# Modeling Tips and Elastic Net

MKTG 6279

Week 4

# Today

- Learning how to learn new methods
- Example with extreme gradient boosting
- LASSO, Ridge, and Elastic Net for dimension reduction
- Example: setting up ML model comparisons

# Analysis tips and learning how to learn

# Why I tell you this

We're reaching a point (between APA I and this course) where we've covered a lot of the basics

- Linear regression
- Logistic regression
- A/B Testing
- Cluster analysis

With only a few weeks left, it's worth thinking about how to become a self-sufficient analyst

# Tip 1: the learning (i.e., Googling) never stops

It should be painfully obvious now that the world of analysis is complicated

- Visualization with ggplot
- Data manipulation with dplyr
- Report generation with Tableau or PowerBI
- Statistical methods for estimation
- Econometric methods for endogeneity
- Computer science methods for processing efficiency
- etc.

# Tip 2: keep it simple, and let the analysis tell the story

What I mean by this: do not underestimate the power of exploratory analysis!

The exploratory analysis guides the model, the model guides the recommendations, the recommendations your future

- In this way, the data tells the story for you

Too often I see either:

1) analysts try to force models onto situations that don't need it
2) analysts rely solely on EDA, and the model is an afterthought

# What else should I learn?

Basically anything in this book: https://www.statlearning.com/

You will be better off knowing how to work with a few methods very well than not being able to use a lot of methods at all

- Start small: if you struggle with t-tests, practice a few of those before moving onto more complicated methods
- There is an inverse relationship between how complicated a method is and how often you will use it

# But since some of you can't resist fancy methods...

Let's add a new skill: extreme gradient boosting

Why? I will show you my process for learning new methods.

You can practice *learning how to learn* - which is more important than any single method in this class.

# Before you begin

Things to keep in mind:

- Don't try (or even bother) to be an expert on every method (except maybe linear regression)
- Get a sense of *when* and *why* you might want to consider a method, but that is about it
- Practice simulating data, implementing the method, and seeing which parameters need to be tuned

Note: I have to do this **all the time**

# Why extreme gradient boosting?

I've heard it is useful, I've used the code a couple times for fun, but I don't really know much more than that.

**Step 1**: Google and skim (takes ~20 minutes)

- At first you're trying to intake a bunch of information about it
- https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/
- https://cran.r-project.org/web/packages/xgboost/vignettes/xgboost.pdf

"It is an efficient and scalable implementation of gradient boosting framework by (Friedman, 2001) (Friedman et al., 2000)."

# Step 2: check the book, chapter 8: Tree Based Methods

- Decision trees divide the $X$ space into distinct subsets, and then forms predictions on those subsets of data
- "Bagging, random forests, boosting, and Bayesian additive regression trees" produces multiple trees and then combines for a single prediction
- Combining many trees does a better job than a single tree

# So what is "Boosting"?

Read section 8.2.3 (takes ~20 minutes)

**Bagging**: sample from the original data into many independent copies, fit a decision tree on each sample, and then average the predictions (essentially bootstrapping)

**Boosting**: fit a tree, shrink the influence and add to original prediction, update the residuals with this new tree

Biggest difference: boosting grows trees sequentially, based on remaining residuals whereas bagging fits many trees independently

See Algorithm 8.2 in the book

# Recap so far

Find terms you recognize, see where this new method fits in

At this point we have a rough idea of what boosting is and how it is a little different

No code yet, no real math - just a conceptual understanding

# So what makes it extreme?

Spend another 10-15 minutes here to refine the concept:

https://en.wikipedia.org/wiki/Gradient_boosting

First, the "gradient" in gradient boosting refers the first derivative of the loss function.

Why do we care? This is how the method knows which "direction" to take to improve the model fit after it finds the residuals.

See more here: https://en.wikipedia.org/wiki/XGBoost

The "extreme" version uses a particular function to navigate the gradient descent - which turns out to be pretty effective.

Is it worth knowing more than this? Maybe, but probably not.

# Check that it works with simulated data

Compare accuracy with standard OLS.

First simulate some data:

```
n     = 50000
beta = c(10,2,4)
X = cbind(1,runif(n),runif(n))
sigma = 3
y = X%*%beta + rnorm(n,sd=sigma)
df = data.frame(y,x1=X[,2],x2=X[,3])
head(df)
```

```
##           y         x1         x2
## 1 13.918329 0.01533534 0.35208213
## 2 14.659576 0.16323121 0.69485211
## 3 16.253969 0.82685780 0.17322415
## 4  7.384394 0.88858032 0.34139136
## 5 12.561312 0.42491623 0.36812010
## 6 16.490314 0.29856383 0.08848559
```

# Check the coefficients

Okay, so ols works:

```
#ols
ols = lm(y~x1+x2,df)
coef(ols)
```

```
## (Intercept)          x1          x2
##    9.980006    2.026759    3.993226
```

# What's the fit of the OLS method?

```
yhat_ols = predict(ols)

#Mean squared error (MSE)
mean((df$y - yhat_ols)^2)
```

```
## [1] 8.942411
```

```
#Root mean squared error (RMSE)
sqrt(mean((df$y - yhat_ols)^2))
```

```
## [1] 2.990386
```

# Now with xgboost

https://xgboost.readthedocs.io/en/release_3.0.0/R-package/xgboost_introduction.html

```
library(xgboost)

xgb = xgboost(data = as.matrix(df[,-1]),
              label = as.matrix(df$y),
              nrounds = 25,verbose = 0)
```

# What's the fit?

```
yhat_xgb = predict(xgb,newdata=as.matrix(df[,-1]))

#MSE
mean((df$y - yhat_xgb)^2)
```

```
## [1] 8.585723
```

```
#RMSE
sqrt(mean((df$y - yhat_xgb)^2))
```

```
## [1] 2.93014
```

So, for a standard linear model you're probably not going to gain much here.

# What about a highly non-linear model?

```
n  = 50000
x1 = x2 = runif(n)
sigma = 3
y = x1 + x1^2 + log(x1) +
  exp(x2) + sqrt(x2) +
  rnorm(n,sd = sigma)
df_nl = data.frame(y,x1,x2)
```

# Linear regression fit

```
ols      = lm(y~x1+x2,df_nl)
yhat_ols = predict(ols)

#MSE
mean((df$y - yhat_ols)^2)
```

```
## [1] 131.2432
```

# Xgboost fit

```
xgb = xgboost(data = as.matrix(df_nl[,-1]),
              label = as.matrix(df_nl$y),
              nrounds = 25,verbose = 0)

yhat_xgb_nl = predict(xgb,newdata=as.matrix(df_nl[,-1]))

#MSE
mean((df_nl$y - yhat_xgb_nl)^2)
```

```
## [1] 8.67822
```

So XGBoost does about the same when the true model is linear, but better if the true model is non-linear.

# Next steps

At this point you have the following:

1) A rough idea of what it does, and where it fits with other methods
2) Working code to use it

- If you want to use it for anything, read more tutorials on how people go about tuning them
- Practice! Some of these methods are just meant to improve predictions, so use them for that and move on - there are so many methods no need to get in the weeds on all of them.
- By simulating a few scenarios, you can see *when* XGBoost might be more valuable (e.g., highly non-linear relationships between $X$ and $y$? lots of correlated variables? something else?)

# And don't overthink it!

1) What's the method and why?
2) How will you evaluate performance?

- Train/test splitting?
- MSE, RMSE, something else?

3) Do we really need something more complicated?

- Compare with plain vanilla linear/logistic regression if possible

# LASSO and Ridge Regression

# Shrinkage Methods

Many of you are probably familiar with *subset selection* methods

- *Stepwise selection* is the most common: start adding (or removing) variables one at a time, keeping variables that are "best"
- See section 6.1 of Introduction to Statistical Learning for a review
- I don't see these used too often in the real world, so I'll skip it

An alternative approach is to **regularize** or **shrink** all of the coefficient estimates towards zero

- It might not be clear why this should improve the fit, but it turns out that shrinking the coefficients can significantly reduce the variance in the estimates

# Ridge Regression and Lasso Regularization

These are the two most popular techniques for shrinking the coefficients

Both involve a tuning parameter $\lambda$, which controls the penalization

1) Ridge regression *shrinks* the coefficients towards zero
2) LASSO regularization *sets* unhelpful coefficients to zero
3) Elastic net combines the two methods into a single model, where one of the parameters "balances" between the methods

# Why do we need these?

**Prediction accuracy**

- With lots of variables, there is a high risk of overfitting the model
- This leads to poor out-of-sample performance
- You also might be in a situation where the number of variables is greater than the number of observations, suggesting that some of the variables *must* be dropped for estimation

**Model interpretability**

- Including irrelevant variables in a regression model leads to unnecessary complexity
- Regression on its own is unlikely to yield coefficient estimates that are exactly zero, we introduce a penalty to help push unhelpful variables to zero

# How it works

Recall in regression the goal is to minimize the sum of squared residuals:

$$\hat{\beta} = \arg\min_{\beta} \sum_{i} (y_i - X_i\beta)^2$$

Read this as: $\hat{\beta}$ is the resulting value of minimizing the function on the right, which is a function of $\beta$ (meaning this is the value that changes)

Or more simply: find me the value that minimizes the sum of squared residuals

$X_i$ is a $1 \times k$ vector, $\beta$ is a $k \times 1$ vector of coefficients, which includes an intercept ($\beta_0$) and $p$ non-zero coefficients

- $\beta = [\beta_0, \beta_1, \ldots, \beta_p]$

# Now add the penalty $\lambda$:

In ridge regression, we use an *L2* penalty:

$$\hat{\beta} = \arg\min_{\beta} \sum_{i} (y_i - X_i\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

In lasso, we use an *L1* penalty:

$$\hat{\beta} = \arg\min_{\beta} \sum_{i} (y_i - X_i\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

# Build the intuition in R:

```r
#function to calculate RSS with penalty:
RSS_penalty = function(beta,y,X,
                       penalty = 0,lambda = 0){

  RSS = sum((y - X%*%beta)^2)

  #ridge
  if(penalty == 1) RSS=RSS+ lambda*sum(beta[-1]^2)

  #lasso
  if(penalty == 2) RSS=RSS+ lambda*sum(abs(beta[-1]))

  RSS
}
```

# Make up data

```
n    = 1000
beta = c(5,6,30)
X = cbind(1,matrix(runif(n*2),n))
set.seed(1); y = X%*%beta + rnorm(n)
RSS_penalty(beta,y,X) #if beta was known
```

```
## [1] 1070.115
```

```
#estimate OLS with optim
betahat = c(0,0,0) #initial guess of beta
ols = optim(betahat,RSS_penalty,y=y,X=X,method="BFGS")
round(ols$value)
```

```
## [1] 1067
```

```
round(ols$par,2)
```

```
## [1]  4.87  6.19 30.05
```

# Add penalities

```
#Ridge
rd = optim(betahat,RSS_penalty,y=y,X=X,method="BFGS",
      penalty=1,lambda=1000)
round(rd$par,2)
```

```
## [1] 21.36  0.49  2.35
```

```
round(rd$value)
```

```
## [1] 74764
```

```
#Lasso
ls = optim(betahat,RSS_penalty,y=y,X=X,method="BFGS",
            penalty=2,lambda=1000)
round(ls$par,2)
```

```
## [1] 10.79  0.06 24.23
```

```
round(ls$value)
```

# In reality, use package `glmnet`

`glmnet` uses **elastic net** regularization, which simply combines the ridge and lasso penalties:

$$\hat{\beta} = \arg\min_{\beta} \sum_i (y_i - X_i\beta)^2 + \lambda \left[ (1-\alpha) \sum_{j=1}^{p} \beta_j^2 + \alpha \sum_{j=1}^{p} |\beta_j| \right]$$

Notice that the parameter $\alpha$ simply weights the penalties: $\alpha = 1$ will lead to a lasso penalty, and $\alpha = 0$ means ridge penalty

More details here:
https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html

# In Class Exercise: Housing Prices with Elastic Net Regularizaion

# The Situation

You work for a company in Ames, Iowa that flips houses

For this to work, you have to be able to spot good bargains on the market

If you have a model that accurately predicts the selling price, you can hunt after the houses selling below what they could normally get

Also, you can predict the selling price after planned improvements to see where you can get more value

# The data

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

1,460 observations of complete data

`SalePrice`: dependent variable of interest

77 explanatory variables

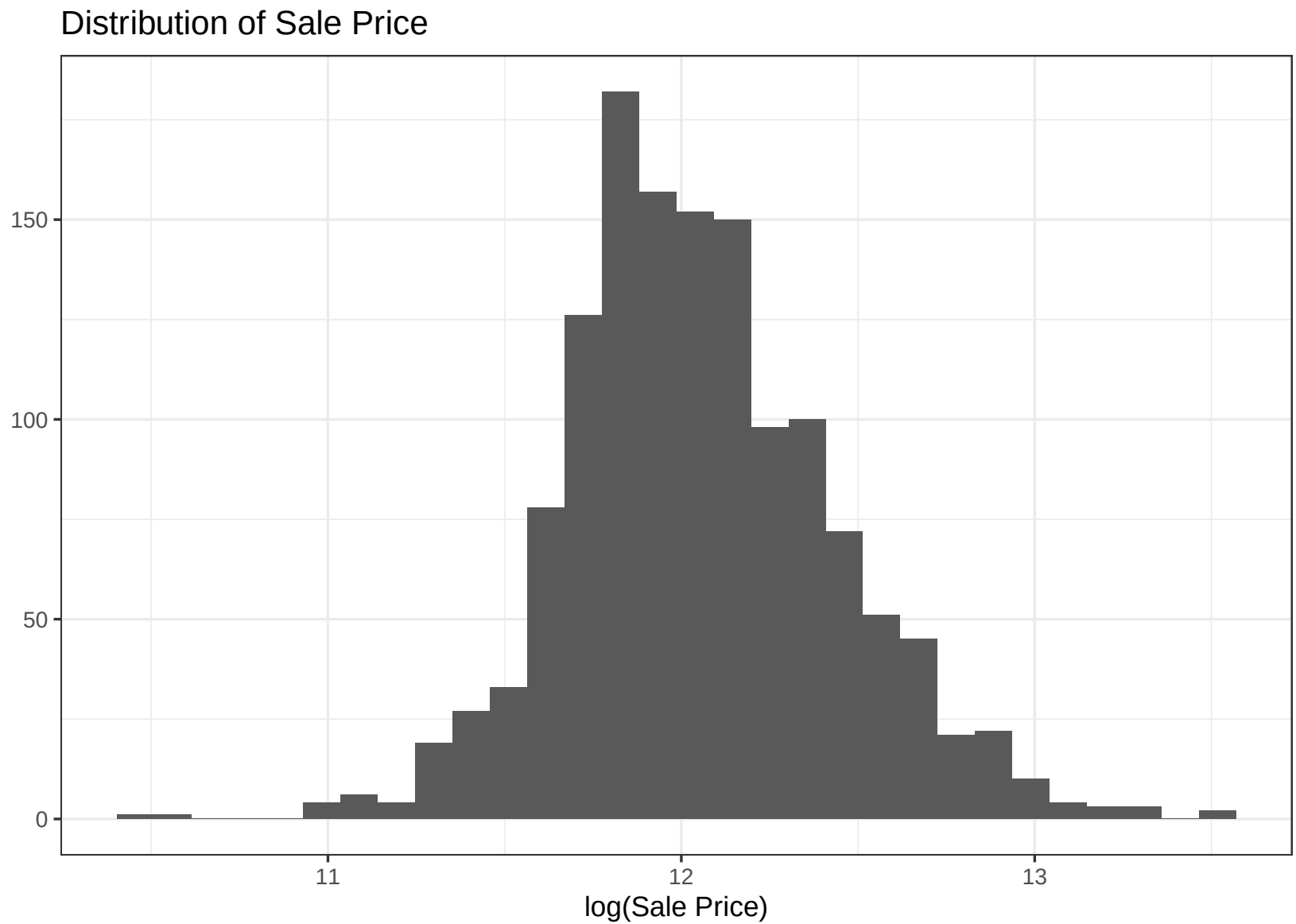
- Lot shape, home style, year built, fence, pool, etc.

**In this example, I only use the 35 numerical variables**

# Load data and prep

```
load('data/IowaHousing.Rdata')

y = as.matrix(house %>% select(logSalePrice))
x = as.matrix(house %>% select(-logSalePrice))

n = nrow(y)
set.seed(1);ind = sample(1:n,n*.75)

y.train = y[ind]
x.train = x[ind,]

y.test = y[-ind]
x.test = x[-ind,]
```

# Distribution of sale prices



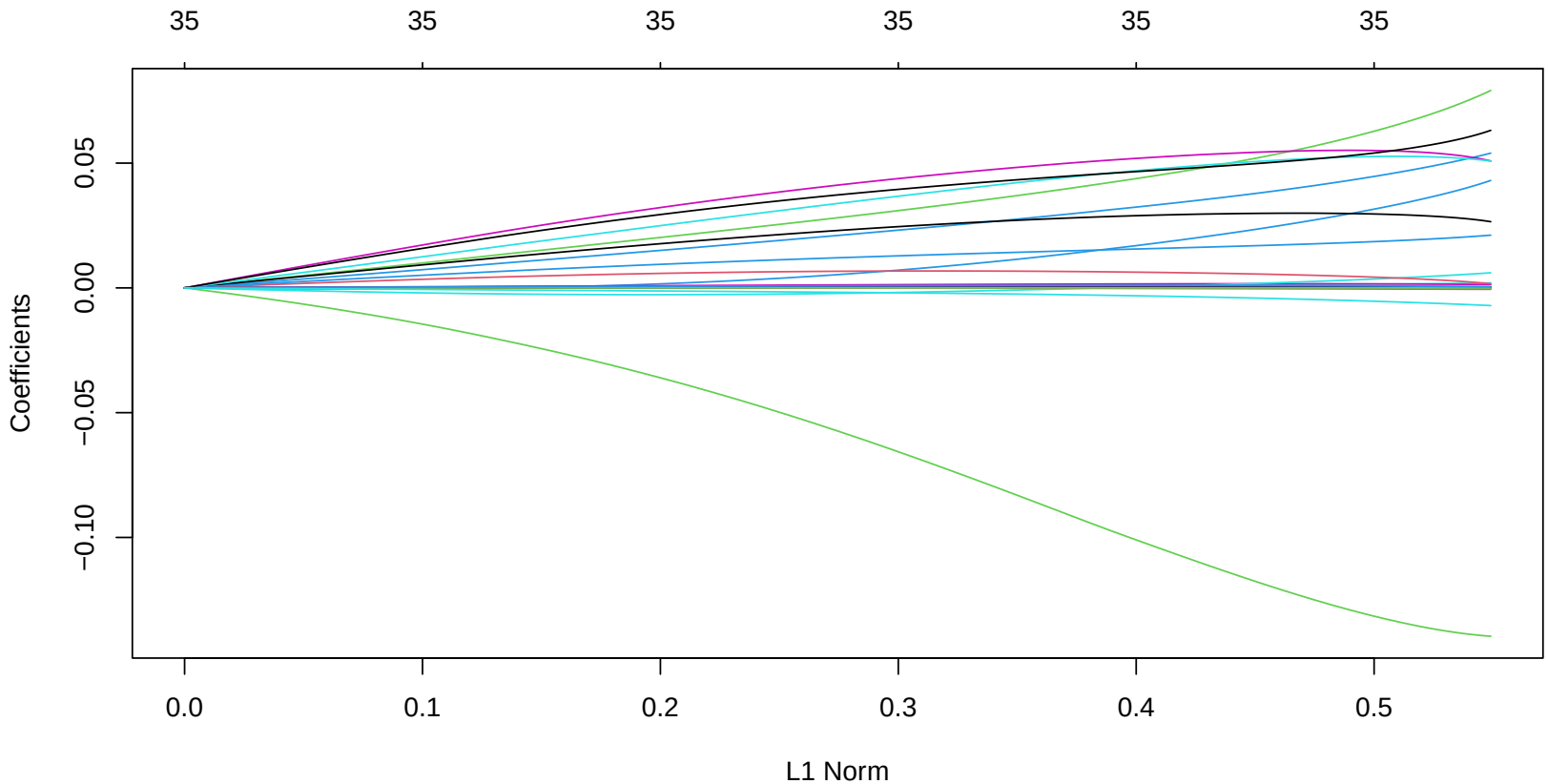
# First try ridge regression

```
#set alpha = 0 for ridge
ridge = glmnet(x.train,y.train,alpha=0)
```

# plot(ridge)

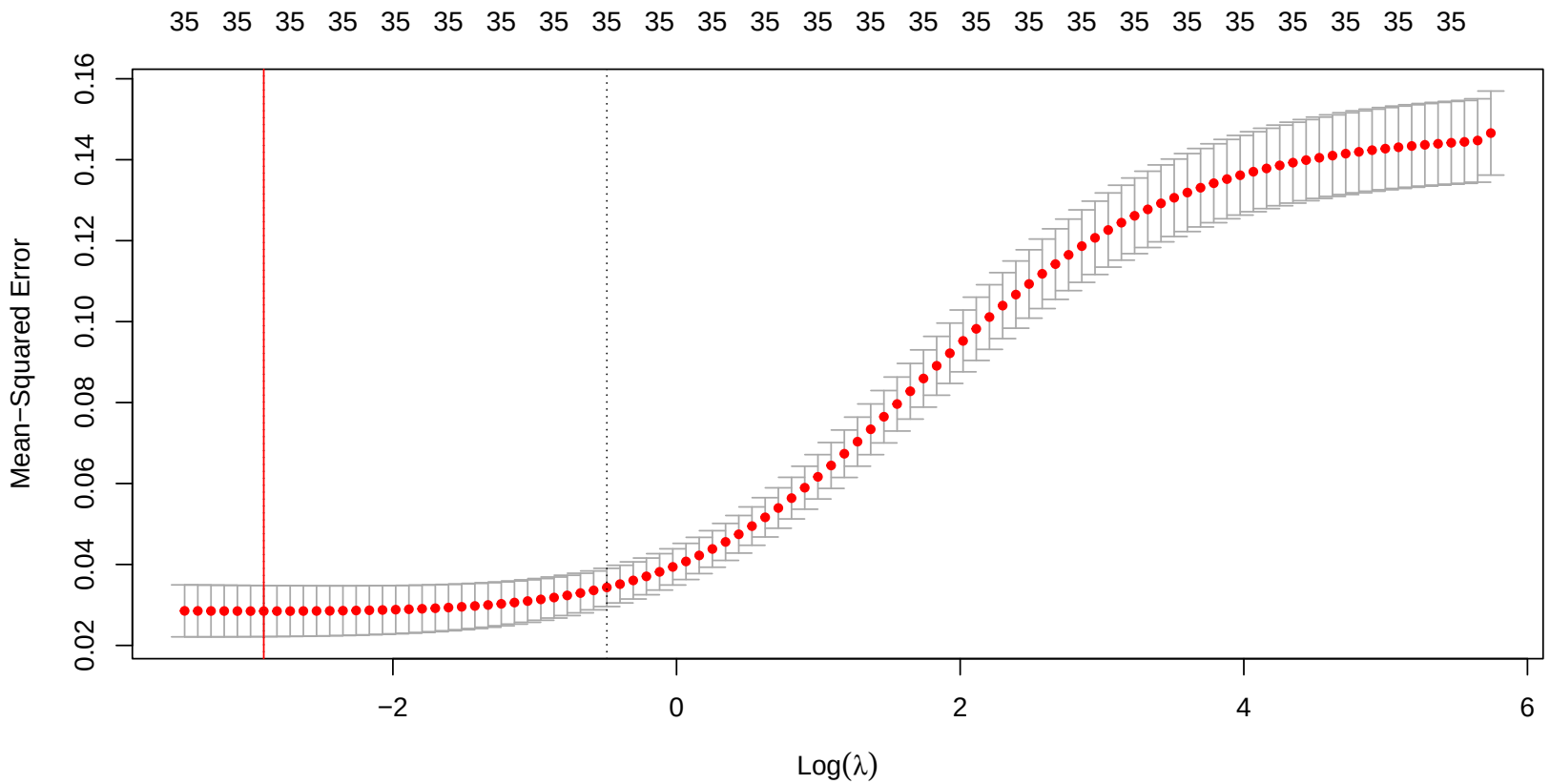
As the penalty increases (towards the left of the graph), we see more shrinkage towards zero (each curve is a variable)

The top axis indicates we always keep 35 variables



# Use cross-validation to select $\lambda$

```
ridge.cv = cv.glmnet(x.train,y.train,alpha=0)
s = ridge.cv$lambda.min #"best"
plot(ridge.cv);abline(v = log(s),col="red")
```



# Look at the coefficients

Don't worry, these won't mean much in ridge regression

```
coef(ridge,s=s)
```

```
## 36 x 1 sparse Matrix of class "dgCMatrix"
##                               s1
## (Intercept)     1.632784e+01
## LotFrontage    -1.095641e-05
## LotArea         1.392595e-06
## OverallQual     7.320797e-02
## OverallCond     3.889764e-02
## YearBuilt       1.789776e-03
## YearRemodAdd    1.441829e-03
## MasVnrArea      3.229153e-05
## BsmtFinSF1      4.166141e-05
## BsmtFinSF2      1.971091e-05
## BsmtUnfSF       5.894859e-06
## TotalBsmtSF     5.525664e-05
```

# How good is the prediction?

```
ridge_yhat = predict(ridge,newx=x.test,s=s)

#MSE
mean((exp(ridge_yhat)-exp(y.test))^2)
```

```
## [1] 653863707
```

```
#compare with OLS
ols = lm(y.train~x.train)
betahat = ols$coefficients
betahat[is.na(betahat)] = 0 #too many variables
ols_yhat = cbind(1,x.test) %*% betahat
mean((exp(ols_yhat)-exp(y.test))^2)
```

```
## [1] 622509254
```

# Is this bad? Maybe...

The advantage of ridge regression rests on the *bias-variance* tradeoff

As the penalty $\lambda$ increases, the predictions get worse (on average) but the variance also decreases

- Predictions are more precise, but maybe biased

Bias: $\mathbb{E}[\hat{y}] - y$

- How far off from the truth ($y$) is the estimate on average?
- High bias suggests underfitting

Variance: $\mathbb{E}[(\hat{y} - \bar{\hat{y}})^2]$

- What is the the average variance within a prediction?
- Higher variance suggests overfitting

# Implications

In situations with large $p$, a small change in the data may lead to a large change in the coefficient estimates (this is high variance)

See Figure 6.5 in Introduction to Statistical Learning

Here is a nice article for more information:

https://daviddalpiaz.github.io/r4sl/simulating-the-biasvariance-tradeoff.html

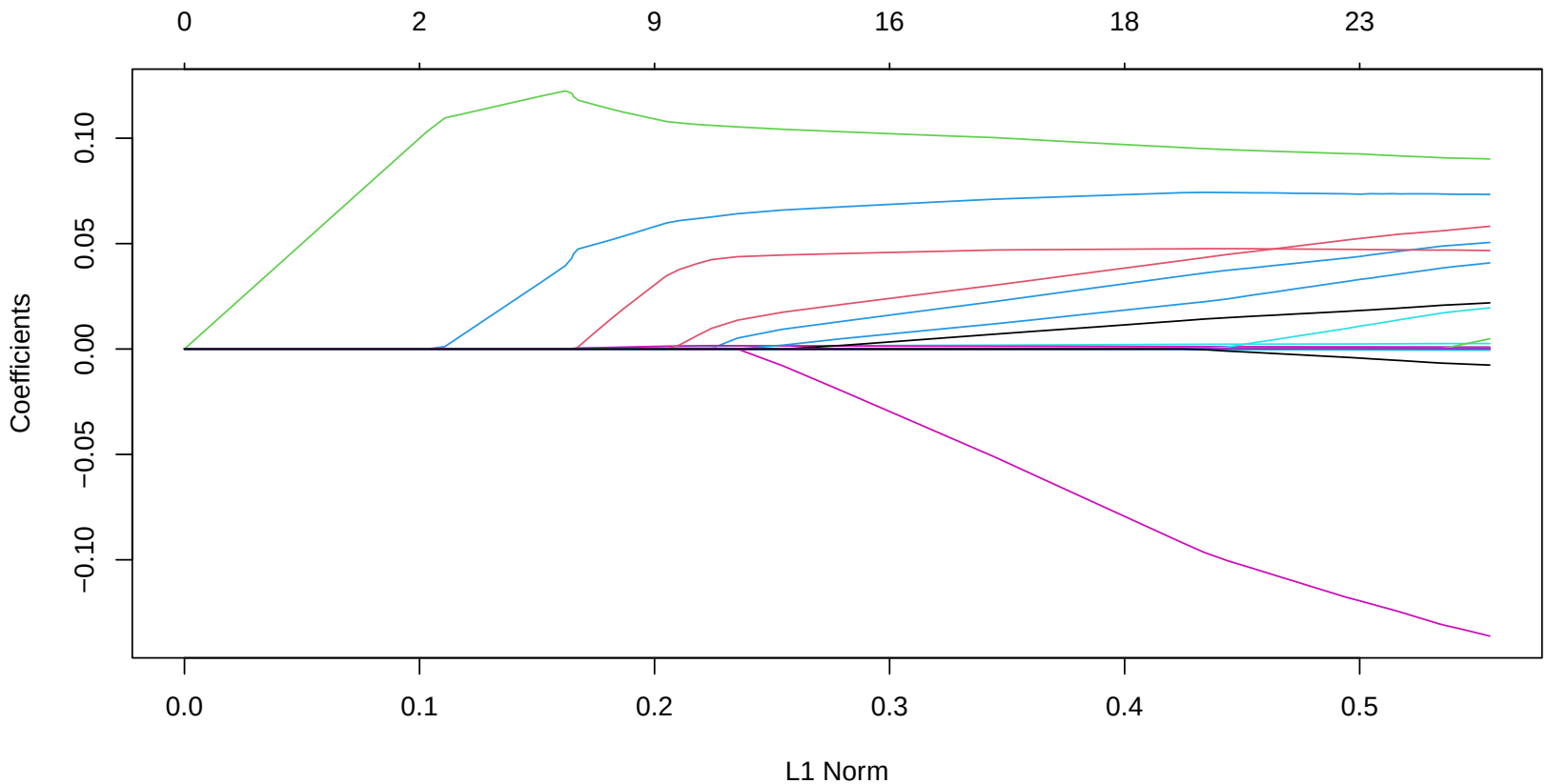
# Now try Lasso

```
#set alpha = 1 for lasso
lasso = glmnet(x.train,y.train,alpha=1)
```

# plot(lasso)

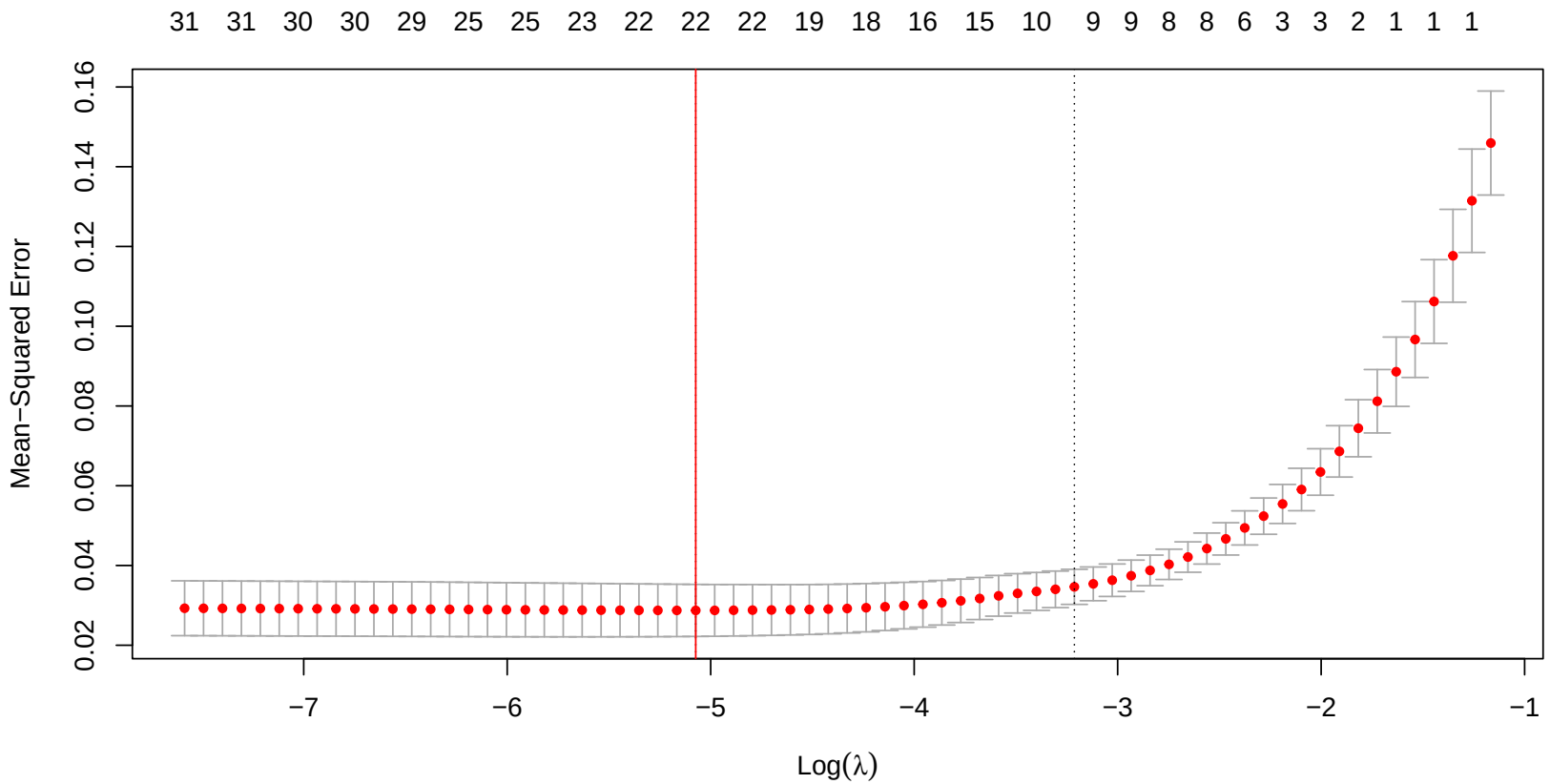
As the penalty increases (towards the left of the graph), we see variables tend to shrink (as was the case in ridge)

In addition, the top axis shows how the variables drop out as the penalty increases



# Use cross-validation to select $\lambda$

```
lasso.cv = cv.glmnet(x.train,y.train,alpha=1)
s = lasso.cv$lambda.min #"best"
plot(lasso.cv);abline(v = log(s),col="red")
```



# Look at the coefficients

These are directly interpretable, notice how some will drop out

```
coef(lasso,s=s)
```

```
## 36 x 1 sparse Matrix of class "dgCMatrix"
##                              s1
## (Intercept)    9.348430e+00
## LotFrontage    .
## LotArea        1.289143e-06
## OverallQual    9.346417e-02
## OverallCond    4.065782e-02
## YearBuilt      2.329634e-03
## YearRemodAdd   1.041040e-03
## MasVnrArea     .
## BsmtFinSF1     2.943547e-05
## BsmtFinSF2     .
## BsmtUnfSF      .
## TotalBsmtSF    4.307412e-05
## X1stFlrSF      3.685418e-05
```

# Ridge vs Lasso

Neither will universally dominate the other in predictive accuracy

The lasso has the advantage of simpler models with more interpretable coefficients, since they are not "shrunk" in any way

But. . . this sometimes means that some of equally important variables will be dropped in lasso, since it assumes that some of them must be zero

In general:

1) If you are looking for a simpler model go with lasso
2) If you want to improve predictive accuracy, try both

# So which to use?

Typically, you would try a bunch of methods

Ideas:

- Why not cross-validate $\alpha$ from 0 to 1, and see what fits best out of sample?
- Also try extreme gradient boosting with XGBoost
- And of course, compare with a plan vanilla regression as a baseline

# Setting it up

I'll use RMSE to evaluate fit

We basically need a data frame to store all of the RMSEs from all of our tests

For elastic net I'll try about 10 $\alpha$ values equally spaced between 0 and 1 and use the cross-validated suggested $\lambda$

For extreme gradient boosting I'll try about 10 $\lambda$ values equally spaced between 0 and 1

```
load('data/IowaHousing.Rdata')
```