# Lab 12: Tree-Based Methods

*Johnny Hong*

*Stat 154, Fall 2017*

## Introduction

In this lab, we will explore various tree-based methods, namely decision trees, random forests, and boosted trees. This lab follows ISL 8.3: Lab: Decision Trees closely. The dataset we are using in this lab is `Carseats` from the `ISLR` package.

```
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.4.2
```

```
attach(Carseats)
High <- ifelse(Sales <= 8, "No", "Yes")
carseats <- data.frame(Carseats, High)
```

## Decision Trees

We will use the library `tree` to fit a decision tree. The syntax for `tree()` is analogous to `lm()`: `response ~ predictor` for the `formula` argument.

```
library(tree)
tree_carseats <- tree(High ~ .-Sales , data=carseats)
```

### Your turn

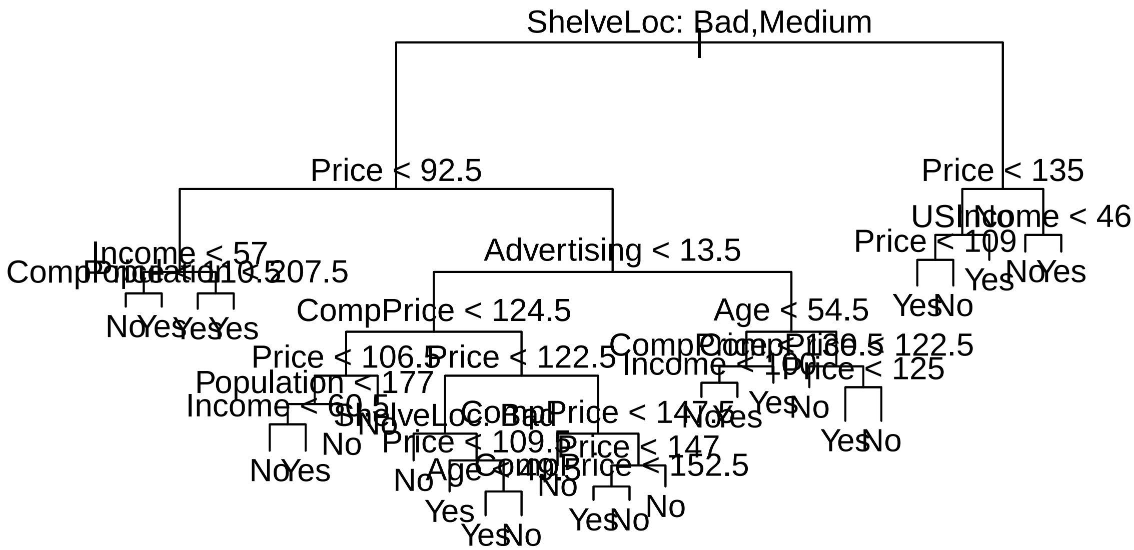
- Run `summary(tree_carseats)` and describe the output.
- Run `plot(tree_carseats)` and `text(tree_carseats, pretty=0)` and describe the output.
- Display `tree_carseats` and describe the output.

```
summary(tree_carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Income"      "CompPrice"   "Population"
## [6] "Advertising" "Age"         "US"
## Number of terminal nodes:  27
```

```
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

```r
plot(tree_carseats)
text(tree_carseats, pretty=0)
```




```r
tree_carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
```

```
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                 343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##               174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                 349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##               175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##        11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##            44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##              88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##              89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##            45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##          23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##            46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##            47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##              94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##              95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##     3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##       6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##        12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##          24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##          25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##        13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##       7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##        14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##        15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

# Random Forests

Random forests are considered one of the best "off-the-shelf" classifiers with minimal tuning. The idea is to build many weakly correlated trees (and hence a *forest*) via bagging and random variable selections (and hence *random*). Then the prediction is done via a majority vote. We will use the library `randomForest` to fit a random forest.

Remark: Random forests are *embarrassingly parallel*, meaning that the fitting can be easily separated into a number of parallel tasks. Packages such as `ranger` and `ParallelForest` provide an easy-to-use implementation of paralleled random forests, allowing efficient computations.

### Your turn

- Randomly select 80% of the observations as the training set and the other 20% as the test set.
- Using the training set, train a random forest with `High` as the response and all other variables except `Sales` as predictors. Make sure you set `importance=TRUE`.
- Compute the test error rate. How does it compare to the out-of-bag (OOB) error rate?
- Use `importance()` to view the importance of each variable. Create a visualization via `varImpPlot()`.
- Which two predictors are the most important variables?

```r
library(randomForest)
```

```
## randomForest 4.6-12

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

##
## Attaching package: 'randomForest'

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

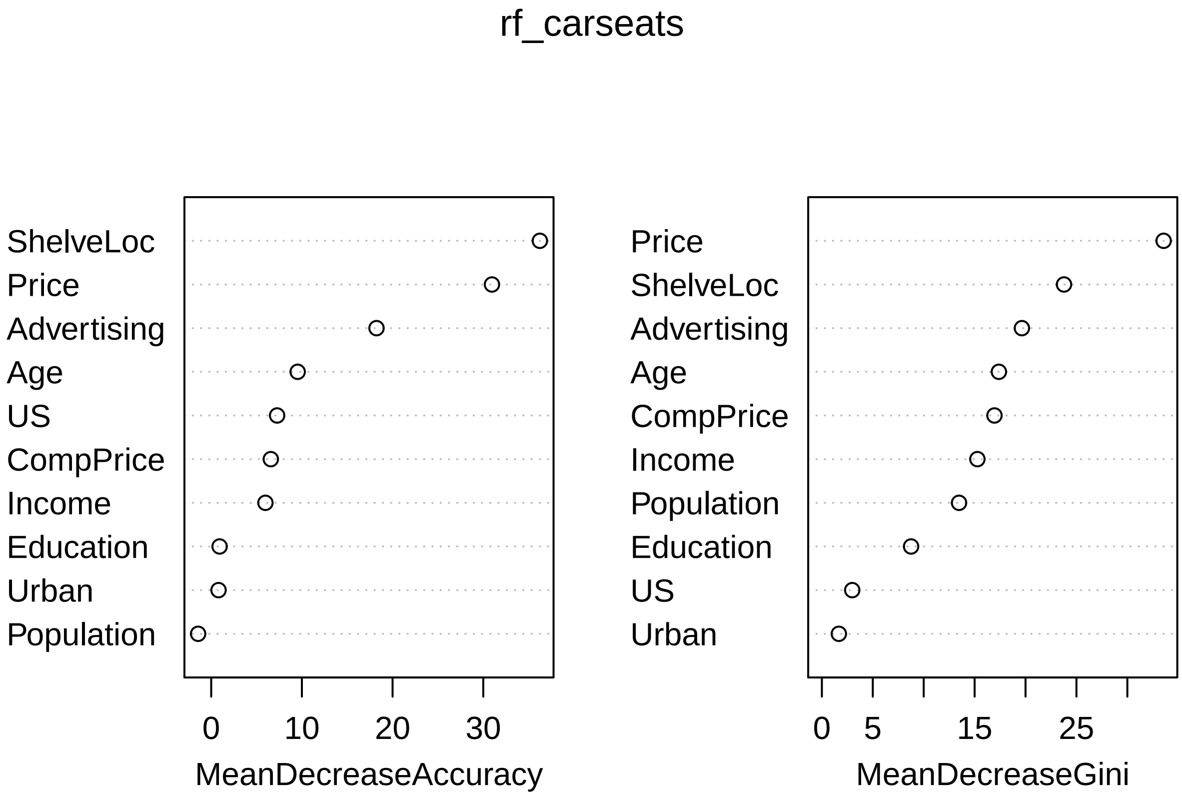
```r
trainIdx <- sample(1:nrow(carseats), floor(nrow(carseats) * 0.8))
rf_carseats <- randomForest(High ~ .-Sales , data=carseats[trainIdx, ],
                            importance=TRUE)
rf_test_pred <- predict(rf_carseats, carseats[-trainIdx, ])
mean(rf_test_pred != carseats[-trainIdx, "High"])
```

```
## [1] 0.1875
```

```r
importance(rf_carseats)
```

```
##                     No        Yes MeanDecreaseAccuracy MeanDecreaseGini
## CompPrice    6.1050287  3.1432031            6.5739805        16.948558
## Income       4.4065269  3.9445498            5.9785689        15.275060
## Advertising  8.9390792 17.3955288           18.2288806        19.643779
## Population  -3.5081623  1.5880450           -1.4419527        13.471229
## Price       24.3048222 24.2473971           30.9584917        33.562149
## ShelveLoc   28.7127280 28.1654714           36.2410203        23.778656
## Age          7.3245710  6.5468605            9.5356549        17.385160
## Education    0.2003174  1.0982306            0.9284251         8.762510
## Urban        0.3419229  0.8018009            0.8096052         1.668041
## US           2.5118795  6.5057399            7.2701509         2.980765
```

```r
varImpPlot(rf_carseats)
```




## Boosted Trees

To improve the performance of decision trees, boosting can be used. The idea of boosting is to iteratively fit a small tree to the residuals from the current model as an attempt to improve the model performance on areas where the current model does not do well. There are three tuning parameters for boosted trees: the number of trees $B$, the shrinkage parameter $\lambda$, and the interaction depth $d$. In this lab we will explore the impact of adjusting $B$ and $d$ on the classification performance. We will use the package `gbm` to fit boosted trees.

### Your turn

- Using the same train-test split as before, compute the test error rate for boosted trees. Train the boosted trees with $B = 5000$ trees. Use 0.5 as the cutoff for the predicted probabilities.
- Run `summary()` for the trained boosted trees.
- Based on the output from `summary()`, which are the two most importance variables?
- Note that when using `predict()`, we can specify the number of trees used via `n.trees`. Compute the test error rate with $B \in \{10, 20, 30, ..., 4950, 5000\}$. Plot the test error rate against the number of trees $B$.

- By default, the *interaction depth* is set to be 1. Redo the last part for $d = 2, 3$, and 4. Do you observe any qualitiative differences among the test error curves?

```
library(gbm)
```

```
## Loading required package: survival
```

```
##
## Attaching package: 'survival'
```

```
## The following object is masked from 'package:caret':
##
##     cluster
```

```
## Loading required package: splines
```

```
## Loading required package: parallel
```

```
## Loaded gbm 2.1.3
```

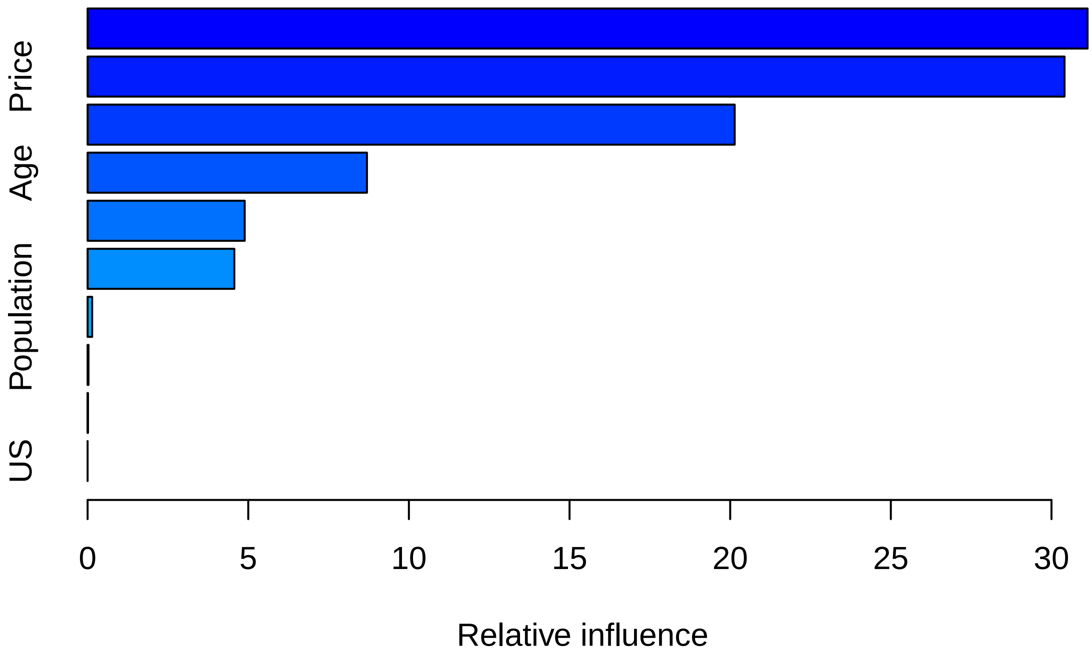
```
carseats01 <- carseats
carseats01$High <- 1 * (carseats01$High == "Yes")
boost_carseats <- gbm(High ~ .-Sales , data=carseats01[trainIdx, ],
                      n.trees=5000)
```

```
## Distribution not specified, assuming bernoulli ...
```

```
boost_test_pred <- predict(boost_carseats, carseats01[-trainIdx, ], n.trees=5000,
                           type="response")
mean(1 * (boost_test_pred > 0.5) != carseats01[-trainIdx, "High"])
```

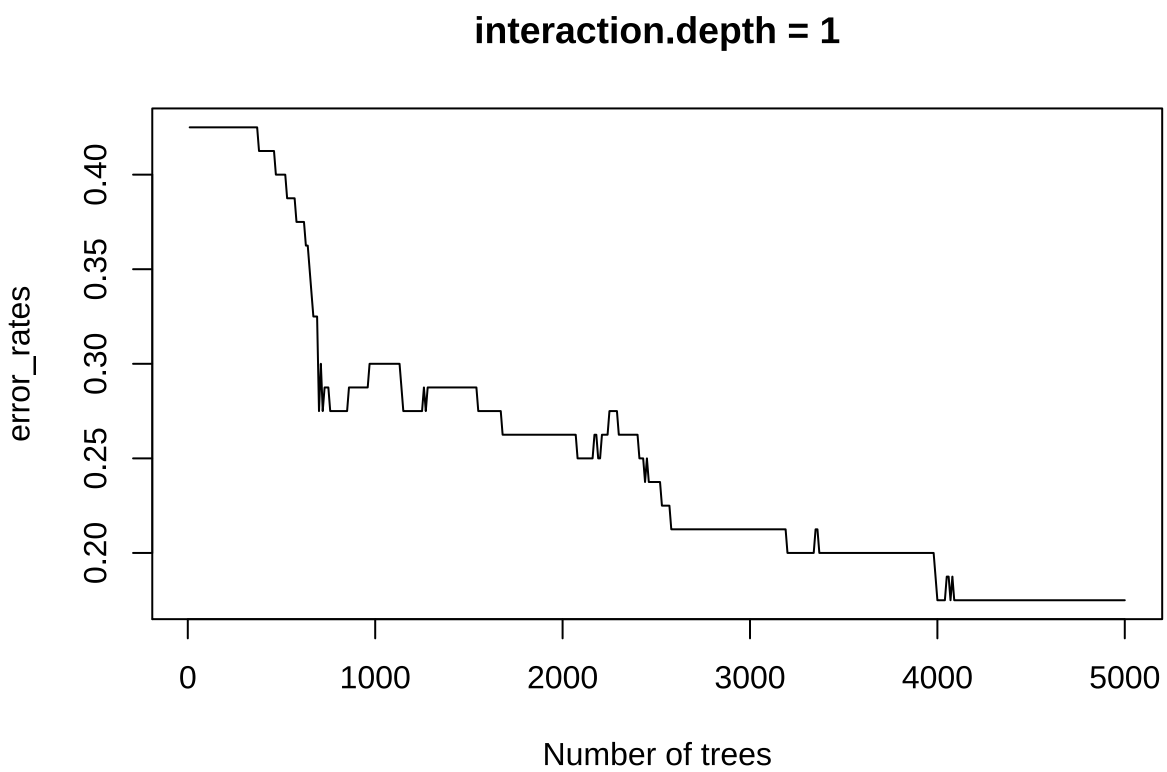
```
## [1] 0.175
```

```
summary(boost_carseats)
```



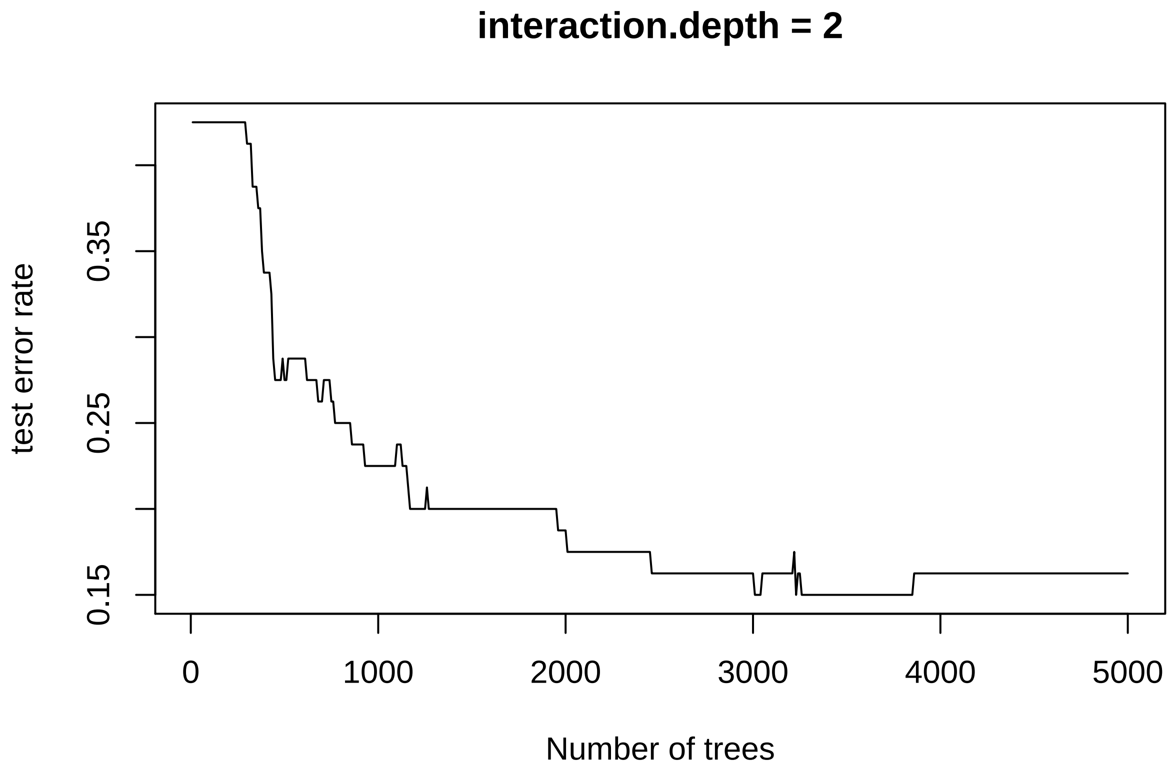

```
##                     var     rel.inf
## ShelveLoc     ShelveLoc 31.12329855
## Price             Price 30.40718053
## Advertising Advertising 20.14026121
## Age                 Age  8.69437352
## CompPrice     CompPrice  4.88956021
## Income           Income  4.56608339
## Population   Population  0.14139968
## Education     Education  0.02769436
## Urban             Urban  0.01014856
## US                   US  0.00000000
```

```
B_vector <- seq(10, 5000, by=10)
error_rates <- rep(NA, length(B_vector))
for (j in 1:length(B_vector)) {
  B <- B_vector[j]
  boost_test_pred <- predict(boost_carseats, carseats01[-trainIdx, ], n.trees=B,
                             type="response")
  error_rates[j] <- mean(1 * (boost_test_pred > 0.5) != carseats01[-trainIdx, "High"])
}
plot(error_rates ~ B_vector, xlab="Number of trees", type="l", main="interaction.depth =
```

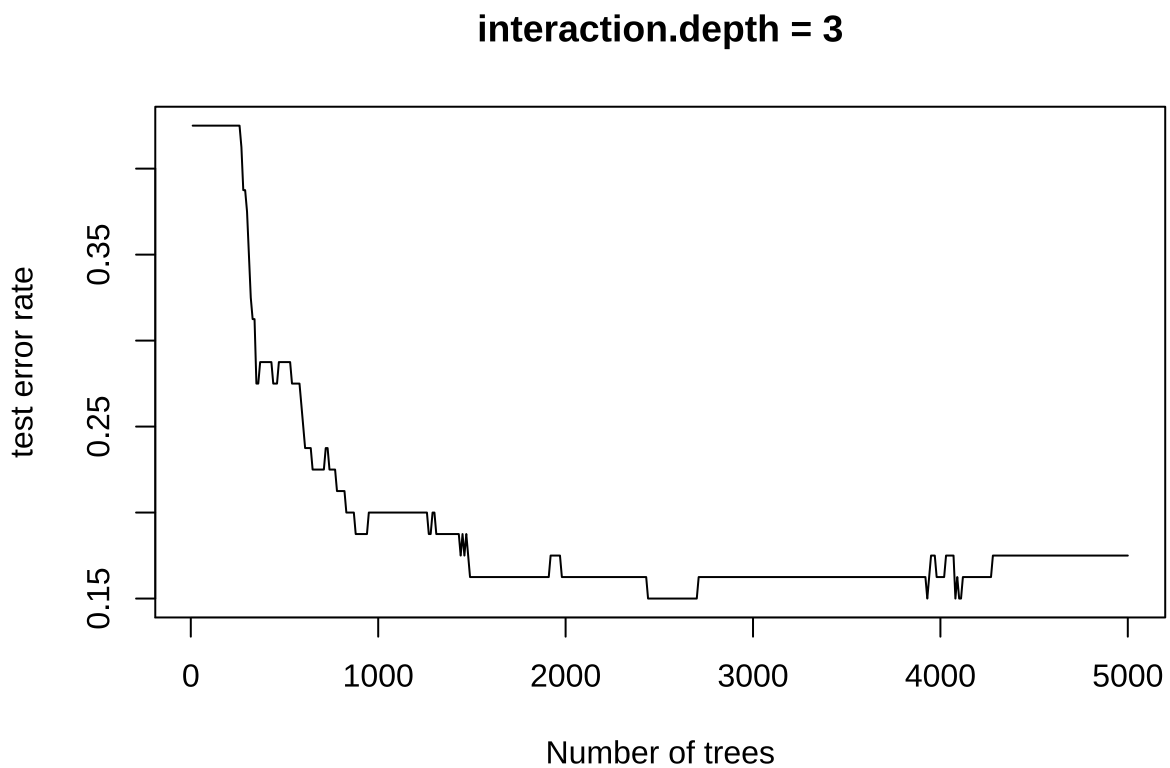



```
for (d in 2:4) {
  error_rates <- rep(NA, length(B_vector))
  boost_carseats <- gbm(High ~ .-Sales , data=carseats01[trainIdx, ],
                        n.trees=5000, interaction.depth=d)
  for (j in 1:length(B_vector)) {
    B <- B_vector[j]
    boost_test_pred <- predict(boost_carseats, carseats01[-trainIdx, ], n.trees=B,
                               type="response")
    error_rates[j] <- mean(1 * (boost_test_pred > 0.5) != carseats01[-trainIdx, "High"])
  }
  plot(error_rates ~ B_vector, xlab="Number of trees",
       type="l", ylab="test error rate",
       main=paste0("interaction.depth = ", d))
}
```

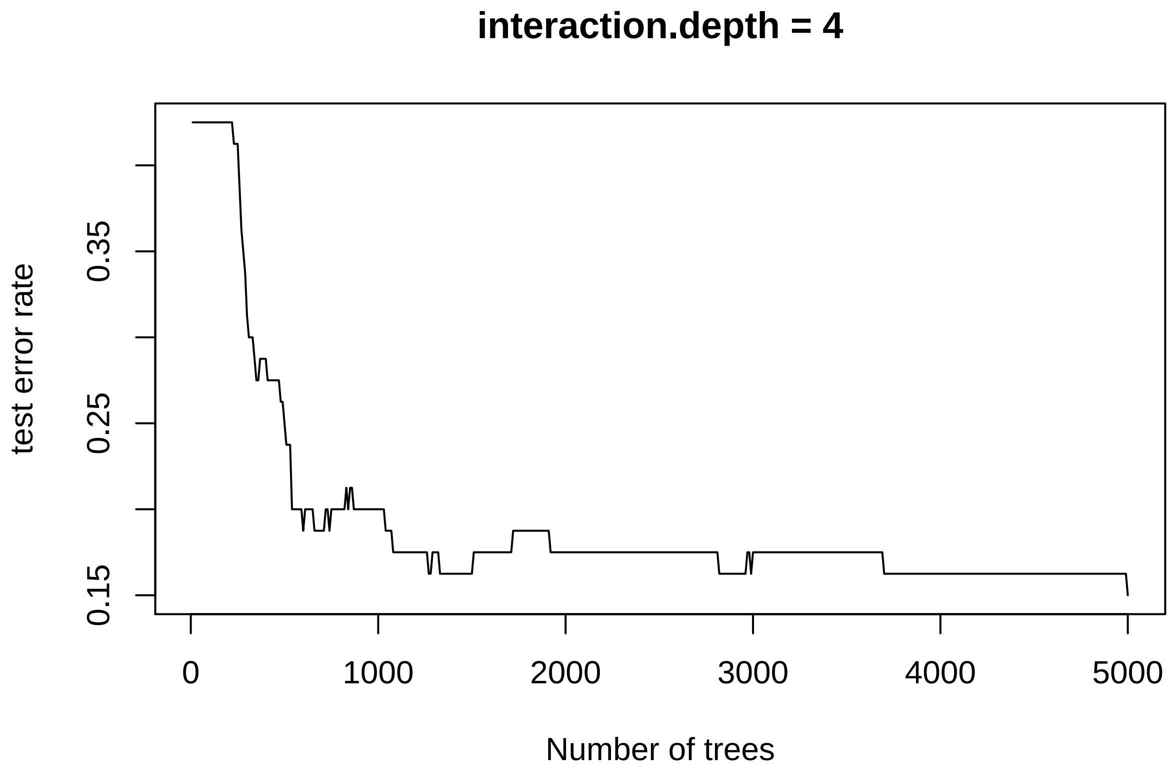
```
## Distribution not specified, assuming bernoulli ...
```




```
## Distribution not specified, assuming bernoulli ...
```




```
## Distribution not specified, assuming bernoulli ...
```



interaction.depth = 4
test error rate
0.15
0.25
0.35
0
1000
2000
3000
4000
5000
Number of trees