Package 'conformalInference'

March 26, 2016

Type Package

Title Tools for conformal inference in regression

Version 1.0.0

Date 2016-01-20

Author Carnegie Mellon Conformal Inference Team: Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman

Maintainer Ryan Tibshirani <ryantibs@stat.cmu.edu>

Depends

Suggests glmnet, lars, plyr, randomForest, SAM, splines

Description A collection of tools for distribution-free inference for regression problems using the theory of conformal prediction.

License GPL-2

RoxygenNote 5.0.1

R topics documented:

`conformalInference-package` *Tools for conformal inference in regression*

Description

A collection of tools for distribution-free inference for regression problems using the theory of conformal prediction.

Details

Conformal inference is a framework for converting any pre-chosen estimator of the regression function into prediction intervals with finite-sample validity, under essentially no assumptions on the data-generating process (aside from the the assumption of i.i.d. observations). The main functions in this package for computing such prediction intervals are `conformal.pred` (shorter intervals, but more expensive) and `conformal.pred.split` (wider intervals, but much more efficient).

Some key references (in reverse chronological order): - "Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, and Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016. - "Classification with Confidence" by Jing Lei (2014). - "Distribution-Free Prediction Bands for Non-parametric Regression" by Jing Lei and Larry Wasserman (2014). - "A Conformal Prediction Approach to Explore Functional Data" by Jing Lei, Alessandro Rinaldo, and Larry Wasserman (2013). - "Distribution Free Prediction Sets" by Jing Lei, James Robins, and Larry Wasserman (2013). - "On-line Predictive Linear Regression" by Vladimir Vovk, Ilia Nouretdinov, and Alex Gammerman (2009). - "Algorithmic Learning in a Random World" by Vladimir Vovk, Alex Gammerman, and Glenn Shafer (2005).

`conformal.pred` *Conformal prediction intervals.*

Description

Compute prediction intervals using conformal inference.

Usage

```
conformal.pred(x, y, x0, train.fun, predict.fun, alpha = 0.1,
  mad.train.fun = NULL, mad.predict.fun = NULL, num.grid.pts = 100,
  grid.factor = 0.25, grid.method = c("linear", "floor", "ceiling"),
  verbose = FALSE)
```

Arguments

x	Matrix of features, of dimension (say) n x p.
y	Vector of responses, of length (say) n.
x0	Matrix of features, each row being a point at which we want to form a prediction interval, of dimension (say) n0 x p.
train.fun	A function to perform model training, i.e., to produce an estimator of E(Y|X), the conditional expectation of the response variable Y given features X. Its input arguments should be x: matrix of features, y: vector of responses, and out: the output produced by a previous call to train.fun, at the *same* features x. The function train.fun may (optionally) leverage this returned output for efficiency purposes. See details below.
predict.fun	A function to perform prediction for the (mean of the) responses at new feature values. Its input arguments should be out: output produced by train.fun, and newx: feature values at which we want to make predictions.
alpha	Miscoverage level for the prediction intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.
mad.train.fun	A function to perform training on the absolute residuals i.e., to produce an estimator of E(R|X) where R is the absolute residual R = |Y - m(X)|, and m denotes the estimator produced by train.fun. This is used to scale the conformal score, to produce a prediction interval with varying local width. The input arguments to mad.train.fun should be x: matrix of features, y: vector of absolute residuals, and out: the output produced by a previous call to mad.train.fun, at the *same* features x. The function mad.train.fun may (optionally) leverage this returned output for efficiency purposes. See details below. The default for mad.train.fun is NULL, which means that no training is done on the absolute residuals, and the usual (unscaled) conformal score is used. Note that if mad.train.fun is non-NULL, then so must be mad.predict.fun (next).
mad.predict.fun	A function to perform prediction for the (mean of the) absolute residuals at new feature values. Its input arguments should be out: output produced by mad.train.fun, and newx: feature values at which we want to make predictions. The default for mad.predict.fun is NULL, which means that no local scaling is done for the conformal score, i.e., the usual (unscaled) conformal score is used.
num.grid.pts	Number of grid points used when forming the conformal intervals (each grid point is a trial value for the interval). Default is 100.
grid.factor	Expansion factor used to define the grid for the conformal intervals. Default is 0.25, which means that the grid extends 25 the range of observed y's, on either side (to the left of the min, to the right of max).
grid.method	One of "linear", "floor", "ceiling". Default is "linear" representing the method used to interpolate the endpoints of the conformal conformal prediction interval from the grid of query values. The first "linear" corresponds to linear interpolation; the other options "floor" and "ceiling" choose the nearest and farthest grid endpoints, respectively, i.e., return the least conservative and most conservative prediction intervals, respectively. When num.grid.pts is large, all three options will perform basically the same.
verbose	Should intermediate progress be printed out? Default is FALSE.

Details

For concreteness, suppose that we want to use the predictions from forward stepwise regression at steps 1 through 5 in the path. In this case, there are m = 5 internal tuning parameter values to predict.fun, in the notation used above, and each of the returned matrices pred, lo, and up will have 5 rows (one corresponding to each step of the forward stepwise path). The code is structured in this way so that we may defined a single pair of functions train.fun and predict.fun, over a set of m = 5 tuning parameter values, instead of calling the conformal function separately m = 5 times.

The third arugment to train.fun, as explained above, is the output produced by a previous call to train.fun, but importantly, at the *same* features x. The function train.fun may (optionally) leverage this returned output for efficiency purposes. Here is how it is used, in this function: when successively querying several trial values for the prediction interval, denoted, say, y0[j], j = 1,2,3,..., we set the out argument in train.fun to be the result of calling train.fun at the previous trial value y0[j-1]. Note of course that the function train.fun can also choose to ignore the returned output out, and most default training functions made available in this package will do so, for simplicity. An exception is the training function produced by `lm.funs`. This will use this output efficiently: the first time it trains by regressing onto a matrix of features x, it computes an appropriate Cholesky factorization; each successive time it is asked to train by regressing onto the same matrix x, it simply uses this Cholesky factorization (rather than recomputing one). Finally, the analogous explanation and discussion here applies to the out argument used by mad.train.fun.

Value

A list with the following components: pred, lo, up, fit. The first three are matrices of dimension m x n0, and the last is a matrix of dimension m x n. Recall that n0 is the number of rows of x0, and m is the number of tuning parameter values internal to predict.fun. In a sense, each of the m columns really corresponds to a different prediction function; see details below. Hence, the rows of the matrices pred, lo, up give the predicted value, and lower and upper confidence limits (from conformal inference), respectively, for the response at the n0 points given in x0. The rows of fit give the fitted values for the n points given in x.

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

See Also

`conformal.pred.jack`, `conformal.pred.split`, `conformal.pred.roo`

Examples

```
## Lasso: use a fixed sequence of 100 lambda values

# Generate some example training data
set.seed(33)
```

```
n = 100; p = 120; s = 10
x = matrix(rnorm(n*p),n,p)
beta = c(rnorm(s),rep(0,p-s))
y = x %*% beta + rnorm(n)

# Generate some example test data
n0 = 50
x0 = matrix(rnorm(n0*p),n0,p)
y0 = x0 %*% beta + rnorm(n0)

# Grab a fixed lambda sequence from one call to glmnet, then
# define lasso training and prediction functions
if (!require("glmnet",quietly=TRUE)) {
  stop("Package glmnet not installed (required for this example)!")
}
out.gnet = glmnet(x,y,nlambda=100,lambda.min.ratio=1e-3)
lambda = out.gnet$lambda
funs = lasso.funs(lambda=lambda)

# Conformal inference, and jacknife and split conformal versions
out.conf = conformal.pred(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict, verb=TRUE)

out.jack = conformal.pred.jack(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.conf$lo)),nrow=n0)
cov.conf = colMeans(out.conf$lo <= y0.mat & y0.mat <= out.conf$up)
len.conf = colMeans(out.conf$up - out.conf$lo)
err.conf = colMeans((y0.mat - out.conf$pred)^2)

cov.jack = colMeans(out.jack$lo <= y0.mat & y0.mat <= out.jack$up)
len.jack = colMeans(out.jack$up - out.jack$lo)
err.jack = colMeans((y0.mat - out.jack$pred)^2)

cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac,newdata=list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(lambda),cov.conf,type="l",ylim=c(0,1),
```

```
     xlab="log(lambda)",ylab="Avg coverage",
     main=paste0("Conformal + lasso (fixed lambda sequence):",
       "\nAverage coverage"))
points(log(lambda),cov.conf,pch=20)
lines(log(lambda),cov.jack,col=3)
points(log(lambda),cov.jack,pch=20,col=3)
lines(log(lambda),cov.split,col=4)
points(log(lambda),cov.split,pch=20,col=4)
abline(h=cov.orac,lty=2,col=2)
legend("bottomleft",col=c(1,3,4,2),lty=c(1,1,1,2),
       legend=c("Conformal","Jackknife conformal",
         "Split conformal","Oracle"))

# Plot average length
plot(log(lambda),len.conf,type="l",
     ylim=range(len.conf,len.orac),
     xlab="log(lambda)",ylab="Avg length",
     main=paste0("Conformal + lasso (fixed lambda sequence):",
       "\nAverage length"))
points(log(lambda),len.conf,pch=20)
lines(log(lambda),len.jack,col=3)
points(log(lambda),len.jack,pch=20,col=3)
lines(log(lambda),len.split,col=4)
points(log(lambda),len.split,pch=20,col=4)
abline(h=len.orac,lty=2,col=2)
legend("topleft",col=c(1,3,4,2),lty=c(1,1,1,2),
       legend=c("Conformal","Jackknife conformal",
         "Split conformal","Oracle"))

# Plot test error
plot(log(lambda),err.conf,type="l",
     ylim=range(err.conf,err.orac),
     xlab="log(lambda)",ylab="Test error",
     main=paste0("Conformal + lasso (fixed lambda sequence):",
       "\nTest error"))
points(log(lambda),err.conf,pch=20)
lines(log(lambda),err.jack,col=3)
points(log(lambda),err.jack,pch=20,col=3)
lines(log(lambda),err.split,col=4)
points(log(lambda),err.split,pch=20,col=4)
abline(h=err.orac,lty=2,col=2)
legend("topleft",col=c(1,3,4,2),lty=c(1,1,1,2),
       legend=c("Conformal","Jackknife conformal",
         "Split conformal","Oracle"))
```

`conformal.pred.jack` *Jackknife conformal prediction intervals.*

Description

Compute prediction intervals using a jackknife variant of conformal inference.

Usage

```
conformal.pred.jack(x, y, x0, train.fun, predict.fun, alpha = 0.1,
  special.fun = NULL, mad.train.fun = NULL, mad.predict.fun = NULL,
  verbose = FALSE)
```

Arguments

x	Matrix of features, of dimension (say) n x p.
y	Vector of responses, of length (say) n.
x0	Matrix of features, each row being a point at which we want to form a prediction interval, of dimension (say) n0 x p.
train.fun	A function to perform model training, i.e., to produce an estimator of E(Y|X), the conditional expectation of the response variable Y given features X. Its input arguments should be x: matrix of features, and y: vector of responses.
predict.fun	A function to perform prediction for the (mean of the) responses at new feature values. Its input arguments should be out: output produced by train.fun, and newx: feature values at which we want to make predictions.
alpha	Miscoverage level for the prediction intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.
special.fun	A function to compute leave-one-out fitted values in an an efficient manner. For example, for many linear smoothers (e.g., linear regression smoothing splines, kernel smoothing — essentially, anything for which the fast leave-one-out cross-validation formula holds), we have the "magic" property: $$y_i - \hat{f}^{(-i)}(x_i) = \frac{y_i - \hat{f}(x_i)}{1 - S_{ii}},$$ where $\hat{f}$ is the linear smoother trained on the entire data set $\hat{f}^{(-i)}$ is the linear smoother trained on all but the ith data point, and S_{ii} is the ith diagonal element of the smoothing matrix. The input arguments to special.fun should be x: matrix of features, y: vector of responses, and out: output produced by train.fun when run on x, y. The default for special.fun is NULL, which means that no special leave-one-out formula is leverage, and the usual (manual) way of computing leave-one-out fitted values is used. Note that if mad.train.fun and mad.predict.fun are passed, in order to obtain prediction intervals with locally varying width, then special.fun will be ignored (since the fast shortcut it offers can no longer be used).
mad.train.fun	A function to perform training on the absolute residuals i.e., to produce an estimator of E(R|X) where R is the absolute residual R = |Y - m(X)|, and m denotes the estimator produced by train.fun. This is used to scale the conformal score, to produce a prediction interval with varying local width. The input arguments to mad.train.fun should be x: matrix of features, and y: vector of absolute residuals. The default for mad.train.fun is NULL, which means that no training is done on the absolute residuals, and the usual (unscaled) conformal score is used. Note that if mad.train.fun is non-NULL, then so must be mad.predict.fun (next).

`mad.predict.fun`
: A function to perform prediction for the (mean of the) absolute residuals at new feature values. Its input arguments should be out: output produced by mad.train.fun, and newx: feature values at which we want to make predictions. The default for mad.predict.fun is NULL, which means that no local scaling is done for the conformal score, i.e., the usual (unscaled) conformal score is used.

`verbose`
: Should intermediate progress be printed out? Default is FALSE.

`alpha`
: Miscoverage level for the prediction intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.

Details

For concreteness, suppose that we want to use the predictions from forward stepwise regression at steps 1 through 5 in the path. In this case, there are m = 5 internal tuning parameter values to predict.fun, in the notation used above, and each of the returned matrices pred, lo, and up will have 5 rows (one corresponding to each step of the forward stepwise path). The code is structured in this way so that we may defined a single pair of functions train.fun and predict.fun, over a set of m = 5 tuning parameter values, instead of calling the conformal function separately m = 5 times.

Value

A list with the following components: pred, lo, up, fit. The first three are matrices of dimension m x n0, and the last is a matrix of dimension m x n. Recall that n0 is the number of rows of x0, and m is the number of tuning parameter values internal to predict.fun. In a sense, each of the m columns really corresponds to a different prediction function; see details below. Hence, the rows of the matrices pred, lo, up give the predicted value, and lower and upper confidence limits, respectively, for the response at the n0 points given in x0. The predicted values are computed based on the entire training set, and the confidence limits are computed using jackknife conformal inference. The rows of fit give the fitted values for the n points given in x.

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

http://www.stat.cmu.edu

See Also

`conformal.pred`, `conformal.pred.split`, `conformal.pred.roo`

Examples

```
## See examples for conformal.pred function
```

`conformal.pred.roo` *In-sample split conformal prediction intervals.*

Description

Compute prediction intervals, having valid in-sample coverage, using rank-one-out (ROO) split conformal inference.

Usage

```
conformal.pred.roo(x, y, train.fun, predict.fun, alpha = 0.1,
  mad.train.fun = NULL, mad.predict.fun = NULL, split = NULL,
  seed = NULL, verbose = FALSE)
```

Arguments

`x`	Matrix of features, of dimension (say) n x p.
`y`	Vector of responses, of length (say) n.
`train.fun`	A function to perform model training, i.e., to produce an estimator of E(Y|X), the conditional expectation of the response variable Y given features X. Its input arguments should be x: matrix of features, and y: vector of responses.
`predict.fun`	A function to perform prediction for the (mean of the) responses at new feature values. Its input arguments should be out: output produced by train.fun, and newx: feature values at which we want to make predictions.
`alpha`	Miscoverage level for the prediction intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.
`mad.train.fun`	A function to perform training on the absolute residuals i.e., to produce an estimator of E(R|X) where R is the absolute residual R = |Y - m(X)|, and m denotes the estimator produced by train.fun. This is used to scale the conformal score, to produce a prediction interval with varying local width. The input arguments to mad.train.fun should be x: matrix of features, and y: vector of absolute residuals. The default for mad.train.fun is NULL, which means that no training is done on the absolute residuals, and the usual (unscaled) conformal score is used. Note that if mad.train.fun is non-NULL, then so must be mad.predict.fun (see next).
`mad.predict.fun`	A function to perform prediction for the (mean of the) absolute residuals at new feature values. Its input arguments should be out: output produced by mad.train.fun, and newx: feature values at which we want to make predictions. The default for mad.predict.fun is NULL, which means that no local scaling is done for the conformal score, i.e., the usual (unscaled) conformal score is used.
`split`	Indices that define the data-split to be used (i.e., the indices define the first half of the data-split). Default is NULL, in which case the split is chosen randomly.
`seed`	Integer to be passed to set.seed before defining the random data-split to be used. Default is NULL, which effectively sets no seed. If both split and seed are passed, the former takes priority and the latter is ignored.
`verbose`	Should intermediate progress be printed out? Default is FALSE.

Details

For concreteness, suppose that we want to use the predictions from forward stepwise regression at steps 1 through 5 in the path. In this case, there are m = 5 internal tuning parameter values to predict.fun, in the notation used above, and each of the returned matrices pred, lo, up, fit will have 5 columns (one for each step of the forward stepwise path). The code is structured in this way so that we may defined a single pair of functions train.fun and predict.fun, over a set of m = 5 tuning parameter values, instead of calling the conformal function separately m = 5 times.

The rank-one-out or ROO variant of split conformal prediction produces intervals with a special in-sample average coverage property. Namely, if we were to observe new response values y* at the given feature values x, then the ROO method produces a prediction band $\hat{C}(x)$ with the asymptotic in-sample average coverage property

$$\liminf_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} P\big(Y_i^* \in \hat{C}(X_i)\big) \geq 1-\alpha.$$

The usual split conformal band would not necessarily share this in-sample property. Of course, the split conformal band has valid predictive (i.e., out-of-sample) coverage, and the ROO split conformal variant maintains this property as well.

Value

A list with the following components: pred, lo, up, fit, split, fit.all, out.all. The first four are matrices of dimension n x m. In a sense, each of the m columns really corresponds to a different prediction function; see details below. Hence, the rows of the matrices pred, lo, up give the predicted value, and lower and upper confidence limits (from rank-one-out split conformal inference), respectively, for the response at the n points given in x. The rows of the matrix fit give the fitted values for the n points given in x. The indices used for the half of the data-split are returned in split. Finally, for convenience, the output from running train.out on the entire (unsplit) data set x,y is stored in out.all, and fit.all is an n x m matrix whose rows store the fitted values from out.all.

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

See Also

conformal.pred, conformal.pred.jack, conformal.pred.split

Examples

```
## Lasso: use a fixed sequence of 100 lambda values

# Generate some example training data
set.seed(33)
```

```
n = 200; p = 500; s = 10
x = matrix(rnorm(n*p),n,p)
beta = c(rnorm(s),rep(0,p-s))
y = x %*% beta + rnorm(n)

# Generate some example test data, but using the same x points
n0 = n
x0 = x
y0 = x0 %*% beta + rnorm(n0)

# Grab a fixed lambda sequence from one call to glmnet, then
# define lasso training and prediction functions
if (!require("glmnet",quietly=TRUE)) {
  stop("Package glmnet not installed (required for this example)!")
}
out.gnet = glmnet(x,y,nlambda=100,lambda.min.ratio=1e-4)
lambda = out.gnet$lambda
funs = lasso.funs(lambda=lambda)

# Rank-one-out conformal inference, and split conformal inference
out.roo = conformal.pred.roo(x, y, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.roo$lo)),nrow=n0)
cov.roo = colMeans(out.roo$lo <= y0.mat & y0.mat <= out.roo$up)
len.roo = colMeans(out.roo$up - out.roo$lo)
err.roo = colMeans((y0.mat - out.roo$pred)^2)

cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac,newdata=list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(lambda),cov.roo,type="l",ylim=c(0,1),
     xlab="log(lambda)",ylab="Avg coverage",
     main=paste0("ROO conformal + lasso (fixed lambda sequence):",
       "\nAverage coverage"))
points(log(lambda),cov.roo,pch=20)
lines(log(lambda),cov.split,col=4)
points(log(lambda),cov.split,pch=20,col=4)
abline(h=cov.orac,lty=2,col=2)
```

```
legend("bottomleft",col=c(1,4,2),lty=c(1,1,2),
       legend=c("ROO conformal","Split conformal","Oracle"))

# Plot average length
plot(log(lambda),len.roo,type="l",
     ylim=range(len.roo,len.split,len.orac),
     xlab="log(lambda)",ylab="Avg length",
     main=paste0("Split conformal + lasso (fixed lambda sequence):",
       "\nAverage length"))
points(log(lambda),len.roo,pch=20)
lines(log(lambda),len.split,col=4)
points(log(lambda),len.split,pch=20,col=4)
abline(h=len.orac,lty=2,col=2)
legend("bottomleft",col=c(1,4,2),lty=c(1,1,2),
       legend=c("ROO conformal","Split conformal","Oracle"))

# Plot test error
plot(log(lambda),err.roo,type="l",
     ylim=range(err.roo,err.split,err.orac),
     xlab="log(lambda)",ylab="Test error",
     main=paste0("Split conformal + lasso (fixed lambda sequence):",
       "\nTest error"))
points(log(lambda),err.roo,pch=20)
lines(log(lambda),err.split,col=4)
points(log(lambda),err.split,pch=20,col=4)
abline(h=err.orac,lty=2,col=2)
legend("bottomleft",col=c(1,4,2),lty=c(1,1,2),
       legend=c("ROO conformal","Split conformal","Oracle"))
```

`conformal.pred.split` *Split conformal prediction intervals.*

Description

Compute prediction intervals using split conformal inference.

Usage

```
conformal.pred.split(x, y, x0, train.fun, predict.fun, alpha = 0.1,
  mad.train.fun = NULL, mad.predict.fun = NULL, split = NULL,
  seed = NULL, verbose = FALSE)
```

Arguments

`x`	Matrix of features, of dimension (say) n x p.
`y`	Vector of responses, of length (say) n.
`x0`	Matrix of features, each row being a point at which we want to form a prediction interval, of dimension (say) n0 x p.

`train.fun` A function to perform model training, i.e., to produce an estimator of E(Y|X), the conditional expectation of the response variable Y given features X. Its input arguments should be x: matrix of features, and y: vector of responses.

`predict.fun` A function to perform prediction for the (mean of the) responses at new feature values. Its input arguments should be out: output produced by train.fun, and newx: feature values at which we want to make predictions.

`alpha` Miscoverage level for the prediction intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.

`mad.train.fun` A function to perform training on the absolute residuals i.e., to produce an estimator of E(R|X) where R is the absolute residual R = |Y - m(X)|, and m denotes the estimator produced by train.fun. This is used to scale the conformal score, to produce a prediction interval with varying local width. The input arguments to mad.train.fun should be x: matrix of features, and y: vector of absolute residuals. The default for mad.train.fun is NULL, which means that no training is done on the absolute residuals, and the usual (unscaled) conformal score is used. Note that if mad.train.fun is non-NULL, then so must be mad.predict.fun (see next).

`mad.predict.fun` A function to perform prediction for the (mean of the) absolute residuals at new feature values. Its input arguments should be out: output produced by mad.train.fun, and newx: feature values at which we want to make predictions. The default for mad.predict.fun is NULL, which means that no local scaling is done for the conformal score, i.e., the usual (unscaled) conformal score is used.

`split` Indices that define the data-split to be used (i.e., the indices define the first half of the data-split, on which the model is trained). Default is NULL, in which case the split is chosen randomly.

`seed` Integer to be passed to set.seed before defining the random data-split to be used. Default is NULL, which effectively sets no seed. If both split and seed are passed, the former takes priority and the latter is ignored.

`verbose` Should intermediate progress be printed out? Default is FALSE.

Details

For concreteness, suppose that we want to use the predictions from forward stepwise regression at steps 1 through 5 in the path. In this case, there are m = 5 internal tuning parameter values to predict.fun, in the notation used above, and each of the returned matrices pred, lo, up, fit will have 5 columns (one for each step of the forward stepwise path). The code is structured in this way so that we may defined a single pair of functions train.fun and predict.fun, over a set of m = 5 tuning parameter values, instead of calling the conformal function separately m = 5 times.

Value

A list with the following components: pred, lo, up, fit, split. The first three are matrices of dimension n0 x m, and the fourth is a matrix of dimension n x m. Recall that n0 is the number of rows of x0, and m is the number of tuning parameter values internal to predict.fun. In a sense, each of the m columns really corresponds to a different prediction function; see details below. Hence, the rows of the matrices pred, lo, up give the predicted value, and lower and upper confidence limits (from split conformal inference), respectively, for the response at the n0 points given in x0. The rows of

fit give the fitted values for the n points given in x. Finally, split contains the indices used for the first half of the data-split.

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

See Also

conformal.pred, conformal.pred.jack, conformal.pred.roo

Examples

```
b
## Lasso: use a fixed sequence of 100 lambda values

# Generate some example training data
set.seed(33)
n = 200; p = 500; s = 10
x = matrix(rnorm(n*p),n,p)
beta = c(rnorm(s),rep(0,p-s))
y = x %*% beta + rnorm(n)

# Generate some example test data
n0 = 1000
x0 = matrix(rnorm(n0*p),n0,p)
y0 = x0 %*% beta + rnorm(n0)

# Grab a fixed lambda sequence from one call to glmnet, then
# define lasso training and prediction functions
if (!require("glmnet",quietly=TRUE)) {
  stop("Package glmnet not installed (required for this example)!")
}
out.gnet = glmnet(x,y,nlambda=100,lambda.min.ratio=1e-4)
lambda = out.gnet$lambda
funs = lasso.funs(lambda=lambda)

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.split$lo)),nrow=n0)
cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
```

```
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac,newdata=list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(lambda),cov.split,type="l",ylim=c(0,1),
     xlab="log(lambda)",ylab="Avg coverage",
     main=paste0("Split conformal + lasso (fixed lambda sequence):",
       "\nAverage coverage"))
points(log(lambda),cov.split,pch=20)
abline(h=cov.orac,lty=2,col=2)
legend("bottomleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot average length
plot(log(lambda),len.split,type="l",
     ylim=range(len.split,len.orac),
     xlab="log(lambda)",ylab="Avg length",
     main=paste0("Split conformal + lasso (fixed lambda sequence):",
       "\nAverage length"))
points(log(lambda),len.split,pch=20)
abline(h=len.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot test error
plot(log(lambda),err.split,type="l",
     ylim=range(err.split,err.orac),
     xlab="log(lambda)",ylab="Test error",
     main=paste0("Split conformal + lasso (fixed lambda sequence):",
       "\nTest error"))
points(log(lambda),err.split,pch=20)
abline(h=err.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

####################

cat("Type return to continue ...\n")
tmp = readLines(n=1)

## Lasso: alternate way, use a dynamic sequence of 100 lambda values

# Lasso training and prediction functions
funs = lasso.funs(nlambda=100)

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)
```

```
y0.mat = matrix(rep(y0,ncol(out.split$lo)),nrow=n0)
cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac,newdata=list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(length(cov.split):1),cov.split,type="l",ylim=c(0,1),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",
     ylab="Avg coverage",
     main=paste0("Split conformal + lasso (dynamic lambda sequence):",
       "\nAverage coverage"))
points(log(length(cov.split):1),cov.split,pch=20)
abline(h=cov.orac,lty=2,col=2)
legend("bottomleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot average length
plot(log(length(len.split):1),len.split,type="l",
     ylim=range(len.split,len.orac),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",
     ylab="Avg length",
     main=paste0("Split conformal + lasso (dynamic lambda sequence):",
       "\nAverage length"))
points(log(length(len.split):1),len.split,pch=20)
abline(h=len.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot test error
plot(log(length(err.split):1),err.split,type="l",
     ylim=range(err.split,err.orac),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",ylab="Test error",
     main=paste0("Split conformal + lasso (dynamic lambda sequence):",
       "\nTest error"))
points(log(length(err.split):1),err.split,pch=20)
abline(h=err.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))
```

`glmnet.funs` *Elastic net, lasso, ridge regression training and prediction functions.*

Description

Construct training and prediction functions for the elastic net, the lasso, or ridge regression, based on the glmnet package, over a sequence of (given or internally computed) lambda values.

Usage

```
elastic.funs(gamma = 0.5, standardize = TRUE, intercept = TRUE,
  lambda = NULL, nlambda = 50, lambda.min.ratio = 1e-04, cv = FALSE,
  cv.rule = c("min", "1se"))

lasso.funs(standardize = TRUE, intercept = TRUE, lambda = NULL,
  nlambda = 50, lambda.min.ratio = 1e-04, cv = FALSE, cv.rule = c("min",
  "1se"))

ridge.funs(standardize = TRUE, intercept = TRUE, lambda = NULL,
  nlambda = 50, lambda.min.ratio = 1e-04, cv = FALSE, cv.rule = c("min",
  "1se"))
```

Arguments

gamma	Mixing parameter (between 0 and 1) for the elastic net, where 0 corresponds to ridge regression, and 1 to the lasso. Default is 0.5.
standardize, intercept	Should the data be standardized, and should an intercept be included? Default for both is TRUE.
lambda	Sequence of lambda values over which training is performed. This must be in decreasing order, and — this argument should be used with caution! When used, it is usually best to grab the sequence constructed by one intial call to glmnet (see examples). Default is NULL, which means that the nlambda, lambda.min.ratio arguments will define the lambda sequence (see next).
nlambda	Number of lambda values over which training is performed. In particular, the lambda sequence is defined by nlambda log-spaced values between lambda.max and lambda.min.ratio * lambda.max, where lambda.max is the smallest value of lambda at which the solution has all zero components, and lambda.min.ratio is a small fraction (see next). Default is 50.
lambda.min.ratio	Small fraction that gets used in conjunction with nlambda to specify a lambda sequence (see above). Default is 1e-4.
cv	Should 10-fold cross-validation be used? If TRUE, then prediction is done with either the (usual) min rule, or the 1se rule. See the cv.rule argument, below. If FALSE (the default), then prediction is done over the same lambda sequence as that used for training.
cv.rule	If the cv argument is TRUE, then cv.rule determines which rule should be used for the predict function, either "min" (the usual rule) or "1se" (the one-standard-error rule). See the glmnet help files for details. Default is "min".

Details

This function is based on the packages glmnet and plyr. If these packages are not installed, then the function will abort. The functions lasso.funs and ridge.funs are convenience functions, they simply call elastic.funs with gamma = 1 and gamma = 0, respectively.

Value

A list with three components: train.fun, predict.fun, active.fun. The third function is designed to take the output of train.fun, and reports which features are active for each fitted model contained in this output.

Author(s)

Ryan Tibshirani

See Also

lars.funs for stepwise, least angle regression, and lasso path training and prediction functions.

Examples

```
## Elastic net: use a fixed sequence of 100 lambda values

# Generate some example training data
set.seed(11)
n = 200; p = 500; s = 10
x = matrix(rnorm(n*p),n,p)
beta = c(rnorm(s),rep(0,p-s))
y = x %*% beta + rnorm(n)

# Generate some example test data
n0 = 1000
x0 = matrix(rnorm(n0*p),n0,p)
y0 = x0 %*% beta + rnorm(n0)

# Grab a fixed lambda sequence from one call to glmnet, then
# define elastic net training and prediction functions
if (!require("glmnet",quietly=TRUE)) {
  stop("Package glmnet not installed (required for this example)!")
}
out.gnet = glmnet(x,y,alpha=0.9,nlambda=100,lambda.min.ratio=1e-4)
lambda = out.gnet$lambda
funs = elastic.funs(gamma=0.9,lambda=lambda)

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.split$lo)),nrow=n0)
cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
```

```
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac, list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(lambda),cov.split,type="l",ylim=c(0,1),
     xlab="log(lambda)",ylab="Avg coverage",
     main=paste0("Split conformal + elastic net (fixed lambda sequence):",
       "\nAverage coverage"))
points(log(lambda),cov.split,pch=20)
abline(h=cov.orac,lty=2,col=2)
legend("bottomleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot average length
plot(log(lambda),len.split,type="l",
     ylim=range(len.split,len.orac),
     xlab="log(lambda)",ylab="Avg length",
     main=paste0("Split conformal + elastic net (fixed lambda sequence):",
       "\nAverage length"))
points(log(lambda),len.split,pch=20)
abline(h=len.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot test error
plot(log(lambda),err.split,type="l",
     ylim=range(err.split,err.orac),
     xlab="log(lambda)",ylab="Test error",
     main=paste0("Split conformal + elastic net (fixed lambda sequence):",
       "\nTest error"))
points(log(lambda),err.split,pch=20)
abline(h=err.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

####################

cat("Type return to continue ...\n")
tmp = readLines(n=1)

## Elastic net: alternate way, use a dynamic sequence of 100 lambda values

# Elastic net training and prediction functions
funs = elastic.funs(gamma=0.9,nlambda=100)
```

```
# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.split$lo)),nrow=n0)
cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac, list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(log(length(cov.split):1),cov.split,type="l",ylim=c(0,1),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",
     ylab="Avg coverage",
     main=paste0("Split conformal + elastic net (dynamic lambda sequence):",
       "\nAverage coverage"))
points(log(length(cov.split):1),cov.split,pch=20)
abline(h=cov.orac,lty=2,col=2)
legend("bottomleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot average length
plot(log(length(len.split):1),len.split,type="l",
     ylim=range(len.split,len.orac),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",
     ylab="Avg length",
     main=paste0("Split conformal + elastic net (dynamic lambda sequence):",
       "\nAverage length"))
points(log(length(len.split):1),len.split,pch=20)
abline(h=len.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot test error
plot(log(length(err.split):1),err.split,type="l",
     ylim=range(err.split,err.orac),
     xlab="log(lambda rank) (i.e., log(1)=0 is smallest)",ylab="Test error",
     main=paste0("Split conformal + elastic net (dynamic lambda sequence):",
       "\nTest error"))
points(log(length(err.split):1),err.split,pch=20)
abline(h=err.orac,lty=2,col=2)
legend("topleft",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

####################
```

```
cat("Type return to continue ...\n")
tmp = readLines(n=1)

## Elastic net: cross-validate over a sequence of 100 lambda values

# Elastic net training and prediction functions
funs = elastic.funs(gamma=0.9,nlambda=100,cv=TRUE)

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

cov.split = mean(out.split$lo <= y0 & y0 <= out.split$up)
len.split = mean(out.split$up - out.split$lo)
err.split = mean((y0 - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac, list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

tab = matrix(c(cov.split,len.split,err.split,
  cov.orac,len.orac,err.orac),ncol=2)
colnames(tab) = c("Split conformal","Oracle")
rownames(tab) = c("Avg coverage","Avg length","Test error")
tab
```

`lars.funs` *Forward stepwise, least angle regression, and lasso training and prediction functions.*

Description

Construct training and prediction functions for forward stepwise, least angle regression net, or the lasso path, based on the `lars` package, over a given number of steps.

Usage

```
lars.funs(type = c("stepwise", "lar", "lasso"), normalize = TRUE,
  intercept = TRUE, max.steps = 20, cv = FALSE, cv.rule = c("min",
  "1se"), use.Gram = TRUE)
```

Arguments

type	One of "stepwise", "lar", or "lasso", indicating which path algorithm should be run. Default is "stepwise".
normalize, intercept	Should the features be normalized (to have unit L2 norm), and should an intercept be included? Default for both is TRUE.
max.steps	Number of steps down the path to be taken before quitting. Default is 20. Prediction is done over steps 1 through max.steps, unless
cv	Should 10-fold cross-validation be used? If TRUE, then prediction is done with either the (usual) min rule, or the 1se rule. See the cv.rule argument, below. If FALSE (the default), then prediction is done over all path steps taken.
cv.rule	If the cv argument is TRUE, then cv.rule determines which rule should be used for the predict function, either "min" (the usual rule) or "1se" (the one-standard-error rule). Default is "min".
use.Gram	Should the Gram matrix (cross product of feature matrix) be stored? Default is TRUE. But storing the Gram matrix may be costly when p (number of colums of feature matrix) is large, say, large compared to n (number rows of feature matrix), so in this case, you may want to choose FALSE.

Details

This function is based on the packages `lars` and `plyr`. If these packages are not installed, then the function will abort.

Value

A list with three components: train.fun, predict.fun, active.fun. The third function is designed to take the output of train.fun, and reports which features are active for each fitted model contained in this output.

Author(s)

Ryan Tibshirani

See Also

`glmnet.funs` for elastic net, lasso, and ridge regression training and prediction functions, defined over a sequence of lambda values.

Examples

```
## Forward stepwise: use 30 path steps

# Generate some example training data
set.seed(33)
n = 100; p = 120; s = 10
x = matrix(rnorm(n*p),n,p)
beta = c(rnorm(s),rep(0,p-s))
```

```
y = x %*% beta + rnorm(n)

# Generate some example test data
n0 = 50
x0 = matrix(rnorm(n0*p),n0,p)
y0 = x0 %*% beta + rnorm(n0)

# Stepwise training and prediction functions
funs = lars.funs(type="stepwise",max.steps=30)

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

y0.mat = matrix(rep(y0,ncol(out.split$lo)),nrow=n0)
cov.split = colMeans(out.split$lo <= y0.mat & y0.mat <= out.split$up)
len.split = colMeans(out.split$up - out.split$lo)
err.split = colMeans((y0.mat - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac, list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

# Plot average coverage
plot(1:length(cov.split),cov.split,type="l",ylim=c(0,1),
     xlab="Number of path steps",ylab="Avg coverage",
     main=paste0("Split conformal + stepwise:\nAverage coverage"))
points(1:length(cov.split),cov.split,pch=20)
abline(h=cov.orac,lty=2,col=2)
legend("bottomright",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot average length
plot(1:length(len.split),len.split,type="l",
     ylim=range(len.split,len.orac),
     xlab="Number of path steps",ylab="Avg length",
     main=paste0("Split conformal + stepwise:\nAverage length"))
points(1:length(len.split),len.split,pch=20)
abline(h=len.orac,lty=2,col=2)
legend("topright",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

# Plot test error
plot(1:length(err.split),err.split,type="l",
     ylim=range(err.split,err.orac),
     xlab="Number of path steps",ylab="Test error",
     main=paste0("Split conformal + stepwise:\nTest error"))
points(1:length(err.split),err.split,pch=20)
```

```
abline(h=err.orac,lty=2,col=2)
legend("topright",col=1:2,lty=1:2,
       legend=c("Split conformal","Oracle"))

####################

cat("Type return to continue ...\n")
tmp = readLines(n=1)

## Forward stepwise: cross-validate over 30 path steps

# Stepwise training and prediction functions
funs = lars.funs(type="stepwise",max.steps=30,cv=TRUE,cv.rule="1se")

# Split conformal inference
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

cov.split = mean(out.split$lo <= y0 & y0 <= out.split$up)
len.split = mean(out.split$up - out.split$lo)
err.split = mean((y0 - out.split$pred)^2)

# Compare to parametric intervals from oracle linear regression
lm.orac = lm(y~x[,1:s])
out.orac = predict(lm.orac, list(x=x0[,1:s]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

tab = matrix(c(cov.split,len.split,err.split,
  cov.orac,len.orac,err.orac),ncol=2)
colnames(tab) = c("Split conformal","Oracle")
rownames(tab) = c("Avg coverage","Avg length","Test error")
tab
```

`lm.funs` *Linear regression training and prediction functions, and special leave-one-out fitting function.*

Description

Construct the training and prediction functions for linear regression, as as a special function for computing leave-one-out fitted values.

Usage

```
lm.funs(intercept = TRUE, lambda = 0)
```

Arguments

intercept	Should an intercept be included in the linear model? Default is TRUE.
lambda	A value or sequence of lambda values to be used in ridge regression. Default is 0, which means ordinary linear regression.

Details

The train.fun function constructed here leverages an optional third argument (in addition to the usual x,y): out, whose default is NULL. If non-NULL, it is assumed to be the result of a previous call to train.fun, on the *same* features x as we are currently considering. This is done for efficiency, and to perform the regression, it uses the saved Cholesky decomposition instead of computing a new one from scratch.

Value

A list with four components: train.fun, predict.fun, special.fun, active.fun. The last function is designed to take the output of train.fun, and reports which features are active for each fitted model contained in this output. Trivially, here, all features are generically active in ridged linear models.

Author(s)

Ryan Tibshirani

Examples

```
## Linear regression: classical setting

# Generate some example training data
set.seed(33)
n = 100; p = 10
x = matrix(rnorm(n*p),n,p)
beta = rnorm(p)
y = x %*% beta + rnorm(n)

# Generate some example test data
n0 = 50
x0 = matrix(rnorm(n0*p),n0,p)
y0 = x0 %*% beta + rnorm(n0)

# Linear regression training and prediction functions
funs = lm.funs()

# Conformal inference, and split conformal inference
out.conf = conformal.pred(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict, verb=TRUE)

out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

cov.conf = mean(out.conf$lo <= y0 & y0 <= out.conf$up)
len.conf = mean(out.conf$up - out.conf$lo)
```

```
err.conf = mean((y0 - out.conf$pred)^2)

cov.split = mean(out.split$lo <= y0 & y0 <= out.split$up)
len.split = mean(out.split$up - out.split$lo)
err.split = mean((y0 - out.split$pred)^2)

# Compare to parametric intervals from linear regression
lm.orac = lm(y~x[,1:p])
out.orac = predict(lm.orac, list(x=x0[,1:p]),
  interval="predict", level=0.9)

cov.orac = mean(out.orac[,"lwr"] <= y0 & y0 <= out.orac[,"upr"])
len.orac = mean(out.orac[,"upr"] - out.orac[,"lwr"])
err.orac = mean((y0 - out.orac[,"fit"])^2)

tab = matrix(c(cov.conf,len.conf,err.conf,
  cov.split,len.split,err.split,
  cov.orac,len.orac,err.orac),ncol=3)
colnames(tab) = c("Conformal","Split conformal","Oracle")
rownames(tab) = c("Avg coverage","Avg length","Test error")
tab

plot(y0, ylim=c(min(out.conf$lo,out.split$lo,out.orac[,"lwr"]),
           max(out.conf$up,out.split$up,out.orac[,"upr"])),
     main="Linear regression: prediction intervals")
segments(1:n0, out.conf$lo, 1:n0, out.conf$up)
segments(1:n0+0.2, out.split$lo, 1:n0+0.2, out.split$up, col=4)
segments(1:n0-0.2, out.orac[,"lwr"], 1:n0-0.2, out.orac[,"upr"], lty=2, col=2)
legend("topleft",col=c(1,4,2),lty=c(1,1,2),
       legend=c("Conformal","Split conformal","Oracle"))
```

`loco` *Variable importance via sample splitting.*

Description

Compute confidence intervals for mean (or median) excess prediction error after leaving out one feature.

Usage

```
loco(x, y, train.fun, predict.fun, active.fun, alpha = 0.1, split = NULL,
  seed = NULL, verbose = FALSE)
```

Arguments

x	Matrix of features, of dimension (say) n x p.
y	Vector of responses, of length (say) n.

train.fun	A function to perform model training, i.e., to produce an estimator of E(Y|X), the conditional expectation of the response variable Y given features X. Its input arguments should be x: matrix of features, and y: vector of responses.
predict.fun	A function to perform prediction for the (mean of the) responses at new feature values. Its input arguments should be out: output produced by train.fun, and newx: feature values at which we want to make predictions.
active.fun	A function which takes the output of train.fun, and reports which features are active for each fitted model contained in this output. Its only input argument should be out: output produced by train.fun.
alpha	Miscoverage level for the confidence intervals, i.e., intervals with coverage 1-alpha are formed. Default for alpha is 0.1.
split	Indices that define the data-split to be used (i.e., the indices define the first half of the data-split, on which the model is trained). Default is NULL, in which case the split is chosen randomly.
seed	Integer to be passed to set.seed before defining the random data-split to be used. Default is NULL, which effectively sets no seed. If both split and seed are passed, the former takes priority and the latter is ignored.
verbose	Should intermediate progress be printed out? Default is FALSE.

Details

In leave-one-covariate-out or LOCO inference, the training data is split in two parts, and the first part is used to train a model, or some number of models across multiple tuning steps (e.g., indexed by different tuning parameter values lambda in the lasso, or different steps along the forward stepwise path). In each model, each variable is left out one at time from the first part of the data, and the entire training procedure is repeated without this variable in consideration. Residuals are computed on both from the original fitted model, and the re-fitted model without the variable in consideration. A pairwise difference between the latter and former residuals is computed, and either a Z-test, sign test, or Wilcoxon signed rank test is performed to test either the mean or median difference here being zero. The Z-test is of course approximately valid under the conditions needed for the CLT; the sign test is distribution-free and exact in finite samples (only assumes continuity of the underlying distribution); the Wilcoxon signed rank test is also distribution-free and exact in finite samples, but may offer more power (it assumes both continuity and symmetry of the underlying distribution).

A few other important notes: p-values here are from a one-sided test of the target parameter (mean or median excess test error) being equal to zero versus greater than zero. Confidence intervals are from inverting the two-sided version of this test. Furthermore, all p-values and confidence intervals have been Bonferroni", corrected for multiplicity.

Value

A list with the following components: inf.z, inf.sign, inf.wilcox, active, master. The first three are lists, containing the results of LOCO inference with the Z-test, sign text, and Wilcoxon signed rank test, respectively. More details on these tests are given below. These lists have one element per tuning step inherent to the training and prediction functions, train.fun and predict.fun. The fourth returned component active is a list, with one element per tuning step, that reports which features are active in the corresponding fitted model. The last returned component master collects all active features across all tuning steps, for easy reference.

Author(s)

Ryan Tibshirani, Larry Wasserman

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

Examples

```
## Use lasso + CV to choose a model, then test variable importance in the
## selected model

# Generate some example training data
set.seed(33)
n = 200; p = 500; s = 5
x = matrix(rnorm(n*p),n,p)
beta = 2*c(rnorm(s),rep(0,p-s))
y = x %*% beta + rnorm(n)

# Lasso training and prediction functions, using cross-validation over 100
# values of lambda, with the 1se rule
funs = lasso.funs(nlambda=100,cv=TRUE,cv.rule="1se")

# Split-sample LOCO analysis
out.loco = loco(x, y, alpha=0.1, train.fun=funs$train, predict.fun=funs$predict,
  active.fun=funs$active.fun, verbose=TRUE)
out.loco

# Plot the Wilcoxon intervals
ylim = range(out.loco$inf.wilcox[[1]][,2:3])
J = length(out.loco$active[[1]])
plot(c(), c(), xlim=c(1,J), ylim=ylim, xaxt="n",
     main="LOCO analysis from lasso + CV model",
     xlab="Variable", ylab="Confidence interval")
axis(side=1, at=1:J, labels=FALSE)
for (j in 1:J) {
  axis(side=1, at=j, labels=out.loco$active[[1]][j], cex.axis=0.75,
       line=0.5*j%%2)
}
abline(h=0)
segments(1:J, out.loco$inf.wilcox[[1]][,2],
         1:J, out.loco$inf.wilcox[[1]][,3],
         col="red", lwd=2)
```

`plot.sim` *Plot function for sim object.*

Description

Plot the results of a set of simulations, stored in an object of class sim (produced by sim.master).

Usage

```
## S3 method for class 'sim'
plot(x, method.nums = 1:length(x$ave.cov),
  method.names = NULL, cols = 1:8, log = "", xlim = NULL, se = FALSE,
  main = NULL, cex.main = 1.25, xlab = "Relative optimism",
  legend.pos = c("bottomleft", "", ""), make.pdf = FALSE, fig.dir = ".",
  file.prefix = "", h = 6, w = 6, mar = NULL, ...)
```

`print.loco` *Print function for loco object.*

Description

Summary and print the results of a set of simulations, stored an object of class sim (produced by sim.master).

Usage

```
## S3 method for class 'loco'
print(x, test = c("wilcox", "sign", "z", "all"), digits = 3,
  ...)
```

Arguments

`x`	The loco object.
`test`	One of "z", "sign", "wilcox", or "all", describing which test results to show. If "z", then the results from only Z-tests are displayed; if "sign", then only sign tests; if "wilcox", then only Wilcoxon tests. If "all", then the results from all tests are displayed. Default is "wilcox".
`digits`	Number of digits to display. Default is 3.
`...`	Other arguments (currently not used).

print.sim *Print function for sim object.*

Description

Summary and print the results of a set of simulations, stored an object of class sim (produced by sim.master).

Usage

```
## S3 method for class 'sim'
print(x, type = c("err", "len", "all"), se = TRUE,
  digits = 3, ...)
```

Arguments

x	The sim object.
type	One of "all", "err", or "len" describing the type of summary to be used. If "err", then for each method, the tuning parameter value that gives the best average test error is selected, and the coverage and length properties at this tuning parameter value are report. If "len", then the analogous thing is done, but for the tuning parameter value that gives the best average length. If "all", then the average coverage, length, and test error results are shown across all tuning parameter values. Default is "err".
se	Should standard errors be displayed (in parantheses)? For each metric, this is defined as the standard deviation divided by the square root of the number of simulation repetitions. Default is TRUE.
digits	Number of digits to display. Default is 3.
...	Other arguments (currently not used).

print.tex *Print function for latex-style tables.*

Description

Print a given table in format digestable by latex.

Usage

```
## S3 method for class 'tex'
print(tab, tab.se = NULL, digits = 3, file = NULL,
  align = "l")
```

`rf.funs` *Random forests training and prediction functions.*

Description

Construct training and prediction functions for random forests.

Usage

```
rf.funs(ntree = 500, varfrac = 0.333, replace = TRUE,
  obsfrac = ifelse(replace, 1, 0.632), nodesize = 5)
```

Arguments

`ntree`	Number of trees to grow. Default is 500.
`varfrac`	Determines the number of variables used as candidates at each split: this is floor(varfrac * p), where p is the total number of variables. Default is 0.333.
`replace`	Should sampling of observations done with or without replacement? Default is FALSE.
`obsfrac`	Determines the number of observations used by each tree: this is floor(obsfrac * n), where n is the number of training observations. Default is 1 when replace is TRUE, and 0.632 when replace is FALSE.
`nodesize`	Sought size of terminal nodes, in growing each tree. Default is 5.

Details

This function is based on the packages `randomForest` and `plyr`. If these packages are not installed, then the function will abort.

Value

A list with three components: train.fun, predict.fun, active.fun. The third function is designed to take the output of train.fun, and reports which features are active for the fitted model contained in this output.

Author(s)

Ryan Tibshirani

Examples

```
## See examples for sam.funs function
```

`sam.funs` *Sparse additive models training and prediction functions.*

Description

Construct training and prediction functions for sparse additive models, based on the SAM package, over a sequence of (given or internally computed) lambda values.

Usage

```
sam.funs(m = 3, lambda = NULL, nlambda = 30, lambda.min.ratio = 0.005)
```

Arguments

`m`	Number of spline functions to be used in the additive expansion of each variable. Default is 3.
`lambda`	Sequence of lambda values over which training is performed. This must be in decreasing order, and — this argument should be used with caution! When used, it is usually best to grab the sequence constructed by one intial call to SAM (see examples). Default is NULL, which means that the nlambda, lambda.min.ratio arguments will define the lambda sequence (see next).
`nlambda`	Number of lambda values over which training is performed. In particular, the lambda sequence is defined by nlambda log-spaced values between lambda.max and lambda.min.ratio * lambda.max, where lambda.max is the smallest value of lambda at which the solution has all zero components, and lambda.min.ratio is a small fraction (see next). Default is 30.

Details

This function is based on the packages SAM and plyr. If these packages are not installed, then the function will abort.

Value

A list with three components: train.fun, predict.fun, active.fun. The third function is designed to take the output of train.fun, and reports which features are active for each fitted model contained in this output.

Author(s)

Ryan Tibshirani

Examples

```
## Sparse additive models: high-dimensional example

# Use the sim.xy function to get a data set with additive structure
set.seed(11)
n = 200; n0 = 200; p = 500; s = 10
obj = sim.xy(n+n0, p, mean.fun="additive", m=4, s=s, snr=3)
x = obj$x[1:n,]
y = obj$y[1:n]
x0 = obj$x[(n+1):(n+n0),]
y0 = obj$y[(n+1):(n+n0)]

# Grab a fixed lambda sequence from one call to samQL, then define the sparse
# additive models training and prediction functions
if (!require("SAM",quietly=TRUE)) {
  stop("Package SAM not installed (required for this example)!")
}
out.sam = samQL(x,y,nlambda=50,lambda.min.ratio=1e-5)
lambda = out.sam$lambda
my.sam.funs = sam.funs(m=5,lambda=lambda)

# Random forest training and prediction functions
if (!require("randomForest",quietly=TRUE)) {
  stop("Package randomForest not installed (required for this example)!")
}
my.rf.funs = rf.funs(ntree=500)

# Split conformal inference with sparse additive models and random forest
out.sam = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=my.sam.funs$train, predict.fun=my.sam.funs$predict)

out.rf = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=my.rf.funs$train, predict.fun=my.rf.funs$predict)

y0.mat = matrix(rep(y0,ncol(out.sam$lo)),nrow=n0)
cov.sam = colMeans(out.sam$lo <= y0.mat & y0.mat <= out.sam$up)
len.sam = colMeans(out.sam$up - out.sam$lo)
err.sam = colMeans((y0.mat - out.sam$pred)^2)

cov.rf = mean(out.rf$lo <= y0 & y0 <= out.rf$up)
len.rf = mean(out.rf$up - out.rf$lo)
err.rf = mean((y0 - out.rf$pred)^2)

# Plot average coverage
plot(log(lambda),cov.sam,type="l",ylim=c(0,1),
     xlab="log(lambda)",ylab="Avg coverage",
     main=paste0("Split conformal + Spam, RF:\nAverage coverage"))
points(log(lambda),cov.sam,pch=20)
abline(h=cov.rf,lty=2,col=4)
legend("bottomright",col=c(1,4),lty=1:2,
       legend=c("Spam","Random forest"))
```

```
# Plot average length
plot(log(lambda),len.sam,type="l",
     ylim=range(len.sam,len.rf),
     xlab="log(lambda)",ylab="Avg length",
     main=paste0("Split conformal + Spam, RF:\nAverage length"))
points(log(lambda),len.sam,pch=20)
abline(h=len.rf,lty=2,col=4)
legend("topleft",col=c(1,4),lty=1:2,
       legend=c("Spam","Random forest"))

# Plot test error
plot(log(lambda),err.sam,type="l",
     ylim=range(err.sam,err.rf),
     xlab="log(lambda)",ylab="Test error",
     main=paste0("Split conformal + Spam, RF:\nTest error"))
points(log(lambda),err.sam,pch=20)
abline(h=err.rf,lty=2,col=4)
legend("topleft",col=c(1,4),lty=1:2,
       legend=c("Spam","Random forest"))
```

`sim.master` *Master function for simulations.*

Description

Run a set of simulations with the specified configuration.

Usage

```
sim.master(n, p, conformal.pred.funs, n0 = n, in.sample = FALSE,
  nrep = 20, seed = NULL, verbose = FALSE, file = NULL, file.rep = 5,
  x.dist = c("normal", "binom", "sn", "mix"), cor = c("none", "pair",
  "auto", "rand"), rho = 0.5, k = 5, standardize = TRUE,
  mean.fun = c("linear", "additive", "rotated"), m = 4,
  sparsity = c("strong", "weak"), s = round(log(p)), weak.decay = 0.5,
  mu.seed = NULL, error.dist = c("normal", "unif", "t", "sn"), sigma = 1,
  snr = 1, sigma.type = c("const", "var"), df = 3, alpha = 5,
  omitted.vars = FALSE)
```

Arguments

`n, p`	Number of observations and features, respectively.
`conformal.pred.funs`	A list containing conformal inference functions to use. Each function here must take three arguments: x, y, x0 — and no more. These will typically be defined with a wrapper around, e.g., `conformal.pred` or `conformal.pred.split`. See examples below.
`n0`	The number of points at which to make predictions. Default is n.

`in.sample` Should the original x points be used for predictions (within each repetition)? If TRUE, then the n0 argument (above) is ignored. Default is FALSE.

`nrep` Number of repetitions over which to average results. Default is 20.

`seed` Seed to be set for the overall random number generation, i.e., set before repetitions are begun (for reproducibility of the simulation results). Default is NULL, which effectively sets no seed.

`verbose` Should intermediate progress be printed out? Default is FALSE.

`file, file.rep` Name of a file to which simulation results will be saved (using saveRDS), and a number of repetitions after which intermediate results will be saved. Setting file to NULL is interpreted to mean that no simulations results should be saved; setting file.rep to 0 is interpreted to mean that simulations results should be saved at the very end, i.e., no intermediate saving. Defaults are NULL and 5, respectively.

`x.dist, cor, rho, k, standardize, mean.fun, m, sparsity, s, weak.decay,` error.dist,sigma,snr,sigma.type,df,alpha,omitted.vars Arguments to pass to `sim.xy`, see the latter's help file for details.

Value

A list with the following components.

- ave.cov, ave.len, ave.err: lists of length m, where m denotes the number of conformal methods evaluated (i.e., the length of the list conformal.pred.funs). Each element of ave.cov corresponds to a conformal method considered, and is a vector whose length is equal to the number of tuning parameters internal to this method, containing the empirical coverages of conformal intervals, averaged across the repetitions. The lists ave.len and ave.err are analogous, except they contain the average lengths of conformal intervals, and average test errors, respectively.
- ave.opt, ave.tim: lists of length m, analogous to ave.cov, ave.len, and ave.err (see above), but where ave.opt contains the average (relative) optimism of the methods, a unitless measure of model complexity defined by (test error - training error) / (test error); and ave.tim contains the average runtimes (in seconds) of the methods.
- sd.cov, sd.len, sd.err, sd.opt, sd.tim: same as above, but containing deviations, rather than averages, across the reptitions.
- cov, len, err, opt, tim: lists of length m, reporting the full set of metrics across repetitions (rather than averages or standard deviations).
- ave.best.err, ave.best.len: matrices of dimension 5 x m, providing a summary of the average coverage, length, error, optimism, and time metrics (see description above), but at only one particular tuning parameter value for each method considered — this is the tuning parameter value that either gives the best average error, or best average length.
- sd.best.err, sd.best.len: matrices of dimension 5 x m, providing a summary of the standard deviation of coverage, length, error, optimism, and time metrics for each method's best tuning parameter value (analogous to the construction of ave.best.err and ave.best.len).

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

See Also

sim.x, sim.mu, sim.y

Examples

```
## Linear regression: test out conformal intervals and parametric intervals
## across a variety of settings

## Not run:

# Set some overall simulation parameters
n = 100; p = 10 # Numbers of observations and features
s = 10 # Number of truly relevant features
n0 = 100 # Number of points at which to make predictions
nrep = 50 # Number of repetitions for a given setting
sigma = 1 # Marginal error standard deviation
snr = 1 # Signal-to-noise ratio
lambda = 0 # Lambda values to try in ridge regression
alpha = 0.1 # Miscoverage level

# Define conformal inference functions: these are basically just wrappers
# around a particular instatiation of conformal.pred, conformal.pred.jack, or
# conformal.pred.split
my.lm.funs = lm.funs(lambda=lambda)
my.conf.fun = function(x, y, x0) {
  conformal.pred(x,y,x0,alpha=alpha,verb="\t\t",
                 train.fun=my.lm.funs$train,
                 predict.fun=my.lm.funs$predict)
}
my.jack.fun = function(x, y, x0) {
  conformal.pred.jack(x,y,x0,alpha=alpha,verb="\t\t",
                      train.fun=my.lm.funs$train,
                      predict.fun=my.lm.funs$predict,
                      special.fun=my.lm.funs$special)
}
my.split.fun = function(x, y, x0) {
  conformal.pred.split(x,y,x0,alpha=alpha,
                       train.fun=my.lm.funs$train,
                       predict.fun=my.lm.funs$predict)
}

# Hack together our own "conformal" inference function, really, just one that
# returns the parametric intervals
my.param.fun = function(x, y, x0) {
  n = nrow(x); n0 = nrow(x0)
  out = my.lm.funs$train(x,y)
  fit = matrix(my.lm.funs$predict(out,x),nrow=n)
```

```
  pred = matrix(my.lm.funs$predict(out,x0),nrow=n0)
  m = ncol(pred)

  x1 = cbind(rep(1,n0),x0)
  z = qnorm(1-alpha/2)
  lo = up = matrix(0,n0,m)

  for (j in 1:m) {
    sig.hat = sqrt(sum((y - fit[,j])^2)/(n-ncol(x1)))
    g = diag(x1 %*% chol.solve(out$chol.R[[j]], t(x1)))
    lo[,j] = pred[,j] - sqrt(1+g)*sig.hat*z
    up[,j] = pred[,j] + sqrt(1+g)*sig.hat*z
  }

  # Return proper outputs in proper formatting
  return(list(pred=pred,lo=lo,up=up,fit=fit))
}

# Now put together a list with all of our conformal inference functions
conformal.pred.funs = list(my.conf.fun, my.jack.fun, my.split.fun, my.param.fun)
names(conformal.pred.funs) = c("Conformal","Jackknife","Split conformal",
       "Parametric")

# Set some overall simulation parameters
n = 100; p = 10 # Numbers of observations and features
s = 10 # Number of truly relevant features
n0 = 100 # Number of points at which to make predictions
nrep = 50 # Number of repetitions for a given setting
sigma = 1 # Marginal error standard deviation
alpha = 0.1 # Miscoverage level

# Define conformal inference functions: these are basically just wrappers
# around a particular instatiation of conformal.pred or conformal.pred.split,
# i.e., particular training and prediction functions
my.lm.funs = lm.funs()
my.conf.fun = function(x, y, x0) {
  conformal.pred(x,y,x0,alpha=alpha,verb="\t\t",
                 train.fun=my.lm.funs$train,
                 predict.fun=my.lm.funs$predict)
}
my.split.fun = function(x, y, x0) {
  conformal.pred.split(x,y,x0,alpha=alpha,
                       train.fun=my.lm.funs$train,
                       predict.fun=my.lm.funs$predict)
}

# Hack together our own "conformal" inference function, really, just one that
# returns the parametric intervals
my.param.fun = function(x, y, x0) {
  lm.obj = lm(y~x)
  mat = predict(lm.obj,list(x=x0),interval="predict",level=1-alpha)
  # Return proper outputs in proper formatting
  return(list(pred=mat[,"fit",drop=F],lo=mat[,"lwr",drop=F],
```

```
              up=mat[,"upr",drop=F],fit=matrix(lm.obj$fit,ncol=1)))
}

# Now put together a list with our three conformal inference functions
conformal.pred.funs = list(my.conf.fun, my.split.fun, my.param.fun)
names(conformal.pred.funs) = c("Conformal","Split conformal","Parametric")

# Set some path, where we can save the rds files containing sim results
path = "."

# Simulation setting A: classical setup --- linear (fully supported) mean
# function; normal, homoskedastic errors; and normal, uncorrelated features
cat("=======================================\n")
cat("               SETTING A               \n")
cat("=======================================\n")
simA = sim.master(n, p, conformal.pred.funs, n0=n0, nrep=nrep, verb=TRUE,
  file=paste0(path,"simA.rds"), x.dist="normal", cor="none",
  mean.fun="linear", s=s, error.dist="normal", sigma=sigma, snr=snr,
  sigma.type="const")

# Simulation setting B: now, nonlinear mean, and t2 errors
cat("=======================================\n")
cat("               SETTING B               \n")
cat("=======================================\n")
simB = sim.master(n, p, conformal.pred.funs, n0=n0, nrep=nrep, verb=TRUE,
  file=paste0(path,"simB.rds"), x.dist="normal", cor="none",
  mean.fun="additive", m=4, s=s, error.dist="t", df=2, sigma=sigma, snr=snr,
  sigma.type="const")

# Simulation setting C: now, nonlinear mean, heteroskedastic t2 errors, and
# auto-correlated mix of x variables
cat("=======================================\n")
cat("               SETTING C               \n")
cat("=======================================\n")
simC = sim.master(n, p, conformal.pred.funs, n0=n0, nrep=nrep, verb=TRUE,
  file=paste0(path,"simC.rds"), x.dist="mix", cor="auto", k=5,
  mean.fun="linear", s=s, error.dist="t", df=2, sigma=sigma, snr=snr,
  sigma.type="var")

## End(Not run)
```

`sim.xy` *Features and responses generation.*

Description

Generate a feature matrix x, and response vector y, following a specified setup.

Usage

```
sim.xy(n, p, x.dist = c("normal", "binom", "sn", "mix"), cor = c("none",
  "pair", "auto", "rand"), rho = 0.5, k = 5, standardize = TRUE,
  mean.fun = c("linear", "additive", "rotated"), m = 4,
  sparsity = c("strong", "weak"), s = round(log(p)), weak.decay = 0.5,
  mu.seed = NULL, error.dist = c("normal", "unif", "t", "sn"), sigma = 1,
  snr = 1, sigma.type = c("const", "var"), df = 3, alpha = 5,
  omitted.vars = FALSE)
```

Arguments

n, p	The dimensions, as in n x p, of the feature matrix x. Note that n is also the length of the response vector y.
x.dist	One of "normal", "binom", "sn", "mix", indicating the distribution for the entries of x. In the first three cases, the columns of x are filled out with i.i.d. normal, standard binomial, or skewed normal (with skewness parameter alpha = 5) draws, respectively. The third option requiring the sn package.In the last case, "mix", each column of x is drawn with equal probability from the above three distributions. Default is "normal".
cor	One of "none", "pair", "auto", "rand", indicating the type of correlation among columns of x. If "pair", then x is post-multiplied by the symmetric square root of the p x p matrix whose diagonals are 1 and off diagonals are rho (see parameter below). If "auto", then each column of x is taken to be a random convex combination of its old value and the previous k (see parameter) columns. If "rand", then each column of x is taken to be a random convex combination of its old value and a randomly k other columns. Default is "none".
rho	If cor is "pair", then this specifies the pairwise correlation. Default is 0.5.
k	If cor is "auto" or "rand", then this specifies the number of other columns that are used in the lagged or random convex combinations (see above).
standardize	Should the columns of x be standardized (i.e., centered, and scaled to have variance 1)? Default is TRUE.
mean.fun	One of "linear", "additive", or "rotated", indicating the functional dependence of the mean on the variables. If "additive", then the mean vector mu is constructed to be an additive function of B-splines of the variables, where each variable is expanded into df B-splines (with knots spaced evenly over the range of observed measurements of the variable). If "rotated", then mu is set to be randomly rotated version of such an additive function. Default is "linear".
m	Number of B-splines per variable in the "additive" or "rotated" cases for mean.fun; default is 4. Ignored when mean.fun is "linear".
sparsity	Either "strong" or weak". In the former case, s (see below) randomly chosen variables of the p total contribute to mu, with coefficients randomly set to +1 or -1. In the latter case, still s variable contribute to mu, as before, but now the remaining p-s variables contribute in a weak sense, in that their coefficients are randomly set to +1 or -1 times a successively increasing power of the weak.decay factor (see below). In the "additive" or "rotated" cases for mean.fun, the above should be interpreted in the appropriate groupwise sense,

e.g., s groups of spline functions are randomly chosen, each group corresponding to the expansion of a single variable.

`s` Number indicating the level of sparsity. Note that the choice s = p would correspond to a fully dense model. Default is round(log(p)).

`weak.decay` Number controlling the decay of the weak coefficients, in the case sparsity is "weak", as their magnitudes get set to a successive power of weak.decay. Default is 0.5.

`mu.seed` Seed to be set before generation of the mean vector mu (ensures that the mean is the same, over multiple calls to this function). Default is NULL, which effectively sets no seed.

`error.dist` One of "normal", "unif", "t", "sn", indicating the error distribution, where the last option "sn" is for the skewed normal, and requires the `sn` package. Default is "normal".

`sigma` Standard deviation of errors. When the error distribution does not have a second moment (i.e., t-distribution with 1 or 2 df), then sigma represents the MAD. Default is 1; when an snr argument is given (see next), the sigma argument is ignored, and the latter is used to specify the error variance.

`snr` Number controlling the signal-to-noise ratio. Specifically, the coefficients in the model for mu as linear function of x (or as a linear function of B-splines of x in the "additive" and "rotated" cases for mean.fun) are assigned to be +/- snr, with equal probability. Default is 1.

`sigma.type` Either "const" or "var", indicating constant or varying error variance. In the latter case, the error standard deviation sigma is multiplied by 1 + abs(mu)/mean(abs(mu)) (this gives a vector of the same length as mu, i.e., a different standard deviation per component).

`df` If error.dist is "t", then this specifies the degrees of freedom of the t-distribution for the errors. Default is 3.

`alpha` If error.dist is "sn", then this specifies the skewness parameter (i.e., shape parameter) of the skewed normal distribution for the errors. The location and scale parameters are set so that the resulting distribution always has mean zero and variance 1. Note that alpha > 0 means right-skewed and alpha < 0 means left-skewed. Default is 5.

`omitted.vars` Should important (active) variables be omitted from x? Induces correlation between the x and the error terms.

Details

In the case of an "additive" or "rotated" mean, this function relies on the packages `plyr` and `splines`. If these packages are not installed, then the function will abort.

Value

A list with the following components: x, y, and mu.

Author(s)

Ryan Tibshirani

References

"Distribution-Free Predictive Inference for Regression" by Max G'Sell, Jing Lei, Alessandro Rinaldo, Ryan Tibshirani, Larry Wasserman, http://arxiv.org/pdf/xxxx.pdf, 2016.

See Also

sim.mu, sim.y

Examples

```
## See examples for sam.funs function
```

---

smooth.spline.funs *Smoothing spline training and prediction functions, and special leave-one-out fitting function.*

Description

Construct the training and prediction functions for smoothing splines, at a particular tuning parameter value or df value (either given or chosen by cross-validation), and construct also a special function for computing leave-one-out fitted values. Based on the smooth.spline function.

Usage

```
smooth.spline.funs(df = NULL, spar = NULL, cv = TRUE, tol.fact = 1e-06)
```

Arguments

df	Desired effective degrees of freedom of the smoothing spline fit. If NULL, then the complexity of the fit controlled either the spar or cv arguments (see below).
spar	Parameter that scales roughly as log(lambda), where lambda is the parameter multiplying the roughness penalty in the smoothing spline criterion. See the smooth.spline help file for details. If NULL, then the complexity of the fit is chosen by the cv argument (see next).
cv	Either TRUE, indicating that leave-one-out cross-validation should be used to choose the smoothing parameter lambda, or FALSE, indicating that generalized cross-validation should be used to choose lamdbda. Note that this argument is only in effect when both df and spar are NULL.
tol.fact	Factor multiplying the interquartile range of the x values, IQR(x), to determine a threshold for uniqueness: the x values are binned into bins of size tol.fact * IQR(x), and values which fall into the same bin are regarded as the same. Default is 1e-6.

Details

Note that the constructed function special.fun only serve as an approximation to the exact leave-one-out fits when the smooth.spline function merges nearby adjacent x points (which happens when adjacent x points are closer than tol), since in this case the smooth.spline function only returns the leverage scores (diagonal elements of the smoother matrix) on the reduced (merged) data set. Another important note is that the constructed special.fun will be formally invalid when cross-validation (or generalized cross-validation) is used to pick the smooth parameter. In both of these cases, special.fun does *not* throw a warning, and it is left up to the user to consider using this function wisely.

Value

A list with four components: train.fun, predict.fun, special.fun, active.fun. The last function is designed to take the output of train.fun, and reports which features are active for each fitted model contained in this output. Trivially, here, there is only one feature and it is always active.

Author(s)

Ryan Tibshirani

Examples

```
## Smoothing splines: some simple univariate examples

# Generate some example training data, clearly heteroskedastic
set.seed(33)
n = 1000
x = runif(n,0,2*pi)
y = sin(x) + x*pi/30*rnorm(n)

# Generate some example test data
n0 = 1000
x0 = runif(n,0,2*pi)
y0 = sin(x0) + x0*pi/30*rnorm(n0)

# Smoothing spline training and prediction functions, where the smoothing
# parameter is chosen via cross-validation
funs = smooth.spline.funs(cv=TRUE)

# Jackknife conformal for out-of-sample prediction intervals (valid on average
# over new x0 values), we can use the special leave-one-out fitting function
out.jack = conformal.pred.jack(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict, special.fun=funs$special)

plot(c(), c(), xlab="x0", ylab="y0", xlim=range(x0),
     ylim=range(c(y0,out.jack$lo,out.jack$up)), col="white",
     main=paste0("Jackknife (out-of-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",mean(out.jack$up-out.jack$lo))))
o = order(x0)
segments(x0[o], out.jack$lo[o], x0[o], out.jack$up[o], col="pink")
lines(x0[o], out.jack$pred[o], lwd=2, col="red")
```

```
points(x0, y0, col="gray50")

# Split conformal for out-of-sample prediction intervals (valid on average over
# new x0 values)
out.split = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

plot(c(), c(), xlab="x0", ylab="y0", xlim=range(x0),
     ylim=range(c(y0,out.split$lo,out.split$up)), col="white",
     main=paste0("Split conformal (out-of-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",mean(out.split$up-out.split$lo))))
o = order(x0)
segments(x0[o], out.split$lo[o], x0[o], out.split$up[o], col="lightgreen")
lines(x0[o], out.split$pred[o], lwd=2, col="darkgreen")
points(x0, y0, col="gray50")

# Rank-one-out split conformal for in-sample prediction intervals (valid on
# averaged over observed x values)
out.roo = conformal.pred.roo(x, y, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict)

plot(c(), c(), xlab="x", ylab="y", xlim=range(x),
     ylim=range(c(y,out.roo$lo,out.roo$up)), col="white",
     main=paste0("ROO-conformal (in-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",mean(out.roo$up-out.roo$lo))))
o = order(x)
segments(x[o], out.roo$lo[o], x[o], out.roo$up[o], col="lightblue")
lines(x[o], out.roo$fit.all[o], lwd=2, col="blue")
points(x, y, col="gray50")

## Adjust now for heteroskedasticity, by using mad.train.fun and mad.predict.fun

# Jackknife conformal is much slower to run, because we can't use the special
# leave-one-out fitting function anymore
out.jack.local = conformal.pred.jack(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict,
  mad.train.fun=funs$train, mad.predict.fun=funs$predict, verb=TRUE)

plot(c(), c(), xlab="x0", ylab="y0", xlim=range(x0),
     ylim=range(c(y0,out.jack.local$lo,out.jack.local$up)), col="white",
     main=paste0("Jackknife (out-of-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",
               mean(out.jack.local$up-out.jack.local$lo))))
o = order(x0)
segments(x0[o], out.jack.local$lo[o], x0[o], out.jack.local$up[o], col="pink")
lines(x0[o], out.jack.local$pred[o], lwd=2, col="red")
points(x0, y0, col="gray50")

# Split conformal, also using smooth splines (under CV) to train on residuals
out.split.local = conformal.pred.split(x, y, x0, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict,
  mad.train.fun=funs$train, mad.predict.fun=funs$predict)
```

```
plot(c(), c(), xlab="x0", ylab="y0", xlim=range(x0),
     ylim=range(c(y0,out.split.local$lo,out.split.local$up)), col="white",
     main=paste0("Split conformal (out-of-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",
               mean(out.split.local$up-out.split.local$lo))))
o = order(x0)
segments(x0[o], out.split.local$lo[o], x0[o], out.split.local$up[o],
         col="lightgreen")
lines(x0[o], out.split.local$pred[o], lwd=2, col="darkgreen")
points(x0, y0, col="gray50")

# Rank-one-out split conformal, again using smoothing splines (under CV) to
# train on residuals
out.roo.local = conformal.pred.roo(x, y, alpha=0.1,
  train.fun=funs$train, predict.fun=funs$predict,
  mad.train.fun=funs$train, mad.predict.fun=funs$predict)

plot(c(), c(), xlab="x", ylab="y", xlim=range(x),
     ylim=range(c(y,out.roo.local$lo,out.roo.local$up)), col="white",
     main=paste0("ROO-conformal (in-sample) prediction intervals\n",
       sprintf("Average length: %0.3f",
               mean(out.roo.local$up-out.roo.local$lo))))
o = order(x)
segments(x[o], out.roo.local$lo[o], x[o], out.roo.local$up[o], col="lightblue")
lines(x[o], out.roo.local$fit.all[o], lwd=2, col="blue")
points(x, y, col="gray50")

# The reason the local from rank-one-out method here don't look entirely smooth
# is because they're actually a mishmash of two separate split procedures on
# each half of the data set (so they'd look smooth over the points within each
# half, but not necessarily overall)
```

Index