Solution to Series 10

1. **a)** cp stands for *cost complexity pruning*. It is a model parameter which penalizes large trees. So, a higher cp leads to shorter trees. It is a scaled version of the parameter α. The latter appears in the penalized goodness of fit:

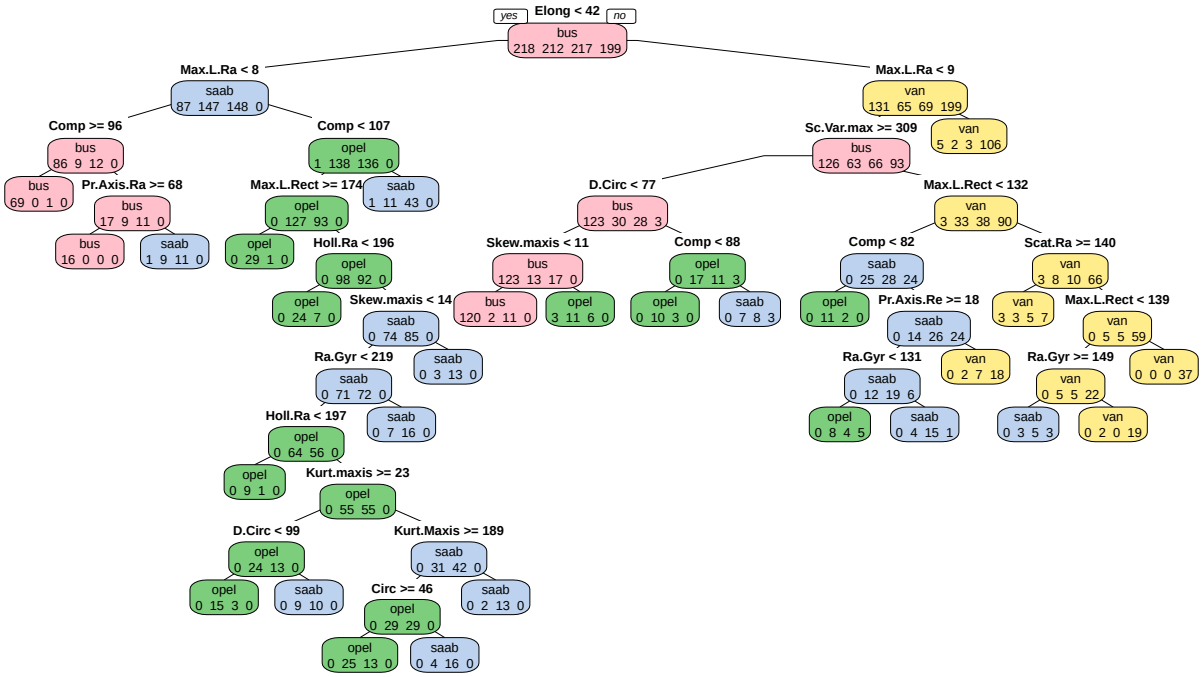
$$\mathcal{R}_\alpha(\mathcal{T}) = \mathcal{R}(\mathcal{T}) + \alpha \times size(\mathcal{T}) \tag{1}$$

The cp is equal to $\alpha/\mathcal{R}(\emptyset)$.
`minsplit` describes the minimum number of observations that must exist in a node in order for a split to be attempted. So, splits which create nodes with less than `minsplit` observations will not be considered.
If `minsplit=1`, any split will be considered. Moreover, if cp=0, the penalized goodness of fit is simply the goodness of fit:

$$\mathcal{R}_\alpha(\mathcal{T}) = \mathcal{R}(\mathcal{T}) \tag{2}$$

Consequentially, every observation should in principle have an individual leaf. In practice it can be that such a tree still has less leafs than observations. For example if a node in a tree has multiple observations, all belonging to the same class, splitting would not change the goodness of fit. In these cases, implementations of classification trees usually will not perform a split.

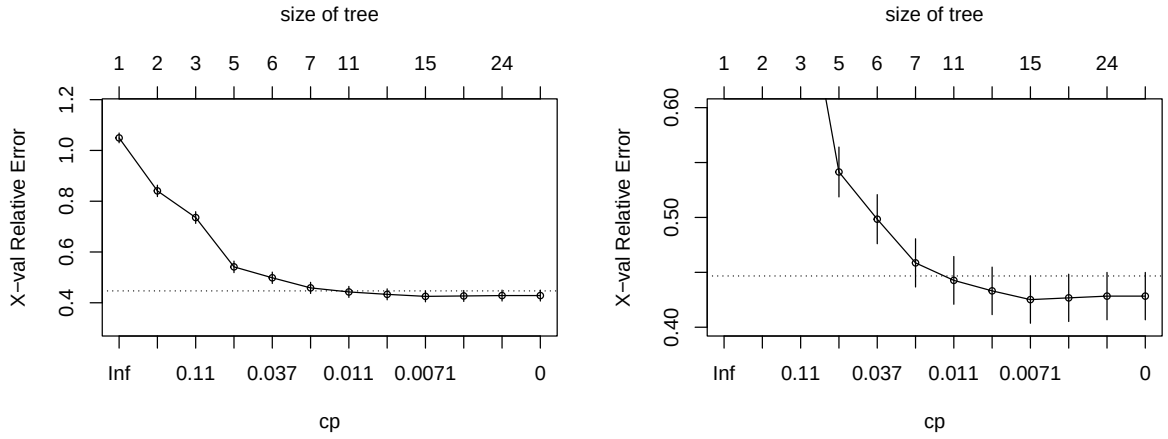
b)
```
> d.vehicle <- read.table("http://stat.ethz.ch/Teaching/Datasets/NDK/vehicle.dat",
                          header = TRUE)
> library(rpart)
> set.seed(10)
> ## train an overfitted tree:
> rp.veh <- rpart(Class ~ ., data = d.vehicle,
                  control = rpart.control(cp = 0.0, minsplit = 30))
>
> require(rpart.plot)## <- to plot the tree nicely by
> prp(rp.veh, type=1, extra=1, tweak=9/8,
    box.col=c('pink', 'palegreen3',
              'lightsteelblue 2','lightgoldenrod 1')[rp.veh$frame$yval])
```



The unpruned CART-fit has 27 terminal nodes and depth 11. Interpretation seems to be difficult because many of the predictors are selected to be substantial for the model at high interaction degrees. Many terminal nodes contain only a very small number of datapoints which indicates overfitting.

c) The x-axis of the cost complexity plot gives various values of the cost complexity pruning parameter cp. The on top of the plot, you can see the corresponding optimal sizes of the tree. The y-axis shows the cross-validated error, relative to the error of the root tree. Around each point in the plot, there is a confidence interval. The latter is obtained when performing the cross-validation.
For the one standard-error rule, we first find the model with the lowest cross-validation error. We then choose the simplest model, which is at most one standard-error worse than that model. In the plot, this is visualized with the dotted line: it lies one standard-error above the best model. The chosen model is the leftmost model, whose cross-validated error lies below this line.
The one-standard-error rule can be justified as follows: On the one hand, we want a model which has a low cross-validation error. On the other hand, we want a simple model. So if we have a simple model, which is almost as good as the best complex model, we prefer the simpler model. And "almost as good" here is defined as at most one standard-error apart (in terms of cv-error).

d) To prune the tree to make it cost-complexity optimal we first look at the cost-complexity plot; for better visibility we zoom in around the limiting line (right figure):

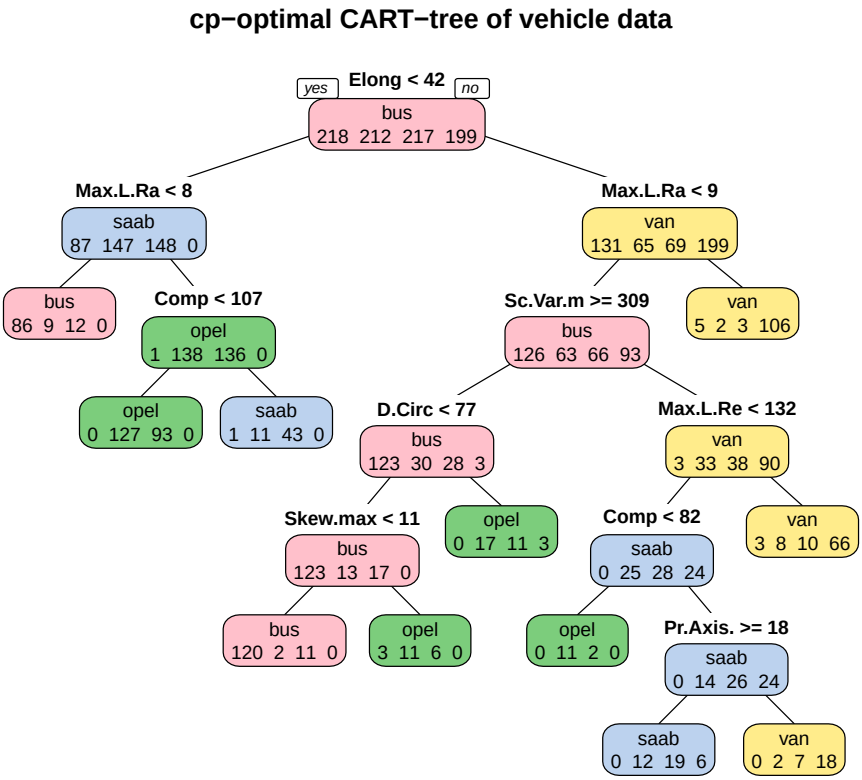


The full tree has the lowest cross-validation error and therefore defines the limit error (dashed line), which will help us to select the "optimal" tree using the one-standard-error rule. The smallest model whose error is below the line has size 11 (corresponding to 10 splits). To get the corresponding cp value, we look at the cost-complexity table:

```
            CP nsplit rel error    xerror       xstd
1  0.205414013      0 1.0000000 1.0493631 0.01921846
2  0.121019108      1 0.7945860 0.8407643 0.02243291
3  0.095541401      2 0.6735669 0.7356688 0.02305907
4  0.050955414      4 0.4824841 0.5414013 0.02270753
5  0.027070064      5 0.4315287 0.4984076 0.02236099
6  0.012738854      6 0.4044586 0.4585987 0.02194666
7  0.008757962     10 0.3535032 0.4426752 0.02175463
8  0.007961783     12 0.3359873 0.4331210 0.02163195
9  0.006369427     14 0.3200637 0.4251592 0.02152534
10 0.001592357     21 0.2643312 0.4267516 0.02154698
11 0.001061571     23 0.2611465 0.4283439 0.02156846
12 0.000000000     26 0.2579618 0.4283439 0.02156846
```

Thus for 10 splits we have a cp value of 0.00875796. The fitted cp-optimally pruned tree turns out to be much more handy and looks as follows:

```
> cp.opt <- rp.veh$cptable[7, "CP"]
> rp.veh.pruned <- prune.rpart(rp.veh, cp = cp.opt)
> prp(rp.veh.pruned, extra=1, type=1,
     main="cp-optimal CART-tree of vehicle data",
     box.col=
       c('pink', 'palegreen3',
              'lightsteelblue 2','lightgoldenrod 1')[rp.veh.pruned$frame$yval])
```



e) The misclassification error is simply the fraction of misclassified observations:

$$\frac{1}{n}\sum_{i=1}^{n} 1\{Y_i \neq \hat{Y}_i\}$$

It generally coincides with the goodness-of-fit $\mathcal{R}(\mathcal{T})$ which is minimized by the regression-tree (if $\alpha = 0$). So, it is an overly optimistic evaluation of the performance.
The leave-one-out cross-validated error is calculated as:

$$\frac{1}{n}\sum_{i=1}^{n} 1\{Y_i \neq \hat{Y}_i^{-i}\}$$

Here $\hat{Y}_i^{-i}$ is the prediction for the i-th observation for the model, which is trained on all but the i-th observation. The data point on which the performance of the model is evaluated is unseen by the model. So, compared to the misclassification error, there is no overfitting. The leave-one-out cross-validated error generally gives us a good estimate of the generalization error. A drawback is that the model needs to be trained n times.
The boostrap generalization error is calculated as follows:

1. Generate $(X_1^*, Y_1^*), \ldots, (X_n^*, Y_n^*)$ by resampling with replacement from the original data.
2. Compute the bootstrapped estimate $\hat{m}^*$ based on $(X_1^*, Y_1^*), \ldots, (X_n^*, Y_n^*)$.
3. Evaluate:
$$err^* = \frac{1}{n}\sum_{i=1}^{n} 1\{Y_i^* \neq \hat{m}^*(X_i^*)\}$$
4. Repeat steps 1-3 B times, approximate the bootstrap generalization error as the average of those values:
$$\frac{1}{B}\sum_{b=1}^{B} err^{*b}$$

In other words, we simulate the distribution of the misclassification error, and then take the average of B obervations from that distribution. The estimate we obtain may be overly optimistic, since some of the data points will be involved for both training and testing. This is addressed by the out-of-bootstrap-sample (OOB) generalization error we consider next.
The out-of-bootstrap-sample generalization error is calculated as follows:

1. Generate $(X_1^*, Y_1^*), \ldots, (X_n^*, Y_n^*)$ by resampling with replacement from the original data. Denote the indices of the unused data by $\mathcal{L}_{out}^*$
2. Compute the bootstrapped estimate $\hat{m}^*$ based on $(X_1^*, Y_1^*), \ldots, (X_n^*, Y_n^*)$.
3. Evaluate:
$$err^* = \frac{1}{|\mathcal{L}_{out}^*|} \sum_{i \in \mathcal{L}_{out}^*} 1\{Y_i^* \neq \hat{m}^*(X_i^*)\}$$
4. Repeat steps 1-3 B times, approximate the bootstrap generalization error as the average of those values:
$$\frac{1}{B} \sum_{b=1}^{B} err^{*b}$$

Similarly to cross-validation, we train the model on one set of data, and then evaluate it on another set of data. Again, there is no overfitting, and the result should be similar to what we obtain in leave-one-out cross-validation. Moreover, for both cross-validation and bootstrapping, we obtain multiple values (n for leave-one-out, B for bootstrapping). So, we get an estimate for the distribution of the generalization error, not just a single value. One advantage of boosting is that we can choose B arbitrarily large, while for cross-validation we can have at most n values.

f) We will use the following function:

```
> misclass.sample <- function(data, ind.training, ind.test)
 {
   tree <- rpart(Class ~ ., data = data[ind.training, ],
                 control = rpart.control(cp=0.0, minsplit = 30))

   ## choose optimal cp according to 1-std-error rule:
   cp <- tree$cptable
   min.ind <- which.min(cp[,"xerror"])
   min.lim <- cp[min.ind, "xerror"] + cp[min.ind, "xstd"]
   cp.opt <-  cp[ cp[,"xerror"] < min.lim, "CP"][1]

   prnd.tree <- prune.rpart(tree, cp=cp.opt)
   ## return  test misclassification rate:
   mean(data$Class[ind.test] !=
        predict(prnd.tree, newdata = data[ind.test, ], type = "class"))
 }
```

Misclassification error:

```
> mean(residuals(rp.veh.pruned))
[1] 0.2624113
```

Leave-one-out cross-validation:

```
> cv.err <- function(data, ind) misclass.sample(data, -ind, ind)
> ##-----
> n <- nrow(d.vehicle)
> cv.samples <- sapply(1:n, cv.err, data = d.vehicle)
> errcv <- mean(cv.samples)
```

Bootstrapping:

```
> B <- 1000
> boot.err <- function(dat, ind) misclass.sample(dat, ind, 1:n)
> boot.samples <- replicate(B, boot.err(d.vehicle, sample(1:n, replace = TRUE)))
> errboot <- mean(boot.samples)
```

We find a bootstrap generalization error of 0.2512 and a CV-value of 0.3322. The standard bootstrap approach tends to underestimate the generalization error since it uses the same data as training and as test set. We can overcome this problem by using the out-of-bootstrap sample to estimate the generalization error.

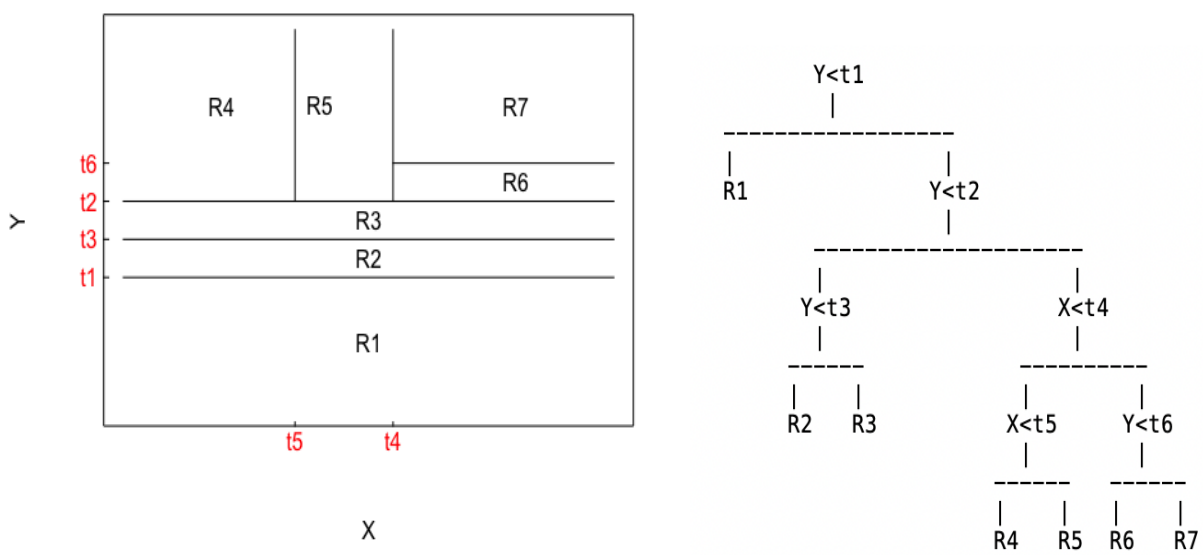
Out-of-bootstrap-sample generalization error:

```
> OOB.err <- function(dat, ind) misclass.sample(dat, ind, -ind)
```

```
> OOB.samples <- replicate(B, OOB.err(d.vehicle, sample(1:n, replace = TRUE)))
> errOOB <- mean(OOB.samples)
```

The generalization error of 0.3213 found now is larger than the (standard) bootstrap generalization error (and the variance may be larger; see also Series 7) and much closer to the CV-value.

2. Here we give one example with six regions.



3. The 10 estimates of $P(\text{Class is Red}|X)$ are

$$0.1, 0.15, 0.2, 0.2, 0.55, 0.6, 0.6, 0.65, 0.7, \text{ and } 0.75,$$

and the corresponding classification results are

Green, Green, Green, Green, Red, Red, Red, Red, Red and Red.

There are 4 Greens and 6 Reds, hence under the majority vote, the final classification is Red.

The average probability of the 10 estimated probabilities is 0.45. Therefore, based on the average probability approach, the final classification is Green. One can see that the results are different by using these two different approaches.

4. a)

```
> library(ISLR)
> data(Carseats)
> # set random seed as the train set is randomly selected,
> # and implementing randomForest contains randomness
> set.seed(39)
> # using random seed 10, you will see that the test MSE of the pruned tree is improved
> # set.seed(10)
>
> n <- nrow(Carseats)
> p <- ncol(Carseats)
> train = sample(n, round(n/2))
> Carseats.train = Carseats[train, ]
> Carseats.test = Carseats[-train, ]
```

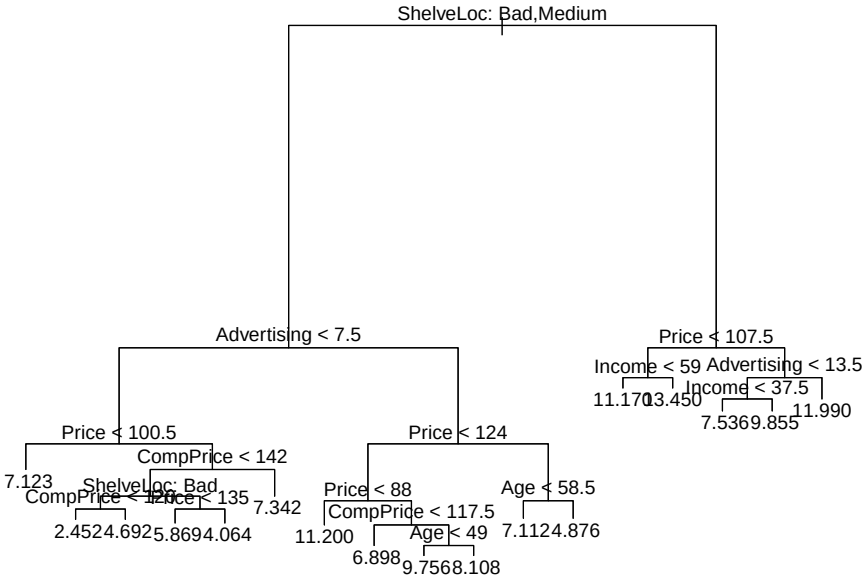
b)

```
> library(tree)           # you may need to update your R-version to install "tree" package
> # fit a regression tree
> tree.carseats = tree(Sales ~ ., data = Carseats.train)
> summary(tree.carseats)

Regression tree:
tree(formula = Sales ~ ., data = Carseats.train)
Variables actually used in tree construction:
[1] "ShelveLoc"   "Advertising" "Price"       "CompPrice"
[5] "Age"         "Income"
Number of terminal nodes:  17
Residual mean deviance:  2.068 = 378.5 / 183
Distribution of residuals:
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-4.13300 -0.97010 -0.09117  0.00000  0.91310  3.10100

> plot(tree.carseats)
> text(tree.carseats, pretty = 0)
> pred.carseats = predict(tree.carseats, Carseats.test)
> mean((Carseats.test$Sales - pred.carseats)^2)

[1] 5.134747
```



The test MSE is 5.134747.

c)

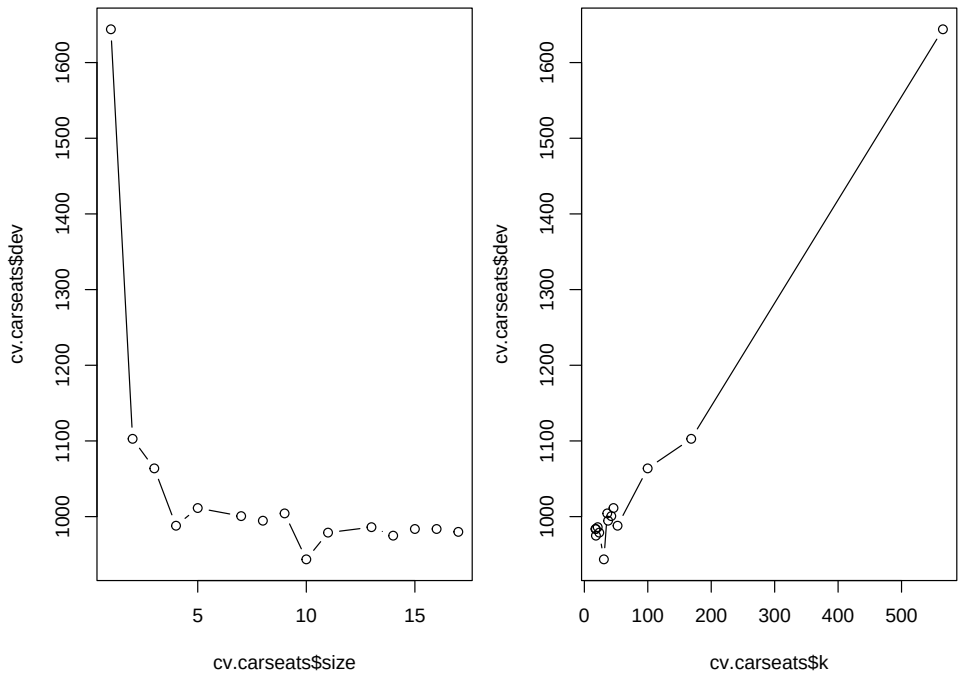
```
> cv.carseats = cv.tree(tree.carseats, FUN = prune.tree)
> par(mfrow = c(1, 2))
> plot(cv.carseats$size, cv.carseats$dev, type = "b")
> plot(cv.carseats$k, cv.carseats$dev, type = "b")
> best.size <- cv.carseats$size[which.min(cv.carseats$dev)]
> best.size

[1] 10
```

```
> pruned.carseats = prune.tree(tree.carseats, best = best.size)
> par(mfrow = c(1, 1))
> plot(pruned.carseats)
> text(pruned.carseats, pretty = 0)
> pred.pruned = predict(pruned.carseats, Carseats.test)
> mean((Carseats.test$Sales - pred.pruned)^2)

[1] 5.358271

>
```



The test MSE of the pruned tree is 5.358271, so pruning the tree didn't improve the test MSE. But note that it is not always true, sometimes pruning the tree can improve the test MSE. For example, when use "set.seed(10)" (so we will have different training and testing sets), you will see that pruning improves the test MSE.

d)

```
> library(randomForest)
> # use the bagging approach to fit a model
> bag.carseats = randomForest(Sales ~ ., data = Carseats.train, mtry = p-1, ntree = 500,
 importance = T)
> bag.pred = predict(bag.carseats, Carseats.test)
> mean((Carseats.test$Sales - bag.pred)^2)

[1] 3.315656

> importance(bag.carseats)

               %IncMSE IncNodePurity
CompPrice   23.3746577    161.519653
Income       6.9992800     89.094905
Advertising 23.5313121    191.145427
Population   0.5537334     57.860247
Price       46.6265611    349.999843
ShelveLoc   69.4954039    602.716975
Age          7.5820194     94.514851
```

```
Education     3.2086797     34.291840
Urban        -0.9076486      5.082664
US            0.8498543      6.547575
```

The test MSE of bagging is 3.315656, and one can see that variables "ShelveLoc", "Price", "Advertising" and "CompPrice" are the 4 most important variables for predicting "Sales".

e)

```
> # First we fit the randomForest using the default "mtry". Note that by default,
> # randomForest() uses p/3 variables when building a random forest of
> # regression trees, and /sqrt(p) variables when building a random forest of
> # classification trees.
>
>
> rf.carseats = randomForest(Sales ~ ., data = Carseats.train, ntree = 500, importance = T)
> rf.pred = predict(rf.carseats, Carseats.test)
> mean((Carseats.test$Sales - rf.pred)^2)

[1] 3.652072

> importance(rf.carseats)

               %IncMSE IncNodePurity
CompPrice   13.9573828     137.05401
Income       4.2797590     129.56698
Advertising 19.0990059     196.89319
Population   1.4010573     106.10286
Price       30.3575555     310.19452
ShelveLoc   42.9651175     407.37688
Age          6.6493877     129.65096
Education   -0.1957945      57.54397
Urban       -1.6583554      11.93021
US           4.5035641      40.98320
```

When using default value for "mtry", the test MSE of randomForest is 3.652072, and variables "ShelveLoc", "Price", "Advertising" and "CompPrice" are the 4 most important variables for predicting "Sales".

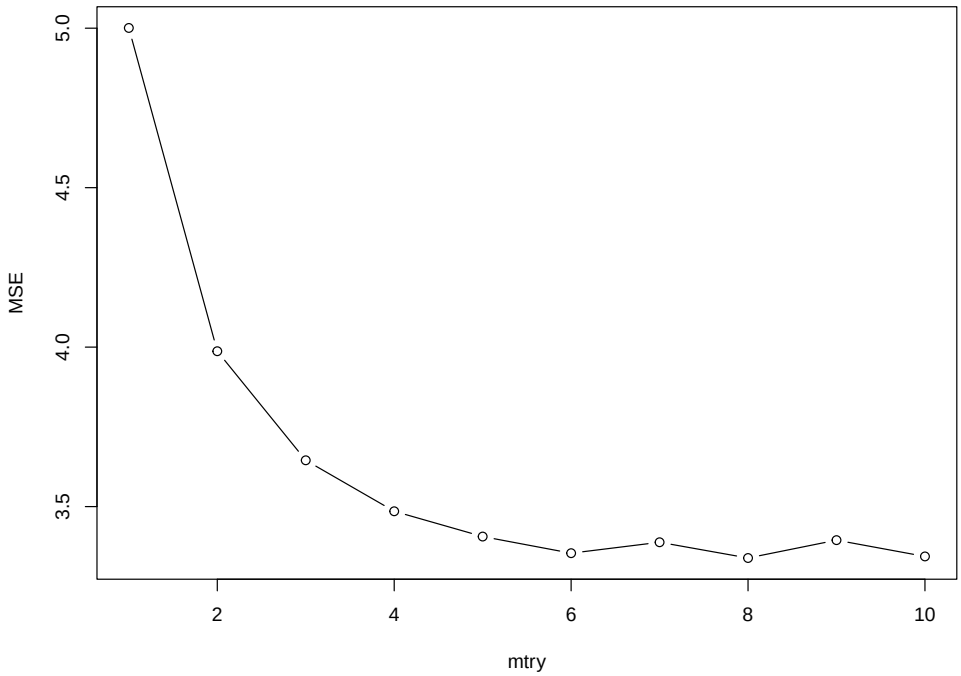
```
> # Now we try different "mtry"
> mse_vector <- rep(NA, p-1)
> for (i in 1:(p-1)) {
   rf.carseats = randomForest(Sales ~ ., data = Carseats.train, mtry=i, ntree = 500,
                              importance = T)
   rf.pred = predict(rf.carseats, Carseats.test)
   mse_vector[i] <- mean((Carseats.test$Sales - rf.pred)^2)
 }
> plot(1:(p-1), mse_vector, xlab="mtry", ylab="MSE", type="b")
> which.min(mse_vector)

[1] 8

> min(mse_vector)

[1] 3.338769

>
```



One can see that using different "mtry"s gives different test MSEs. In the above case, "mtry=8" gives the mininmal test MSE 3.338769.