Boosting and Block-CV

Lecture 12

Connor Dowd

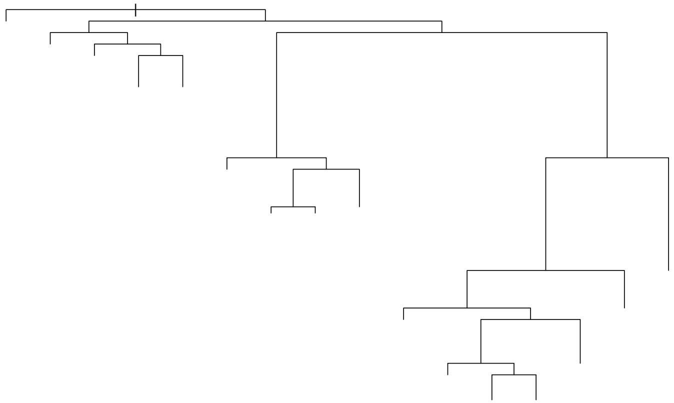
May 6th, 2021

Today's Class

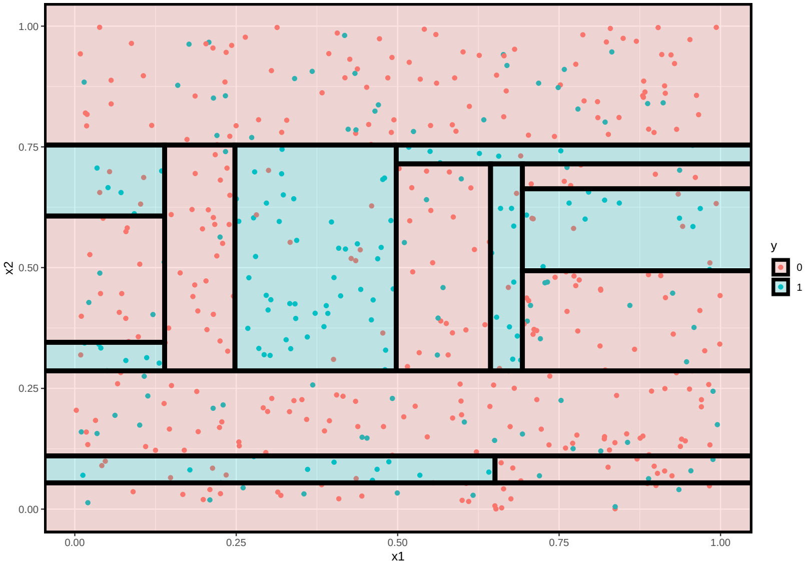
1. Review
   - Trees
   - Bagging
   - Forests
2. Other Basic Ensemble Methods
   - Weighted Averages
3. Boosting
4. Moving Block CV
   - For predicting the future.

Review

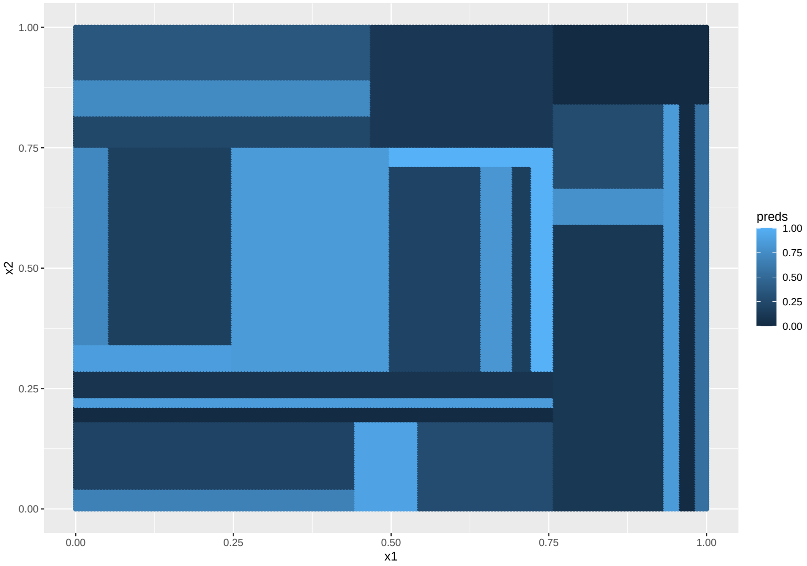
Trees



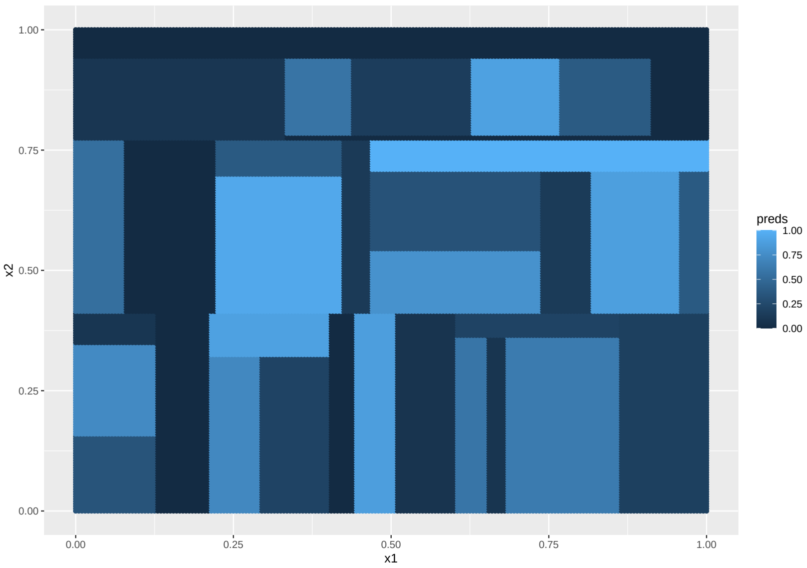
Trees



Example – B1

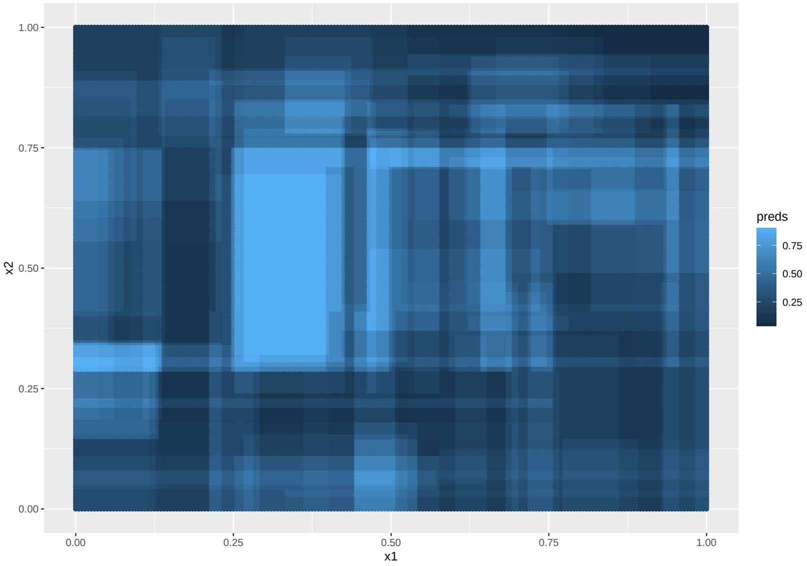

1.00
0.75
0.50
0.25
0.00
x2
0.00
0.25
0.50
0.75
1.00
x1
preds
1.00
0.75
0.50
0.25
0.00

Example – B7



x2
1.00
0.75
0.50
0.25
0.00
x1
0.00
0.25
0.50
0.75
1.00
preds
1.00
0.75
0.50
0.25
0.00

Averaged across 7 models



Random Forests

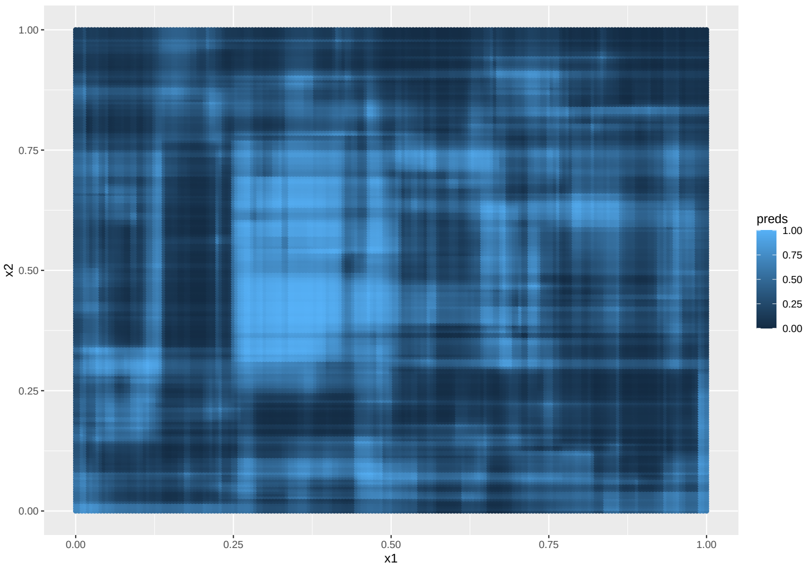
Forests take the notion of Bagging, and make some minor improvements – mostly in the name of speed.

By introducing variation into the variables under consideration, they create *even more instability* between trees.

But it turns out, because we are averaging across our trees, this leads to improvements in predictive power.

They search across a wider range of models.

Forests



preds
1.00
0.75
0.50
0.25
0.00
x2
1.00
0.75
0.50
0.25
0.00
x1
0.00
0.25
0.50
0.75
1.00

Other Ensembles

Weighted Averages

So far, to make a prediction $\hat{y}$ using covariates X and multiple models $M_1, ..., M_k$ we've been using a simple average.

$$\hat{y}_{avg} = \frac{1}{K} \sum_{k=1}^{K} M_k(X) = \sum_{k=1}^{K} \frac{1}{K} M_k(X)$$

But some of these basic models are better than others.

Weighted – Example

Suppose I have two models: - M_1 always predicts $\hat{y}_1 = M_1(X) = \bar{y}$ - M_2 uses p variables and linear regression to predict $\hat{y}_2 = M_2(X) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_p x_p$

These are not equivalently good. M_2 is *likely* better (not always – see overfitting again).

We could plug them both into our naive average, and get a prediction.

Weighted – Example

$$\hat{y}_{avg} = \frac{\hat{y}_1 + \hat{y}_2}{2} = \frac{M_1(x) + M_2(x)}{2} = \frac{\bar{y}}{2} + \frac{\hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_p x_p}{2}$$

Notice – this will take our regression predictions, and pull them towards the mean. Essentially “shrinking” our predictions towards the low-variance mean.

Its even shrinking all the coefficients. You could show that our prediction is identical to a linear prediction where we’ve halved each coefficient, except the intercept.

Weighted – Example

But this naive average doesn't make a ton of sense. We have reasons to believe these models perform at different levels.

Or alternately, perhaps "shrinking" the regression predictions towards the mean is a very good idea. But we may want to figure out how much shrinkage is best.

With two models, this is perhaps straightforward. We need to pick one parameter, w. Our weights will be $w_1 = w$ and $w_2 = 1 - w$.

Weighted – Example

This will yield predictions

$$\hat{y}_{weighted} = \sum_{k=1}^{2} w_k M_k(X) = w\bar{y} + (1-w)X'\hat{\beta}$$

When $w = 0$, we use the pure regression coefficients. When $w = 1$, we use purely the mean.

How to choose? We've seen this before. Cross validation can help us out.

- Just create a sequence of 100 possible weights, iterate through them with a hold out sample, find the weight w that has the best out of sample performance.

Weighted – Example

But what about when we have $k = 10,000$? Now we need a vector of k weights $\mathbf{w}$ which sum to 1. In theory cross-validation could still help us find the optimal weights by iterating through – but it will be very slow. We need to optimize across too many dimensions.

So then what?

- Precision Weights. (AKA Inverse Variance Weights)

Cross-validation still critical. But we will use it differently.

Inverse Variance Weights

The basic idea is that a single measure of the prediction errors of each model can help us generate the best weights.

The models with the smallest average errors should get the most weight, and the models with the largest average errors should get the least weight.

If you are minimizing MSE, this will look like weights which are proportional to the “precision" (aka the inverse of the variance).

Procedure

1. Estimate K models $M_1, ..., M_K$
2. Use cross-validation (k-fold) to estimate OOS errors for each model.
3. Calculate average OOS error MSE_k for each model.
4. Find weights $\mathbf{w} \propto MSE_k$ such that $\sum w_k = 1$
5. Make your predictions using those weights

This will be reasonably quick. For K models and m folds in the cross validation, you'll only need to fit Km models. The weights pop out analytically.

Further improvements.

Our new predictions are:

$$M(X) = \sum_{k=1}^{K} \frac{1}{w_k} M_k(X) = \omega \mathbf{M}$$

Where ω is a diagonal matrix with entries $\omega_{k,k} = w_k$ and $\mathbf{M}$ is a vector with entries $M_k = M_k(X)$.

There may be some correlation between predictions coming from different models. In principle, we can further optimize by looking at the the full covariance matrix of out-of-sample prediction errors $\hat{\Sigma}$.

$$M_\Sigma(X) = \hat{\Sigma}\mathbf{M}$$

Much like hedge funds like to upweight stocks that are uncorrelated with other stocks to 'hedge' against market risk, this upweights predictions whose errors are uncorrelated with other prediction's errors.

Why stop there?

The weighted average includes the constraint that our weights sum to 1, but otherwise, it looks a lot like:

$$M(X) = \mathbf{wM}$$

This is a linear model. We could just... estimate a linear model on top of our models.

Or we could... build a tree on top. Or a forest.

Comments

Purely for prediction purposes, including many different model types, and being flexible about when you rely on each, is incredibly valuable.

But the risk of overfitting *to your holdout sample* quickly grows high.

The recommended procedure in these settings, where you are using anything beyond the naive average or variance weighted avrage, is usually to have 3 partitions of your data.

1. In-In-sample, used to build each of the K sub-models.
2. Holdout A – used to build the ensemble model by estimating the OOS error of each of the K sub-models.
3. Holdout B – pure holdout – can be used to compare a few different ensemble ideas, or simply to benchmark your whole procedure.

Boosting

Example Data: Zillow Zestimates

Predict Zillow's zestimate's log-error using a bunch of covariates.

Why? Predicting this can improve our overall prediction.

Predicting Errors

If we look at our predictions and try to predict the errors that remain, we can adjust our naive predictions.

But this is fundamentally difficult. If it were straightforward to predict the errors we were making with the model we were using, we *already* would have done it.

Boosting – Ensembles 2

Boosting changes things up slightly. It says "we want to do a better job of predicting when we are wrong".

Thus far, we have been giving each tree slightly different data, and averaging across each tree.

But each tree still was optimizing the same thing – mean prediction error[1] :

$$\frac{1}{n}\sum_{i=1}^{n} l(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n} 1(\hat{y} \neq y)$$

[1] This could be MSE, etc depending on the model.

Boosting Insight 1

Some observations are harder to predict.

We want to build a model that predicts everything well.

We can give *more weight* to observations that are hard to predict.

$$\frac{1}{n}\sum_{i=1}^{n} 1(\hat{y} \neq y) = \sum_{i=1}^{n} \frac{1}{n} 1(\hat{y} \neq y)$$

The weights ideally sum to 1, but we need not have each weight be $\frac{1}{n}$. Feel familiar?

Boosting Algorithm

1. Start with a class of models F (e.g. tree, glm, etc). And weights $w_i = \frac{1}{n}$
2. Fit a model M_k in that class using those weights.
3. Find the prediction error for each observation from that model.
4. Increase the weights for observations where predictions were most wrong, decrease the weights for observations where predictions were most correct.
5. Repeat steps 2-4 until you've built K models.

Model M_K will be using weights based on how poorly each previous model did at predicting each observation.

Boosted Predictions

How to make a prediction?

- We don't want to use M_1 – it makes a lot of mistakes M_K doesn't.
- We don't want to use M_K – it may be overfit to oddball observations.

Average across each of the K models.

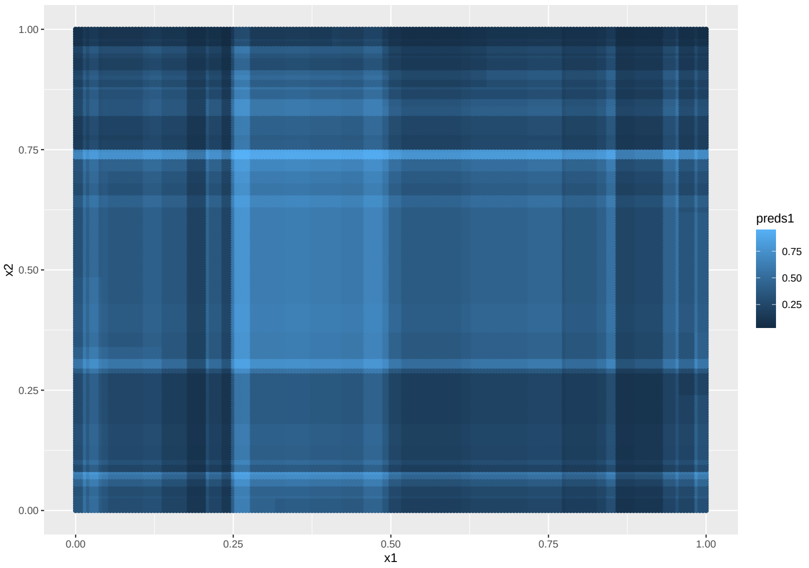
Boosting

```r
#Package
library(mboost)
#Tree version
mod1 = blackboost(y~.,data=df,family=Binomial())
#GLM
mod2 = glmboost(y~.,data=df,family=Binomial())
#GAM (relative of trees)
mod3 = gamboost(y~.,data=df,family=Binomial())

#Make predictions
grid$preds1 = predict(mod1,newdata=grid,type="response")
grid$preds2 = predict(mod2,newdata=grid,type="response")
grid$preds3 = predict(mod3,newdata=grid,type="response")
```

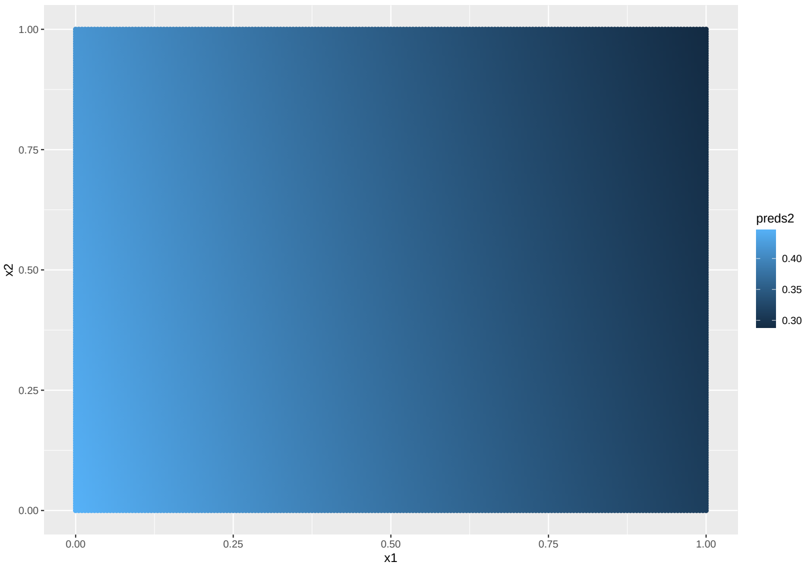
Boosting Trees

```
ggplot(grid,aes(x=x1,y=x2,col=preds1))+geom_point()
```



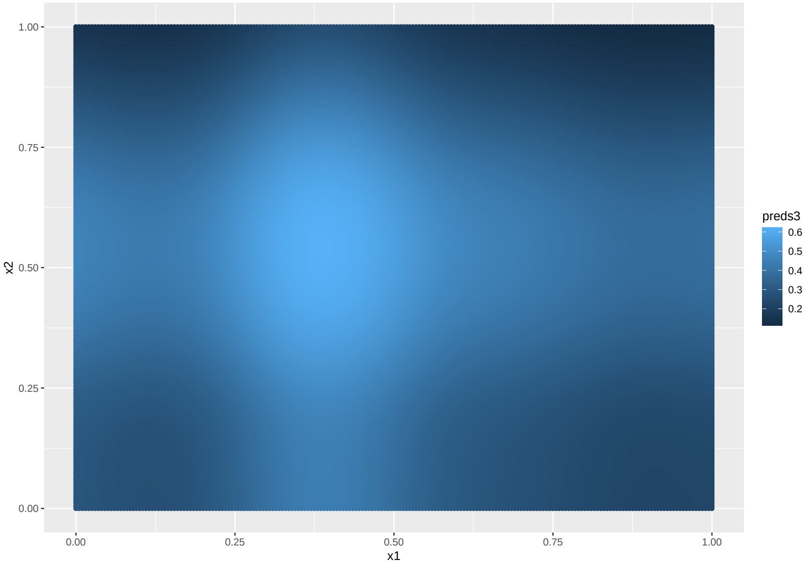
Boosting GLM

```r
ggplot(grid,aes(x=x1,y=x2,col=preds2))+geom_point()
```



Boosting GAM

```r
ggplot(grid,aes(x=x1,y=x2,col=preds3))+geom_point()
```



Comments

- Different approach to improving prediction errors
  - Takes advantage of *some* ability to determine what observations are difficult
    - e.g. “outlier detection”
- Slow – fitting 1000s of models
- Not ‘embarassingly parallel’
  - Each model depends on output of prior model – tasks aren’t easily split up
  - But ‘mboost’ can parallelize for you
- Can be used with any model *type*.

Cross-Validation 2: Time Series

K-Fold Cross Validation:

A refresher

1. Break data into k partitions
2. Estimate model on data in all but one partition
3. Find prediction errors using left-out partition
4. Repeat 2-3 until each partition has been held out once

This basic procedure is fine for a wide variety of settings. But we need to be careful about step (1).

Independence

When our observations are independent, we can randomly choose where to split our data up.

```
k = 5
partitions = sample(rep(1:k,length.out=n))
```

But when there is some dependence structure, as in time series we need to be more careful to retain that structure.

We don't want to pair March 15th, 2020 with May 2, 2021 in a random sample.

Goal Oriented CV

Suppose we have a goal – that goal is to predict something a week in the future.

We have data from ∼400 days. That means there were ∼390 times we could have predicted a week into the future.

- We can replicate *what we are doing* with the data *we have*.

Rolling block CV

Suppose you have a model for predicting a week into the future, which uses data from the past month.

We can estimate that model based on days 1-30, and see how wrong its forecast for day 37 is.

And then again for days 2-31 and day 38.

And so forth.

This will let us use real OOS predictions for doing model selection and development.

Rolling block CV

If you wanted to predict a certain day of the week... that might cut the number of these samples you have down quite a bit – from $\sim$300 to $\sim$50.

More broadly, looking at the (not well commented) code I posted for my initial prediction exercise may help.

Rolling Block

```r
n_pred = function(day1,lag=30,lead=30,square=F) {
  min = day1-lag
  pred_day = day1+lead #less than 5 day window, more than 3
  data = data %>% filter(days <= day1 & days >= min)
  mod = lm(one_dosers~days,data=data)
  if (square) mod = lm(one_dosers~days+days.sq,data=data)
  newdat = data.frame(days=pred_day,days.sq=pred_day^2)
  pred = predict(mod,newdata=newdat)
  pred
}
```

Rolling Block

```
pred_true_err = function(day1,lag=30,lead=30,square=F) {
  pred = n_pred(day1,lag,lead,square)
  true = data$one_dosers[data$days == day1+lead]
  err = pred-true
  today = data$one_dosers[data$days == day1]
  perc.err = err/(pred-today)
  c(day1+lead,pred,true,err,perc.err)
}
```

Wrap up

Things to do

Predictions.

Homework 5 will be posted tonight. Due next wednesday.

See you Tuesday.

Bye!