ADDITIONAL TOPICS ON THE LASSO

-STATISTICAL MACHINE LEARNING-

Lecturer: Darren Homrighausen, PhD
\( \ell_1 \)-REGULARIZED REGRESSION

**Reminder** Known as

- ‘lasso’
- ‘basis pursuit’

The estimator satisfies

\[
\hat{\beta}_{\text{lasso}}(t) = \arg\min_{||\beta||_1 \leq t} ||Y - X\beta||_2^2
\]

In its corresponding Lagrangian dual form:

\[
\hat{\beta}_{\text{lasso}}(\lambda) = \arg\min_{\beta} ||Y - X\beta||_2^2 + \lambda||\beta||_1
\]
Some additional topics

1. Grids and cross-validation
2. Sparse matrices: In some cases, most of entries in $\mathbf{X}$ are zero and hence we can store/manipulate $\mathbf{X}$ much cheaper using sparse matrices
3. Elastic net: For use when covariates are highly related to each other
4. Refitted lasso: A proposal for reducing the lasso bias
5. Scaled sparse regression
Grids and cross-validation
Some comments about glmnet

(Note: This section’s examples are in terms of Ridge regression. There are the same problems with lasso and elastic net. I just am picking one for simplicity)

Some further details

- Note that in this figure:

  many solutions have almost the same CV error

In fact, since CV is a risk estimate, it is random

- The lower end point of the grid is somewhat arbitrary chosen
The way that glmnet works is to
1. form a grid of $\lambda$ values,
2. find the cross-validation error for each ridge solution on that grid
3. compute the minimum cross-validated $\lambda$: $\hat{\lambda}$
4. report $\hat{\beta}_{\text{ridge}}(\hat{\lambda})$ as the final solution

The important piece is that the final solution depends on which grid we choose

The function `cv.glmnet` comes with a plotting function

```r
ridge.cv = cv.glmnet(x=X,y=Y,alpha=0)
plot(ridge.cv)
```
**Some Comments about glmnet**

- The left-most dotted, vertical line occurs at the *CV* minimum
- The right-most dotted, vertical line is the
  - largest value of $\lambda$ ...
  - such that the error is within one standard-error of the minimum

(the so called one-standard-error rule)
Though glmnet automatically allocates a grid, it isn’t necessary any good

Sometimes...

- the grid values are too far apart near the minimum
- the grid doesn’t allow small/large enough $\lambda$ values
Some comments about glmnet

Example of a bad minimum: Grid values too far apart
Some comments about glmnet

Example of a bad minimum: Grid values too large

How to fix it:

```r
ridge.cv = cv.glmnet(x=X,y=Y,alpha=0)
min.lambda = min(ridge.cv$lambda)
lambda.new = seq(min.lambda,min.lambda*.001,length=100)
ridge.cv = cv.glmnet(x=X,y=Y,alpha=0,lambda=lambda.new)
lambda.hat = ridge.cv$lambda[which.min(ridge.cv$cvm)]
```
Some comments about glmnet

New minimum, after moving $\lambda$ grid **smaller:**
Sparse matrices
Sparse matrices

load("../data/hiv.rda")
X = hiv.train$x
> X[5:12,1:10]

      p1 p2 p3 p4 p5 p6 p7 p8 p9 p10
[1,]  0  0  0  0  0  0  0  0  0  0
[2,]  0  0  0  0  0  0  0  0  0  0
[3,]  0  0  0  0  0  0  0  0  0  0
[4,]  0  0  0  0  0  0  0  0  0  0
[5,]  0  0  0  0  0  0  0  1  0  0
[6,]  0  0  0  0  0  0  0  0  0  0
[7,]  1  0  0  0  0  0  0  0  0  0
[8,]  0  0  0  0  0  0  0  0  0  0

Many zero entries!
Sparse matrices

All numbers in R take up the same space
(Space in this context means RAM aka memory)

```r
> print(object.size(0),units='auto')
48 bytes
> print(object.size(pi),units='auto')
48 bytes
```

Idea: If we can tell R in advance which entries are zero, it doesn’t need to save that number
Sparse matrices

This can be accomplished in several ways in R

One is with the Matrix package

library('Matrix')

Xspar = Matrix(X,sparse=T)
Sparse matrices

Let's take a look at the space difference

```python
> print(object.size(X), units='auto')
1.1 Mb
> print(object.size(Xspar), units='auto')
140.7 Kb
```

Pretty substantial! Only 12.1% as large
Sparse matrices

Lastly, we can create sparse matrices without having the original matrix $X$ ever in memory.

This is usually done with three vectors of the same length:

- A vector with row numbers
- A vector with column numbers
- A vector with the entry value

```r
i = c(1, 2, 2)
j = c(2, 2, 3)
val = c(pi, 1.01, 100)

sparseMat = sparseMatrix(i = i, j = j, x = val, dims = c(4, 4))

regularMat = as(Matrix(sparseMat, sparse = F), 'dgeMatrix')
```
Sparse matrices

```r
> print(sparseMat)
4 x 4 sparse Matrix of class "dgCMatrix"

[1,] 3.141593 . .
[2,] 1.010000 100 .
[3,] . . . .
[4,] . . . .

> print(regularMat)
4 x 4 Matrix of class "dgeMatrix"

[1,]  0 3.141593  0  0
[2,]  0 1.010000 100  0
[3,]  0  0.000000  0  0
[4,]  0  0.000000  0  0
```
Sparse matrices ‘act’ like regular (dense) matrices

They just only keep track of which entries are non zero and perform the operation on these entries

For our purposes, glmnet (and other methods) automatically check to see if $X$ is a sparse matrix object

This can be a substantial speed/storage savings for large, sparse matrices
The full SVD takes $O(\min\{n^2p + p^3\})$ operations
(This can be done with the `svd` function in R)

```r
svd_out = svd(X, nu=nu, nv=nv)
U = svd.out$u
V = svd.out$v
D = diag(svd.out$d)
```

**nu** $\in \{0, n\}$, **nv** $\in \{0, p\}$

**Note:** Though the parameters can be set to intermediary values, these are ignored
Often, we only need a few \((q)\) singular values/ vectors For this, we can use Krylov subspace techniques in \(O(\min\{npq\})\)

(This can be done with the \texttt{irlba} package in \texttt{R})

The \texttt{irlba} function leverages the sparse matrix data structure

\[
\text{svd\_out} = \text{irlba}(X, \text{nu}=\text{nu}, \text{nv}=\text{nv})
\]

\[
U = \text{svd\_out}\_u \\
V = \text{svd\_out}\_v \\
D = \text{diag}(\text{svd\_out}\_d)
\]

\(\text{nu} \in [0, n], \text{nv} \in [0, p]\)

**Example:** The netflix prize dataset was 480,189 rows by 17,770 columns with 100,480,507 non-zero entries

This can be computed in seconds on many computers
The \texttt{irlba} function comes with additional choices:

- \textbf{adjust}: With \texttt{irlba}, you don’t want to just compute $q$ singular vectors if you need $q$, instead compute $q + \text{adjust}$ to enhance convergence.
  (More is better, but 5 is usually fine)

- \textbf{maxit}: \texttt{irlba} is iterative by nature. Check the output object \texttt{iter} to make sure the computation didn’t terminate based on iterations.
Elastic net
**Elastic net**

The ridge solution is always unique and does well when the covariates are highly related to each other:

$$\hat{\beta}_{\text{ridge}, \lambda} = \arg\min_{\beta} ||Y - X\beta||_2^2 + \lambda ||\beta||_2^2 = (X^T X + \lambda I)^{-1} X^T Y$$

The lasso solution

$$\hat{\beta}_{\text{lasso}, \lambda} = \arg\min_{\beta} ||Y - X\beta||_2^2 + \lambda ||\beta||_1$$

isn’t necessarily unique, but it can do model selection

However, it can do poorly at model selection if the covariates are highly related to each other
The elastic net was introduced to combine both of these behaviors. It solves

\[
\hat{\beta}_{\alpha,\lambda} = \arg\min_{\beta} \left[ \|Y - X\beta\|_2^2 + \lambda \left( (1 - \alpha)\|\beta\|_2^2 + \alpha\|\beta\|_1 \right) \right]
\]

We can do the elastic net in R with `glmnet`

```r
alpha = 0.5
out.elasticNet = glmnet(x = X, y = Y, alpha=alpha)
```

The parameter `alpha` needs to be set.

There does not exist any convention for this, but CV can be used.
(You have to write this up yourself, though. Usually, people just play around with different values)
Refitted lasso
Refitted lasso

Since lasso does both

- regularization
- model selection

it can produce a solution that produces too much bias

A common approach is to do the following two steps:

1. choose the $\lambda$ via the ‘one-standard-error rule’
2. refit the (unregularized) least squares solution on the selected covariates
**Refitted lasso**

We can do this in R via

```r
# Get CV curve
lasso.cv.glmnet = cv.glmnet(X,Y,alpha=1)

# Get beta hat with one-standard-error rule
# (remove intercept index -> [-1])
betaHat.temp = coef(lasso.cv.glmnet,s='lambda.1se')[-1]
# Identify which covariates are nonzero
selectedCovs = which(abs(betaHat.temp) > 1e-16)

# Run regular least squares using those covariates
refitted.lm = lm(Y~.,data=data.frame(X[,selectedCovs]))

## Output: either predictions or coefficients
##
Yhat.refitted = predict(refitted.lm,X_0[,selectedCovs])
betaHat.refitted = refitted.lm$coefficients```

Refitted lasso

**IMPORTANT:** Do not attempt to do inference with the reported p-values. These are absolutely not valid!

However, the parameter values are estimates of the effect of that covariate.
Scaled-sparse regression
Theoretically, the optimal value for $\lambda$ looks like:

$$\lambda = C \sigma \sqrt{\frac{n}{\log(p)}}$$

for some constant $C$.

- If we knew the true $\sigma$, we could find an optimal $\lambda$
- If we knew the optimal $\lambda$, we could find the $\sigma$

This speaks to using an iterative approach
Scaled sparse regression (SSR) jointly estimates the regression coefficients and noise level in a linear model.

It alternates between

1. estimating $\sigma$ via

$$\hat{\sigma} = \sqrt{\frac{1}{n} \left\| \mathbf{Y} - \mathbf{X}\hat{\beta}_{\text{lasso}}(\lambda) \right\|_2^2}$$

2. setting

$$\lambda = C\hat{\sigma} \sqrt{\frac{n}{\log(p)}}$$

($C$ is usually set to something like $1/2$)
Scaled-sparse regression

We can do this in R via

```r
library(scalreg)
lasso.ssr = scalreg(X = X, y = Y, LSE=F)
> names(lasso.ssr)
[1] "hsigma"    "coefficients" "residuals"
[4] "fitted.values" "type"     "call"
```
Scaled-sparse regression

Also, the LSE parameter indicates if we want to do refitted lasso

Running

```r
lasso.ssr = scalreg(X = X, y = Y, LSE=T)
```

Creates an object `lse`

```r
> names(lasso.ssr)
[1] "hsigma" "coefficients" "residuals"
[4] "fitted.values" "type" "lse"
```

This object has all the relevant information. For instance predictions

```r
Yhat.ssr.refitted = X_0 %*% lasso.ssr$lse$coefficients
```