Boosting 3: Implementations

-Introduction to Data Science-

Lecturer: Darren Homrighausen, PhD

Outline

Now we will discuss two current, popular algorithms and their R implementations

- GBM
- XGBoost

GBM

Gradient Boosting Machines (GBM)

Recall: Boosting uses forward stepwise minimization of a loss function to build an additive model

GBM takes this idea and implements this for

- lots of loss functions
- adds subsampling
- includes methods for choosing B
- reports variable importance measures

GBM: LOSS FUNCTIONS

- gaussian: squared error
- laplace: absolute value

  (The Bayes' rule for this loss is the median($Y|X$). Hence, it is more robust than for squared error loss)
- bernoulli: the Bernoulli family with logistic link
- adaboost: exponential
- multinomial: more than two classes
- poisson: Count data

  (Useful if the supervisor is smallish number of counts (e.g. less than 20))
- coxph: For right censored, survival data

GBM: SUBSAMPLING

It has been noted that boosting performance can be improved if, at each step b, a new random subsample of observations is used

This is known as stochastic gradient boosting

This has two possible benefits

- Reduces computations/storage

  (But increases read/write time)
- Can improve performance

(This is the bag.fraction parameter)

GBM: subsampling

The improvement due to subsampling suggests the usual ‘variance reduction through lowering covariance” interpretation

The effect is complicated, though as subsampling

- increases the variance of each term in the sum

  (Due to fewer terms training each base learner)
- decreases the covariance between each term in the sum

  (Forcing the base learner to focus on different observations)

Another way of viewing this is that subsampling regularizes the boosting procedure

GBM: choosing B

There are three built in methods:

- Independent test set: using the nTrain parameter to say ‘use only this amount of data for training’

  (Be sure to uniformly permute your data set first.)
- Out-of-bag (OOB) estimation: If bag.fraction is > 0, then gbm use OOB at each iteration to find a good B

  (Note: OOB tends to select a too-small B)
- K-fold cross validation (CV): It will fit cv.folds$+1$ models

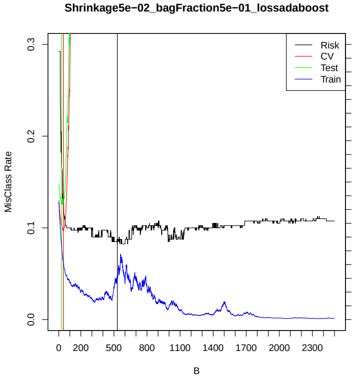
  (The ‘$+1$’ is the fit on all the data that is reported)

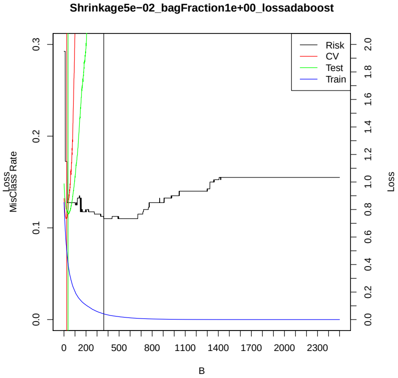
Simulation

GBM: Bag fraction

Let's look at a simulation for

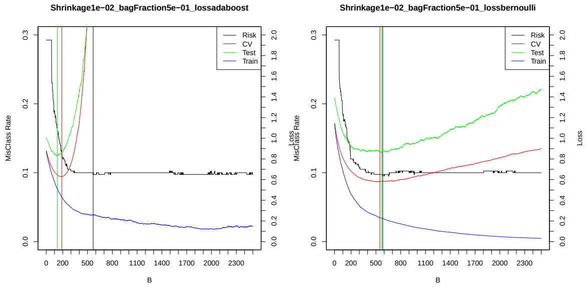
- adaboost
- bag fraction = 0.5 vs. 1.0
- Test, CV, and training estimates of the risk as well as the actual risk





Bag fraction of 0.5 improves the best risk

GBM: Loss function



Bernoulli loss: CV-minimum $B \approx$ Risk-minimum B

Comparing loss functions

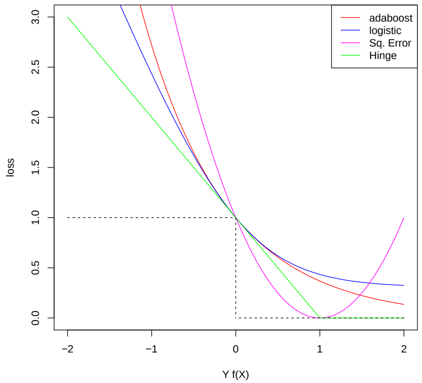


Figure: Here, I've rescaled the logistic loss so that it is easier to compare to Adaboost

Reminder: The boosting classifier is of the form $\mathrm{sgn}(f(X))$

GBM: variable importance measure

For tree-based methods, there are variable importance measures:

RELATIVE.INFLUENCE: For each feature X_j and tree T_b

$$\text{Influence}_j(T_b) = \sum_{\text{Split on } X_j} (\text{Reduction in loss})^2$$

This is aggregetated to form

$$\text{Influence}_j = \frac{1}{B}\sum_{b=1}^{B} \text{Influence}_j(T_b)$$

(There is also permutation.test.gbm, but it is currently labeled experimental)

GBM: variable importance measure

Example using the spam data:

```
gbm.sum = summary(boost.out)
head(gbm.sum)
                    var   rel.inf
punc_exclam punc_exclam 22.564132
punc_dollar punc_dollar 20.547832
remove           remove 11.900176
hp                   hp  7.977975
free               free  6.563215
capAvg           capAvg  5.019234
```

Partial dependence plot

GBM provides additional plots for the effect of each feature on the final prediction

The plots are analogous to interpreting a coefficient in multiple regression:

$$f(X) = \sum_{j=1}^{p} \beta_j x_j$$

The coefficient β_j is the difference in the mean of Y for a 1 unit change in x_j given all the other features are in the model and are held constant

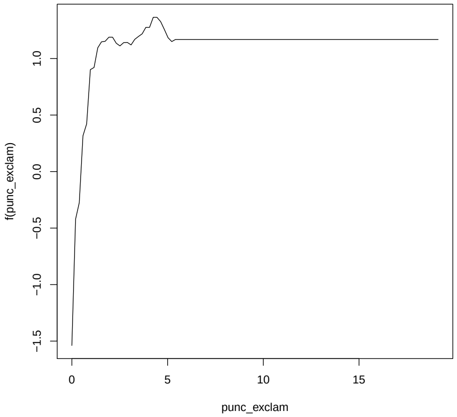
Boosting fits a nonparametric model, so the interpretation is more complicated

The idea is that we can 'integrate' out the effect of the other features in order to 'hold them constant'

Partial dependence plot

Let's look at the the plot for the most important feature

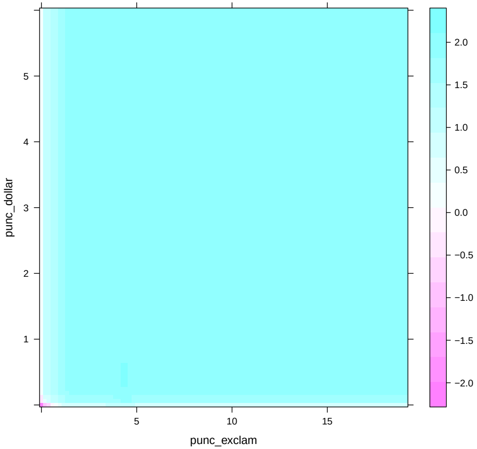
```
most.influential = which(names(X)%in%gbm.sum[1:1,1])
plot(boost.out,i.var=most.influential)
```



Partial dependence plot

Let's look at the the plot for the most important feature

```
most.influential = which(names(X)%in%gbm.sum[1:2,1])
plot(boost.out,i.var=most.influential)
```



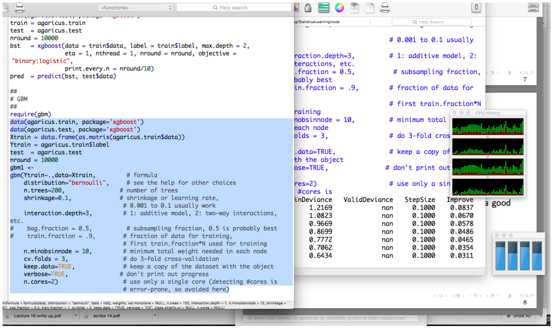
GBM: sample code

```
gbm(Ytrain~.,data=Xtrain,
    distribution="bernoulli",
    n.trees=500,
    shrinkage=0.01,
    interaction.depth=3,
    bag.fraction = 0.5,
    n.minobsinnode = 10,
    cv.folds = 3,
    keep.data=TRUE,
    verbose=TRUE,
    n.cores=2)
```

Keep adding trees with gbm.more

(If this is taking too long, increase the learning rate, shrinkage)

GBM: Figures

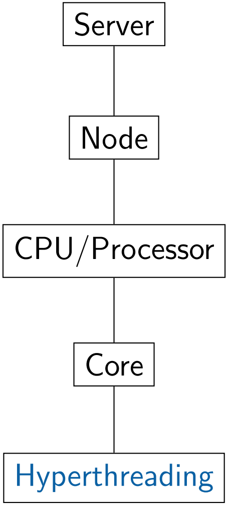


```
train = agaricus.train
test  = agaricus.test
nround = 10000
bst   = xgboost(data = train$data, label = train$label, max.depth = 2,
                eta = 1, nthread = 1, nround = nround, objective =
"binary:logistic",
                print.every.n = nround/10)
pred  = predict(bst, test$data)

##
# GBM
##
require(gbm)
data(agaricus.train, package='xgboost')
data(agaricus.test, package='xgboost')
Xtrain = data.frame(as.matrix(agaricus.train$data))
Ytrain = agaricus.train$label
test  = agaricus.test
nround = 10000
gbm1 <-
gbm(Ytrain~.,data=Xtrain,          # formula
    distribution="bernoulli",      # see the help for other choices
    n.trees=200,                  # number of trees
    shrinkage=0.1,                # shrinkage or learning rate,
                                   # 0.001 to 0.1 usually work
    interaction.depth=3,           # 1: additive model, 2: two-way interactions,
etc.
#    bag.fraction = 0.5,            # subsampling fraction, 0.5 is probably best
#    train.fraction = .9,          # fraction of data for training,
                                   # first train.fraction*N used for training
    n.minobsinnode = 10,           # minimum total weight needed in each node
    cv.folds = 3,                  # do 3-fold cross-validation
    keep.data=TRUE,                # keep a copy of the dataset with the object
    verbose=TRUE,                 # don't print out progress
    n.cores=2)                     # use only a single core (detecting #cores is
                                   # error-prone, so avoided here)
```

```
m(formula = formula(data), distribution = "bernoulli", data = list(), weights, var.monotone = NULL, n.trees = 100, interaction.depth = 1, n.minobsinnode = 10, shrinkage =
01, bag.fraction = 0.5, train.fraction = 1, cv.folds = 0, keep.data = TRUE, verbose = "CV", class.stratify.cv = NULL, n.cores = NULL)
```

```
                             # 0.001 to 0.1 usually

raction.depth=3,             # 1: additive model, 2:
teractions, etc.
.fraction = 0.5,              # subsampling fraction,
bably best
in.fraction = .9,            # fraction of data for

                             # first train.fraction*N
raining
nobsinnode = 10,             # minimum total
each node
olds = 3,                    # do 3-fold cros

.data=TRUE,                  # keep a copy of
th the object
ose=TRUE,                   # don't print out

res=2)                       # use only a sin
 #cores is
inDeviance   ValidDeviance   StepSize   Improve
     1.2169             nan     0.1000    0.0837
     1.0823             nan     0.1000    0.0670
     0.9669             nan     0.1000    0.0578
     0.8699             nan     0.1000    0.0486
     0.7772             nan     0.1000    0.0465
     0.7062             nan     0.1000    0.0354
     0.6434             nan     0.1000    0.0311
```

Distributed computing hierarchy



Example: A server might have

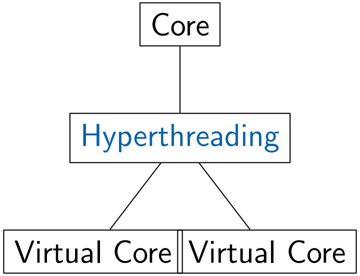
- 64 nodes
- 2 processors per node
- 16 cores per processor
- hyper threading

The goal is to somehow allocate a job so that these resources are used efficiently

Jobs are composed of threads, which are specific computations

Hyperthreading

Developed by Intel, Hypertheading allows for each core to pretend to be two cores



This works by trading off computation and read-time for each core

Boosting: Learning slow

It is best to set the learning rate at a small number.

This is usually calibrated by the computational demands of the problem.

A good strategy is to pick a number, say .001

Run with n.trees relatively small and see how long it takes

Keep adding trees with gbm.more. If this is taking too long, increase the learning rate

XGBoost

XGBOOST

This stands for:

EXTREME GRADIENT BOOSTING

It has some advances related to gbm

XGBOOST: ADVANCES

- SPARSE MATRICES: Can use sparse matrices as inputs

  (In fact, it has its own matrix-like data structure that is recommended)
- OPENMP: Incorporates OpenMP on Windows/Linux

  (OpenMP is a message passing parallelization paradigm for shared memory parallel programming)
- LOSS FUNCTIONS: You can specifiy your own loss/evaluation functions

  (You need to use xgb.train for this)