Using stepreg

Walter K. Kremers, Mayo Clinic, Rochester MN

28 July 2023

The Package

When fitting lasso models we wanted to compare these to standard stepwise regression models. Keeping a more modern approach we tune by either number of terms included in the model or by the p critical value for model inclusion (James, Witten, Hastie and Tibshirani, An Introduction to Statistical Learning with applications in R, 2nd ed., Springer, New York, 2021).

When fitting lasso models we often use one-hot coding for predictor factors when setting up the design matrix. This allows lasso to identify and add to the model a term for any one group that might be particularly different from the others. By the penalty lasso stabilizes the model coefficients and keeps them form going to infinity, while ridge will generally uniquely identify coefficients despite any strict collinearities.

Before writing this program we tried different available packages to fit stepwise models for the Cox repression framework but all we tried had difficulties with numerical stability for the large and wide clinical datasets we were working with, and which involved one-hot coding. There may well be a package that would be stable for the data we were analyzing but we decided to write this small function to be able to tune for stability.

The stepreg() and cv.stepreg() funcitons in the *glmnetr* package were written for convenience and stability as opposed to speed or broad applicability. The main goal is to use the stepwise procedure on occasion as a reference for the lasso models. For many clinical datasets the lasso clearly outperformed the stepwise procedure, and ran much faster. For some simulated data sets with simplified covariance structures, i.e. independence of the underlying predictors, the lasso did not appear to do much better than the stepwise procedure tuned by number of model terms or p.

Data requirements

The data requirements for stepreg() and cv.stepreg() are similar to those of cv.glmnetr() and we refer to the *Using glmnetr* vignette for a description.

An example dataset

To demonstrate usage of *cv.stepreg* we first generate a data set for analysis, run an analysis and evaluate. Following the *Using glmnetr* vignette, the code

```
# Simulate data for use in an example survival model fit
# first, optionally, assign a seed for random number generation to get replicable results
set.seed(116291950)
simdata=glmnetr.simdata(nrows=1000, ncols=100, beta=NULL)
```

generates simulated data for analysis. We extract data in the format required for input to the *cv.stepreg* (and *glmnetr*) programs.

```
# Extract simulated survival data
xs = simdata$xs         # matrix of predictors
y_ = simdata$yt         # vector of survival times
event = simdata$event   # indicator of event vs. censoring
```

Inspecting the predictor matrix we see

```
# Check the sample size and number of predictors
print(dim(xs))
```

```
## [1] 1000  100
```

```
# Check the rank of the design matrix, i.e. the degrees of freedom in the predictors
rankMatrix(xs)[[1]]
```

```
## [1] 94
```

```
# Inspect the first few rows and some select columns
print(xs[1:10,c(1:12,18:20)])
```

```
##       X1 X2 X3 X4 X5 X6 X7 X8 X9 X10 X11 X12         X18         X19
##  [1,]  1  0  0  0  1  0  1  0  0   0   0   0 -1.20889802  0.05697053
##  [2,]  1  1  0  0  0  0  0  0  1   0   0   0  0.39535407  0.42731333
##  [3,]  1  0  0  1  0  1  0  0  0   0   0   0  1.04460756 -0.74696019
##  [4,]  1  1  0  0  0  0  0  1  0   0   0   0  0.02885863 -1.27765142
##  [5,]  1  0  0  1  0  1  0  0  0   0   0   0 -1.20517197 -1.28745400
##  [6,]  1  0  0  0  1  0  1  0  0   0   0   0 -1.15820960 -0.06884111
##  [7,]  1  0  0  0  1  0  0  0  1   0   0   0  0.15171338  1.09539635
##  [8,]  1  0  0  1  0  0  1  0  0   0   0   0 -0.13924591 -0.42455023
##  [9,]  1  1  0  0  0  0  0  1  0   0   0   0 -0.06932632  0.17279181
## [10,]  1  0  0  1  0  0  1  0  0   0   0   0  0.67742010  1.18594584
##               X20
##  [1,] -0.56563100
##  [2,]  0.18523531
##  [3,]  0.96427444
##  [4,]  0.20324327
##  [5,] -1.69822884
##  [6,]  1.45879996
##  [7,]  1.47683112
##  [8,]  0.07333966
##  [9,]  1.03965647
## [10,] -1.47355053
```

Cross validation (CV) informed stepwise model fit

To fit stepwise regression models where the number of model terms are informed by cross validation to select df, the number of model terms, and p, the entry threshold, one can use the function cv.stepreg() function.

```r
# Fit a stepwise regression model informed by cross validation
# cv.stepwise.fit=suppressWarnings(cv.stepreg(xs,NULL,y_,event,family="cox",folds_n=5,
#    steps_n=30,track=0))
cv.stepwise.fit = cv.stepreg(xs,NULL,y_,event,family="cox",folds_n=5,steps_n=30,track=0)
```

Note, in the derivation of the stepwise regression models, individual coefficients may be unstable even when the model may be stable which elicits warning messages. Thus we "wrapped" the call to cv.stepreg() within the suppressWarnings() function to suppress excessive warning messages in this vignette. The first term in the call to cv.stepreg(), xs, is the design matrix for predictors. The second input term, here NULL, is for the start time in case (start, stop) time data setup is used in a Cox survival model. The third term is the outcome variable for the linear regression or logistic regression model and the time of event or censoring in case of the Cox model, and finally the forth term is the event indicator variable for the Cox model taking the value 1 in case of an event or 0 in case of censoring at time y_. The forth term would be NULL for either linear or logistic regression. Currently the options for family are "guassian" for linear regression, "binomial" for logistic regression (both using the *stats* glm() function) and "cox" for the Cox proportional hazards regression model using the coxph() function of the R *survival* package. If one sets track=1 the program will update progress in the R console. For track=0 it will not. To summarize the model fit and inspect the coefficient estimates we use the summary() function.

```r
# summarize model fit ...
summary(cv.stepwise.fit)
```

```
##
##  CV best df = 13, CV best p enter = 0.01 for 16 predictors
##      in the full data model, from 100 candidate predictors
##
##   df loglik.null    loglik       pvalue concordance         std        X2
## 1 13   -3726.413 -3718.864 0.0001020687   0.8777104 0.005278002 -2.477179
## 2 16   -3709.825 -3705.723 0.0041783660   0.8796415 0.005219351 -2.544254
##           X3         X7        X10        X11       X12       X14       X16
## 1 -0.3557345  0.0000000 0.6428104 -0.4952736 0.0000000 -1.051699 -2.190889
## 2 -0.4123862 -0.5812514 0.6538633 -0.4939628 0.4246715 -1.387424 -1.647604
##         X18       X19        X20        X21      X23       X24        X25
## 1 0.7719787 -1.128058 -0.4775669 -0.1921267 1.057341 0.7013581 -0.4682348
## 2 0.7966722 -1.150425 -0.4928893 -0.1818494 1.075441 0.7174526 -0.4877742
##          X62
## 1  0.0000000
## 2 -0.1259569
```

To extract beta's or calculate predicteds we use the predict() function.

```r
# get betas  ...
betas = predict(cv.stepwise.fit)
t( betas[1:20,] )
```

```
##    X1        X2         X3 X4 X5 X6         X7 X8 X9       X10        X11
## df  0 -2.477179 -0.3557345  0  0  0  0.0000000  0  0 0.6428104 -0.4952736
## p   0 -2.544254 -0.4123862  0  0  0 -0.5812514  0  0 0.6538633 -0.4939628
##          X12 X13       X14 X15       X16 X17       X18       X19        X20
## df 0.0000000   0 -1.051699   0 -2.190889   0 0.7719787 -1.128058 -0.4775669
## p  0.4246715   0 -1.387424   0 -1.647604   0 0.7966722 -1.150425 -0.4928893
```

```r
# predicteds ...
preds = predict(cv.stepwise.fit, xs)
t( preds[1:14,] )
```

```
##          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## df -4.131092 -2.554348 -1.851470 -1.008533 0.1122397 -4.722600 -1.033689
## p  -4.652185 -2.777916 -1.515435 -0.979273 0.3337369 -5.318352 -1.121909
##          [,8]      [,9]     [,10]      [,11]     [,12]    [,13]     [,14]
## df -2.501702 -2.455804 -4.196633 -0.9470070 -4.034808 5.359022 -3.474064
## p  -2.543347 -2.617922 -4.385983 -0.4020953 -4.200559 5.430460 -3.462096
```

Nested cross validation

Because the values for lambda and gamma informed by CV are specifically chosen to give a best fit, model fit statistics for the CV derived model will be biased. To address this one can perform a CV on the CV derived estimates, that is a nested cross validation as argued for in SRDM (Simon R, Radmacher MD, Dobbin K, McShane LM. Pitfalls in the Use of DNA Microarray Data for Diagnostic and Prognostic Classification. J Natl Cancer Inst (2003) 95 (1): 14-18. https://academic.oup.com/jnci/article/95/1/14/2520188). This is done here by the nested.glmnetr() function.

```r
# A nested cross validation to evaluate a cross validation informed stepwise fit
#nested.cox.fit = suppressWarnings(nested.glmnetr(xs,NULL,y_,event,family="cox",
#                                  dostep=1,doaic=1,folds_n=3,steps_n=30,track=0))
y_ = simdata$y_
nested.gau.fit = suppressWarnings(nested.glmnetr(xs,NULL,y_,NULL,family="gaussian",
                                  dostep=1,doaic=1,folds_n=3,steps_n=30,track=0))
summary(nested.gau.fit)
```

```
##
##  Sample information including number of records, number of columns in
##  design (predictor, X) matrix, and df (rank) of design matrix:
##      family            n   xs.columns        xs.df null.dev/obs
##   "gaussian"       "1000"        "100"         "94"      "7.963"
##
##  Tuning parameters for models :
## steps_n folds_n
##    "30"     "3"
##
##
##  Nested Cross Validation averages for LASSO (1se and min), Relaxed LASSO, and gamma=0 LASSO :
##
##      deviance per record :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##   1.049   1.028   1.035   1.018   1.045   1.025   1.128
##
##      deviance per record (linerly calibrated) :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##   1.014   1.015   1.020   1.014   1.043   1.023   1.091
##
##      number of nonzero model terms :
##     1se     min    1seR    minR 1seR.G0 minR.G0
```

```
##    28.3    43.3    20.0    26.3    12.3    14.0
##
##       linear calibration coefficient :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##   1.074   1.040   1.037   1.009   0.999   0.997   1.069
##
##       agreement (r-square) :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##   0.873   0.872   0.872   0.873   0.869   0.871   0.863
##
##  Naive agreement for cross validation informed LASSO :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##   0.874   0.883   0.868   0.875   0.878   0.888   0.892
##
##  Number of non-zero terms in cross validation informed LASSO :
##     1se     min    1seR    minR 1seR.G0 minR.G0   ridge
##      17      34      12      14      12      14      99
##
##
##  Nested Cross Validation STEPWISE regression model (df):
##       Average linear calibration coefficient: 0.996
##       Average deviance : 1.041
##       Average model df : 13
##       R-square         : 0.869
##  Naive R-square based upon the same (all) data as model derivation (df): 0.877
##     Model df 14
##
##  Nested Cross Validation STEPWISE regression model (p):
##       Average linear calibration coefficient p : 0.996
##       Average deviance : 1.032
##       Average model p  : 0.033
##       Average model df : 18
##       R-square         : 0.871
##  Naive R-square based upon the same (all) data as model derivation (p): 0.881
##     Model df 18
##
##  Cross Validation results for STEPWISE regression model: (AIC)
##       Average linear calibration coefficient : 0.974
##       Average deviance : 1.089
##       Average model df : 29
##       Concordance      : 0.864
##  Naive R-square based upon the same (all) data as model derivation (AIC) : 0.888
##     Model df 31
```

```
#names(nested.gau.fit)
```

For this example we use only 3 folds, instead of 5 or 10 like we would typically use in practice, to get reasonable run times as this is just for the purpose of demonstration.
| Before providing analysis results the output first reports sample size, and if this were for a Cox regression the number of events too would be described, followed by the number of predictors and the df (degrees of freedom) of the design matrix, as well as some information on "Tuning parameters" to compare the lasso model with stepwise procedures as described in JWHT (James, Witten, Hastie and Tibshirani, An Introduction to Statistical Learning with applications in R, Springer, New York, 2021). In general we have found in practice that the lasso performs better.

| Next are the nested cross validation results. First are the per record (or per event in case of the Cox model) log-likelihoods (or partial likelihoods) which reflect the amount of information in each observation (event). Since we are not using large sample theory to base inferences we feel the per record are more intuitive, and they allow comparisons between datasets with unequal sample sizes. Next are the average number of model terms which reflect the complexity of the different models, even if in a naive sense, followed by the agreement statistics, concordance or r-square. These nested cross validated concordances should be essentially unbiased for the given design, unlike the naive concordances where the same data are used to derive the model and calculate the concordances (see SRDM). | In addition to evaluating the CV informed model fits using another layer of CV, the nested.glmnetr() function also fits CV informed models based upon the whole data set. Here we see, not unexpectedly, that the concordances estimated from the nested CV are slightly smaller than the concordances naively calculated using the original dataset. Depending on the data the nested CV and naive agreement measures can be very similar or disparate.
| Fit information for the CV fit can be gotten by extracting the object$cv.stepreg.fit object and calling the summary() and predict() functions.

```
# Summary of a CV model fit from a nested CV output object
summary(nested.gau.fit$cv.stepreg.fit)
```

```
##
##   CV best df = 13, CV best p enter = 0.01 for 17 predictors
##       in the full data model, from 100 candidate predictors
##
##    df loglik.null    loglik       pvalue   rsquare rsquareadj      Int        X2
## 1 13   -2456.327 -1408.878 0.0002011804 0.8769172  0.8752944 2.539021 -2.218679
## 2 17   -2456.327 -1390.572 0.0071320243 0.8813421  0.8792879 2.535310 -2.368534
##            X3        X4        X6        X8       X10       X12       X14
## 1  0.0000000 0.5328523 0.5132137 0.0000000 0.6473428 0.0000000 -1.604049
## 2 -0.3005069 0.4101063 0.5151283 0.2443046 0.7498331 0.4039423 -1.628451
##          X16       X18       X19        X20        X21      X23       X24
## 1 -1.878449 0.8879417 -1.105587 -0.5419426 -0.1331026 1.086596 0.6867091
## 2 -1.749086 0.8947635 -1.105268 -0.5422141 -0.1266067 1.089848 0.6950041
##           X25         X62
## 1 -0.4403300  0.00000000
## 2 -0.4370275 -0.08718791
```

```
# get betas  ...
betas = predict(nested.gau.fit$cv.stepreg.fit)
t( betas[1:10,] )
```

```
##          Int X1         X2         X3        X4 X5        X6 X7        X8 X9
## df 2.539021  0 -2.218679  0.0000000 0.5328523  0 0.5132137  0 0.0000000  0
## p  2.535310  0 -2.368534 -0.3005069 0.4101063  0 0.5151283  0 0.2443046  0
```

```
# get predicteds ...
preds = predict(nested.gau.fit$cv.stepreg.fit,xs)
t( preds[1:8,] )
```

```
##          [,1]       [,2]     [,3]     [,4]     [,5]      [,6]     [,7]      [,8]
## df -2.011860  0.1284216 1.511165 1.706261 3.432633 -2.702739 1.578755 0.8751885
## p  -2.018274 -0.1322521 1.571207 1.851558 3.365406 -2.724134 1.559432 0.8872069
```