Additional topics on the lasso

-Statistical Machine Learning-

Lecturer: Darren Homrighausen, PhD

ℓ_1-REGULARIZED REGRESSION

REMINDER Known as

- 'lasso'
- 'basis pursuit'

The estimator satisfies

$$\hat{\beta}_{\text{lasso}}(t) = \underset{||\beta||_1 \leq t}{\operatorname{argmin}} ||\mathbb{Y} - \mathbb{X}\beta||_2^2$$

In its corresponding Lagrangian dual form:

$$\hat{\beta}_{\text{lasso}}(\lambda) = \underset{\beta}{\operatorname{argmin}} ||\mathbb{Y} - \mathbb{X}\beta||_2^2 + \lambda ||\beta||_1$$

Some additional topics

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

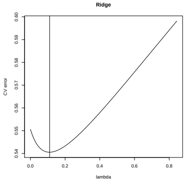
Grids and cross-validation

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

Some comments about glmnet

The way that glmnet works is to

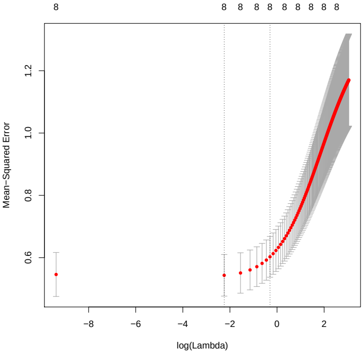
1. form a grid of λ values,
2. find the cross-validation error for each ridge solution on that grid
3. compute the minimum cross-validated λ: $\hat{\lambda}$
4. report $\hat{\beta}_{\mathrm{ridge}}(\hat{\lambda})$ as the final solution

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

The function cv.glmnet comes with a plotting function

```
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 λ ...
  - such that the error is within one standard-error of the minimum

(the so called one-standard-error rule)

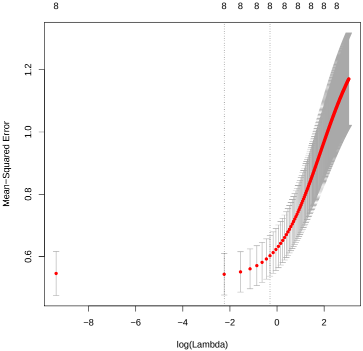
Some comments about glmnet

Though glmnet automatically allocates a grid, it isn't necessary any good

Sometimes...

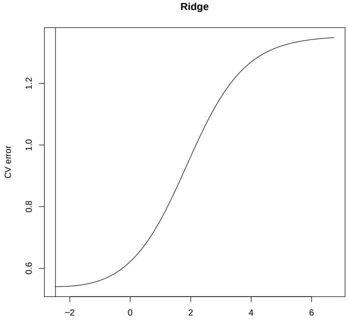
- the grid values are too far apart near the minimum
- the grid doesn't allow small/large enough λ values

Some comments about glmnet



Example of a bad minimum: Grid values too far apart

Some comments about glmnet



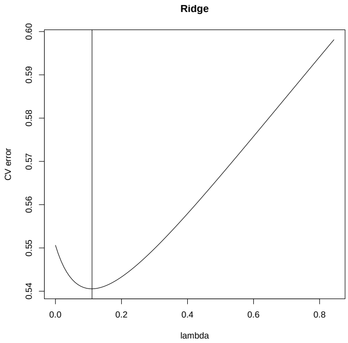
Example of a bad minimum: Grid values too large

How to fix it:

```
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 λ 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)

```
> 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

```
> 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 $\mathbb{X}$ 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

```
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

```
> 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]     [,2] [,3] [,4]
[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

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 $\mathbb{X}$ is a sparse matrix object

This can be a substantial speed/storage savings for large, sparse matrices

SVD

The full SVD takes $O(\min\{n^2p + p^3\})$ operations

(This can be done with the svd function in 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

SVD

Often, we only need a few (q) singular values/ vectors For this, we can use Krylov subspace techniques in $O(npq)$

(This can be done with the irlba package in R)

The irlba function leverages the sparse matrix data structure

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

nu $\in [0, n]$, nv $\in [0, p]$

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

SVD

The irlba function comes with additional choices:

- adjust: With irlba, you don't want to just compute q singular vectors if you need q, instead compute $q + \mathrm{adjust}$ to enhance convergence.

  (More is better, but 5 is usually fine)
- maxit: irlba is iterative by nature. Check the output object iter to make sure the computation didn't terminate based on iterations.

Alternatively, there are other iterative and even parallel SVD solvers (SLEPc is a major one (stands for Scalable Library for Eigenvalue Problem Computations))

Elastic net

Elastic net

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

$$\hat{\beta}_{ridge}(\lambda) = \underset{\beta}{\operatorname{argmin}} ||\mathbb{Y} - \mathbb{X}\beta||_2^2 + \lambda||\beta||_2^2 = (\mathbb{X}^\top\mathbb{X} + \lambda I)^{-1}\mathbb{X}^\top Y$$

The lasso solution

$$\hat{\beta}_{\mathrm{lasso}}(\lambda) = \underset{\beta}{\operatorname{argmin}} ||\mathbb{Y} - \mathbb{X}\beta||_2^2 + \lambda||\beta||_1$$

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

However, it can do poorly at model selection if the features are highly related to each other

Elastic net

The elastic net was introduced to combine both of these behaviors

It solves

$$\hat{\beta}_{\alpha,\lambda} = \underset{\beta}{\text{argmin}} \left[||\mathbb{Y} - \mathbb{X}\beta||_2^2 + \lambda \left((1-\alpha)||\beta||_2^2 + \alpha||\beta||_1 \right) \right]$$

We can do the elastic net in R with glmnet

```
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 λ via the ‘one-standard-error rule’
2. refit the (unregularized) least squares solution on the selected features

Refitted lasso

We can do this in R via

```
X = matrix(rnorm(50),nrow=10,ncol=5)
Y = X %*% c(1,2,0,0,0) +rnorm(10)

Xtest = matrix(rnorm(50),nrow=10,ncol=5)
Ytest = Xtest %*% c(1,2,0,0,0) +rnorm(10) + 1

require(glmnet)
#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 features are nonzero
selectedFeatures = which(abs(betaHat.temp) > 1e-16)

# Run regular least squares using those features
refitted.lm  = lm(Y˜X[,selectedFeatures])
```

Refitted lasso: part 2

Continuing...

```
Yhat.refit    = drop(Xtest[,selectedFeatures] %*%
                   coef(refitted.lm)[-1] + coef(refitted.lm)[1])
betaHat.refit = as.numeric(refitted.lm$coefficients)
Yhat.lasso    = Xtest %*%
                 coef(lasso.cv.glmnet,s=’lambda.min’)[-1] +
                 coef(lasso.cv.glmnet,s=’lambda.min’)[1]
betaHat.lasso = as.numeric(coef(lasso.cv.glmnet,s=’lambda.min’))

> cat(’Refitted lasso: ’,betaHat.refit,
+     ’ with indices: ’,selectedFeatures,’\n’)
Refitted lasso: -0.10899 0.96786 2.0931  with indices:  1 2
> print(betaHat.lasso)
[1] -0.16044  0.73933  1.78871  0.00000 0.26314  0.00000
>
> print( sum((Ytest - Yhat.refit)**2) )
[1] 22.69664
> print( sum((Ytest - Yhat.lasso)**2) )
[1] 26.64522
```

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 feature

Scaled-sparse regression

Scaled-sparse regression

Theoretically, the optimal value for λ looks like:

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

for some constant C.

- If we knew the true σ, we could find an optimal λ
- If we knew the optimal λ, we could find the σ

This speaks to using an iterative approach

Scaled-sparse regression

Scaled sparse regression (SSR) jointly estimates the regression coefficients and noise level in a linear model

It alternates between

1. estimating σ via

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

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

```
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

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

Creates an object lse

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

This object has all the relevant information. For instance predictions

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

Rules for discarding features

SAFE rule

Compute all the inner products $x_j^\top Y$, $||x_j||_2$, and $||Y||_2$

Let $\lambda_{\max} := \max_{1\leq j\leq p}|x_j^\top Y|$ and fix a $\lambda \in (0, \lambda_{\max}]$
(Remember that $\hat{\beta}_{\mathrm{lasso}}(\lambda_{\max}) \equiv 0$)

Then the SAfe Feature Elimination (SAFE) rule is

$$|x_j^\top Y| \leq \lambda - \frac{||x_j||_2\,||Y||_2}{\lambda_{\max}}(\lambda_{\max} - \lambda)$$

implies that $\hat{\beta}_{j,\mathrm{lasso}}(\lambda) = 0$ (El Ghaoui et al. (2010))

There has been additional research on this topic

The Basic strong rule discards the j^{th} feature if

$$|x_j^\top Y| \leq \lambda - \lambda_{\max}$$

(Tibshirani, R et al. (2012). Can erroneously eliminate features!)

Other rules

There are other SAFE/strong rules for other methods

- logistic lasso
- Support vector machines

Additionally, a commonly used method (which actually preserves significance tests!) is to set $\hat{\beta}_j = 0$ if $||x_j - \overline{x}_j||_2 \leq \eta$ for some threshold η

(This rule is most often used in conjunction with “False Discovery Rate” or “multiple comparisons” type of procedure)

Intuitively, there isn't enough information along X_j to estimate $\hat{\beta}_j$ anyways

Also, there is Sure independence screening which retains the j largest w_j of:

$$w = \frac{1}{n}\mathbb{X}^\top Y$$

($\mathbb{X}$ and Y are standardized. Fan, Lv (2008))