Boosting 3: Implementations

-Statistical Machine Learning-

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: AdaBoost effectively uses forward stagewise minimization of the exponential loss function

GBM takes this idea and

- generalizes to other loss functions
- adds subsampling
- includes methods for choosing B
- reports variable importance measures

GBM: loss functions

- gaussian: squared error
- laplace: absolute value
- bernoulli: logistic
- adaboost: exponential
- multinomial: more than one class (unordered)
- poisson: Count data
- coxph: For right censored, survival data

GBM: SUBSAMPLING

Early implementations of AdaBoost randomly sampled the weights (w)

This wasn't essential and has been altered to use deterministic weights

Friedman (2002) introduced stochastic gradient boosting that uses a new subsample at each boosting iteration to find and project the gradient

This has two possible benefits

- Reduces computations/storage

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

GBM: SUBSAMPLING

You can expect performance gains when both of the following occur:

- There is a small sample size
- The base learner is relatively complex

This 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
- decreases the covariance of each term in the sum

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 can use OOB at each iteration to find a good B

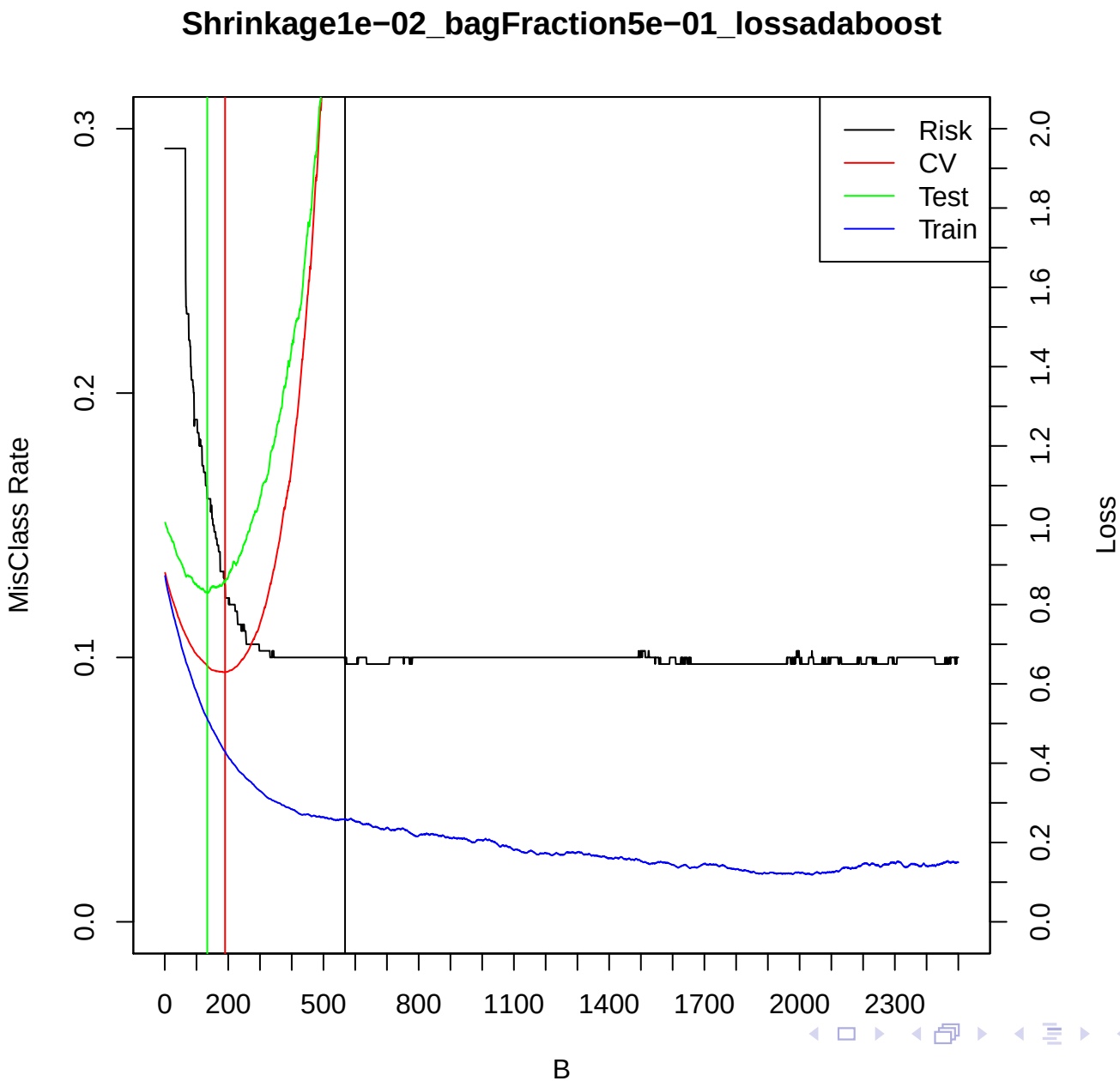
  (Note: OOB tends to select a too-small B)

- K-fold cross validation (CV): It will fit cv.folds+1 models

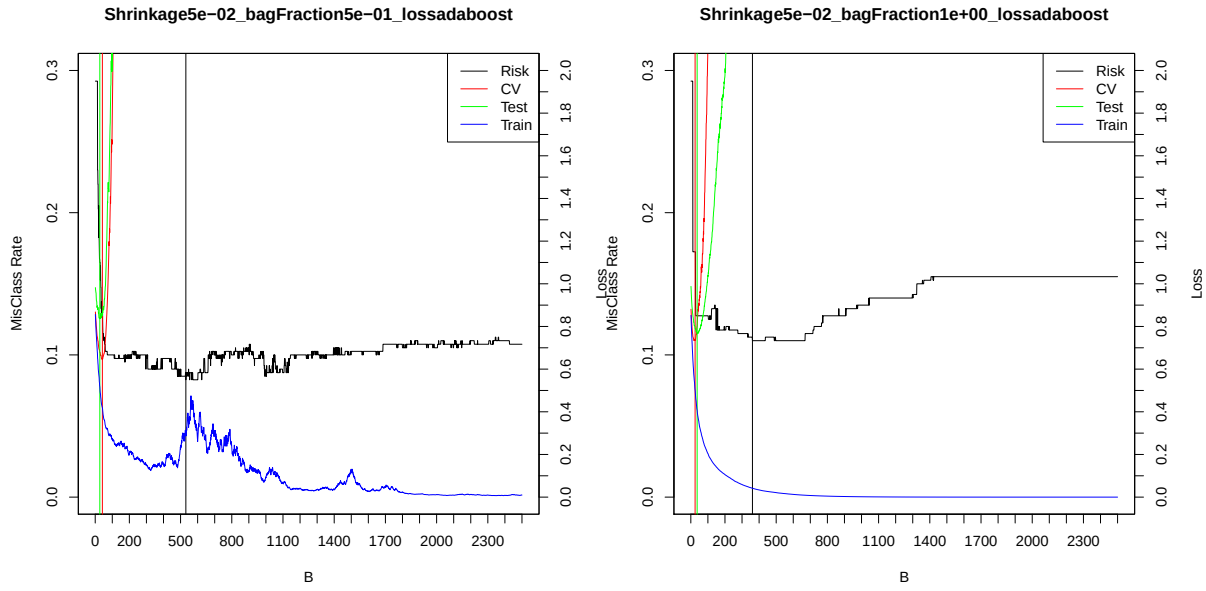
  (The '+1' is the fit on all the data that is reported)

Simulation: Continued

GBM: Some observations

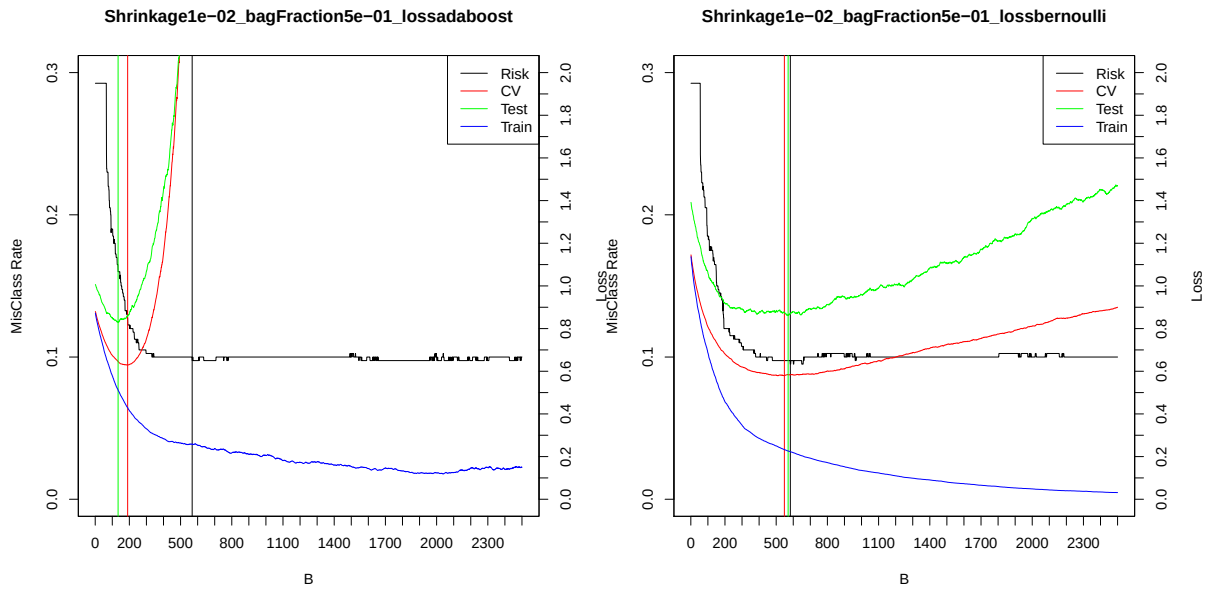


GBM: Bag fraction



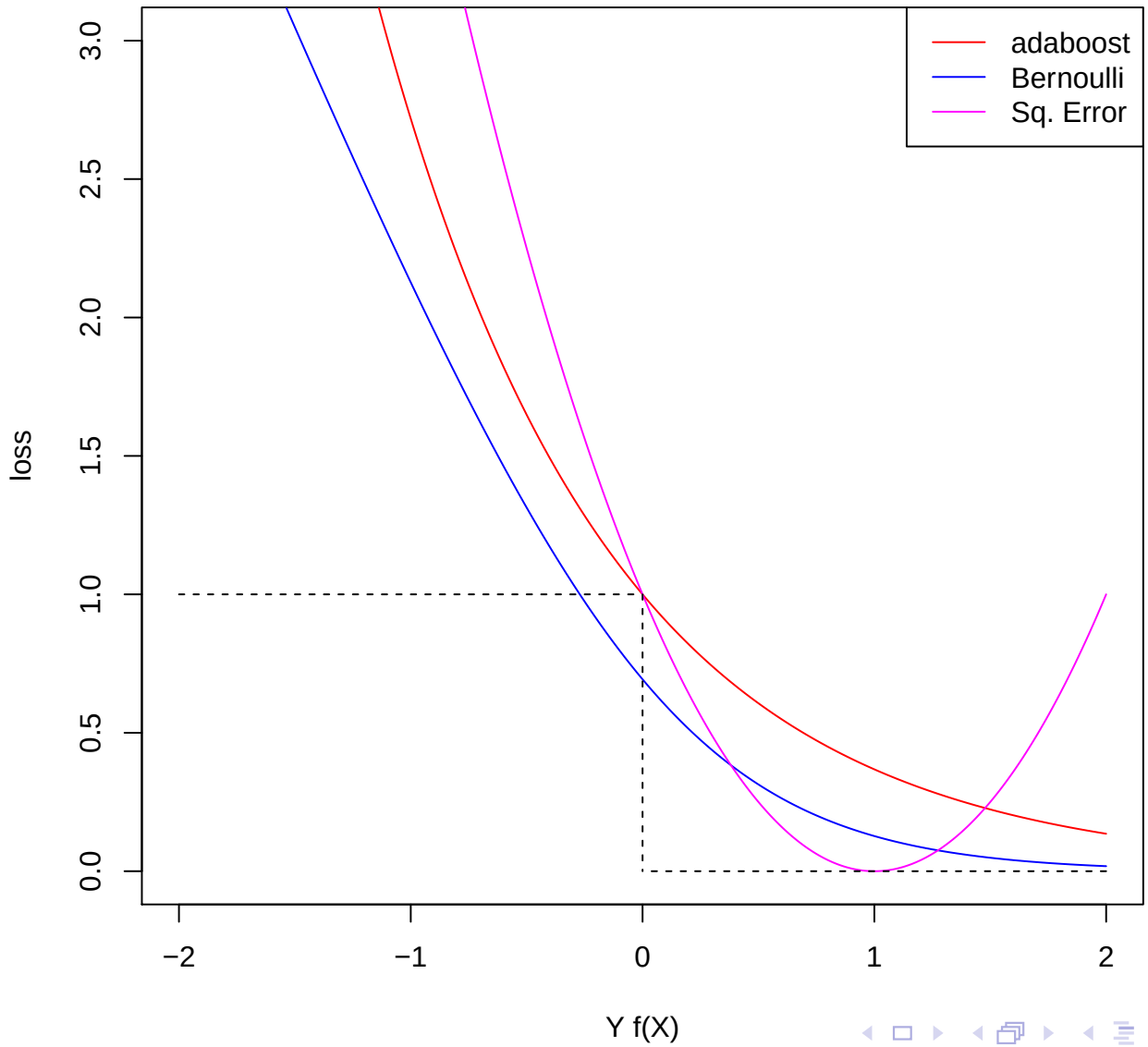
Bag fraction of 0.5 improves the best risk

GBM: Loss function



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

Comparing loss functions



GBM: VARIABLE IMPORTANCE MEASURE

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

RELATIVE.INFLUENCE: For each feature X_j and tree T_b

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

(See equations (43) - (45) in Friedman (2001) for details)

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)

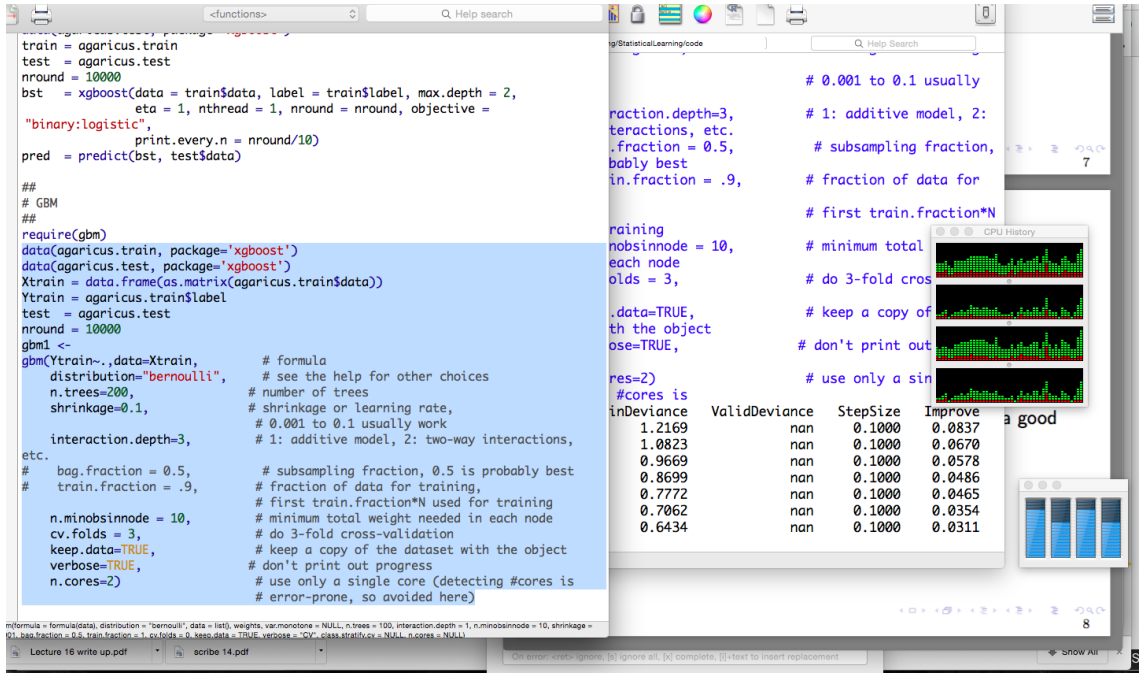
GBM: SAMPLE CODE

```
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)
```

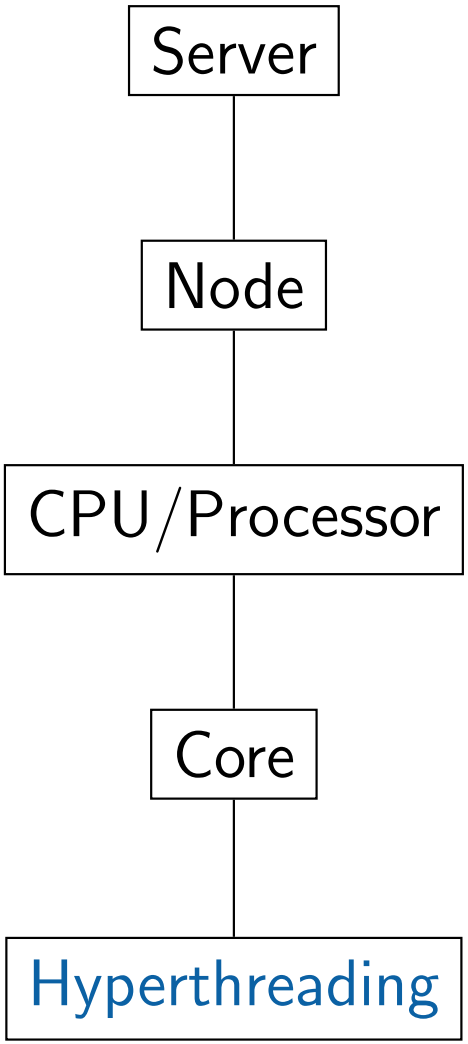
Keep adding trees with gbm.more

(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)
# 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
a good
CPU History
Lecture 16 write up.pdf
scribe 14.pdf

Distributed computing hierarchy



Example: A server might have

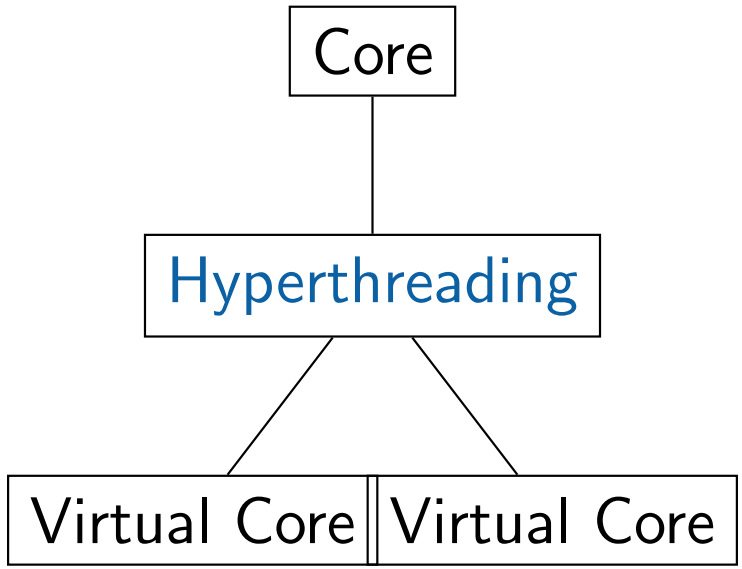
- 64 nodes
- 2 processors per node
- 16 cores per processor
- hyper threading

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

XGBoost

XGboost

This stands for:

Extreme Gradient Boosting

It has some advances over 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)