Intro to Sparse Modelling

Extremely Brief Intro to Sparse Modelling

This is the exercise for a Biol 5081 class at York, Fall 2023

I recommend these two videos as a very basic jumping off point:\

https://www.youtube.com/watch?v=Q81RR3yKn30//

https://www.youtube.com/watch?v=NGf0voTMlcs//

Just as a reminder after having seen either these videos or my powerpoint: Sparse Modelling is a general framework to 1) choose among predictor values when n less than p, 2) increase the interpretablity of models (using fewer predictors) and 3) prevent overfitting (hopefully improving out-of-sample prediction).

There are many different ways to do sparse modelling, which have different pros and cons, and have historically been used with more or less frequency by different fields.

Examples I went through are: 1) ridge regression 2) LASSO 3) elastic net

I also briefly mentioned GEMMA's BSLMM and SuSie.

I did not get into Bayesian Lasso (BLASSO), with or without Reversible Jump (RJ), or Random Forest, which we've talked about before.

```r
if (!require("monomvn")) install.packages("monomvn")
library("monomvn")
if (!require("glmnet")) install.packages("glmnet")
library("glmnet")
if (!require("MASS")) install.packages("MASS")
library("MASS")
```

Before we really do anything, I'm going to simulate some data for us. First, I'm going to simulate a ton of multivariate normal parameters, and then I'm going to choose 5 to make y dependent on. I've been stealing simulation ideas from: https://aosmith.rbind.io/2018/04/23/simulate-simulate-part-2/#a-single-simulation-for-the-two-level-model

```r
set.seed(42)
mu <- rep(0, 2000)  ### this gives the means of the variables, and the number of variables
### we're going to simulate 1000 individuals, but we're only going to analyze 800 -
### then try to predict the other 200
Sigma <- diag(2000)
### simulates the predictor variable, 2000 predictors for 100 individuals
x <- t(matrix(mvrnorm(n = 1000, mu = mu, Sigma = Sigma), nrow = length(mu)))
Y <- 1.1 * x[, 1] + 0.7 * x[, 2] + 0.3 * x[, 3] + 0.15 * x[, 4] + 0.05 * x[, 5] +
    rnorm(1000, 0, 0.5)  ### simulates the response variable

## for later, this means our real slopes should be:
sim_slopes <- c(1.1, 0.7, 0.3, 0.15, 0.05, rep(0, 1995))
```

Excellent, now we have some data, I'm gonna run through ridge, lasso and elastic net in glmnet. Ridge and lasso can both be done in monomvn as well, but as far as I can tell, elasticnet can't be, so ymmv.

As we talked about, ridge and lasso are regularization methods that use a lambda penalty to shrink the slope of each parameter, so that the parameter estimates are less dependent on the particular sample data. lambda is chosen using cross validation (CV). The difference between ridge and lasso is that ridge multiples the lambda against the slope^2, whereas lasso multiples lambda against abs(slope), when trying to minimize sum of squares.

ridge minimizes sum of squares+lambda*slope^2 lasso minimizes sum of squares+lambda*abs(slope)

Because of this difference, lasso can force slopes to 0, whereas ridge cannot.

Elastic net is a mix of both.

Here's the intro to glmnet: https://glmnet.stanford.edu/articles/glmnet.html.

NB: alpha is the elasticnet mixing parameter, with alpha=0 is ridge, 1 is LASSO and anything between is a form of elasticnet.

NB2: Lambda is fit differently for each parameter - I don't know why, but this messed up my thinking a bunch.

```r
elastic.cv <- cv.glmnet(x[1:800, ], Y[1:800], type.measure = "deviance", nfolds = 5,
    alpha = 0.5)
### this ask about quality of inference - are the betas close?
cor.test(coef(elastic.cv)[2:2001, ], sim_slopes)
predicted_y_elastic <- predict(elastic.cv, newx = x[801:1000, ])
cor.test(Y[801:1000], predicted_y_elastic)

ridge.cv <- cv.glmnet(x[1:800, ], Y[1:800], type.measure = "deviance", nfolds = 5,
    alpha = 0)
### this ask about quality of inference - are the betas close?
cor.test(coef(ridge.cv)[2:2001, ], sim_slopes)
predicted_y_ridge <- predict(ridge.cv, newx = x[801:1000, ])
cor.test(Y[801:1000], predicted_y_ridge)

lasso.cv <- cv.glmnet(x[1:800, ], Y[1:800], type.measure = "deviance", nfolds = 5,
    alpha = 1)
### this ask about quality of inference - are the betas close?
cor.test(coef(lasso.cv)[2:2001, ], sim_slopes)
predicted_y_lasso <- predict(lasso.cv, newx = x[801:1000, ])
cor.test(Y[801:1000], predicted_y_lasso)
```

Questions to ponder: which method was the best method for inference? Which method was the best for prediction. Why is this?

glmnet has a bunch of internal functions (like coef above, note I used matrix(coef) because otherwise it prints . instead of 0 because sparse), print and predict. Predict is of course useful for when we have predictor variables, but no response variables.

Choose your own adventures:

Simulate GWAS for unicorn horns, mass, and colour variation:

This simulation is based on Alex Buerkle's Population Genetics lab notes.

Some things to note about these simulations - loci are assumed to be independent. This isn't realistic, there would certainly be LD between markers. Secondly, I haven't simulated chromosomes, just allele frequencies.

```
set.seed(42)
nloci <- 10000  ## we will have data from nloci
nind <- 2000  ## we will have data from nind that were sampled from the true population
## high parameter means high polymorphism, must be positive and > zero, can be
## thought of as analygous to nucleotide polymorphism
sim.theta <- 5

## simulate allele frequencies at nloci number of loci, by random draws from a
## beta
sim.p <- rbeta(nloci, sim.theta, sim.theta)


# simulate genotypes in the sample this gives us individuals in rows and snps in
# columns
sim.x <- matrix(rbinom(nloci * nind, 2, prob = sim.p), nrow = nind, ncol = nloci)
str(sim.x)
```

```
##  int [1:2000, 1:10000] 0 2 1 2 1 2 1 1 2 1 ...
```

```
### one more thing to simulate is the phenotypes, and the number of markers that
### they're based on.

### let's do one that's categorical and based on a few SNPs, like the horn traits
### in soay sheep

### large effect sizes, not a lot of error
y_cat <- 5 + 0.7 * sim.x[, 1] + 0.7 * sim.x[, 2] + 0.7 * sim.x[, 3] + 0.7 * sim.x[,
    4] + rnorm(2000, 0, 1)

### more snps, larger error, smaller effect sizes
y_quant <- 0.35 * sim.x[, 1] + 0.35 * sim.x[2] + 0.35 * sim.x[, 3] + 0.35 * sim.x[,
    4] - 0.35 * sim.x[, 5] - 0.35 * sim.x[, 6] - 0.35 * sim.x[, 7] + 0.35 * sim.x[,
    8] + 0.35 * sim.x[, 9] + 0.35 * sim.x[, 10] + rnorm(2000, 0, 2)

### 100 snps, larger error, quite small effect sizes
y_polygenic <- 0.1 * sim.x[, 1] + 0.1 * sim.x[, 2] + 0.1 * sim.x[, 3] + 0.1 * sim.x[,
    4] + 0.1 * sim.x[, 5] + 0.1 * sim.x[, 6] + 0.1 * sim.x[, 7] + 0.1 * sim.x[, 8] +
    0.1 * sim.x[, 9] + 0.1 * sim.x[, 10] + 0.1 * sim.x[, 11] + 0.1 * sim.x[, 12] +
    0.1 * sim.x[, 13] + 0.1 * sim.x[, 14] + 0.1 * sim.x[, 15] + 0.1 * sim.x[, 16] +
    0.1 * sim.x[, 17] + 0.1 * sim.x[, 18] + 0.1 * sim.x[, 19] + 0.1 * sim.x[, 20] +
    0.1 * sim.x[, 21] + 0.1 * sim.x[, 22] + 0.1 * sim.x[, 23] + 0.1 * sim.x[, 24] +
    0.1 * sim.x[, 25] + 0.1 * sim.x[, 26] + 0.1 * sim.x[, 27] + 0.1 * sim.x[, 28] +
    0.1 * sim.x[, 29] + 0.1 * sim.x[, 30] + 0.1 * sim.x[, 31] + 0.1 * sim.x[, 32] +
    0.1 * sim.x[, 33] + 0.1 * sim.x[, 34] + 0.1 * sim.x[, 35] + 0.1 * sim.x[, 36] +
    0.1 * sim.x[, 37] + 0.1 * sim.x[, 38] + 0.1 * sim.x[, 39] + 0.1 * sim.x[, 40] +
    0.1 * sim.x[, 41] + 0.1 * sim.x[, 42] + 0.1 * sim.x[, 43] + 0.1 * sim.x[, 44] +
```

```r
    0.1 * sim.x[, 45] + 0.1 * sim.x[, 46] + 0.1 * sim.x[, 47] + 0.1 * sim.x[, 48] +
    0.1 * sim.x[, 49] + 0.1 * sim.x[, 50] + 0.1 * sim.x[, 51] + 0.1 * sim.x[, 52] +
    0.1 * sim.x[, 53] + 0.1 * sim.x[, 54] + 0.1 * sim.x[, 55] + 0.1 * sim.x[, 56] +
    0.1 * sim.x[, 57] + 0.1 * sim.x[, 58] + 0.1 * sim.x[, 59] + 0.1 * sim.x[, 60] +
    0.1 * sim.x[, 61] + 0.1 * sim.x[, 62] + 0.1 * sim.x[, 63] + 0.1 * sim.x[, 64] +
    0.1 * sim.x[, 65] + 0.1 * sim.x[, 66] + 0.1 * sim.x[, 67] + 0.1 * sim.x[, 68] +
    0.1 * sim.x[, 69] + 0.1 * sim.x[, 70] + 0.1 * sim.x[, 71] + 0.1 * sim.x[, 72] +
    0.1 * sim.x[, 73] + 0.1 * sim.x[, 74] + 0.1 * sim.x[, 75] + 0.1 * sim.x[, 76] +
    0.1 * sim.x[, 77] + 0.1 * sim.x[, 78] + 0.1 * sim.x[, 79] + 0.1 * sim.x[, 80] +
    0.1 * sim.x[, 81] + 0.1 * sim.x[, 82] + 0.1 * sim.x[, 83] + 0.1 * sim.x[, 84] +
    0.1 * sim.x[, 85] + 0.1 * sim.x[, 86] + 0.1 * sim.x[, 87] + 0.1 * sim.x[, 88] +
    0.1 * sim.x[, 89] + 0.1 * sim.x[, 90] + 0.1 * sim.x[, 91] + 0.1 * sim.x[, 92] +
    0.1 * sim.x[, 93] + 0.1 * sim.x[, 94] + 0.1 * sim.x[, 95] + 0.1 * sim.x[, 96] +
    0.1 * sim.x[, 97] + 0.1 * sim.x[, 98] + 0.1 * sim.x[, 99] + 0.1 * sim.x[, 100] +
    rnorm(2000, 0, 3)

### splitting up testing and training for the exercise
sim.x.training <- sim.x[1:1600, ]
sim.x.test <- sim.x[1601:2000, ]

y_cat.training <- y_cat[1:1600]
y_cat.test <- y_cat[1601:2000]

y_quant.training <- y_quant[1:1600]
y_quant.test <- y_quant[1601:2000]

y_polygenic.training <- y_polygenic[1:1600]
y_polygenic.test <- y_polygenic[1601:2000]
```

Here, I've simulated three different genetic architectures, and set testing and training data - In groups, analyze the Unicorn data with Lasso, Ridge and ElasticNet. Which genetic architectures can be recovered with these different methods?

In groups, analyze the Unicorn data with Lasso, Ridge and ElasticNet.

Which method recovers all of the SNPs (i.e. has non-zero estimates for the SNPs that you simulated)?

Which method best predicts OOS unicorn horns?

What happens when you use different genetic architectures (i.e. y_cat vs y_quant vs y_polygenic)?

Extra time - what happens if you change the training/testing individuals? How does this change your perception of out of sample prediction?