Computer:

1. Local method: kernel smoothing

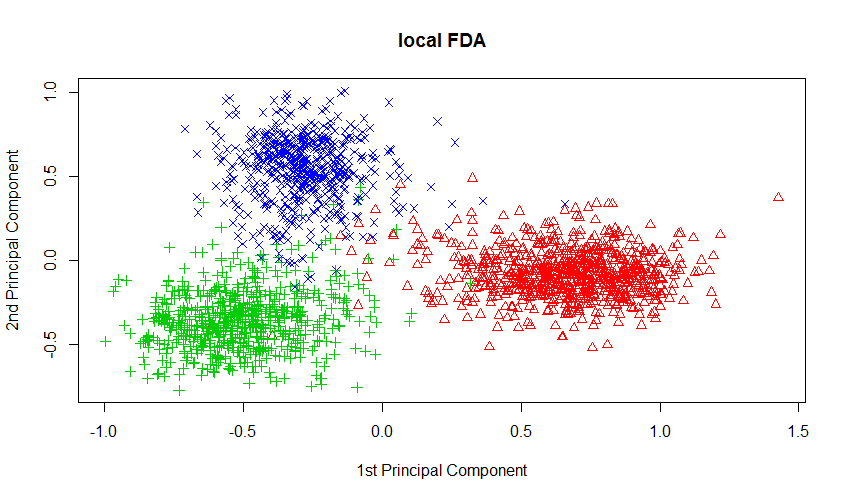
library(glmnet)

## Warning: package 'glmnet' was built under R version 3.2.2

## Warning: package 'foreach' was built under R version 3.2.2

setwd("C:/Users/Chang Yen-hsiu/Desktop/Third Semester/Financial Data Mining/Homework")  
rm(list=ls())  
load("hw3.RData")  
x.train <- rbind(train2,train3,train8)  
mu.hat <- apply(x.train,2,mean)  
x.train <- x.train - rep(1,nrow(x.train))%\*%t(mu.hat) # center data  
x.test <- rbind(test2,test3,test8)  
x.test <- x.test- rep(1,nrow(x.test))%\*%t(mu.hat)  
n2 <- nrow(train2) ;n3 <- nrow(train3) ;n8 <- nrow(train8)  
y.train <- as.factor(c(rep(1,n2),rep(2,n3),rep(3,n8)))  
train <- data.frame(y=y.train,x=x.train) # train data  
N2 <- nrow(test2) ;N3 <- nrow(test3) ;N8 <- nrow(test8)  
y.test <- as.factor(c(rep(1,N2),rep(2,N3),rep(3,N8)))  
test <- data.frame(y=y.test,x=x.test) # test data

library(kernlab)  
library(gplm)  
library(lfda)  
## Tricube Kernel Function  
kernel.function <- function(sigma){  
 rval <- function(x, y = NULL) {  
 if (!is(x, "vector"))   
 stop("x must be a vector")  
 if (!is(y, "vector") && !is.null(y))   
 stop("y must a vector")  
 if (is(x, "vector") && is.null(y)) {  
 return(1)  
 }  
 if (is(x, "vector") && is(y, "vector")) {  
 if (!length(x) == length(y))   
 stop("number of dimension must be the same on both data points")  
 return(((1 - (sqrt(t(x)%\*%y)/lambda)^3))^3\* ((1 -  
 (sqrt(t(x)%\*%y)/lambda)^3)>= 0))  
 }  
 }  
 return(new("rbfkernel", .Data = rval, kpar = list(sigma = sigma)))  
}  
kernel\_1.2 <- kernel.function(1.2)  
kernel.matrix\_1.2 <- kernelMatrix(kernel=kernel\_1.2,x=x.train)  
  
fda.kernel <- klfda(kernel.matrix\_1.2, as.numeric(y.train)-2, 2, metric = c("weighted"))  
# xx <- kmatrixGauss(x.train, sigma = 1)  
# fda.kernel <- klfda(xx, y.train, 2, metric = c("weighted"))  
  
plot(fda.kernel$Z[,1],fda.kernel$Z[,2],col=as.numeric(y.train)+1,  
 pch=as.numeric(y.train)+1,xlab="1st Principal Component",  
 ylab="2nd Principal Component", main="local FDA")



Local LDA:

## Local LDA   
## train error 0.0020564   
## test error 0.0643526

PCA:

## OLS\_PCA Logistic\_PCA LDA\_PCA  
## train error 0.1532885 0.1460383 0.1589850  
## test error 0.2396226 0.2377358 0.2320755

FDA

## OLS\_FDA Logistic\_FDA LDA\_FDA  
## train error 0.02278612 0.02123252 0.02382185  
## test error 0.08490566 0.08490566 0.08867925

The result of local LDA with local FDA is better than others.

1. SVM

library(e1071)

## Warning: package 'e1071' was built under R version 3.2.2

# Support Vector Classifier  
set.seed(1)  
tune.out <- tune(svm, y~.,data=train, kernel="linear",  
 ranges=list(cost=c(0.001, 0.01, 0.1, 1,5,10,100)))  
summary(tune.out)

##   
## Parameter tuning of 'svm':  
##   
## - sampling method: 10-fold cross validation   
##   
## - best parameters:  
## cost  
## 0.01  
##   
## - best performance: 0.02692965   
##   
## - Detailed performance results:  
## cost error dispersion  
## 1 1e-03 0.03106939 0.011447144  
## 2 1e-02 0.02692965 0.009710463  
## 3 1e-01 0.03781315 0.015470606  
## 4 1e+00 0.03729502 0.014830317  
## 5 5e+00 0.03729502 0.014830317  
## 6 1e+01 0.03729502 0.014830317  
## 7 1e+02 0.03729502 0.014830317

# train and test error  
svc.train.err <- mean(predict(tune.out$best.model,train)!=y.train)  
#[1] 0.009321595  
svc.test.err <-mean(predict(tune.out$best.model,test)!=y.test)  
#[1] 0.06226415  
# SVM with the radial basis kernel  
set.seed(1)  
tune.radial <- tune(svm, y~.,data=train, kernel="radial",  
 ranges=list(cost=c(10^(0:3)),  
 gamma=c(10^(-5:-2))))  
summary(tune.radial)

##   
## Parameter tuning of 'svm':  
##   
## - sampling method: 10-fold cross validation   
##   
## - best parameters:  
## cost gamma  
## 10 0.001  
##   
## - best performance: 0.01968111   
##   
## - Detailed performance results:  
## cost gamma error dispersion  
## 1 1 1e-05 0.30038459 0.054457125  
## 2 10 1e-05 0.04350462 0.012028242  
## 3 100 1e-05 0.02951765 0.011467099  
## 4 1000 1e-05 0.02952032 0.011982833  
## 5 1 1e-04 0.04402008 0.011782774  
## 6 10 1e-04 0.02692698 0.010297923  
## 7 100 1e-04 0.02745046 0.010384834  
## 8 1000 1e-04 0.03263447 0.012478609  
## 9 1 1e-03 0.02175365 0.008744960  
## 10 10 1e-03 0.01968111 0.011142903  
## 11 100 1e-03 0.01968378 0.009714501  
## 12 1000 1e-03 0.01968378 0.009714501  
## 13 1 1e-02 0.06473479 0.020183274  
## 14 10 1e-02 0.05851717 0.015255120  
## 15 100 1e-02 0.05851717 0.015255120  
## 16 1000 1e-02 0.05851717 0.015255120

# svm.fit <- svm(y~.,data=train,kernel="radial",cost=10,gamma=0.001)  
# train and test error  
svm.r.train.err <-mean(train[,"y"]!=predict(tune.radial$best.model))  
#[1] 0.001035733  
svm.r.test.err <- mean(test[,"y"]!=predict(tune.radial$best.model,newdata=test))   
#[1] 0.04528302  
# SVM with polynomial kernel  
set.seed(1)  
tune.poly=tune(svm, y~.,data=train, kernel="polynomial",  
 ranges=list(cost=c(10^(0:2)),degree=c(2,3,4)))  
summary(tune.poly)

##   
## Parameter tuning of 'svm':  
##   
## - sampling method: 10-fold cross validation   
##   
## - best parameters:  
## cost degree  
## 10 3  
##   
## - best performance: 0.01760857   
##   
## - Detailed performance results:  
## cost degree error dispersion  
## 1 1 2 0.02796058 0.017805693  
## 2 10 2 0.03054858 0.011289622  
## 3 100 2 0.03106672 0.011171192  
## 4 1 3 0.02226644 0.008817516  
## 5 10 3 0.01760857 0.009832824  
## 6 100 3 0.01760857 0.009832824  
## 7 1 4 0.08442658 0.023599248  
## 8 10 4 0.03468832 0.013791231  
## 9 100 4 0.03365472 0.012723344

# train and test error  
svm.p.train.err <- mean(train[,"y"]!=predict(tune.poly$best.model))   
#[1] 0  
svm.p.test.err <- mean(test[,"y"]!=predict(tune.poly$best.model,newdata=test))  
#[1] 0.03773585  
  
error.table <- rbind(c(svc.train.err,svc.test.err),c(svm.r.train.err,svm.r.test.err),c(svm.p.train.err,svm.p.test.err))  
colnames(error.table) <- c("Train errors","Test errors")  
rownames(error.table) <- c("support vector classifier"," SVM with the radial"," SVM with polynomial ")  
error.table

## Train errors Test errors  
## support vector classifier 0.009321595 0.06226415  
## SVM with the radial 0.001035733 0.04528302  
## SVM with polynomial 0.000000000 0.03773585

Please see tha table.

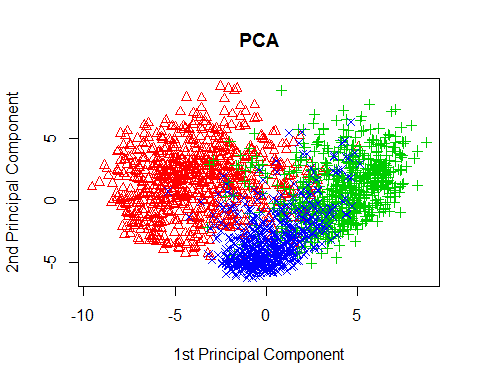
1. Kernel PCA

library(kernlab)

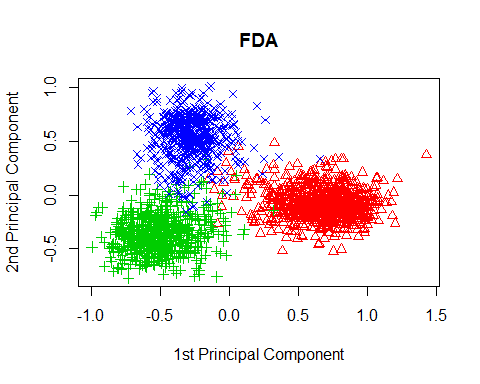
## Warning: package 'kernlab' was built under R version 3.2.2

## Warning in .recacheSubclasses(def@className, def, doSubclasses, env):  
## undefined subclass "externalRefMethod" of class "kfunction"; definition not  
## updated

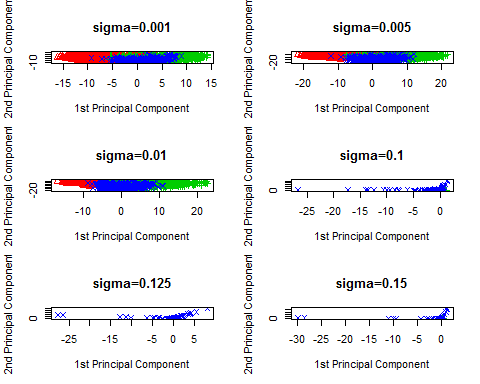
## PCA  
x.svd <- svd(x.train)  
plot(x.train%\*%x.svd$v[,1],x.train%\*%x.svd$v[,2],col=as.numeric(y.train)+1,  
 pch=as.numeric(y.train)+1,xlab="1st Principal Component",  
 ylab="2nd Principal Component",main="PCA")



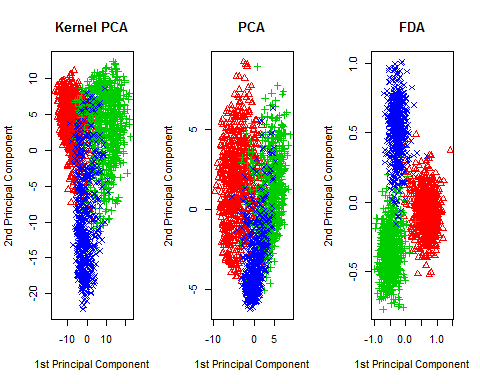
## FDA  
# compute mean estimate: overall and group means  
n <- n2+n3+n8  
mu.hat <- apply(x.train,2,mean)  
mu.1.hat <- apply(x.train[1:n2,],2,mean)  
mu.2.hat <- apply(x.train[(n2+1):(n2+n3),],2,mean)  
mu.3.hat <- apply(x.train[(n2+n3+1):n,],2,mean)  
  
# between class covariance  
S.b <- ((n2)\*(mu.1.hat-mu.hat)%\*%t(mu.1.hat-mu.hat)+  
 (n3)\*(mu.2.hat-mu.hat)%\*%t(mu.2.hat-mu.hat)+  
 (n8)\*(mu.3.hat-mu.hat)%\*%t(mu.3.hat-mu.hat))/(n-1)  
# within class covariance  
S.w <- (t(x.train[1:n2,] - rep(1,n2)%\*% t(mu.1.hat)) %\*% (x.train[1:n2,] - rep(1,n2)%\*% t(mu.1.hat)) +  
 t(x.train[(n2+1):(n2+n3),] - rep(1,n3)%\*% t(mu.2.hat)) %\*% (x.train[(n2+1):(n2+n3),] - rep(1,n3)%\*% t(mu.2.hat)) +  
 t(x.train[(n2+n3+1):n,] - rep(1,n8)%\*% t(mu.3.hat)) %\*% (x.train[(n2+n3+1):n,] - rep(1,n8)%\*% t(mu.3.hat)))/(n-3)  
# total variance  
S.t <- t(x.train - rep(1,n)%\*% t(mu.hat)) %\*% (x.train - rep(1,n)%\*% t(mu.hat))  
  
# define relative matrix  
S <- solve(S.w) %\*% S.b  
# eigen decomp. of S  
S.eig <- eigen(S)  
  
# retain the leading two scores  
fst.score.fda <- x.train %\*% Re(S.eig$vectors[,1])  
snd.score.fda <- x.train %\*% Re(S.eig$vectors[,2])  
  
# Plot FDA two scores  
plot(fst.score.fda,snd.score.fda,col=as.numeric(y.train)+1,  
 pch=as.numeric(y.train)+1,xlab="1st Principal Component",  
 ylab="2nd Principal Component", main="FDA")



## Kernel PCA  
# choice of sigma for the Radial Basis kernel  
par(mfrow=c(3,2))  
sigma.list <- c(0.001,0.005,.01,.1,.125,.15)  
for(sigma in sigma.list){  
 kpc <- kpca(x.train,kernel="rbfdot", kpar=list(sigma=sigma),features=2)  
 plot(rotated(kpc),col=as.numeric(y.train)+1,pch=as.numeric(y.train)+1,  
 xlab="1st Principal Component",ylab="2nd Principal Component",  
 main=paste("sigma=",sigma,sep=""))  
}



par(mfrow=c(1,1))  
# Choose the best parameter with the least overlap, sigma=0.01  
# kernel PCA  
kpc <- kpca(x.train,kernel="rbfdot", kpar=list(sigma=0.01),features=2)  
#plot kernel PC scores  
par(mfrow=c(1,3))  
plot(rotated(kpc),col=as.numeric(y.train)+1,pch=as.numeric(y.train)+1,  
 xlab="1st Principal Component", ylab="2nd Principal Component",  
 main="Kernel PCA")  
plot(x.train%\*%x.svd$v[,1],x.train%\*%x.svd$v[,2],col=as.numeric(y.train)+1,  
 pch=as.numeric(y.train)+1,xlab="1st Principal Component",  
 ylab="2nd Principal Component",main="PCA")  
plot(fst.score.fda,snd.score.fda,col=as.numeric(y.train)+1,  
 pch=as.numeric(y.train)+1,xlab="1st Principal Component",  
 ylab="2nd Principal Component", main="FDA")



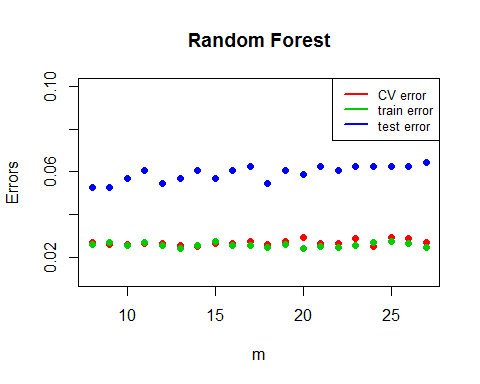
par(mfrow=c(1,1))

1. RF

library(randomForest)

## Warning: package 'randomForest' was built under R version 3.2.2

set.seed(1)  
nfolds <- 5  
n.train <- nrow(train)  
s <- split(sample(n.train),rep(1:nfolds,length=n.train))  
random.cv.err <- rep(NA,20)  
random.train.err <- rep(NA,20)  
random.test.err <- rep(NA,20)  
m <- 8:27  
# CV errors  
for(i in 1:20){  
 random.pred <- rep(NA,n.train)   
 for(j in seq(nfolds)){  
 random.temp <- randomForest(y~.,data=train[-s[[j]],],mtry=m[i]  
 ,importance=TRUE)  
 random.pred[s[[j]]] <- predict(random.temp,newdata=train[s[[j]],])  
 }  
 random.cv.err[i] <- mean(random.pred!=train[,"y"])  
}  
# train and test errors  
for(i in 1:20){  
 random.temp <- randomForest(y~.,data=train,mtry=m[i],importance=TRUE)  
 random.pred.train <- predict(random.temp)  
 random.pred.test <- predict(random.temp,newdata=test)  
 random.train.err[i] <- mean(random.pred.train!=train[,"y"])  
 random.test.err[i] <- mean(random.pred.test!=test[,"y"])  
}  
# Plot  
plot(8:27,random.cv.err,pch=19,col=2,ylim=c(0.01,0.1),ylab="Errors",xlab="m",  
 main="Random Forest")  
points(8:27,random.train.err,pch=19,col=3)  
points(8:27,random.test.err,pch=19,col=4)  
legend("topright",legend=c("CV error","train error","test error"),  
 col=c(2:4),lty=1,lwd=2,cex=.8)



I just plot errors against m from m=8 to m=27, because it takes so long to run 256 m’s.

# training and test errors with the best m  
random.train.err[which.min(random.cv.err)] #[1] 0.02537545

## [1] 0.02537545

random.test.err[which.min(random.cv.err)] #[1] 0.06037736

## [1] 0.06037736

According to CV errors, the best m is 14. Training and test errors with the best m are 0.02537545 and 0.06037736, respectively.

1. Boosting

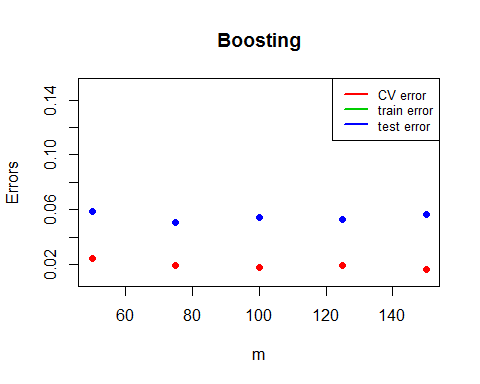
##################### adabag #############################  
library(adabag)

## Warning: package 'adabag' was built under R version 3.2.2

## Warning: package 'mlbench' was built under R version 3.2.2

## Warning: package 'caret' was built under R version 3.2.2

set.seed(1)  
nfolds <- 5  
n.train <- nrow(train)  
s <- split(sample(n.train),rep(1:nfolds,length=n.train))  
boost.cv.err <- rep(NA,5)  
boost.train.err <- rep(NA,5)  
boost.test.err <- rep(NA,5)  
n.trees <- seq(50,150,length=5)  
for(i in 1:5){  
 boost.pred <- rep(NA,n.train)   
 for(j in seq(nfolds)){  
 boost.temp <- boosting(y~.,data=train[-s[[j]],],mfinal=n.trees[i])  
 boost.pred[s[[j]]] <- predict(boost.temp,newdata=train[s[[j]],])$class  
 }  
 boost.cv.err[i] <- mean(boost.pred!=train[,"y"])  
}  
  
# train and test errors  
for(i in 1:5){  
 boost.temp <- boosting(y~.,data=train,mfinal=n.trees[i])  
 boost.pred.train <- predict(boost.temp,newdata=train[,-1])$class  
 boost.pred.test <- predict(boost.temp,newdata=test)$class  
 boost.train.err[i] <- mean(boost.pred.train!=train[,"y"])  
 boost.test.err[i] <- mean(boost.pred.test!=test[,"y"])  
}  
  
# Plot  
plot(n.trees,boost.cv.err,pch=19,col=2,ylim=c(0.01,0.15),ylab="Errors",  
 xlab="m",main="Boosting")  
points(n.trees,boost.train.err,pch=19,col=3)  
points(n.trees,boost.test.err,pch=19,col=4)  
legend("topright",legend=c("CV error","train error","test error"),  
 col=c(2:4),lty=1,lwd=2,cex=.8)



Training errors are all zero in the plot above. And I used Adaboost rather than GBM here.

# training and test errors with the best number of trees  
n.trees[which.min(boost.cv.err)]

## [1] 150

boost.train.err[which.min(boost.cv.err)] # [1] 0

## [1] 0

boost.test.err[which.min(boost.cv.err)] # [1] 0.05660377

## [1] 0.05660377

According to CV errors, the best number of trees is 150. Training and test errors with the best number of trees are 0 and 0.05660377, respectively. In this case, adaboost could be better than random forest. But, the speed of adaboost is very slow. Moreover, because the adaboost is very slow, I just tried several number of trees. Maybe the result will become better if we further increase the number of trees.