Quick Overfiew

This function tunes blackbox regression models, outputting a predictive model given the following criteria. It relies on a learning algorithm with an n by p matrix of covariates and a n by 1 vector of responses/outcomes. The learning algorithm maps inputs to outputs. We also have a "regularization method" (either dropout, noise addition, or robust) with a fixed number of cross validation folds and (for dropout or noise addition) an overall fixed number of Monte Carlo simulations. Mean squared error and mean absolute deviation are used for optimization criteria. So it essentially finds the optimal amount of regularization conditional on the amount of simulation, the data, and a measure of performance.

Explanation/Context

In simple terms, the below R code creates a function **claim()** that is a tuned blackbox model that relies on a given regularization method to optimize the performance of a learning algorithm. Before claim is reached, the user (whoever is running the code) will input some necessary information using the readline() function. The user will be prompted to give the number of Y observations, the dimension of the estimator (commonly known as β), the number of cross validation folds, whether the evaluation method is squared error or absolute deviation, which regularization method to choose, and the number of Monte Carlo simulations to run if the regularization method requires this procedure. The function claim() itself simply needs a learning algorithm, X and Y data, and a column bound c. Note that the column bound is only nonzero if the regularization method is robust (I couldn't figure out how to use readline to create a matrix).

To go through the logic of the claim() function broadly, it first makes sure that X,Y,c will perform as needed for how the functions have been defined (using matrix logic). Then functions are created to execute parts of the optimization process. Functions are created to calculate MSE and MAE, then tuning functions are created. Then a modification function is used to represent the regularization methods. Dropout and Noise Addition have the same tuning function, which goes through and tries to find the best parameter (phi and sigma, respectively). There is a separate tuning function for robust because there you are looking for the "worst" parameter (delta) which gives a worst-case minimization after the worst delta has been found (this is useful for creating an estimator that can handle the extremes in a sense). All the tuning functions use cross-validation to segment the data. In each of the tuning functions, the specified criteria (mae or mad) is used to make the decisions. After all the functions are created, the parameters are found used for the specified regularization method. The return() of the claim() function is the learning algorithm based on an X created using modification function (again, based on the chosen regularization method) with this best parameter. Therefore, our primary function creates a tuned blackbox model.

This function was successfully created without error (installing the RLab function). Do not press ctrl+A and run because of the readline functions. If there are problems with running this yourself, try pasting the readline functions one at a time. Also, it's an implicit requirement that the user knows the dimensions of the data/estimator and won't make a mistake when entering everything. Email paul.bousquet@duke.edu for more information

R Code

```
#install.packages("Rlab")
library(Rlab)
# these functions will allow the user to enter necessary values
# make sure to run these one at a time otherwise R will put NA
n <- as.numeric(readline("Enter an integer for number of Y obs  "))
p <- as.numeric(readline("Enter an integer for dimension of estimator  "))
K <- as.numeric(readline("Enter an integer for number of CV folds  "))
crit <- as.character(readline("Enter (in lowercase) mse or mae  "))
mmode <- as.numeric(readline("Enter 1 for Dropout, 2 for NoiseAdd, 3 for Robust  "))
if (mmode==1 || mmode==2){
 M <- as.numeric(readline("Enter an integer for number of simulations  "))
}
# This is the main function that will return a model based on the input
# set c to 0 if this isn't a robust model
# if robust set it to a px1 matrix (same as beta) or a constant
# set learning algorithm as function per the problem
claim <- function(X,Y,c,learna){
 X <- as.matrix(X)
 Y <- as.matrix(Y)
 c <- as.matrix(c)
 # calculating criteria
 calcms <- function(X,Y,pfun){
   sse <- 0
   for (i in 1:n){
     pred <- pfun(X)
     sse <- sse+ (Y[i] - pred)^2
   }
   return(sse/n)
 }
 calcma <- function(X,Y,pfun){
   sad <- 0
   for (i in 1:n){
     pred <- pfun(X)
     sad <- sad + abs(Y[i]-pred)
   }
   return(sad/n)
 }
 #different modification/regularzation functions
 modif <- function(X,mas){
   if (mmode==1){
     for (j in 1:M){
       for (i in 1:n){
         Z = rbern(1,1-mas)
         X[i,] <- (X[i,]*Z)/(1-mas)
       }
     }
     return(X)
   }
   if (mmode==2){
     for (j in 1:M){
       X <- X + rnorm(n,0,mas)
     }
     return(X)
   }
   if (mmode==3){
     X + mas
     return(X)
   }
 }
 # tuning function for noise and dropout
 # params be a possible range of parameters (a fx1 matrix for an integer f)
 tun <- function(learna,X,Y,params){
   # keeping track of the best performance as the loops iterate
```

```
  bestp <- 0
  # setting high initial average so the first iteration will set the..
  # ..first benchmark for average performance
  besta <- 10e99
  for (j in 1:length(params)){
    para <- params[j]
    # storing variables
    currp <- 0
    #shuffle the data
    X <-X[sample(nrow(X)),]
    Y <- Y[sample(nrow(Y)),]

    # K folds
    folds <- cut(seq(1,nrow(X)),breaks=K,labels=FALSE)

    #K Cross-validation
    for(i in 1:K){
      #Segmenting and modifying data
      testi <- which(folds==i,arr.ind=TRUE)
      testx <- X[testi, ]
      trainx <- X[-testi, ]
      testy <- Y[testi, ]
      trainy <- Y[-testi, ]
      # imposing the modifying method
      testx <- modif(testx,para)
      trainx <- modif(trainx,para)
      #using alg to make prediction
      predf <- learna(trainx,trainy)
      if (crit=="mse"){
        currp <- currp + calcms(testx,testy,predf)
      }else{
        currp <- currp + calcma(testx,testy,predf)
      }
    }
    # average performance
    avp <- currp/K
    if (avp < besta){
      besta <- avp
      bestp <- para
    }
  }
  return(bestp)
}
# robust tuning
robtun <- function(learna,X,Y){
  # storing variable
  worstp <- 0
  deltmax <- matrix()
  # iterate this a high number of times since the process is random
  for (iter in 1:200){
    deltamat <- matrix(,nrow = n,ncol = 1)
    for (d in 1:length(c)){
      colb <- c[d]
      # creating random distributional variables
      randv <- as.matrix(runif(n))
      rands <- sum(randv)
      scaledv <- randv/rands
      scaledv <- scaledv + (colb^2)
      for (i in 1:n){
        sqroot <- sqrt(randv[i])
        # randomly change sign
        if (runif(1)<.5){
          sqroot <- sqroot*(-1)
        }
        deltamat[i] <- sqroot
```

```
      }
    }
    # now basically do the reverse of the regular tuning function
    X <- modif(X)
    currp <- 0
    X <-X[sample(nrow(X)),]
    Y <- Y[sample(nrow(Y)),]

    # K folds
    folds <- cut(seq(1,nrow(X)),breaks=K,labels=FALSE)

    #K Cross-validation
    for(i in 1:K){
      #Segmenting and modifying data
      testi <- which(folds==i,arr.ind=TRUE)
      testx <- X[testi, ]
      trainx <- X[-testi, ]
      testy <- Y[testi, ]
      trainy <- Y[-testi, ]
      # imposing the modifying method
      testx <- modif(testx,deltamat)
      trainx <- modif(trainx,deltamat)
      #using alg to make prediction
      predf <- learna(trainx,trainy)
      if (crit=="mse"){
        currp <- currp + calcms(testx,testy,predf)
      }else{
        currp <- currp + calcma(testx,testy,predf)
      }
    }
    # average performance
    avp <- currp/K
    if (avp > worstp){
      worstp <- avp
      deltmax <- deltamat
    }
  }
  return(deltmax)
}
tunep <- 0
# these if conditions will take a range of parameters and use the tuning..
#..functions to collect the best one
if (mmode==1){
  paramr <- matrix(0,nrow = 20,ncol=1)
  for (i in 1:20){
    paramr[i] <- .5*i
  }
  tunep <- tun(learna,X,Y,paramr)
}
if (mmode==2){
  paramr <- matrix(0,nrow = 50,ncol = 1)
    for (i in 1:50){
      paramr[i] <- .1*i
    }
  tunep <- tune(learna,X,Y,paramr)
}
if (mmode==3){
  tunep <- robtun(learna,X,Y)
}
# return your given algo with the tuned parameter
return(learna(modif(X,tunep),Y))
}
```