Mathematical techniques in data science

Lecture 18: Boosting

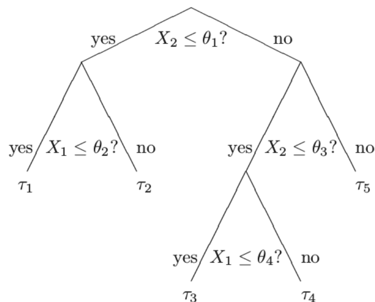
March 29th, 2019

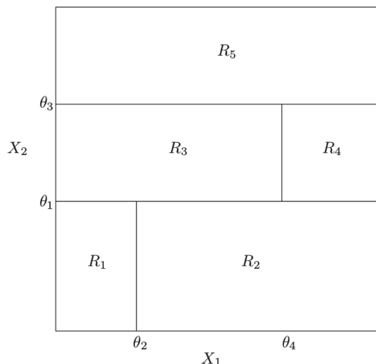
Schedule

Week	Chapter
1	Chapter 2: Intro to statistical learning
3	Chapter 4: Classification
4	Chapter 9: Support vector machine and kernels
5, 6	Chapter 3: Linear regression
7	Chapter 8: Tree-based methods + Random forest
8	
9	Neural network
12	PCA $\rightarrow$ Manifold learning
11	Clustering: K-means $\rightarrow$ Spectral Clustering
10	Bootstrap + Bayesian methods + UQ
13	Reinforcement learning/Online learning/Active learning
14	Project presentation

Tree-based methods:

- Partition the feature space into a set of rectangles.
- Fit a simple model (e.g. a constant) in each rectangle.
- Conceptually simple yet powerful.





Izenman, 2013, Figure 9.1.

How to grow a decision tree?

Regression tree:

- Data: $y \in \mathbb{R}^n$, $X \in \mathbb{R}^{n \times p}$.
- Each observation: $(y_i, x_i) \in \mathbb{R}^{p+1}$, $i = 1, \ldots, n$.

Suppose we have a partition of $\mathbb{R}^p$ into M regions $R_1, \ldots, R_m$.

We predict the response using a constant on each R_i:

$$f(x) = \sum_{i=1}^{m} c_i \cdot \mathbf{1}_{x \in R_i}.$$

In order to minimize $\sum_{i=1}^{n} (y_i - f(x_i))^2$, one needs to choose:

$$\hat{c}_i = \mathrm{ave}(y_j : x_j \in R_i).$$

How do we determine the regions R_i, i.e., how do we "grow" the tree?

We need to decide:

1. Which variable to split.
2. Where to split that variable.

Stopping rules and pruning

- Generally, the process is stopped for a given region when there are **less than** 5 observations in that region.

Problem with previous methodology:

- Likely to **overfit** the data.
- Can lead to poor prediction error.

Pruning the tree. Strategy: Grow a large tree (overfits), and the prune it (better).

- **Weakest link pruning:**

(a.k.a cost complexity pruning)
Let $T \subset T_0$ be a **subtree** of T_0 with $|T|$ **terminal nodes**. For $\alpha > 0$, define:

$$C_\alpha(T) := \sum_{m=1}^{|T|} \sum_{i:x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha \cdot |T|.$$

Pick a subtree minimizing $C_\alpha(T)$.

The bootstrap

- We saw before that decision trees often overfit the data.
- We will now discuss techniques that can be used to mitigate that problem.

Bootstrapping: General statistical method that relies on resampling data with replacement.

Idea: Given data (y_i, x_i), $i = 1, \ldots, n$, construct *bootstrap samples* by sampling n of the observations **with replacement** (i.e., allow repetitions):

Sample 1	Sample 2	Sample 3
(y_{i_1}, x_{i_1})	(y_{j_1}, x_{j_1})	(y_{k_1}, x_{k_1})
(y_{i_2}, x_{i_2})	(y_{j_2}, x_{j_2})	(y_{k_2}, x_{k_2})
$\vdots$	$\vdots$	$\vdots$
(y_{i_n}, x_{i_n})	(y_{j_n}, x_{j_n})	(y_{k_n}, x_{k_n})

- Each bootstrap sample mimics the statistical properties of the original data.
- Often used to estimate parameter variability (or uncertainty).

Bagging

Bagging:(bootstrap aggregation) Suppose we have a model $y \approx \hat{f}(x)$ for data $(y_i, x_i) \in \mathbb{R}^{p+1}$.

1. Construct $B \in \mathbb{N}$ bootstrap samples.
2. Train the method on the b-th bootstrap sample to get $\hat{f}^{*b}(x)$.
3. Compute the average of the estimators:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{i=1}^{B} \hat{f}^{*b}(x).$$

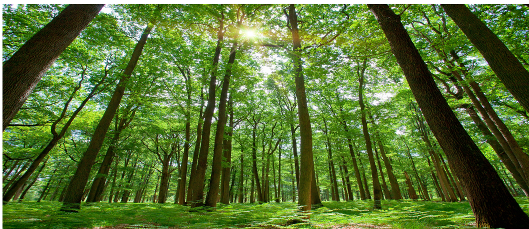
- Bagging is often used with regression trees.
- Can improve estimators significantly.

Note: Each bootstrap tree will typically involve different features than the original, and might have a different number of terminal nodes.

The bagged estimate is the average prediction at x from these B trees.

For classification: Use a majority vote from the B trees.

Random forests (cont.)



Random forests: Each time a split in a tree is considered, a random selection of m predictors is chosen as split candidates from the full set of p predictors.

- Typical value for m is $\sqrt{p}$.
- We construct $T_1, \dots, T_B$ trees using that method on bootstrap samples. The **random forest (regression) predictor** is

$$\hat{f}^B_{\text{rf}}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x).$$

For classification: use majority vote.

Boosting

Like bagging, boosting is a general approach that can be applied to many models. *Combines weak learners into a single strong learner.*

Boosting: Recursively fit trees to residuals. (Compensate the shortcoming of previous model.)

Input: $(y_i, x_i) \in \mathbb{R}^{p+1}$, $i = 1, \ldots, n$. Initialize $\hat{f}(x) = 0$, $r_i = y_i$.

For $b = 1, \ldots, B$:

1. Fit a tree estimator $\hat{f}^b$ with d splits to the training data.
2. Update the estimator using:
$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \cdot \hat{f}^b(x).$$
3. Update the residuals:
$$r_i \leftarrow r_i - \lambda \cdot \hat{f}^b(x_i).$$

Output: Boosted tree:

$$\hat{f}(x) = \sum_{i=1}^{B} \lambda \hat{f}^b(x).$$

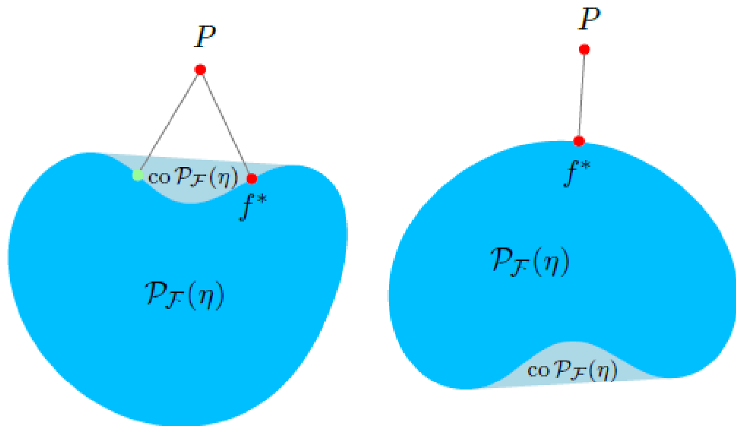
Note: $\lambda > 0$ is a *learning rate*.

Boosting

- Bagging
- Random forests
- Boosting

Main idea: we can combine weak learners into a single strong learner

Non-convexity of the hypothesis space



Moving out of the hypothesis space

$$H(x) = \sum_t \rho_t h_t(x)$$

$H = \text{sign}$ (0.42 [figure] + 0.65 [figure] + 0.92 [figure])

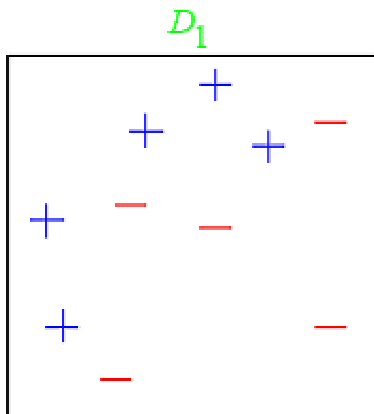
= [figure]

Adaboost

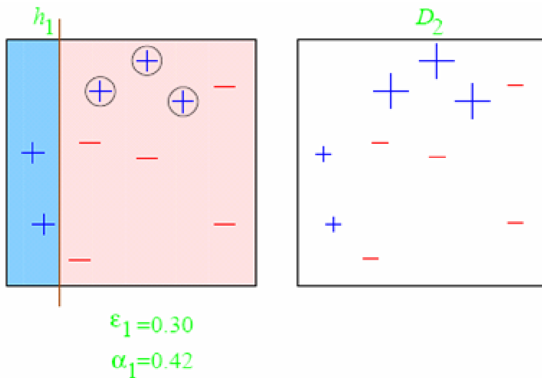
Algorithm 10.1 *AdaBoost.M1.*

1. Initialize the observation weights $w_i = 1/N,\ i = 1, 2, \ldots, N$.
2. For $m = 1$ to M:
   (a) Fit a classifier $G_m(x)$ to the training data using weights w_i.
   (b) Compute
   $$\mathrm{err}_m = \frac{\sum_{i=1}^N w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_i}.$$
   (c) Compute $\alpha_m = \log((1 - \mathrm{err}_m)/\mathrm{err}_m)$.
   (d) Set $w_i \leftarrow w_i \cdot \exp[\alpha_m \cdot I(y_i \neq G_m(x_i))],\ i = 1, 2, \ldots, N$.
3. Output $G(x) = \mathrm{sign}\left[\sum_{m=1}^M \alpha_m G_m(x)\right]$.

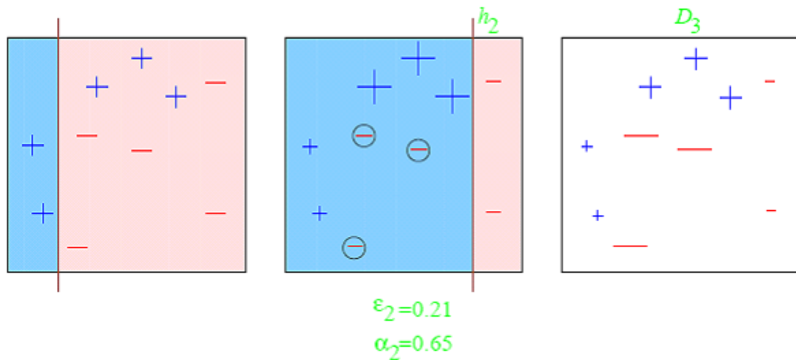
Adaboost



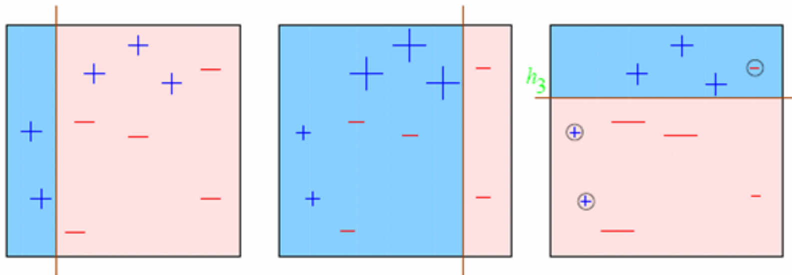
Adaboost



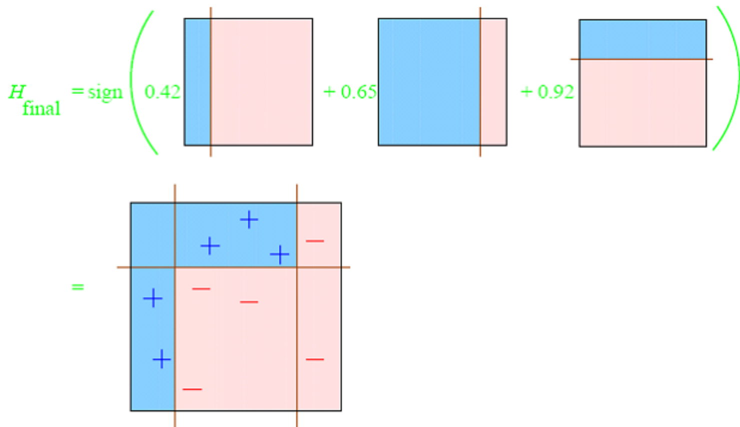
Adaboost



Adaboost

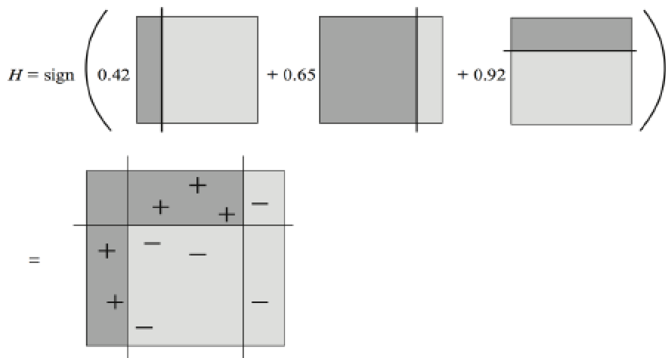


Adaboost

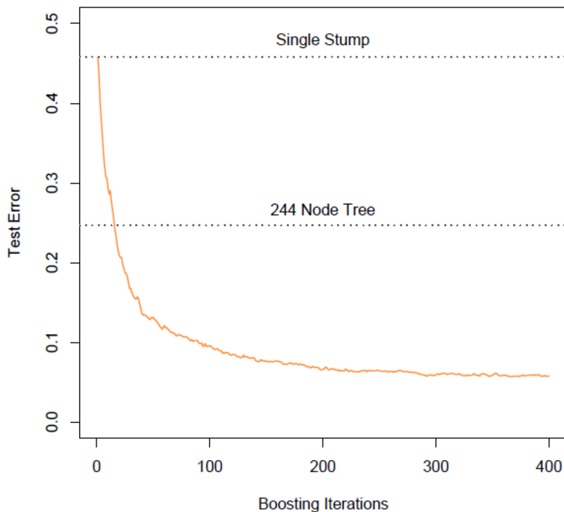


Adaboost

$$H(x) = \sum_t \rho_t h_t(x)$$



Gradient boosting



Gradient boosting

- Invent Adaboost, the first successful boosting algorithm [Freund et al., 1996, Freund and Schapire, 1997]
- Formulate Adaboost as gradient descent with a special loss function[Breiman et al., 1998, Breiman, 1999]
- Generalize Adaboost to Gradient Boosting in order to handle a variety of loss functions
  [Friedman et al., 2000, Friedman, 2001]

Gradient descent

Gradient Descent

Minimize a function by moving in the opposite direction of the gradient.

$$\theta_i := \theta_i - \rho \frac{\partial J}{\partial \theta_i}$$

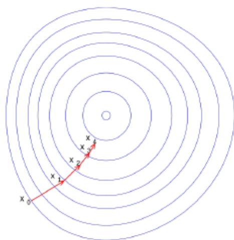


Figure: Gradient Descent. Source: http://en.wikipedia.org/wiki/Gradient_descent

Gradient boosting

Boosting: Recursively fit trees to residuals. (Compensate the shortcoming of previous model.)

Input: $(y_i, x_i) \in \mathbb{R}^{p+1}$, $i = 1, \ldots, n$. Initialize $\hat{f}(x) = 0$, $r_i = y_i$.

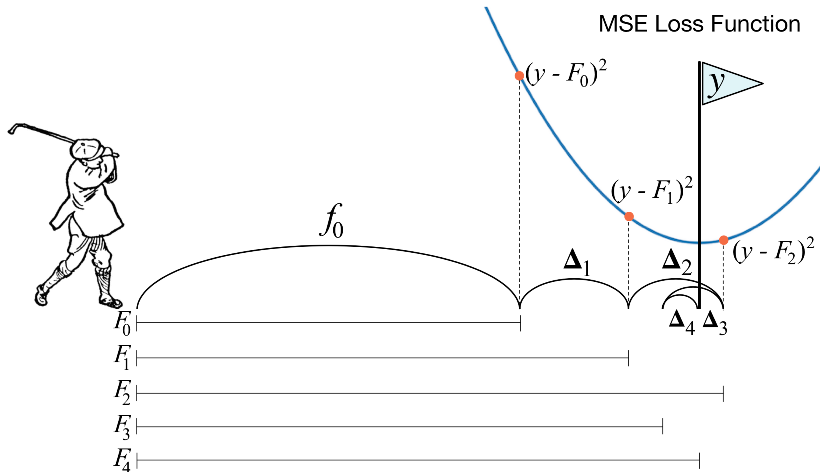
For $b = 1, \ldots, B$:

1. Fit a tree estimator $\hat{f}^b$ with d splits to the training data.
2. Update the estimator using:
$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \cdot \hat{f}^b(x).$$
3. Update the residuals:
$$r_i \leftarrow r_i - \lambda \cdot \hat{f}^b(x_i).$$

Output: Boosted tree:

$$\hat{f}(x) = \sum_{i=1}^{B} \lambda \hat{f}^b(x).$$

Gradient boosting



Gradient boosting

Algorithm 10.3 *Gradient Tree Boosting Algorithm.*

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm},\ j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Gradient boosting

sklearn.ensemble: Ensemble Methods

The `sklearn.ensemble` module includes ensemble-based methods for classification, regression and anomaly detection.

User guide: See the Ensemble methods section for further details.

`ensemble.AdaBoostClassifier` ([…])	An AdaBoost classifier.
`ensemble.AdaBoostRegressor` ([base_estimator, …])	An AdaBoost regressor.
`ensemble.BaggingClassifier` ([base_estimator, …])	A Bagging classifier.
`ensemble.BaggingRegressor` ([base_estimator, …])	A Bagging regressor.
`ensemble.ExtraTreesClassifier` ([…])	An extra-trees classifier.
`ensemble.ExtraTreesRegressor` ([n_estimators, …])	An extra-trees regressor.
`ensemble.GradientBoostingClassifier` ([loss, …])	Gradient Boosting for classification.
`ensemble.GradientBoostingRegressor` ([loss, …])	Gradient Boosting for regression.
`ensemble.IsolationForest` ([n_estimators, …])	Isolation Forest Algorithm
`ensemble.RandomForestClassifier` ([…])	A random forest classifier.
`ensemble.RandomForestRegressor` ([…])	A random forest regressor.
`ensemble.RandomTreesEmbedding` ([…])	An ensemble of totally random trees.
`ensemble.VotingClassifier` (estimators[, …])	Soft Voting/Majority Rule classifier for unfitted estimators.

Gradient boosting

dmlc
XGBoost eXtreme Gradient Boosting

build passing | build passing | build passing | docs passing | license Apache 2.0 | CRAN 0.82.1 | pypi package 0.82

Community | Documentation | Resources | Contributors | Release Notes

XGBoost is an optimized distributed gradient boosting library designed to be highly ***efficient***, ***flexible*** and ***portable***. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples.

License

© Contributors, 2016. Licensed under an Apache-2 license.

Gradient boosting

Unique features of XGBoost

XGBoost is a popular implementation of gradient boosting. Let's discuss some features of XGBoost that make it so interesting.

- **Regularization:** XGBoost has an option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting
- **Handling sparse data:** Missing values or data processing steps like one-hot encoding make data sparse. XGBoost incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data
- **Weighted quantile sketch:** Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data
- **Block structure for parallel learning:** For faster computing, XGBoost can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted and stored in in-memory units called blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling
- **Cache awareness:** In XGBoost, non-continuous memory access is required to get the gradient statistics by row index. Hence, XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored
- **Out-of-core computing:** This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory