.042883
## 4      0.281841      0.099017      0.386697         -0.181455         -0.167972
##   std_coefs_class_2 std_coefs_class_3 std_coefs_class_4 std_coefs_class_5
## 1         -1.680708         -1.506114         -1.838779         -2.334113
## 2         -0.037538         -0.031198          0.000000          0.318554
## 3          0.020898         -0.028407          0.000000          0.015147
## 4         -0.137294         -0.027624          0.198895          0.069876
##   std_coefs_class_6
## 1         -3.142265
## 2          0.396037
## 3          0.000000
## 4          0.272892
##
## H2OMultinomialMetrics: glm
## ** Reported on training data. **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("diamonds_sid_923e_1")`
## MSE: (Extract with `h2o.mse`) 0.691549
## RMSE: (Extract with `h2o.rmse`) 0.8315943
## Logloss: (Extract with `h2o.logloss`) 1.836315
## Mean Per-Class Error: 0.8364399
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Null Deviance: (Extract with `h2o.nulldeviance`) 202494.1
## Residual Deviance: (Extract with `h2o.residual_deviance`) 198101.7
## R^2: (Extract with `h2o.r2`) 0.761016
```

```
## AIC: (Extract with `h2o.aic`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##        D     E  F     G    H   I  J  Error              Rate
## D      0  2634  0  3795  335  10  1 1.0000 =   6,775 / 6,775
## E      0  3846  0  5377  547  25  2 0.6074 =   5,951 / 9,797
## F      0  3105  1  5695  709  31  1 0.9999 =   9,541 / 9,542
## G      0  3841  4  6244 1143  60  0 0.4470 =  5,048 / 11,292
## H      0  2270  5  4515 1377 131  6 0.8342 =   6,927 / 8,304
## I      0  1241  2  2722 1279 166 12 0.9694 =   5,256 / 5,422
## J      0   419  0  1418  840 123  8 0.9972 =   2,800 / 2,808
## Totals 0 17356 12 29766 6230 546 30 0.7842 = 42,298 / 53,940
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-7 Hit Ratios:
##   k hit_ratio
## 1 1  0.215832
## 2 2  0.428606
## 3 3  0.613367
## 4 4  0.764850
## 5 5  0.875714
## 6 6  0.960660
## 7 7  1.000000
```

To see how the variables relate to other categorical variables, we can change the predicted variable to cut and check for any improvements. Using the h2o.glm function, we build a multinomial logistic regression model to assess how well these predictors explain the variance in cut. The model shows some improvement in performance compared to previous attempts to predict color, indicating a more promising relationship. The R^2 value of this model indicates these three variables predict about 64.4% of the variance in the cut variable. This value is good, but not enough to form optimistic conclusions on the strength.

```r
model <- h2o.glm(
  x = c("carat", "depth", "table"),
  y = "cut",
  training_frame = diamonds.hex,
  family = "multinomial"   #we change the type model to use a categorical response
)
```

```
##   |                                                                              |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: glm
## Model ID:  GLM_model_R_1730373408053_38
## GLM Model: summary
##        family        link                               regularization
## 1 multinomial multinomial Elastic Net (alpha = 0.5, lambda = 5.384E-4 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
```

```
## 1                          20                           15                     5
##        training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm multinomial coefficients
##       names coefs_class_0 coefs_class_1 coefs_class_2 coefs_class_3
## 1 Intercept   -116.986607    -65.391773     65.009758    -15.868226
## 2     carat      0.384385     -0.071755     -0.153704      0.233963
## 3     depth      1.244608      0.612940     -0.369633     -0.019432
## 4     table      0.611457      0.437645     -0.751250      0.273035
##   coefs_class_4 std_coefs_class_0 std_coefs_class_1 std_coefs_class_2
## 1    -19.675122         -4.693487         -2.454474         -1.102178
## 2     -0.066945          0.182203         -0.034013         -0.072858
## 3      0.169968          1.783052          0.878111         -0.529544
## 4      0.140502          1.366295          0.977914         -1.678660
##   std_coefs_class_3 std_coefs_class_4
## 1         -1.193634         -1.160255
## 2          0.110901         -0.031733
## 3         -0.027839          0.243500
## 4          0.610094          0.313951
##
## H2OMultinomialMetrics: glm
## ** Reported on training data. **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("diamonds_sid_923e_1")`
## MSE: (Extract with `h2o.mse`) 0.3758187
## RMSE: (Extract with `h2o.rmse`) 0.6130405
## Logloss: (Extract with `h2o.logloss`) 1.071721
## Mean Per-Class Error: 0.5162363
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Null Deviance: (Extract with `h2o.nulldeviance`) 148145.8
## Residual Deviance: (Extract with `h2o.residual_deviance`) 115624.6
## R^2: (Extract with `h2o.r2`) 0.6441662
## AIC: (Extract with `h2o.aic`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error              Rate
## Fair       767  187   201     288       167 0.5236 =     843 / 1,610
## Good       153  653  1312    1470      1318 0.8669 =   4,253 / 4,906
## Ideal        5    8 19759     986       793 0.0832 =  1,792 / 21,551
## Premium      0   56  2361    9112      2262 0.3393 =  4,679 / 13,791
## Very Good    6  273  4528    4475      2800 0.7683 =  9,282 / 12,082
## Totals     931 1177 28161   16331      7340 0.3865 = 20,849 / 53,940
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.613478
```

```
## 2 2  0.847367
## 3 3  0.944512
## 4 4  0.992584
## 5 5  1.000000
```

Keeping cut as the categorical variable to be predicted, we decided to bring back the other three numerical variables, x, y, and z, into the model to see if the relationship improves.

The new R^2 value is 0.663, meaning the model with the remaining three numerical variables predicts 2% more of the variance of cut than the previous model. Although it is an improvement, this value is still not significant to conclude strong prediction relationships.

```r
model <- h2o.glm(
  x = c("carat", "depth", "table", "x", "y","z"),
  y = "cut",
  training_frame = diamonds.hex,
  family = "multinomial"   #we change the type model to use a categorical response
)
```

```
##   |                                                                          |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: glm
## Model ID:  GLM_model_R_1730373408053_39
## GLM Model: summary
##        family        link                                regularization
## 1 multinomial multinomial Elastic Net (alpha = 0.5, lambda = 5.384E-4 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
## 1                         35                          28                    5
##        training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm multinomial coefficients
##       names coefs_class_0 coefs_class_1 coefs_class_2 coefs_class_3
## 1 Intercept   -122.748311    -73.255153     60.859005    -18.833981
## 2     carat     -0.106213     -0.487505      0.071675      0.419367
## 3     depth      1.272552      0.676681     -0.344669      0.001958
## 4     table      0.664531      0.494999     -0.697201      0.312936
## 5         x      0.719787     -0.532597     -0.549452      4.562624
## 6         y     -0.466074      0.927825      0.446376     -4.593546
## 7         z      0.000000     -0.345650      0.000000     -0.152582
##   coefs_class_4 std_coefs_class_0 std_coefs_class_1 std_coefs_class_2
## 1    -23.908856         -4.619151         -2.373205         -1.015337
## 2     -0.008030         -0.050346         -0.231083          0.033975
## 3      0.182521          1.823086          0.969427         -0.493780
## 4      0.203203          1.484887          1.106070         -1.557888
## 5     -3.156762          0.807429         -0.597446         -0.616353
## 6      2.805793         -0.532319          1.059701          0.509821
## 7      0.532263          0.000000         -0.243925          0.000000
```

```
##   std_coefs_class_3 std_coefs_class_4
## 1         -1.130703         -1.087654
## 2          0.198784         -0.003806
## 3          0.002805          0.261484
## 4          0.699252          0.454056
## 5          5.118173         -3.541131
## 6         -5.246448          3.204593
## 7         -0.107677          0.375618
##
## H2OMultinomialMetrics: glm
## ** Reported on training data. **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("diamonds_sid_923e_1")`
## MSE: (Extract with `h2o.mse`) 0.355871
## RMSE: (Extract with `h2o.rmse`) 0.5965492
## Logloss: (Extract with `h2o.logloss`) 1.026107
## Mean Per-Class Error: 0.4821073
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Null Deviance: (Extract with `h2o.nulldeviance`) 148145.8
## Residual Deviance: (Extract with `h2o.residual_deviance`) 111443.4
## R^2: (Extract with `h2o.r2`) 0.6630532
## AIC: (Extract with `h2o.aic`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error              Rate
## Fair       764  179   204     329       134 0.5255 =     846 / 1,610
## Good       138  660  1298    1393      1417 0.8655 =   4,246 / 4,906
## Ideal        6    9 19569     908      1059 0.0920 =  1,982 / 21,551
## Premium      0   20  2037    9826      1908 0.2875 =  3,965 / 13,791
## Very Good    7  235  4533    2959      4348 0.6401 =  7,734 / 12,082
## Totals     915 1103 27641   15415      8866 0.3480 = 18,773 / 53,940
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.651965
## 2 2  0.872673
## 3 3  0.946070
## 4 4  0.992232
## 5 5  1.000000
```

Next, we created the correlation plot and matrix to explore the relationships between numerical variables in the dataset. The "?" in the plot indicate when a relationship with a categorical variable was tested, and those values can not be found without a numerical assignment.

The dark blue colors indicate strong positive correlation, whereas the peach colors indicate negative correlation. The lighter the color, the weaker the correlation.

There exists different correlation relationships within several pairings: 1. Carat has strong positive correlations with price, x, y, and z (and vice versa). These relationships indicate the higher the carat, the higher the dimensions and price. 2. Price has strong positive correlations with x, y, and z (and vice versa). Similarly to Carat, with higher price we can expect higher dimensions. 3. x, y, and z all have strong positive correlations with each other. This relationship is intuitive as all three variables represent physical measurements of the diamonds. The higher one is, we are likely to see an increase in the other two. 4. Depth and Table have a mild negative correlation. This relationship indicates that as the depth of the diamond increases, the table decreases.

```
cor_matrix <- h2o.cor(diamonds.hex)
print(cor_matrix)
```

```
##         carat cut color clarity       depth      table       price           x
## 1  1.00000000 NaN   NaN     NaN  0.02822431  0.1816175  0.92159130  0.97509423
## 2         NaN NaN   NaN     NaN         NaN        NaN         NaN         NaN
## 3         NaN NaN   NaN     NaN         NaN        NaN         NaN         NaN
## 4         NaN NaN   NaN     NaN         NaN        NaN         NaN         NaN
## 5  0.02822431 NaN   NaN     NaN  1.00000000 -0.2957785 -0.01064740 -0.02528925
## 6  0.18161755 NaN   NaN     NaN -0.29577852  1.0000000  0.12713390  0.19534428
## 7  0.92159130 NaN   NaN     NaN -0.01064740  0.1271339  1.00000000  0.88443516
## 8  0.97509423 NaN   NaN     NaN -0.02528925  0.1953443  0.88443516  1.00000000
## 9  0.95172220 NaN   NaN     NaN -0.02934067  0.1837601  0.86542090  0.97470148
## 10 0.95338738 NaN   NaN     NaN  0.09492388  0.1509287  0.86124944  0.97077180
## 11 0.10948976 NaN   NaN     NaN -0.15160685  0.3953228  0.08907288  0.11455188
##              y          z cut_dichotomy
## 1   0.95172220 0.95338738    0.10948976
## 2          NaN        NaN           NaN
## 3          NaN        NaN           NaN
## 4          NaN        NaN           NaN
## 5  -0.02934067 0.09492388   -0.15160685
## 6   0.18376015 0.15092869    0.39532281
## 7   0.86542090 0.86124944    0.08907288
## 8   0.97470148 0.97077180    0.11455188
## 9   1.00000000 0.95200572    0.10819169
## 10  0.95200572 1.00000000    0.09198991
## 11  0.10819169 0.09198991    1.00000000
```

```
# Load library to be able to make the plot
library(corrplot)
```

```
## corrplot 0.94 loaded
```

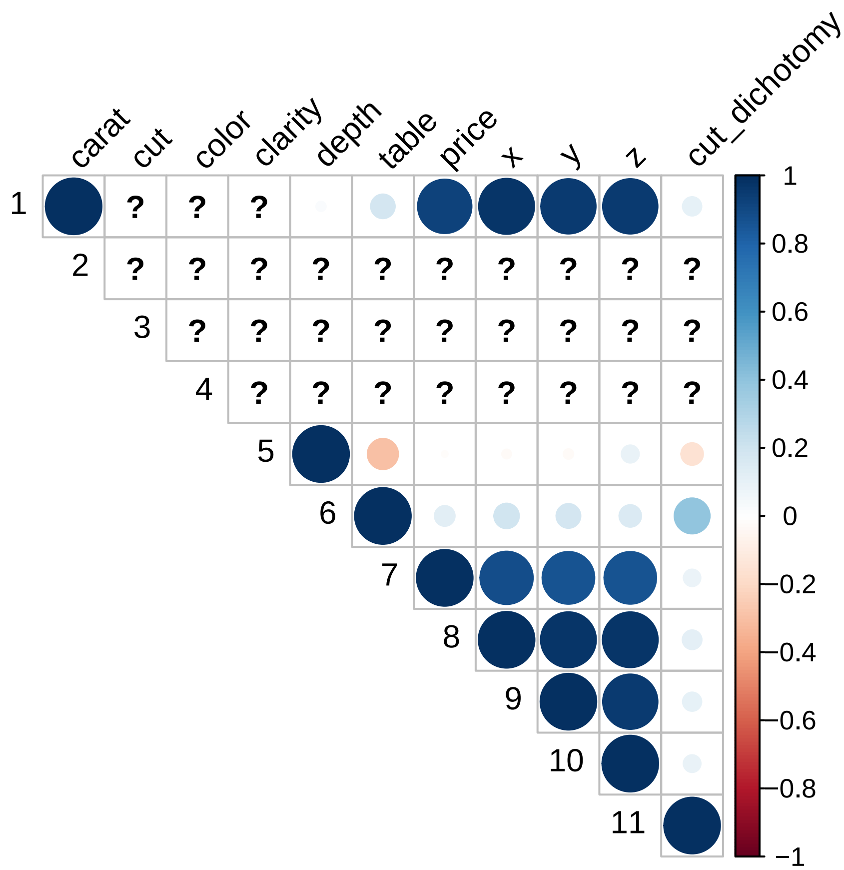
```
str(cor_matrix)
```

```
## 'data.frame':    11 obs. of  11 variables:
##  $ carat        : num  1 NaN NaN NaN 0.0282 ...
##  $ cut          : num  NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
##  $ color        : num  NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
##  $ clarity      : num  NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN ...
##  $ depth        : num  0.0282 NaN NaN NaN 1 ...
##  $ table        : num  0.182 NaN NaN NaN -0.296 ...
##  $ price        : num  0.9216 NaN NaN NaN -0.0106 ...
##  $ x            : num  0.9751 NaN NaN NaN -0.0253 ...
```

```
##  $ y           : num  0.9517 NaN NaN NaN -0.0293 ...
##  $ z           : num  0.9534 NaN NaN NaN 0.0949 ...
##  $ cut_dichotomy: num  0.109 NaN NaN NaN -0.152 ...
```

```r
cor_matrix <- as.matrix(cor_matrix)

# Correlation plot
corrplot(cor_matrix, method = "circle", type = "upper", tl.col = "black", tl.srt = 45)
```



We use the following code for numerical variables to extract a specific correlation value of interest, that is, between depth and price. Returned is a value of -0.10, indicating a very mild negative correlation between the two. This may indicate that, as depth increases, we would expect the price to ever so slightly decrease. However, this relationship is not very strong.

```r
if (!require(datarium)) install.packages("datarium")
```

```
## Loading required package: datarium
```

```r
library(datarium)

cor(diamonds$depth, diamonds$price)
```

```
## [1] -0.0106474
```

Since price had such strong correlations with several of the numerical variables, we had an interest to see how well the model would predict it.

Despite this, we found an R^2 value of 0.614, indicating the numerical variables only predict about 61.4% of the variance in the price variable. In the correlation plot from earlier, we did notice very weak correlations with variables depth, table, and the dichotomous cut value, so this result reflects the loss of coverage from those variables.

```r
model <- h2o.glm(
  x = c("carat", "depth", "table", "x", "y", "z"),
  y = "price",
  training_frame = diamonds.hex,
  family = "gaussian" #in this case will be a gaussian model
)
```

```
##   |                                                                          |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2ORegressionModel: glm
## Model ID:  GLM_model_R_1730373408053_40
## GLM Model: summary
##     family     link                              regularization
## 1 gaussian identity Elastic Net (alpha = 0.5, lambda = 7.3531 )
##   number_of_predictors_total number_of_active_predictors number_of_iterations
## 1                          6                           6                    1
##        training_frame
## 1 diamonds_sid_923e_1
##
## Coefficients: glm coefficients
##       names coefficients standardized_coefficients
## 1 Intercept -4118.087100               3932.799722
## 2     carat  1068.886797                506.664361
## 3     depth    -8.913323                -12.769417
## 4     table    16.140217                 36.065162
## 5         x   411.359811                461.447288
## 6         y   390.934504                446.499853
## 7         z   627.797656                443.036082
##
## H2ORegressionMetrics: glm
## ** Reported on training data. **
##
## MSE:  6148173
## RMSE:  2479.551
## MAE:  1718.844
## RMSLE:  NaN
## Mean Residual Deviance :  6148173
## R^2 :  0.613695
## Null Deviance :858473135517
## Null D.o.F. :53939
## Residual Deviance :331632477674
## Residual D.o.F. :53933
## AIC :996263.1
```

II. Split the Data using techniques in Carat/H2O

In the following section, we performed data splitting on the diamonds.hex dataset to create training and testing subsets for model evaluation. We used the h2o.splitFrame function to randomly partition the data, allocating 80% of the rows to the training set and the remaining 20% to the testing set. The seed is set to ensure reproducibility of the results. After splitting the data, we check and print the number of rows in both the training and testing sets to confirm that the split has been done correctly.

```
splits = h2o.splitFrame(data = diamonds.hex, ratios = c(0.8), seed = 198)
train = splits[[1]]
test = splits[[2]]


#check number of rows in train set and test set
#Original set has 53940 rows

print(paste0("Number of rows in train set: ", h2o.nrow(train)))
```

```
## [1] "Number of rows in train set: 43160"
```

```
print(paste0("Number of rows in test set: ", h2o.nrow(test)))
```

```
## [1] "Number of rows in test set: 10780"
```

III. Apply tecniques

FIRST TECNIQUE

"Random Forest with H2o"

In the following code, we implement a technique called Random Forest using the H2O package. We chose the numerical predictors from the model to be the predictors and the variable "cut" to be the variable of prediction.

```
rf = h2o.randomForest(x = c("depth", "table", "x", "y", "z", "carat"),
                      y = c("cut"), training_frame = train, model_id = "our.rf",
                      seed = 1234)
```

```
##   |                                                                              |
```

```
print(rf)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: drf
## Model ID:  our.rf
## Model Summary:
```

```
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1              50                      250             8461390        20
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        20   20.00000        364       5113  2693.16800
##
##
## H2OMultinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("RTMP_sid_923e_4")`
## MSE: (Extract with `h2o.mse`) 0.184556
## RMSE: (Extract with `h2o.rmse`) 0.4295999
## Logloss: (Extract with `h2o.logloss`) 0.8203084
## Mean Per-Class Error: 0.2322226
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## R^2: (Extract with `h2o.r2`) 0.8259413
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair      1158   95    11      33        19 0.1201 =    158 / 1,316
## Good       107 2775    54     164       838 0.2953 =  1,163 / 3,938
## Ideal       15   31 15606     773       779 0.0929 = 1,598 / 17,204
## Premium      2   44  1196    9123       680 0.1740 = 1,922 / 11,045
## Very Good    8  601  2174    1841      5033 0.4788 =  4,624 / 9,657
## Totals    1290 3546 19041   11934      7349 0.2193 = 9,465 / 43,160
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.780700
## 2 2  0.942516
## 3 3  0.988160
## 4 4  0.994045
## 5 5  1.000000
```

Below are the results when we model cut by the numerical variables using Random Forest technique. We find the R^2 value to be 0.824, which is fairly strong as this means this model predicts about 82.4% of the variance in the cut variable.

Looking at the confusion matrix, all errors are below 0.5, meaning the majority of the cut classes across all predictor variables were predicted correctly. The average was found to be only about 0.22, which is much better than the previous models.

```r
#option 1 test data

rf_perf1 = h2o.performance(model = rf, newdata = test)
print(rf_perf1)
```

```
## H2OMultinomialMetrics: drf
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.1834532
## RMSE: (Extract with `h2o.rmse`) 0.4283144
## Logloss: (Extract with `h2o.logloss`) 0.6847535
## Mean Per-Class Error: 0.2333504
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## R^2: (Extract with `h2o.r2`) 0.8235178
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair       262   19     2       8         3 0.1088 =       32 / 294
## Good        32  668    16      33       219 0.3099 =      300 / 968
## Ideal        1    9  3930     212       195 0.0959 =    417 / 4,347
## Premium      0   11   308    2252       175 0.1799 =    494 / 2,746
## Very Good    2  156   509     478      1280 0.4722 =  1,145 / 2,425
## Totals     297  863  4765    2983      1872 0.2215 = 2,388 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.778479
## 2 2  0.943414
## 3 3  0.990445
## 4 4  0.996475
## 5 5  1.000000
```

Confusion matrix

```r
confusion_matrix <- h2o.confusionMatrix(rf_perf1)
print(confusion_matrix)
```

```
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair       262   19     2       8         3 0.1088 =       32 / 294
## Good        32  668    16      33       219 0.3099 =      300 / 968
## Ideal        1    9  3930     212       195 0.0959 =    417 / 4,347
## Premium      0   11   308    2252       175 0.1799 =    494 / 2,746
## Very Good    2  156   509     478      1280 0.4722 =  1,145 / 2,425
## Totals     297  863  4765    2983      1872 0.2215 = 2,388 / 10,780
```

Prediction with test data

The table below shows the prediction probabilities for each cut category. The first column indicates the predicted cut categories in the test set. The remaining columns show the predicted probabilities for each cut category. These values represent the model's confidence in the predictions. For example, given that the values in the cells represent the likelihood that a given data point falls into each cut category, the first diamond is predicted by the model to have a "Good" cut with a probability of 74.8%.

Interpreting the values: A higher probability, a value close to 1, for an individual category indicates strong confidence in that prediction. If a diamond has multiple probabilities close to each other, this may suggest the model's uncertainty in the prediction.

```r
predictions = h2o.predict(rf, test)
```

```
##   |                                                                              |
```

```r
print(predictions)
```

```
##     predict Fair         Good       Ideal    Premium  Very Good
## 1      Good    0 7.475095e-01 0.018501022 0.00000000 0.23398947
## 2      Good    0 9.656937e-01 0.002143508 0.00000000 0.03216284
## 3     Ideal    0 1.303461e-01 0.550483309 0.01984964 0.29932094
## 4     Ideal    0 8.346647e-02 0.556828520 0.03655460 0.32315041
## 5 Very Good    0 1.270023e-03 0.026504825 0.00000000 0.97222515
## 6 Very Good    0 8.890275e-05 0.000000000 0.02871502 0.97119608
##
## [10780 rows x 6 columns]
```

Reduced Model

Here, we are creating a new random forest, keeping cut as the variable to be predicted, but taking away three of the six numerical variables, price, depth, and table.

The model gives us an R^2 value of 0.696, which is almost 12% less variability being explained than in the previous model. This is likely due to the fact that, although there was not a strong correlation with cut and the three removed variables, they still had some proportion of prediction for the variable.

Comparing the error rates, the values are ever so slightly higher in this model, indicating the variables x, y, and z do not predict the cut variable as accurately as they do when table, depth, and price are taken into account in the model. The total error rate is 0.3655, compared to the earlier error value of 0.22, indicating that the model with more variables predicted cut more accurately.

```r
## option 2
rf = h2o.randomForest(x = c("x", "y", "z"),
                      y = c("cut"), training_frame = train, model_id = "our.rf",
                      seed = 1234)
```

```
##   |                                                                              |
```

```r
print(rf)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: drf
## Model ID:  our.rf
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1              50                      250             7184569        20
```

```
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        20   20.00000        462       4226  2286.16800
##
##
## H2OMultinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
##
## Training Set Metrics:
## =====================
##
## Extract training frame with `h2o.getFrame("RTMP_sid_923e_4")`
## MSE: (Extract with `h2o.mse`) 0.3218141
## RMSE: (Extract with `h2o.rmse`) 0.5672867
## Logloss: (Extract with `h2o.logloss`) 0.9251122
## Mean Per-Class Error: 0.383534
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## R^2: (Extract with `h2o.r2`) 0.6964902
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error              Rate
## Fair       936  196    20     126        38 0.2888 =     380 / 1,316
## Good       118 2127   257     268      1168 0.4599 =   1,811 / 3,938
## Ideal       16   53 13393    2432      1310 0.2215 =  3,811 / 17,204
## Premium     12   52  4152    6105       724 0.4473 =  4,940 / 11,045
## Very Good   12  628  3382     809      4826 0.5003 =   4,831 / 9,657
## Totals    1094 3056 21204    9740      8066 0.3655 = 15,773 / 43,160
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.634546
## 2 2  0.884245
## 3 3  0.972961
## 4 4  0.995088
## 5 5  1.000000
```

Reduced Model Test Set:

```
rf_perf2 = h2o.performance(model = rf, newdata = test)
print(rf_perf2)
```

```
## H2OMultinomialMetrics: drf
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.3200809
## RMSE: (Extract with `h2o.rmse`) 0.5657569
## Logloss: (Extract with `h2o.logloss`) 0.9022237
```

```
## Mean Per-Class Error: 0.377806
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## R^2: (Extract with `h2o.r2`) 0.6920818
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair       216   40     5      28         5 0.2653 =       78 / 294
## Good        29  524    67      56       292 0.4587 =      444 / 968
## Ideal        2   13  3408     600       324 0.2160 =    939 / 4,347
## Premium      3   12  1069    1492       170 0.4567 =  1,254 / 2,746
## Very Good    2  156   844     192      1231 0.4924 =  1,194 / 2,425
## Totals     252  745  5393    2368      2022 0.3626 = 3,909 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.637384
## 2 2  0.888590
## 3 3  0.974212
## 4 4  0.995918
## 5 5  1.000000
```

Confusion matrix:

```r
confusion_matrix <- h2o.confusionMatrix(rf_perf2)
print(confusion_matrix)
```

```
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair       216   40     5      28         5 0.2653 =       78 / 294
## Good        29  524    67      56       292 0.4587 =      444 / 968
## Ideal        2   13  3408     600       324 0.2160 =    939 / 4,347
## Premium      3   12  1069    1492       170 0.4567 =  1,254 / 2,746
## Very Good    2  156   844     192      1231 0.4924 =  1,194 / 2,425
## Totals     252  745  5393    2368      2022 0.3626 = 3,909 / 10,780
```

Prediction with test data

The prediction table reveals insightful results about the strength of this model in the confidence it has in its own predictions. It is interesting that the "Fair" category has extremely low probabilities, indicating the model has extremely little confidence in the predictions for that category of cuts. The "Very Good" category has the highest levels of probability across all rows, indicating the model has high confidence in the predictions for this category.

```r
predictions = h2o.predict(rf, test)
```

```
##   |                                                                              |
```

```r
print(predictions)
```

```
##      predict         Fair      Good      Ideal    Premium Very Good
## 1       Good 0.0008680226 0.60447959 0.10821445 0.05244203 0.2339959
## 2       Good 0.0009011650 0.58192390 0.15134800 0.06250504 0.2033219
## 3 Very Good 0.0007309412 0.03252362 0.16467690 0.02876680 0.7733017
## 4 Very Good 0.0007059754 0.03547889 0.15746905 0.02728595 0.7790601
## 5 Very Good 0.0007331174 0.02615674 0.08362763 0.02433166 0.8651509
## 6 Very Good 0.0006610895 0.18199411 0.11871624 0.02764195 0.6709866
##
## [10780 rows x 6 columns]
```

SECOND TECHNIQUE

"Neural Network"

To run a new technique, called Neural Network, we first define the variables for predictors and one for response. Then, we run the model with the training data set.

The difference between neural networks and random forests is that neural networks are flexible models that excel at capturing complex patterns in high-dimensional data. Random forests are ensemble methods that build multiple decision trees to improve accuracy and interpretability, performing well on structured data with less need for extensive tuning.

```r
# Set the predictor and response variables
# Define specific predictor column names
predictors <- c("depth", "table", "x", "y", "z", "carat")
response = "cut" # Cut is our response

# Define and train the neural network model
model = h2o.deeplearning(
  x = predictors,
  y = response,
  training_frame = train,
  validation_frame = test,
  activation = "RectifierWithDropout", # Activation function
  hidden = c(10, 10),
  epochs = 100, # Number of training epochs
  rate = 0.01, # Learning rate
  input_dropout_ratio = 0.2, # Input dropout ratio
  hidden_dropout_ratios = c(0.5, 0.5) # Dropout ratios for each hidden layer
)
```

```
## Warning in .h2o.processResponseWarnings(res): rate cannot be specified if adaptive_rate is enabled..

##   |                                                                              |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: deeplearning
## Model ID:  DeepLearning_model_R_1730373408053_41
## Status of Neuron Layers: predicting cut, 5-class classification, multinomial distribution, CrossEntr
##   layer units             type dropout       l1       l2 mean_rate rate_rms
## 1     1     6            Input 20.00 %       NA       NA        NA       NA
## 2     2    10 RectifierDropout 50.00 % 0.000000 0.000000  0.000839 0.000441
## 3     3    10 RectifierDropout 50.00 % 0.000000 0.000000  0.001109 0.000964
## 4     4     5          Softmax      NA 0.000000 0.000000  0.003700 0.002953
##   momentum mean_weight weight_rms mean_bias bias_rms
## 1       NA          NA         NA        NA       NA
## 2 0.000000   -0.068971   0.396350  0.239330 0.246824
## 3 0.000000   -0.100792   0.324020  0.248506 0.264572
## 4 0.000000   -0.146622   1.386363 -0.854034 1.148264
##
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 10103 samples **
##
## Training Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.3968287
## RMSE: (Extract with `h2o.rmse`) 0.6299434
## Logloss: (Extract with `h2o.logloss`) 1.099913
## Mean Per-Class Error: 0.635469
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error            Rate
## Fair         4   19   110     110        69 0.9872 =      308 / 312
## Good         0    0   363     506        58 1.0000 =      927 / 927
## Ideal        0    0  3843     239         0 0.0585 =    239 / 4,082
## Premium      0    0   352    2155         0 0.1404 =    352 / 2,507
## Very Good    0    0   996    1259        20 0.9912 =  2,255 / 2,275
## Totals       4   19  5664    4269       147 0.4039 = 4,081 / 10,103
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.596061
## 2 2  0.805701
## 3 3  0.914778
## 4 4  0.977828
## 5 5  1.000000
##
##
##
```

```
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on validation data. **
## ** Metrics reported on full validation frame **
##
## Validation Set Metrics:
## =====================
##
## Extract validation frame with `h2o.getFrame("RTMP_sid_923e_6")`
## MSE: (Extract with `h2o.mse`) 0.3942737
## RMSE: (Extract with `h2o.rmse`) 0.6279122
## Logloss: (Extract with `h2o.logloss`) 1.091131
## Mean Per-Class Error: 0.6349678
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error            Rate
## Fair         4   11   111     108        60 0.9864 =      290 / 294
## Good         0    0   383     537        48 1.0000 =      968 / 968
## Ideal        0    0  4097     249         1 0.0575 =    250 / 4,347
## Premium      0    1   371    2374         0 0.1355 =    372 / 2,746
## Very Good    0    0  1049    1365        11 0.9955 =  2,414 / 2,425
## Totals       4   12  6011    4633       120 0.3983 = 4,294 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.601670
## 2 2  0.807699
## 3 3  0.918089
## 4 4  0.979406
## 5 5  1.000000
```

Evaluating the model's performance on the test data set, we observe various extremely high error rates, with two being 1.000. This indicates that, for fair and good, they were not predicted right at all by our model. This is a concerning result, as we expect some proportion to be predicted. The total error percentage is almost 40%, indicating a large amount of predictions made by the model were inaccurate.

The output does not give us an R^2, which is the value representing the percentage of variance explained by the predictions, but we can use the value of the Root Mean Squared Error, or RMSE. In this model, the RMSE is 0.615. RMSE measures the average magnitude of the errors between predicted values and actual values, providing a measure of how well the model's predictions match the observed data. A Root Mean Squared Error of 0.6152 indicates that, on average, the predictions made by the model deviate from the actual values by approximately 0.6152 units of our variable. This value is typically slightly less than the R^2, so we can expect our R^2 to be around this number, indicating that at least the majority of variance of the variable "cut" is predicted.

```r
# Evaluate the model performance on the test set
perf = h2o.performance(model, newdata = test)

# Print the model performance
print(perf)
```

```
## H2OMultinomialMetrics: deeplearning
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.3942737
## RMSE: (Extract with `h2o.rmse`) 0.6279122
## Logloss: (Extract with `h2o.logloss`) 1.091131
## Mean Per-Class Error: 0.6349678
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error             Rate
## Fair         4   11   111     108        60 0.9864 =      290 / 294
## Good         0    0   383     537        48 1.0000 =      968 / 968
## Ideal        0    0  4097     249         1 0.0575 =    250 / 4,347
## Premium      0    1   371    2374         0 0.1355 =    372 / 2,746
## Very Good    0    0  1049    1365        11 0.9955 =  2,414 / 2,425
## Totals       4   12  6011    4633       120 0.3983 = 4,294 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.601670
## 2 2  0.807699
## 3 3  0.918089
## 4 4  0.979406
## 5 5  1.000000
```

Predict on the test set The following code prints the predicted versus actual cuts for all 10,780 observations. We are only showing the first ten.

```
# Predict on the test set
predictions = h2o.predict(model, newdata = test)
```

```
##   |                                                                               |
```

```
# Convert predictions and actual values to factors for comparison
predicted_values = as.factor(as.vector(predictions$predict))
actual_values = as.factor(as.vector(test$cut))
# Combine them into a data frame with two columns
results_df = data.frame(Predicted = predicted_values, Actual = actual_values)
# Display the results
print(head(results_df,10))
```

```
##    Predicted    Actual
## 1      Ideal      Good
```

```
## 2      Ideal      Good
## 3      Ideal Very Good
## 4      Ideal Very Good
## 5      Ideal Very Good
## 6    Premium Very Good
## 7    Premium      Good
## 8      Ideal      Good
## 9      Ideal      Good
## 10     Ideal Very Good
```

Confusion matrix

The results from the confusion matrix show the predicted classifications of diamonds based on the model but sorted by the actual group they belong to. For instance, among the actual "Ideal" diamonds, 4019 were correctly predicted as "Ideal," while 1,421 were incorrectly classified. The model performed relatively well for "Premium" diamonds, correctly predicting 2,413 instances, but struggled with "Very Good" diamonds, where only 131 were correctly identified out of a total of 1,410. Overall, this matrix indicates a mix of correct and incorrect predictions across categories, suggesting that the model may need improvement to enhance its accuracy in classifying diamond qualities.

```r
# Generate the confusion matrix
confusion_matrix = table(Predicted = predicted_values, Actual = actual_values)
# Print the confusion matrix
print(confusion_matrix)
```

```
##            Actual
## Predicted   Fair Good Ideal Premium Very Good
##   Fair         4    0     0       0         0
##   Good        11    0     0       1         0
##   Ideal      111  383  4097     371      1049
##   Premium    108  537   249    2374      1365
##   Very Good   60   48     1       0        11
```

Neural Network attempt to improve

Here we are defining which variables are the predictors and the response, and running the model with train data.

```r
# Set the predictor and response variables
# Define specific predictor column names
predictors <- c("depth", "table", "x", "y", "z", "carat")
response = "cut" # cut is our response

# Define and train the neural network model
model = h2o.deeplearning(
  x = predictors,
  y = response,
  training_frame = train,
  validation_frame = test,
  activation = "RectifierWithDropout", # Activation function
  hidden = c(20, 20), # Two hidden layers with 10 neurons each
  epochs = 150, # Number of training epochs
```

```r
  rate = 0.001, # Learning rate
  input_dropout_ratio = 0.2, # Input dropout ratio
  hidden_dropout_ratios = c(0.5, 0.5) # Dropout ratios for each hidden layer
)
```

```
## Warning in .h2o.processResponseWarnings(res): rate cannot be specified if adaptive_rate is enabled..

##   |                                                                              |
```

```r
print(model)
```

```
## Model Details:
## ==============
##
## H2OMultinomialModel: deeplearning
## Model ID:  DeepLearning_model_R_1730373408053_42
## Status of Neuron Layers: predicting cut, 5-class classification, multinomial distribution, CrossEntr
##   layer units             type dropout       l1       l2 mean_rate rate_rms
## 1     1     6            Input 20.00 %       NA       NA        NA       NA
## 2     2    20 RectifierDropout 50.00 % 0.000000 0.000000  0.000452 0.000721
## 3     3    20 RectifierDropout 50.00 % 0.000000 0.000000  0.001105 0.001016
## 4     4     5          Softmax      NA 0.000000 0.000000  0.007329 0.004771
##   momentum mean_weight weight_rms  mean_bias bias_rms
## 1       NA          NA         NA         NA       NA
## 2 0.000000    0.119501   0.447515  -0.290111 0.437678
## 3 0.000000   -0.173766   0.498913   0.031845 0.367801
## 4 0.000000   -4.377141   3.039993 -18.361819 0.811023
##
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on training data. **
## ** Metrics reported on temporary training frame with 9911 samples **
##
## Training Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.3161765
## RMSE: (Extract with `h2o.rmse`) 0.5622957
## Logloss: (Extract with `h2o.logloss`) 0.8856064
## Mean Per-Class Error: 0.4576101
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,train = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error           Rate
## Fair       124  146     2      15        25 0.6026 =     188 / 312
## Good         7  280    14     210       370 0.6822 =     601 / 881
## Ideal        1    2  3426     218       277 0.1269 =   498 / 3,924
## Premium      0    0   251    2233        77 0.1281 =   328 / 2,561
## Very Good    0   40   550    1081       562 0.7483 = 1,671 / 2,233
## Totals     132  468  4243    3757      1311 0.3316 = 3,286 / 9,911
```

```
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,train = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.668449
## 2 2  0.899102
## 3 3  0.969428
## 4 4  0.998083
## 5 5  1.000000
##
##
##
##
## H2OMultinomialMetrics: deeplearning
## ** Reported on validation data. **
## ** Metrics reported on full validation frame **
##
## Validation Set Metrics:
## =====================
##
## Extract validation frame with `h2o.getFrame("RTMP_sid_923e_6")`
## MSE: (Extract with `h2o.mse`) 0.3138387
## RMSE: (Extract with `h2o.rmse`) 0.5602131
## Logloss: (Extract with `h2o.logloss`) 0.8794443
## Mean Per-Class Error: 0.4576014
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>,valid = TRUE)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error            Rate
## Fair       115  142     2      11        24 0.6088 =      179 / 294
## Good         9  307    19     223       410 0.6829 =      661 / 968
## Ideal        1    4  3833     245       264 0.1182 =    514 / 4,347
## Premium      0    0   265    2398        83 0.1267 =    348 / 2,746
## Very Good    0   39   592    1191       603 0.7513 =  1,822 / 2,425
## Totals     125  492  4711    4068      1384 0.3269 = 3,524 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>,valid = TRUE)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.673098
## 2 2  0.901577
## 3 3  0.967904
## 4 4  0.998330
## 5 5  1.000000
```

Evaluating the model performance on the test data set, we can immediately notice the RMSE is reduced to around 0.57, which means our predictions in this model are, on average, a smaller distance away from the actual observations than the original model. Likewise, the previous model gave us an error value of around 0.4, whereas this model gives us a total error value of 0.335. Changing the restrictions within the model reduced the RMSE and error value, meaning our model was able to more accurately predict the values to what they actually were observed to be.

```r
perf = h2o.performance(model, newdata = test)

# Print the model performance
print(perf)
```

```
## H2OMultinomialMetrics: deeplearning
##
## Test Set Metrics:
## =====================
##
## MSE: (Extract with `h2o.mse`) 0.3138387
## RMSE: (Extract with `h2o.rmse`) 0.5602131
## Logloss: (Extract with `h2o.logloss`) 0.8794443
## Mean Per-Class Error: 0.4576014
## AUC: (Extract with `h2o.auc`) NaN
## AUCPR: (Extract with `h2o.aucpr`) NaN
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
## =========================================================================
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
##           Fair Good Ideal Premium Very Good  Error            Rate
## Fair       115  142     2      11        24 0.6088 =     179 / 294
## Good         9  307    19     223       410 0.6829 =     661 / 968
## Ideal        1    4  3833     245       264 0.1182 =   514 / 4,347
## Premium      0    0   265    2398        83 0.1267 =   348 / 2,746
## Very Good    0   39   592    1191       603 0.7513 = 1,822 / 2,425
## Totals     125  492  4711    4068      1384 0.3269 = 3,524 / 10,780
##
## Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
## =======================================================================
## Top-5 Hit Ratios:
##   k hit_ratio
## 1 1  0.673098
## 2 2  0.901577
## 3 3  0.967904
## 4 4  0.998330
## 5 5  1.000000
```

Predict on the test set (only printing 10 of 10,000+ rows):

```r
# Predict on the test set
predictions = h2o.predict(model, newdata = test)
```

```
##   |                                                                              |
```

```r
# Convert predictions and actual values to factors for comparison
predicted_values = as.factor(as.vector(predictions$predict))
actual_values = as.factor(as.vector(test$cut))
# Combine them into a data frame with two columns
results_df = data.frame(Predicted = predicted_values, Actual = actual_values)
# Display the results
print(head(results_df,10))
```

```
##    Predicted    Actual
## 1  Very Good      Good
## 2  Very Good      Good
## 3      Ideal Very Good
## 4      Ideal Very Good
## 5      Ideal Very Good
## 6    Premium Very Good
## 7       Good      Good
## 8       Good      Good
## 9  Very Good      Good
## 10     Ideal Very Good
```

Confusion matrix

The confusion matrix for this model has very interesting results. First, the model is only predicting 53 Fair observations, which is a significantly small proportion of the actual fair observations, with 254. For this category, the error rate was 0.82, so this result is somewhat expected. Similarly, the error rate for the "Good" category was 0.92, and the mode is only predicting under 200 observations for this variable. The model mostly predicts the observations to be "Ideal", "Premium", or "Very Good", and predicts each of those variables relatively well.

```r
# Generate the confusion matrix
confusion_matrix = table(Predicted = predicted_values, Actual = actual_values)
# Print the confusion matrix
print(confusion_matrix)
```

```
##            Actual
## Predicted   Fair Good Ideal Premium Very Good
##   Fair       115    9     1       0         0
##   Good       142  307     4       0        39
##   Ideal        2   19  3833     265       592
##   Premium     11  223   245    2398      1191
##   Very Good   24  410   264      83       603
```

#THIRD TECHNIQUE USING CARET # "Decision Trees"

To be able to use this technique we have to downside the amount of data and cross-validations. We select randomly 5000 rows which represents 10% of the total data set. The random forest model was trained on a dataset of 4,003 samples and 10 predictors to classify diamonds into five quality categories: 'Fair,' 'Good,' 'Ideal,' 'Premium,' and 'Very Good.' Using a three-fold cross-validation approach, the model achieved varying accuracy rates based on the number of predictors considered (mtry), with the highest accuracy of approximately 86.01% and a Kappa statistic of 0.8043 when 11 predictors were used. The Kappa value indicates a strong level of agreement between predicted and actual classifications beyond what would be expected by chance. Ultimately, the model selected 11 as the optimal number of predictors to maximize classification accuracy.

```r
# Set a seed for reproducibility
set.seed(123)

# Sample a smaller subset for testing
diamonds_random <- diamonds[sample(nrow(diamonds), 5000), ]

# Create a training (80%) and testing (20%) split
trainIndex <- createDataPartition(diamonds_random$cut, p = 0.8, list = FALSE)
```

```r
trainData <- diamonds_random[trainIndex, ]
testData <- diamonds_random[-trainIndex, ]

# Define the training control
trainControl <- trainControl(method = "cv", number = 3)  # Reduced to 3 folds

# Train the model using the random forest algorithm
model <- train(cut ~ ., data = trainData, method = "rf",
               trControl = trainControl)

# Print the model details
print(model)
```

```
## Random Forest
##
## 4003 samples
##   10 predictor
##    5 classes: 'Fair', 'Good', 'Ideal', 'Premium', 'Very Good'
##
## No pre-processing
## Resampling: Cross-Validated (3 fold)
## Summary of sample sizes: 2669, 2669, 2668
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.8246317  0.7512863
##   11    0.8613540  0.8060802
##   21    0.8583566  0.8019077
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 11.
```

Test data

Similarly to the original model, the model on the test data also recommends the strongest model has 11 predictors. The accuracy for this model is slightly lower than before, with the highest accuracy value of 0.82. The kappa statistic for this accuracy level is 0.75, indicating that the model has a strong agreement between the predicted and observed model, but it is still not as good as the previous model.

```r
#Test set model

# Train the model using the random forest algorithm for the testing set
model_test = train(cut ~ ., data = testData, method = "rf",
trControl = trainControl)
# Print the model details
print(model_test)
```

```
## Random Forest
##
## 997 samples
##  10 predictor
##   5 classes: 'Fair', 'Good', 'Ideal', 'Premium', 'Very Good'
##
```

```
## No pre-processing
## Resampling: Cross-Validated (3 fold)
## Summary of sample sizes: 665, 665, 664
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.7803376  0.6852495
##   11    0.8244781  0.7538404
##   21    0.8254731  0.7557923
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 21.
```

Predictions on the test data

```r
# Make predictions on the test data
predictions = predict(model, newdata = testData)
# Display the predictions
print(head(predictions, 10))
```

```
##  [1] Ideal     Very Good Very Good Ideal     Ideal     Very Good Premium
##  [8] Very Good Very Good Ideal
## Levels: Fair Good Ideal Premium Very Good
```

Confusion matrix

The confusion matrix for the testing model shows very ideal results in prediction accuracy. The predicted categories were almost completely correct, with only a few observations in the other categories. Unlike what we have seen in previous models, there are a lot of zeros in the confusion matrix in the categories that do not match. This is a very good result, as we do not have many incorrect predictions. The overall accuracy percentage is around 86% and the kappa statistic is 0.8, revealing that the model is not perfect, but for real data, it does a good job at predicting the data.

```r
# Create a confusion matrix
confMatrix = confusionMatrix(predictions, testData$cut)
# Print the confusion matrix and other metrics
print(confMatrix)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Fair Good Ideal Premium Very Good
##   Fair        24    1     1       0         0
##   Good         1   87     5       0         0
##   Ideal        2    4   389       0         0
##   Premium      0    0     0     213        83
##   Very Good    0    0     0      44       143
##
## Overall Statistics
##
##                Accuracy : 0.8586
##                  95% CI : (0.8354, 0.8796)
##     No Information Rate : 0.3962
```

```
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8021
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: Fair Class: Good Class: Ideal Class: Premium
## Sensitivity              0.88889     0.94565       0.9848         0.8288
## Specificity              0.99794     0.99337       0.9900         0.8878
## Pos Pred Value           0.92308     0.93548       0.9848         0.7196
## Neg Pred Value           0.99691     0.99447       0.9900         0.9372
## Prevalence               0.02708     0.09228       0.3962         0.2578
## Detection Rate           0.02407     0.08726       0.3902         0.2136
## Detection Prevalence     0.02608     0.09328       0.3962         0.2969
## Balanced Accuracy        0.94341     0.96951       0.9874         0.8583
##                      Class: Very Good
## Sensitivity                    0.6327
## Specificity                    0.9429
## Pos Pred Value                 0.7647
## Neg Pred Value                 0.8975
## Prevalence                     0.2267
## Detection Rate                 0.1434
## Detection Prevalence           0.1876
## Balanced Accuracy              0.7878
```

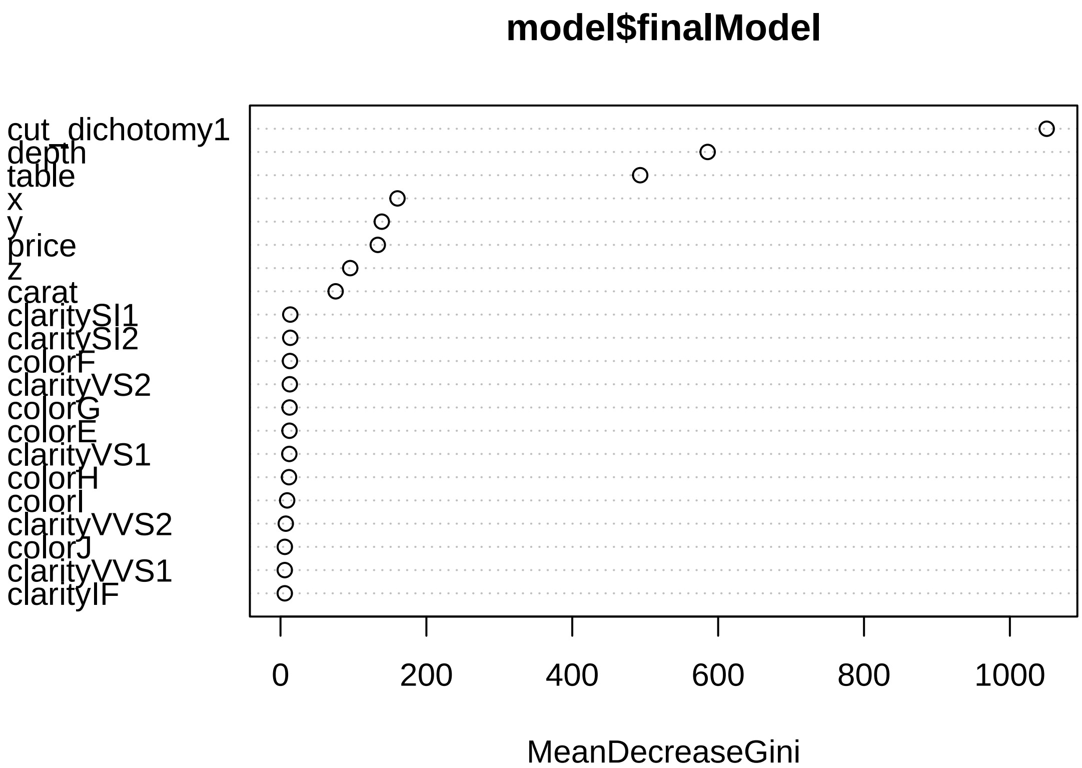
Visualize the results

The plot given by this code shows the level of importance of each feature in the model compared to each other. The further to the right the points fall, the higher importance they have in determining the prediction.

Clarity and color appear to have the least amount of importance on the predictions, whereas the cut_dichotomy has the greatest. Since we are predicting cut, this result is intuitive. Of the numerical variables, depth has the strongest value, followed by table. The X variable, representing the length, has the most importance when compared to the other measurement variables, y and z. Price has a lower importance to the prediction compared to y, but a higher prediction value compared to z. Carat falls slightly behind z, but ahead of clarity and color.

These results are interesting to show how the variables we are using to make predictions individually influence the decision our model makes.

```r
# Plot variable importance
varImpPlot(model$finalModel)
```



IV. ESEMBLE

The Ensemble function from Caret package in R is used to train multiple models at the same time; for this exercise the models to be tested were Random forest. Stochastic Gradiend Boosting and k-Nearest Neighbors.

```
# Convert cut to a factor with valid names
trainData$cut <- factor(trainData$cut, labels = make.names(levels(trainData$cut)))

# Define the training control
trainControl = trainControl(method = "cv", number = 10,
savePredictions = "final")
# Define the models to be used in the ensemble
capture.output({
models = caretList(
cut ~ ., data = trainData, trControl = trainControl,
methodList = c("rf", "gbm", "knn")
)
}, file = if(.Platform$OS.type == "windows") "NUL" else "/dev/null")
# Print the summary of the models
print(models)
```

```
## $rf
## Random Forest
```

```
##
## No pre-processing
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    2    0.8298639  0.7588643
##   11    0.8663229  0.8128213
##   21    0.8665785  0.8132684
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 21.
##
## $gbm
## Stochastic Gradient Boosting
##
## No pre-processing
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  Accuracy   Kappa
##   1                   50      0.8513724  0.7913270
##   1                  100      0.8578693  0.8006972
##   1                  150      0.8583606  0.8015246
##   2                   50      0.8576180  0.8003724
##   2                  100      0.8633568  0.8085510
##   2                  150      0.8681013  0.8152225
##   3                   50      0.8583686  0.8016280
##   3                  100      0.8653606  0.8113396
##   3                  150      0.8733538  0.8226448
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 150, interaction.depth =
##  3, shrinkage = 0.1 and n.minobsinnode = 10.
##
## $knn
## k-Nearest Neighbors
##
## No pre-processing
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.3919675  0.11702063
##   7  0.3824574  0.09538996
##   9  0.3784581  0.08452475
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 5.
##
## attr(,"class")
## [1] "caretList" "list"
```

The Ensemble function from Caret package in R is used to train multiple models at the same time; for this exercise the models to be tested were Random forest. Stochastic Gradiend Boosting and k-Nearest Neighbors.

The first technique, Random Forest, is used to build multiple decision trees during training and merges their outputs to improve accuracy and control overfitting. This model's performance was evaluated using different values for the parameter "mtry," which determines the number of variables randomly sampled at each split in the tree-building process. With a mtry of 21, the highest accuracy achieved was 0.8666, which indicates the proportion of correct predictions made by the model compared to the total predictions, here we aim to the highest accuracy. Also it provided a Kappa value, which measures the proportion of agreement between predicted classifications and actual classifications, it ranges between -1, less agreement, to 1 perfect agreement, with of approximately 0.8133 it indicates a strong level of agreement between the model's predictions for the categorical variable "cut" and the actual values of it.

The second technique Stochastic Gradient Boosting will build a model in a stage-wise fashion (building models incrementally). The results suggest that using deeper trees (interaction.depth of 3), where a higher depth allows the model to capture more complex patterns but can also lead to overfitting. With 150 trees yields the best performance in terms of accuracy with 0.8734 and Kappa value of 0.8226, indicating also a strong predictive performance and agreement between the model's predictions for the variable "cut" and actual outcomes. The tuning parameters "shrinkage" (controlling the contribution of each tree, also known as learning rate) and "n.minobsinnode" (the minimum number of observations in a node) were kept constant at 0.1 and 10, respectively. The consistent increase in metrics of accuracy and Kappa as the interaction depth increases indicates that the model benefits from capturing more complex patterns in the data, while the tuning parameters like shrinkage and minimum observations help mitigate overfitting risks.

The third technique used was k-Nearest Neighbors (knn), here the model evaluated different values of "k," as k =5, which represents the number of nearest neighbors considered when making a classification decision. The highest accuracy recorded was 0.3920, it indicates a low proportion of correctly classified instances compared to the total number of instances. Finally it gave a Kappa value of 0.1170. In this technique the predictions for all categories are very low, with no significant results observed and also indicating a relatively weak performance compared to the other models.

It can be concluded that Random Forest and Stochastic Gradient Boosting models demonstrated superior accuracy and Kappa values, indicating better classification performance than the k-Nearest Neighbors model.

Create an ensemble model using caretEnsemble

```r
ensembleModel = caretEnsemble(models, metric = "Accuracy", trControl = trainControl)
# Print the summary of the ensemble model
print(ensembleModel)
```

```
## The following models were ensembled: rf, gbm, knn
##
## caret::train model:
## Greedy Mean Squared Error Optimizer
##
## 4003 samples
##   15 predictor
##    5 classes: 'Fair', 'Good', 'Ideal', 'Premium', 'Very.Good'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3601, 3604, 3603, 3602, 3604, 3602, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8763565  0.8268873
```

```
##
## Tuning parameter 'max_iter' was held constant at a value of 100
##
## Final model:
## Greedy MSE
## RMSE:  0.186949
## Weights:
##              Fair Good Ideal Premium Very.Good
## rf_Fair       0.65 0.00  0.00    0.00      0.00
## rf_Good       0.01 0.45  0.00    0.00      0.00
## rf_Ideal      0.00 0.00  0.35    0.00      0.00
## rf_Premium    0.00 0.00  0.00    0.27      0.00
## rf_Very.Good  0.00 0.00  0.00    0.00      0.27
## gbm_Fair      0.34 0.00  0.00    0.00      0.00
## gbm_Good      0.00 0.54  0.00    0.00      0.00
## gbm_Ideal     0.00 0.00  0.65    0.00      0.00
## gbm_Premium   0.00 0.00  0.00    0.73      0.00
## gbm_Very.Good 0.00 0.00  0.00    0.00      0.73
## knn_Fair      0.00 0.00  0.00    0.00      0.00
## knn_Good      0.00 0.01  0.00    0.00      0.00
## knn_Ideal     0.00 0.00  0.00    0.00      0.00
## knn_Premium   0.00 0.00  0.00    0.00      0.00
## knn_Very.Good 0.00 0.00  0.00    0.00      0.00
```

The analysis involved training an ensemble model using 4,003 samples and 15 predictors, classifying instances into one of five "cut" categories: 'Fair', 'Good', 'Ideal', 'Premium', and 'Very.Good'. The model optimization was performed using the Mean Squared Error (MSE) . The ensemble model achieved an accuracy of 0.8764, indicating that approximately 87.64% of the predictions were correct. Also the Kappa value of 0.8269 signifies a high level of agreement between predicted and actual classifications for the variable "cut". The RMSE value is 0.1869, suggesting that, on average, the predictions deviate from the actual values by approximately 0.19.

The weights suggest that Random Forest primarily predicts 'Fair' and 'Good', while Gradient Boosting focuses more on 'Ideal', 'Premium', and 'Very.Good'.

As seen before for the technique k-Nearest Neighbors all the class weights are 0.00, indicating that k-NN did not contribute to the predictions effectively for the variable diamonds "cut".

So we can assume that the ensemble model utilizing Random Forest and Gradient Boosting shows better results in classifying instances into the categories of the variable "cut" . The high accuracy and Kappa values reflect a robust predictive model.

Make predictions

```r
# Make predictions on the test data using the ensemble model
ensemblePredictions = predict(ensembleModel, newdata = testData)
# Display the predictions
# print(ensemblePredictions)
stack_predict = as.matrix(ensemblePredictions)
# Apply a function to determine
result_vector = apply(stack_predict, 1, function(row) {
colnames(stack_predict)[which.max(row)]
})
result_vector = factor(result_vector)
head(data.frame(result_vector,testData$cut),10)
```

```
##    result_vector testData.cut
## 1          Ideal        Ideal
## 2      Very.Good    Very Good
## 3      Very.Good    Very Good
## 4          Ideal        Ideal
## 5          Ideal        Ideal
## 6      Very.Good    Very Good
## 7        Premium      Premium
## 8        Premium      Premium
## 9      Very.Good    Very Good
## 10         Ideal        Ideal
```

Confusion matrix

```
print(levels(result_vector))
```

```
## [1] "Fair"      "Good"      "Ideal"     "Premium"   "Very.Good"
```

```
print(levels(testData$cut))
```

```
## [1] "Fair"      "Good"      "Ideal"     "Premium"   "Very Good"
```

```
#we have to align the levels in order to create the confusion matrix
result_vector <- factor(result_vector, levels = levels(testData$cut))

# Create a confusion matrix
confMatrix = confusionMatrix(result_vector, testData$cut)
# Print the confusion matrix and other metrics
print(confMatrix)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Fair Good Ideal Premium Very Good
##   Fair        25    2     1       0         0
##   Good         0   85     3       0         0
##   Ideal        2    5   391       0         0
##   Premium      0    0     0     215        80
##   Very Good    0    0     0       0         0
##
## Overall Statistics
##
##                Accuracy : 0.885
##                  95% CI : (0.861, 0.9062)
##     No Information Rate : 0.4883
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.823
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
```

```
##
##                      Class: Fair Class: Good Class: Ideal Class: Premium
## Sensitivity              0.92593      0.9239       0.9899         1.0000
## Specificity              0.99616      0.9958       0.9831         0.8653
## Pos Pred Value           0.89286      0.9659       0.9824         0.7288
## Neg Pred Value           0.99744      0.9903       0.9903         1.0000
## Prevalence               0.03337      0.1137       0.4883         0.2658
## Detection Rate           0.03090      0.1051       0.4833         0.2658
## Detection Prevalence     0.03461      0.1088       0.4920         0.3646
## Balanced Accuracy        0.96104      0.9599       0.9865         0.9327
##                      Class: Very Good
## Sensitivity                   0.00000
## Specificity                   1.00000
## Pos Pred Value                    NaN
## Neg Pred Value                0.90111
## Prevalence                    0.09889
## Detection Rate                0.00000
## Detection Prevalence          0.00000
## Balanced Accuracy             0.50000
```

The confusion matrix results for the ensemble model shows an accuracy of 88.5%, with a Kappa of 0.823, indicating a good agreement between the predicted and actual values of the variable "cut.

As seen in the matrix , it clearly noted the prediction of 'Fair', 'Good', 'Ideal', and 'Premium' diamonds cut. The sensitivity for those cuts were 92.6%, 92.4% and 98.9% respectively, which indicates the high ability to recognize the cuts when comparing to the actual values. The 'Premium' class received perfect sensitivity of 100%, signifying that the model accurately predicted every 'Premium' instance in the test set.

Specificity metrics were also good across the diamond cut, reflecting the model's effectiveness in correctly rejecting non-target instances. For example, the specificity for 'Fair' was 99.6%, while 'Good' achieved 99.6%.

On the other hand, the model faces challenges with the 'Very Good' cut, registering a sensitivity of 0%. This means that no instances of 'Very Good' were correctly identified, indicating a weakness in the model's performance for this diamond "cut". Even when the specificity of 100% for 'Very Good', which shows that all non-'Very Good' predictions were accurate, the lack of positive predictions revealed an area for improvement.

The ensemble model demonstrates strong overall performance, achieving high accuracy and significant alignment between predicted and actual values for the majority of classes. However, in order to predict 'Very Good' cut some changes will need to be made, as the current model has difficulty accurately identifying it.