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

- GBM
- XGBoost
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
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 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` 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)
GBM: variable importance measure

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

- relative.influence
- permutation.test.gbm
  (This is currently labeled experimental)

These have similar definition relative to bagging, however they use all of the data instead of the OOB
GBM: sample code

```r
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)
```
GBM: Figures
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
Developed by Intel, Hyperthreading allows for each core to pretend to be two cores.

This works by trading off computation and read-time for each core.
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 specify your own loss/evaluation functions (You need to use `xgb.train` for this)