Relaxed fits and other additions in `glmnet` 3.0

Trevor Hastie, Balasubramanian Narasimhan and Rob Tibshirani

October 15, 2019

Contents

Introduction

In our vignette "glmnet" we give details for fitting lasso and elastic-net regularized models, for CV and various aspects of glmnet modeling. In this vignette, we highlight some of the new tools and features in the major revision glmnet 3.0.

The main edition is the introduction of the *relaxed lasso*. The idea is to take a glmnet fitted object, and then for each lambda, refit the variables in the active set without any penalization. This gives the `relaxed` fit (note, there have been other definitions of a relaxed fit, but this is the one we prefer). This could of course be done for elastic net fits as well as lasso. However, if the number of variables gets too close to the sample size N, the relaxed path will be truncated. Furthermore, for binomial and other nonlinear GLMs convergence can be an issue with our current implementation if the number of variables is too large, and perversely if the relaxed fit is too strong.

Suppose the `glmnet` fitted linear predictor at λ is $\hat{\eta}_\lambda(x)$ and the relaxed version is $\tilde{\eta}_\lambda(x)$. We also allow for shrinkage between the two:

$$\tilde{\eta}_{\lambda,\gamma} = (1-\gamma)\tilde{\eta}_\lambda(x) + \gamma\hat{\eta}_\lambda(x).$$

$\gamma \in [0,1]$ is an additional tuning parameter which can be selected by cross validation.

The debiasing will potentially improve prediction performance, and CV will typically select a model with a smaller number of variables. This procedure is very competitive with forward-stepwise and best-subset regression, and has a considerable speed advantage when the number of variables is large. This is especially true for best-subset, but even so for forward stepwise. The latter has to plod through the variables one-at-a-time, while glmnet will just plunge in and find a good active set.

Further details may be found in Friedman, Hastie, and Tibshirani (2010), Simon et al. (2011), Tibshirani et al. (2012), Simon, Friedman, and Hastie (2013) and Hastie, Tibshirani, and Tibshirani (2017).

Simple relaxed fit

To get things going, we show the most basic use. We use the same data used in the `glmnet` vignette.

```
library(glmnet)
data(QuickStartExample)
fit=glmnet(x,y, relax=TRUE)
print(fit)
```

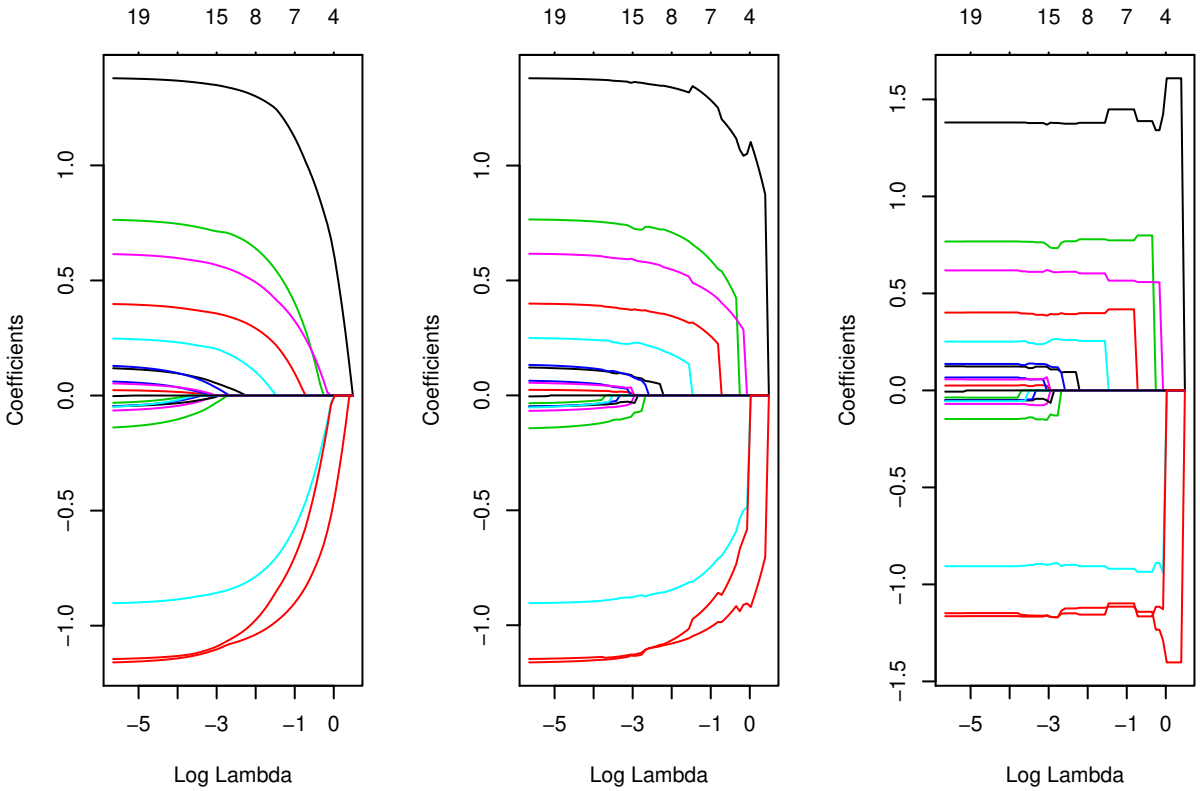
```
##
## Call:  glmnet(x = x, y = y, relax = TRUE)
## Relaxed
##
##    Df    %Dev %Dev R  Lambda
## 1   0 0.00000 0.0000 1.63100
## 2   2 0.05528 0.5890 1.48600
## 3   2 0.14590 0.5890 1.35400
## 4   2 0.22110 0.5890 1.23400
## 5   2 0.28360 0.5890 1.12400
## 6   2 0.33540 0.5890 1.02400
## 7   4 0.39040 0.7656 0.93320
## 8   5 0.45600 0.8059 0.85030
## 9   5 0.51540 0.8059 0.77470
## 10  6 0.57350 0.8799 0.70590
## 11  6 0.62550 0.8799 0.64320
## 12  6 0.66870 0.8799 0.58610
## 13  6 0.70460 0.8799 0.53400
## 14  6 0.73440 0.8799 0.48660
## 15  7 0.76210 0.9010 0.44330
## 16  7 0.78570 0.9010 0.40400
## 17  7 0.80530 0.9010 0.36810
## 18  7 0.82150 0.9010 0.33540
## 19  7 0.83500 0.9010 0.30560
## 20  7 0.84620 0.9010 0.27840
## 21  7 0.85550 0.9010 0.25370
## 22  7 0.86330 0.9010 0.23120
## 23  8 0.87060 0.9077 0.21060
## 24  8 0.87690 0.9077 0.19190
## 25  8 0.88210 0.9077 0.17490
## 26  8 0.88650 0.9077 0.15930
## 27  8 0.89010 0.9077 0.14520
## 28  8 0.89310 0.9077 0.13230
## 29  8 0.89560 0.9077 0.12050
## 30  8 0.89760 0.9077 0.10980
## 31  9 0.89940 0.9087 0.10010
## 32  9 0.90100 0.9087 0.09117
## 33  9 0.90230 0.9087 0.08307
## 34  9 0.90340 0.9087 0.07569
## 35 10 0.90430 0.9096 0.06897
## 36 11 0.90530 0.9109 0.06284
## 37 11 0.90620 0.9109 0.05726
## 38 12 0.90700 0.9113 0.05217
## 39 15 0.90780 0.9123 0.04754
## 40 16 0.90860 0.9126 0.04331
## 41 16 0.90930 0.9126 0.03947
```

```
## 42 16 0.90980 0.9126 0.03596
## 43 17 0.91030 0.9128 0.03277
## 44 17 0.91070 0.9128 0.02985
## 45 18 0.91110 0.9131 0.02720
## 46 18 0.91140 0.9131 0.02479
## 47 19 0.91170 0.9132 0.02258
## 48 19 0.91200 0.9132 0.02058
## 49 19 0.91220 0.9132 0.01875
## 50 19 0.91240 0.9132 0.01708
## 51 19 0.91250 0.9132 0.01557
## 52 19 0.91260 0.9132 0.01418
## 53 19 0.91270 0.9132 0.01292
## 54 19 0.91280 0.9132 0.01178
## 55 19 0.91290 0.9132 0.01073
## 56 19 0.91290 0.9132 0.00978
## 57 19 0.91300 0.9132 0.00891
## 58 19 0.91300 0.9132 0.00812
## 59 19 0.91310 0.9132 0.00740
## 60 19 0.91310 0.9132 0.00674
## 61 19 0.91310 0.9132 0.00614
## 62 20 0.91310 0.9132 0.00559
## 63 20 0.91310 0.9132 0.00510
## 64 20 0.91310 0.9132 0.00464
## 65 20 0.91320 0.9132 0.00423
## 66 20 0.91320 0.9132 0.00386
## 67 20 0.91320 0.9132 0.00351
```

There is an extra column `%Dev R` where the `R` stands for "relaxed", which is the percent deviance explained by the relaxed fit. This is always higher than its neighboring column, which is the same for the penalized fit (on the training data).

The fit object is class `relaxed`, which inherits from class `glmnet`. One can plot it, with additional flexibility.

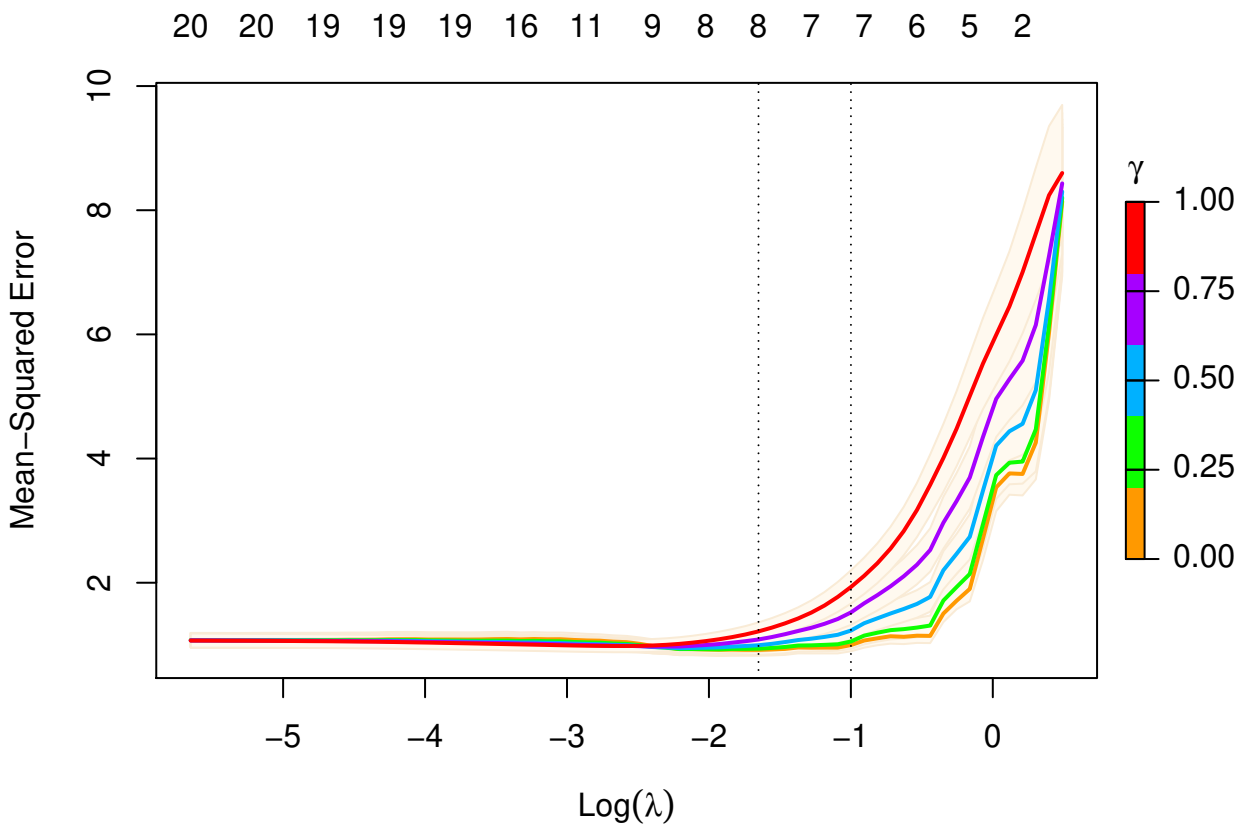
```
par(mfrow=c(1,3))
plot(fit)
plot(fit,gamma=0.5)
plot(fit,gamma=0)
```



So again, `gamma=1` is the traditional `glmnet` fit, while `gamma=0` is the unpenalized fit, and `gamma=0.5` is a mixture of the two (at the coefficient level, and hence also the linear predictors).

We can also select `gamma` using `cv.glmnet`, which by default uses the 5 values `c(0, 0.25, 0.5, 0.75, 1)`.

```
cfit=cv.glmnet(x,y,relax=TRUE)
plot(cfit)
```



The plot command has an `se.bands` option if you don't like the default shading of these bands.

Just like before, you can make predictions from a CV object, and it uses the selected values for `lambda` and gamma.

```
predict(cvfit,newx)
```

A new feature in `glmnet` is a print method for `cv.glmnet` and a `cv.relaxed` object.

```
print(cfit)
```

```
##
## Call:  cv.glmnet(x = x, y = y, relax = TRUE)
##
## Measure: Mean-Squared Error
##
##     Gamma Lambda Measure      SE Nonzero
## min     0 0.1919  0.9184 0.09504       8
## 1se     0 0.3681  0.9955 0.10006       7
```

More details on relaxed fitting

Although `glmnet` has a `relax` option, you can created a relaxed version by post-processing a `glmnet` object.

```
fit=glmnet(x,y)
fitr=relax.glmnet(fit,x=x,y=y)
```

This will rarely need to be done; one use case is if the original fit took a long time, and the user wanted to avoid refitting it. Note that in the call the arguments are named, since they are passed in via the `...` argument to `relax.glmnet`.

Needless to say, *any* of the families fit by `glmnet` can also be fit with the `relaxed` option.

As mentioned, a `relaxed` object is also a `glmnet` object. Apart from the class modification, it has an additional componet named `relaxed` which is itself a `glmnet` object, but with the relaxed coefficients. The default behavior of extractor functions like `predict` and `coef`, as well as `plot` will be to present results from the `glmnet` fit, unless a value of `gamma` is given different from the default value `gamma=1` (see the plots above). The `print` method gives additional info on the relaxed fit.

Likewise, a `cv.relaxed` object inherits from class `cv.glmnet`. Here the `predict` method by default uses the optimal relaxed fit; if predictions from the CV-optimal *original* `glmnet` fit are desired, one can directly use `predict.cv.glmnet`. Similarly for the `print` command, which we illustrate here.

```
print(cfit)
```

```
##
## Call:  cv.glmnet(x = x, y = y, relax = TRUE)
##
## Measure: Mean-Squared Error
##
##     Gamma Lambda Measure      SE Nonzero
## min     0 0.1919  0.9184 0.09504       8
## 1se     0 0.3681  0.9955 0.10006       7
```

```
print.cv.glmnet(cfit)
```

```
##
## Call:  cv.glmnet(x = x, y = y, relax = TRUE)
##
## Measure: Mean-Squared Error
##
##      Lambda Measure     SE Nonzero
## min 0.06897  0.9811 0.1005      10
## 1se 0.13228  1.0590 0.1155       8
```

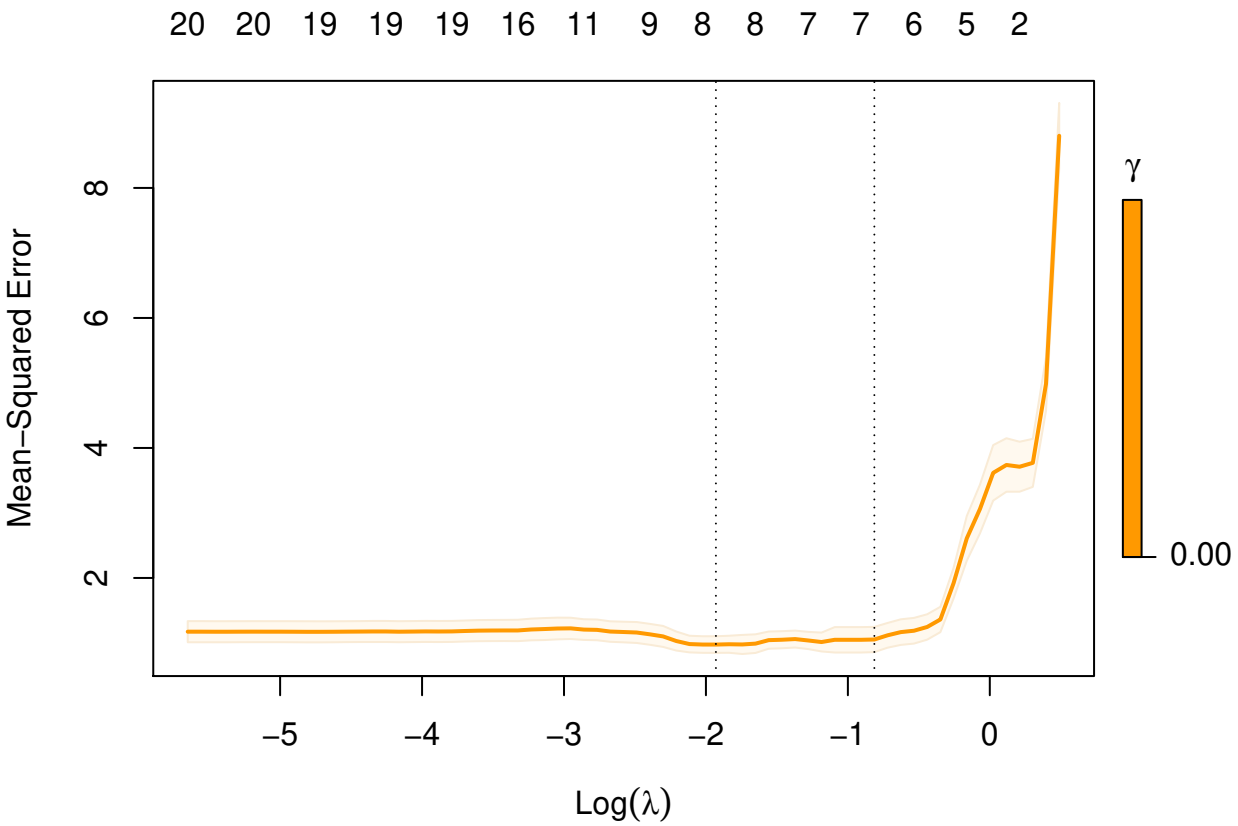
Relaxed fits and glms

`glmnet` itself is used to fit the relaxed fits, by using a single value of zero for `lambda`. However, for nonlinear models such as binomial, multinomial and poisson, there can be convergence issues. This is because `glmnet` does not do stepsize optimization, rather relying on the pathwise fit to stay in the "quadratic" zone of the log likelihood. We have an optional `path=TRUE` option for `relax.glmnet`, which actually fits a regurized path toward the `lambda=0` solution, and thus avoids the issue. The default is `path=FALSE` since this option adds to the computing time.

Forward stepwise and relaxed fit

One use case for a relaxed fit is as a faster version of forward stepwise regression. With a large number `p` of variables, forward-stepwise regression can be tedious. Lasso on the other hand, because of its convexity, can plunge in and identify good candidate sets of variables over 100 values of `lambda`, even though `p` could be in the 10s of thousands. In a case like this, one can have `cv.glmnet` do the selection.

```
fitr=cv.glmnet(x,y,gamma=0,relax=TRUE)
plot(fitr)
```



Notice that we only allow `gamma=0`, so in this case we are not considering the blended fits.

Progress bar

We finally have a progress bar for `glmnet` and `cv.glmnet`. Ever run a job on a big dataset, and wonder how long it will take? Now you can use the `trace.it = TRUE` argument to these functions.

```
fit=glmnet(x,y,trace=TRUE)
```

```
##
|==================================            |65%
```

Here we abbreviated the argument to `trace`. This display changes in place as the fit is produced. Also very helpful with `cv.glmnet`

```
fit=cv.glmnet(x,y,trace=TRUE)
```

```
##
Training
|==================================================| 100%
Fold: 1/10
|==================================================| 100%
Fold: 2/10
|==================================================| 100%
Fold: 3/10
```

```
|==========================================| 100%
Fold: 4/10
|==========================================| 100%
Fold: 5/10
|==========================================| 100%
Fold: 6/10
|============================              |  70%
```

Tracing of the folds works a little differently when distributed computing is used.

Here the `trace` argument should be used in each call to `glmnet` or `cv.glmnet`. One can set this option session wide via a call to `glmnet.control` with its new `itrace` argument:

```
glmnet.control(itrace=1)
```

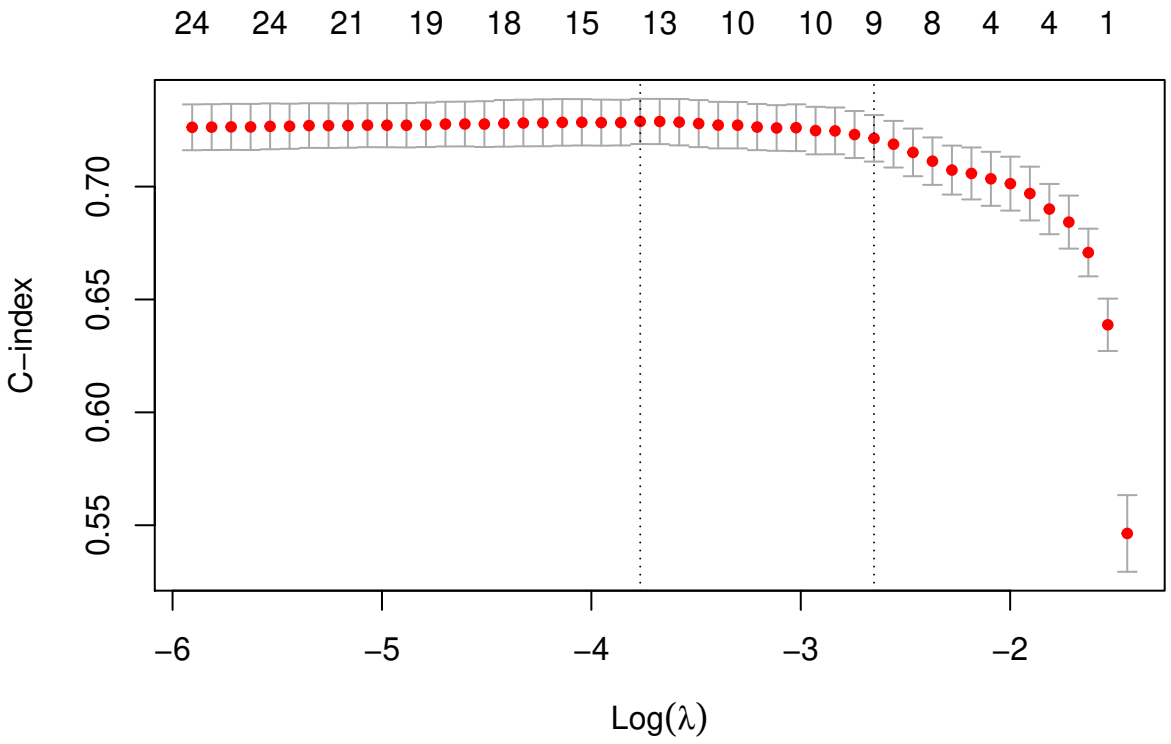
To reset it, one makes a similar call and sets `itrace=0`.

C index for Cox models

We have a new performance measure for the Cox model: the Harrel *C index*. This is like the AUC measure of concordance for survival data, but only considers comparable pairs. Pure concordance would record the fraction of pairs for which the order of the death times agree with the order of the predicted risk. But with survival data, if an observation is right censored at a time *before* another observation's death time, they are not comparable.

```
data(CoxExample)
```

```
cvfit=cv.glmnet(x,y,family="cox",type.measure="C")
plot(cvfit)
```



Assessing models on test data

Once we have fit a series of models using `glmnet`, we often assess their performance on a set of evaluation or test data. We usually go through the process of building a prediction matrix, and then deciding on the measure, and computing the values for a series of values for `lambda` and now `gamma`. Here we provide three functions for making these tasks easier.

Performance measures

The function `assess.glmnet` computes the same performance measures produced by `cv.glmnet`, but on a validation or test dataset.

```
data(BinomialExample)
itrain=1:70
fit=glmnet(x[itrain,],y[itrain],family="binomial",nlambda=20)
assess.glmnet(fit,newx=x[-itrain,],newy=y[-itrain])
```

```
## $deviance
##        s0        s1        s2        s3        s4        s5        s6
## 1.3877348 1.2319096 1.0555731 0.9020831 0.8564873 0.8716707 0.9415079
##        s7        s8        s9       s10       s11       s12       s13
## 1.0669161 1.3180742 1.6236987 1.9513754 2.2105092 2.4922882 2.7629164
##       s14       s15       s16       s17       s18
## 2.9792768 3.1275960 3.2759824 3.4240188 3.5722750
## attr(,"measure")
## [1] "Binomial Deviance"
##
## $class
##        s0        s1        s2        s3        s4        s5        s6
## 0.4666667 0.3333333 0.2000000 0.1666667 0.1666667 0.1666667 0.2000000
##        s7        s8        s9       s10       s11       s12       s13
## 0.2000000 0.2000000 0.2333333 0.2333333 0.2000000 0.2000000 0.2000000
##       s14       s15       s16       s17       s18
## 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667
## attr(,"measure")
## [1] "Misclassification Error"
##
## $auc
##  [1] 0.5714286 0.7991071 0.8437500 0.8928571 0.9017857 0.9062500 0.8973214
##  [8] 0.8928571 0.8705357 0.8526786 0.8392857 0.8214286 0.8214286 0.8169643
## [15] 0.8169643 0.8169643 0.8125000 0.8125000 0.8080357
## attr(,"measure")
## [1] "AUC"
##
## $mse
##        s0        s1        s2        s3        s4        s5        s6
## 0.5006803 0.4265596 0.3477022 0.2836649 0.2605716 0.2537474 0.2701316
##        s7        s8        s9       s10       s11       s12       s13
## 0.2937508 0.3324507 0.3610377 0.3651420 0.3570217 0.3549473 0.3535171
##       s14       s15       s16       s17       s18
## 0.3514806 0.3496168 0.3480045 0.3467192 0.3455224
## attr(,"measure")
## [1] "Mean-Squared Error"
##
```

```
## $mae
##        s0        s1        s2        s3        s4        s5        s6
## 0.9904762 0.8979473 0.7675067 0.6352868 0.5533314 0.5012508 0.4736140
##        s7        s8        s9       s10       s11       s12       s13
## 0.4565569 0.4683982 0.4715432 0.4538161 0.4305615 0.4135260 0.4026998
##       s14       s15       s16       s17       s18
## 0.3941315 0.3871481 0.3816993 0.3776846 0.3744014
## attr(,"measure")
## [1] "Mean Absolute Error"
```

This produces a list with *all* the measures suitable for a binomial model, computed for the entire sequence of lambdas in the fit object. Here the function identifies the model family from the fit object.

A second use case builds the prediction matrix first

```
pred=predict(fit,newx=x[-itrain,])
assess.glmnet(pred,newy=y[-itrain],family="binomial")
```

Here we have to provide the `family` as an argument; the results (not shown) are the same. Users can see the various measures suitable for each family via

```
glmnet.measures()
```

```
## $gaussian
## [1] "mse" "mae"
##
## $binomial
## [1] "deviance" "class"    "auc"      "mse"      "mae"
##
## $poisson
## [1] "deviance" "mse"      "mae"
##
## $cox
## [1] "deviance" "C"
##
## $multinomial
## [1] "deviance" "class"    "mse"      "mae"
##
## $mgaussian
## [1] "mse" "mae"
##
## $GLM
## [1] "deviance" "mse"      "mae"
```

The assess function can also take the result of `cv.glmnet` as input. In this case the predictions are made at the optimal values for the parameter(s).

```
cfit=cv.glmnet(x[itrain,],y[itrain],family="binomial", nlambda = 30)
assess.glmnet(cfit,newx=x[-itrain,],newy=y[-itrain])
```

```
## $deviance
##         1
## 0.9482246
## attr(,"measure")
## [1] "Binomial Deviance"
##
## $class
##   1
```

```
## 0.2
## attr(,"measure")
## [1] "Misclassification Error"
##
## $auc
## [1] 0.875
## attr(,"measure")
## [1] "AUC"
##
## $mse
##         1
## 0.3028376
## attr(,"measure")
## [1] "Mean-Squared Error"
##
## $mae
##         1
## 0.6797343
## attr(,"measure")
## [1] "Mean Absolute Error"
```

This used the default value of `s=lambda.1se`, just like `predict` would have done. Users can provide additional arguments that get passed on to predict:

```
assess.glmnet(cfit,newx=x[-itrain,],newy=y[-itrain], s="lambda.min")
```

```
## $deviance
##        1
## 0.877155
## attr(,"measure")
## [1] "Binomial Deviance"
##
## $class
##         1
## 0.1666667
## attr(,"measure")
## [1] "Misclassification Error"
##
## $auc
## [1] 0.8973214
## attr(,"measure")
## [1] "AUC"
##
## $mse
##        1
## 0.273071
## attr(,"measure")
## [1] "Mean-Squared Error"
##
## $mae
##         1
## 0.6069619
## attr(,"measure")
## [1] "Mean Absolute Error"
```

One interesting use case is to get the results of CV using other measures, via the `keep` argument. In this case

the `fit.preval` object is a matrix of prevalidated predictions made using the folds `foldid`

```
cfit=cv.glmnet(x,y,family="binomial",keep=TRUE, nlambda = 30)
assess.glmnet(cfit$fit.preval,newy=y,family="binomial")
```

```
## $deviance
##        s0        s1        s2        s3        s4        s5        s6
## 1.3856468 1.2709239 1.1737326 1.1009454 1.0030400 0.9022307 0.8308365
##        s7        s8        s9       s10       s11       s12       s13
## 0.7872770 0.7797691 0.8036724 0.8589054 0.9462189 1.0695191 1.2325427
##       s14       s15       s16       s17       s18       s19       s20
## 1.4174117 1.6029760 1.7744423 1.9058043 2.0279530 2.1295714 2.2180721
##       s21       s22       s23       s24       s25       s26       s27
## 2.2964327 2.3758962 2.4415127 2.5057143 2.5709974 2.6373618 2.7014168
##       s28       s29
## 2.7359325 2.7462087
## attr(,"measure")
## [1] "Binomial Deviance"
##
## $class
##   s0   s1   s2   s3   s4   s5   s6   s7   s8   s9  s10  s11  s12  s13  s14
## 0.44 0.30 0.24 0.24 0.21 0.19 0.17 0.14 0.15 0.13 0.14 0.14 0.17 0.20 0.19
##  s15  s16  s17  s18  s19  s20  s21  s22  s23  s24  s25  s26  s27  s28  s29
## 0.19 0.19 0.18 0.17 0.17 0.17 0.17 0.17 0.17 0.17 0.18 0.18 0.18 0.18 0.18
## attr(,"measure")
## [1] "Misclassification Error"
##
## $auc
##  [1] 0.4042208 0.7483766 0.7966721 0.8100649 0.8441558 0.8770292 0.8956981
##  [8] 0.9042208 0.9078734 0.9070617 0.9038149 0.8973214 0.8896104 0.8879870
## [15] 0.8883929 0.8859578 0.8883929 0.8892045 0.8908279 0.8920455 0.8924513
## [22] 0.8932630 0.8932630 0.8944805 0.8956981 0.8956981 0.8969156 0.8952922
## [29] 0.8961039 0.8920455
## attr(,"measure")
## [1] "AUC"
##
## $mse
##        s0        s1        s2        s3        s4        s5        s6
## 0.4995799 0.4433811 0.3974031 0.3651486 0.3242961 0.2834930 0.2553781
##        s7        s8        s9       s10       s11       s12       s13
## 0.2351367 0.2262225 0.2263545 0.2345364 0.2476188 0.2641884 0.2815758
##       s14       s15       s16       s17       s18       s19       s20
## 0.2968524 0.3080594 0.3157392 0.3211751 0.3251428 0.3283560 0.3308609
##       s21       s22       s23       s24       s25       s26       s27
## 0.3324919 0.3339406 0.3351352 0.3361829 0.3372384 0.3383140 0.3394635
##       s28       s29
## 0.3405847 0.3419652
## attr(,"measure")
## [1] "Mean-Squared Error"
##
## $mae
##        s0        s1        s2        s3        s4        s5        s6
## 0.9915585 0.9256767 0.8556042 0.7920840 0.7135086 0.6296387 0.5604366
##        s7        s8        s9       s10       s11       s12       s13
## 0.5033007 0.4614274 0.4304225 0.4096567 0.3973073 0.3908453 0.3882463
```

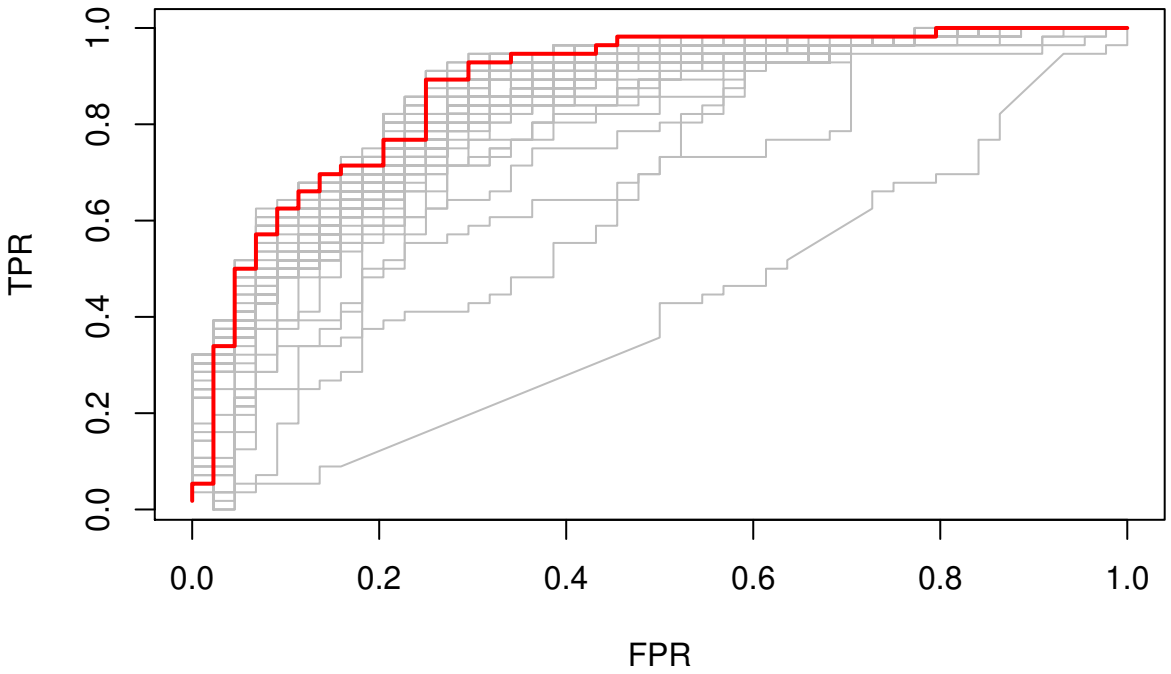
```
##       s14       s15       s16       s17       s18       s19       s20
## 0.3879676 0.3851510 0.3803985 0.3757085 0.3716115 0.3684525 0.3660858
##       s21       s22       s23       s24       s25       s26       s27
## 0.3640030 0.3626859 0.3617648 0.3611738 0.3609122 0.3608514 0.3610453
##       s28       s29
## 0.3616784 0.3625781
## attr(,"measure")
## [1] "Mean Absolute Error"
```

Users can verify that the first measure here `deviance` is identical to the component `cvm` on the `cfit` object.

ROC curves for binomial data

In the special case of binomial models, users often would like to see the ROC curve for validation or test data. Here the function `roc.glmnet` provides the goodies. Its first argument is as in `assess.glmnet`. Here we illustrate one use case, using the prevlidated CV fit as before.

```
cfit=cv.glmnet(x,y,family="binomial", type.measure="auc", keep=TRUE)
rocs=roc.glmnet(cfit$fit.preval,newy=y)
which=match(cfit$lambda.min,cfit$lambda)
plot(rocs[[which]],type="l")
nopr=sapply(rocs,lines,col="grey")
lines(rocs[[which]],lwd=2,col="red")
```



In this case `roc.glmnet` returns a list of cross-validated ROC data, one for each model along the path. In the third line we identify the CV winner. Then we plot all the curves in grey, and the winner in red.

Confusion matrices for classification

For binomial and multinomial models, we often which to examine the classification performance on new data. The function `confusion.glmnet` will do that.

```
data(MultinomialExample)
set.seed(101)
itrain=sample(1:500,400,replace=FALSE)
cfit=cv.glmnet(x[itrain,],y[itrain],family="multinomial")
cnf=confusion.glmnet(cfit,newx=x[-itrain,],newy=y[-itrain])
print(cnf)
```

```
##          True
## Predicted  1  2  3 Total
##     1     13  6  4    23
##     2      7 25  5    37
##     3      4  3 33    40
##     Total 24 34 42   100
##
##  Percent Correct:  0.71
```

It produces a table of class `confusion.table` which inherits from calss `table`, and we also provide a print method.

The first argument to `confusion.glmnet` should be either a `glmnet` object, or a `cv.glmnet` object, from which predictions can be made, or a matrix/array of predictions, such as the *kept* `fit.predval` object from `cv.glmnet`.

In the second case we need to specify the `family`, otherwise *confusion* can exist between `binomial` and `multinomial` prediction matrices. Here we show a multinomial example

```
cfit=cv.glmnet(x,y,family="multinomial",type="class",keep=TRUE)
cnf=confusion.glmnet(cfit$fit.preval,newy=y,family="multinomial")
which=match(cfit$lambda.min,cfit$lambda)
print(cnf[[which]])
```

```
##          True
## Predicted   1   2   3 Total
##     1      76  22  14   112
##     2      39 129  23   191
##     3      27  23 147   197
##     Total 142 174 184   500
##
##  Percent Correct:  0.704
```

Since the `fit.preval` object has predictions for the whole path, the result of `confusion.glmnet` here is a list of confusion tables. We identify and print the one corresponding to the minimum classification error.

Fitting big and/or sparse GLMs

We include a function `bigGlm` for fitting a single GLM model (unpenalized), but allowing all the options of `glmnet`. In other words, coefficient upper and/or lower bounds and sparse `x` matrices. This is not too much more than fitting a model with a single value of `lambda=0` (with some protection from edge cases). There is also a `predict` and `print` method.

```
data(BinomialExample)
fit=bigGlm(x,y,family="binomial",lower.limits=-1)
print(fit)
```

```
##
## Call:  bigGlm(x = x, y = y, family = "binomial", lower.limits = -1)
```

```
##
##   Df   %Dev Lambda
## 1 30 0.7757      0
```

Producing x from mixed variables, and missing data

We have created a function `makeX` that makes it easy to create the model matrix `x` needed as input to `glmnet`. It takes as input a data frame, which can contain vectors, matrices and factors. Some of the features are

- Factors are *one-hot* encoded to form indicator matrices
- Missing values in the resultant matrix can be replaced by the column means
- The `sparse` option returns a matrix in column-sparse format. This is useful if the data are large, and factors have many levels.
- Two dataframes can be provided, `train` and `test`. This ensures the factor levels correspond, and also imputes missing data in the test data from means in the training data.

  We start with a simple case with some factors.

```
set.seed(101)
X = matrix(rnorm(20),10,2)
X3=sample(letters[1:3],10,replace=TRUE)
X4=sample(LETTERS[1:3],10,replace=TRUE)
df=data.frame(X,X3,X4)
makeX(df)
```

```
##            X1         X2 X3a X3b X3c X4A X4B X4C
## 1  -0.3260365  0.5264481   0   1   0   0   0   1
## 2   0.5524619 -0.7948444   0   0   1   0   1   0
## 3  -0.6749438  1.4277555   1   0   0   0   1   0
## 4   0.2143595 -1.4668197   1   0   0   1   0   0
## 5   0.3107692 -0.2366834   1   0   0   0   1   0
## 6   1.1739663 -0.1933380   0   1   0   1   0   0
## 7   0.6187899 -0.8497547   1   0   0   1   0   0
## 8  -0.1127343  0.0584655   0   1   0   1   0   0
## 9   0.9170283 -0.8176704   0   1   0   0   0   1
## 10 -0.2232594 -2.0503078   0   0   1   0   0   1
```

Or if a sparse output was desired:

```
makeX(df,sparse=TRUE)
```

```
## 10 x 8 sparse Matrix of class "dgCMatrix"
##            X1         X2 X3a X3b X3c X4A X4B X4C
## 1  -0.3260365  0.5264481   .   1   .   .   .   1
## 2   0.5524619 -0.7948444   .   .   1   .   1   .
## 3  -0.6749438  1.4277555   1   .   .   .   1   .
## 4   0.2143595 -1.4668197   1   .   .   1   .   .
## 5   0.3107692 -0.2366834   1   .   .   .   1   .
## 6   1.1739663 -0.1933380   .   1   .   1   .   .
## 7   0.6187899 -0.8497547   1   .   .   1   .   .
## 8  -0.1127343  0.0584655   .   1   .   1   .   .
## 9   0.9170283 -0.8176704   .   1   .   .   .   1
## 10 -0.2232594 -2.0503078   .   .   1   .   .   1
```

And now some missing values

```
Xn=X
Xn[3,1]=NA;Xn[5,2]=NA
X3n=X3;
X3n[6]=NA
X4n=X4
X4n[9]=NA
dfn=data.frame(Xn,X3n,X4n)
makeX(dfn)
```

```
##            X1         X2 X3na X3nb X3nc X4nA X4nB X4nC
## 1  -0.3260365  0.5264481    0    1    0    0    0    1
## 2   0.5524619 -0.7948444    0    0    1    0    1    0
## 3          NA  1.4277555    1    0    0    0    1    0
## 4   0.2143595 -1.4668197    1    0    0    1    0    0
## 5   0.3107692         NA    1    0    0    0    1    0
## 6   1.1739663 -0.1933380   NA   NA   NA    1    0    0
## 7   0.6187899 -0.8497547    1    0    0    1    0    0
## 8  -0.1127343  0.0584655    0    1    0    1    0    0
## 9   0.9170283 -0.8176704    0    1    0   NA   NA   NA
## 10 -0.2232594 -2.0503078    0    0    1    0    0    1
```

which we can replace with column-mean imputations (and make sparse, if we like)

```
makeX(dfn,na.impute=TRUE,sparse=TRUE)
```

```
## 10 x 8 sparse Matrix of class "dgCMatrix"
##            X1         X2      X3na      X3nb      X3nc      X4nA      X4nB
## 1  -0.3260365  0.5264481 .         1.0000000 .         .         .
## 2   0.5524619 -0.7948444 .         .         1.0000000 .         1.0000000
## 3   0.3472605  1.4277555 1.0000000 .         .         .         1.0000000
## 4   0.2143595 -1.4668197 1.0000000 .         .         1.0000000 .
## 5   0.3107692 -0.4622295 1.0000000 .         .         .         1.0000000
## 6   1.1739663 -0.1933380 0.4444444 0.3333333 0.2222222 1.0000000 .
## 7   0.6187899 -0.8497547 1.0000000 .         .         1.0000000 .
## 8  -0.1127343  0.0584655 .         1.0000000 .         1.0000000 .
## 9   0.9170283 -0.8176704 .         1.0000000 .         0.4444444 0.3333333
## 10 -0.2232594 -2.0503078 .         .         1.0000000 .         .
##         X4nC
## 1  1.0000000
## 2  .
## 3  .
## 4  .
## 5  .
## 6  .
## 7  .
## 8  .
## 9  0.2222222
## 10 1.0000000
```

Finally if a test set is available as well

```
X = matrix(rnorm(10),5,2)
X3=sample(letters[1:3],5,replace=TRUE)
X4=sample(LETTERS[1:3],5,replace=TRUE)
Xn=X
Xn[3,1]=NA;Xn[5,2]=NA
```

```
X3n=X3;
X3n[1]=NA
X4n=X4
X4n[2]=NA
dftn=data.frame(Xn,X3n,X4n)
makeX(dfn,dftn,na.impute=TRUE, sparse=TRUE)
```

```
## $x
## 10 x 8 sparse Matrix of class "dgCMatrix"
##            X1         X2      X3na      X3nb      X3nc      X4nA      X4nB
## 1  -0.3260365  0.5264481 .         1.0000000 .         .         .
## 2   0.5524619 -0.7948444 .         .         1.0000000 .         1.0000000
## 3   0.3472605  1.4277555 1.0000000 .         .         .         1.0000000
## 4   0.2143595 -1.4668197 1.0000000 .         .         1.0000000 .
## 5   0.3107692 -0.4622295 1.0000000 .         .         .         1.0000000
## 6   1.1739663 -0.1933380 0.4444444 0.3333333 0.2222222 1.0000000 .
## 7   0.6187899 -0.8497547 1.0000000 .         .         1.0000000 .
## 8  -0.1127343  0.0584655 .         1.0000000 .         1.0000000 .
## 9   0.9170283 -0.8176704 .         1.0000000 .         0.4444444 0.3333333
## 10 -0.2232594 -2.0503078 .         .         1.0000000 .         .
##         X4nC
## 1  1.0000000
## 2  .
## 3  .
## 4  .
## 5  .
## 6  .
## 7  .
## 8  .
## 9  0.2222222
## 10 1.0000000
##
## $xtest
## 5 x 8 sparse Matrix of class "dgCMatrix"
##            X1         X2      X3na      X3nb      X3nc      X4nA      X4nB
## 11 -0.5098443 -0.7556130 0.4444444 0.3333333 0.2222222 .         1.0000000
## 12  1.5661805  1.7384118 .         .         1.0000000 0.4444444 0.3333333
## 13  0.3472605  0.7580952 .         1.0000000 .         .         1.0000000
## 14  1.0059925  2.1152294 1.0000000 .         .         1.0000000 .
## 15 -0.5829222 -0.4622295 .         1.0000000 .         .         1.0000000
##         X4nC
## 11 .
## 12 0.2222222
## 13 .
## 14 .
## 15 .
```

References

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2010. "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software, Articles* 33 (1): 1–22. https://doi.org/10.18637/jss.v033.i01.

Hastie, Trevor, Robert Tibshirani, and Ryan Tibshirani. 2017. "Extended Comparisons of Best Subset Selection, Forward Stepwise Selection, and the Lasso."

Simon, Noah, Jerome Friedman, and Trevor Hastie. 2013. "A Blockwise Descent Algorithm for Group-Penalized Multiresponse and Multinomial Regression."

Simon, Noah, Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2011. "Regularization Paths for Cox's Proportional Hazards Model via Coordinate Descent." *Journal of Statistical Software, Articles* 39 (5): 1–13. https://doi.org/10.18637/jss.v039.i05.

Tibshirani, Robert, Jacob Bien, Jerome Friedman, Trevor Hastie, Noah Simon, Jonathan Taylor, and Ryan Tibshirani. 2012. "Strong Rules for Discarding Predictors in Lasso-Type Problems." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 74 (2): 245–66. https://doi.org/10.1111/j.1467-9868.2011.01004.x.