UWEO StatR201 Lecture 7

Advanced Model Selection Methods

Assaf Oron, February 2013

Tonight’s Menu

Finishing up “standard” (…-subset) model-selection: - Diagnostics - How to select the “best” model when using Cross-Validation

Penalized Least-Squares:

- Motivation, Formulation, Variants
- **The Lasso. Definitely tonight’s centerpiece.**

Dimension Reduction Methods:

- Motivation, Matrix-Algebra Methodology
- Principal Components Analysis (PCA) Regression
- Partial Least Squares (PLS) Regression

…finally, if there’s time we will go over the HW4 assignment, and maybe do an improvised “R trick/workaround” session. But tonight’s material covers a lot of conceptual ground - so we’ll see if we get to it…

Model-Selection Diagnostics

Once you've found a "best model" via some method, it is always a good idea to run it directly (if it's not already done), and examine the sign and magnitude of all effect estimates:

```
step(lm(gp100m ~ .^2, data = autos[, -c(1, 8, 9)]), trace = FALSE, k = log(298))
```

```
##
## Call:
## lm(formula = gp100m ~ volume + hp + weight + accel + year + diesel +
##     volume:weight + volume:year + hp:weight + hp:accel + accel:year,
##     data = autos[, -c(1, 8, 9)])
##
## Coefficients:
##   (Intercept)         volume             hp         weight          accel
##     -6.49e+00       4.76e-02      -4.12e-02       5.46e-04       6.78e-01
##          year     dieselTRUE  volume:weight    volume:year      hp:weight
##      1.27e-01      -6.17e-01      -2.51e-06      -4.83e-04       6.98e-06
##     hp:accel     accel:year
##      2.21e-03      -1.06e-02
```

OK? **No!**

1. Covariates come in drastically different units, so effect magnitudes are meaningless.
2. There are **interactions involving all covariates**, so the sign of all individual-covariate terms are meaningless as well (*why?*).

Model-Selection Diagnostics: Scale the Covariates

```
step(lm(gp100m ~ .^2, data = data.frame(cbind(autos$continent, scale(autos[,
    -c(1, 8, 9)])))), trace = FALSE, k = log(298))
```

```
##
## Call:
## lm(formula = gp100m ~ cyl + hp + weight + accel + year + diesel +
##     hp:accel + hp:year + accel:year, data = data.frame(cbind(autos$continent,
##     scale(autos[, -c(1, 8, 9)]))))
##
## Coefficients:
## (Intercept)          cyl           hp       weight        accel
##      0.0404       0.1545       0.3061       0.4733       0.1424
##        year       diesel     hp:accel      hp:year   accel:year
##     -0.3086      -0.0629       0.1086      -0.1588      -0.1004
```

- Each individual covariate term now signify the effect when its “interaction partners” are at their mean value.
- The scaling makes the problem **dimensionless:** now the magnitude is equal to that term’s partial (=conditional) correlation with y, and are comparable. **Bigger numbers mean larger impact.**
- I’m beginning to think that `accel` might indeed just mean plain acceleration (and not *“time to…”* as I originally said).
- Note that I had to pull `continent` out (it being a factor) before scaling, then tack it back on. You can also use `model.matrix`, as we’ll see later.

Cross-Validation Diagnostics: Visualizing the Fit

It is not a bad idea to take the model above, fit it and examine the fit – but being an in-sample method, it is not always advisable to "fix" that fit too closely. With Cross-Validation, OTOH, you should **definitely** do it:

```
cvid = sample(rep(1:5, 60), size = 298)  ## K=5 CV groups
table(cvid)
```

```
## cvid
##  1  2  3  4  5
## 60 60 59 60 59
```

Note how the CV groups were sampled. This is known as "block permutation", an easy trick to make sure CV groups are random and yet (nearly) exactly equal-sized.

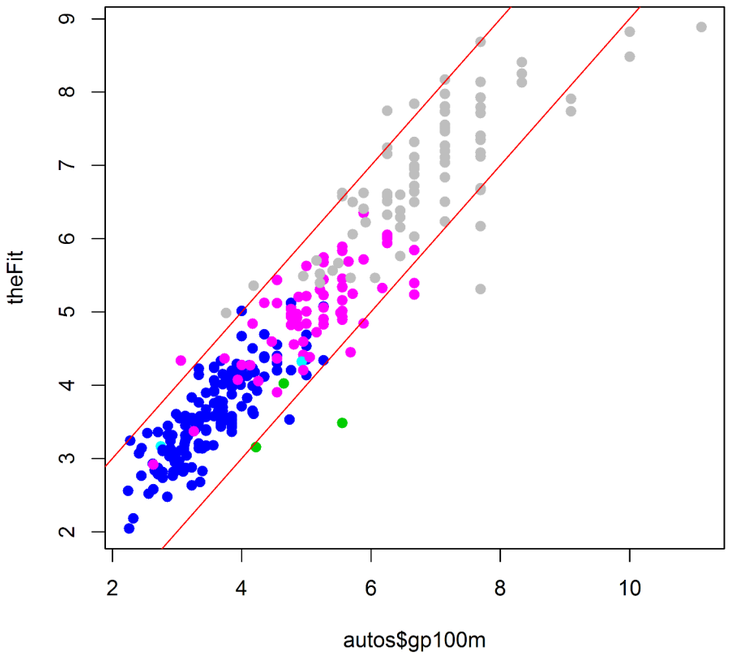
```
mynames = c(names(autos)[c(2:8, 11)], "weight:year")
firstCV = cvConSubsets(autos, "gp100m", cov.list = mynames, cv.group = cvid,
    forced = c(4, 6))
```

```
## Thu Feb 21 07:45:49 2013
##  Size and initial combinations:  2 36 ...now  1
##  Size and initial combinations:  3 84 ...now  7
##  Size and initial combinations:  4 126 ...now  21..
##  Size and initial combinations:  5 126 ...now  35...
##  Size and initial combinations:  6 84 ...now  35...
##  Size and initial combinations:  7 36 ...now  21..
##  Size and initial combinations:  8 9 ...now  7
##  Size and initial combinations:  9 1 ...now  1
## Started:  Thu Feb 21 07:45:49 2013      Ended:  Thu Feb 21 07:45:53 2013
## Run Completed. It is advisable to perform statistics on top/bottom performing models, rather than choose the single best one.
## Thu Feb 21 07:45:53 2013
```

Cross-Validation Diagnostics: Visualizing the Fit

We visualize the “best” model’s CV fit by plugging it back into `cvEngine` **with the same CV grouping** and `rawpreds=TRUE`:

```
theTerms = attr(terms(as.formula(firstCV$model[1])), "term.labels")
theFit = cvEngine(autos, theTerms, outcome = "gp100m", cv.group = cvid, rawpreds = TRUE)
plot(theFit ~ autos$gp100m, pch = 19, col = autos$cyl)
abline(-1, 1, col = 2)
abline(1, 1, col = 2)
```



- Note the plotting convention: we place **observations at x and predictions at y.** This makes visually low values underpredicted, and vice versa.
- Since the RMSE and Q3AE are ≈ 0.5, points beyond the red lines of ± 1 prediction error, are **pretty serious outliers.**
- Overall, this is a **high signal-to-noise dataset.** Any suggestions why?
- As I suspected, cars with odd # of cylinders might *not play nice.* **Now, go after the other outliers!**

Questions? (online/in-class)

How to Choose the “Best” Cross-Validated Model?

- The easiest option would be just to pick the one at the top.
- However, given that there’s always some overfitting going on, and considering the variability of the CV process itself…
- **…if the #1 model is large, perhaps a more-parsimonious model “close enough to the top” should be taken.**

```
options(width = 120)
head(firstCV[, -(2:3)], 10)
```

```
##     nVars  CV.R2 CV.RMSE CV.QAE                                                                  model
## 129     9 0.8789  0.5684 0.5676 gp100m~cyl+volume+hp+weight+accel+year+continent+diesel+weight:year
## 121     7 0.8785  0.5693 0.5526            gp100m~hp+weight+accel+year+continent+diesel+weight:year
## 98      6 0.8781  0.5703 0.5296                      gp100m~hp+weight+accel+year+diesel+weight:year
## 96      6 0.8770  0.5728 0.5409                        gp100m~hp+weight+accel+year+continent+diesel
## 99      6 0.8770  0.5728 0.5461                  gp100m~hp+weight+year+continent+diesel+weight:year
## 127     8 0.8770  0.5729 0.5584        gp100m~cyl+hp+weight+accel+year+continent+diesel+weight:year
## 128     8 0.8769  0.5731 0.5478     gp100m~volume+hp+weight+accel+year+continent+diesel+weight:year
## 61      5 0.8768  0.5732 0.5604                           gp100m~hp+weight+year+diesel+weight:year
## 57      5 0.8768  0.5732 0.5314                                 gp100m~hp+weight+accel+year+diesel
## 45      5 0.8767  0.5736 0.5691                          gp100m~cyl+weight+year+diesel+weight:year
```

How to Choose the “Best” Cross-Validated Model?

Following Hastie *et al.*, we can calculate a quick-and-dirty SE for our CV prediction metric:

```
cvse = rep(NA, 5)
for (a in unique(cvid)) cvse[a] = rmse(theFit[cvid == a], autos$gp100m[cvid ==
    a])
cvse
```

```
## [1] 0.6143 0.4218 0.5935 0.5447 0.5627
```

```
predSE = sd(cvse)/5
predSE
```

```
## [1] 0.01504
```

```
table(firstCV$nVars[firstCV$CV.RMSE < min(firstCV$CV.RMSE) + predSE])
```

```
##
##  3  4  5  6  7  8  9
##  1  6 12 17 15  7  1
```

```
parsi = min(firstCV$nVars[firstCV$CV.RMSE < min(firstCV$CV.RMSE) + predSE])
head(firstCV[firstCV$nVars == parsi, ], 2)
```

```
##   nVars Insample.R2 Insample.RMSE  CV.R2 CV.RMSE CV.QAE
## 8     3      0.8799         0.565 0.8743  0.5792 0.5486
## 5     3      0.8791         0.567 0.8703  0.5881 0.5801
##                       model
## 8 gp100m~weight+year+diesel
## 5     gp100m~hp+weight+year
```

- Some people might calculate the SE based on the sample size n; IMHO the relevant sample size is the number of CV blocks.
- As you can see, in this dataset there are rather parsimonious models within shot of the “best” one.
- **Try it on your machine, see what you get! A live demonstration of the variability inherent in the process.**
- **With the outliers and a CV prediction sub-sample of 60, RMSE is not very robust and its SE might be inflated.** Try and see what model is recommended, when you use the Q3AE instead!

Questions? (online/in-class)

And now… The Black Box Gets Blacker

We now move to even more pragmatic and heuristic approaches to predictive regression. These are various algorithms, often not originating in the field of statistics. Hastie *et al.* do a great job of providing statistical analogues and interpretations to them. But generally, these are definitely **not** likelihood methods.

Ridge regression, for example, minimizes this:

$$(y - \mathbf{X}\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2,$$

where $\lambda > 0$ is a **tuning parameter**, from now on a regular fixture in our class. Ridge regression is fairly old; it was motivated by a wish to "fix" **ill-posed** (numerically unstable) regression problems. There is a closed-form solution:

$$\tilde{\beta} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T y,$$

where $\mathbf{I}$ is the identity matrix. So if X is singular (=degenerate, collinear) and hence has (or almost has) no inverse, using ridge regression provides a stable solution – a biased one, but still a solution. For this reason, methods of this general family are often referred to as **regularization.**

But there's an interesting side effect: $\tilde{\beta}$ is **shrunk** towards zero, compared with the ordinary-least-squares $\hat{\beta}$. Essentially, the penalty imposes a *budget constraint* on effect magnitude.

And its nature dictates that **the worse-performing covariates suffer more** (*why?*)… but they never become zero.

Terminology note: from now on, p is the number of terms **excluding** the intercept. So a full regression model matrix is $n \times (p + 1)$.

From Ridge Regression to Lasso

Much later, Tibshirani (and others) thought it useful to look at a more general form of penalized least squares to minimize:

$$(y - \mathbf{X}\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q,$$

with $q > 0$ a constant. It turns out that

- For $q \geq 1$, some solution (whether closed-form or numerical) exists;
- For $q \leq 1$, it is possible to set individual $\tilde{\beta}$'s to zero - if one can calculate the solution.

So $q = 1$**, the absolute-value penalty, is the only case for which there is a solution algorithm, and individual covariates can be turned off and on.**

This is the Lasso: a tool that performs shrinkage, fitting and model-selection in one fell swoop. Hastie *et al.* call its effect "soft thresholding."

Lasso also possesses what is known as **The Oracle Property:** under certain assumptions, as $n \to \infty$ it is likely to select the true model if it exists in the dataset.

Lasso in R

Lasso is definitely the “Swiss Army Knife” of model selection. **Some people (such as we did at MESA-Air) only use its model-selection faculty, then go back to likelihood-based methods and fit the selected model.** Others use the Lasso fit as well.

There is a ton of packages offering Lasso in various flavors. The most prominent are

- `glmnet`, written by Hastie *et al.* and still maintained by Hastie,
- `lars`, an older package written by the same bunch and retooled for some complementary functionality, and
- `lasso2`, written by some of the biggest names in R.

`glmnet` (which I prefer) and `lars` accept a matrix for $\mathbf{X}$ and vector for y, while `lasso2` can handle R `formulae`.

They differ in other more fundamental ways. But all three suffer from one cardinal limitation: **they cannot keep together variables represented by several columns (e.g., multi-category, spline basis, an interaction and its members).** So some columns might be turned off, even as others remain.

Depending upon your application, this might or might not be a deal-breaker. Be forewarned, though, that not only Lasso suffers from this “column-relationship-blindness”. It is shared by many black-box algorithms, because they accept numerical matrices as arguments, rather than data frames.

Fortunately, there do exist alternatives in R. accepting grouped covariates.

One can probably run an entire class just on the various Lasso flavors and extensions, and their implementation in R (which is likely *“ground zero”* for these developments). Here, we will only explore `glmnet` and a spin-off that accepts grouped covariates.

Lasso in R: `glmnet`

```
library(glmnet)
automat = model.matrix(lm(gp100m ~ ., data = autos[, -c(1, 9)]))[, -1]
lass1 = glmnet(x = automat, y = autos$gp100m)
names(lass1)
```

```
##  [1] "a0"        "beta"      "df"        "dim"       "lambda"
##  [6] "dev.ratio" "nulldev"   "npasses"   "jerr"      "offset"
## [11] "call"      "nobs"
```

```
dim(lass1$beta)
```

```
## [1]  9 80
```

As usual, `glmnet` returns a list with various components. The main one to look at is `beta`, a matrix holding effect estimates for all values of λ (each column is the fit at one λ value). Here's a simple utility for viewing the nonzero values only in a given column:

```
lassVars = function(lassout, tune, minabs = 0) lassout$beta[abs(lassout$beta[,
    tune]) > minabs, tune]
lassVars(lass1, 5)  # 5th-largest lambda penalty (out of 80)
```

```
##        hp    weight
## 0.0010017 0.0004851
```

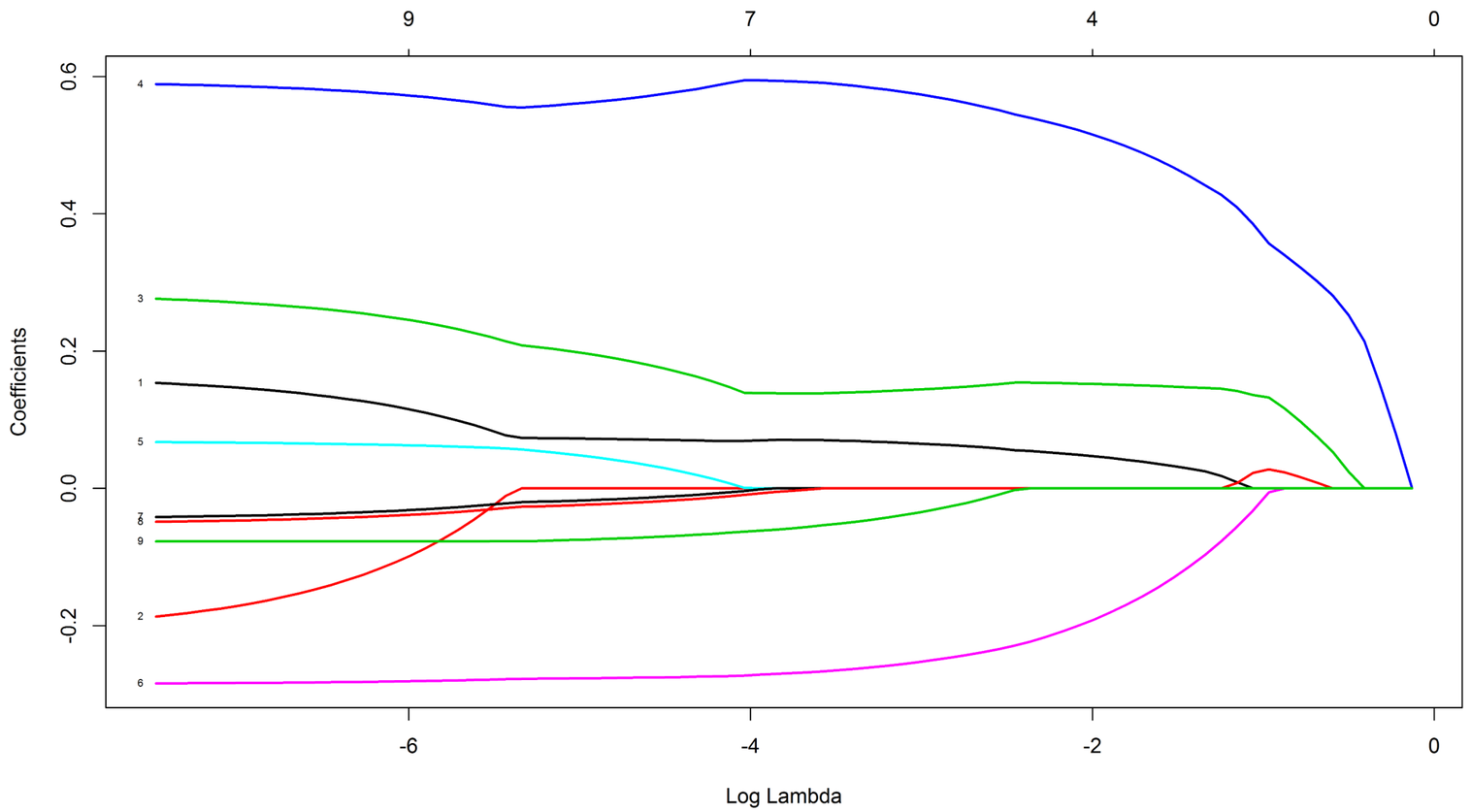
```
lassVars(lass1, 25)  # 25th largest
```

```
##       cyl        hp    weight      year
##  0.052843  0.006372  0.001037 -0.096113
```

Inside the function, covariates are automatically scaled. However, the default output back-transforms `beta` to the original units, which are often meaningless as a diagnostic tool. So let's repeat this while scaling everything up front.

Lasso in R: `glmnet`

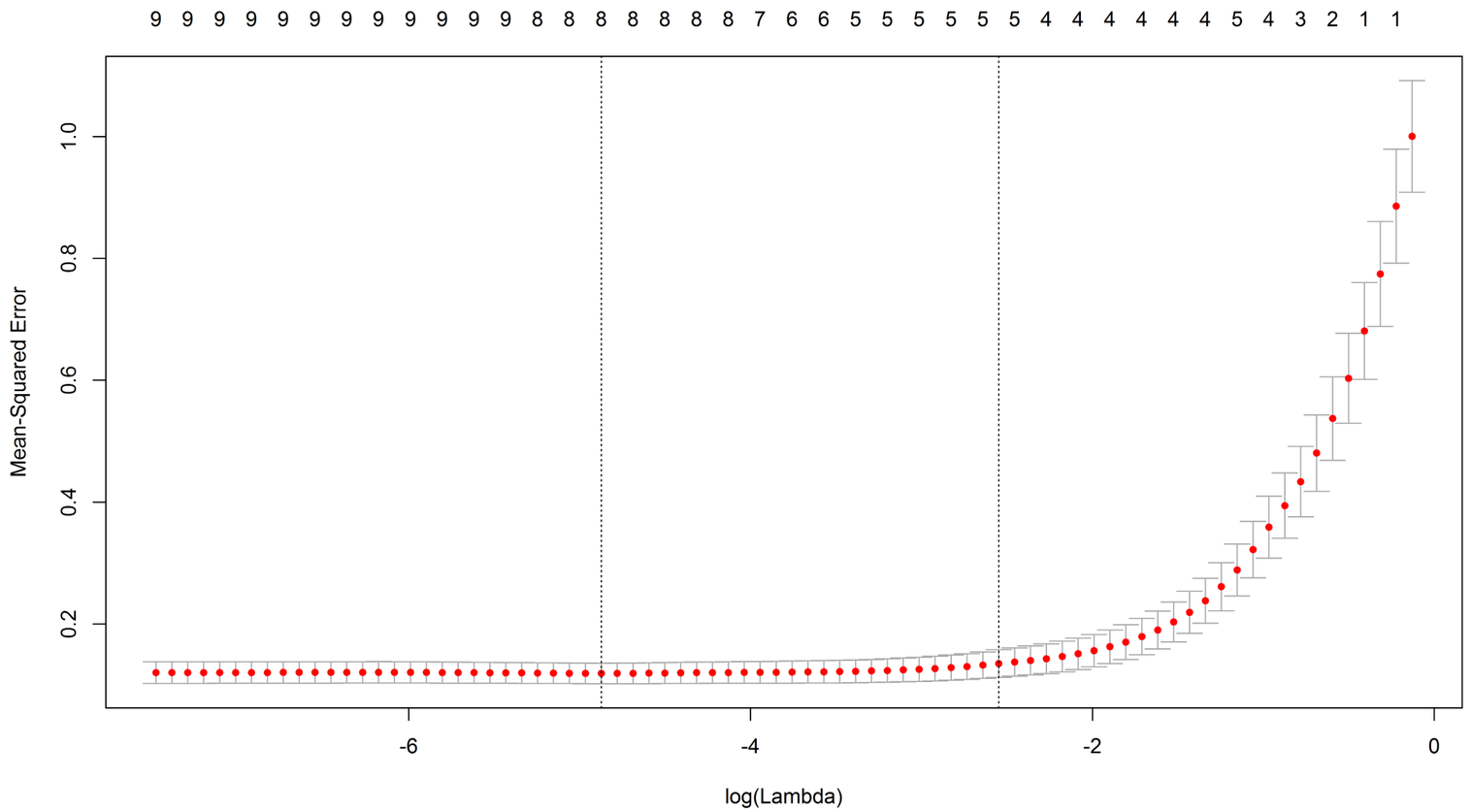
```
lass1 = glmnet(x = scale(automat), y = scale(autos$gp100m))
plot(lass1, label = TRUE, xvar = "lambda", lwd = 2)
```



The numbers along the top are the model size. How do we choose the right λ? You might guess the answer…

Lasso in R: `glmnet`

```
lass1cv = cv.glmnet(x = scale(automat), y = scale(autos$gp100m), foldid = cvid)
plot(lass1cv)
```



`glmnet` has built-in CV, either accepting your `foldid` vector, or generating its own groups.

And now we see the rationale for coming up with an “as good as best” but more parsimonious model (right-hand dotted line) rather than the “very best” one which is often an overkill.

Lasso in R: `glmnet`

How do we identify the "best" and "as good as best" models? The CV and non-CV `glmnet` objects use the exact same λ's:

```
options(width = 120)
table(lass1$lambda == lass1cv$lambda)
```

```
##
## TRUE
##   80
```

```
bestid = which.min(lass1cv$lambda)
asgoodid = which(lass1cv$lambda == lass1cv$lambda.1se)
c(asgoodid, bestid)
```

```
## [1] 27 80
```

```
round(lassVars(lass1, asgoodid), 3)
```

```
##        cyl         hp     weight       year dieselTRUE
##      0.058      0.152      0.551     -0.234     -0.009
```

```
round(lassVars(lass1, bestid), 3)
```

```
##        cyl     volume         hp     weight      accel       year continent2 continent3 dieselTRUE
##      0.154     -0.187      0.276      0.589      0.067     -0.284     -0.042     -0.049     -0.077
```

Questions? (online/in-class)

Now try this, while allowing for **all two-way interactions.**

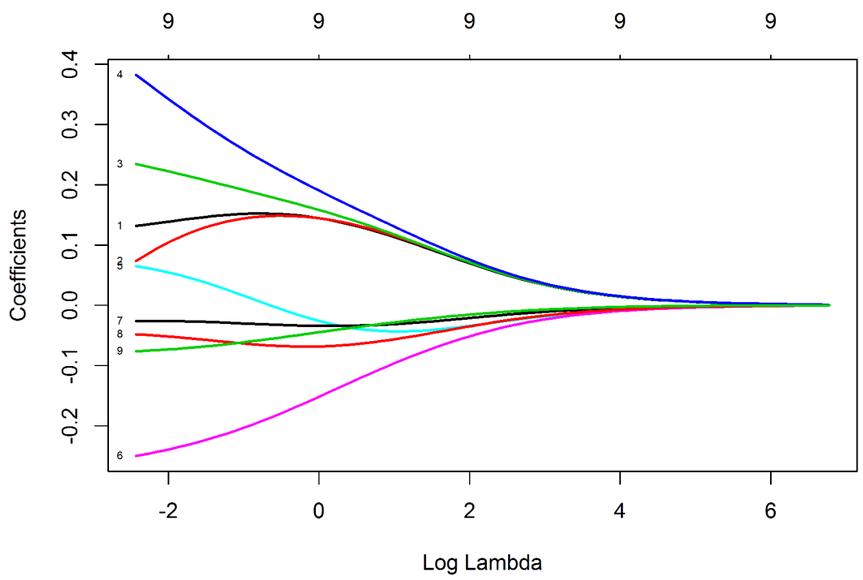
Lasso/Elastic-Nets Options

There's much more to the Lasso than this vanilla implementation:

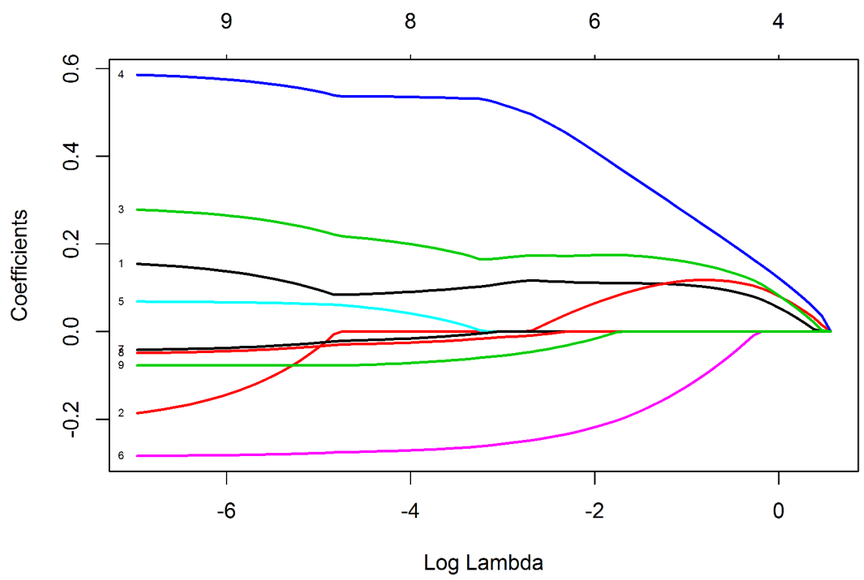
1. `glmnet` as its name suggests, can be run on pretty much any GLM flavor, including logistic and multinomial - **making it a viable option for classification problems.**

2. The name's second half comes from **Elastic Nets**, a fairly recent development from the Lasso team. The penalty is really a linear combination of α Lasso (absolute value) penalty, and $1 - \alpha$ ridge-regression (quadratic) penalty. The default as we ran it, is $\alpha = 1$ or pure Lasso; $\alpha = 0$ is pure ridge.

```
plot(glmnet(alpha = 0, x = scale(automat), y = scale(autos$gp100m)), label = TRUE,
    xvar = "lambda", lwd = 2)
```




```
plot(glmnet(alpha = 0.5, x = scale(automat), y = scale(autos$gp100m)),
    xvar = "lambda", lwd = 2)
```



Handling Multi-Column Covariates via `grpreg`

As said earlier, `glmnet` cannot ensure that grouped covariates remain together. A newer package, however, can do this and a lot more:

```
library(splines)
options(width = 130)
mod1 = lm(gp100m ~ weight + ns(year, knots = c(73, 77, 80)) + continent + hp +
    volume + cyl + accel + diesel, data = autos)
automat2 = model.matrix(mod1)[, -1]
dim(automat2)
```

```
## [1] 298  12
```

```
library(grpreg)
lass2 = grpreg(X = scale(automat2), y = scale(autos$gp100m), penalty = "grLasso",
    group = c(1, rep(2, 4), 3, 3:8))
names(lass2)
```

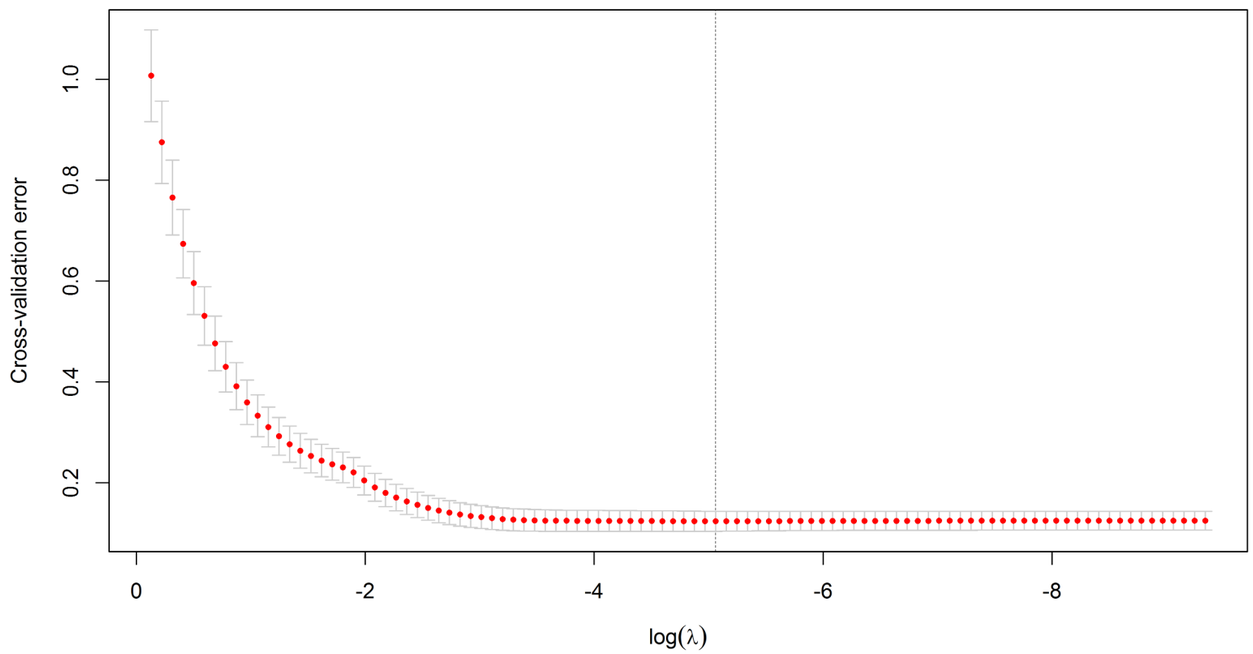
```
##  [1] "beta"    "family"  "group"   "lambda"  "alpha"   "loss"    "n"       "penalty" "df"      "iter"
```

`grpreg` has additional penalty options developed by the package authors (and with which I'm not familiar). It also incorporates the elastic-nets and its `alpha` parameter, and in general has a very similar interface (it asks for capital `X`, tho :)

On the other hand, as of now it can only fit linear and logistic regressions, not the entire suite of GLMs handled by `glmnet`.

Handling Multi-Column Covariates via `grpreg`

```
lass2cv = cv.grpreg(X = scale(automat2), y = scale(autos$gp100m), penalty = "grLasso",
    group = c(1, rep(2, 4), 3, 3:8), nfolds = 5)
plot(lass2cv)  ### where's the 1-SE line when you need it?...
```




```
min(which(lass2cv$cve < lass2cv$cve[lass2cv$min] + lass2cv$cvse[lass2cv$min]))
```

```
## [1] 29
```

Handling Multi-Column Covariates via `grpreg`

Note that `grepreg.cv` doesn't seem to allow you to "bring your own CV groups." Its components are also quite different. Here's how to extract the "as good as best" model.

```
names(lass2cv)
```

```
## [1] "cve"        "cvse"       "lambda"     "fit"        "min"
## [6] "lambda.min"
```

```
asgood = min(which(lass2cv$cve < lass2cv$cve[lass2cv$min] + lass2cv$cvse[lass2cv$min]))
lassVars(lass2, asgood, minabs = 1e-04)
```

```
##                          weight ns(year, knots = c(73, 77, 80))1
##                         0.50711                         -0.04391
## ns(year, knots = c(73, 77, 80))2 ns(year, knots = c(73, 77, 80))3
##                        -0.10429                         -0.05222
## ns(year, knots = c(73, 77, 80))4                              hp
##                        -0.12811                          0.20604
##                             cyl                     dieselTRUE
##                         0.09136                         -0.02426
```

Note: both packages allow you to do the equivalent of the `forced` argument in our little regression-CV function. That is, you can specify covariates that remain above the fray, always in the model. This is good for, e.g., exploring interactions of the dominant `weight` variable in this dataset.

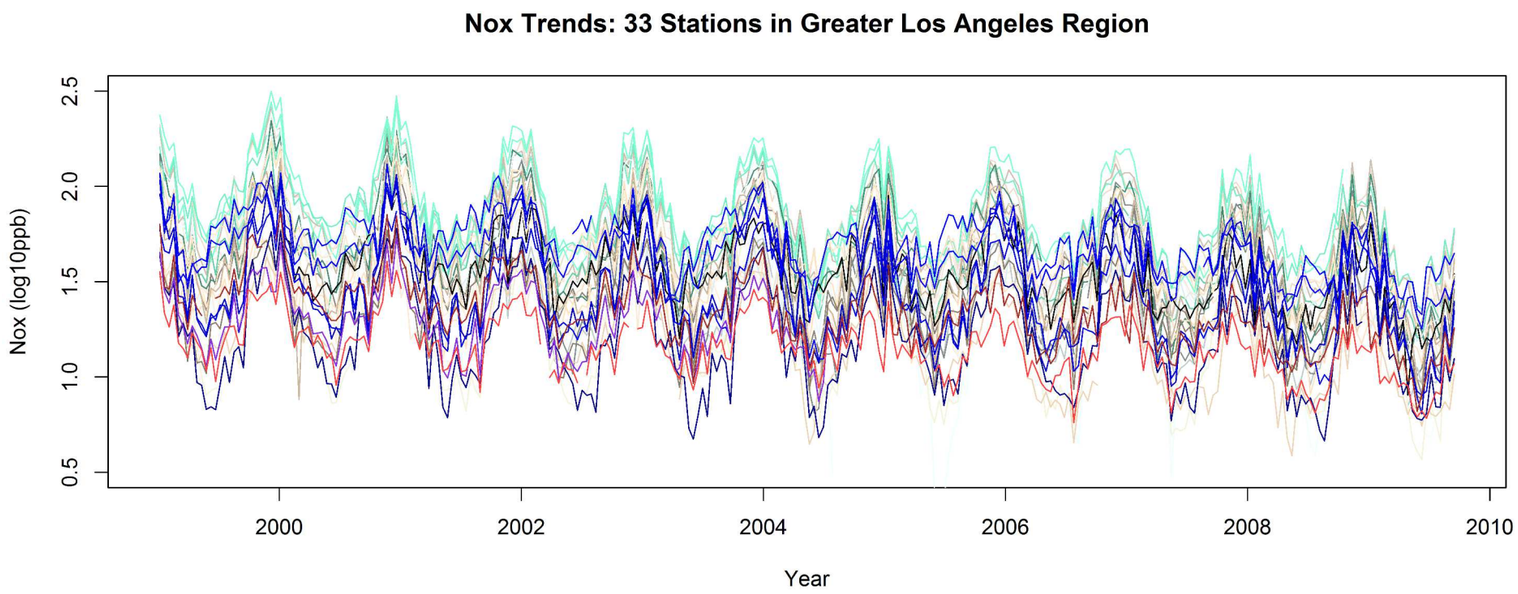
- In `grpreg` this is done by setting `group.multiplier` to zero for the relevant covariate;
- In `glmnet`, `penalty.factor` does the same job.

Questions? (online/in-class)

Dimension Reduction: Singular Value Decomposition and PCA

Another approach to the *"too many covariates"* problem, comes directly from pure linear algebra. Just like one can diagonalize a square matrix and get its **eigenvalues** and **eigenvectors** – a similar trick can be done on a non-square matrix. This is SVD.

```
source("../Code/svdMissing.r")
lanox = read.csv("../Datasets/la11mat33.csv", as.is = T, row.names = 1)
ladates = as.Date(rownames(lanox))
plot(range(ladates), c(0.5, 2.5), type = "n", xlab = "Year", ylab = "Nox (log10ppb)",
    main = "Nox Trends: 33 Stations in Greater Los Angeles Region")
for (a in 1:33) lines(log10(lanox[, a]) ~ ladates, col = colors()[a])
```



Clearly, these trends show a strong seasonality and share a lot in common.

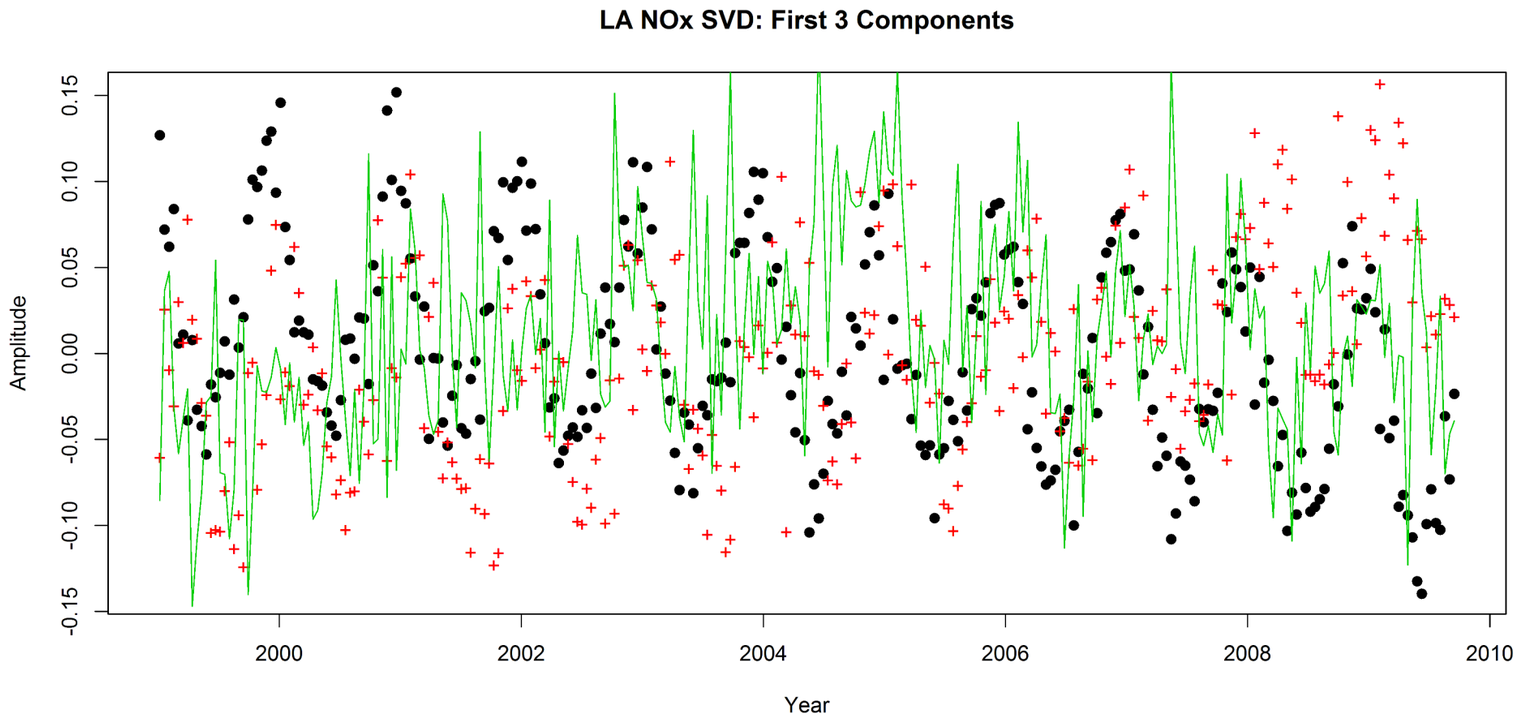
SVD takes $\mathbf{X}$ and decomposes it to a 3-matrix product $\mathbf{X} = \mathbf{UDV}$.

- $\mathbf{U}$ has the same size as $\mathbf{X}$, and is often called **the "scores".**
- $\mathbf{D}$ is a diagonal $p \times p$ matrix, holding the numbers analogous to the eigenvalues (they *are* the eigenvalues of $\mathbf{X}$'s covariance matrix).
- $\mathbf{V}$ is also $p \times p$, storing the **loadings:** how much each original covariate contributes to each "score".

Dimension Reduction: Singular Value Decomposition and PCA

SVD re-arranges the information, by identifying and pooling common patterns, and arranging them from strongest to weakest.

```
source("../Code/svdMissing.r")
lasvd = svdWmissing(log10(lanox))
plot(-lasvd$svd$u[, 1] ~ ladates, pch = 19, xlab = "Year", ylab = "Amplitude",
    main = "LA NOx SVD: First 3 Components")
points(lasvd$svd$u[, 2] ~ ladates, col = 2, pch = "+")
lines(lasvd$svd$u[, 3] ~ ladates, col = 3)
```



- The first pattern (1st column of $\mathbf{U}$; black dots) is the universal NOx signature, peaking near the winter solstice and vice versa.
- The second pattern (red crosses) is unique to the LA basin, and seems to follow the changing frequency and intensity of onshore winds: it peaks in late summer.
- The third pattern (and all subsequent ones) was deemed uninformative, and reflects mostly artifacts related to monitors with partial data.

Note: we needed our own function, in order to handle missing data. With complete data, SVD can be done either directly via the svd function, or via `prcomp`, a more friendly principal-components wrapper. Both are on the standard R installation.

Dimension Reduction: PCA Regression

The idea is extremely simple:

- Perform SVD on the model matrix (sans intercept), to generate the **new** model matrix $\mathbf{U}$;
- Perform CV to choose the number of components; this number is our **tuning parameter.**

The implementation…. a bit tricky.

- It is tempting to just calculate the SVD, then run CV regression p times.
- **However:** when we predict test data or real-life data, are we going to re-calculate the entire dataset's (training+test) SVD each time a data point is added ????
- Of course not. We will keep our component definition fixed.
- **Which means, paradoxically, that during CV we need to calculate the components from scratch for each CV cycle. And unfortunately, in our experience the 'CV' algorithms that come with the package are not reliable - so we'll have to do this by hand.**

PCA: Generating the Components with CV

```
library(pls)
autopca=matrix(NA,nrow=298,ncol=9)
for (a in 1:5)  ## looping over CV groups
{
  ### Note the need for a new data frame, and the syntax
  centers=colMeans(automat[cvid!=a,])
  scales=apply(automat[cvid!=a,],2,sd)
  autoin=data.frame(gp100m=autos$gp100m[cvid!=a],X=I(scale(automat[cvid!=a,])))

  tmpreg=pcr(gp100m~X,data=autoin,ncomp=9) ## the y part doesn't matter here
  autout=data.frame(X=I(scale(automat[cvid==a,],center=centers,scale=scales)))
  autopca[cvid==a,]=predict(tmpreg,newdata=autout,type='scores')
}
```

Note: it is critically important to scale the X matrix when doing dimension reduction! Note that we scale the out-of-sample data according to the in-sample data's moments. As we should.

Questions? (online/in-class)

Wait! We're not done. Now, need to do CV regression to determine the number of components - using the exact same CV groups as before (this is important).

Tip: why not first do `pairsPlus` together with the `gp100m` variable, to know what to expect?…

From PCA to PLS

The motivation for Partial Least Squares (PLS) is that yes, PCA/SVD does a great job of condensing the information.

BUT: the method only cares about the information on the X side. What's the use of condensing information, into vectors that might be very poor at predicting y?

Hence PLS. It is an algorithm closely related to SVD, but it "diagonalizes" X into vectors arranged by the amount of information that's best correlated with y.

Again, in my experience… PLS tends to overfit (that is also the opinion of Hastie *et al.*).

To the algorithm's defense, it originated in chemistry, in diffraction spectra analysis - an application with (I think) an extremely high signal-to-noise ratio. In that context it makes sense. In more typical, noisier datasets, it will correlate your X too closely to the noise that's mixed into y.

The methodology for PLS CV is identical to the one shown above for PCA. In fact, both reside in the `pls` package. PLS uses the `plsr` command, which has identical syntax to `pcr`.

The only difference is that when preparing the data.frame, the y needs to be treated like the X:

- scaled,
- set up as a matrix (with 1 column, if necessary), and
- encapsulated with the `I( )` operator (this operator generates a hierarchy within the data frame, enabling the entire matrix to be called as a single term).