Yonsei University, Department of Applied Statistics

Chapter 8 실습

12기 YBIGTA 신보현

February 28, 2019

1. Classification Trees

tree library에서 회귀 나무와 분류 나무를 적합할 수 있다.
이번 실습에서는 Carseats 데이터를 이용하여 분류 나무를 적합해보자. 연속형 변수인 Sales 변수를 binary 변수로 우선 바꿔보자.

```
knitr::opts_chunk$set(comment=NA, fig.width=4, fig.height=4,fig.align='center',message=FALSE)
library(tree)
library(ISLR)
attach(Carseats)
high = ifelse(Sales <=8, 'no', 'yes')
carseats = data.frame(Carseats,high)
```

이제 tree() 함수를 이용하여 Sales을 제외한 모든 예측 변수를 통해 high 값을 분류해보자. 구문은 lm() 함수와 유사하다.

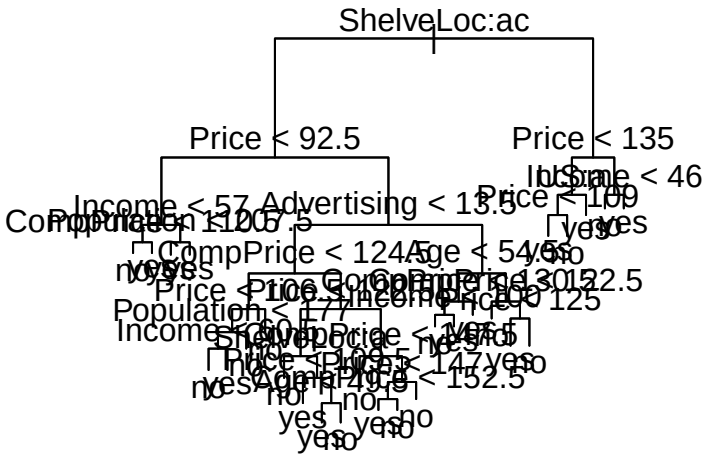
```
tree.carseats = tree(high~.-Sales,data=carseats)
summary(tree.carseats)
```

```
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
```

summary() 함수를 통해 terminal nodes의 개수, 훈련 오차 비율 등을 알 수 있다. 여기서 훈련 오차 비율이 9%이고 deviance는 0.4575이다. 여기서 deviance는 $-2\sum_m \sum_k n_{mk} log\hat{p}_{mk}$을 뜻하는데 n_{mk}는 m번째 terminal node에서 k번째 클래스에 속하는 관측치의 개수이다. 작은 deviance는 나무가 훈련 데이터에 잘 적합되었다는 뜻이다. summary() 함수를 통해 보고된 residual mean deviance은 deviance를 $n - |T_0|$로 나눈 값인데, 여기서 이 값은 $400 - 27 = 373$이다.

나무의 가장 매력적인 특징 중 하나는 시각적으로 표현이 가능하다는 것이다. plot() 함수를 통해서 나무 구조를 볼 수 있고 text() 함수를 통해서 node의 라벨을 나타낼 수 있다. pretty=0이라는 옵션은 범주형 변수에 대해서 카테고리의 이름을 보여주도록 한다.

```
plot(tree.carseats)
text(tree.carseats)
```



결과를 통해서 shelvLoc 변수가 나무의 가장 위에 있기 때문에 tree가 해당 변수를 가장 중요하다고 판단했음을 알 수 있다.

분류 나무의 성능을 정확히 파악하기 위해서는 훈련 데이터 오차를 측정하는 것이 아니라 테스트 데이터 오차를 측정해야 한다. 이를 위해 데이터를 두 부분으로 나누고 predict() 함수가 쓰인다. 분류 나무에 대해서는 type='class'라는 옵션이 R로 하여금 분류 예측을 하게 한다.

```
set.seed(2)
train = sample(1:400,200)
tree.carseats = tree(high~.-Sales, data=carseats, subset= train)
pred.tree = predict(tree.carseats,data=carseats[-train,],type='class')
table(pred.tree,carseats$high[-train])
```

```
pred.tree no yes
      no  73  51
      yes 43  33
```

```
(73+33)/200
```

```
[1] 0.53
```

정확도가 0.53으로 확인되었다.
다음으로 가지를 치는 것이 나무의 성능을 향상시키는지 확인해본다. cv.tree() 함수는 나무 복잡도의 최적의 단계를 결정하기 위해 CV를 실행한다. 분류 회귀에 대해서는, FUN=prune.misclass을 명시해야 하는데, 이는 CV를 시행할 때 classification error rate로 기준을 정하고 싶다는 뜻이다. FUN의 기본 값으로는 회귀 나무의 deviance을 계산해주니 분류 회귀에서 이를 수정하는 것을 기억하자.

```
set.seed(3)
cv.carseats = cv.tree(tree.carseats,FUN=prune.misclass)
names(cv.carseats)
```

```
[1] "size"   "dev"    "k"      "method"
```

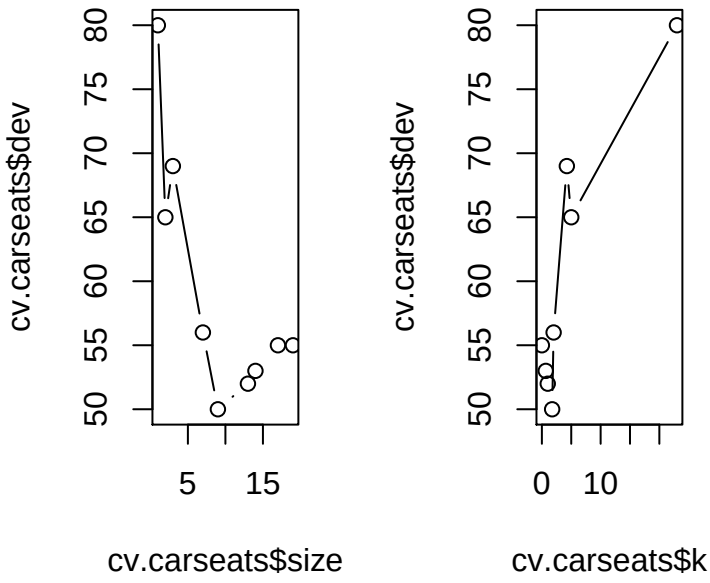
```
cv.carseats
```

```
$size
[1] 19 17 14 13  9  7  3  2  1

$dev
[1] 55 55 53 52 50 56 69 65 80

$k
[1]       -Inf  0.0000000  0.6666667  1.0000000  1.7500000  2.0000000
[7]  4.2500000  5.0000000 23.0000000
```

```
$method
[1] "misclass"


attr(,"class")
[1] "prune"         "tree.sequence"
```

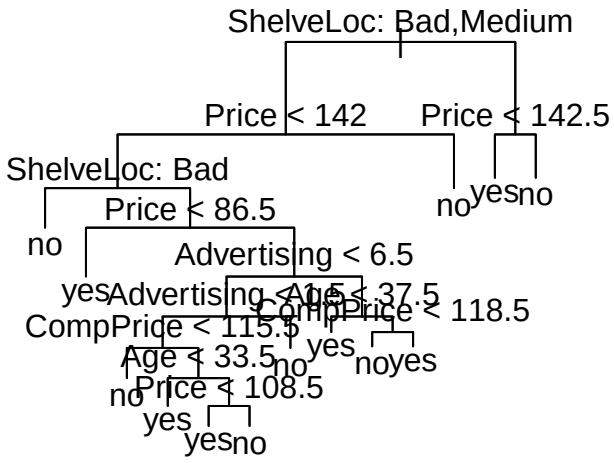
결과로는 고려된 terminal nodes의 수(size), 그에 따른 오차 비율과 cost-complexity parameter인 k(α에 대응 되는 값)을 보여준다. 또한 dev는 그 이름에도 불구하고 앞서 FUN=prune.misclass라고 지정했으니 이는 CV 오차 비율을 뜻한다.
결과를 보면, 9개의 terminal nodes를 가진 나무가 가장 낮은 CV 오차 비율을 가짐을 알 수 있다. CV 오차 비율을 size와 k의 함수로 그려보자.

```
par(mfrow=c(1,2))
plot(cv.carseats$size,cv.carseats$dev,type='b')
plot(cv.carseats$k,cv.carseats$dev,type='b')
```



size, 즉 terminal nodes의 개수가 9일 때 최소 값을, k가 1일 때 최소 값을 가진다. 이를 이용하여 prune.misclass() 함수를 통해서 9개의 노드를 가지고 k=1로 나무의 가지를 쳐보자.

```
prune.carseats = prune.misclass(tree.carseats,best=9,k=1)
plot(prune.carseats)
text(prune.carseats,pretty=0)
```




```
tree.pred = predict(prune.carseats,data=carseats[-train,],type='class')
table(tree.pred,carseats$high[-train])
```

```
tree.pred no yes
      no  66  50
      yes 50  34
```

```
(66+37)/200
```

```
[1] 0.515
```

가지를 치기 이전과 크게 차이가 없었다.

2. Regression Trees

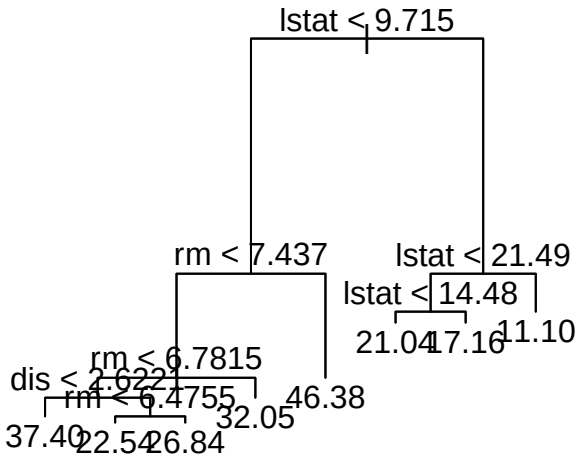
이번에는 Boston 데이터에 회귀 나무를 적용해본다.

```
library(MASS)
set.seed(1)
train = sample(1:nrow(Boston),nrow(Boston)/2)
tree.boston = tree(medv~.,data=Boston,subset=train)
summary(tree.boston)
```

```
Regression tree:
tree(formula = medv ~ ., data = Boston, subset = train)
Variables actually used in tree construction:
[1] "lstat" "rm"    "dis"
Number of terminal nodes:  8
Residual mean deviance:  12.65 = 3099 / 245
Distribution of residuals:
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-14.10000  -2.04200  -0.05357   0.00000   1.96000  12.60000
```

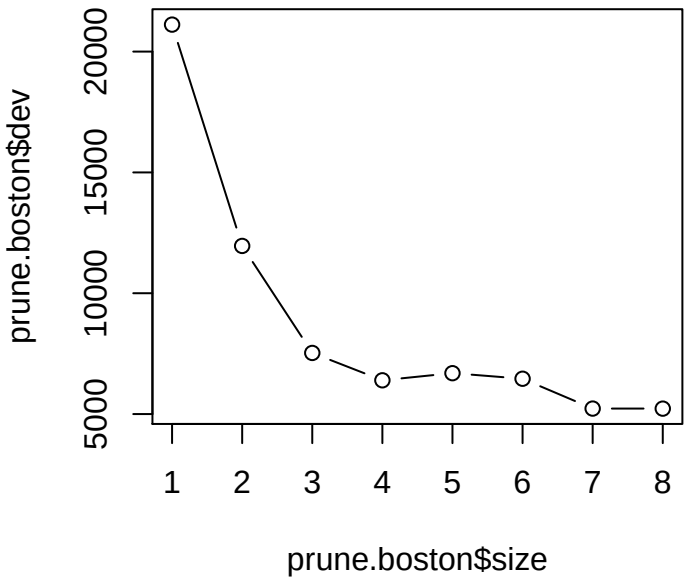
회귀 나무의 결과로 세 개의 변수만이 사용되었음에 주목하자. 회귀나무에서 deviance는 그 나무의 sum of squared errors이다. 이제 나무를 그려보자.

```
plot(tree.boston)
text(tree.boston,pretty=0)
```



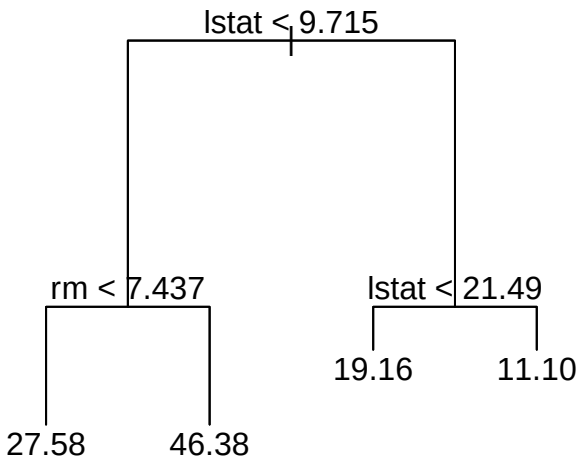
lstat 변수는 낮은 사회 경제적 위치에 있는 개인의 비율을 의미한다. 생성된 회귀 나무로부터 낮은 lstat 값을 가지면 더 높은 medv 값을 가지는 경향이 있음을 확인할 수 있다.
이제 cv.tree() 함수를 통해서 가지를 치는 것이 회귀 나무 성능을 향상시키는지 확인해본다.

```
prune.boston = cv.tree(tree.boston)
plot(prune.boston$size,prune.boston$dev,type='b')
```



위 그림을 통해 size, 즉 terminal nodes의 개수가 7개 또는 8개일 때, 가장 낮은 CV 오차를 가지는데, 이는 사실 가지를 안 치는 것과 다름 없으므로 비슷한 CV 오차를 가지는 4개로 가지를 쳐보자.

```
prune.boston = prune.tree(tree.boston,best=4)
plot(prune.boston)
text(prune.boston,pretty=0)
```



이제 가지를 치기 전의 회귀 나무와 가지를 친 회귀 나무의 성능을 비교해보자.

```
tree.pred = predict(tree.boston,newdata=Boston[-train,])
mean((Boston$medv[-train]-tree.pred)^2)
```

```
[1] 25.04559
```

```
tree.prune.pred = predict(prune.boston,newdata=Boston[-train,])
mean((Boston$medv[-train]-tree.prune.pred)^2)
```

```
[1] 32.22697
```

가지를 친 나무의 MSE가 살짝 더 높게 나왔다. 아마도 이 데이터에 대해서는 가지를 치지 않는 것이 더 좋은 선택임을 의미하는 듯 하다.

3. Bagging and Random Forests

randomForest 패키지를 이용하여 Boston 데이터에 bagging과 random forests을 적용해본

다. bagging은 $m = p$인 random forest의 특별한 경우임을 상기해보자. 따라서 randomForest() 함수를 통해 random forest와 bagging 모두를 시행할 수 있다. 먼저 bagging을 시행해보자.

```
library(randomForest)
set.seed(1)
bagging.boston = randomForest(medv~.,data=Boston,subset=train,
mtry=length(Boston)-1,importance=TRUE)
bagging.boston
```

```

Call:
 randomForest(formula = medv ~ ., data = Boston, mtry = length(Boston) -      1, importance = TR
               Type of random forest: regression
                     Number of trees: 500
No. of variables tried at each split: 13

          Mean of squared residuals: 11.15723
                    % Var explained: 86.49
```

```
names(bagging.boston)
```

```
 [1] "call"            "type"            "predicted"
 [4] "mse"             "rsq"             "oob.times"
 [7] "importance"      "importanceSD"    "localImportance"
[10] "proximity"       "ntree"           "mtry"
[13] "forest"          "coefs"           "y"
[16] "test"            "inbag"           "terms"
```

mtry=length(Boston)-1은 Boston의 모든 예측변수를 나무의 각 분리마다 고려한다는 옵션이다. 즉, bagging을 시행한다는 뜻이다. bagging이 얼마나 효과적으로 반응변수를 예측할까?

```
bagging.pred = predict(bagging.boston,newdata=Boston[-train,])
mean((Boston$medv[-train]-bagging.pred)^2)
```

```
[1] 13.50808
```

앞서 시행한 decision tree보다는 MSE가 낮아졌음을 확인할 수 있다. randomForest() 함수에서 ntree 옵션을 변경하여 나무의 수를 조정할 수도 있다. 하지만 ntree을 늘리는 것은 과적합으로 이어질 수 있으므로 구글링 결과, 기본 값으로 두는 것이 가장 나을 것 같았다. ranrandomForest() 함수에서 mtry 옵션을 변경함으로써 random forest을 실행할 수 있다. 기본 값으로 회귀 나무의 random forest에 대해서는 $p/3$ 개의 변수를, 분류 나무의 random forest에 대해서는 $\sqrt{p}$개의 변수를 사용한다. 여기서는 변수가 13개이므로 mtry=4을 적용해보자.

```
set.seed(1)
rf.boston = randomForest(medv~.,data=Boston,subset=train,
mtry=4,importance=TRUE)
rf.pred = predict(rf.boston,newdata=Boston[-train,])
mean((Boston$medv[-train]-rf.pred)^2)
```

```
[1] 11.6076
```

bagging의 테스트 MSE 추정치보다 조금 더 향상된 값을 보임을 확인할 수 있다.
importance() 함수를 이용해서 각 변수의 중요도를 살펴보자.

```
importance(rf.boston)
```

```
          %IncMSE IncNodePurity
crim    12.712371    1247.87271
zn       3.340046      84.97987
indus   10.611878    1308.70340
chas     1.733390      98.53459
nox     13.843947    1168.44941
rm      29.465247    5636.39110
age      6.565646     660.34542
```

```
dis     12.830180    1453.81733
rad      4.679142     161.38834
tax      9.195031     736.79049
ptratio 11.521584    1191.12896
black    8.385275     418.42770
lstat   26.313510    6179.11950
```

IncMSE에 대한 설명

The first one can be 'interpreted' as follows: if a predictor is important in your current model, then assigning other values for that predictor randomly but 'realistically' (i.e.: permuting this predictor's values over your dataset), should have a negative influence on prediction, i.e.: using the same model to predict from data that is the same except for the one variable, should give worse predictions.

So, you take a predictive measure (MSE) with the original dataset and then with the 'permuted' dataset, and you compare them somehow. One way, particularly since we expect the original MSE to always be smaller, the difference can be taken. Finally, for making the values comparable over variables, these are scaled.

계산 과정을다시 살펴보면,

1. 회귀 나무를 키우고 OOB MSE를 계산한다. 이를 MSE()라고 한다.

2. p개의 변수 각각에 대해서 j 번째 변수의 값을 변경해보고 OOB MSE를 계산한다. 이를 MSE(j)라고 한다.

3. j 번째 변수의 %IncMSE는 (MSE(j)-MSE())/MSE()*100 이다. 높으면 높을 수록, MSE(j)와 MSE()의 차이가 크다는 뜻이고, 이는 j 번째 변수의 값을 임의로 지정했을 때 MSE가 크다는 뜻이며 따라서 j 번째 변수가 영향력이 있다는 뜻이므로 중요한 변수라는 뜻이다.

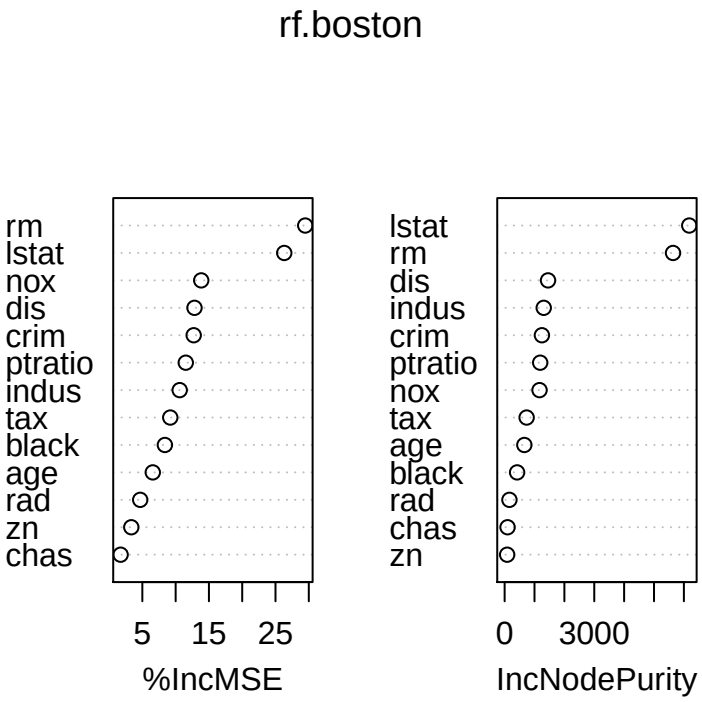
IncNodePurity에 대한 설명

For the second one: at each split, you can calculate how much this split reduces node impurity (for regression trees, indeed, the difference between RSS before and after the split). This is summed over all splits for that variable, over all trees.

IncNodePurity relates to the loss function which by best splits are chosen. The loss function is mse for regression and gini-impurity for classification. More useful variables achieve higher increases in node purities, that is to find a split which has a high inter node 'variance' and a small intra node 'variance'. IncNodePurity is biased and should only be used if the extra computation time of calculating %IncMSE is unacceptable. Since it only takes ~5-25% extra time to calculate %IncMSE, this would almost never happen.

그림을 통해 중요한 변수를 순서대로 확인할 수 있다.

```
varImpPlot(rf.boston)
```

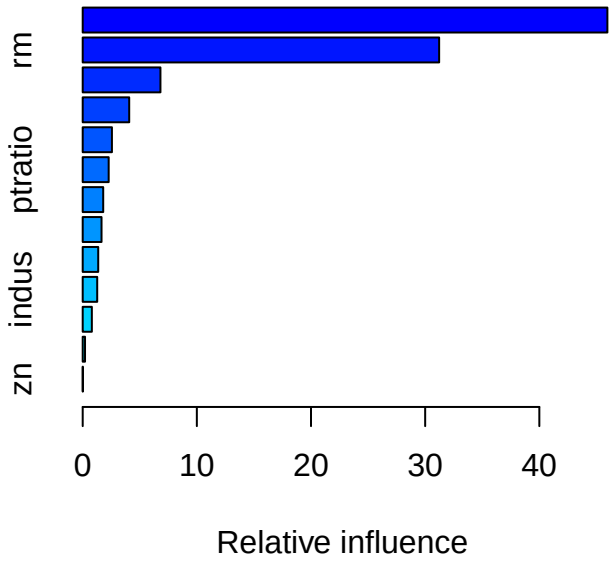


lstat와 rm이 가장 중요한 변수임을 확인할 수 있다.

4. Boosting

gbm 패키지의 gbm 함수를 이용하여 boosted 회귀 나무를 Boston 데이터에 적합해본다. 연속형 변수의 경우에는, distribution='gaussian'으로, 분류 문제의 경우에는 distribution='bernoulli'로 옵션을 맞춘다. n.trees는 나무의 수를, interaction.depth는 각 나무의 깊이를 의미한다.

```
library(gbm)
set.seed(1)
boost.boston = gbm(medv~., data=Boston[train,],
distribution='gaussian',n.trees=5000,interaction.depth=4)
summary(boost.boston)
```



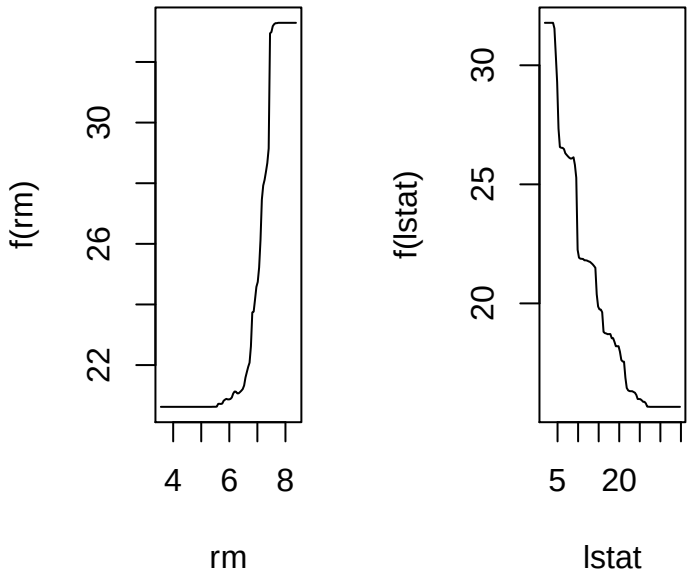

```
            var    rel.inf
lstat     lstat 45.9627334
rm           rm 31.2238187
dis         dis  6.8087398
crim       crim  4.0743784
nox         nox  2.5605001
ptratio ptratio  2.2748652
black     black  1.7971159
age         age  1.6488532
tax         tax  1.3595005
indus     indus  1.2705924
chas       chas  0.8014323
rad         rad  0.2026619
zn           zn  0.0148083
```

lstat와 rm이 가장 중요한 변수임을 확인할 수 있다.

또한 이 변수들에 대해 partial dependence plots을 만들 수 있다. 이것은 다른 변수의 영향력을 배제하고 해당 변수만의 marginal effect을 보여준다. 예상대로 medv 변수는 rm와

증가관계, lstat와 감소 관계이다.

```
par(mfrow=c(1,2))
plot(boost.boston,i='rm')
plot(boost.boston,i='lstat')
```



이제 테스트 오차에 대한 추정치를 구해보자.

```
boost.pred = predict(boost.boston,newdata=Boston[-train,],n.trees=5000)
mean((Boston$medv[-train]-boost.pred)^2)
```

```
[1] 11.84434
```

앞의 baggingm random forest와 크게 차이가 나지 않는다.
boosting을 실행할 때, shrinkage parameter λ의 기본 값은 0.001이지만 shrinkage=0.2와 같이 쉽게 수정할 수 있다.

```
set.seed(1)
boost.boston = gbm(medv~., data=Boston[train,],
distribution='gaussian',n.trees=5000,interaction.depth=4,shrinkage=0.2)
```