Overview

A. Ian McLeod, aimcleod@uwo.ca

2016 Western Science

Purpose

Prediction always involves a tradeoff between “signal and noise” in the sense that if we allow our model to overfit the data we are including too much of the noise in the model part. This overfit model may appear to have good prediction performance on the training data but if fails to generalize so that the prediction performance on out-of-sample or test data does not match the performance on the training data. Cross-validation can give us an estimate of what the test data error is without actually using new fresh data. This magic is similar to that of bootstrapping where we can estimate the standard deviations of parameter estimates by re-sampling the training data. The most important pitfalls to the validity of cross-validation are incorrect model specification and/or selection bias due to some pre-processing or data snooping.

Other CRAN packages that provide general frameworks with resampling strategies include boot, mlr and caret. Both mir and caret are also very general and very complex. But our package gencve is easier to use and extensible to other prediction methods in R. It is planned to extend this package to include k-fold cross-validation and estimating ROC curves by cross-validation.

Suggested applications

Empirical comparisons of prediction algorithms
For example CART and C5.0 are two similar decision tree approaches. One could investigate the performance of each algorithm for a particular dataset or a range of datasets to compare the overall performance.

Choosing the best algorithm for prediction
A forecaster may use gcv() or cgcv() to find the best prediction algorithm for their particular problem.

References

Trevor Hastie, Robert Tibshirani, Jerome H. Friedman (2009), The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Ed. Springer.

Shao, Jun (1993), Linear Model Selection by Cross-validation Journal of the American Statistical Association,Vol. 88, Iss. 422, 1993.

Kim, J. H. (2009), Estimating Classification Error Rate: Repeated Cross-validation, Repeated Hold-out and Bootstrap. Computational Statistics and Data Analysis, 53, 3735-3745.

Main functions

- gcv() for regression
- cgcv() for classification

gcv()

gcv() is used for cross-validation with regression functions. In the example below we compare backward stepwise regression and LASSO for the prostate dataset. The output from running the script below follows after the script.

```
#Source: prostate_stepVSlasso.R
#compare backward stagewise regression with LASSO for prostate data
#d chosen to approximate iterated 10-fold cv
#
X <- prostate[,1:8]
y <- prostate[,9]
n <- length(y)
system.time(ans <- gcv(X, y, MaxIter=10^3, d=n/10, yhat=yhat_step, NCores=8))
ans
system.time(ans <- gcv(X, y, MaxIter=10^3, d=n/10, yhat=yhat_lars, NCores=8))
ans
```

The output from gcv is a vector with three components: EPE (expected prediction error, usually squared error), its estimated standard deviation and the average correlation between predictions and the test values.

```
> system.time(ans <- gcv(X, y, MaxIter=10^3, d=n/10, yhat=yhat_step, NCores=8))
   user  system elapsed
   0.03    0.33   15.26
> ans
           epe      sd_epe     pcorr
[1,] 0.5530234 0.008035521 0.7648303
> system.time(ans <- gcv(X, y, MaxIter=10^3, d=n/10, yhat=yhat_lars, NCores=8))
   user  system elapsed
   0.03    0.03    6.24
> ans
           epe      sd_epe    pcorr
[1,] 0.5630057 0.007547835 0.759917
```

We see the difference is very small and it is not statistically signficant at 5%. As might be expected lars() is much faster.

cgcv()

We use the rmix() function to generate a random sample with $n = 400$. There are 200 observations in each class with $p = 2$ covariates. The function rmix() generates sample from for each class from different mixture distributions so that the underlying Bayes error rate is about 20.76%. The cross-validation results are in good agreement.

```
> library("gencve")
> n <- 400
> Xy <- rmix(n)
> X <- Xy[,1:2]
> y <- Xy[,3]
> #
> cgcv(X, y, yh=yh_RF, NCores=8, d=n/5)
     cost     pcorr
0.1942125 0.6126685
> cgcv(X, y, yh=yh_kNN, NCores=8, d=n/5, method="LOOCV")
     cost     pcorr
0.2006750 0.6014624
```

Regression

The regression cv function gcv() takes a function argument, yhat. The first two arguments of this function are dataframes corresponding to training and test data. The last column in each dataset must be the response variable. The output are the predictions for the test data.

The built-in functions discussed next.

yhat_lm()

```
> yhat_lm
function(dfTrain, dfTest) {
  ans <- lm(y~., data=dfTrain)
  predict(ans, newdata=dfTest)
}
```

This simply fits a regression to the full training set and then predicts on the test data.

yhat_step()

```
> yhat_step
function(dfTrain, dfTest, ic=c("BIC","AIC")) {
    ansFull <- lm(y~., data=dfTrain)
    ic <- match.arg(ic)
    k <- ifelse(identical(ic, "AIC"), 2, log(nrow(dfTrain)))
    junk <- capture.output(ansStep <- step(ansFull, k=k))
    predict(ansStep, newdata=dfTest)
  }
```

Backward stagewise regression is used. By default the BIC is used, setting ic=”AIC” to use AIC instead.

yhat_nn()

```
> yhat_nn
function(dfTrain, dfTest, normalize=TRUE){
  nte <- nrow(dfTest)
  ntr <- nrow(dfTrain)
  p <- ncol(dfTrain)-1
  Xtr <- as.matrix.data.frame(dfTrain)
  ytr <- Xtr[,p+1]
  Xtr <- Xtr[,1:p]
  Xte <- as.matrix.data.frame(dfTest)
  Xte <- Xte[,1:p]
  if (normalize) {#rescale training. Then rescale test with same!!
    Xtr <- scale(Xtr)
    a <- attr(Xtr, "scaled:center")
    b <- attr(Xtr, "scaled:scale")
    Xte <- sweep(Xte, 2, a)
    Xte <- sweep(Xte, 2, b, FUN="/")
  }
  yHat <- numeric(nte)
  for (i in 1:nte) {
    xi <- Xte[i,]
    edist <- rowSums((Xtr-matrix(xi, byrow=TRUE, ncol=p, nrow=ntr))^2)
    yHat[i] <- ytr[which.min(edist)]
  }
yHat
}
```

The code is more complicated but the method is simply the nearest neighbor regression. For any test input variables, its nearest neighbor in the training data is found and the corresponding value of the response variable is used as the prediction. By default, the inputs are normalized using R's scale() function.

yhat_lars()

```
> yhat_lars
function(dfTrain, dfTest, normalize=TRUE){
  Xtr <- as.matrix.data.frame(dfTrain)
  ytr <- Xtr[,ncol(Xtr)]
  Xtr <- Xtr[,-ncol(Xtr)]
  ans <- lars(Xtr, ytr, type="lasso", normalize=normalize)
  iopt <- which.min(summary(ans)$Cp) #best Cp
  Xte <- as.matrix.data.frame(dfTest)
  yte <- Xte[,ncol(Xte)]
  Xte <- Xte[,-ncol(Xte)]
  predict(ans, newx=Xte, s=iopt)$fit
}
```

The lars() function from the package lars is used to fit LASSO regression. The regularization parameter is chosen using Mallows' Cp.

yhat_gel()

The tuning parameter is selected using regularized k-fold CV.

```
> yhat_gel
function(dfTrain, dfTest, alpha=1) {
#Gaussian, elastic net
  n <- nrow(dfTrain)
  n <- nrow(dfTrain)
  Xtr <- as.matrix.data.frame(dfTrain) #assumes Gaussian!
  Xte <- as.matrix.data.frame(dfTest)
  p <- ncol(Xtr)-1
  ytr <- Xtr[, p+1]
  Xtr <- Xtr[,-(p+1)]
  Xte <- Xte[,-(p+1)]
  ans<- glmnet(x=Xtr, y=ytr, alpha=alpha)
  ans_cv <- cv.glmnet(Xtr, y=ytr, alpha=alpha)
  lambdaHat <- ans_cv$lambda.1se
  predict(ans, newx=Xte, s=lambdaHat)[,1]
}
```

The elastic net availble in the glmnet package is used. The argument alpha=1 specifies LASSO penalty. Ridge regression corresponds to alpha=0 although sometimes there appear to be numerical problems and alpha=0.02 approximates RR more closely. The following example compares RR with plain regression for the prostate data. By default, the cross-validation used 1000 replications. We see that it took only 36 seconds. It appears that for this data OLS works better using mean-square error loss.

```
> library("gencve")
> X <- prostate[,1:8]
> y <- prostate[,9]
> startTime <- proc.time()[3]
> gcv(X, y, NCores=8, yhat=yhat_lm)
           epe     sd_epe     pcorr
[1,] 0.5608039 0.00790333 0.7610035
> gcv(X, y, NCores=8, yhat=yhat_gel, alpha=0.02)
           epe      sd_epe     pcorr
[1,] 0.6360201 0.008817766 0.7229646
> proc.time()[3] - startTime #startTime #total time
elapsed
  36.48
```

yhat_plus()

The plus package on CRAN provides non-convex penalties SCAD and MCP. Both of these penalized regression methods have the oracle property.

```
> yhat_plus
function(dfTrain, dfTest, normalize=TRUE, ic=c("BIC","AIC"),
                         method=c("scad", "mc+", "lasso")){
  method <- match.arg(method)
  ic <- match.arg(ic)
  Xtr <- as.matrix.data.frame(dfTrain)
  ytr <- Xtr[,ncol(Xtr)]
  Xtr <- Xtr[,-ncol(Xtr)]
  ans <- plus(Xtr, ytr, method=method, normalize=normalize)
  n <- nrow(Xtr)
  Dev <- n*log((1-ans$r.square)/n)
  k <- ifelse(identical(ic, "AIC"), 2, log(n))
  ICs <- Dev + k*ans$dim
  lamIC <- ans$lam[which.min(ICs)]
  Xte <- as.matrix.data.frame(dfTest)
  yte <- Xte[,ncol(Xte)]
  Xte <- Xte[,-ncol(Xte)]
  capture.output(
    out<-(predict(ans, newx=Xte, lam=ans$lam)$newy)[which.min(ICs),])
  out
}
```

We use the plus package with BIC to select the tuning parameter and compare the three penalties: LASSO, SCAD and MCP using 10^4 replications. It takes only 92 seconds. It appears that LASSO works the best with prostate using BIC and the plus implementation.

```
> library("gencve")
> X <- prostate[,1:8]
> y <- prostate[,9]
> MXIT <- 10^4
> startTime <- proc.time()[3]
> gcv(X, y, NCores=8, MaxIter = MXIT, yhat=yhat_plus, ic="BIC", method="lasso")
           epe      sd_epe     pcorr
[1,] 0.5666443 0.002604045 0.7581182
> gcv(X, y, NCores=8, MaxIter = MXIT, yhat=yhat_plus, ic="BIC", method="scad")
           epe      sd_epe     pcorr
[1,] 0.6019023 0.003149085 0.7404611
> gcv(X, y, NCores=8, MaxIter = MXIT, yhat=yhat_plus, ic="BIC", method="mc+")
           epe      sd_epe     pcorr
[1,] 0.6145634 0.003258138 0.7340169
> proc.time()[3] - startTime #startTime #total time
elapsed
  92.04
```

yhat_RF()

Random forest regression is implemented.

```
> yhat_RF
function(dfTrain, dfTest) {
  Xtr <- as.matrix.data.frame(dfTrain) #assumes Gaussian!
  Xte <- as.matrix.data.frame(dfTest)
  p <- ncol(Xtr)-1
  ytr <- Xtr[, p+1]
  Xtr <- Xtr[,-(p+1)]
  Xte <- Xte[,-(p+1)]
  ans <- randomForest::randomForest(x=Xtr, y=ytr)
  randomForest:::predict.randomForest(ans, newdata=Xte)
}
```

yhat_SVM()

Support vector machine regression is implemented.

```
> yhat_SVM
function(dfTrain, dfTest) {
  Xtr <- as.matrix.data.frame(dfTrain)
  Xte <- as.matrix.data.frame(dfTest)
  p <- ncol(Xtr)-1
  ytr <- Xtr[, p+1]
  Xtr <- Xtr[,-(p+1)]
  Xte <- Xte[,-(p+1)]
  ans <- svm(x=Xtr, y=ytr)
  predict(ans, newdata=Xte)
}
```

OLS, RR, SVM and RF are compared for the pollution dataset. The rankings are 1:RF, 2:SVM, 3:RR and 4:OLS.

```
> library("gencve")
> X <- pollution[,1:15]
> y <- pollution[,16]
> startTime <- proc.time()[3]
> gcv(X, y, NCores=8, yhat=yhat_lm)
          epe   sd_epe     pcorr
[1,] 2291.873 60.1043 0.6385349
> gcv(X, y, NCores=8, yhat=yhat_gel, alpha=0.02)
          epe    sd_epe     pcorr
[1,] 2131.023 43.34461 0.6702941
> gcv(X, y, NCores=8, yhat=yhat_RF)
          epe    sd_epe     pcorr
[1,] 1869.073 39.85663 0.7190189
> gcv(X, y, NCores=8, yhat=yhat_SVM)
         epe    sd_epe     pcorr
[1,] 2001.37 37.70748 0.6948377
> proc.time()[3] - startTime #startTime #total time
elapsed
  60.26
```

Classification

The regression cv function cgcv() takes a function argument, yh. The first two arguments of this function are dataframes corresponding to training and test data. The last column in each dataset must be a factor variable, the class. The function uses the built-in cost function, misclassification-rate() and returns this value.

The built-in functions discussed next.

yh_lda()

Linear discriminant analysis, LDA, we use the lda() function in MASS.

```
> yh_lda
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- dfTr[,p+1]
  Xte <- dfTe[,1:p]
  yte <- dfTe[,p+1]
  ans <- MASS::lda(x=X, grouping=y)
  yh <- MASS:::predict.lda(ans, newdata=Xte)$class
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=cor(as.numeric(yte), as.numeric(yh))))
}
```

We will compare LDA, QDA and logistic below.

yh_qda()

Quadratic discriminant analysis using MASS::qda().

```
> yh_qda
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- dfTr[,p+1]
  Xte <- dfTe[,1:p]
  yte <- dfTe[,p+1]
  ans <- MASS::qda(x=X, grouping=y)
  yh <- MASS:::predict.qda(ans, newdata=Xte)$class
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=cor(as.numeric(yte), as.numeric(yh))))
}
```

yh_logistic()

Logistic regression classification using glm() and glmnet(). When the argument alpha=NULL, no penalty is used and the R's glm() is used for fitting. For 0<=alpha<=1, elastic net penalty with glmnet is used. The tuning parameter is selected using regularized k-fold CV.

```
> yh_logistic
function(dfTr, dfTe, alpha=NULL){
  fixupFactor <- function(u) factor(as.character(u)) #needed, eg: churn etc
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- fixupFactor(dfTr[,p+1])
  stopifnot(length(levels(y))==2)
  Xte <- dfTe[,1:p]
  yte <-  fixupFactor(dfTe[,p+1])
  if (is.null(alpha)) {#glm
    ans<- glm(y ~., data=X, family="binomial")
    prob <- ifelse(predict.glm(ans, newdata=Xte, type="response")<0.5, 1, 2)
    yh <- levels(y)[prob]
  } else {#elastic net
    stopifnot(alpha<=1 || alpha>=0)
    X <- as.matrix.data.frame(X)
    Xte <- as.matrix.data.frame(Xte)
    ans<- glmnet(x=X, y=y, family="binomial", alpha=alpha)
    ans_cv <- cv.glmnet(X, y=y, family="binomial", alpha=alpha)
    lambdaHat <- ans_cv$lambda.1se
    yh <- predict(ans, newx=Xte, type="class", s=lambdaHat)[,1]
  }
  yh <- factor(yh)
  unlist(list(cost=misclassificationrate(yte, yh),
                     pcor=cor(as.numeric(yte), as.numeric(yh))))
}
```

Compare the test prediction using logistic regression, LDA and QDA for the best 10 genes in the Singh microarray data. Recall that for this dataset we are providing with separate training and test datasets.

```
> library("gencve")
> #select 10 best genes in training set and corresponding genes in test
> yh_lda(SinghTrain[,c(1:10, 101)], SinghTest[,c(1:10, 101)])
 cost pcorr
    0     1
> yh_qda(SinghTrain[,c(1:10, 101)], SinghTest[,c(1:10, 101)])
      cost      pcorr
0.05882353 0.84888889
> yh_logistic(SinghTrain[,c(1:10, 101)], SinghTest[,c(1:10, 101)])
      cost       pcor
0.02941176 0.92951600
```

Compare logistic, regularized logistic regression with LDA using the churnTest and churnTrain dataset. We remove the state labels and replace the factor “area_code” with two dummy indicator variables.

```
> yh_lda(dfTr, dfTe)
     cost     pcorr
0.1349730 0.2828066
> yh_logistic(dfTr, dfTe)
     cost      pcor
0.1283743 0.2737595
> yh_logistic(dfTr, dfTe, alpha=1)
     cost      pcor
0.1265747 0.2367406
```

Complete R script for this example

```
#Source: churn_LDA_logistic_LASSOLogistic.R
#
library("gencve")
dim(churnTrain) #is 3333-by-20.
#put into right format: dfTr
ytr <- churnTrain[, 20]
Xtr <- churnTrain[, 2:19]
Xtr <- Xtr[, -2] #remove area_code and convert to indicators
areacode<-churnTrain[,"area_code"]
m <- model.matrix(~areacode)[,2:3]
Xtr <- cbind(Xtr, m)
#replace international_plan with 0/1 encoding
Xtr[,"international_plan"] <- as.numeric(Xtr[,"international_plan"])-1
#replace voice_mail_plan with 0/1 encoding
Xtr[,"voice_mail_plan"] <- as.numeric(Xtr[,"voice_mail_plan"])-1
dfTr <- cbind(Xtr, ytr)
#put into right format: dfTe
yte <- churnTest[, 20]
```

```
Xte <- churnTest[, 2:19]
Xte <- Xte[, -2] #remove area_code and convert to indicators
areacode<-churnTest[,"area_code"]
m <- model.matrix(~areacode)[,2:3]
Xte <- cbind(Xte, m)
#replace international_plan with 0/1 encoding
Xte[,"international_plan"] <- as.numeric(Xte[,"international_plan"])-1
#replace voice_mail_plan with 0/1 encoding
Xte[,"voice_mail_plan"] <- as.numeric(Xte[,"voice_mail_plan"])-1
dfTe <- cbind(Xte, yte)
#
yh_lda(dfTr, dfTe)
yh_logistic(dfTr, dfTe)
yh_logistic(dfTr, dfTe, alpha=1)
```

yh_kNN()

The yh_kNN() implement kNN with three methods for selecting k. The default is LOOCV. Another choice is MLE or with “NN” the value of k may be selected. In the former case, with method=”NN”, the default is k=1 since often this works surprisingly well in some cases.

```
> yh_kNN
function(dfTr, dfTe,
                   method=c("LOOCV", "MLE", "NN"), k=1) {
  method<- match.arg(method)
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- dfTr[,p+1]
  Xte <- dfTe[,1:p]
  yte <- dfTe[,p+1]
  kOpt <- switch(method,
            LOOCV = kNN_LOOCV(X, y),
            MLE = kNN_MLE(X, y),
            NN = k)
  yh <- class::knn(X, Xte, y, k=kOpt)
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=cor(as.numeric(yte), as.numeric(yh))))
}
```

Comparing random forest (RF) and kNN for the random mixture example with $n = 400$, we find kNN with LOOCV is very accurate since it is not significantly different from the optimal error rate of 20.76%. The RF does not work as well in this example. kNN with LOOCV required 282 seconds.

```
> n <- 400
> Xy <- rmix(n)
> X <- Xy[,1:2]
> y <- Xy[,3]
> set.seed(373743)
> #
> startTime <- proc.time()[3]
> cgcv(X, y, yh=yh_RF, NCores=8, d=n/5)
     cost     pcorr
0.2474625 0.5073852
```

```
> proc.time()[3]-startTime #total time
elapsed
  32.12
>
> startTime <- proc.time()[3]
> cgcv(X, y, yh=yh_kNN, NCores=8, d=n/5, method="LOOCV")
     cost     pcorr
0.2055125 0.5932820
> proc.time()[3]-startTime #total time
elapsed
  282.5
```

The complete script for this example.

```
#Source: rmix_RF_kNN7.R
#
library("gencve")
n <- 400
Xy <- rmix(n)
X <- Xy[,1:2]
y <- Xy[,3]
set.seed(373743)
#
startTime <- proc.time()[3]
cgcv(X, y, yh=yh_RF, NCores=8, d=n/5)
proc.time()[3]-startTime #total time

startTime <- proc.time()[3]
cgcv(X, y, yh=yh_kNN, NCores=8, d=n/5, method="LOOCV")
proc.time()[3]-startTime #total time
```

yh_CART()

yh_CART() implements classification using the rpart() function. We will compare this method with C50 and RF for the BFOS random digits and the xor problem.

```
> yh_CART
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- factor(dfTr[,p+1])
  Xte <- dfTe[,1:p]
  yte <- factor(dfTe[,p+1])
  ans<- rpart::rpart(y ~., data=X)
  yh <- rpart:::predict.rpart(ans, Xte, type="class")
  YTE <- as.numeric(yte)
  YH <-  as.numeric(yh)
  if (var(YTE)*var(YH) > 0) {
    pcorr <- cor(YTE, YH)
  } else {
    pcorr <- NA
  }
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=pcorr))
}
```

yh_C50()

Another tree classifier C5.0 is provided.

```
> yh_C50
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- factor(dfTr[,p+1])
  Xte <- dfTe[,1:p]
  yte <- factor(dfTe[,p+1])
  ans<- C50::C5.0(y ~., data=X)
  yh <- C50::predict.C5.0(ans, Xte, type="class")
  YTE <- as.numeric(yte)
  YH <-  as.numeric(yh)
  if (var(YTE)*var(YH) > 0) {
    pcorr <- cor(YTE, YH)
  } else {
    pcorr <- NA
  }
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=pcorr))
}
```

yh_RF()

The random forest (RF) uses bootstrap aggregating or bagging of full grown trees on bootstrap samples.

```
> yh_RF
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- factor(dfTr[,p+1])
  Xte <- dfTe[,1:p]
  yte <- factor(dfTe[,p+1])
  ans<- randomForest::randomForest(y ~., data=X)
  yh <- randomForest:::predict.randomForest(ans, Xte, type="response")
  YTE <- as.numeric(yte)
  YH <-  as.numeric(yh)
  if (var(YTE)*var(YH) > 0) {
    pcorr <- cor(YTE, YH)
  } else {
    pcorr <- NA
  }
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=pcorr))
}
```

We compare RF, CART and C5.0 for the random digits problem with Bayes error rate about 26%.

Training and test samples of size 10^4 are used. Both C5.0 and RF are accurate and have converged to the optimal rate. The entire script took about 6 seconds.

```
> XyTr <- rdigitsBFOS(n, alpha=0.1)
Electronic Digit Recognition Problem, alpha = 0.1 BayesRate = 0.25994
> dim(XyTr)
[1] 10000     8
> XyTe <- rdigitsBFOS(n, alpha=0.1)
Electronic Digit Recognition Problem, alpha = 0.1 BayesRate = 0.25994
> startTime <- proc.time()[3]
> yh_CART(XyTr, XyTe)
     cost     pcorr
0.3002000 0.6409184
> yh_C50(XyTr, XyTe)
     cost     pcorr
0.2544000 0.6657022
> yh_RF(XyTr, XyTe)
     cost     pcorr
0.2548000 0.6531856
> proc.time()[3]-startTime #total time
elapsed
   5.64
```

The complete script for this example is given below.

```
#Source: rdigitsBFOS_CART_C50_RF.R
#
library("gencve")
n <- 1000
set.seed(17744331)
XyTr <- rdigitsBFOS(n, alpha=0.1)
dim(XyTr)
XyTe <- rdigitsBFOS(n, alpha=0.1)
startTime <- proc.time()[3]
yh_CART(XyTr, XyTe)
yh_C50(XyTr, XyTe)
yh_RF(XyTr, XyTe)
proc.time()[3]-startTime #total time
```

yh_svm()

The SVM (support vector machine) classifier is widely used for high dimensional classification problems. In the next section we compare with Naive Bayes.

```
> yh_svm
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- factor(dfTr[,p+1])
  Xte <- dfTe[,1:p]
  yte <- factor(dfTe[,p+1])
  ans <- svm(x=X, y=y)
  yh <- predict(ans, newdata=Xte)#predict.svm not exported!!
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=cor(as.numeric(yte), as.numeric(yh))))
}
```

yh_NB()

The Naive Bayes classifier often sometimes gives good predictions with high dimensional problems.

```
> yh_NB
function(dfTr, dfTe){
  p <- ncol(dfTr)-1
  X <- dfTr[,1:p]
  y <- factor(dfTr[,p+1])
  Xte <- dfTe[,1:p]
  yte <- factor(dfTe[,p+1])
  ans<- e1071::naiveBayes(x=X, y=y)
  yh <- e1071:::predict.naiveBayes(ans, Xte, type="class")
  YTE <- as.numeric(yte)
  YH <-  as.numeric(yh)
  if (var(YTE)*var(YH) > 0) {
    pcorr <- cor(YTE, YH)
  } else {
    pcorr <- NA
  }
  unlist(list(cost=misclassificationrate(yte, yh),
              pcorr=pcorr))
}
```

We compare the Naive Bayes and SVM classifiers for the Singh microarray datasets using $p = 100$ genes. Both methods produced perfect results on the test data. The total time is less than 1 second.

```
> yh_RF(SinghTrain, SinghTest)
 cost pcorr
    0     1
> yh_RF(SinghTrain, SinghTest)
 cost pcorr
    0     1
```

```
#Source: Singh_RF_NaiveBayes.R
#
library("gencve")
startTime <- proc.time()[3]
yh_RF(SinghTrain, SinghTest)
yh_RF(SinghTrain, SinghTest)
proc.time()[3]-startTime #total time
```

General interest utility functions

- xvif(): variance inflation factors computed directly from the design matrix.
- dShao(): recommended test data sample size used by Shao (1993)

- featureSelect(): a simple and commonly used feature selection method that works well in some prediction problems but suffers from selection bias so statistical inferences are not valid.

Built-in cost functions

For regression we have mae(), mape(), smape() and mse(). For classification misclassificationrate() is provided.

Data sets included

```
> data(package="gencve")
Data sets in package ‘gencve’:

Detroit                        Detroit Homicide Data for 1961-73
SinghTest                      Singh Prostate Microarray Test Data
SinghTrain                     Singh Prostate Microarray Training Data
churnTest (churn)              Customer Churn Data
churnTrain (churn)             Customer Churn Data
fires                          Forest Fires in Montesinho Natural Park
kyphosis                       Data on Children who have had Corrective Spinal Surgery
meatspec                       Meat Spectrometry to Determine Fat Content
pollution                      Pollution Data from McDonald and Schwing
prostate                       Prostate Cancer Data
```

Data generation models (DGM)

- rmix(): generates classification data from a mixture of 20 normal distributions with Bayes error rate 20.8% Derivation of the Bayes rate is discussed in a separate vignette.
- ridigitsBFOS(): random digits corrupted by noise as suggested in BFOS book and specified Bayes error rate. Derivation of the Bayes rate is discussed in a separate vignette.
- rxor(): famous xor problem with added noise with Bayes error rate zero.
- ShaoReg(): regression model used in cross-validation experiments reported by Shao (1993). More details in a separate vignette.

Extensions

You can extend the package by writing your own functions to use with gcv() or cgcv(). We should a few examples of how this can be done.

Extending gcv()

The argument yhat in gcv() specifes a function with two arguments. The first argument specifies the training data and the second argument specifies the test data. Each argument must be a data.frame with the last column the response or output variable. The function must then fit to the train data and then predict using the test data. The output must be the vector of predictions.

Some examples of such functions that are provided already in the package are given in the overview vignette. Some additional examples are provided below along with their usage with gcv()

grpreg::grpreg()

The grpreg package on CRAN provides group LASSO and other penalized regression methods. By setting its argument group so all inputs are in group 1, it may be applied to the regular ungrouped case. This is what we have implemented in yhat_reg() below. Arguments ic and penalty are provided.

```
yhat_reg <- function(dfTrain, dfTest, ic=c("BIC","AIC"),
                          penalty = c("lasso", "ridge", "mcp", "scad")) {
  ic <- match.arg(ic)
  penalty <- match.arg(penalty)
  if (identical(penalty, "ridge")) {
    penalty2 <- "grLasso"
    alpha <- 0.025
  } else {
    alpha <- 1
    penalty2 <- switch(penalty,
                       lasso = "grLasso",
                       mcp = "grMCP",
                       scad = "grSCAD")
  }
  ic <- "AIC"
  n <- nrow(dfTrain)
  Xtr <- as.matrix.data.frame(dfTrain) #assumes Gaussian!
  Xte <- as.matrix.data.frame(dfTest)
  p <- ncol(Xtr)-1
  ytr <- Xtr[, p+1]
  Xtr <- Xtr[,-(p+1)]
  Xte <- Xte[,-(p+1)]
  ans <- grpreg::grpreg(Xtr, ytr, alpha=alpha, group=rep(1, p), penalty=penalty2)
  lambdaOpt <- grpreg::select(ans, ic)$lambda
  predict(ans, X=Xte, type="response", lambda=lambdaOpt)
}
```

We compare EPE grpreg() using SCAD, MCP and LASSO with the prostate data.

Principal component regression and partial least squares

```
yhat_PCR <-
function(dfTrain, dfTest, ncomp="default", scaleQ=TRUE,
                        ic=c("BIC","AIC"), alg=c("PCR", "PLSR")) {
  ic <- match.arg(ic)
  alg <- match.arg(alg)
  if (!is.numeric(ncomp)) {
    ncomp <- ncol(dfTrain)-1 #assume n>p
  }
  if (identical(alg, "PCR")) {
    fitfun <- pls::pcr
  } else {
    fitfun <- pls::plsr
  }
  ans <- fitfun(y ~ .,  data = dfTrain, ncomp=ncomp , scale=scaleQ)
  res<-resid(ans)[,1,]
  ntr <- nrow(dfTrain)
  LL <- apply(res, MARGIN=2, FUN=function(x) -(ntr/2)*log(sum(x^2)/ntr))
  k <- ifelse(identical(ic, "AIC"), 2, log(ntr))
  iIC <- which.min(-2*LL + k*(1:ncomp))
  ansIC <- pcr(y~., data=dfTrain, ncomp=iIC)
  predict(ansIC, dfTest)[1:nrow(dfTest),1,iIC]
```

Comparison with boot::cv.glm()

boot::cv.glm() uses k-fold cross-validation on glm objects. In our comparison we use $k = 10$. Notice that the result from a single iteration of k-fold CV is quite variable unlike the result using delete-d with $d = n/10$. Iterating cv.glm() 500 times and averaging, the result agrees with gcv().

```
> #Source: prostate_cvglm_gcv.R
> #
> library("boot")
> ans <- glm(lpsa ~., data=prostate)
> set.seed(71441555)
> cv.glm(prostate, ans, K=10)$delta
[1] 0.5501038 0.5440883
> cv.glm(prostate, ans, K=10)$delta
[1] 0.5395390 0.5341667
> cv.glm(prostate, ans, K=10)$delta
[1] 0.5767807 0.5692789
> cv.glm(prostate, ans, K=10)$delta
[1] 0.5639577 0.5573455
> #
> NREP <- 500
> tot <- 0
> for (i in 1:NREP) {
+   tot <- tot + cv.glm(prostate, ans, K=10)$delta
+ }
> tot/NREP
[1] 0.5476549 0.5418626
> #
> gcv(prostate[,1:8], prostate[,9], NCores=8, yhat=yhat_lm)
           epe      sd_epe     pcorr
[1,] 0.5499466 0.007984467 0.7663384
> gcv(prostate[,1:8], prostate[,9], NCores=8, yhat=yhat_lm)
           epe      sd_epe     pcorr
[1,] 0.5487371 0.007861839 0.7669304
```

R script

```
#Source: prostate_cvglm_gcv.R
#
```

```
library("boot")
ans <- glm(lpsa ~., data=prostate)
set.seed(71441555)
cv.glm(prostate, ans, K=10)$delta
cv.glm(prostate, ans, K=10)$delta
cv.glm(prostate, ans, K=10)$delta
cv.glm(prostate, ans, K=10)$delta
#
NREP <- 500
tot <- 0
for (i in 1:NREP) {
  tot <- tot + cv.glm(prostate, ans, K=10)$delta
}
tot/NREP
#
gcv(prostate[,1:8], prostate[,9], NCores=8, yhat=yhat_lm)
gcv(prostate[,1:8], prostate[,9], NCores=8, yhat=yhat_lm)
```