Multivariate Data Analysis

Assignment #6

고려대학교 공과대학

산업경영공학부

2017170857 이우준

[Q0] 사전작업

0.1.1 종속 변수 확인 및 필요 없는 변수 제거

```
# Load data & Preprocessing
data <- read.csv("Earthquake_Damage.csv")
dim(data)
Nrow <- dim(data)[1]
Ncol <- dim(data)[2]
#non_numeric index
factor_col = c()
for (i in c(1:Ncol)){
  if (is.numeric(data[,i])==FALSE){
    factor_col = c(factor_col, i)
  }
}
factor_col
input_data <- data[,-40]
scal <- input_data[,-factor_col]
data_scaling <- scale(scal, center = TRUE, scale = TRUE)
data_factor <- data[,factor_col]
data_target <- data[,40]
```

확인주어진 데이터 셋의 40개의 변수와 260601개의 instance로 이루어져 있다. 이 중 damagae_grade 변수가 종속변수이며, (1,2,3) 단계로 나누어져있는 categorical 변수이다. 나머지 39개의 변수 중 1번 변수인 building_id는 instance를 구분 하기위한 id로 통계적 의미가 없는 변수로 최종 데이터셋에서는 제거해준다.

0.1.2 Numeric이 아닌 변수

기본 입력변수 39 개 종류들을 확인해보았다. 우선 다른 변수들과 달리 int 형 변수가 아닌 chr 형 변수로 이루어진 (9~15, 27 번)변수는 string 형 categorical 변수이다.

Colum_num	Var	Colum_num	Var
9	land_surface_condition	13	other_floor_type
10	foundation_type	14	position
11	roof_type	15	plan_configuration
12	ground_floor_type	27	legal_ownership_status

이 들은 아래 단계에서 dummy variable 을 이용하여 one hot encoding 을 시행한다.

```
#one_hot_encoding
for (i in c(1:ncol(data_factor))){
  for(x in unique(data_factor[,i])){
    data_factor[paste(colnames(data_factor[i]),x, sep = "_")] <- ifelse(data_factor[,i] == x, 1,
0)
  }
}
```

One hot encoding을 통해 만들어진 dummy variable들의 이름은 원래의 variable 명 + _ + factor 명 의 규칙으로 구현하였다. 아래는 8개의 변수 중 대표로 land_surface_condition 변수의 one hot encoding 후 생성된 dummy variable의 예시이다.

land_surface_condition	n	o	t
land_surface_condition_n	1	0	0
land_surface_condition_o	0	1	0
land_surface_condition_t	0	0	1

```
#final_data
data_normal <- data.frame(data_factor,data_scaling, Class = data_target)
data_normal <- data_normal[,-c(1:8)]
```

0.2.1최종 데이터셋 제작

```
#train, validation, test set split
set.seed(123456)
tav_idx <- sample(1:Nrow, size=200000)
data_tav <- data_normal[tav_idx,]
data_tst <- data_normal[-tav_idx,]
val_idx <- sample(tav_idx,size=50000)
data_val <- data_normal[val_idx,]
data_trn <- data_normal[tav_idx[!(tav_idx%in%val_idx)],]
trn_target <- data_trn[,69]
trn_input <- data_trn[,-69]
val_target <- data_val[,69]
val_input <- data_val[,-69]
tav_target <- data_tav[,69]
tav_input <- data_tav[,-69]
tst_target <- data_tst[,69]
tst_input <- data_tst[,-69]
#target one hot enconding ver
class_target <- class.ind(data_normal[,69])
data_class <- data.frame(data_normal[,-69], Class = class_target)
data_class_tav <- data_class[tav_idx,]
data_class_tst <- data_class[-tav_idx,]
data_class_val <- data_class[val_idx,]
data_class_trn <- data_class[tav_idx[!(tav_idx%in%val_idx)],]
# target/input 분할
trn_target_class <- data_class_trn[,c(69,70,71)]
trn_input_class <- data_class_trn[,-c(69,70,71)]
val_target_class <- data_class_val[,c(69,70,71)]
val_input_class <- data_class_val[,-c(69,70,71)]
tav_target_class <- data_class_tav[,c(69,70,71)]
tav_input_class <- data_class_tav[,-c(69,70,71)]
tst_target_class <- data_class_tst[,c(69,70,71)]
tst_input_class <- data_class_tst[,-c(69,70,71)]
```

아래 데이터 분석을 위해 데이터를 Training Set, Validation Set, Test Set으로 분리해주었다. 또 Trainig이 끝난 이후 선택된 최적의 모델을 학습시키기 위한 Validation Set 과 Training Set이 합쳐진 TAV 셋도한 제작해주었다. 전체 데이터는 크게 Target 데이터가 one hot encoding을 한 버전과 안 한 버전 둘로 나누어 필요에 따라 사용하였다. 통

[Q1] 단일모형 성능비교

1. 1 Multionomial Logistic Regression

```
#Multinomial logistic regression
start <- proc.time()
mlr <- multinom(Class ~ ., data = data_tav)
proc.time() - start
mlr_prey <- predict(mlr, newdata = data_tst)
mlr_cfmatrix <- table(data_tst$Class, mlr_prey)
mlr_cfmatrix
perf_summary[1,] <- perf_eval_multi(mlr_cfmatrix)
perf_summary
```

위에서 정리한 데이터를 바탕으로 multinom 함수를 이용하여 multinomial logistic regression 을 시행하였다. Perf_eval_multi 함수를 이용하여 해당 모델의 설명력을 ACC, BCR 를 바탕으로 측정하였다. 분석을 통해 나온 confusion matrix 와 성능은 아래와 같다.

	1	2	3
1	1252	4346	289
2	865	29228	4364
3	88	15133	5036

Confusion Matrix를 통해 해당 모델이 2번 class를 구분하는데 성능은 좋지만 다른 class에 대해선 그렇지 않다는 것을 알 수 있다. 아래는 perf_eval_multi 함수를 통한 성능 분석 결과 이다.

Model	ACC	BCR
Multinomial Logistic Regression	0.5860629	0.3552882

ACC는 0.586으로 어느정도 분석력을 가지고 있다 할 수 있지만 BCR은 0.333이하로 성능이 좋지 않다고 할 수 있다.다른모델과 의 성능비교는 아래의 세개의 분석결과를 토대로 성능을 비교하겠다.

1. 2 Artificial neural network

```
#ANN
#hyperparameter 후보군 중 최적 조합 찾기
nH <- seq(7,11,2)
max_iter <- seq(100,200,50)
rang <- seq(0.1, 0.7, 0.3)
mat_ANN <- matrix(0,length(nH)*length(max_iter)*length(rang),5)
colnames(mat_ANN) <- c("Hidden Nodes", "Max Iteration","rang", "ACC", "BCR")
mat_ANN
start <- proc.time()
n<-1
for (i in 1:length(nH)) {
  cat("Training ANN: the number of hidden nodes:", nH[i], "\n")
  for (j in 1:length(max_iter)) {
    cat("Training ANN: the number of max iteration: ", max_iter[j], "\n")
    for (k in 1:length(rang)){
      tmp_nnet <- nnet(trn_input_class,trn_target_class, size = nH[i], decay = 5e-4, maxit =
max_iter[j], rang=rang[k])
      prey <- predict(tmp_nnet, val_input_class)
      mat_ANN[n,1:3] <- c(nH[i],max_iter[j],rang[k])
      mat_ANN[n,4:5] <- perf_eval_multi2(max.col(val_target_class), max.col(prey))
      n <- n+1
    }
  }
}
proc.time() - start
#ACC 기준의 성능 평가
ordered_ACC <- mat_ANN[order(mat_ANN[,4], decreasing = TRUE),]
colnames(ordered_ACC) <- c("Hidden Nodes", "Max Iteration","rang", "ACC", "BCR")
ordered_ACC[c(1:5),]
#BCR 기준의 성능 평가
ordered_BCR <- mat_ANN[order(mat_ANN[,5], decreasing = TRUE),]
colnames(ordered_BCR) <- c("Hidden Nodes", "Max Iteration","rang", "ACC", "BCR")
ordered_BCR[c(1:5),]
mat_ANN_final <- matrix(0,2,5)


colnames(mat_ANN_final) <- c("Hidden Nodes", "Max Iteration","rang", "ACC", "BCR")
```

ANN 모델 생성에 앞서 ANN 모델을 생성하는데 관여하는 hyperparameter 세가지를 조절하여 최적의 hyper parameter 조합을 찾았다. Hidden Node의 범위는 (7,9,11)로, Max Iteration의 범위는 (100,150,200), Rang의 범위는 (0.1,0.4,0.7)로 설정하여 총 27가지의 모델의 성능을 비교하였다.

```
> ordered_ACC[c(1:5),]
     Hidden Nodes Max Iteration rang     ACC       BCR
[1,]            9           150  0.7 0.94918 0.0000000
[2,]           11           100  0.7 0.91474 0.0000000
[3,]            9           100  0.4 0.85224 0.2773236
[4,]            7           200  0.1 0.55904 0.0000000
[5,]            7           150  0.7 0.55470 0.3665433
> ordered_BCR[c(1:5),]
     Hidden Nodes Max Iteration rang     ACC       BCR
[1,]            9           200  0.1 0.52594 0.4922679
[2,]            7           200  0.4 0.51320 0.4591558
[3,]            7           150  0.1 0.53462 0.4428870
[4,]            7           150  0.4 0.53140 0.4354077
[5,]           11           200  0.7 0.53434 0.4339676
```

27 가지의 모델을 ACC 와 BCR 기준으로 각각 최적의 hyperparameter 조합을 찾으면, ACC 기준 Hidden node 9 개 max iteration 150 개, rang 0.7 인것을 알 수 있고 BCR 기준 hidden nodes 9 개, Max iteration 200 개, rang 0.1 인것을 알 수 있다.

```
#ACC 기준
start <- proc.time()
nnet_ACC <- nnet(tav_input_class, tav_target_class, size = 9 , maxit =150, rang = 0.7, decay =
5e-4 )
prey_ACC <- predict(nnet_ACC, tst_input_class)
proc.time() - start
mat_ANN_final[1,1:3] <- c(9,150,0.7)
mat_ANN_final[1,4:5] <- perf_eval_multi2(max.col(tst_target_class), max.col(prey_ACC))
#BCR 기준
start <- proc.time()
nnet_BCR <- nnet(tav_input_class, tav_target_class, size = 9 ,maxit =200, rang = 0.1, decay = 5e-
4)
prey_BCR <- predict(nnet_BCR, tst_input_class)
proc.time() - start
mat_ANN_final[2,1:3] <- c(9,200,0.1)
mat_ANN_final[2,4:5] <- perf_eval_multi2(max.col(tst_target_class), max.col(prey_BCR))
mat_ANN_final
```

```
> mat_ANN_final
     Hidden Nodes Max Iteration rang       ACC       BCR
[1,]            9           150  0.7 0.5767562 0.0000000
[2,]            9           200  0.1 0.5451065 0.4387583
```

```
perf_summary[2,]<-mat_ANN_final[2,4:5]
perf_summary
```

두가지의 기준으로 성능을 평가한 결과 BCR기준으로 선정한 hyperparameter에서 종합적으로 더 우수한 성적이 나왔으므로 BCR 기준으로 생성한 모델의 성능 평가 결과를 ANN 모델의 성능 평가로 설정하였다.

Model	ACC	BCR
ANN	0.5451065	0.4387583

1. 3 CART

```
#CART
CART.trn = data.frame(trn_input, Class = as.factor(trn_target))
CART.val = data.frame(val_input, Class = as.factor(val_target))
CART.tav = data.frame(tav_input, Class = as.factor(tav_target))
CART.tst = data.frame(tst_input, Class = as.factor(tst_target))
minsplit <- seq(5,35,10)
maxdepth <- seq(10,50,20)
cp <- seq(0.01,0.03,0.01)
mat_CART = matrix(0,length(minsplit)*length(maxdepth)*length(cp),5)
colnames(mat_CART) <- c("minsplit", "maxdepth", "cp", "ACC", "BCR")
mat_CART
ctrl = rpart.control
start <- proc.time()
n<-1
for (i in 1:length(minsplit)) {
  cat("Training CART: the number of minsplit:", minsplit[i], "\n")
  for (j in 1:length(maxdepth)) {
    cat("Training CART: the number of maxdepth: ", maxdepth[j], "\n")
    for (k in 1:length(cp)){
      ctrl <- rpart.control(minsplit[i],maxdepth[j],cp[k])
      tmp_CART <- rpart(Class ~ ., CART.trn, control = ctrl)
      prey <- predict(tmp_CART, CART.val)
      mat_CART[n,1:3] <- c(minsplit[i],maxdepth[j],cp[k])
      tmp_cm <- table(CART.val$Class,round(prey))
      mat_CART[n,4:5] <- perf_eval_multi2(CART.val$Class,round(prey))
      n <- n+1
    }
  }
}
CART_ACC <- mat_CART[order(mat_CART[,4], decreasing = TRUE),]
CART_ACC[,]
ctrl1 <- rpart.control(minsplit=5, maxdepth=10, cp=0.01)
CART_best <- rpart(Class ~ ., CART.tav, control=ctrl1)
prey_CART <- predict(CART_best, newdata= CART.tst)
cm_CART <- table(CART.tst$Class, round(prey_CART))
perf_summary[3,]<-perf_eval_multi2(CART.tst$Class, round(prey_CART))
perf_summary
```

총 21 가지의 조합의 Accuracy(ACC)와 Balancec Correction Rate(BCR)를 기록한 q3_mat 을 ACC, BCR 기준으로 정렬해주었다.CART 모델 또한 총 36 가지의 hyperparameter 의 조합의 모델을 생성하여 성능을 비교 한후 최적의 hyper parameter 조합을 구하였다. 각 hyperparameter 의 범위는 다음과 같다. Min_split (5,15,25,35), max_depth (10,30,50), cp (0.01,0.02,0.03) 을 바탕으로 성능평가를 해보았다.

```
> CART_ACC[,]
      minsplit maxdepth   cp     ACC BCR
 [1,]        5       10 0.01 0.63850   0
 [2,]        5       30 0.01 0.63850   0
 [3,]        5       50 0.01 0.63850   0
 [4,]       15       10 0.01 0.63850   0
 [5,]       15       30 0.01 0.63850   0
```

성능 평가의 결과 모든 조합의 똑같이 나와 어느 값이 최적의 hyperparamter 의 조합이라 얘기하기 어려웠다. 시간 관계상 첫번째로 선정된 min split 5 maxdepth 10, cp 0.01.을 최적의 hyperparameter 조합으로 가정하여 최종 모델의 성능을 정리하였다.

1. 4 최종 모델비교

Model	ACC	BCR
Multinomial Logistic Regression	0.5860629	0.3552882
ANN	0.5451065	0.4387583
CART	0.6439168	0

3 가지의 단일 모델들의 성능을 비교해 보면 ACC 기준으로 보았을때는 CART 의 성능이 가장 우수하지만 이 모델은 Actual 1,2,3, 중에서 Predict 1 로 분류된 경우가 없어 BCR 이 0 이 나온다. 고로 복합적인 성능 면에서는 좋다고 할 수 없다. 세가지 모델중 ACC 와 BCR 이 복합적으로 높은 ANN 모델이 가장 나은 모델이라고 할 수 있는데 이또한 BCR 의 값이 0.333 이하로 랜덤으로 맞추었을 때보다 성능이 떨어져 설명력이 높은 모델이라고는 할 수 없다.

[Q2] CART Bagging

```
#CART BAGGING
mat_CART.bagging = matrix(0,4,3)
colnames(mat_CART.bagging) <- c("No. of Bootstrap", "ACC", "BCR")
mat_CART.bagging
mf <- seq(30,300,90)
```

Bagging 을 하 기 앞서 boot strap 의 수를 조절하는 부분을 일부분 수정하였다. 문제에서는 30~300 까지 30 단위씩 조정하여 10 가지의 경우를 비교해보아라 하엿지만, 컴퓨팅 능력의 한계와 시간 제한으로 bootstrap 간 간격을 90 으로 조절하여 30, 120, 210, 300 총 4 가지의 경우를 확인하겠다.

```
ctrl = rpart.control
start <- proc.time()
n<-1
for (i in 1:4) {
  cat("CART Bagging Bootstrap : ", mf[i], "\n")
  ctrl <- rpart.control(minsplit = 5, maxdepth = 10, cp = 0.01)
  CART.bagging <- bagging(Class ~ ., CART.trn, control = ctrl, nbagg = mf[i], coob=TRUE)
  prey <- predict(CART.bagging, CART.val)
  mat_CART.bagging[n,1] <- mf[i]
  tmp_cm <- table(CART.val$Class,round(prey))
  print(tmp_cm)
  mat_CART.bagging[n,2:3] <- perf_eval_multi2(CART.val$Class,round(prey))
  print(mat_CART.bagging)
  n <- n+1
}
mat_CART.bagging
Time <- proc.time() - start
Time
```

학습결과는 아래의 표와 같다.

Bootstrap	ACC	BCR
30	0.64492	0.4074415
120	0.64492	0.4074415
210	0.64492	0.4074415
300	0.64492	0.4074415

해당 결과를 통해 bootstrap 의 수와 관계없이 항상 같은 성능이 나오는 것을 알 수 있다. 고로 최적의 bootstrap 수는 컴퓨팅시간이 가장 짧은 30 으로 설정하여 최적의 모델을 알아보겠다.

```
#최적의 값 test
ctrl <- rpart.control(minsplit = 5, maxdepth = 10, cp = 0.01)
CART.bagging <- bagging(Class ~ ., CART.tav, control = ctrl, nbagg = 30, coob=TRUE)
prey <- predict(CART.bagging, CART.tst, type = "class")
tmp_cm <- table(CART.tst$Class,round(prey))
print(tmp_cm)
perf_eval_multi2(CART.tst$Class,round(prey))
```

Model	ACC	BCR
CART Bagging	0.1308394	0

최적의 Bootstrap 수와 , hyperparameter 의 조합으로 만든 CART Bagging 의 모델의 성능은 매우 안 좋은 것으로 나타났다. ACC, BCR 모두 0333 보다 작아 Random guessing 보다 성능이 안 좋게 나옴을 알 수 있다. 특히나 1 번 class 로 전혀 예측하지 않아 BCR 의 값은 0 이 나오는 걸 보아 해당 모델은 매우 성능이 나쁘다고 할 수 있다.

단일모델과 비교해보면

Model	ACC	BCR
CART Bagging	0.1308394	0
CART	0.6439168	0

두 모델 동일하게 Class 1 을 분류하지 않아 BCR 의 값이 0 이 나옴을 알 수 있다. 두 모델 둘다 Class 2 로 예측하는 경향이 강하다. 하지만 단일모형의 ACC 값이 Bagging 을 적용한 모델대비 약 5 배 정도 높은 것을 통해 단일모형의 성능이 Bagging 을 적용한 모델보다 우수하다고 할 수 있다.

이를 ACC 기준, BCR 기준으로 정리한 값의 가장 우월한 값을 도출해 보았다.

```
> q4_mat_ACC[1,]
   rang     ACC     BCR
0.40000 0.64478 0.50548
> q4_mat_BCR[1,]
   rang     ACC     BCR
0.40000 0.64478 0.50548
```

q4_mat_ACC나, q4_mat_BCR 의 가장 최적의 rang 값은 0.4로 동일하다는 것을 알 수 있다. 이를 통해 최적의 hyperparameter조합은 다음과 같다는 것을 알 수 있다.

Hidden Nodes	max_it	rang
9	100	0.4

[Q3] Random Forest

```
#Random Forest
RF.trn <- CART.trn
RF.tst <- CART.tst
RF.val <- CART.val
RF.tav <- CART.tav
ntr <- seq(30,300,30)
mat_RF = matrix(0,10,3)
colnames(mat_RF) <- c("No. of Tree", "ACC", "BCR")
mat_RF
ptm <- proc.time()
n<-1
for (i in 1:length(ntr)){
  RF.model <- randomForest(Class ~ ., data = RF.trn, ntree = ntr[i], importance = TRUE, do.trace
= TRUE)

  # Check the result
  print(RF.model)
  plot(RF.model)

  # Variable importance
  Var.imp <- importance(RF.model)
  barplot(Var.imp[order(Var.imp[,4], decreasing = TRUE),4])

  # Prediction
  RF.prey <- predict(RF.model, newdata = RF.val, type = "class")
  RF.cfm <- table(RF.prey, RF.val$Class)

  mat_RF[n,1] <- ntr[i]
  mat_RF[n,2:3] <- perf_eval_multi(RF.cfm)
  RF.cfm
  mat_RF
  n <- n+1
}
mat_RF
RF.Time <- proc.time() - ptm
RF.Time
```

Random Forest 의 tree 개수를 30 부터 300 까지 30 개씩 늘려가며 학습하였다. 반복문을 10 회 수행하여 정리된 성능 데이터를 mat_RF 에 저장하였다. 이는 아래에 나와있는 표와 같다

Number of tree	ACC	BCR
30	0.7055824	0.7028603
60	0.7052029	0.7052618
90	0.7099338	0.7099243
120	0.7089982	0.7081217
150	0.7107638	0.7104599
180	0.7095757	0.7089964
210	0.7097648	0.7089964
240	0.7094602	0.7098866
270	0.7108959	0.7118976
300	0.7121170	0.7123419

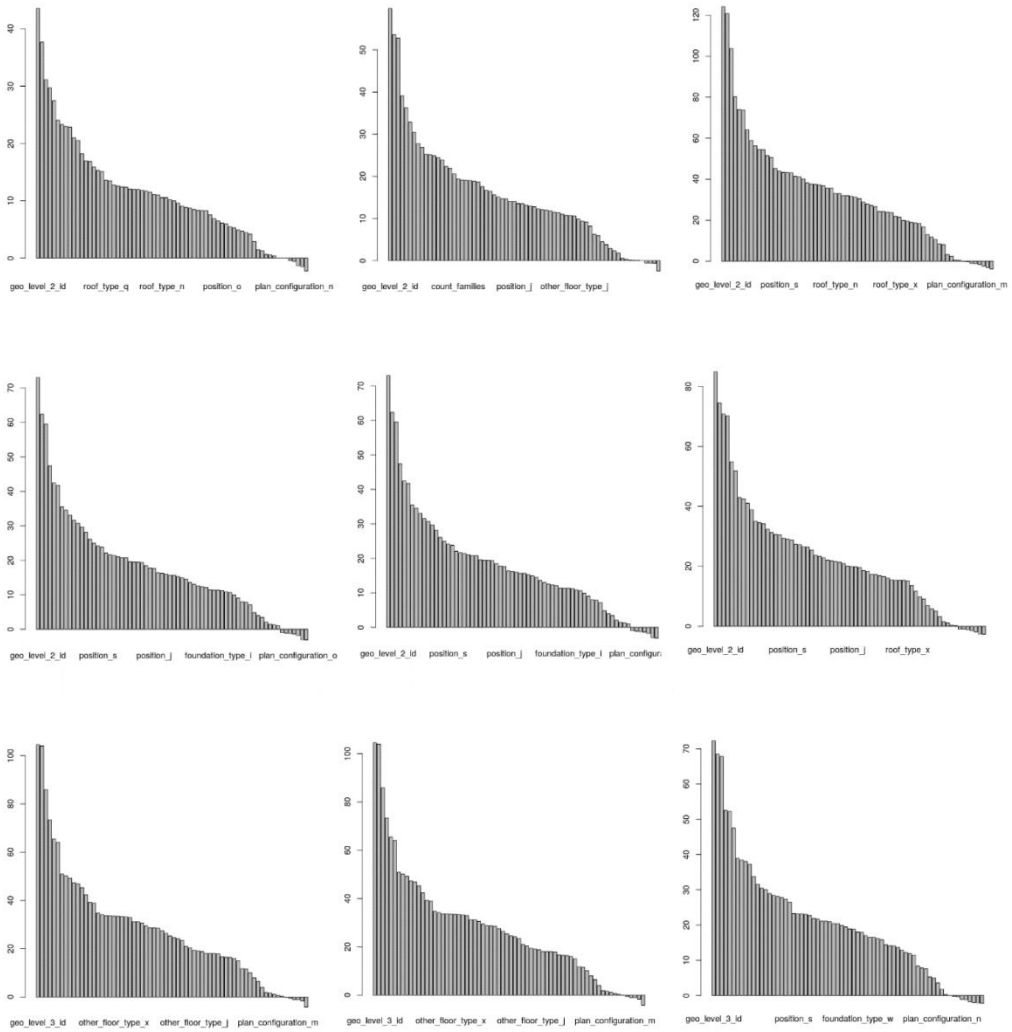
위 표를 통해 Tree 의 수가 늘어날수록 ACC,BCR 성능 모두 점차늘어나는 것을 알 수있다 고로 최적의 Bootstrap 수는 ntrees 가 300 개일때라는 것을 위 정보를 통해 알 수 있다. 구한 최적의 Bootstrap 수, TAV Set 을 이용 Test 셋으로 성능을 평가해보았다.

```
#베스트 모델의 평가
RF.model <- randomForest(Class ~ ., data = RF.tav, ntree = 300 , importance = TRUE, do.trace =
TRUE)
print(RF.model)
plot(RF.model)
Var.imp <- importance(RF.model)
barplot(Var.imp[order(Var.imp[,4], decreasing = TRUE),4])
head(Var.imp[order(Var.imp[,4], decreasing = TRUE),4])
RF.prey <- predict(RF.model, newdata = RF.tst, type = "class")
RF.cfm <- table(RF.prey, RF.tst$Class)
perf_eval_multi2(RF.prey, RF.tst$Class)
perf_eval_multi(RF.cfm)
RF.cfm
```

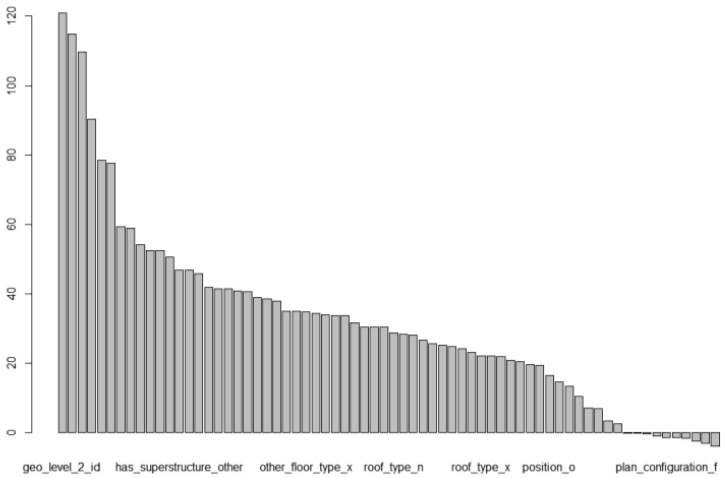
Bootstrap 이 300 일 때 변수 중요도를 출력해 보았다.

```
land_surface_condition_t                32.9551785
land_surface_condition_o                17.4566235
land_surface_condition_n                25.1820608
foundation_type_r                       24.0171766
foundation_type_w                       13.5062522
foundation_type_i                       26.4334921
foundation_type_u                       -2.6620755
foundation_type_h                       24.4002081
roof_type_n                             21.3332044
roof_type_q                             21.5693428
roof_type_x                             27.8830107
ground_floor_type_f                     15.2981955
ground_floor_type_x                     25.3640195
ground_floor_type_v                     21.5001337
ground_floor_type_z                     12.8409304
ground_floor_type_m                      3.6356900
other_floor_type_q                      21.7495466
other_floor_type_x                      24.5537166
other_floor_type_j                      16.1117745
other_floor_type_s                      11.1626710
position_t                              21.0373195
position_s                              24.0972471
position_j                               7.4685801
position_o                               4.0866627
plan_configuration_d                     8.3808722
plan_configuration_u                     5.7618582
plan_configuration_s                    -0.8240514
plan_configuration_q                     5.0115379
plan_configuration_m                    -0.5413808
plan_configuration_c                    -1.9324776
plan_configuration_a                     9.8794724
plan_configuration_n                     1.0016708
plan_configuration_f                     0.0000000
plan_configuration_o                     1.1254072
legal_ownership_status_v                19.4676090
legal_ownership_status_a                18.3511362
legal_ownership_status_r                 9.2762032
legal_ownership_status_w                 8.1428957
geo_level_1_id                          98.6271985
geo_level_2_id                          81.7735942
geo_level_3_id                          63.8536849
count_floors_pre_eq                     23.6684249
age                                     98.7658592
area_percentage                         33.6318793
height_percentage                       28.2622087
has_superstructure_adobe_mud            28.9095495
has_superstructure_mud_mortar_stone     38.2331270
has_superstructure_stone_flag           29.4170638
has_superstructure_cement_mortar_stone   7.4826615
has_superstructure_mud_mortar_brick     33.5642875
has_superstructure_cement_mortar_brick  12.4383515
has_superstructure_timber               41.4536756
has_superstructure_bamboo               25.1252402
has_superstructure_rc_non_engineered    14.4678729
has_superstructure_rc_engineered        32.7835221
has_superstructure_other                34.8746338
count_families                          47.4088018
has_secondary_use                        7.3080285
has_secondary_use_agriculture           18.5946931
has_secondary_use_hotel                 -2.3120182
has_secondary_use_rental                13.5669022
has_secondary_use_institution           -1.2566250
has_secondary_use_school                -5.0834870
has_secondary_use_industry              -2.4527955
has_secondary_use_health_post           -1.8964759
has_secondary_use_gov_office            -2.2196393
has_secondary_use_use_police            -2.4742422
has_secondary_use_other                 12.1654712
```

이를 보기 쉽게 그래프 모양으로도 plot 해보았다.



해당 그래프와 수치데이터를 통해 geolevel2_id, geolevel3_id 변수가 중요하다는 것을 확인 할 수 있다. 이를 Head 함수를 통해 보다 자세히 나타내면




```
> head(Var.imp[order(Var.imp[,4], decreasing = TRUE),4])
   geo_level_2_id    geo_level_3_id    geo_level_1_id   area_percentage               age height_percentage
        120.88477         114.99350         109.79044          90.34486          78.48908          77.69182
```

주요 변수는 geo_level_2_id, geo_level3_id, geo_level_1_id, height_percentage 순으로 중요도가 높다는 것을 알 수 있고 해당 변수들이 종속변수에 유의미한 영향을 준다고 볼 수 있다.

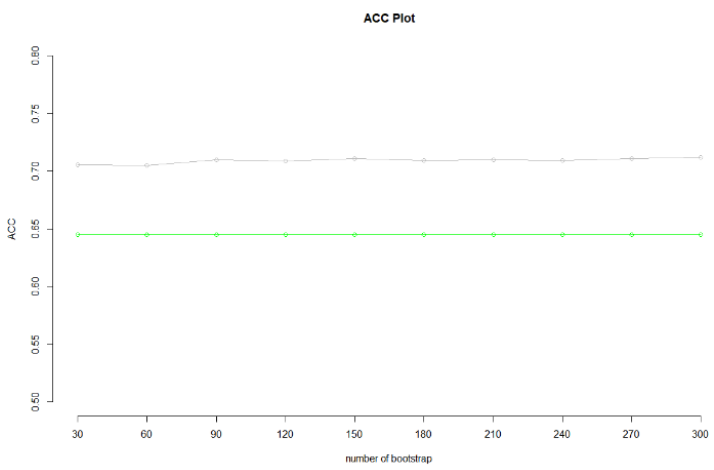
선택된 Baggging 의 수와, hyperparameter 의 조합으로 뽑힌 최적의 조합을 TAV set 을 이용하여 학습시키고 Test Set 으로 평가한 결과는 아래와 같다.

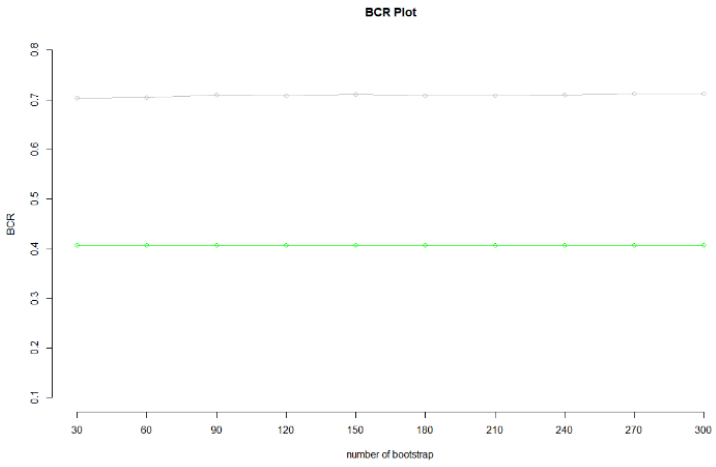
Model	ACC	BCR
Random Forest	0.7098563	0.7100259

단순 정확도 ACC 도 0.709 로 성능이 좋고 균형정확도 BCR 도 0.71 로 성능이 매우 좋다고 할수 있다.

```
## CART Bagging and Random Forest Performance Measure Plot
 x <- seq(30,300,30)
bag_acc <- rep(0.64492,10)
bag_BCR <- rep(0.4074415,10)
RF_acc <- c(0.7055824, 0.7052029, 0.7099338, 0.7089982, 0.7107638, 0.7095757, 0.7097648,
0.7094602, 0.7108959, 0.7121170)
RF_BCR <- c(0.7028603, 0.7052618, 0.7099243, 0.7081217, 0.7104599, 0.7089964, 0.7089964,
0.7098866, 0.7118976, 0.7123419)
plot(bag_acc,axes=F, main = "ACC Plot",xlab="number of bootstrap",ylab="ACC",type='o',
col='green', ylim = c(0.5,0.8))
axis(1, at = 1:10, lab=x)
axis(2, ylim=c(0.4,0.8))
lines(RF_acc, type='o', col='gray')
plot(bag_BCR,axes=F, main = "BCR Plot", xlab="number of bootstrap",ylab="BCR",type='o',
col='green', ylim = c(0.1,0.8))
axis(1, at = 1:10, lab=x)
axis(2, ylim=c(0.3,0.6))
lines(RF_BCR, type='o', col='gray')
```

CART Bagging 모형과 Random Forest 분류정확도를 Plot 해 보면 아래 표와 같은 결과를 알 수 있다.





위 그래프는 Bootstrap수에 따라 달라지는 ACC, BCR 성능지표를 나타낸 것이다. Random Forest의 Performance measure는 회색의 선, CART Bagging Performance measure는 연두색의 그래프로 나타내었다. ACC와 BCR 평가지표 모두 Random Forest가 우수함을 볼 수 있다.

[Q4] ANN 30회 반복 수행

Q2에서 구한 최적의 hyperparameter 조합, Hidden nodes 9, Max iteration 200, rang 0.1을 바탕으로 ANN 단일모형 제작을 30회 반복하였다.

```
mat_ANN_final <- matrix(0,30,3)
colnames(mat_ANN_final) <- c("No.", "ACC", "BCR")
start <- proc.time()
n <- 1
for (i in 1:30){
  final_nnet <- nnet(tav_input_class,tav_target_class, size = 9 , rang = 0.1, decay = 5e-4, maxit
=200 )
  final_prey <- predict(final_nnet, tst_input_class)
  mat_ANN_final[n,1] <- i
  mat_ANN_final[n,2:3] <- perf_eval_multi2(tst_target_class, max.col(final_prey))
  table(tst_target_class,max.col(final_prey))
  mat_ANN_final
  n <- n+1
}
proc.time() - start
mat_ANN_final
mean(mat_ANN_final[,2])
var(mat_ANN_final[,2])
mean(mat_ANN_final[,3])
var(mat_ANN_final[,3])
```

반복 수행의 결과는 아래의 표와 같다.

No.	ACC	BCR	No.	ACC	BCR
1	0.643733	0.0	16	0.507857	0.466381
2	0.632547	0.486123	17	0.619363	0.0
3	0.557376	0.472945	18	0.639562	0.0
4	0.651373	0.501234	19	0.610663	0.439721
5	0.638759	0.0	20	0.572552	0.403826
6	0.628453	0.510562	21	0.629575	0.0
7	0.613822	0.409865	22	0.596314	0.475628
8	0.535726	0.428564	23	0.578236	0.0
9	0.618146	0.449712	24	0.5924631	0.475629
10	0.534817	0.467694	25	0.617562	0.497161
11	0.658245	0.0	26	0.583461	0.0
12	0.593478	0.507927	27	0.609237	0.412847
13	0.626970	0.430190	28	0.579613	0.423347
14	0.645375	0.0	29	0.646128	0.0
15	0.617584	0.0	30	0.624480	0.506601

	ACC	BCR
MEAN	0.585324	0.292199
STdev	0.037271	0.224057

최적의 ANN 모델을 30번 반복 수행하여 구한 성능의 결과는 위의 표와 같다. ACC의 평균은 0.585, 분산은 0.03이다. BCR의 평균은 0.292이고 분산은 0.224이다. 단순정확도가 균형정확도 보다 수치도 높을 뿐만 아니라

분산도 적게 나오는 것을 알 수 있다. B단순정확도 기준으로 보았을때는 모델의 정확도가 약 60퍼센트로 상대적으로 우수해 보이지만, 균형정확도를 보았을때는 전혀 그렇지 않다.

또한 이를 통해 ACC가 0.6보다 클태는 bCR이 0.0인 것을 알 수 가있다. 이는 해당 모델이 Actual 1,2,3, 이 Predicted 1로 분류되는 경우가 전혀 없었기 때문이다.

[Q5] ANN Bagging

ANN Bagging 을 진행하기전 Computing 파워의 한계로 전체 데이터셋을 1/100로 샘플링하여 ANN Bagging을 진행하였다. 아래는 그 sampling 코드 이다.

```
set.seed(111)
trn_idx_red <- sample(1:nrow(data_trn), size=1500)
val_idx_red <- sample(1:nrow(data_val), size=500)
tst_idx_red <- sample(1:nrow(data_tst), size=606)
data_trn_red <- data_trn[trn_idx_red,]
data_val_red <- data_val[val_idx_red,]
data_tst_red <- data_tst[tst_idx_red,]
data_tav_red <- rbind(data_trn_red,data_val_red)
trn_target_red <- data_trn_red[,69]
trn_input_red <- data_trn_red[,-69]
val_target_red <- data_val_red[,69]
val_input_red <- data_val_red[,-69]
tav_target_red <- data_tav_red[,69]
tav_input_red <- data_tav_red[,-69]
tst_target_red <- data_tst_red[,69]
tst_input_red <- data_tst_red[,-69]

CART.trn.red = data.frame(trn_input_red, Class = as.factor(trn_target_red))
CART.val.red = data.frame(val_input_red, Class = as.factor(val_target_red))
CART.tav.red = data.frame(tav_input_red, Class = as.factor(tav_target_red))
CART.tst.red = data.frame(tst_input_red, Class = as.factor(tst_target_red))
```

축소된 데이터를 바탕으로 ANN Bagging을 30~300 Bootstrap까지 30씩 증가시켜 총 10번 시행하였다.

```
mat_ANN.Bagging= matrix(0,30,3)
colnames(mat_ANN.Bagging) <- c("Iter. No.", "ACC", "BCR")
mat_ANN.Bagging
boots <- seq(30,300,30)
ptm <- proc.time()
n<-1
for (i in (1:30)){
  cat("ANN Bagging iteration : ",i, "\n")
  Bagging.ANN.model <- avNNet(trn_input_red, as.factor(trn_target_red), size = 9, decay = 5e-4,
maxit = 200,
                              repeats = boots[i], bag = TRUE, allowParallel = TRUE, trace = TRUE)
  Bagging.Time <- proc.time() - ptm
  Bagging.Time

  Bagging.ANN.prey <- predict(Bagging.ANN.model, newdata = val_input_red)
  Bagging.ANN.cfm <- table(val_target_red, max.col(Bagging.ANN.prey))
  Bagging.ANN.cfm

  mat_ANN.Bagging[n,1] <- i
  mat_ANN.Bagging[n,2:3] <- perf_eval_multi2(val_target_red, max.col(Bagging.ANN.prey))
  n <- n+1

}
mat_ANN.Bagging
mean(mat_ANN.Bagging[,2])
var(mat_ANN.Bagging[,2])
mean(mat_ANN.Bagging[,3])
var(mat_ANN.Bagging[,3])
```

또한 시행중 메모리 용량의 한계로 Bootstrap 값을 repeats 옵션에 입력해준 이후 30회를 반복 수행하였다. 아래는 성능의 평균과 표준편차를 기록한 표이다..

Bootstrap	ACC_MEAN	ACC_SD	BCR_MEAN	BCR_SD
30	0.5776	0.0000371	0.260868	0.0004743
60	0.5663	0.0000283	0.261904	0.0004291
90	0.5849	0.0000209	0.259081	0.0004212
120	0.5895	0.0000229	0.255069	0.0004313
150	0.5763	0.0000238	0.265832	0.0004819
180	0.5918	0.0000159	0.259532	0.0004958
210	0.6028	0.0000209	0.249673	0.0000450
240	0.6148	0.0000339	0.275932	0.0004135
270	0.6055	0.0000272	0.285724	0.0004894
300	0.6031	0.0000284	0.265123	0.0004902

ANN Bagging을 30회 반복적으로 수행하였을 때 ACC의 평균은 대략적으로 0.6에 수렴하며 표 준편차는 매우 작음을 알 수있다. BCR의 평균은 ACC보다는 변동폭이 크지만 모두 0.3보다 작으며 표준편차가 모두 0 에 가깝다. 최적의 Bootstrap의 경우 270일때이지만, 다른 Bootstrap 간의 유의미한 차이는 보이지 않는다. 또한 Bootstrap의 수와 성능은 비례하지 않는 다는 것을 알 수 있다.

```
#Testing
best_Bagging.ANN.model <- avNNet(tav_input_red, as.factor(tav_target_red), size = 9, decay = 5e-
4, maxit = 200,
                                 repeats = 240, bag = TRUE, allowParallel = TRUE, trace = TRUE)
best_Bagging.ANN.prey <- predict(best_Bagging.ANN.model, newdata = tst_input_red)
Bagging.ANN.cfm <- table(max.col(tst_target_red), max.col(best_Bagging.ANN.prey))
Bagging.ANN.cfm
perf_eval_multi2(tst_target_red, max.col(best_Bagging.ANN.prey))
table(tst_target_red, max.col(best_Bagging.ANN.prey))
```

Model	ACC	BCR
ANN_Bagging	0.5891089	0.3094046

Test셋에대한 모델의 성능은 위와 같다. 이를 ANN 단일모형과 ANN 30회 반복 단일모형과의 성능을 비교해 보았다.

Model	ACC	BCR
ANN	0.5451065	0.4387583
ANN 30	0.585324	0.292199
ANN_Bagging	0.5891089	0.3094046

ANN Bagging 모델이 ANN 단일모형보다 ACC 측면에서는 약 4% 더 성능이 우수한 것을 확인 할 수 있다. 하지만 BCR 기준으로 보았을땐 ANN 단일모형의 모델이 더 나은 성능을 갖고 있다고 할 수 있다. 물론 ANN Bagging을 모델링하고 검증한 데이터셋이 ANN 단일모형을 제작한 데이터 셋에서 Sampling 한 데이터이기 때문 같은 기준선상에 두고 비교하는 게 아니여서 정확한 비교라고 할 수는 없다.

[Q6] AdaBoost

```
bag_ctrl <- rpart.control(minsplit=10, maxdepth=10, cp=0.01)
ada_10 <- boosting(Class~., CART.trn, boos=TRUE, mfinal=10, control=bag_ctrl)
ada_20 <- boosting(Class~., CART.trn, boos=TRUE, mfinal=20, control=bag_ctrl)
ada_30 <- boosting(Class~., CART.trn, boos=TRUE, mfinal=30, control=bag_ctrl)
ada_40 <- boosting(Class~., CART.trn, boos=TRUE, mfinal=40, control=bag_ctrl)
ada_50 <- boosting(Class~., CART.trn, boos=TRUE, mfinal=50, control=bag_ctrl)
pred_10 <- predict.boosting(ada_10, CART.val)
pred_20 <- predict.boosting(ada_20, CART.val)
pred_30 <- predict.boosting(ada_30, CART.val)
pred_40 <- predict.boosting(ada_40, CART.val)
pred_50 <- predict.boosting(ada_50, CART.val)
ada_mat <- matrix(0,5,3)
colnames(ada_mat) <- c("mfinal", "ACC", "BCR")
ada_mat[1,2:3] <- perf_eval_multi(t(pred_10$confusion))
ada_mat[2,2:3] <- perf_eval_multi(t(pred_20$confusion))
ada_mat[3,2:3] <- perf_eval_multi(t(pred_30$confusion))
ada_mat[4,2:3] <- perf_eval_multi(t(pred_40$confusion))
ada_mat[5,2:3] <- perf_eval_multi(t(pred_50$confusion))
ada_mat[1,1] <- 10
ada_mat[2,1] <- 20
ada_mat[3,1] <- 30
ada_mat[4,1] <- 40
ada_mat[5,1] <- 50
t(pred_10$confusion)
t(pred_20$confusion)
t(pred_30$confusion)
t(pred_40$confusion)
t(pred_50$confusion)
```

AdaBoost에 사용되는 Hyperparameter는 두가지가 있다. Boos 변수와, mfinal 변수이다. Boos 변수는 Boolean 변수로 True 라면 train set의 일부가 각 관측치의 가중치에 사용되는 것이고, mfinal 은 Iteratoin의 반복수 행횟수를 뜻한다. 이번 경우 Boos는 True일 경우에만 학습하고 mfinal은 10 20 30 40 50 총 5가지 상황에 대해서 분류정확도를 비교하였다. 해당 결과는 아래의 표와 같다.

boos	mfinal	ACC	BCR
TRUE	10	0.65482	0.4382326
TRUE	20	0.64384	0.3028075
TRUE	30	0.65186	0.3283501
TRUE	40	0.64390	0.3028189
TRUE	50	0.65334	0.4413700

이 또한 단순정확도 ACC는 약 0.65로 mfinal의 변화에 큰 영향이 없지만 BCR의 경우 0.3~0.44까지 변동폭이 ACC에 비하여 큰 폭으로 존재하였다. 해당 결과를 통해 최적의 조합은 Boos가 True이고 mfinal이 50일때인 것을 확인하였다. 이때 이에 최적의 조합에 대하여 test 셋에 대한 성능을 평가해 보았다.

```
#최적의 조합으로 test set 으로 성능 평가
ada_best <- boosting(Class~., CART.tav, boos=TRUE, mfinal=50, control=bag_ctrl)
pred_best <- predict.boosting(ada_best, CART.tst)
perf_eval_multi(t(pred_best$confusion))
```

Model	ACC	BCR
AdaBoost	0.6396099	0.2924276

해당 기법의 단순 정확도 ACC는 약 0.64로 성능이 나쁘지 않음을 확인할 수있었지만 BCR은 0.29로 설명력이 떨어진다고 판단할 수 있었다. 이 전에 학습한 모델들과의 성능을 비교하기 위하여 한 표에 정리해보았다.

Model	ACC	BCR
Multinomial Logistic Regression	0.5860629	0.3552882
ANN	0.5451065	0.4387583
CART	0.6439168	0
CART Bagging	0.1308394	0
ANN 30	0.585324	0.292199
ANN_Bagging	0.5891089	0.3094046
Random Forest	0.7098563	0.7100259
AdaBoost	0.6396099	0.2924276

지금껏 학습한 모델과 비교해보았을 때 단순정확도 ACC 는 다른 모델들에 비해 매우 높은 성능을 보인다고 할 수 잇다. 하지만 BCR의 경우 30퍼센트도 안되는 수준으로 성능이 우수하지 않다. 비슷한 ACC 수치를 가진 CART와 비교하면 CART와 ACC의 값은 거의 유사하지만 BCR 값이 더 우수하므로 AdaBoost의 성능이 CART 보다 더 낫다고 할 수 있다. 하지만 종합적으론 여전히 Random Forest 기법이 가장 성능이 좋은 것을 알 수 있다.

[Q7] Gradient Boosting

```
GBM.trn <- data.frame(trn_input, Class = trn_target)
GBM.val <- data.frame(val_input, Class = val_target)
GBM.tst <- data.frame(tst_input, Class = tst_target)
GBM.tav <- data.frame(tav_input, Class = tav_target)
# Training the GBM
nt <- seq(500,1000,250)
sr <- seq(0.03,0.09, 0.03)
bf <- seq(0.6, 0.8, 0.1)
gbm_mat <- matrix(0,length(nt)*length(sr)*length(bf),5)
colnames(gbm_mat) <- c("n.tress","shrinkage","big.fraction", "ACC", "BCR")
ptm <- proc.time()
n<-1
for (i in 1:length(nt)) {
  cat("Training gbm: the number of trees:", nt[i], "\n")
  for (j in 1:length(sr)) {
    cat("Training gbm: shrinkage is: ", sr[j], "\n")
    for (k in 1:length(bf)){
      GBM.model <- gbm.fit(GBM.trn[,1:68], GBM.trn[,69], distribution = "gaussian",
                           n.trees = nt[i], shrinkage = sr[j], bag.fraction = bf[k], nTrain = 500)
      GBM.prey <- predict(GBM.model, GBM.val[,1:68], type = "response")
      GBM.prey <- round(GBM.prey)
      GBM.cfm <- table(GBM.prey, GBM.val$Class)
      GBM.cfm
      perf_eval_multi(GBM.cfm)
      gbm_mat[n,1:3] <- c(nt[i],sr[j],bf[k])
      gbm_mat[n,4:5] <- perf_eval_multi(GBM.cfm)
      n <- n+1
    }
  }
}
gbm_mat
GBM.Time <- proc.time() - ptm
```

```
GBM.Time
gbm_ACC <- gbm_mat[order(gbm_mat[,4], decreasing = TRUE),]
colnames(gbm_ACC) <- c("n.tress","shrinkage","big.fraction", "ACC", "BCR")
gbm_ACC[1,]
```

gbm.fit 함수를 이용하여 Gradient Boosting Machine 을 구현하였다. 총 세가지의 hyperparameter 를 조절하여 최적의 hyperparameter 조합을 찾았다. 조정에 이용된 세 변수는 n.tress, shrinkage, big fraction 이다. n.trees 는 Fitting 에 사용되는 전체 tree 의 개수이고, shrinkage 는 각 tree 의 확장에 적용되는 비율을 나타낸다. 마지막으로 big fraction 은 Training Set 중에서 tree 확장에 사용되는 비율을 나타낸다. 각각의 hyperparmeter 의 범위는 아래와 같이 조정하였다.

n.trees: (500,750,1000)
shrinkage: (0.03, 0.06, 0.09)
big fraction: (0.6, 0.7, 0.8)

아래는 학습의 결과이다.

```
> gbm_mat
      n.tress shrinkage big.fraction     ACC       BCR
 [1,]     500      0.03          0.6 0.59402 0.5764395
 [2,]     500      0.03          0.7 0.59938 0.5840995
 [3,]     500      0.03          0.8 0.58758 0.5730033
 [4,]     500      0.06          0.6 0.60536 0.5838820
 [5,]     500      0.06          0.7 0.60690 0.5841267
 [6,]     500      0.06          0.8 0.60778 0.5833547
 [7,]     500      0.09          0.6 0.58696 0.5655419
 [8,]     500      0.09          0.7 0.60158 0.5763856
 [9,]     500      0.09          0.8 0.60640 0.5813278
[10,]     750      0.03          0.6 0.60528 0.5838676
[11,]     750      0.03          0.7 0.60472 0.5840673
[12,]     750      0.03          0.8 0.60230 0.5838585
[13,]     750      0.06          0.6 0.60050 0.5790918
[14,]     750      0.06          0.7 0.60426 0.5817026
[15,]     750      0.06          0.8 0.60836 0.5837862
[16,]     750      0.09          0.6 0.61176 0.5865104
[17,]     750      0.09          0.7 0.60784 0.5870879
[18,]     750      0.09          0.8 0.60190 0.5800056
[19,]    1000      0.03          0.6 0.60554 0.5821334
[20,]    1000      0.03          0.7 0.60206 0.5811677
[21,]    1000      0.03          0.8 0.60616 0.5849911
[22,]    1000      0.06          0.6 0.60542 0.5829663
[23,]    1000      0.06          0.7 0.60512 0.5811823
[24,]    1000      0.06          0.8 0.60476 0.5839131
[25,]    1000      0.09          0.6 0.60276 0.5817620
[26,]    1000      0.09          0.7 0.60632 0.5838657
[27,]    1000      0.09          0.8 0.60466 0.5800168
```

GBM 의 hyperparameter 의 조정 거의 모든 경우의 ACC 와 BCR 이 0.6 에 가깝게 나왔다. 그나마 가장 성능이 좋은 것을 고르면 아래의 조합과 같다.

n.trees	Shrinkage	Big fraction	ACC	BCR
750	0.09	0.6	0.6117600	0.5865104

이를 바탕으로 Test 셋에 대하여 성능을 평가해 보았다.

```
ptm <- proc.time()
GBM.best <- gbm.fit(GBM.tav[,1:68], GBM.tav[,69], distribution = "gaussian",
                    n.trees = 750, shrinkage = 0.09, bag.fraction = 0.6, nTrain = 500)
summary(GBM.best)
GBM.prey <- predict(GBM.best, GBM.tst[,1:68], type = "response")
GBM.prey <- round(GBM.prey)
GBM.cfm <- table(GBM.prey, GBM.tst$Class)
perf_eval_multi(GBM.cfm)
GBM.cfm
GBM.Time <- proc.time() - ptm
GBM.Time
```

Model	ACC	BCR
GBM	0.5739179	0.6027088

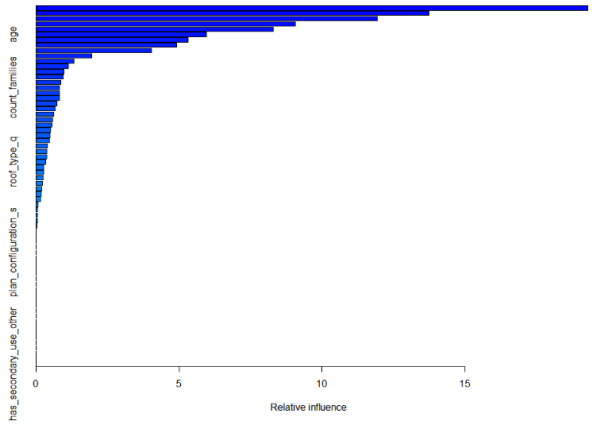
GBM은 단순정확도는 0.6에 못 미치지만 BCR의 경우 0.6을 넘어 다른모델의 비해 성능이 우수함을 알 수 있다.

Model	ACC	BCR
Multinomial Logistic Regression	0.5860629	0.3552882
ANN	0.5451065	0.4387583
CART	0.6439168	0
CART Bagging	0.1308394	0
ANN 30	0.585324	0.292199
ANN_Bagging	0.5891089	0.3094046
Random Forest	0.7098563	0.7100259
AdaBoost	0.6396099	0.2924276
GBM	0.5739179	0.6027088

다른 모델과의 종합적 비교를 해보면 GBM이 갖는 특징적인 장점이 있다. 대부분의 모델은 ACC가 높으면 BCR이 낮아지는 trade off 가 있다. 하지만 GBM은 그러지 않는 결과를 보여주고 있다. 비록 GBM의 단순 정확도, ACC 는 우수한편은 아니지만 BCR이 높은편에 속하므로 분류성능이 우수하다고 할 수 있겠다. 하지만 여전히 가장 성능이 좋은 모델은 Random Forest 이다.

변수별 중요도를 plot 해보면 아래와 같다.

var	rel.inf
geo_level_1_id	23.7970597
ground_floor_type_v	23.5408828
foundation_type_r	13.3461779
roof_type_x	11.1285189
age	8.7436079
has_superstructure_mud_mortar_stone	5.7198582
foundation_type_w	3.7479268
count_floors_pre_eq	2.3905817
foundation_type_i	1.9198031
area_percentage	1.6691561
has_superstructure_cement_mortar_brick	1.2474359
geo_level_2_id	0.9684405
foundation_type_u	0.4287030
has_superstructure_timber	0.3960193
other_floor_type_s	0.3757621
ground_floor_type_f	0.3064248
has_superstructure_stone_flag	0.2736413
geo_level_3_id	0.0000000



Random Forest와 주요 변수 비교를 하면 아래의 표와 같이 나타낼 수 있다.

Random Forest	Rel.Imp	Gradient Boosting Machine	Rel.Imp
geo_level_2_id	120.8847	geo_level_1_id	23.7970
geo_level_3_id	114.9935	ground_floor_type_v	23.5408
geo_level_1_id	109.7904	foundation_type_r	13.3461
area_percentage	90.34486	roof_type_x	11.1285
age	78.48908	age	8.7436
height_percentage	77.69182	has_superstructure_mud_mortar_stone	5.7198

Random Forest의 경우 geo_level_2_id, geo_level_3_id, geo_level_1_id, area_percentage, age, height_percentage 순으로 변수가 중요하며, GBM의 경우 geo_level_1_id, ground_floor_type_v, foundation_type_r, roof_type_x, age, has_superstructure_mud_mortar_stone 순서로 중요한 것을 알 수 있다. 공통적으로 geo_level_1_id, age 변수를 중요하다고 하는 것 또한 알 수 있다.두 모델 다 같은 데이터 셋을 통하여 학습 시킨 결과물이지만 서로 중요시 하는 변수가 다르다는 것을 알 수 있다.

[Q8] 모델 비교

Model	ACC	BCR
Multinomial Logistic Regression	0.5860629	0.3552882
ANN	0.5451065	0.4387583
CART	0.6439168	0
CART Bagging	0.1308394	0
ANN 30	0.585324	0.292199
ANN_Bagging	0.5891089	0.3094046
Random Forest	0.7098563	0.7100259
AdaBoost	0.6396099	0.2924276
GBM	0.5739179	0.6027088

Q1~Q7까지 알아본 총 9가지의 모델 중 BCR 관점에서 가장 성능이 우수한 것은 Random Forest 였다. 그다음으로는 GBM, ANN, MLR 등이 뒤를 이었다. 특이한 점은 CART와 CART_Bagging으로 만들어진 모델은 BCR 특성이 0이였다. 이 둘은 다른 모델에 비해 공격적인 모델로 1로 Predict 된 값이 존재하질 않아 BCR의 값이 0으로 나온 것이다. 이 이유는 전체 데이터 셋에서 1로 분류된 데이터 값이 매우 적기 때문에 발생했다고 볼 수 있다. 이에 따라 전제척인 데이터의 불균형을 해소하여 분류 성능을 향상시키는 방법을 Extra Question에서 해결하겠다.

[Extra Q]

전체적인 데이터의 불균형을 맞춰주기 위하여 class 2와 class 3의 데이터를 임의로 삭제하여 class 1가 동일한 데이터의 수를 가지게 만들어 주었다. (Down Sampling)

```
#Down Sampling
table(CART.tav$Class)
tree_1 <- CART.tav[which(CART.tav$Class==1),]
tree_2 <- CART.tav[which(CART.tav$Class==2),]
tree_3 <- CART.tav[which(CART.tav$Class==3),]
tree2_red <- tree_2[sample(1:nrow(tree_2), size = 19300),]
tree3_red <- tree_3[sample(1:nrow(tree_3), size = 19300),]
tree_red <- rbind(tree_1,tree2_red,tree3_red)
table(tree_red$Class)
str(tree_red$Class)
```

이를 바탕으로 random forest 알고리즘으로 test셋의 데이터를 분류하여 confusion matrix와 성능을 살펴보았다.

```
down <- randomForest(Class ~ ., data = tree_red, ntree = 300, importance = TRUE, do.trace = TRUE)
final_rf_pred <- predict(down, newdata = CART.tst, type = "class")
perf_eval_multi2(CART.tst$Class, final_rf_pred)
table(CART.tst$Class, final_rf_pred)
```

	1	2	3
1	4648	922	254
2	6032	18877	9386
3	899	5245	14338

위의 confusion matrix를 보았을 때 전에 비하여 Predict 1로 분류된 데이터의 수가 늘어난 것을 확인할 수 있다. 이에대한 ACC BCR 성능은 아래와 같다

Model	ACC	BCR
Down Sampling	0.6247917	0.6749750

Down Sampling 한 데이터를 바탕으로 만든 Randomforest의 성능은 ACC BCR 둘다 60퍼센트 이상을 나타내면서 우수한 성능을 가지고 있다고 할 수 있다. 물론 전체적인 학습데이터가 줄어 기존 Random Forest 보다 절대적인 성능이 떨어진 것을 확인 할 수있다. 위와 같은 문제를 해결하기위해 다음은 up sampling을 시행하였다.

```
#Up Sampling
# 종속 변수 비율 조정
tree_12 <- rbind(tree_1,tree_2)
tree_23 <- rbind(tree_3,tree_2)
table(tree_23$Class)
up_data_12 <- ovun.sample(Class~., data = tree_12, method = "both", N=160000)$data
up_data_23 <- ovun.sample(Class~., data = tree_23, method = "both", N=160000)$data
table(up_data_23$Class)
up_data <- rbind(up_data_12,up_data_23[which(up_data_23$Class==3),])
table(up_data$Class)
str(up_data$Class)
```

전체적인 데이터의 불균형을 맞춰주기 위하여 instance의 수가 적은 Class 1, 3의 숫 자는 늘려주고 Class 2의 값을 줄여 각 클래스의 instance 의 수를 약 80000개로 조절하였다.

```
up <- randomForest(Class ~ ., data = up, ntree = 300,  importance = TRUE, do.trace = TRUE)
final_up <- predict(up, newdata = CART.tst, type = "class")
perf_eval_multi2(CART.tst$Class, final_up)
cm<- table(CART.tst$Class, final_up)
cm1 <- cbind(cm[,2],cm[,1],cm[,3])
perf_eval_multi(cm1)
```

Up sampling을 시행한 결과 confusion matrix는 아래와 같다.

	1	2	3
1	4446	1157	221
2	4731	21129	8435
3	677	5344	14461

Up sampling의 경우도 downsampling과 마찬가지로 Predcit 1로 분류된 비율이 높아진 것을 확인 할 수 있다. 이때의 성능은 아래와 같다.

Model	ACC	BCR
Up Sampling	0.6606492	0.6924803

Down Sampling을 하여 구한 ACC 와 BCR 보다 Up sampling 하여 구한 모델의 성능의 ACC, BCR 이 조금이나마 더 높게 측정된 것을 확인 할 수 있다. 이는 Up sampling을 하여 구한 데이터 셋의 절대적인 양이 더 많아 학습의 정도가 더 높은 모델이 생성되었다고 할 수 있겠다. 하지만 여전히 오리지널 random forest 보다는 성능이 좋지 않다는 것을 알 수 있다.

Model	ACC	BCR
Random Forest	0.7098563	0.7100259
Down Sampling	0.6247917	0.6749750
Up Sampling	0.6606492	0.6924803

앞서 Down Sampling과 Up Sampling을 시행한 이유는 데이터의 imbalance 때문에 모델의 학습이 잘 일어나지 않아 이를 해결하기 위해 적용한 방법이었다. 하지만 기대와 달리 성능 향상은 나타나지 않았다. 이는 전체 데이터셋을 사용하지 않고 임의로 데이터를 추출하거나 제거하여 학습에 부정적인 방향으로 영향을 주었다고 해석 할 수 있다. 그러므로 데이터의 불균형을 해소하기 위해 임의로 데이터를 조정하는 것이 항상 성능향상을 일으킨다고 할 수 없다.