Homework 3

Jingyi Guo(jingyig1), Pittsburgh Campus

11/20/2017

```
rm(list=ls())
setwd("/Users/apple/Desktop/ML2/Homework3")
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 3.3.2
```

```
library(MASS)
library(randomForest)
```

```
## randomForest 4.6-12
```

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

```
library(pROC)
```

```
## Warning: package 'pROC' was built under R version 3.3.2
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
library(xgboost)
```

```
## Warning: package 'xgboost' was built under R version 3.3.2
```

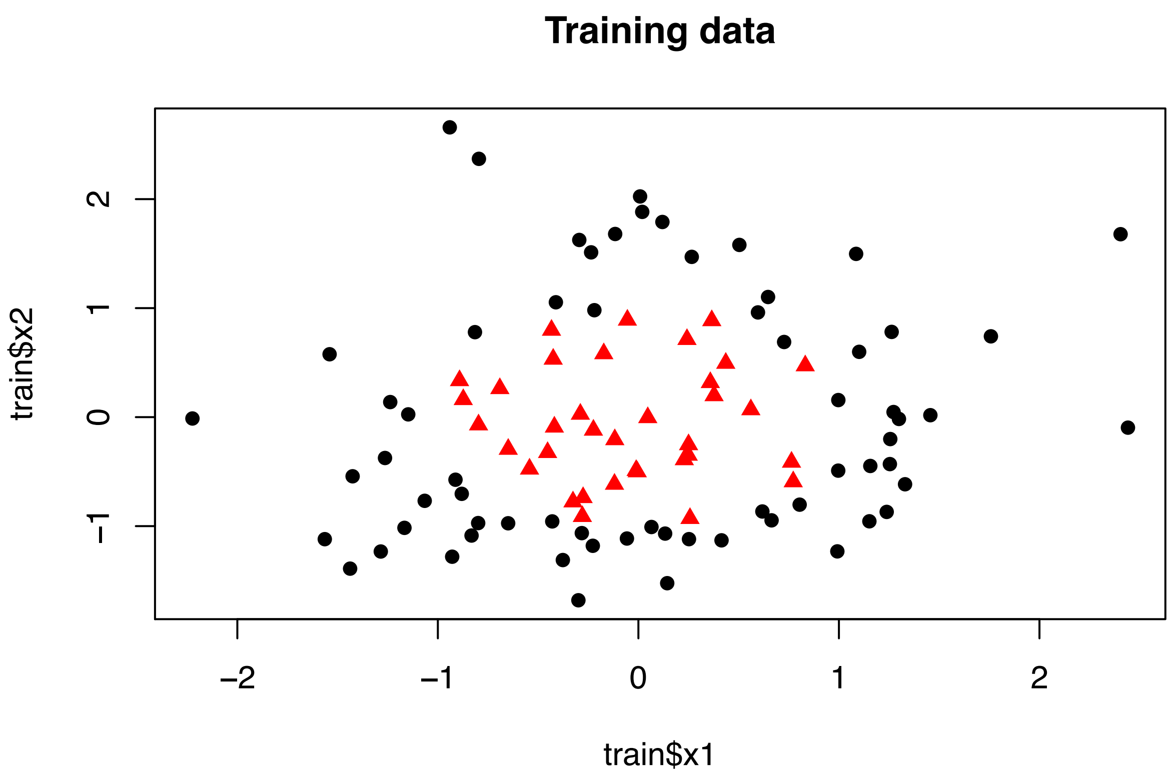
```
library(Matrix)
```

```
## Warning: package 'Matrix' was built under R version 3.3.2
```

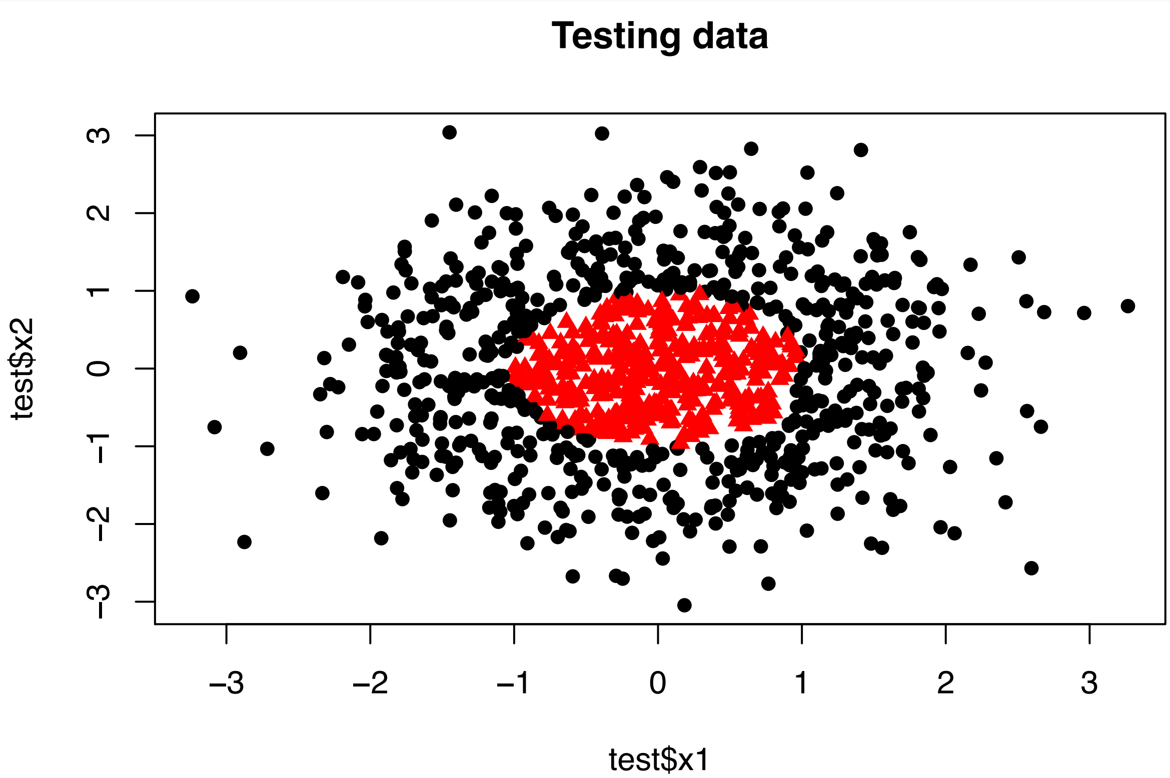
1.

(a)

```
set.seed(0)
get_circle_data = function(n){
  X = matrix(rnorm(2*n),ncol=2)
  Y = as.numeric(X[,1]^2+X[,2]^2<1)
  data.frame(x1=X[,1],x2=X[,2],y=as.factor(ifelse(Y==1,1,-1)))
}
train = get_circle_data(100)
test = get_circle_data(1000)
plot(train$x1,train$x2,pch=as.numeric(train$y) + 15,col=train$y, main="Training data")
```

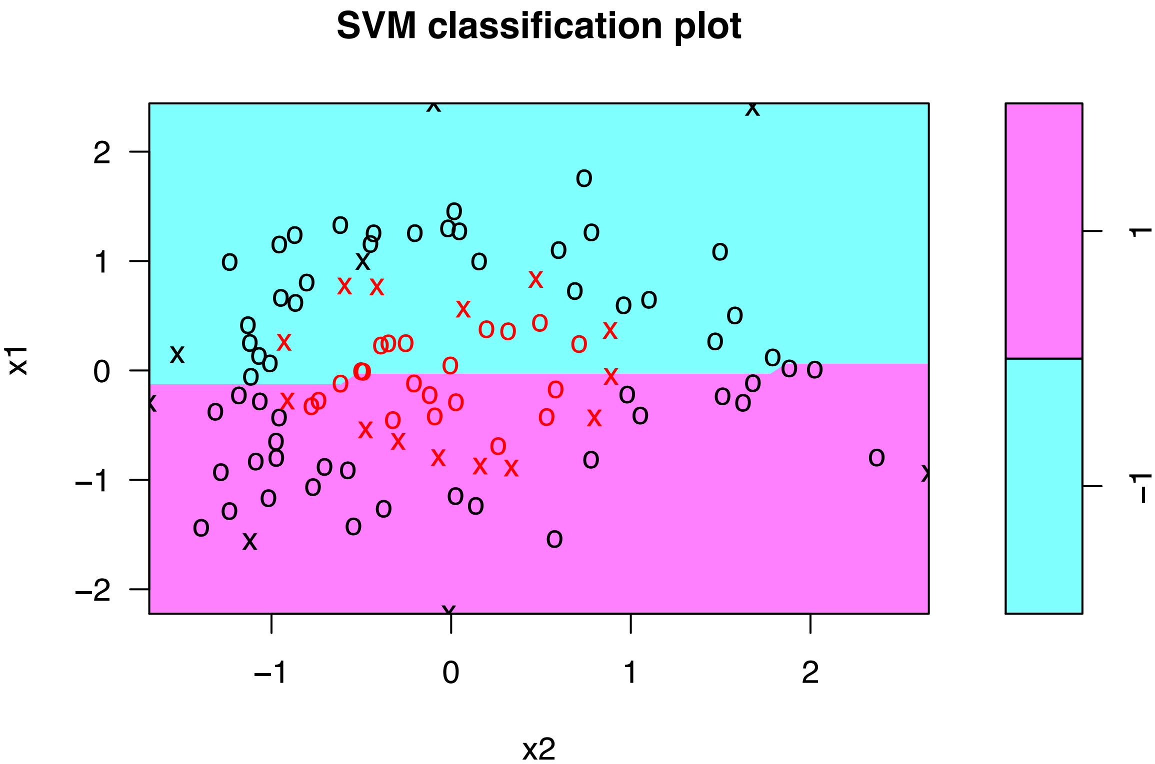



```
plot(test$x1,test$x2, pch=as.numeric(test$y) + 15, col=test$y, main="Testing data")
```



(b)

```
linearsvmfit=svm(y~.,data=train,kernel='linear',cost=1e7)
plot(linearsvmfit, train)
```



Linear SVM does a poor job fitting, the straight line decision boundary doesn't capture the actual boundary.

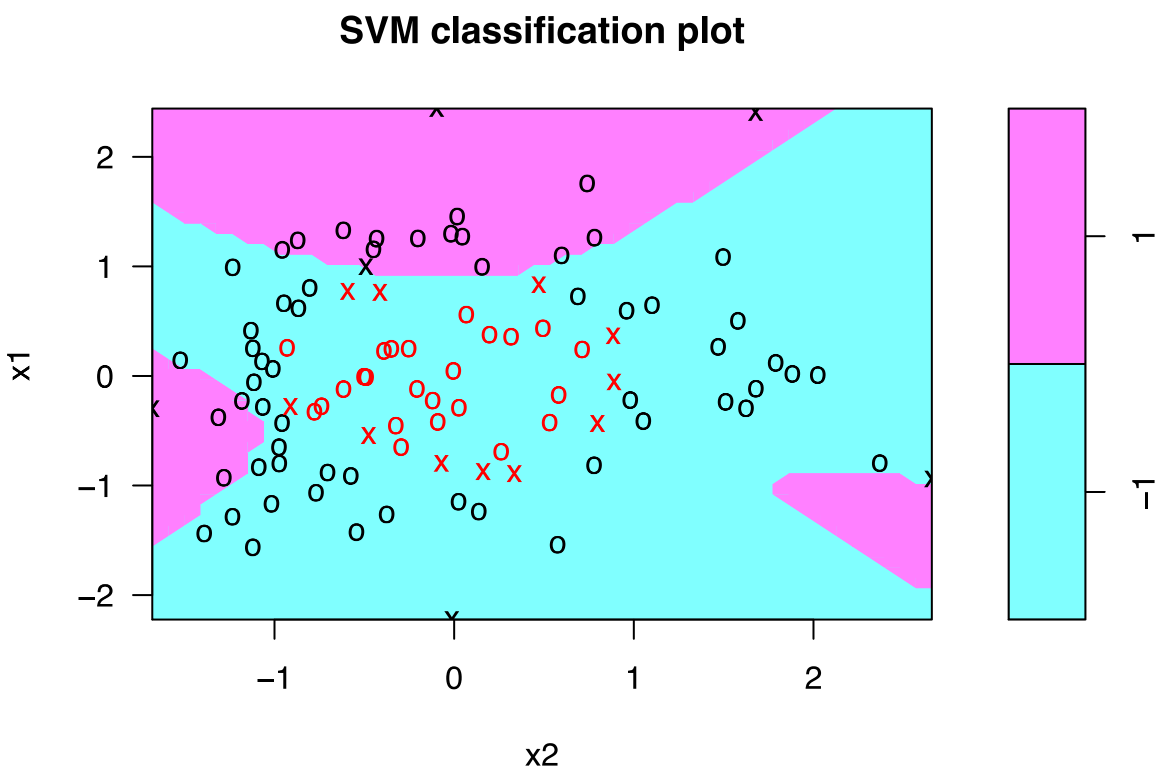
```
sum(predict(linearsvmfit, test) != test$y)/nrow(test)
```

```
## [1] 0.503
```

The misclassification error is 0.503.

(c)

```
polysvmfit=svm(y~.,data=train,kernel='polynomial',cost=1e7)
plot(polysvmfit, train)
```



Polynomial kernal works even worse. The decision boundary doesn't capture the actual boundary.

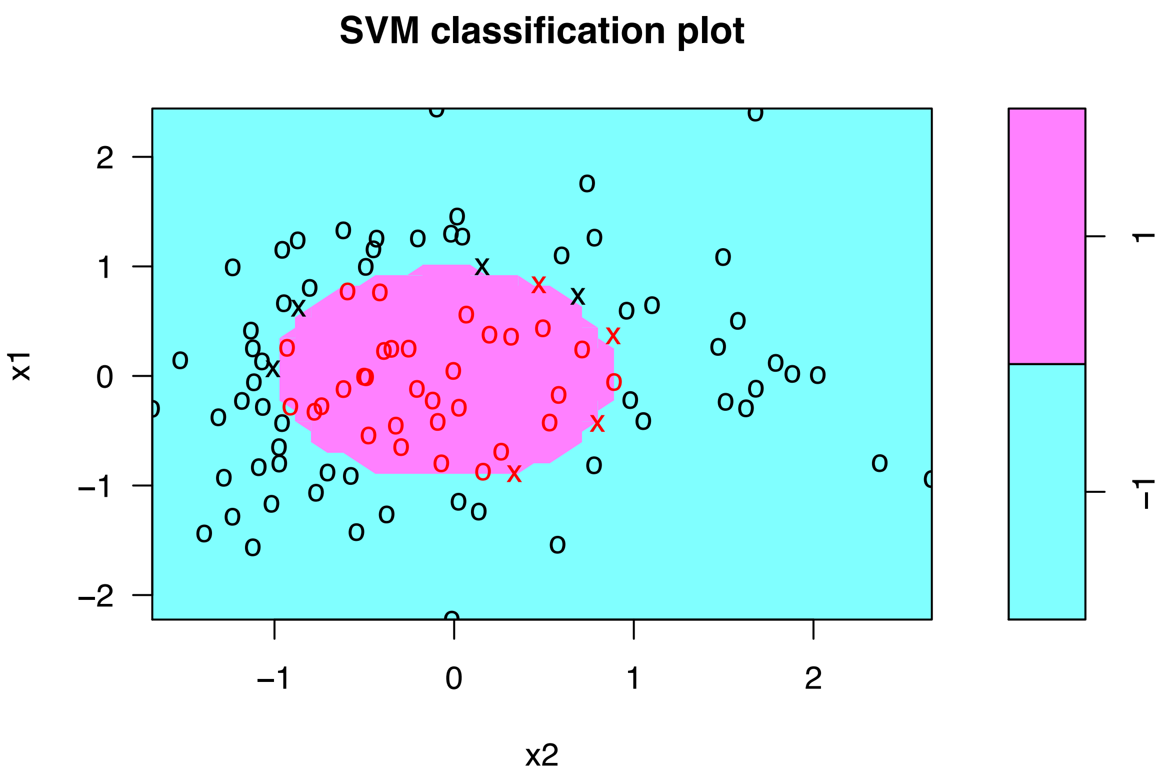
```
sum(predict(polysvmfit, test) != test$y)/nrow(test)
```

```
## [1] 0.568
```

The misclassification error is 0.568, even bigger.

(d)

```
polysvmfit2=svm(y~.,data=train,kernel='polynomial',cost=1000, degree = 2)
plot(polysvmfit2, train)
```



Setting degree=2 makes the fitting much better. The decision boundary is close to the circle.

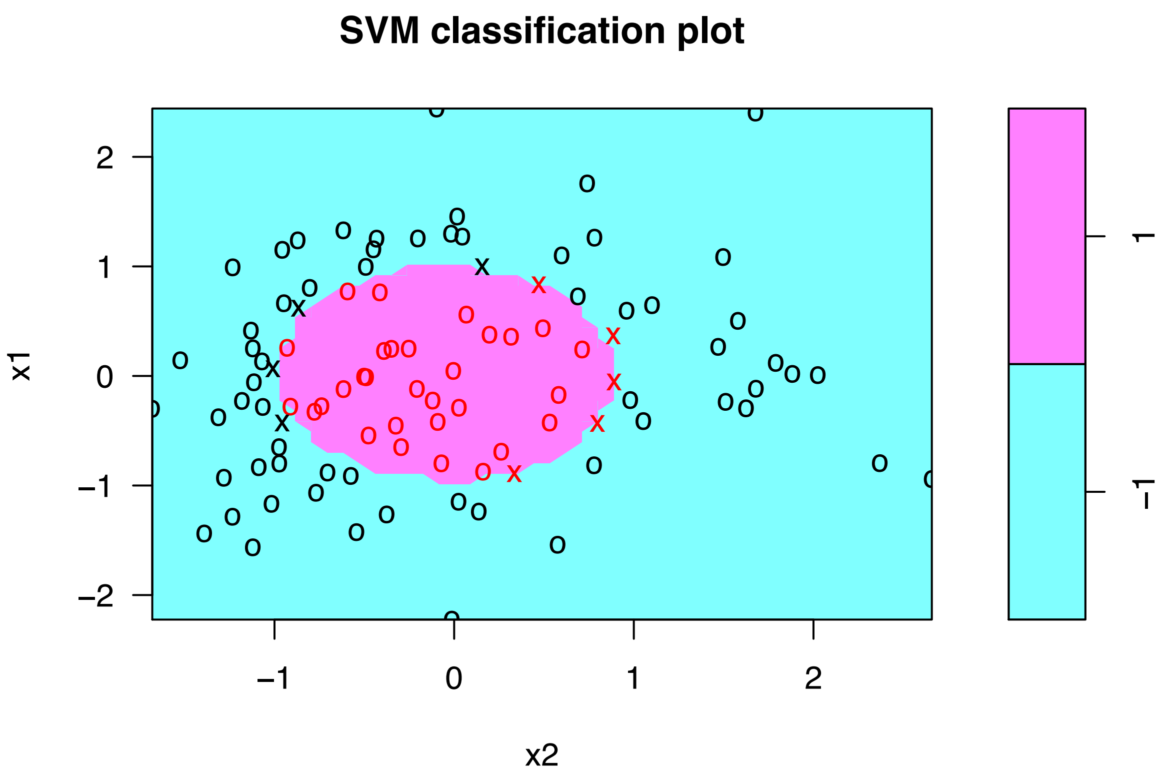
```
sum(predict(polysvmfit2, test) != test$y)/nrow(test)
```

```
## [1] 0.015
```

The misclassification error is 0.015, much smaller.

(e)

```
tune.poly = tune(svm, y~., data=train, kernel='polynomial', degree=2, ranges=list(cost=c(1000, 1e4, 1e5
plot(tune.poly$best.model, train)
```




```r
print(tune.poly$best.parameters)
```

```
##     cost gamma
## 35 1e+07     1
```

The parameters for the best model: cost=1e+07, gamma=1.

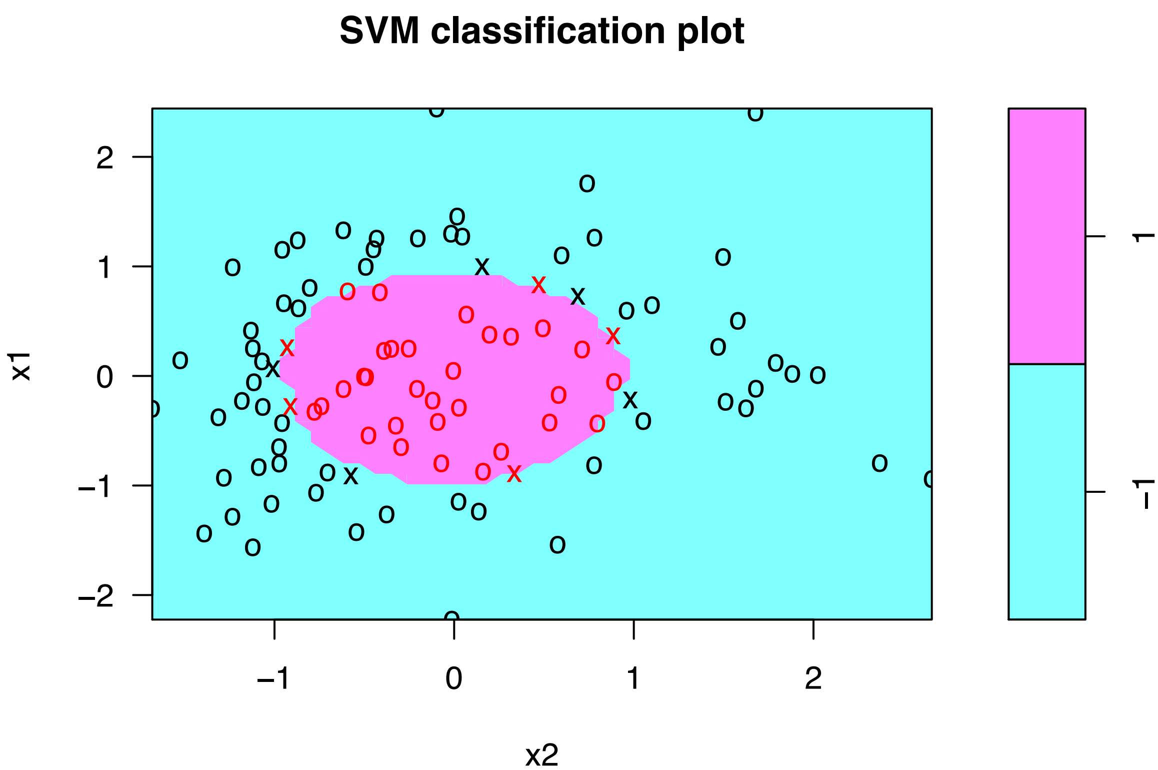
```r
sum(predict(tune.poly$best.model, test) != test$y)/nrow(test)
```

```
## [1] 0.014
```

The misclassification error is 0.014.

(f)

```r
radsvmfit=svm(y~.,data=train,kernel='radial',cost=1000)
plot(radsvmfit, train)
```



Radial kernal works pretty well. The decision boundary is close to the circle.

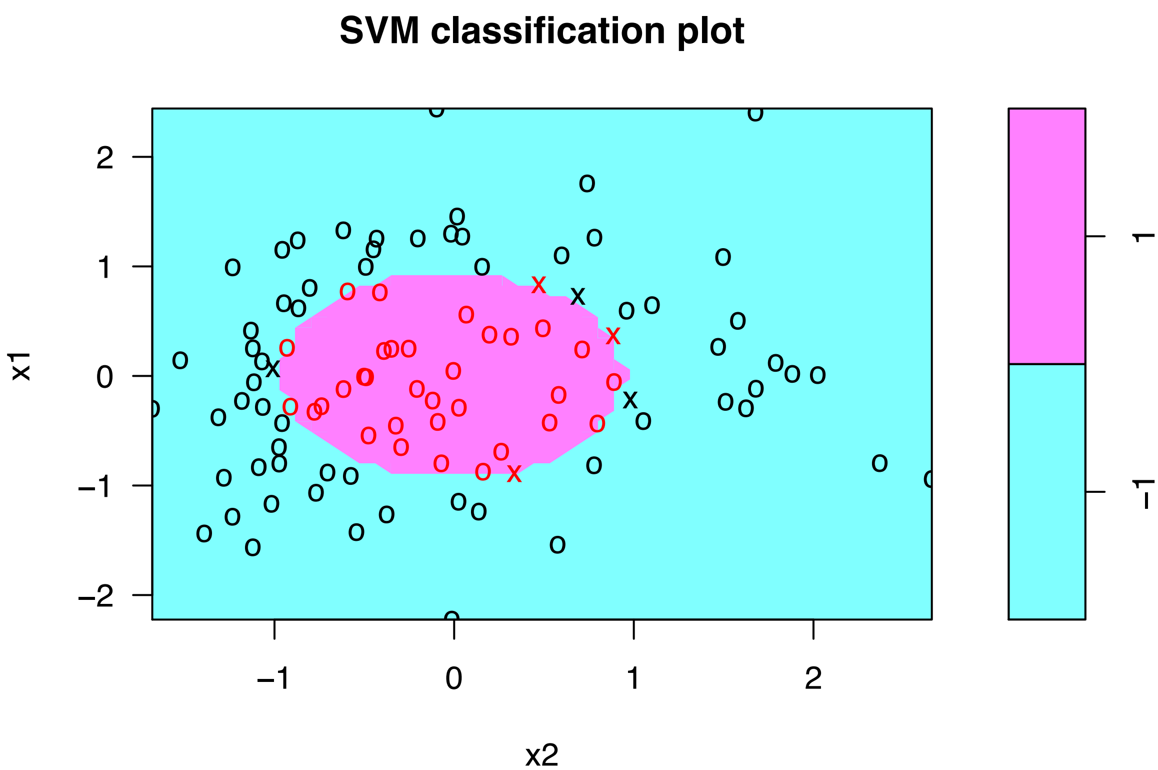
```
sum(predict(radsvmfit, test) != test$y)/nrow(test)
```

```
## [1] 0.009
```

The misclassification error is 0.009.

(g)

```
tune.rad = tune(svm, y~., data=train, kernel='radial', ranges=list(cost=c(.1,1,10,100,1000, 1e4, 1e5, 1e
plot(tune.rad$best.model, train)
```




```
print(tune.rad$best.parameters)
```

```
##    cost gamma
## 8 1e+06  0.01
```

The parameters for the best model: cost=1e+06, gamma=0.01.

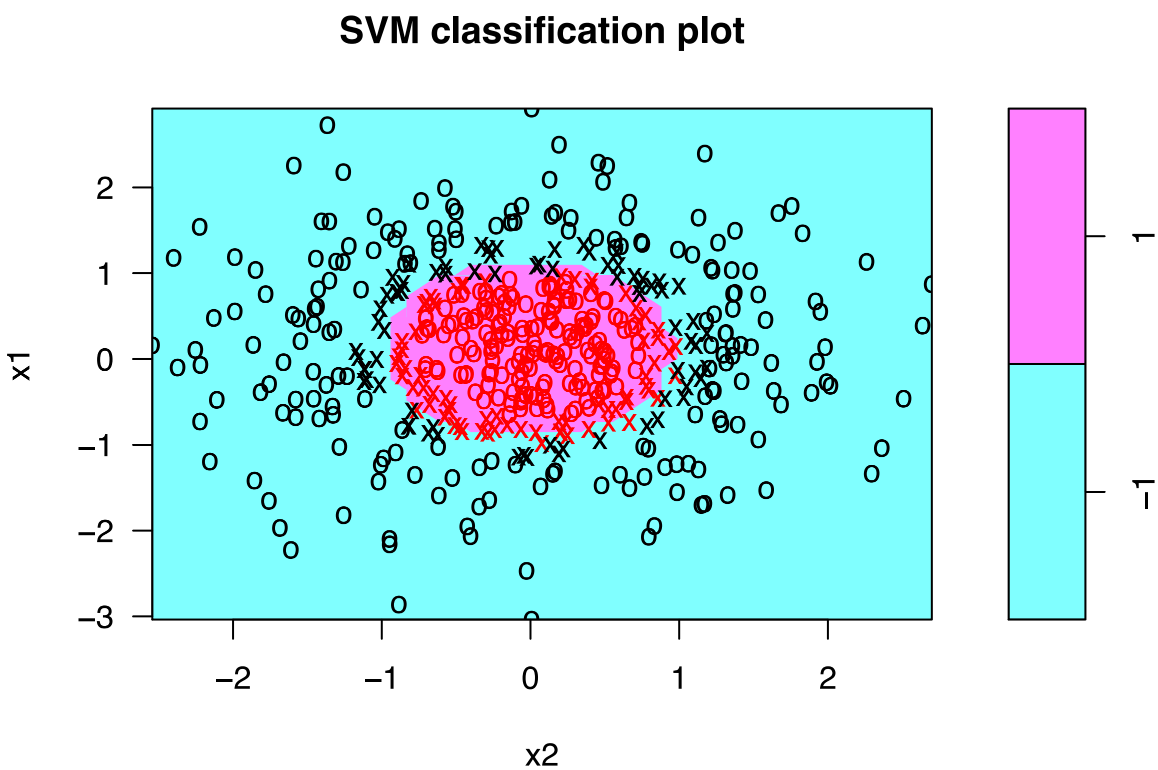
```
sum(predict(tune.rad$best.model, test) != test$y)/nrow(test)
```

```
## [1] 0.009
```

The misclassification error is 0.009.

(h)

```
train = get_circle_data(500)
tune.poly = tune(svm, y~., data=train, kernel='polynomial', degree=2,
                ranges=list(cost=c(1000, 1e4, 1e5, 1e6, 1e7, 1e8),
                            gamma=c(0.001, 0.005, 0.01, 0.05, .1, 1)))
plot(tune.poly$best.model, train)
```




```r
print(tune.poly$best.parameters)
```

```
##    cost gamma
## 3 1e+05 0.001
```

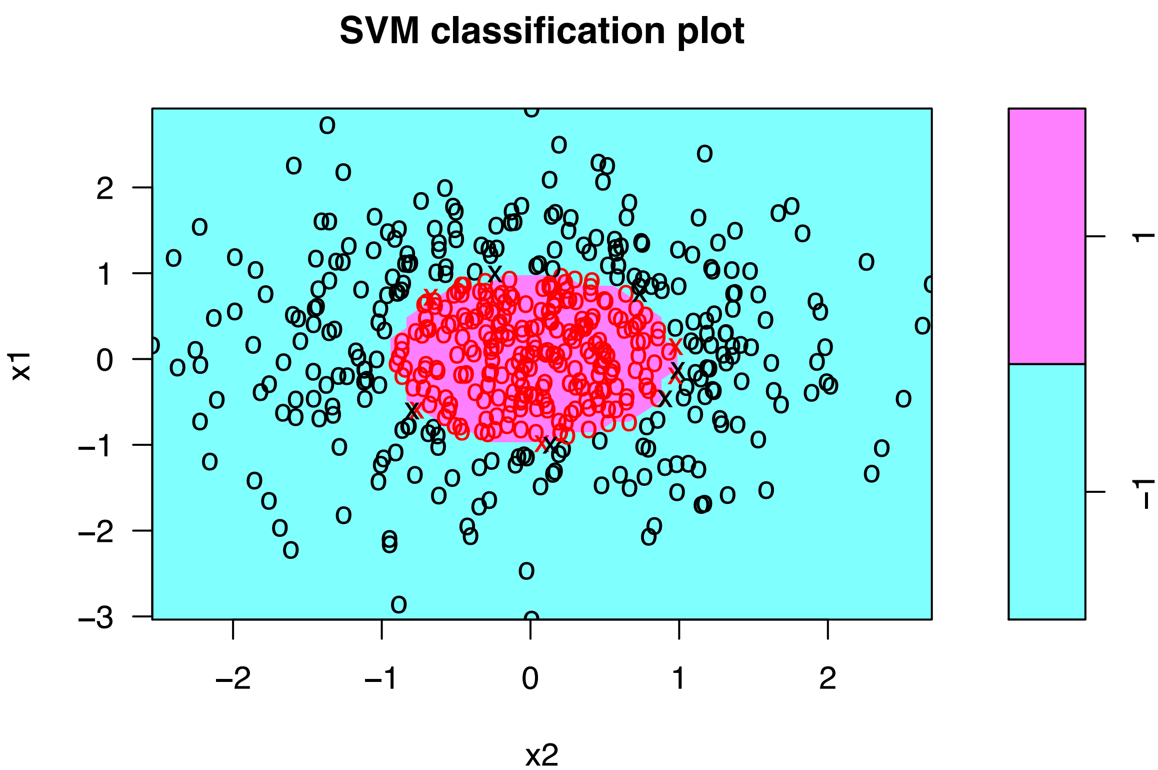
The parameters for the best model: cost=1e+05, gamma=0.001.

```r
sum(predict(tune.poly$best.model, test) != test$y)/nrow(test)
```

```
## [1] 0.044
```

The misclassification error is 0.044.

```r
tune.rad = tune(svm, y~., data=train, kernel='radial',
                ranges=list(cost=c(.1,1,10,100,1000, 1e4, 1e5, 1e6),
                            gamma=c(0.01, 0.05, .1, .5,1,2,3)))
plot(tune.rad$best.model, train)
```




```r
print(tune.rad$best.parameters)
```

```
##     cost gamma
## 30 10000   0.5
```

The parameters for the best model: cost=1e+04, gamma=0.5.

```r
sum(predict(tune.rad$best.model, test) != test$y)/nrow(test)
```

```
## [1] 0.008
```

The misclassification error is 0.008.

I would pick the SVM classifier with radial kernel because it has the lowest test set misclassification error. Misclassification error for radial SVM is 0.6%, for AdaBoost is 1.6%, for polynomial SVM is 3.1%. According to the comparison of classification error, AdaBoost is better than polynomial SVM but worse than radial SVM.

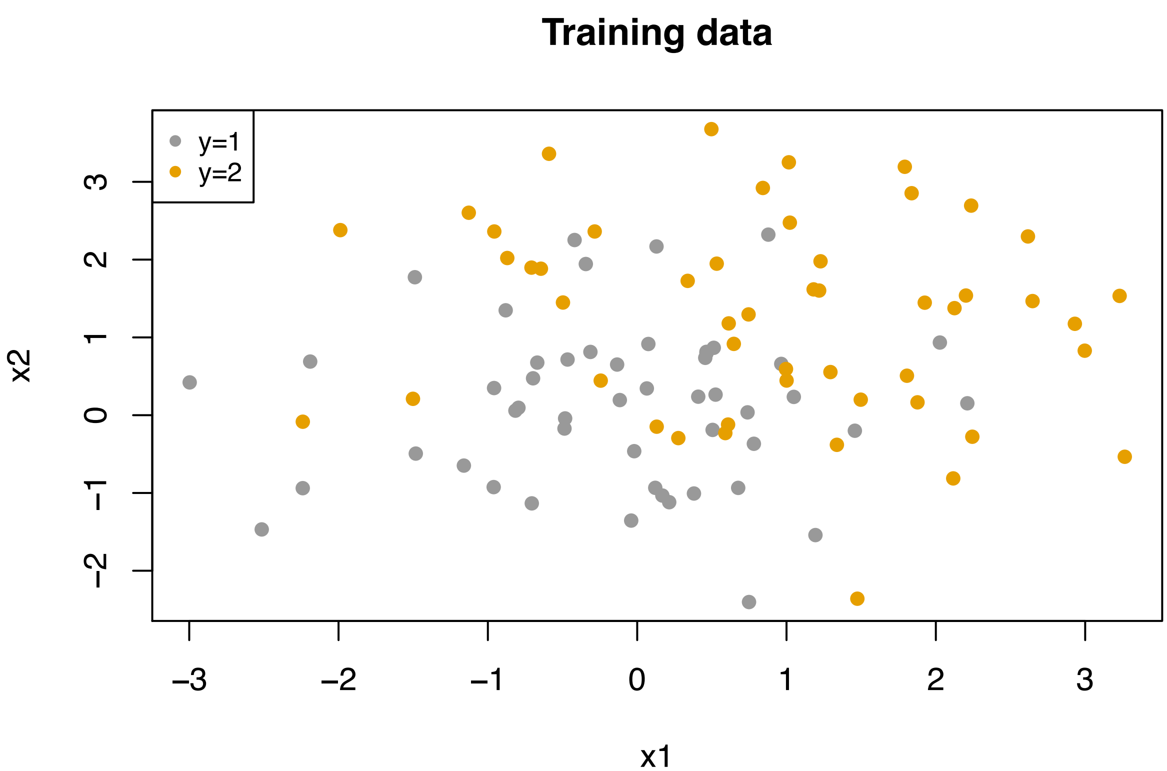
2.

```r
getdata = function(n,p){
  rho = .3
  Sig1 = diag(p)
  Sig2 = matrix(rho,p,p)
  diag(Sig2)=2
  mu1 = matrix(rep(0,p))
  mu2 = matrix(rep(1,p))
  X1 = mvrnorm(n/2,mu1,Sig1)
  X2 = mvrnorm(n/2,mu2,Sig2)
  y1 = rep(1,n/2)
  y2 = rep(2,n/2)
```

```
  X = rbind(X1,X2)
  y = as.factor(c(y1,y2))
  list(X=X,y=y)
}
```

(a)

```
n = 100
p = 20
set.seed(1)
train = getdata(n,p)
test = getdata(n,p)
cbPalette <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00")
plot(train$X[,1],train$X[,2], col=cbPalette[train$y], pch=16,
     main="Training data", xlab = 'x1', ylab = 'x2')
legend("topleft", legend=c("y=1", "y=2"), pch=c(16,16),col=cbPalette[c(1,2)],cex=0.8)
```



(b)

```
lda_fit = lda(train$X, train$y)
qda_fit = qda(train$X, train$y)
mean(predict(lda_fit)$class != train$y)
```

```
## [1] 0.08
```

Training error for LDA is 0.08.

```r
mean(predict(qda_fit)$class != train$y)
```

```
## [1] 0
```

Training error for QDA is 0.

QDA method has smaller training error than LDA because the data are generated based on different variables. LDA made incorrect assumptions, while QDA satisfied assumptions.

(c)

```r
mean(predict(lda_fit, newdata = test$X)$class != test$y)
```

```
## [1] 0.19
```

Test error for LDA is 0.19.

```r
mean(predict(qda_fit, newdata = test$X)$class != test$y)
```

```
## [1] 0.38
```

Test error for QDA is 0.38.

Here LDA has smaller test set error.

(d)

Compared with LDA, QDA has more parameters to estimate, so it's more likely to perform poorly out-of-sample when covariance matrices estimated from training set are not close to the true values.

3.

See last page

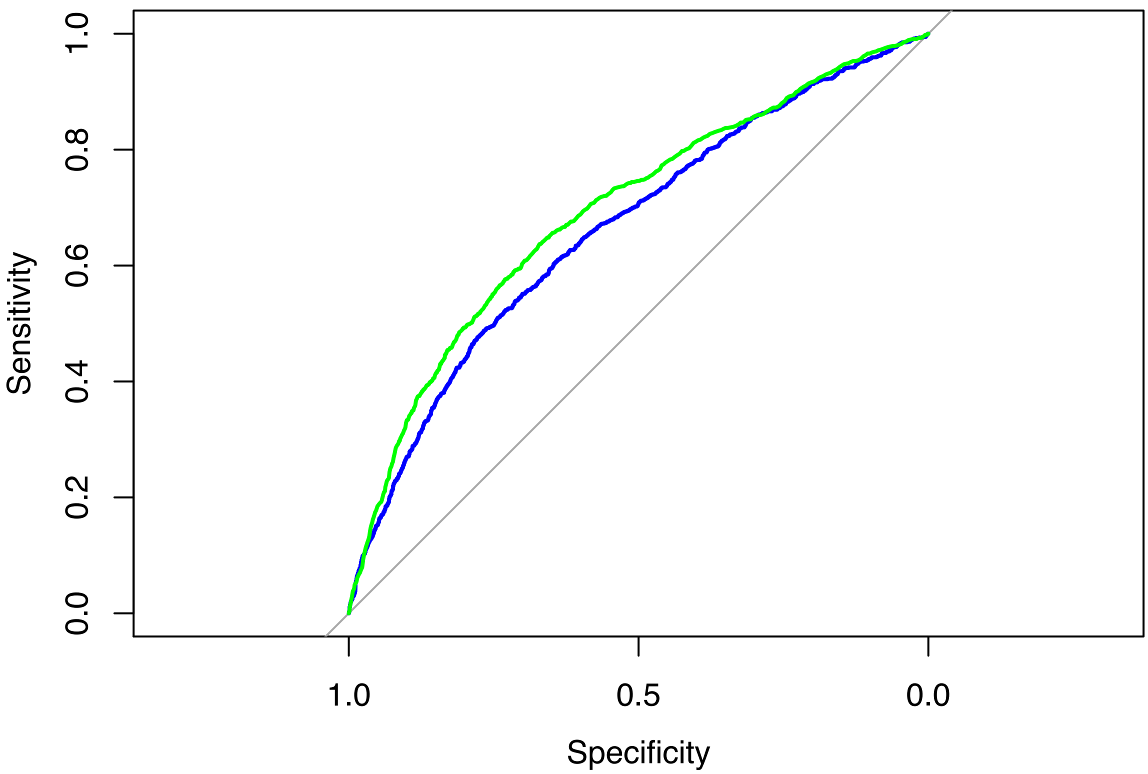
4.

```r
marketing = read.csv('marketing.csv')
set.seed(1)
idx.test = sample(1:nrow(marketing),floor(0.2*nrow(marketing)))
test = marketing[idx.test,]
train_full = marketing[-idx.test,]
#Split off another piece of our training set as a validation set.  We will use this for tuning
idx.valid = sample(1:nrow(train_full),floor(0.25*nrow(train_full)))
valid = train_full[idx.valid,]
train = train_full[-idx.valid,]

#Fit logistic regression
fitlm = glm(y~.,data = train_full, family='binomial')
guess_lm = predict(fitlm,newdata=test, type='response')

#Fit a balanced random forest
```

```
library(randomForest)
nsmall = sum(train$y=='yes')
forest_bal = randomForest(train_full[,1:8], train_full[,9], strata=train_full$y, sampsize=c(nsmall,nsmal
guess_bal = predict(forest_bal, test[,1:8], type='prob')[,2]

#Draw roc curves
roc(test[,9], guess_lm, col='blue', plot = TRUE, add=FALSE)
```

```
##
## Call:
## roc.default(response = test[, 9], predictor = guess_lm, plot = TRUE,     col = "blue", add = FALSE)
##
## Data: guess_lm in 7959 controls (test[, 9] no) < 1083 cases (test[, 9] yes).
## Area under the curve: 0.6634
```

```
roc(test[,9], guess_bal, col='green', plot = TRUE, add=TRUE)
```




```
##
## Call:
## roc.default(response = test[, 9], predictor = guess_bal, plot = TRUE,     col = "green", add = TRUE)
##
## Data: guess_bal in 7959 controls (test[, 9] no) < 1083 cases (test[, 9] yes).
## Area under the curve: 0.694
```

```
library(xgboost)
#Reformat the data for xgboost
train_expanded = sparse.model.matrix(y ~ .-1, data = train)
valid_expanded = sparse.model.matrix(y ~ .-1, data = valid)
test_expanded = sparse.model.matrix(y ~ .-1, data = test)
train_y = (train$y == 'yes')
```

```
valid_y = (valid$y == 'yes')
test_y = (test$y == 'yes')
dtrain = xgb.DMatrix(data=train_expanded, label=train_y)
dvalid = xgb.DMatrix(data=valid_expanded, label=valid_y)
dtest = xgb.DMatrix(data=test_expanded, label=test_y)
```

(a)

```
boost_simple = xgb.train(list(objective='binary:logistic'), dtrain, nround=10, verbose=2)
```

```
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 72 extra nodes, 
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 82 extra nodes, 
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 80 extra nodes, 
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 104 extra nodes,
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 110 extra nodes,
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 110 extra nodes,
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 116 extra nodes,
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 112 extra nodes,
## [14:35:20] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 108 extra nodes,
## [14:35:21] amalgamation/../src/tree/updater_prune.cc:74: tree pruning end, 1 roots, 114 extra nodes,
```

```
guess_simpleboost = predict(boost_simple, dtest)
mean((guess_simpleboost > 0.5) != test_y)
```

```
## [1] 0.1196638
```

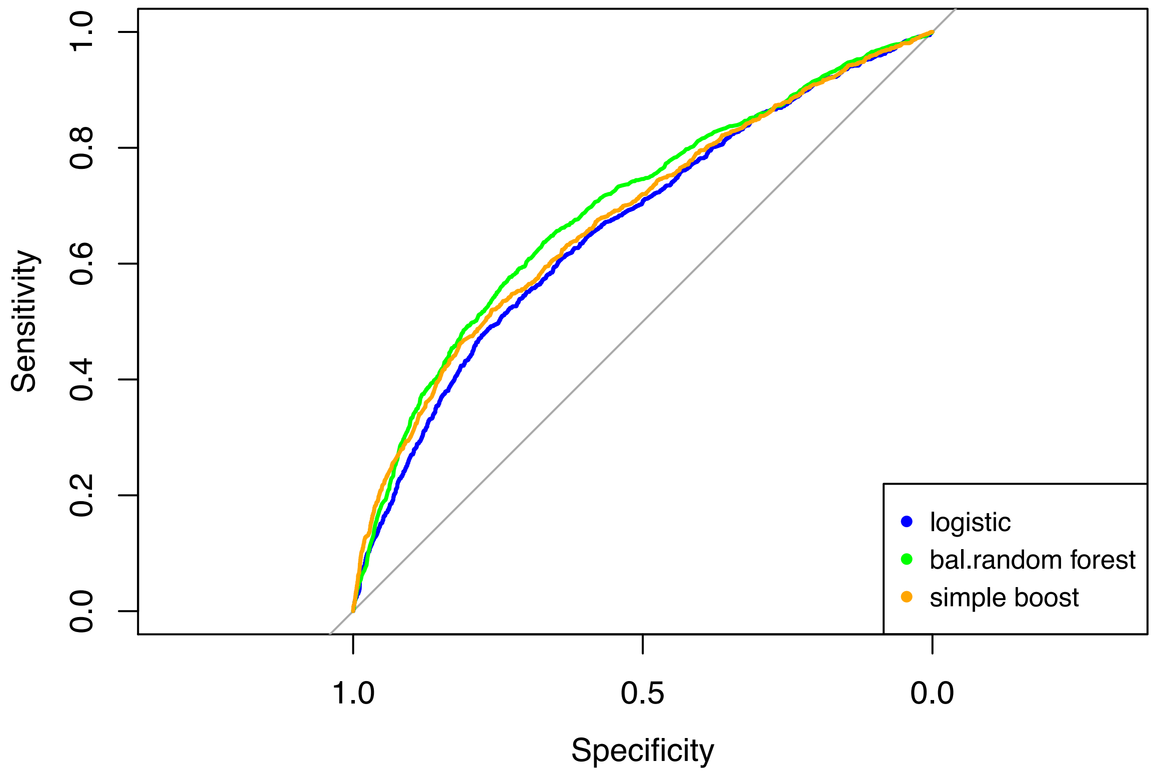
Misclassification error (threshold=0.5) for simple boosted tree is 0.1196638.

```
mean((guess_bal > 0.5) != test_y)
```

```
## [1] 0.2817961
```

Misclassification error (threshold=0.5) for balanced random forest is 0.2817961.

```
ROC = roc(test_y, guess_lm, col='blue', plot = TRUE, add=FALSE)
ROC = roc(test_y, guess_bal, col='green', plot = TRUE, add=TRUE)
ROC = roc(test_y, guess_simpleboost, col = 'orange', plot = T, add=T)
legend("bottomright", legend=c("logistic", "bal.random forest", "simple boost"), pch=c(16,16),col=c('bl
```



When we set threshold to be 0.5, the ROC curve of balanced random forest is highest, indicating that the random forest has better ROC.

(b)

```
nround=c(50, 100, 300)
max_depth=c(10, 20)
eta=c(0.05, 0.1, 0.3)
subsample=c(0.5, 0.8)
scale_pos_weight=c(0.2, 0.5)
param_matrix = data.frame(as.matrix(expand.grid(nround, max_depth, eta,subsample, scale_pos_weight)))
param_names = c('nround', 'max_depth','eta', 'subsample','scale_pos_weight')
colnames(param_matrix)= param_names
tune_boost = function(param_matrix, dtrain, eval_metric = 'auc', quiet=F){
    objectives = vector('numeric', length = 0)
    for (i in 1:nrow(param_matrix)){
    param = param_matrix[i, ]
    paramlist = list(objective='binary:logistic', eval_metric=eval_metric,
                   max_depth=param$max_depth, eta=param$eta,
                   scale_pos_weight=param$scale_pos_weight,
                   subsample=param$subsample)
    watchlist = list(train=dtrain, validation=dvalid)
    out = xgb.train(paramlist, dtrain, nround=param$nround, verbose=0,
                    watchlist=watchlist, callbacks = list(cb.evaluation.log()))
    objectives[i] =  as.numeric(out$evaluation_log[out$niter, 3])
    if (!quiet){
        cat('param:', as.numeric(param),
```

```
      '  validation error:', objectives[i], '\n')
    }}
    if (eval_metric=='auc'){
      best_param = param_matrix[which.max(objectives),]
      best_obj = max(objectives)
    }
    else{
      best_param = param_matrix[which.min(objectives),]
      best_obj = min(objectives)
    }
    return(list("best_param" = best_param, "best_obj"=best_obj,
                'params'=param_matrix, 'objs'=objectives))
}
out_error = tune_boost(param_matrix, dtrain, eval_metric = 'error')
```

```
## param: 50 10 0.05 0.5 0.2   validation error: 0.117562
## param: 100 10 0.05 0.5 0.2   validation error: 0.117562
## param: 300 10 0.05 0.5 0.2   validation error: 0.117341
## param: 50 20 0.05 0.5 0.2   validation error: 0.117562
## param: 100 20 0.05 0.5 0.2   validation error: 0.117562
## param: 300 20 0.05 0.5 0.2   validation error: 0.11701
## param: 50 10 0.1 0.5 0.2   validation error: 0.117562
## param: 100 10 0.1 0.5 0.2   validation error: 0.117341
## param: 300 10 0.1 0.5 0.2   validation error: 0.116125
## param: 50 20 0.1 0.5 0.2   validation error: 0.117562
## param: 100 20 0.1 0.5 0.2   validation error: 0.117452
## param: 300 20 0.1 0.5 0.2   validation error: 0.115572
## param: 50 10 0.3 0.5 0.2   validation error: 0.117341
## param: 100 10 0.3 0.5 0.2   validation error: 0.115904
## param: 300 10 0.3 0.5 0.2   validation error: 0.115904
## param: 50 20 0.3 0.5 0.2   validation error: 0.116899
## param: 100 20 0.3 0.5 0.2   validation error: 0.116125
## param: 300 20 0.3 0.5 0.2   validation error: 0.116235
## param: 50 10 0.05 0.8 0.2   validation error: 0.117562
## param: 100 10 0.05 0.8 0.2   validation error: 0.117562
## param: 300 10 0.05 0.8 0.2   validation error: 0.116899
## param: 50 20 0.05 0.8 0.2   validation error: 0.117562
## param: 100 20 0.05 0.8 0.2   validation error: 0.117341
## param: 300 20 0.05 0.8 0.2   validation error: 0.116125
## param: 50 10 0.1 0.8 0.2   validation error: 0.117562
## param: 100 10 0.1 0.8 0.2   validation error: 0.117231
## param: 300 10 0.1 0.8 0.2   validation error: 0.116014
## param: 50 20 0.1 0.8 0.2   validation error: 0.117231
## param: 100 20 0.1 0.8 0.2   validation error: 0.11701
## param: 300 20 0.1 0.8 0.2   validation error: 0.114798
## param: 50 10 0.3 0.8 0.2   validation error: 0.116678
## param: 100 10 0.3 0.8 0.2   validation error: 0.115572
## param: 300 10 0.3 0.8 0.2   validation error: 0.114245
## param: 50 20 0.3 0.8 0.2   validation error: 0.115572
## param: 100 20 0.3 0.8 0.2   validation error: 0.114576
## param: 300 20 0.3 0.8 0.2   validation error: 0.116678
## param: 50 10 0.05 0.5 0.5   validation error: 0.117231
## param: 100 10 0.05 0.5 0.5   validation error: 0.116567
## param: 300 10 0.05 0.5 0.5   validation error: 0.116125
```

```
## param: 50 20 0.05 0.5 0.5   validation error: 0.117452
## param: 100 20 0.05 0.5 0.5   validation error: 0.116457
## param: 300 20 0.05 0.5 0.5   validation error: 0.114134
## param: 50 10 0.1 0.5 0.5   validation error: 0.116457
## param: 100 10 0.1 0.5 0.5   validation error: 0.115904
## param: 300 10 0.1 0.5 0.5   validation error: 0.114355
## param: 50 20 0.1 0.5 0.5   validation error: 0.11712
## param: 100 20 0.1 0.5 0.5   validation error: 0.114466
## param: 300 20 0.1 0.5 0.5   validation error: 0.114134
## param: 50 10 0.3 0.5 0.5   validation error: 0.117452
## param: 100 10 0.3 0.5 0.5   validation error: 0.116235
## param: 300 10 0.3 0.5 0.5   validation error: 0.121655
## param: 50 20 0.3 0.5 0.5   validation error: 0.115904
## param: 100 20 0.3 0.5 0.5   validation error: 0.117784
## param: 300 20 0.3 0.5 0.5   validation error: 0.119221
## param: 50 10 0.05 0.8 0.5   validation error: 0.116788
## param: 100 10 0.05 0.8 0.5   validation error: 0.116235
## param: 300 10 0.05 0.8 0.5   validation error: 0.115904
## param: 50 20 0.05 0.8 0.5   validation error: 0.115904
## param: 100 20 0.05 0.8 0.5   validation error: 0.114908
## param: 300 20 0.05 0.8 0.5   validation error: 0.114245
## param: 50 10 0.1 0.8 0.5   validation error: 0.116014
## param: 100 10 0.1 0.8 0.5   validation error: 0.115572
## param: 300 10 0.1 0.8 0.5   validation error: 0.116346
## param: 50 20 0.1 0.8 0.5   validation error: 0.115351
## param: 100 20 0.1 0.8 0.5   validation error: 0.114576
## param: 300 20 0.1 0.8 0.5   validation error: 0.114355
## param: 50 10 0.3 0.8 0.5   validation error: 0.115904
## param: 100 10 0.3 0.8 0.5   validation error: 0.117452
## param: 300 10 0.3 0.8 0.5   validation error: 0.120327
## param: 50 20 0.3 0.8 0.5   validation error: 0.115461
## param: 100 20 0.3 0.8 0.5   validation error: 0.115572
## param: 300 20 0.3 0.8 0.5   validation error: 0.119664
```

```r
out_auc = tune_boost(param_matrix, dtrain, eval_metric = 'auc')
```

```
## param: 50 10 0.05 0.5 0.2   validation error: 0.694089
## param: 100 10 0.05 0.5 0.2   validation error: 0.702991
## param: 300 10 0.05 0.5 0.2   validation error: 0.700381
## param: 50 20 0.05 0.5 0.2   validation error: 0.69662
## param: 100 20 0.05 0.5 0.2   validation error: 0.704863
## param: 300 20 0.05 0.5 0.2   validation error: 0.698978
## param: 50 10 0.1 0.5 0.2   validation error: 0.698271
## param: 100 10 0.1 0.5 0.2   validation error: 0.697342
## param: 300 10 0.1 0.5 0.2   validation error: 0.691491
## param: 50 20 0.1 0.5 0.2   validation error: 0.700477
## param: 100 20 0.1 0.5 0.2   validation error: 0.695503
## param: 300 20 0.1 0.5 0.2   validation error: 0.690209
## param: 50 10 0.3 0.5 0.2   validation error: 0.690035
## param: 100 10 0.3 0.5 0.2   validation error: 0.681649
## param: 300 10 0.3 0.5 0.2   validation error: 0.682183
## param: 50 20 0.3 0.5 0.2   validation error: 0.683534
## param: 100 20 0.3 0.5 0.2   validation error: 0.685399
## param: 300 20 0.3 0.5 0.2   validation error: 0.669458
## param: 50 10 0.05 0.8 0.2   validation error: 0.69449
```

```
## param: 100 10 0.05 0.8 0.2   validation error: 0.703086
## param: 300 10 0.05 0.8 0.2   validation error: 0.702106
## param: 50 20 0.05 0.8 0.2   validation error: 0.697065
## param: 100 20 0.05 0.8 0.2   validation error: 0.705692
## param: 300 20 0.05 0.8 0.2   validation error: 0.69873
## param: 50 10 0.1 0.8 0.2   validation error: 0.701402
## param: 100 10 0.1 0.8 0.2   validation error: 0.705903
## param: 300 10 0.1 0.8 0.2   validation error: 0.697208
## param: 50 20 0.1 0.8 0.2   validation error: 0.701746
## param: 100 20 0.1 0.8 0.2   validation error: 0.699935
## param: 300 20 0.1 0.8 0.2   validation error: 0.690251
## param: 50 10 0.3 0.8 0.2   validation error: 0.695403
## param: 100 10 0.3 0.8 0.2   validation error: 0.682704
## param: 300 10 0.3 0.8 0.2   validation error: 0.683178
## param: 50 20 0.3 0.8 0.2   validation error: 0.686315
## param: 100 20 0.3 0.8 0.2   validation error: 0.684112
## param: 300 20 0.3 0.8 0.2   validation error: 0.67445
## param: 50 10 0.05 0.5 0.5   validation error: 0.70246
## param: 100 10 0.05 0.5 0.5   validation error: 0.706669
## param: 300 10 0.05 0.5 0.5   validation error: 0.701679
## param: 50 20 0.05 0.5 0.5   validation error: 0.700929
## param: 100 20 0.05 0.5 0.5   validation error: 0.704517
## param: 300 20 0.05 0.5 0.5   validation error: 0.697802
## param: 50 10 0.1 0.5 0.5   validation error: 0.707551
## param: 100 10 0.1 0.5 0.5   validation error: 0.701508
## param: 300 10 0.1 0.5 0.5   validation error: 0.692347
## param: 50 20 0.1 0.5 0.5   validation error: 0.703323
## param: 100 20 0.1 0.5 0.5   validation error: 0.702923
## param: 300 20 0.1 0.5 0.5   validation error: 0.684986
## param: 50 10 0.3 0.5 0.5   validation error: 0.679563
## param: 100 10 0.3 0.5 0.5   validation error: 0.676105
## param: 300 10 0.3 0.5 0.5   validation error: 0.669797
## param: 50 20 0.3 0.5 0.5   validation error: 0.676958
## param: 100 20 0.3 0.5 0.5   validation error: 0.675534
## param: 300 20 0.3 0.5 0.5   validation error: 0.67944
## param: 50 10 0.05 0.8 0.5   validation error: 0.701835
## param: 100 10 0.05 0.8 0.5   validation error: 0.707422
## param: 300 10 0.05 0.8 0.5   validation error: 0.707317
## param: 50 20 0.05 0.8 0.5   validation error: 0.70412
## param: 100 20 0.05 0.8 0.5   validation error: 0.705943
## param: 300 20 0.05 0.8 0.5   validation error: 0.697427
## param: 50 10 0.1 0.8 0.5   validation error: 0.702938
## param: 100 10 0.1 0.8 0.5   validation error: 0.703317
## param: 300 10 0.1 0.8 0.5   validation error: 0.699726
## param: 50 20 0.1 0.8 0.5   validation error: 0.706614
## param: 100 20 0.1 0.8 0.5   validation error: 0.703508
## param: 300 20 0.1 0.8 0.5   validation error: 0.689808
## param: 50 10 0.3 0.8 0.5   validation error: 0.698106
## param: 100 10 0.3 0.8 0.5   validation error: 0.693111
## param: 300 10 0.3 0.8 0.5   validation error: 0.679892
## param: 50 20 0.3 0.8 0.5   validation error: 0.691799
## param: 100 20 0.3 0.8 0.5   validation error: 0.683689
## param: 300 20 0.3 0.8 0.5   validation error: 0.676357
```

Best paramaters for each eval_metric:

```
best_params = data.frame(rbind(out_error$best_param,out_auc$best_param),row.names = c('error', 'auc'))
colnames(best_params) = param_names
print(best_params)
```

```
##       nround max_depth  eta subsample scale_pos_weight
## error    300        20 0.05       0.5              0.5
## auc       50        10 0.10       0.5              0.5
```

```
print(out_error$best_obj)
```

```
## [1] 0.114134
```

```
print(out_auc$best_ob)
```

```
## [1] 0.707551
```

Best misclassification rate in validation set is 0.114134, best AUC in validation set is 0.707551.

(c)

```
fit_boost = function(param, eval_metric, dtrain){
  paramlist = list(objective='binary:logistic', eval_metric=eval_metric,
                 max_depth=param$max_depth, eta=param$eta,
                 scale_pos_weight=param$scale_pos_weight,
                 subsample=param$subsample)
  out = xgb.train(paramlist, dtrain, nround=param$nround, verbose=0)
}
boost_error = fit_boost(best_params['error', ], eval_metric = 'error', dtrain = dtrain)
boost_auc = fit_boost(best_params['auc', ], eval_metric = 'auc', dtrain = dtrain)
guess_boost_error = predict(boost_error, dtest)
guess_boost_auc = predict(boost_auc, dtest)
```

```
cat("misclassification error (threshold=0.5) for bal. random forest:",
    mean((guess_bal > 0.5) != test_y))
```

```
## misclassification error (threshold=0.5) for bal. random forest: 0.2817961
```

```
cat("misclassification error (threshold=0.5) for error-tuned boosted tree:",mean((guess_boost_error > 0
```

```
## misclassification error (threshold=0.5) for error-tuned boosted tree: 0.1176731
```

```
cat("misclassification error (threshold=0.5) for auc-tuned boosted tree:",mean((guess_boost_auc > 0.5)
```

```
## misclassification error (threshold=0.5) for auc-tuned boosted tree: 0.1195532
```

```
ROC = roc(test_y, guess_bal, col='green', plot = TRUE, add=F)
cat('auc for random forest: ', ROC$auc, '\n')
```

```
## auc for random forest:  0.6939628
```

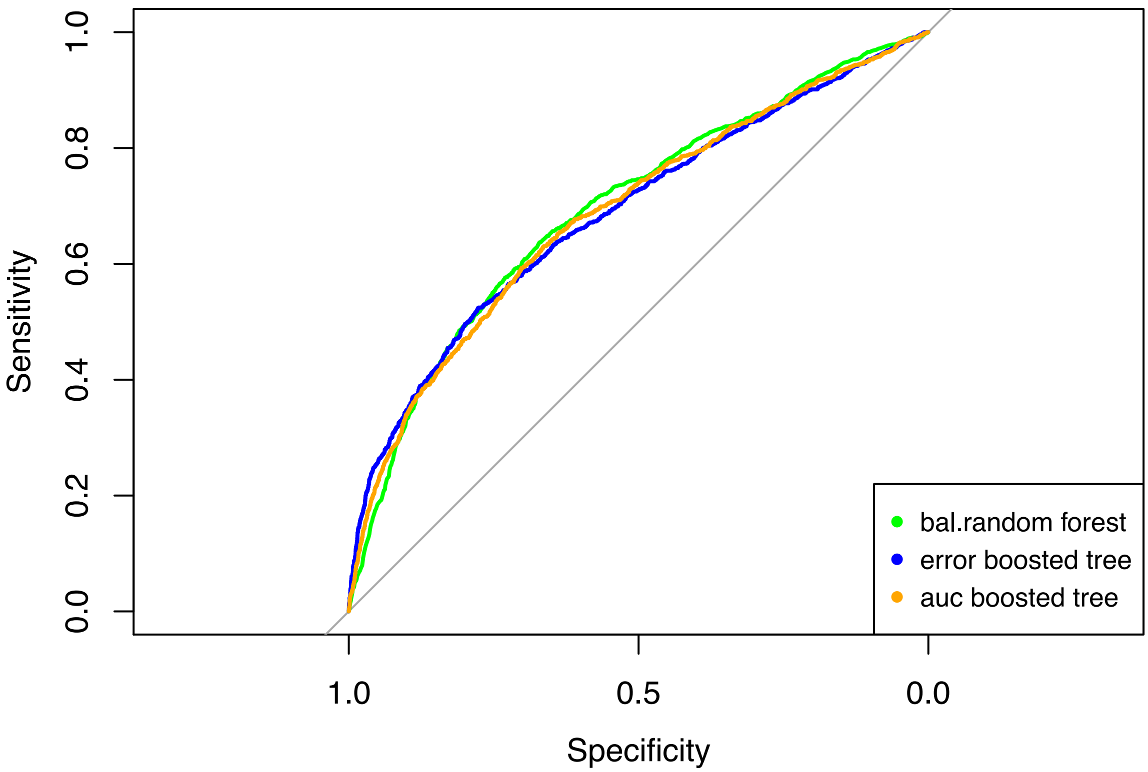
```
ROC = roc(test_y, guess_boost_error, col='blue', plot = TRUE, add=T)
cat('auc for error boosted tree: ', ROC$auc, '\n')
```

```
## auc for error boosted tree:  0.6865643
```

```
ROC = roc(test_y, guess_boost_auc, col = 'orange', plot = TRUE, add=T)
cat('auc for auc booste tree: ', ROC$auc, '\n')
```

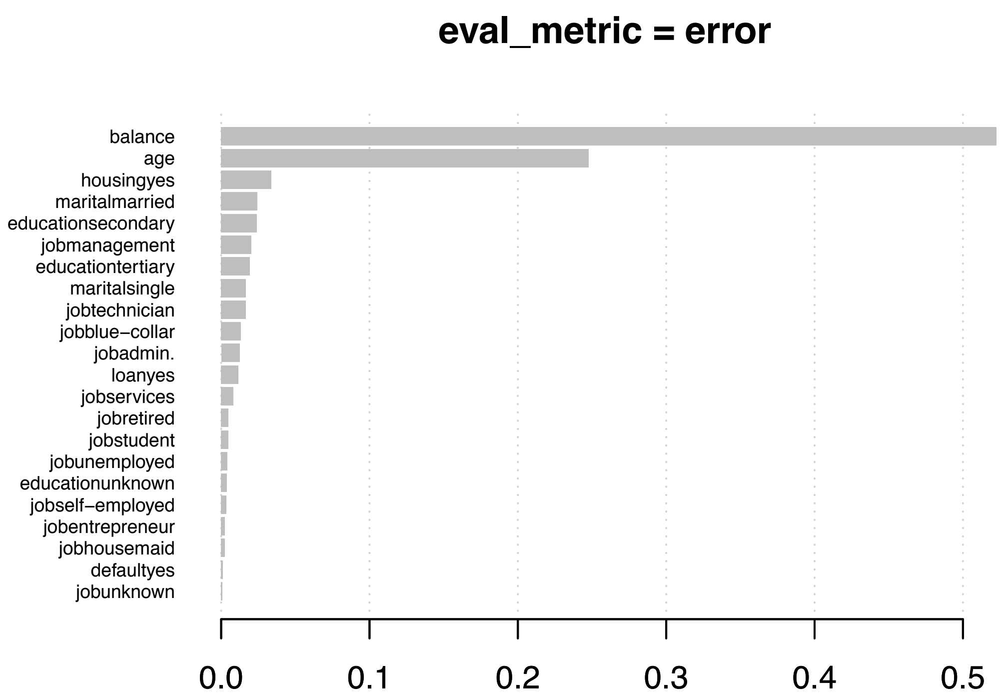
```
## auc for auc booste tree:  0.6869178
```

```
legend("bottomright", legend=c("bal.random forest", "error boosted tree", "auc boosted tree"),pch=c(16,
```

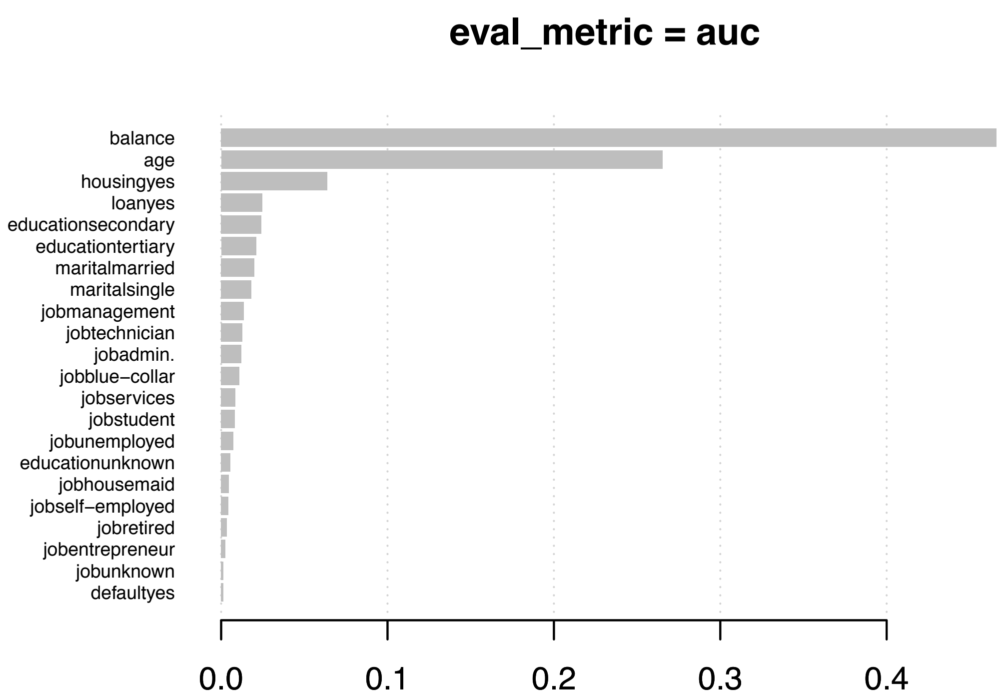


The tuned boosted tree based on error has the lowest misclassification rate while the tuned boosted tree based on auc has the highest AUC in the test set.Compared with random forest model, both of the boosted trees performs better in terms of their eval_metric, but neither of them can beat random forest for both AUC and error.

```
xgb.plot.importance(xgb.importance(colnames(test_expanded), boost_error),main = 'eval_metric = error')
```




```
xgb.plot.importance(xgb.importance(colnames(test_expanded), boost_auc), main = 'eval_metric = auc')
```



3.

(a)

① Show

$\because \|b\beta\| = b\|\beta\|$

multiply both side by b, we have:

$$b y_i (x_i^T \beta + \beta_0) \geq b M (1-\varepsilon_i)\|\beta\|$$

$$\Leftrightarrow \quad y_i (x_i^T \beta b + b\beta_0) \geq M(1-\varepsilon_i)\|b\beta\|$$

② Rewrite

Let $\|\beta\| = \frac{1}{M}$ ~~(Form1)~~

we have: $y_i(x_i^T\beta + \beta_0) \geq 1-\varepsilon_i$

We can rewrite as:

$$\min \|\beta\| \qquad \text{(Form2)}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \varepsilon_i \leq C$$

$$\varepsilon_i \geq 0$$

$$y_i(x_i^T\beta + \beta_0) \geq 1-\varepsilon_i$$

③ Question 1

Information is encoded in the ~~length~~ length of β, i.e. $\|\beta\|$

④ Question 2

$$\beta_{form2} = \beta_{form1} \cdot \frac{1}{M}$$

(b)

① Write

$$\min \|\beta\| + D\sum_{i=1}^{n}\varepsilon_i \qquad \text{(Form 3)}$$

$$\text{s.t.} \quad \varepsilon_i \geq 0$$

$$y_i(x_i^T\beta + \beta_0) \geq 1-\varepsilon_i$$

② Argue

When D is large, large ε_i will be penalized heavily, so $\hat{\varepsilon}_i$ will be small. The lagrangian of Form 2 have similar form as Form 3, so they have ~~equi~~ equivalent solutions.

③ Question

D get smaller, because bigger C means less ~~poor~~ penalty on ε.

(c)

① Rewrite

$$\varepsilon_i \geq 0$$

$$\varepsilon_i \geq 1 - y_i(x_i^T\beta + \beta_0)$$

② Write

$$\varepsilon_i = \max\{0,\ 1-y_i(x_i^T\beta+\beta_0)\} = (1-y_i(x_i^T\beta+\beta_0))^+$$

③ Substitute

$$\min \|\beta\| + D\sum_{i=1}^{n}(1-y_i(x_i^T\beta+\beta_0))^+ \qquad \text{(Form 4)}$$

(d) $\min\limits_{\beta} \frac{\lambda}{2}\|\beta\| + \sum_{i=1}^{n} \left(1 - y_i(x_i^T\beta + \beta_0)\right)^+$ (Form 5)

① Question1

$$\lambda = \frac{2}{D}$$

② Question2

When C increases, D decreases, λ increases, $\|\beta\|$ decreases.