Lecture 6: Boosting and Bagging

Alan Lee

Department of Statistics
STATS 784 Lecture 6

August 11, 2017

Outline

Today's agenda

In this lecture we discuss how prediction methods may be improved by combining the results of several predictions, and look at ways of measuring variable importance. Today we will cover

- Boosting
- Bagging
- Random Forests
- Variable Importance

We will use the Boston housing data as a running example. Note: to HTF refer to Hastie, Tibshirani and Friedman, *The Elements of Statistical Learning, 2nd Ed*

Boosting

The basic idea here is to keep fitting models to modified versions of the data and adding the results together. The first use of this idea was in a classifier called ADABOOST which we will discuss briefly in Lecture 8. In the case of continuous targets, the method becomes

1. Fit a model (say linear regression or a tree) to the data. This becomes the "current model"
2. Then repeat
    2.1 Extract the residuals
    2.2 Fit a model to the residuals
    2.3 Add (some multiple) of the resulting model to the current model.
3. Stop when PE stops improving

Ref for Boosting: ISLR p321, HTF Chapter 10, APM Section 8.6

Math details

See “Forward stagewise modelling”: (HTF p 341) Many regression models fit an additive combination of “basis functions” i.e. express f as

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m)$$

where the β_m are regression coefficients and $b(x, \gamma)$ is a “basis function” which depends on a parameter γ. Examples:

- Linear: $b(x, \gamma) = x_j$
- Trees: $b(x, \gamma) = I(x \in R_m)$
- Neural nets: $b(x, \gamma) = \sigma(\gamma_0 + \gamma^T x)$

FSM Algorithm

1. Set $f_0(x) = 0$.
2. For $m = 1, 2 \ldots, M$
   2.1 Let r_i be the residuals from the current fit. Compute
   $$(\hat{\beta}\hat{\gamma}) = \text{argmin}_{\beta,\gamma} \sum_{i=1}^{n} (r_i - \beta b(x, \gamma))$$
   That is, select the basis function that best fits the residuals.
   2.2 Set $f_m(x) = f_{m-1}(x) + \nu \hat{\beta} b(x, \hat{\gamma})$.

Points to note

- In the case of linear regression, the basis functions are just the individual targets, so the algorithm is very similar to forward selection.
- The constant ν is called the regularization constant and is usually set at around 0.01-0.05. The idea is that many small adjustments are better than a few big ones, so we choose M to be large and ν to be small.
- When boosting trees, we fit a whole tree to the residuals. These are best taken to be small trees (say 4-8 terminal nodes)
- Implemented in R by the `gbm` and `mboost` packages - see documentation in the R help files.
- We need to choose M (number of boosting steps) (and in the case of trees) J (number of terminal nodes).
- The basic model being fitted is called the “base learner”.

Example: Boston housing data

Let's first try boosting using linear regression as the base learner. We will use the `glmboost` function in the `mboost` package. We will take $\nu = 0.05$ and try $M = 1000$.

```
library(MASS)
data(Boston)
# boosting using linear regression
library(mboost)
myfit.r = glmboost(log(medv)~., data = Boston,
   control = boost_control(mstop = 1000, nu = 0.05))
```

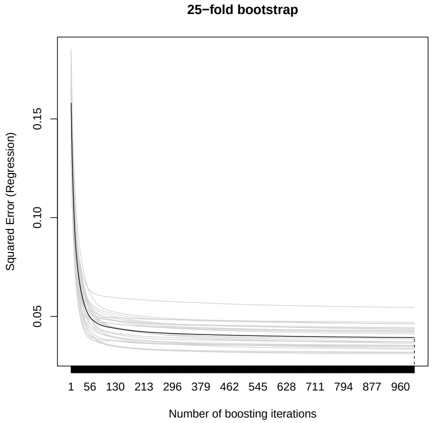
Example: Boston housing data

How do we know if we have got M right. We plot the estimated PE against different M values:

```
> Boston.risk.r = cvrisk(myfit.r)
> plot(Boston.risk.r)
> mean(Boston.risk.r[,mstop(Boston.risk.r)])
[1] 0.03928071
```

(The corrsponding figure for a linear model is approximately the same.) The last line calculates the best PE from the plot, shown on the next slide.

Example: PE plot

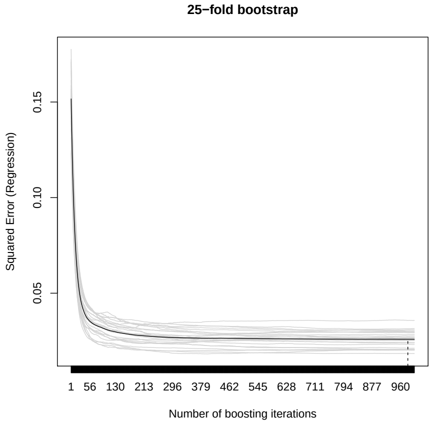


Example: Boston housing data

Now let's try boosting using trees as the base learner. We will use the blackboost function in the mboost package. We will take $\nu = 0.05$ and try $M = 1000$.

```
> myfit.r = blackboost(log(medv)~., data = Boston,
+     control = boost_control(mstop = 1000, nu = 0.05))
> Boston.risk.r = cvrisk(myfit.r)
> mean(Boston.risk.r[,mstop(Boston.risk.r)])
[1] 0.02591913
> mstop(Boston.risk.r)
 [1] 981
> plot(Boston.risk.r)
```

Example: PE plot for trees



Bagging

Bagging stands for *B*ootstrap *Agg*regation. The idea is: to make a prediction at x, we

- Draw B bootstrap samples
- Fit a model f to each sample
- Average the predictions from each model

Often done with trees, with a small tweek in the above. This results in Random Forests. Random Forests were invented by Leo Breiman, a Berkeley professor, and further developed by Adele Cutler (an Auckland graduate) Ref for Bagging: HTF Chapter 8 (section 8.7). Ref for Random Forests ISLR p 316, HTF Chapter 15, APM section 8.5. See also `http://www.stat.berkeley.edu/ breiman/RandomForests/`

Random Forests

We apply the bagging algorithm above to trees, modifying it as follows. To predict the response at x:

- Draw B bootstrap samples.
- For each sample, fit a tree of some specified depth. At each split, select the splits using only a randomly chosen subset of m variables, rather than choosing from all of them. The software default for m is one third of the number of features.
- Unlike boosting, we make the trees quite large - the recommendation in the software is to stop growing the trees when the terminal nodes contain less than 5 cases.
- Average the predictions $f(x)$ to obtain the final prediction. The the way the splits are chosen means the individual predictions are not highly correlated so we get a greater benefit in reduced variance.

Random Forests (cont)

- We don't need to cross-validate as the estimate of PE is made using only "out-of-bag" samples (the cases not included in th bootstrap sample.)
- As for boosting, we choose B large enough for the PE to settle down. Overfitting is not an issue - unlike boosting increasing B will not overfit the model. However, we don't want B to be too big as it lengthens the computational time.

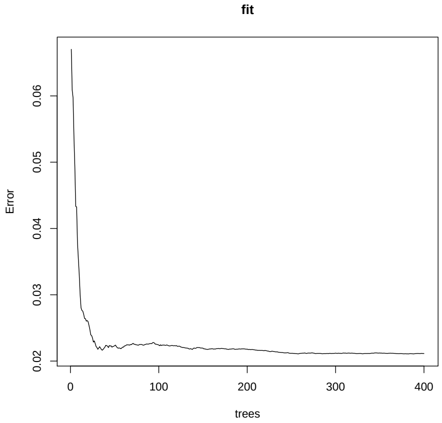
Both m and the mininum nodesize are tuning parameters and the models should be optimised over these.

Example

We use the R functionrandomForest in the package of the same name.

```
> library(randomForest)
> fit = randomForest(log(medv) ~ ., data=Boston, ntree=400,
+   mtry=5, nodesize=5, importance=TRUE)
> plot(fit) # this gives a plot of PE against B
> fit$mse[400]
[1] 0.02113493
```

Example: PE plot for RF



Variable Importance

Now we address the questions

1. Which features are important in making our prediction?
2. What is the relationship between a feature and the target?

Variable Importance: basics

Linear functions:	Size of standardized regression coefficients, p -values, correlation with target
gams:	Approximate significance of terms, correlation with target
ppr:	Large standardised coefficients in ridge terms
MARS:	Variables in basis functions having large coefficients (after standardization)
NN:	Large weights (standardised variables)
Trees, RF:	See next slide

Variable Importance: trees

For a single tree, we can add up the decreases in the RSS associated with all splits involving a particular variable. For random forests, we simply aggregate these over all trees in the forest. There are functions `importance` and `varImpPlot` in the `randomForest` package to do this.

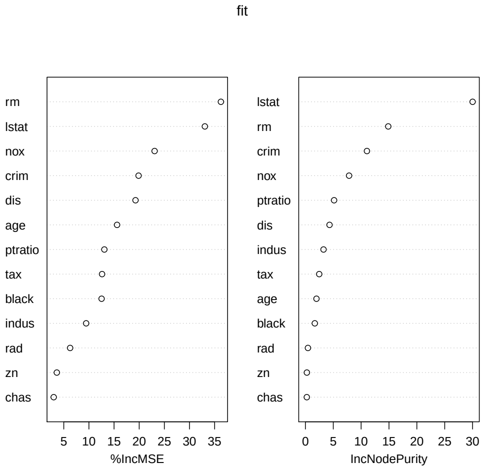
An alternative method is to fit the model, record the OOB prediction error for each tree, then average this over all the trees. Then, randomly permute the values of a variable, and repeat the PE calculation. The difference is a measure of the variable's importance, as permuting values will destroy any predictive ability the variable might have.

Variable Importance: trees

```
> importance(fit)
          %IncMSE IncNodePurity
crim    19.875541    11.0419894
zn       3.617168     0.2625693
indus    9.451135     3.2505651
chas     2.998207     0.2490390
nox     23.061170     7.8358205
rm      36.246579    14.8684280
age     15.596956     1.9836929
dis     19.266619     4.3283789
rad      6.265112     0.4547750
tax     12.616478     2.4778239
ptratio 13.073179     5.1453404
black   12.520710     1.6876769
lstat   33.057411    30.0070478
> varImpPlot(fit)
```

Note: IncNodePurity refers to the first method on the previos slide, %IncMSE the second.

Example: Importance plot



Partial dependence plots

How can we explore the relationship between individual variables (or subsets of variables) and the predictor? One way is to consider the average value of f for a fixed value x_j of a particular variable (say X_j), averaged over the other variables. For example, suppose we want to understand the relationship between X_1 and f. For a fixed value x_1 of X_1, we can calculate

$$\frac{1}{n}\sum_{i=1}^{n}\hat{f}(x_1, x_{i2}, \ldots, x_{ik}).$$

We can repeat this for different values of x_1 and plot the result. This is called a Partial Dependence Plot.

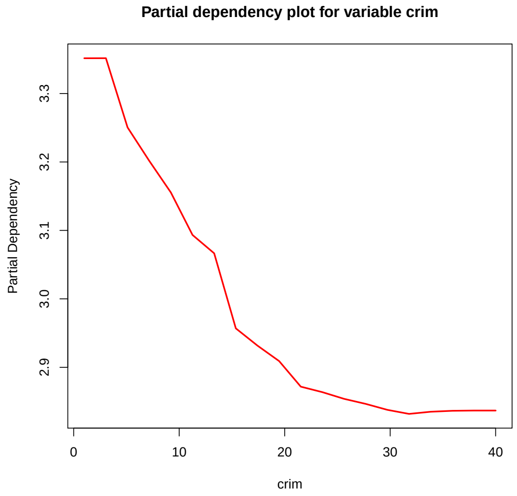
Code

```
###########################################
# partial dependence plots for variable crim
x = seq(1,40, length=20)
mypdp = numeric(20)
for(i in 1:20){
  newdata = Boston
  newdata[,13] = x[i]
  mypdp[i]=mean(predict(fit, newdata=newdata))
print(i)
}

plot(x,mypdp, type="l", xlab = "crim",
ylab = "Partial Dependency", col="red", lwd=2,
main= "Partial dependency plot for variable crim")
```

Note that this code will work for any prediction method: here we do it for random forests.

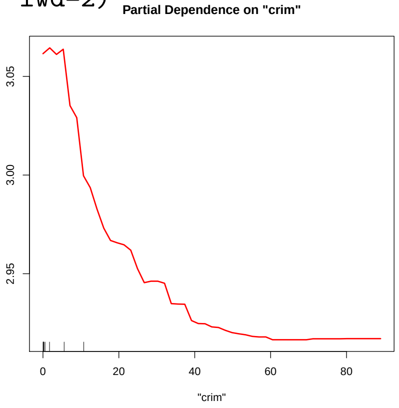
The plot


Partial dependency plot for variable crim
Partial Dependency
3.3
3.2
3.1
3.0
2.9
0
10
20
30
40
crim

RF plot

For random forests there is a built-in function

```
> partialPlot(fit, pred.data=Boston, x.var = "crim",
    col="red", lwd=2)
```


Partial Dependence on "crim"
"crim"