Assignment #5

HANDWRITTEN DIGIT RECOGNITION

20190552 손지영

사용 데이터

- **MNIST** : 손으로 쓰여진 숫자들의 이미지 70,000개로 구성

```
library(dslabs)
library(ggplot2)
library(caret)
library(rpart)
library(rpart.plot)
library(randomForest)
library(tidyr)
library(gridExtra)
```

1번

아래의 순서에 따라 data preprocessing을 수행하자.

1.(A) ~ (C)

```
# 데이터 불러오기
nist = read_mnist()
str(nist)
```

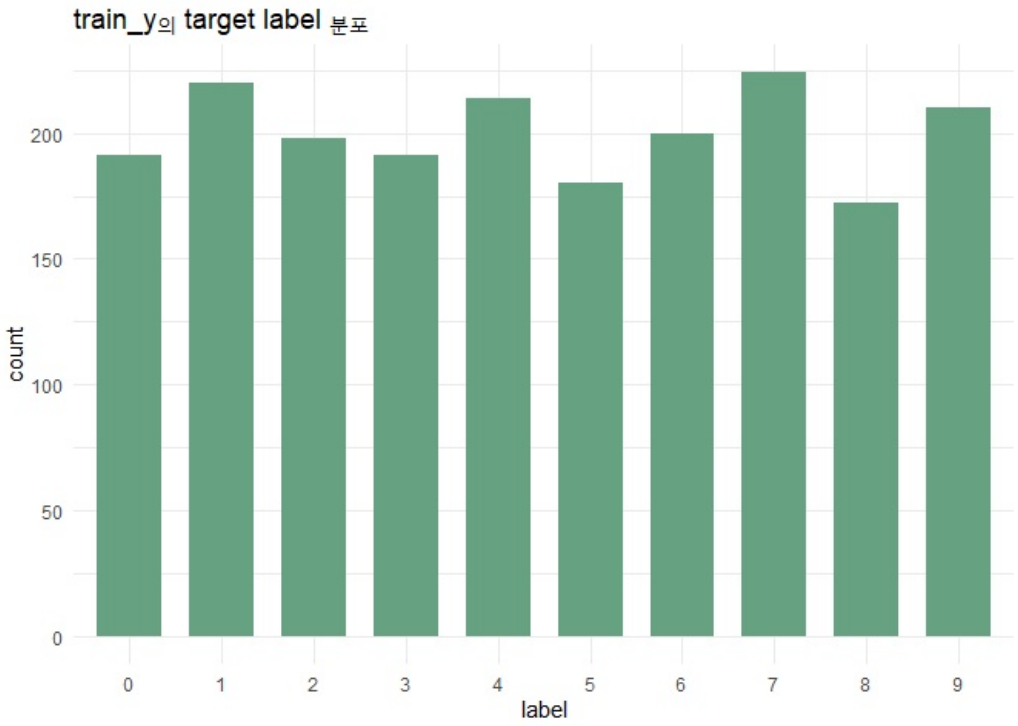
```
## List of 2
##  $ train:List of 2
##   ..$ images: int [1:60000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ labels: int [1:60000] 5 0 4 1 9 2 1 3 1 4 ...
##  $ test :List of 2
##   ..$ images: int [1:10000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
##   ..$ labels: int [1:10000] 7 2 1 0 4 1 4 9 5 9 ...
```

```
# 처음 2000개만 train으로 저장하기
train_x = nist$train$images[1:2000, ]
train_y = nist$train$labels[1:2000]

# 열 이름 변경하기
colnames(train_x) = paste0('V', 1:ncol(train_x))

# train_y 분포 확인하기
df_train_y = data.frame(label=train_y)
df_train_y$label = factor(df_train_y$label)
```

```
p1 = ggplot(df_train_y, aes(x = label)) +
  geom_bar(width = 0.7, position = "identity", fill = '#66a182') +
  labs(title='train_y의 target label 분포')+
  theme_minimal()
p1
```



막대 그래프를 통해 train_y의 분포를 확인해본 결과 각 클래스의 데이터 수는 거의 200정도에 분포해있음을 확인할 수 있다.

1.(D)

```
# 필요없는 열 찾기
del_col = nearZeroVar(train_x)
cnt = length(del_col)

# 삭제된 열의 개수
cnt
```

```
## [1] 540
```

```
# 남은 열의 개수
print(784-cnt)
```

```
## [1] 244
```

nearZeroVar 함수를 사용하여 column을 삭제한 결과 총 540개의 열이 삭제되었음을 확인할 수 있다. 따라서 앞으로 244개의 열을 가지고 예측을 진행하게 된다.

1.(E) ~ (F)

```
# train 데이터에서 필요없는 열 삭제
del_train_x = subset(train_x, select=-del_col)
str(del_train_x)
```

```
##  int [1:2000, 1:244] 18 0 0 0 0 0 253 105 63 0 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:244] "V154" "V155" "V156" "V157" ...
```

```
train = as.data.frame(cbind(del_train_x, train_y))

# target이 class이기 때문에 factor로 변환하기
train$train_y = factor(train$train_y)
colnames(train)[245] = 'y'

# Test set은 10,000개의 데이터 셋을 모두 사용
test_x = nist$test$images
test_y = nist$test$labels

# 동일하게 처리하기
colnames(test_x) = paste0('V', 1:ncol(test_x))
del_test_x = subset(test_x, select=-del_col)
test = as.data.frame(cbind(del_test_x, test_y))
test$test_y = factor(test$test_y)
colnames(test)[245] = 'y'
```

```
# test_y의 분포도 살펴보기
df_test_y = data.frame(label=test_y)
df_test_y$label = factor(df_test_y$label)
p2 = ggplot(df_test_y, aes(x = label)) +
  geom_bar(width = 0.7, position = "identity", fill = '#66a182') +
  labs(title='test_y의 target label 분포')+
  theme_minimal()
grid.arrange(p1, p2, ncol = 2)
```

추가적으로 test_y에서의 분포도 함께 비교해보았다. test_y에서 모든 클래스의 수가 비슷하지만 1번 클래스만 데이터 수가 유독 많다는 것을 확인할 수 있으며, train_y와 데이터 양상은 다소 다른 것을 확인할 수 있다.

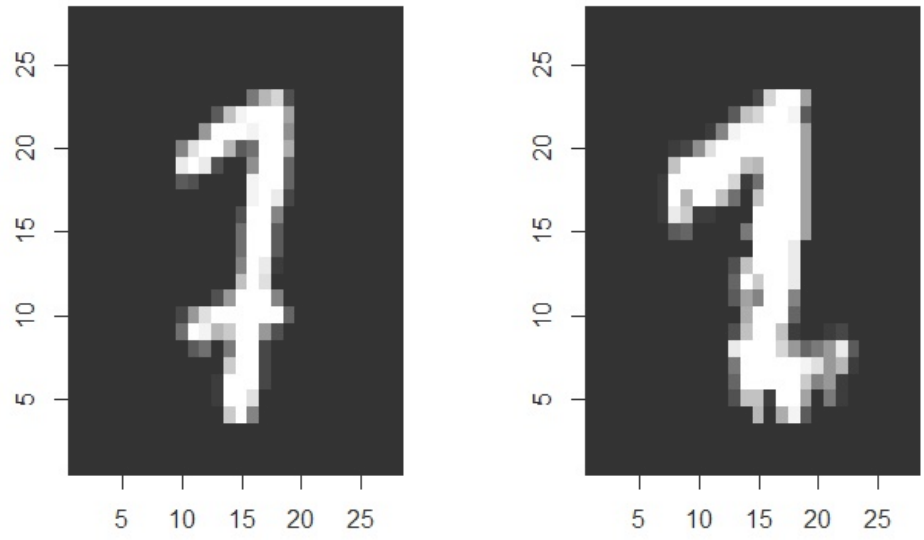
2번

test set의 image 행렬의 행 번호를 입력받아 숫자 이미지를 출력하는 함수 print_image()를 만들어보자. 이 함수를 활용하여 test set 중에

서 이미지로부터 실제 숫자값을 유추하기 어려운 예를 몇 개 찾아보자.

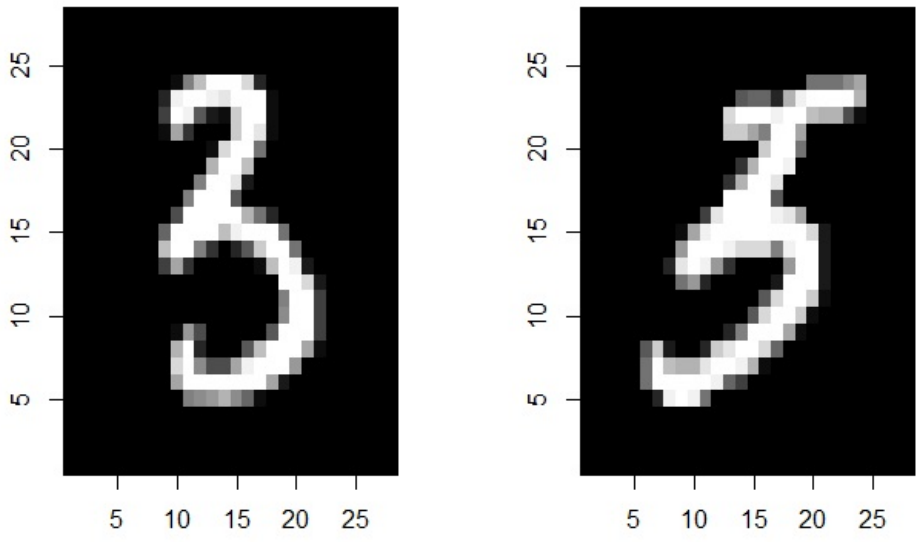
```
# 이미지 출력 함수 만들기
print_image = function(x) {
  image(1:28, 1:28, matrix(nist$test$images[x,], nrow=28)[ , 28:1],
        col = gray(seq(0, 1, 0.05)), xlab = "", ylab="")
}

# 7과 2가 비슷한 이미지
par(mfrow = c(1,2))
print_image(283) # 7
print_image(660) # 2
```



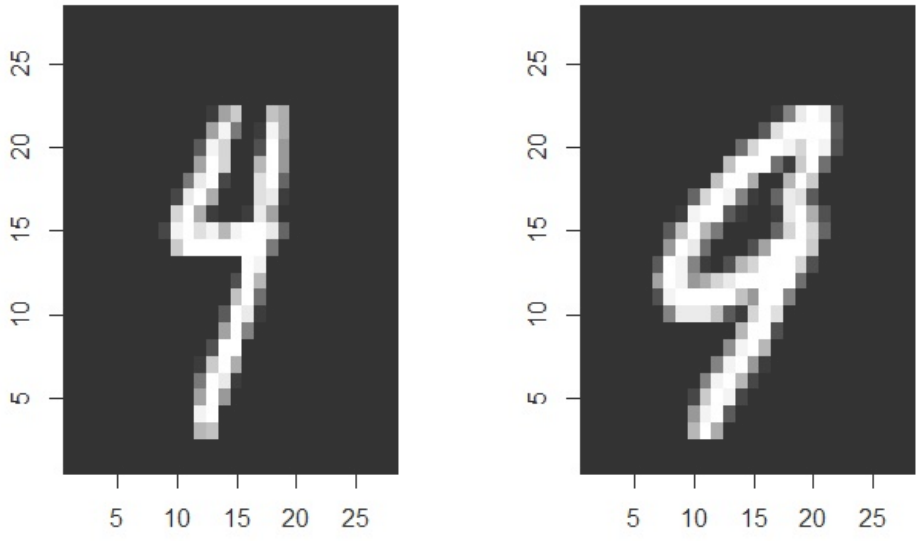
왼쪽 이미지는 7, 오른쪽 이미지는 2로 두 모양이 거의 비슷하여 구분하기 어렵다.

```
# 3과 5가 비슷한 이미지
par(mfrow = c(1,2))
print_image(2595) # 3
print_image(2036) # 5
```



왼쪽 이미지는 3, 오른쪽 이미지는 5로 두 모양이 거의 유사하여 구분하기 어렵다.

```
# 4와 9가 비슷한 이미지
par(mfrow = c(1,2))
print_image(4568) # 4
print_image(7581) # 9
```



왼쪽 이미지는 4, 오른쪽 이미지는 9로 실제 4가 9처럼 보이기도 한 것을 확인할 수 있다.

3번

아래의 순서로 tree를 만들어보자.

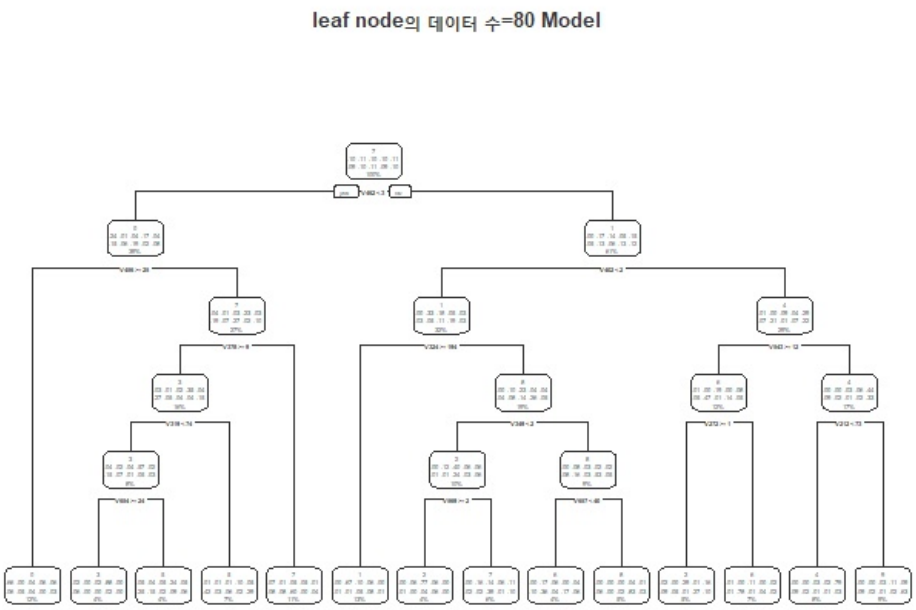
3.(A)

Cost complexity parameter α=0 일때, leaf node가 가지는 최소 데이터의 수가 80인 Tree를 만들고 시각화해보자. Tree는 몇 개의 leaf node를 가지는가? Tree의 depth는 얼마인가?

```
# y를 factor 변환
train$y <- as.factor(train$y)

# Decision Tree 생성
set.seed(1); ct = rpart(y~., data=train, method='class',
           control=list(cp=0, minbucket=80))

# Tree 그림 시각화
rpart.plot(ct, main='leaf node의 데이터 수=80 Model')
```



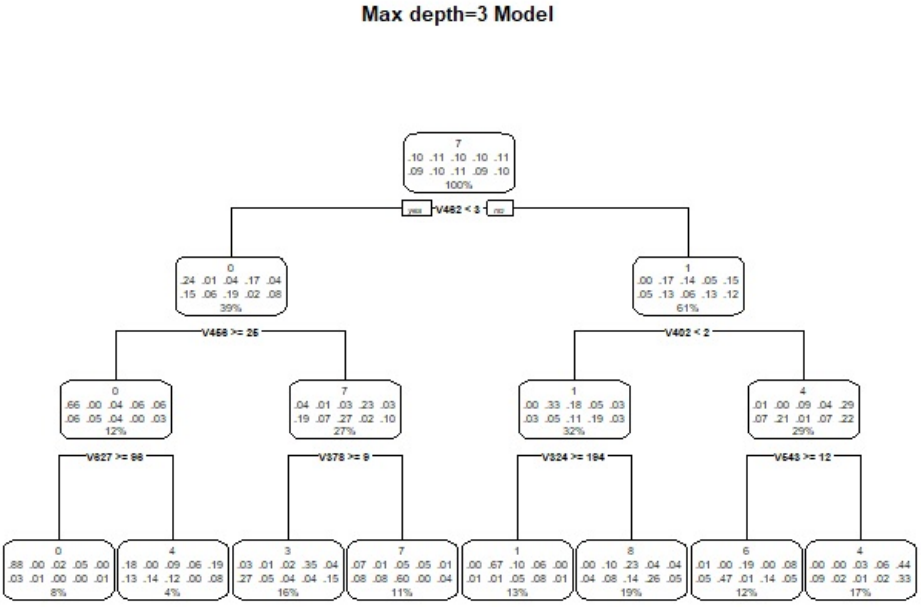
위의 그림을 통해 총 14개의 leaf node를 가진다는 것을 알 수 있다. 또한 depth는 2, 4, 5개를 가진 부분 모두 존재한다는 것을 확인할 수 있다.

3.(B)

Cost complexity parameter α=0 일때, depth가 최대 3인 Tree를 만들고 시각화해보자. Tree는 몇개의 leaf node를 가지는가? 만들어진 tree가 실제 classification에 활용될 수 있을까?

```
# maxdepth=3으로 하는 Tree 모델 생성
set.seed(1); ct = rpart(y~., data=train, method='class',
                        control=list(cp=0, maxdepth=3))

# Tree 그림 시각화
rpart.plot(ct, main='Max depth=3 Model')
```



해당 모델에서는 max depth를 3으로 제한했기 때문에 tree의 depth가 3개로 그려졌으며, 이에 따라 tree의 모양이 이전의 그림보다 더 단순해졌다. 또한 leaf node의 개수 역시 8개로 줄어들었다. 만들어진 tree 속 leaf node의 결과를 비교해보면 모든 class가 분리되지 않은 것을 확인할 수 있다.

예를 들어, 그림 속에서 4의 클래스로 분리된 leaf node는 2개이다. 먼저 4로 분리된 8번째 위치한 leaf node의 경우 class 4가 0.44로 월등하게 높다. 하지만, 2번째에 위치한 leaf node의 경우 class 4가 0.19로 가장 높긴하지만, class 0이 0.18, class 5가 0.13, class 6이 0.14로 class 4의 확률이 월등하게 높다고 보기 어렵다. 또한 class 2, 5, 9는 leaf node로 분리되지 않아 해당 모델의 너무 단순하다는 것을 알 수 있다.

따라서 총 10개의 클래스를 분리해야하는 조건에서 해당 tree는 적합하지 않다고 판단된다.

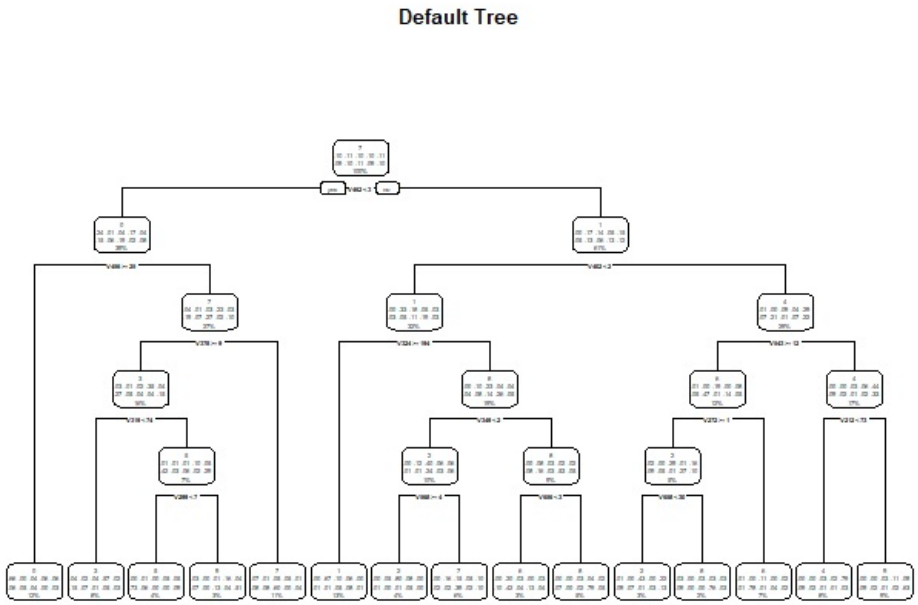
3.(C)

rpart() 함수의 default 옵션으로 Tree를 만든 후 cross validation을 활용한 pruning 과정을 수행해보자.

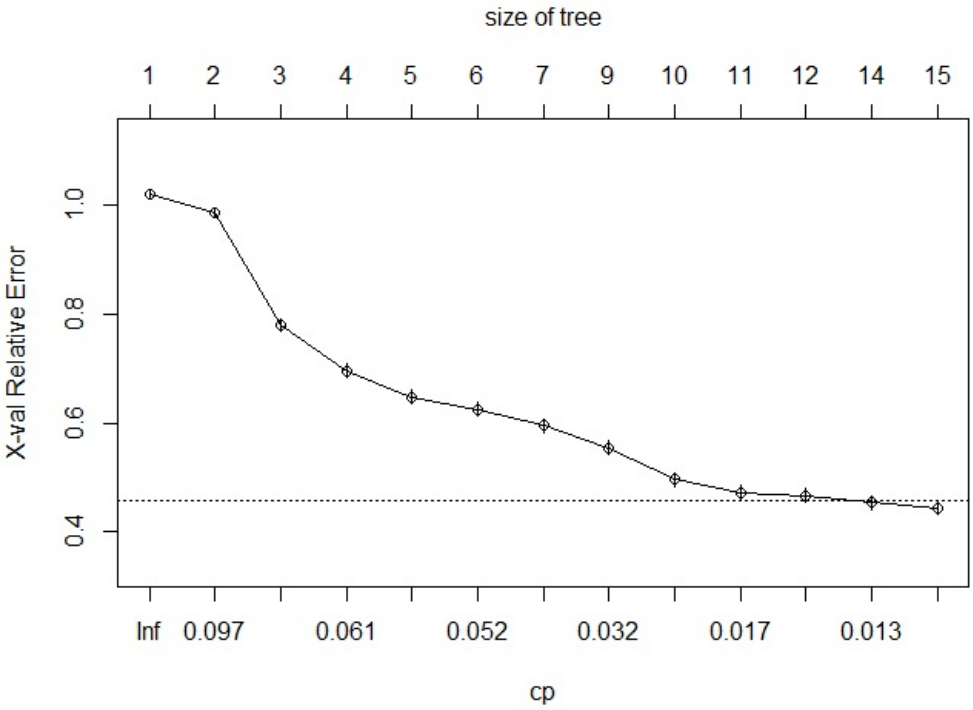
Default Tree

```
# default 모델 생성
set.seed(1); ct = rpart(y~., data=train, method='class')

# Tree 그림 시각화
rpart.plot(ct, main='Default Tree')
```




```
# 최적 cp 시각화
plotcp(ct)
```




```
# cp에 따른 xerror 결과 출력
printcp(ct)
```

```
##
## Classification tree:
## rpart(formula = y ~ ., data = train, method = "class")
##
## Variables actually used in tree construction:
##  [1] V212 V272 V299 V319 V324 V349 V378 V402 V456 V462 V543 V568 V656 V658
##
## Root node error: 1776/2000 = 0.888
##
## n= 2000
##
```

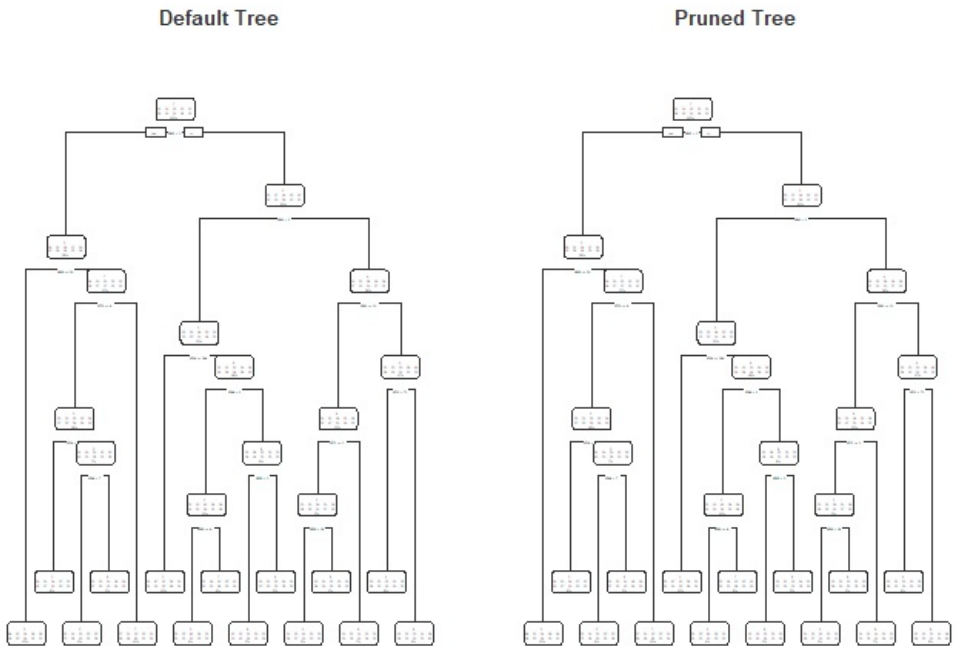
```
##          CP nsplit rel error  xerror      xstd
## 1  0.099099      0   1.00000 1.02027 0.0073485
## 2  0.094032      1   0.90090 0.98761 0.0082704
## 3  0.067005      2   0.80687 0.77928 0.0116252
## 4  0.056306      3   0.73986 0.69595 0.0122348
## 5  0.052365      4   0.68356 0.64809 0.0124461
## 6  0.051802      5   0.63119 0.62500 0.0125141
## 7  0.038570      6   0.57939 0.59572 0.0125693
## 8  0.025901      8   0.50225 0.55405 0.0125889
## 9  0.016892      9   0.47635 0.49606 0.0125010
## 10 0.016329     10   0.45946 0.47241 0.0124262
## 11 0.014077     11   0.44313 0.46678 0.0124050
## 12 0.011261     13   0.41498 0.45327 0.0123488
## 13 0.010000     14   0.40372 0.44369 0.0123043
```

plot 결과와 print 결과값을 통해 xerror를 가장 작게하는 최적 cp(α)는 0.01임을 확인할 수 있다. 또한 1-se rule을 적용하면, plot에서 점선 아래에 해당되는 tree 중 가장 크기가 작은 tree 역시 cp(α)=0.01인 것을 알 수 있다. 따라서 다음으로 최적 cp(α)에 해당하는 tree를 best model로 하여 결과를 비교해보겠다.

Pruned Tree

```
# xerror를 가장 작게 하는 cp 선택
best_cp = ct$cptable[which.min(ct$cptable[, 'xerror']), 'CP']
best_ct = prune(ct, cp=best_cp)

# 두 모델 비교하여 시각화
par(mfrow = c(1, 2))
rpart.plot(ct, main='Default Tree')
rpart.plot(best_ct, main='Pruned Tree')
```



먼저 xerror를 최소로 갖는 model을 저장하고 prune 함수를 통해 해당 모델의 pruned tree를 생성하였다. 그 후, plot으로 두 tree를 비교해본 결과 default tree와 pruned tree의 모습이 같은 것을 확인하였다. 이와 같은 경우는 원래 모델이 이미 최적화된 모델이여서 모델의 단순화가 크게 효과가 없다는 것을 의미한다.

3.(D)

C에서 얻은 tree로 test set에 대한 예측을 수행하고, confusion matrix를 계산해보자. Test set에 대한 예측 정확도는 얼마인가?

Accuracy

```
# test set에 대한 예측
pred = predict(best_ct, newdata=test, type='class')
cm = confusionMatrix(factor(pred), test$y)
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 760   0  43  38  45  54 113  30  20  40
##          1   0 856 181  90   5  15   7  54 143  17
##          2   5  28 454  17  46  33  52  32  21  25
##          3  20  11  24 466  30 110  42   4  33  51
##          4   1   0  19  18 531  64  29  15  23  61
##          5   6   2   1  61  12 269  18   3  11  37
##          6   1  96  48  15  42  36 566   4  77  23
##          7 158 136 237 111  91  82 108 845  23 110
##          8  12   5   6  32  23  70  11  14 541  27
##          9  17   1  19 162 157 159  12  27  82 618
##
## Overall Statistics
##
##                Accuracy : 0.5906
##                  95% CI : (0.5809, 0.6003)
##     No Information Rate : 0.1135
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5444
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.7755   0.7542   0.4399   0.4614   0.5407   0.3016
## Specificity            0.9575   0.9422   0.9711   0.9638   0.9745   0.9834
## Pos Pred Value         0.6649   0.6257   0.6367   0.5891   0.6978   0.6405
## Neg Pred Value         0.9752   0.9677   0.9378   0.9409   0.9512   0.9350
## Prevalence             0.0980   0.1135   0.1032   0.1010   0.0982   0.0892
## Detection Rate         0.0760   0.0856   0.0454   0.0466   0.0531   0.0269
## Detection Prevalence   0.1143   0.1368   0.0713   0.0791   0.0761   0.0420
## Balanced Accuracy      0.8665   0.8482   0.7055   0.7126   0.7576   0.6425
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.5908   0.8220   0.5554   0.6125
## Specificity            0.9622   0.8823   0.9778   0.9293
## Pos Pred Value         0.6233   0.4445   0.7301   0.4928
## Neg Pred Value         0.9569   0.9774   0.9532   0.9553
## Prevalence             0.0958   0.1028   0.0974   0.1009
## Detection Rate         0.0566   0.0845   0.0541   0.0618
## Detection Prevalence   0.0908   0.1901   0.0741   0.1254
## Balanced Accuracy      0.7765   0.8521   0.7666   0.7709
```

```
# Accuracy 출력
cm$overall["Accuracy"]
```

```
## Accuracy
##   0.5906
```

default 모델에서 pruning 과정을 통해 얻은 tree로 test set 예측을 수행한 결과, 모델의 정확도는 0.591로 구할 수 있었다.

Balanced Accuracy

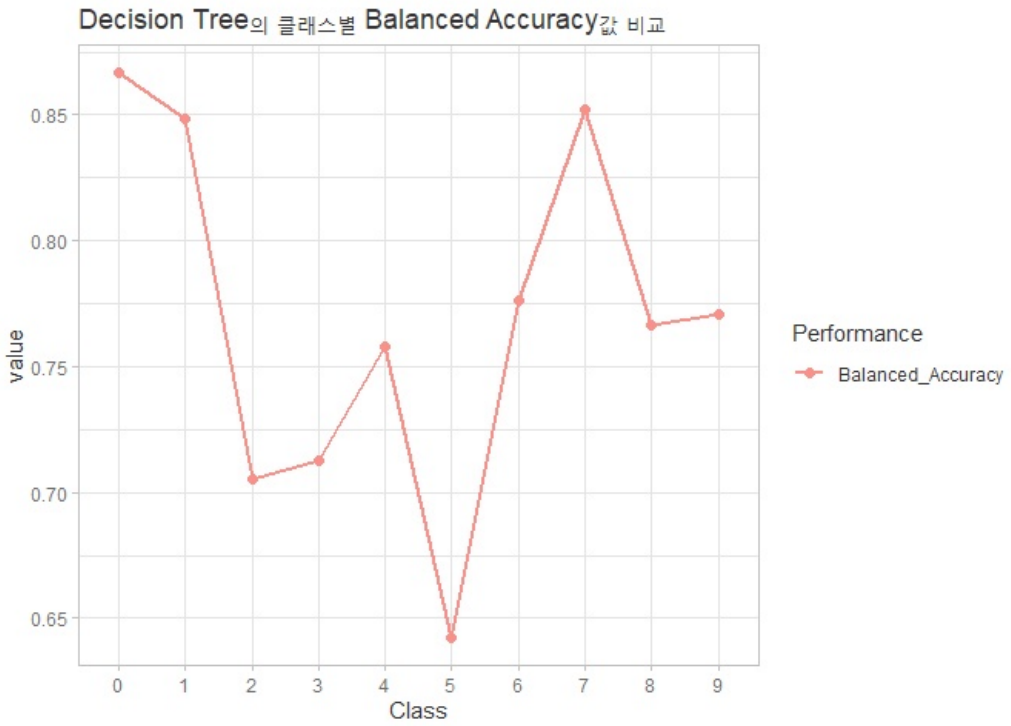
추가적으로 class별로 Balanced Accuracy을 비교해보았다.

(Balanced Accuracy는 다중 클래스 분류에서 클래스의 불균형한 분포를 고려하여 모델의 성능을 종합적으로 평가하는 지표이다.)

```
# class별 Balanced_Accuracy값 저장
df_result = data.frame(Class=0:9,
                       Balanced_Accuracy = unname(cm$byClass[, 11]))

# plot을 그리기 위해 데이터 형태 변환
df_gather = gather(df_result, Performance, value, Balanced_Accuracy)

# class에 따른 Balanced_Accuracy값 시각화
ggplot(df_gather, aes(x=Class, y=value, color=Performance)) +
  geom_point(size=2) +
  geom_line(size=1)+
  theme_light() +
  scale_x_discrete(limits = c(0:9)) +
  labs(title='Decision Tree의 클래스별 Balanced Accuracy값 비교')
```



Balanced Accuracy는 클래스의 불균형을 고려하는 지표로 Accuracy와 구하는 방식이 조금 다르다. 따라서 모델의 전체 Accuracy에 어떤 변수가 안좋은 영향을 끼쳤는지 정확히 판단할 수는 없지만 그럼에도 Balanced Accuracy가 아주 낮은 클래스는 전체적인 성능에 안좋은 영향을 끼쳤다고 판단할 수 있다. 따라서 모델이 class 2, 3, 5에서 다른 class에 비해 더욱 예측을 못했음을 알 수 있다.

4번

Random Forest를 만들어보자.

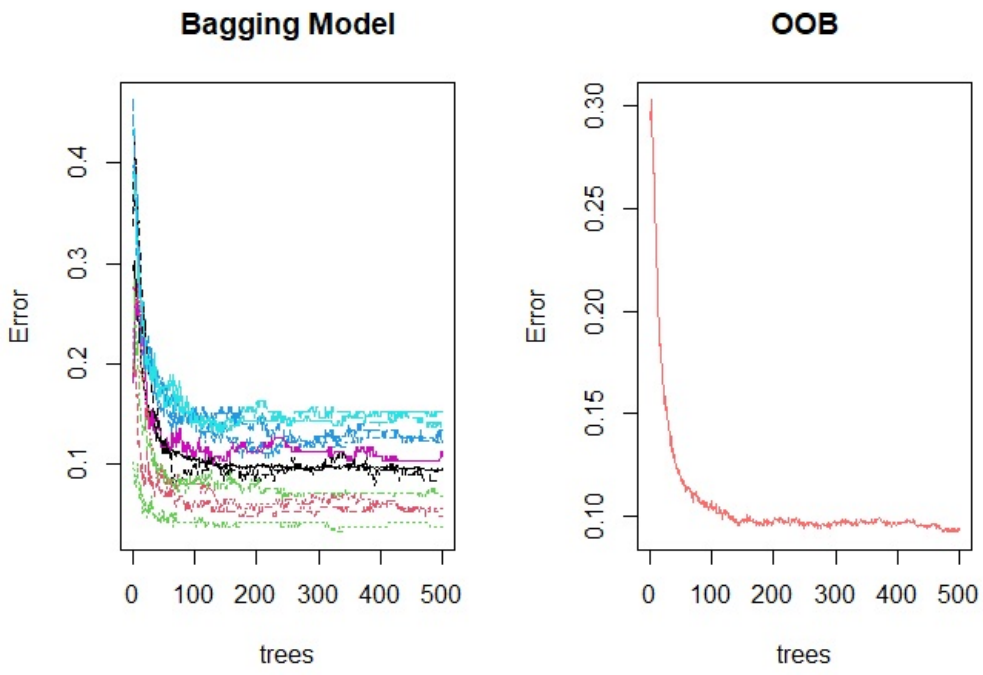
4.(A)

randomForest() 함수를 사용하여 bagging model을 만들어보자. mtry를 제외한 옵션은 모두 default 값을 사용한다. plot() 함수를 사용하여 Bagging model에서 tree의 수의 증가에 따른 OOB classification error rate의 변화를 그래프로 출력해보자. 어떤 경향을 보이는가?

```r
# bagging model 생성
set.seed(1); bag = randomForest(y~., data=train, mtry=2000)
bag
```

```
##
## Call:
##  randomForest(formula = y ~ ., data = train, mtry = 2000)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 244
##
##         OOB estimate of  error rate: 9.35%
## Confusion matrix:
##     0   1   2   3   4   5   6   7   8   9 class.error
## 0 182   1   0   0   0   2   1   1   4   0  0.04712042
## 1   0 212   4   1   0   1   1   1   0   0  0.03636364
## 2   3   1 174   2   5   1   3   6   2   1  0.12121212
## 3   4   0   2 164   1   9   2   2   3   4  0.14136126
## 4   2   2   1   1 191   2   5   0   0  10  0.10747664
## 5   0   0   1   3   3 165   2   0   0   6  0.08333333
## 6   2   1   2   0   3   2 189   1   0   0  0.05500000
## 7   1   3   1   0   4   0   0 209   1   5  0.06696429
## 8   0   5   5   5   0   2   2   0 149   4  0.13372093
## 9   3   0   1   8  10   1   2   6   1 178  0.15238095
```

```r
# 모델의 OOB classification error rate 변화율 시각화
par(mfrow = c(1, 2))
plot(bag, main='Bagging Model')
plot(x=1:500, y=bag$err.rate[, 'OOB'], type = 'l',
     xlab='trees', ylab='Error', col='Indian Red1',
     main='OOB')
```



training set에 대한 out-of-bag classification error rate를 나타낸 그림은 위와 같다. 위의 그림을 해석하면, tress의 수가 증가할수록 Error rate가 감소하는 것을 확인할 수 있다.

4.(B)

Bagging model로 test set에 대한 예측을 수행하고, confusion matrix를 계산해보자. Test set에 대한 예측 정확도는 얼마인가? 3번에서

계산한 tree model에 비해서 성능이 얼마나 향상되었는가?

```
# test set 예측
pred_bag = predict(bag, newdata=test, type='class')
cm_bag = confusionMatrix(pred_bag, test$y)
cm_bag
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  939    0   11    6    1   19   21    2    7    5
##          1    0 1105    2    1    2    8    2    7    5    4
##          2    1    8  928   33    5    0   24   33   20    8
##          3    2    6    9  822    2   22    0    6   24   18
##          4    3    0   12    2  863    8   25    2    9   25
##          5    9    1    8   84    3  790   12    0   24   17
##          6    9    4   18    3   17   15  864    0   28    8
##          7   11    1   24   29    2   11    2  913    7   14
##          8    3   10   14   18   11   11    7    3  803    3
##          9    3    0    6   12   76    8    1   62   47  907
##
## Overall Statistics
##
##                Accuracy : 0.8934
##                  95% CI : (0.8872, 0.8994)
##     No Information Rate : 0.1135
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8815
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9582   0.9736   0.8992   0.8139   0.8788   0.8857
## Specificity            0.9920   0.9965   0.9853   0.9901   0.9905   0.9827
## Pos Pred Value         0.9288   0.9727   0.8755   0.9023   0.9094   0.8333
## Neg Pred Value         0.9954   0.9966   0.9884   0.9793   0.9869   0.9887
## Prevalence             0.0980   0.1135   0.1032   0.1010   0.0982   0.0892
## Detection Rate         0.0939   0.1105   0.0928   0.0822   0.0863   0.0790
## Detection Prevalence   0.1011   0.1136   0.1060   0.0911   0.0949   0.0948
## Balanced Accuracy      0.9751   0.9850   0.9423   0.9020   0.9346   0.9342
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.9019   0.8881   0.8244   0.8989
## Specificity            0.9887   0.9887   0.9911   0.9761
## Pos Pred Value         0.8944   0.9004   0.9094   0.8084
## Neg Pred Value         0.9896   0.9872   0.9812   0.9885
## Prevalence             0.0958   0.1028   0.0974   0.1009
## Detection Rate         0.0864   0.0913   0.0803   0.0907
## Detection Prevalence   0.0966   0.1014   0.0883   0.1122
## Balanced Accuracy      0.9453   0.9384   0.9078   0.9375
```

```
cm_bag$overall["Accuracy"]
```

```
## Accuracy
##   0.8934
```

두 모델의 Accuracy를 표로 작성하면 아래와 같다.

Model	Accuracy

Decision Model	0.591
Bagging Model	0.893

위의 표를 통해 bagging model이 decision model에 비해 약 0.303 정도 높은 Accuracy를 갖음을 확인할 수 있다.

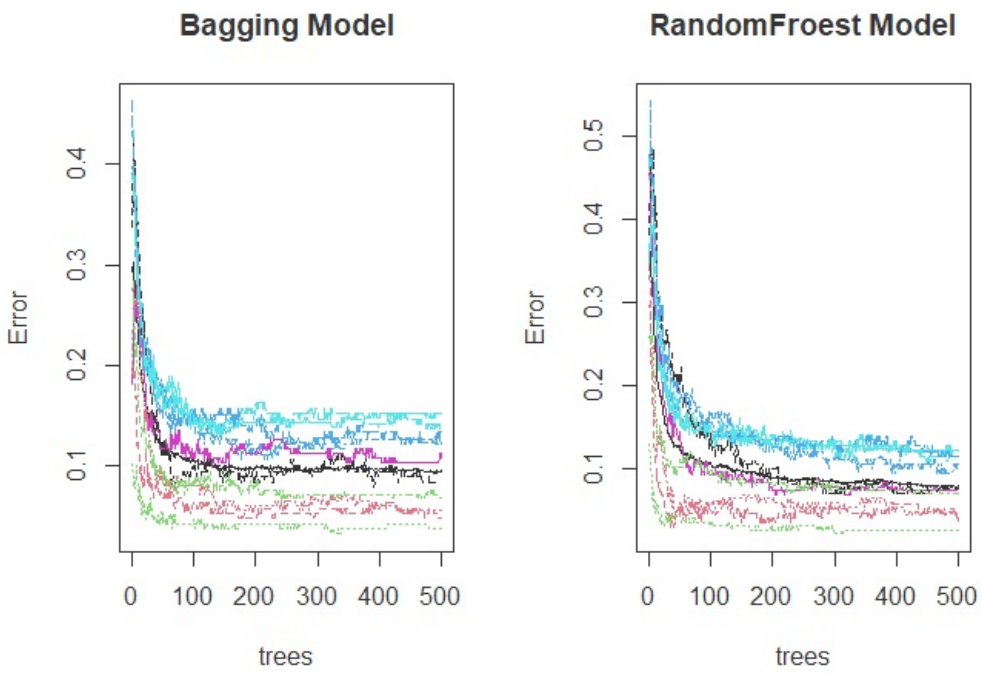
4.(C)

randomForest() 함수의 default 옵션을 사용하여 random forest model을 만들어보자. 그리고 Bagging과 random forest 모델의 Tree의 수의 증가에 따른 OOB classification error rate의 변화를 하나의 그래프에 그려보고 두 모델의 성능을 비교해보자.

전체 Error rate 비교

```
# randomforest model 생성
set.seed(1); rand = randomForest(y~., data=train)

# 두 모델에 대한 error rate 시각화
par(mfrow = c(1, 2))
plot(bag, main='Bagging Model')
plot(rand, main='RandomFroest Model')
```



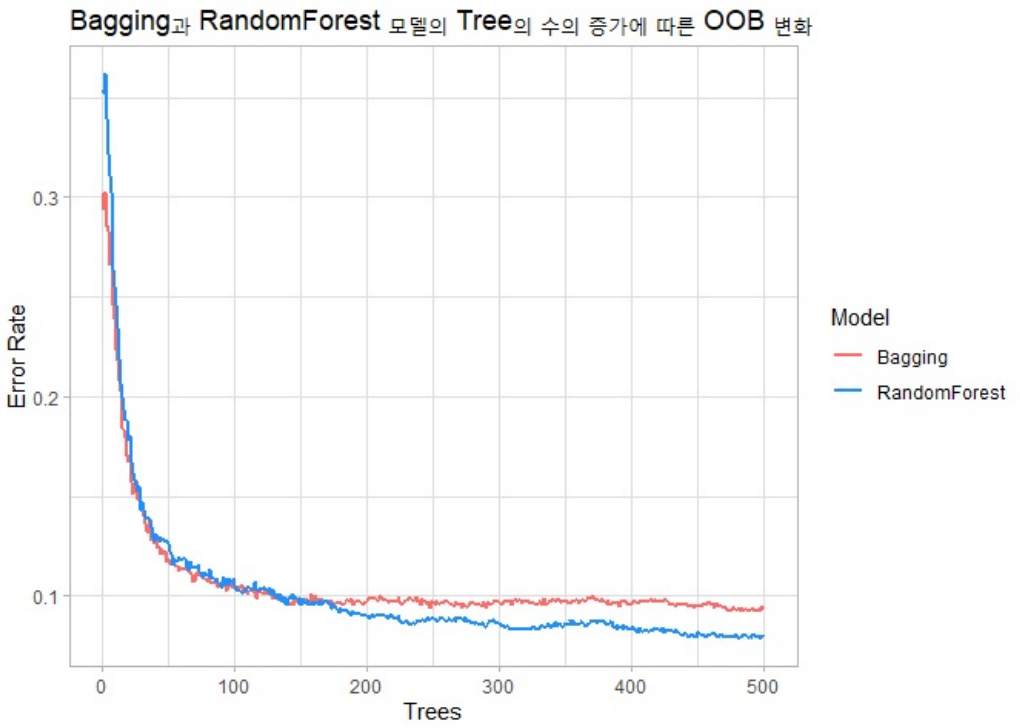
먼저 전체에 대한 error rate를 살펴본 결과, 두 모델 모두 tree의 개수가 커질수록 error rate가 감소하는 양상을 보인다. 또한 bagging 모델이 randomforest 모델에 비해 전체적으로 높은 error rate를 갖고 있는 것으로 보인다. 따라서 다음으로는 OOB열에 대한 값만 추출하여 비교해보도록 한다.

OOB값 비교

```
# 두 모델의 OOB열 값 저장
df = data.frame(num=1:500, bagging=bag$err.rate[, 'OOB'], random=rand$err.rate[, 'OOB'])

# 두 모델의 OOB값 변화 시각화
ggplot(df) +
  geom_line(aes(x=num, y=bagging, color='Bagging'), size=1) +
  geom_line(aes(x=num, y=random, color='RandomForest'), size=1) +
```

```
  scale_color_manual(values = c("Bagging" = "Indian Red1", "RandomForest" = "Dodger Blue")) +
  labs(x='Trees', y='Error Rate', color='Model',
       title='Bagging과 RandomForest 모델의 Tree의 수의 증가에 따른 OOB 변화') +
  theme_light()
```



error rate는 모델이 잘못 분류하는 오류의 비율을 나타내는 것으로 해당 값이 낮을수록 예측이 더 정확한 모델이라고 할 수 있다. 만약 타겟 데이터의 클래스 불균형이 심하다면 error rate가 더 낮다고 해서 해당 모델이 더 우수하다고 판단할 수 없지만, 일반적으로 모델의 성능이 더 우수하다고 추측할 수 있다. 따라서 위의 과정을 통해 class의 불균형이 심하지 않다는 것을 확인했기 때문에 error rate가 더 낮은 randomforest 모델이 조금 더 우수할 것이라 추측할 수 있다.

4.(D)

Random forest model로 test set에 대한 예측을 수행하고, confusion matrix를 계산해보자. Test set에 대한 예측 정확도는 얼마인가? Bagging model에 비해서 성능이 얼마나 향상되었는가?

```
# 예측 수행
pred_rand = predict(rand, newdata=test, type='class')
cm_rand = confusionMatrix(factor(pred_rand), test$y)
cm_rand
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  954    0    7    2    1   15   18    1    9    7
##          1    0 1112    2    0    1   10    4   12    3    5
##          2    3    3  942   22    2    2   26   36   10    8
##          3    0    3    7  884    0   20    1    1   20   15
##          4    3    0   14    3  899    8   19    2   16   22
##          5    7    3    7   57    0  793    9    1   21   11
##          6    7    5   15    4   13   13  879    1   19    4
##          7    4    1   20   18    1    9    0  930    8   13
##          8    2    8   14   12    4    7    2    3  822    2
##          9    0    0    4    8   61   15    0   41   46  922
##
## Overall Statistics
##
##                Accuracy : 0.9137
```

```
##                  95% CI : (0.908, 0.9191)
##     No Information Rate : 0.1135
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9041
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.9735   0.9797   0.9128   0.8752   0.9155   0.8890
## Specificity            0.9933   0.9958   0.9875   0.9925   0.9904   0.9873
## Pos Pred Value         0.9408   0.9678   0.8937   0.9295   0.9118   0.8724
## Neg Pred Value         0.9971   0.9974   0.9899   0.9861   0.9908   0.9891
## Prevalence             0.0980   0.1135   0.1032   0.1010   0.0982   0.0892
## Detection Rate         0.0954   0.1112   0.0942   0.0884   0.0899   0.0793
## Detection Prevalence   0.1014   0.1149   0.1054   0.0951   0.0986   0.0909
## Balanced Accuracy      0.9834   0.9878   0.9502   0.9339   0.9529   0.9381
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity            0.9175   0.9047   0.8439   0.9138
## Specificity            0.9910   0.9918   0.9940   0.9805
## Pos Pred Value         0.9156   0.9263   0.9384   0.8405
## Neg Pred Value         0.9913   0.9891   0.9833   0.9902
## Prevalence             0.0958   0.1028   0.0974   0.1009
## Detection Rate         0.0879   0.0930   0.0822   0.0922
## Detection Prevalence   0.0960   0.1004   0.0876   0.1097
## Balanced Accuracy      0.9543   0.9482   0.9190   0.9472
```

```
cm_rand$overall["Accuracy"]
```

```
## Accuracy
##   0.9137
```

두 모델의 Accuracy를 표로 작성하면 아래와 같다.

Model	Accuracy
Bagging Model	0.893
Random Forest	0.914

위의 표를 통해 randomforest model이 bagging model에 비해 0.0203 정도 성능이 향상된 것을 확인할 수 있다.

4.(E)

D번의 confusion matrix 결과로부터, 분류가 가장 정확한 숫자는 몇인가? 가장 분류가 어려운 숫자는 몇인가?

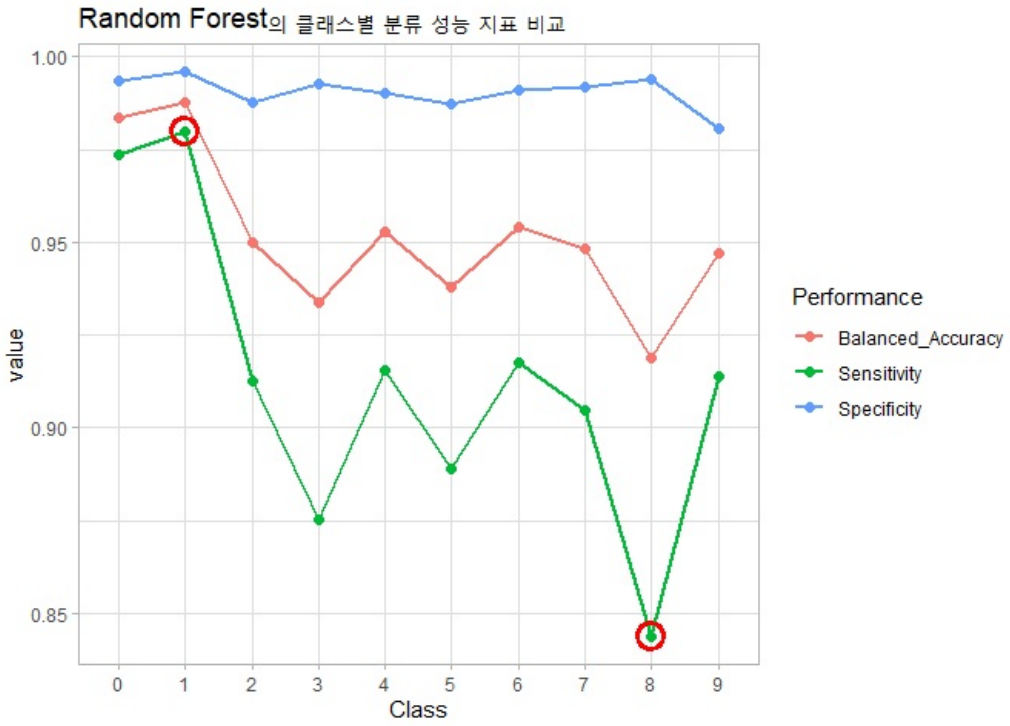
Random Forest

```
# 시각화를 위한 성능지표값 저장
df_result = data.frame(Class=0:9,
                       Sensitivity = unname(cm_rand$byClass[, 1]),
                       Specificity = unname(cm_rand$byClass[, 2]),
                       Balanced_Accuracy = unname(cm_rand$byClass[, 11]))
df_gather = gather(df_result, Performance, value, Sensitivity, Specificity, Balanced_Accuracy)

# 빨간 동그라미를 표시하기 위한 값 저장
max_class = df_result$Class[which.max(df_result$Sensitivity)]
```

```
max_val = max(df_result$Sensitivity)
min_class = df_result$Class[which.min(df_result$Sensitivity)]
min_val = min(df_result$Sensitivity)

# Random Forest의 클래스별 분류 성능 지표 시각화
ggplot(df_gather, aes(x=Class, y=value, color=Performance)) +
  geom_point(size=2) +
  geom_line(size=1)+
  theme_light() +
  scale_x_discrete(limits = c(0:9)) +
  labs(title='Random Forest의 클래스별 분류 성능 지표 비교') +
  geom_point(aes(x = max_class, y = max_val, shape = as.factor(max_class)),
             size = 5, color = "red", shape=1, stroke=2) +
  geom_point(aes(x = min_class, y = min_val, shape = as.factor(min_class)),
             size = 5, color = "red", shape=1, stroke=2)
```



먼저 각 클래스 별로 성능을 비교하기 위해 Balanced Accuracy, Sensitivity, Specificity 3가지를 비교해보았다. 그 결과 specificity의 각 클래스별 차이는 크지 않고, balanced accuracy와 sensitivity는 클래스별로 값이 매우 다름을 확인할 수 있었다. 또한 Sensitivity는 class 8에서 최소값을 가지고 Specificity는 class 2에서 최소값을 가진다는 사실을 통해 성능마다 최소값을 가지는 class가 다르다는 것을 알 수 있다. 따라서 한가지의 지표를 선정하여 분류 성능을 판단해야한다.

문제에서 요구한 "분류가 가장 정확한"의 의미를 고려하였을때 각 클래스에 대한 Positivie 예측 성능을 평가하는 Sensitivity값이 적절할 것이라 판단하였다. 따라서 Sensitivity값을 기준으로 값이 가장 큰 class 1이 분류가 가장 정확한 숫자이며, class 8이 분류가 가장 어려운 숫자임을 알 수 있다.

이미지 확인

위의 과정을 통해 8이 가장 분류하기 어려운 숫자임을 확인하였다. 따라서 confusionMatrix를 통해 8을 잘못 예측한 이미지 몇 개를 확인해보고자 한다.

```
# confusionmatrix
cm_rand$table
```

```
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  954    0    7    2    1   15   18    1    9    7
##          1    0 1112    2    0    1   10    4   12    3    5
##          2    3    3  942   22    2    2   26   36   10    8
```

```
##          3    0    3    7  884    0   20    1    1   20   15
##          4    3    0   14    3  899    8   19    2   16   22
##          5    7    3    7   57    0  793    9    1   21   11
##          6    7    5   15    4   13   13  879    1   19    4
##          7    4    1   20   18    1    9    0  930    8   13
##          8    2    8   14   12    4    7    2    3  822    2
##          9    0    0    4    8   61   15    0   41   46  922
```

위의 confusionmatrix를 살펴보면, 9와 5에서 잘못 예측한 비율이 높기 때문에 해당 숫자의 이미지를 확인해보도록 한다. 먼저 잘못 9와 5로 잘못 예측한 인덱스를 구한다.

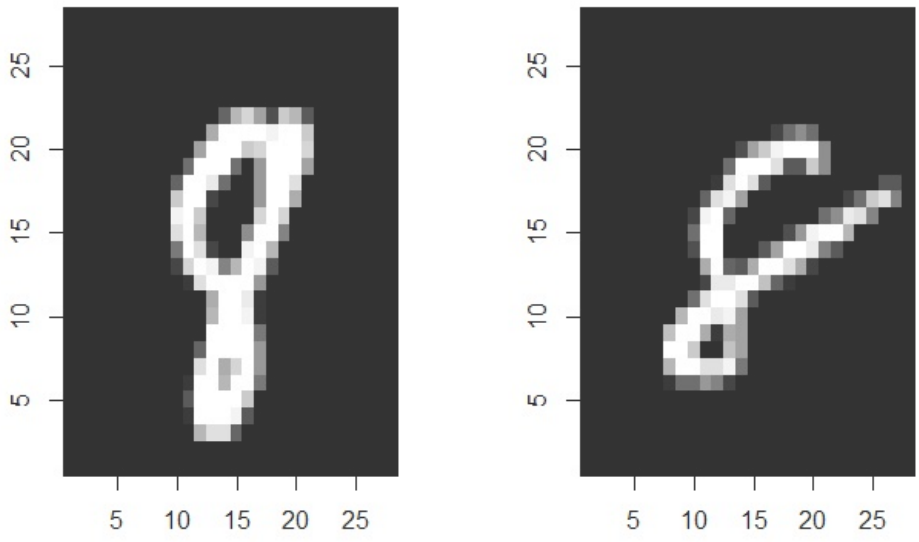
```
# 실제는 8이지만 9로 예측한 인덱스 출력
lst_sub = subset(test, test$y==8 & pred_rand==9)
row.names(lst_sub)
```

```
##  [1] "234"  "404"  "487"  "573"  "948"  "1069" "1102" "1126" "1150" "1186"
## [11] "1268" "1783" "1969" "2100" "2150" "2234" "2382" "2424" "2991" "3205"
## [21] "3264" "3290" "3320" "3952" "3955" "3965" "4142" "4640" "4749" "4864"
## [31] "5279" "5361" "5381" "5424" "6005" "6025" "6556" "6604" "6899" "7095"
## [41] "7122" "8295" "8298" "9137" "9876" "9893"
```

```
# 실제는 8이지만 5로 예측한 인덱스 출력
lst_sub = subset(test, test$y==8 & pred_rand==5)
row.names(lst_sub)
```

```
##  [1] "269"  "1201" "1235" "1365" "1760" "1775" "1814" "2759" "3066" "3560"
## [11] "3568" "3728" "3772" "5210" "5679" "6496" "6642" "6726" "8477" "8531"
## [21] "9281"
```

```
# 이미지 출력
par(mfrow = c(1,2))
print_image(7122) # 9
print_image(1760) # 5
```



먼저 왼쪽 그림은 실제는 8이지만 아래 부분이 작아 9처럼 보이기도 한다. 다음으로 오른쪽 그림은 위의 동그라미가 벌어져서 5의 윗부분처럼 보이기도 한다. 해당 이미지들은 눈으로는 쉽게 판단이 가능하지만 모델로 구별이 어려울 수 있을 것이라 생각된다.

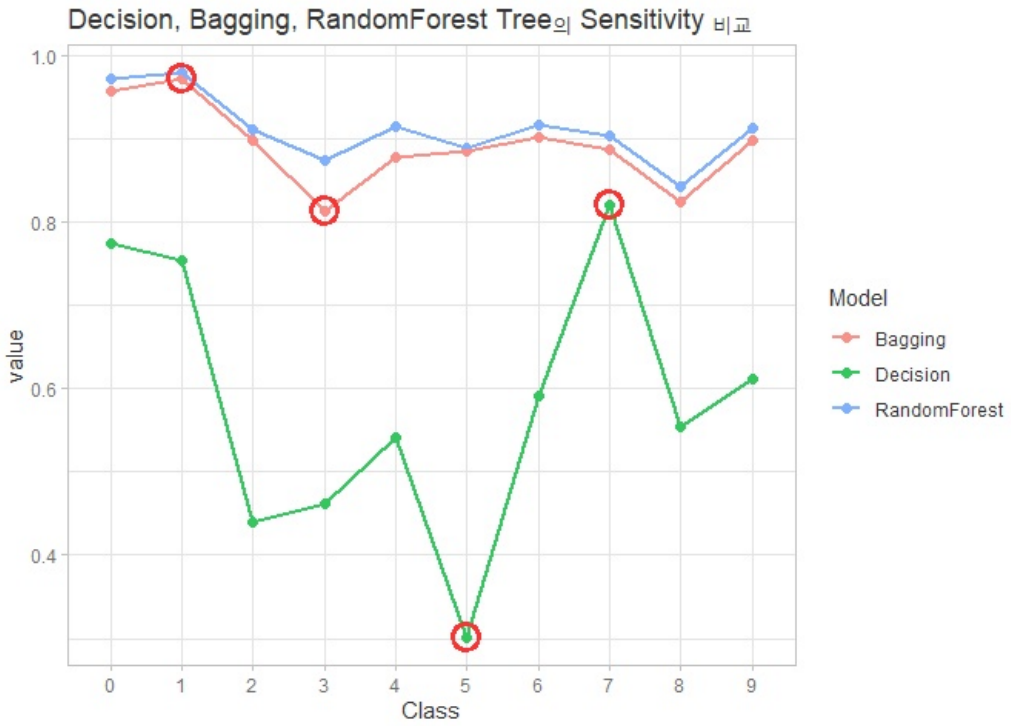
3 Model 비교

```
# 각 모델의 Sensitivity값 저장
df_result = data.frame(Class=0:9,
                       RandomForest = unname(cm_rand$byClass[, 1]),
                       Bagging = unname(cm_bag$byClass[, 1]),
                       Decision = unname(cm$byClass[, 1]))
df_gather = gather(df_result, Model, value, RandomForest, Bagging, Decision)

max_class_bag = df_result$Class[which.max(df_result$Bagging)]
max_val_bag = max(df_result$Bagging)
min_class_bag = df_result$Class[which.min(df_result$Bagging)]
min_val_bag = min(df_result$Bagging)

max_class_de = df_result$Class[which.max(df_result$Decision)]
max_val_de = max(df_result$Decision)
min_class_de = df_result$Class[which.min(df_result$Decision)]
min_val_de = min(df_result$Decision)

ggplot(df_gather, aes(x=Class, y=value, color=Model)) +
  geom_point(size=2) +
  geom_line(size=1)+
  theme_light() +
  scale_x_discrete(limits = c(0:9)) +
  labs(title='Decision, Bagging, RandomForest Tree의 Sensitivity 비교') +
  geom_point(aes(x = max_class_bag, y = max_val_bag, shape = as.factor(max_class_bag)),
             size = 5, color = "red", shape=1, stroke=2) +
  geom_point(aes(x = min_class_bag, y = min_val_bag, shape = as.factor(min_class_bag)),
             size = 5, color = "red", shape=1, stroke=2) +
  geom_point(aes(x = max_class_de, y = max_val_de, shape = as.factor(max_class_de)),
             size = 5, color = "red", shape=1, stroke=2) +
  geom_point(aes(x = min_class_de, y = min_val_de, shape = as.factor(min_class_de)),
             size = 5, color = "red", shape=1, stroke=2)
```



추가적으로 앞서 구했던 decision 모델과 bagging 모델의 분류 성능을 살펴보았다. 먼저 bagging 모델과 같은 경우 random forest와 동일하게 class 1에서 분류를 가장 정확하게 했지만 class 3에서 분류 성능이 가장 떨어졌다. decision 모델의 경우 random forest와 달리 class 7에서 가장 정확하게 분류하며 class 5에서 분류 성능이 떨어진다는 것을 알 수 있었다. 따라서 모델에 따라서 각 class에 따른 상대적 분류 성능이 다르다는 것을 알 수 있었다.

4.(F)

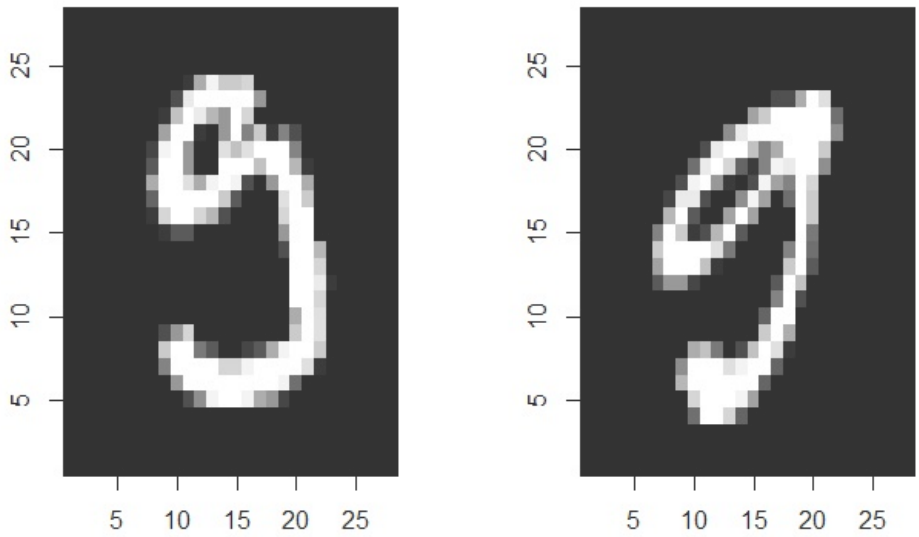
실제 값은 9지만 Random forest model에 의해 0으로 예측되는 test data를 찾아 이미지를 몇 개 출력해보자. 눈으로 확인했을 때 9와 0의 구별이 어려운가?

```
# 실제 값은 9이지만 0으로 예측되는 인덱스 추출
lst_sub = subset(test, test$y==9 & pred_rand==0)
row.names(lst_sub)
```

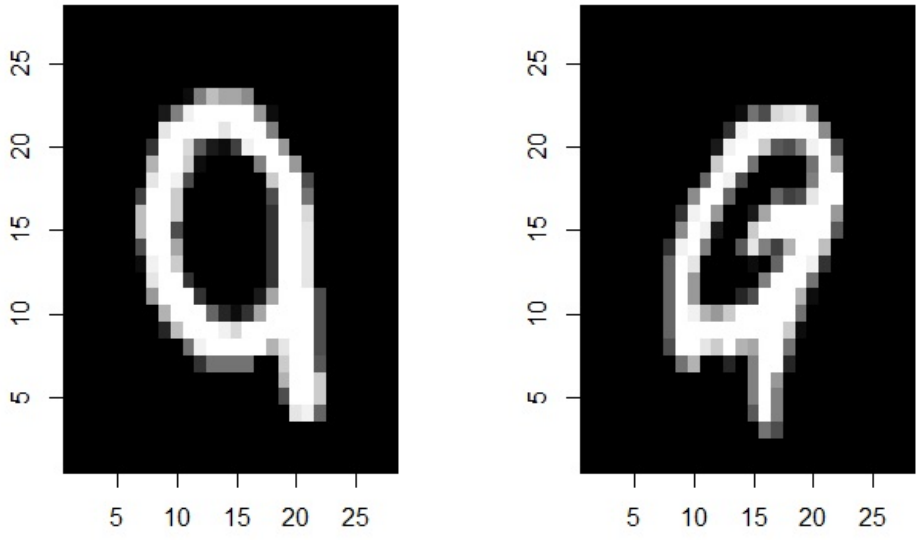
```
## [1] "1248" "2381" "2649" "3724" "4164" "4875" "6506"
```

총 7개의 오분류 이미지를 찾을 수 있었다. 다음으로 인덱스를 이용해 실제 그림을 살펴보도록 한다.

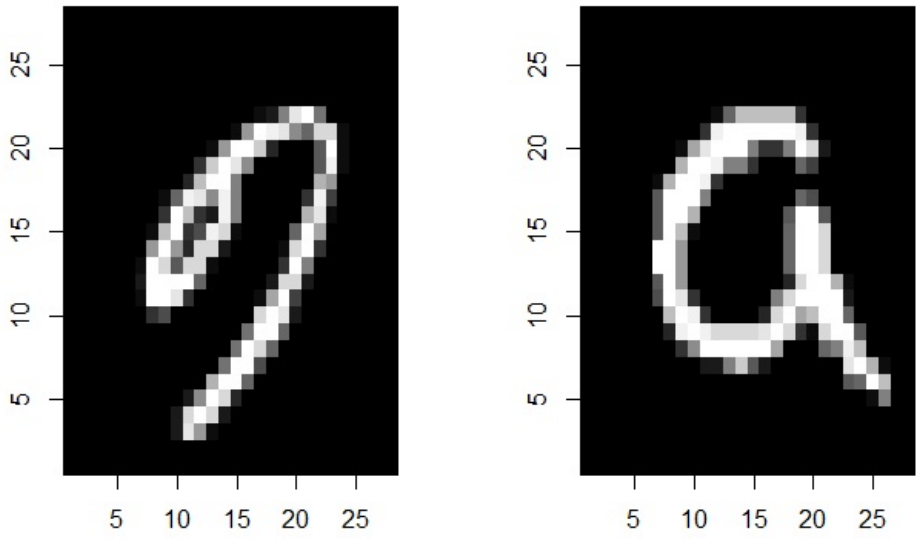
```
par(mfrow = c(1,2))
print_image(1248)
print_image(2649)
```



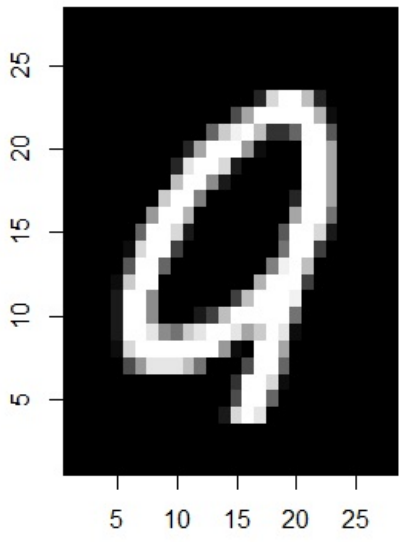

```
par(mfrow = c(1,2))
print_image(2381)
print_image(3724)
```




```
par(mfrow = c(1,2))
print_image(4164)
print_image(4875)
```




```
par(mfrow = c(1,2))
print_image(6506)
```



먼저 처음 2개의 이미지는 눈으로 확인했을때 9임을 쉽게 알 수 있다. 하지만 나머지 이미지의 경우, 주의 깊게 보지 않는다면 9보다는 0, 또는 다른 기호라고 판단할수도 있을 것이라 생각된다.