28

ENSEMBLE AND BOOSTING METHODS

28.1 Motivation and overview

So far, we have covered several relatively simple machine learning algorithms, including logistic regression, SVM, and shallow decision trees. The resulting regressors or classifiers from these algorithms, which are often referred to **weak learners**, or **weak estimators**, or **base estimators**, generally have *low variance but with high bias* when facing complex machine learning problems. In this chapter we discuss several powerful techniques to combine weak learners into a stronger one, which is expected to have *lower bias and variance*, improved generalizability, and improved robustness compared to a single estimator. These combining techniques are collectively known as **ensemble learning**.

There are two basic approaches to ensemble learning. The main idea of the first approach is taking averages of predicted numerical values, or taking majority of classification probabilities over several weak learners. In averaging methods, the principle is to build several estimators independently and then to combine their predictions (e.g., by voting). On average, the combined estimator is usually better than any of the single base estimator because its variance is reduced. However, the bias might not be reduced as much.

We illustrate this point through an example. Consider there are $n = 11$ independent classifiers, each with probability p to vote correctly. If we take the majority vote, the probability to vote correctly is

$$Pr(major\ vote\ correct) = \sum_{k=6}^{n} \binom{n}{k} p^k (1-p)^{n-k}.$$

As long as $p > 0.5$ (i.e., better than a random guess), probability of correct majority vote is greater than probability of correct individual vote [Figure 28.1.1].

Another approach is known as **boosting**, where a series of weak learner are built sequentially, with the objective of reducing the bias of the combined estimator. We will cover AdaBoost, gradient boosting, and their recent variant XGBoot. However, the variance might not be reduced as much.

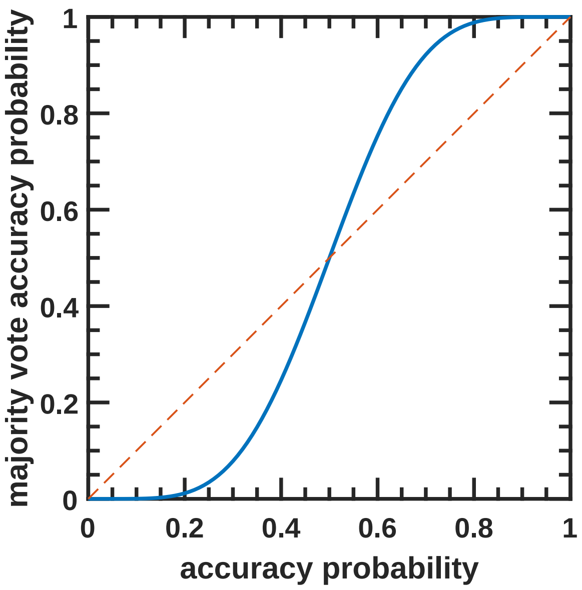

Figure 28.1.1: The correctness probability of a majority vote is greater than the correctness probability of individual votes when individual accuracy probability is greater than 0.5.

28.2 Bagging Methods

28.2.1 A basic bagging method

In the most vanilla bagging methods, we bootstrap the training data to create multiple bootstrap training samples. And we train different base estimators on each bootstrap sample. We form our final estimator by averaging over base estimators or taking majority vote. The algorithm is given by algorithm 41.

Besides this basic bagging strategy, there are also other bagging methods[1, p. 5]:

- Bayesian model averaging where estimators are combined with a weight proportional to their posterior probabilities.
- Random forest where base estimators built from data subsets randomly sampled from data sets are combined.

Bagging methods are a simple strategy to improve model testing performance and robustness. The underlying rationale is each base estimator are not fully correlated to each other, so by averaging the base estimator, we achieve variance reduction via Law of Large Numbers. However, bagging generally cannot reduce bias. Therefore bagging is more appealing to complex models that usually suffer from large variance and low bias (e.g., deep trees and neural networks).

Bagging methods also offer a convenient way to estimate generalization error (i.e., model prediction error on unseen data) [Methodology 22.2.1]. As we build a base estimator from bootstrapped samples on each iteration, around one third samples [Remark 13.5.1] will not be sampled and used to build the model. The rest of samples can be used to estimate generalization error [2].

Algorithm 41: A basic bagging algorithm

```
**Input:** Training data $D = \{(x_1, y_1), ..., (x_m, y_m)\}$ where $x_i \in X$, $y_i \in \mathcal{Y}$; Base
        learning alogorithm $\mathcal{L}$; Number of base learners $T$.
1 Initialize $t = 1$.
2 repeat
3     Generate bootstrapped sample $D_t$ from $D$ via sampling with replacement.
4     Train weak learner $h_t$ using bootstrapped sample $D_t$.
5     Set $t = t + 1$.
6 until $t = T$;
7 Construct the final classifiers/regressors $H$: The averaging method for regression
      $H(x) = \frac{1}{K}\sum_{i=1}^{K} h_i(x).$
  and the voting method for classification
      $H(x) = \arg\max_{y \in \mathcal{Y}} \sum_{i=1}^{K} \mathbf{1}(h_i(x) = y).$
**Output:** $H(x)$
```

28.2.2 Tree bagging

Decision trees with large depth largely suffer from **high variance**; on the other hand, shallow decision trees suffer from **high bias** despite their low variance. By applying bagging methods to decision trees, we are able to create estimators with the desired low bias and low variance properties.

Methodology 28.2.1 (tree bagging). *Given a set of training samples B_0 of size n, we can use bootstrap to create samples $B_1, ..., B_K$. From each sample set B_i, we can train a shallow decision tree model $\hat{f}^i(x)$. The averaging method for regression*

$$\hat{f}_{avg}(x) = \frac{1}{K}\sum_{i=1}^{K} \hat{f}^i(x).$$

and the voting method for classification

$$\hat{f}_{avg}(x) = \arg\max_{y\in\mathcal{Y}} \sum_{i=1}^{K} \mathbf{1}(\hat{f}^i(x) = y).$$

Increasing K will not result in overfitting, and usually large K is used.

Because bootstrapped training samples are more or less similar, the resulting trees will have correlation in terms of output values (output can be viewed as random variables since samples are randomly drawn). To see that how many trees are needed to can reduce variance, we have the following Lemma.

Lemma 28.2.1 (variance reduction). *An average of N i.i.d. random variables, each with variance σ^2, has variance $\frac{1}{N}\sigma^2$. If random variables have pairwise correlation ρ, the variance of the average is*

$$Var[\hat{X}] = Var[\frac{\sum_{i=1}^N X_i}{N}] = \rho\sigma^2 + \frac{1-\rho}{N}\sigma^2.$$

In particular,

- $Var[\hat{X}]_{N\to\infty} = \rho\sigma^2$
- *If* $\rho = 1$, $Var[\hat{X}] = \sigma^2$.

Proof.

$$\begin{aligned} Var[\frac{\sum_{i=1}^N X_i}{N}] &= \frac{\sum_{i=1}^N Var[X_i] + \sum_{i=1}^{N-1}\sum_{j=i+1}^N \rho\sqrt{Var[X_i]Var[X_j]}}{N^2} \\ &= \frac{\sum_{i=1}^N \sigma^2 + \sum_{i=1}^{N-1}\sum_{j=i+1}^N \rho\sigma^2}{N^2} \\ &= \frac{\rho\sigma^2}{N} + \frac{\sigma^2}{N}(1-\rho) \end{aligned}$$

□

Each base tree in the ensemble model should be controlled to avoid overfitting, which can be achieved by the following hyper-parameters.

Remark 28.2.1 (hyper-parameters controlling overfitting).

- **n_estimators** specifies the number of trees in the forest. In general, increasing tree number will increase performance, however, at the expense of computational time. It is desired to find a sweet spot that balance performance and computational cost.
- **max_depth** specifies the depth of each tree in the forest. The deeper the trees, the more splits they have, and more complex models will be generated to achieve better

fitting to the training data. However, increasing the depth of tree will also be likely to cause overfitting.

- **min_samples_split** specifies the minimum number of samples required in a node before splitting. Increasing the number will limit the depth, and thus the complexity, of the tree; decreasing the number might lead to overfitting.
- **min_samples_leaf** specifies the minimum number of samples required to construct a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. Increasing the number will limit the depth, and thus the complexity of the tree.
- **max_feature** specifies the number of features to be consider during the search of best splits. The best practice for classification problem is $\sqrt(2)$ of the total number of features.

28.2.3 Random Forest

Random Forests improve bagged decision trees in terms of reducing correlation between base trees, ultimately leading to a lower variance. Decision trees algorithm like CART using a greedy splitting algorithm that minimizes error. The base trees from bootstrapped samples can have considerable structural similarities and thus high correlation in their predictions. For example, if there is one strong predictor in the X, then all trees will choose it as a first split and as a result all trees are similar.

Combining highly correlated predictions from base models can significantly reducing the variance reduction power in ensemble learning, which works the best when predictions from base tree are uncorrelated or weakly correlated. Random forest adds randomness into the base tree training algorithm that enables the construction of less correlated base trees.

Specifically, when selecting a split point, random forest algorithms also search a random subset of features to determine splitting points instead of looking through all features as in CART.

The only difference of bagging and random forest is in the splitting. The random forest only randomly select a subset of predictors for splitting. The subset selection procedure can help de-correlate different trees trained from bootstrapped samples. Similar to bagging method, random forest method generally cannot reduce bias in the base model.

We summarize random forest construction in the following.

Methodology 28.2.2 (random forest construction). *[3, p. 320]*

- *Given a set of training samples B_1 of size n, we can use bootstrap to create samples $B_1, ..., B_K$.*
- *From each sample B_i, we can train a decision tree model $\hat{f}^i(x)$.* ***When a split is considered, we randomly select m features for splitting.***
- *The averaging method for regression*

$$\hat{f}_{avg}(x) = \frac{1}{K}\sum_{i=1}^{K} \hat{f}^i(x).$$

and the voting method for classification

$$\hat{f}_{avg}(x) = \arg\max_{y\in\mathcal{Y}} \sum_{i=1}^{K} \mathbf{1}(\hat{f}^i(x) = y).$$

Remark 28.2.2 (advantages of random forest)**.**

- Random Forests usually give high accuracy in classification and small error in regression tasks. They are quite competitive with the best known machine learning methods.
- Random forests are stable and robust to perturbations and noises. If we change the data a little or feature, the individual trees may change a lot but a random forest as a whole is stable in the sense of performance.

Remark 28.2.3 (hyperparameters)**.** Beside hyperparameters that controls the base learner, such as **max_depth**, **min_samples_split**, **min_samples_leaf**, and **max_feature**, the main hyperparameters that control random forest algorithm is **n_estimators** and **max_features**.

- **n_estimators** specifies the number of trees in the forest. In general, increasing tree number will increase performance, however, at the expense of computational time.
- **max_features** specifies the number of features to consider when splitting. Using smaller number of features during splitting can usually lead to greater the reduction of variance, but at the expense of increase the bias of each base tree. Empirically, we consider all features in regression tasks and consider squared root number of features for classification tasks.

Remark 28.2.4 (computational complexity)**.** In Remark 27.2.6, we have discussed that to grow a decision tree with H levels, given N training example and d features, the total cost will be $O(NdH)$. Starting from here, grow a random forest with B base tree, with each base tree having $p \leq d$ features will have a total cost of $O(BNpH)$.

In the testing stage, evaluate an example will cost $O(BH)$.

Remark 28.2.5 (extremely random forest, extra tree). We can add more randomness into the base tree learning algorithm, leading to an algorithm known as **extremely random forest**. Instead of computing the locally optimal feature/split combination (like the random forest), for each feature under consideration, we can select a random splitting point. This has two advantages: more diversified trees and less splitting evaluation.

28.3 Adaboost

28.3.1 Adaboost classifier

The core idea of Adaboost is to sequentially train multiple base learners based on a dynamically-weighted training sample set [Figure 28.3.1]. The weights are adjusted each round to give more weight on mis-classified examples. Final estimator is an weighted average of base estimators, with larger weight given to better performing base estimators.

Common weak estimators used in practice include: decision tree stumps, multi-layer neural networks, etc. In considering which base estimator can be boosted, we need to consider if base estimator can be trained with weighted samples. In the case where the base estimator cannot operate on weighted samples, we can re-sample the examples according to the distribution specified by the weights.

A generic Adaboost classifier algorithm is given by algorithm 42, with its core training workflow described by the diagram of Figure 28.3.1. Critical steps are adjusting sample weights and the weights associated with each base learner. Note that

- If an classifier misclassifies example i, then $y_i h_t(x_i) < 0$, and the weight of example i will be increased.
- If a base leaner has a smaller classification error, it will be associated with a larger weight in the final combined estimator.

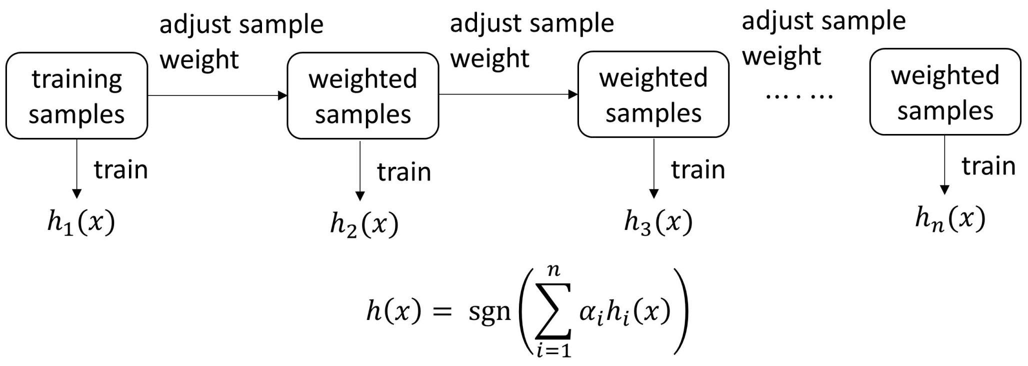

Figure 28.3.1: Illustration of adaptive boosting where sample weights are adjusted iteratively based on the classification error.

Algorithm 42: Generic Adaboost classifier algorithm

```
Input: Training data $(x_1, y_1), ..., (x_N, y_N)$ where $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y} = \{-1, 1\}$
1 Initialize sample $w_1(i) = 1/N$ and set $t = 1$.
2 repeat
3     Train weak learner $h_t$ using distributin $D_t$.
4     Compute the weighted error rate
          $$e_t = \sum_{i=1}^{N} w_t(i) I(h_t(x_i) \neq y_i).$$
      Choose
          $$\alpha_t = \frac{1}{2} \log \frac{1 - e_t}{e_t}.$$
5     Update sample weights:
          $$w_{t+1}(i) = \frac{w_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$
      where $Z_t$ is the normalizing constant given by
          $$Z_t = \sum_{i=1}^{N} w_t(i) \exp(-\alpha_t y_i h_t(x_i)).$$
6     $t = t + 1$.
7 until $t = T$;
8 Construct the final classifier:
          $$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x)).$$
Output: $H(x)$
```

Adaboost not only has the desired simplicity for practice purpose, it also has nice theoretical results regarding training error reduction. The following Lemma shows **the training error upper bound will decrease exponentially fast**. It worth noting that the key working mechanism in error reduction is via reducing bias, instead of variance reduction via averaging (e.g., random forest and bagging), since base estimators can be highly correlated.

Lemma 28.3.1 (error reduction in Adaboost). *[4, p. 139] Consider training data $D = \{(x_1, y_1), ..., (x_N, y_N)\}$ used to training a final Adaboost classifier H. The training error will satisfy*

$$\frac{1}{N}\sum_{i=1}^{N} I(H(x_i) \neq y_i) \leq \frac{1}{N}\exp(-y_i f(x_i)) = \prod_t Z_t$$

where $f(x) = \sum_t \alpha_t h_t(x), H(x) = sign(f(x))$.

Further more, let e_t be the weighted training error given as

$$e_t = \sum_{i=1}^{N} w_t(i) I(h_t(x_i) \neq y_i).$$

If $e_t < 0.5$ (i.e., better than random guess), then

$$\frac{1}{N}\sum_{i=1}^{N} \delta(H(x_i) \neq y_i) \leq \prod_{t=1}^{T} Z_t \leq \exp(-2\sum_{t=1}^{T}(1/2 - e_t)^2).$$

Proof. (1) When $H(x_i) \neq y_i$, $y_i f(x_i) > 1$ and $\exp(-y_i f(x_i)) > 1$; therefore,

$$I(H(x_i) \neq y_i) \leq \exp(-y_i f(x_i)).$$

Based on the definition

$$w_i^{(t)} \exp\left(-\alpha_t y_i h_t\left(x_i\right)\right) = Z_t w_i^{(t+1)}.$$

Then we can perform the following expansion

$$\begin{aligned}
&\frac{1}{N}\sum_i \exp\left(-y_i f\left(x_i\right)\right) \\
&= \frac{1}{N}\sum_i \exp\left(-\sum_{t=1}^{T} \alpha_t y_i h_t\left(x_i\right)\right) \\
&= \sum_i w_1(i) \prod_{t=1}^{T} \exp\left(-\alpha_t y_i h_t\left(x_i\right)\right) \\
&= Z_1 \sum_i w_2(i) \prod_{t=1}^{T} \exp\left(-\alpha_t y_i h_t\left(x_i\right)\right) \\
&= \cdots \\
&= Z_1 Z_2 \cdots Z_{T-1} \sum_i w_t(i) \exp\left(-\alpha_t y_i G_t\left(x_i\right)\right) \\
&= \prod_{t=1}^{T} Z_T
\end{aligned}$$

(2)

$$
\begin{aligned}
Z_t &= \sum_{i=1}^{N} w_t(i) \exp\left(-\alpha_m y_i h_t\left(x_i\right)\right) \\
&= \sum_{y_i = h_t(x_i)} w_t(i) e^{-\alpha_t} + \sum_{y_i \neq h_t(x_i)} w_t^{(} i) e^{\alpha_t} \\
&= \left(1 - e_t\right) \mathrm{e}^{-\alpha_t} + e_t e^{\alpha_t} \\
&= 2\sqrt{e_t\left(1 - e_t\right)} = \sqrt{1 - 4\gamma_m^2}
\end{aligned}
$$

□

28.3.2 Adaboost regressor

The similar sample re-weighted technique can be applied to regression problems as well to yield Adaboost regressors. In algorithm 43, weights of difficult examples are constantly increased to make the estimator focus on difficult examples[5].

Algorithm 43: Adaboost regressor algorithm

```
    Input: Training data $D_0 = \{(x_1, y_1), ..., (x_N, y_N)\}$ where $x_i \in X$,
           $y_i \in Y = \{-1, 1\}$
 1  Initialize sample $w_1(i) = 1$ and set $t = 1$.
 2  repeat
 3      Construct the data set $D_t$ by sampling from $D_0$. The probability of drawing
          sample $i$ is $p_i = w_i / \sum w_i$.
 4      Train weak learner using distributin $D_t$.
 5      Get weak classifier $h_t : X \to \mathbb{R}$.
 6      Compute the loss $L_i$ for sample $i$ in $D_0$.
 7      Compute the average loss associated with sample $D_t$: $\overline{L} = \sum_{i=1}^{|D_t|} L_i p_i$.
 8      Form $\beta = \frac{\overline{L}}{1-\overline{L}}$. $\beta$ is an indicator of prediction confidence. Low $\beta$ means high
          confidence.
 9      For each sample in $D_0$, update weight $w_i = w_i \beta^{(1-L_i)}$.
10      $t = t + 1$.
11  until $t = T$;
12  Construct the final predictor $h_f$:
13
        $$h_f = \inf\left\{ y \in Y : \sum_{t: h_l \leq y} \log\left(1/\beta_t\right) \geq \frac{1}{2} \sum_t \log\left(1/\beta_t\right) \right\}$$
      which is predictor has median prediction confidence (characterized by
      $\log(1/\beta_t)$).
    Output: $h_f(x)$
```

Remark 28.3.1. The loss function can take the following forms

$$L_i = \frac{|\hat{y}_i - y_i|}{D}$$

$$L_i = \frac{|\hat{y}_i - y_i|^2}{D^2}$$

$$L_i = 1 - \exp\left(-\frac{|\hat{y}_i - y_i|}{D}\right)$$

28.3.3 Additive model framework

28.3.3.1 *Generic additive model algorithm*

In ensemble learning, we are given training data $(x_1, y_1), ..., (x_N, y_N)$ and that we want to solve the following minimization problem

$$\min_{\beta_m, \gamma_m} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b\left(x_i; \gamma_m\right)\right)$$

where $b(x; \gamma)$ are base function, and β_i are scalar multipliers.

Directly solving all γ, b can be challenging. Adaboost approximate the solution by iteratively reweighing samples and training different base estimators based on weighted samples. An alternative way, known as **forward stagewise algorithm**, aims to solve γ_i, γ_i one at a time via a greedy strategy.

For example, at iteration t we solve for (β_t, γ_t) via

$$(\beta_t, \gamma_t) = \arg\min_{\beta, \gamma} \sum_{i=1}^{N} L(y_i, f_{t-1}(x_i) + \beta b(x_i; \gamma).$$

where $f_{t-1}(x) = \sum_{i=1}^{t-1} \beta_i b(x; \gamma_i)$.

It is quite clear that base estimator are introduced sequentially to reduce the bias, which ultimately leads to error reduction. Different from bagging (e.g., random forest), base estimators generated in the sequential process can be highly correlated, and weighted sum of base estimators usually offers insignificant variance reduction.

We will later show that additive model framework is a very general framework, which in fact includes Adaboost as a special case. By specifying different loss functions and base estimators, we greatly expand the realm of ensemble learning.

A generic additive model algorithm is given by algorithm 44.

Algorithm 44: A generic additive model algorithm

```
    Input: Training data $(x_1, y_1), ..., (x_N, y_N)$ where $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y} = \{-1, 1\}$, loss
           function $L$, candidate base functions $\{b(x;\gamma)\}$
1   Set $t = 1$, and $f_0(x) = 0$.
2   repeat
3       Train base estimator and compute estimator weight via
            $$(\beta_t, \gamma_t) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{t-1}(x_i) + \beta b(x_i;\gamma).$$
4       Update $f_t(x) = f_{t-1}(x) + \beta_t b(x;\gamma_t)$.
5       t = t + 1.
6   until $t = T$;
7   Construct the final estimators:
            $$f(x) = f_T(x).$$
    Output: $f(x)$
```

28.3.3.2 *Adaboost as a special additive model*

In this section, we show that **Adaboost algorithm is a type of additive model with exponential loss function.** The additive model framework can be later on generalize to gradient boosting machines.

Consider a forward stagewise additive model with

- loss function set to
$$L(y, h(x)) = \exp(-yh(x))$$
- the total loss is
$$\sum_{i=1}^{N} L(y_i, \sum_{m=1}^{M} \beta_M h_m(x_i)),$$
- optimal solution at step m is

$$(\beta_m, h_m) = \arg\min_{\beta,h} \sum_{i=1}^{N} \exp(-y_i(f_{m-1}(x_i) + \beta h(x_i))$$

Note that

$$\begin{aligned}(\beta_m, h_m) &= \arg\min_{\beta,h} \sum_{i=1}^{N} \exp(-y_i(f_{m-1}(x_i) + \beta h(x_i)) \\ &= \arg\min_{\beta,h} \sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta h(x_i))\end{aligned}$$

where

$$w_i^{(m)} = \exp(-y_i(f_{m-1}(x_i).$$

Minimizing with respect to h: Note that $y_i, h(x_i) \in \{-1, 1\}$, we have

$$\begin{aligned}\sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta h(x_i)) &= \sum_{y(i)=h(x_i)} w_i^{(m)} e^{-\beta} + \sum_{y(i)\neq h(x_i)} w_i^{(m)} e^{\beta} \\ &= e^{-\beta} \sum_{i=1}^{N} w_i^{(m)} + (e^{\beta} - e^{-\beta}) \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq h(x_i))\end{aligned}$$

If $\beta > 0$[1], then the minimizer h is given by

$$h_m(x) = \arg\min_{h} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq h(x_i)).$$

Minimizing with respect to β: We can rearrange the cost by

$$\sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta h(x_i)) = (\sum_{i=1}^{N} w_i^{(m)})(e^{-\beta} - (e^{-\beta} + e^{\beta}) err_m),$$

where

$$err_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq h(x_i))}{err_m}.$$

Then setting

$$\partial_\beta (\sum_{i=1}^{N} w_i^{(m)})(e^{-\beta} + (e^{-\beta} - e^{\beta}) err_m) = (\sum_{i=1}^{N} w_i^{(m)})(-e^{-\beta} + (e^{-\beta} + e^{\beta}) err_m) = 0,$$

gives

$$\beta_m = \frac{1}{2} \log(\frac{1 - err_m}{err_m}).$$

Remark 28.3.2. This step

$$(\beta_m, h_m) = \arg\min_{\beta,h} \sum_{i=1}^{N} w_i^{(m)} \exp(-y_i \beta h(x_i))$$

has the idea of get a base learner using weighted samples.

1 we show will that this is always true.

28.4 Gradient boosting machines

28.4.1 Fundamental

We have just seen that the additive model framework can be viewed as a generic framework to combine multiple weak learner to a final strong one, with Adaboost included as a special case. To reiterate, addictive models aim to solve the following minimization problem

$$\min_{\beta_{1:M},\gamma_{1:M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b\left(x_i;\gamma_m\right)\right)$$

where $b(x;\gamma)$ are base function, β_i are scalar multipliers, f are typical weak estimators such as decision tree stumps.

Rather than directly solving coefficients $\gamma_{1:M}$ and parameters $\beta_{1:M}$, a tremendously challenging problem, it is proposed to solve γ_m and β_m sequentially by iteratively solving following subproblem:

At iteration t we solve for (β_t, γ_t) via

$$(\beta_t, \gamma_t) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{t-1}(x_i) + \beta b(x_i;\gamma).$$

where $f_{t-1}(x) = \sum_{i=1}^{t-1} \beta_i b(x;\gamma_i)$.

Analytically solving the minimization subproblem is only possible for special cases. For example, when weak estimators are linear functions and loss function is mean squared error, we get the sequential linear regression method. When loss function is exponential loss, we get the Adaboost algorithm.

For an arbitrary loss function, a viable solution to the subproblem is gradient descent. Particularly, if we let $b(x;\gamma_t)$ approximate the gradient of L with respect to f_t, and let β be an appropriate step size that guarantees decrement of loss function, we are approximately performing gradient descent and can lead to decrement on the loss function. This is exactly the core idea underlying gradient boosting methods[6].

Altogether, we can view gradient boosting as a generic algorithm for additive modeling framework. By properly choosing suitable loss functions, we can unify regression and classification in the same framework:

- Common loss functions for regression include mean squared error, mean absolute deviation, Huber loss, etc.
- Common loss functions for classification include exponential loss ($Y \in \{-1, -1\}$), deviance (for $Y \in \{-1, 1\}$), and logistic loss (for $Y \in \{0, 1\}$).

A generic gradient boosting algorithm is given by algorithm 45. Core procedure in each iteration are:

- Compute gradient of L with respect to current combined learner.
- Train a base learner to approximate the gradient and solve the additive coefficient.
- Update the combined leaner.

Algorithm 45: Generic gradient boosting algorithm

```
  Input: Initial guess $f_0(x) = \arg\min_\alpha \sum_{i=1}^N L(y_i, \alpha)$
1 Set $m = 1$.
2 repeat
3     For $i = 1, 2, ..., N$ compute pseodu-residual
          $r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\Big|_{f=f_{m-1}}$
4     Fit a base estimator $b(x;\beta)$ to minimize
          $\beta_m = \arg\min_{\beta,\gamma} \sum_{i=1}^N (r_{im} - \gamma b(x_i;\beta))^2.$
5         $\alpha_m = \arg\min_\alpha L(y_i, f_{m-1}(x_i) + \alpha b(x_i;\beta_m)).$
6     Update $f_m(x) = f_{m-1}(x) + \alpha_m b(x;\beta_m)$.
7     Set $m = m + 1$.
8 until $m = M$;
9 Construct the boosted model,
          $f(x) = f_M$
  Output: the boosted model
```

28.4.2 Gradient boosting tree

In popular gradient boosting methods, the most widely used base estimators are decision tree stumps. We now discuss gradient boosting in the setting of trees for both regression and classification problems.

Since both regression and classification problems are unified by loss function specifications, in gradient tree boosting, we use regression tree stumps to fit gradients evaluated at the boosted function so far. Because a regression tree is representing a function

$f(x) = \sum_{j=1}^{J} \alpha_j 1(X \in R_j)$, where R_j are subsets partitioning the input space, α_j can be obtained by analytically or numerically minimizing a loss function at iteration m, given by

$$\alpha_j = \arg\min_{\alpha} \sum_{x_i \in R_j} L(y_i, f_{m-1}(x_i) + \alpha).$$

The complete algorithm is given by

Algorithm 46: Gradient tree boosting algorithm

```
Input: Initial guess $f_0(x) = \arg\min_\alpha \sum_{i=1}^N L(y_i, \alpha)$. Initial $f_0$ is a constant that
       minimizes the loss function.
1 Set $m = 1$.
2 repeat
3     For $i = 1, 2, ..., N$ compute pseodu-residual
          $$r_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}_{f=f_{m-1}}$$
4     Fit a regression tree to the targets $\{r_{im}\}$ givin terminal regions
       $R_{jm}, j = 1, 2, ..., J_m$
5     For $j = 1, 2, ..., J_m$, compute
          $$\alpha_{jm} = \arg\min_\alpha \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \alpha).$$
6     Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \alpha_{jm} I(x \in R_{jm})$.
7     Set $m = m + 1$.
8 until $m = M$;
9 Construct the boosted model,
          $$f(x) = f_M$$
Output: the boosted model
```

One critical component in this algorithm is solving the optimization problem of

$$\alpha_{jm} = \arg\min_{\alpha} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \alpha).$$

Given different loss functions specific to regression or classification problems, α can be solved either analytically or numerically, as we explore in details in the following.

Commonly used loss functions for regression problems and their solution to above optimization are given below[6][7, p. 360].

- If loss is $L(y,F) = |y-F|$, then
$$r_{im} = sign(y_i - f_{m-1}(x_i)),$$
and
$$\alpha_{jm} = median_{x_i \in R_{jm}}\{y_i - f_{m-1}(x_i)\}.$$
- If loss is $L(y,F) = \frac{1}{2}(y-F)^2$, then
$$r_{im} = (y_i - f_{m-1}(x_i)),$$
and
$$\alpha_{jm} = mean_{x_i \in R_{jm}}\{y_i - f_{m-1}(x_i)\}.$$
- If loss is Huber loss
$$L(y,F) = \begin{cases} \frac{1}{2}(y-F)^2, & |y-F| \le \delta \\ \delta(|y-F| - \delta/2) & |y-F| > \delta \end{cases}$$
then
$$r_{im} = \begin{cases} y_i - f_{m-1}(x_i), & |y_i - f_{m-1}(x_i)| \le \delta \\ \delta \cdot \text{sign}(y_i - f_{m-1}(x_i)), & |y_i - f_{m-1}(x_i)| > \delta \end{cases}$$
and
$$\tilde{r}_{jm} = \text{median}_{x_i \in R_{jm}}\{r_{im}\}$$
$$\alpha_{jm} = \tilde{r}_{jm} + \frac{1}{N_{jm}} \sum_{x_i \in R_{jm}} \text{sign}(r_{mi}) - \tilde{r}_{jm} \cdot \min\left(\delta_m, |r_{mi}| - \tilde{r}_{jm}\right)$$

For classification problems, we can use exponential loss ($Y \in \{-1,1\}$), deviance ($Y \in \{-1,1\}$), or logistic loss ($Y \in \{0,1\}$). Here we use logistic loss as an example to demonstrate numerical methods to solve coefficients α. Note that the output on the leaf node is a real number, instead of a label as in a regular decision tree.

The logistic loss for a sample (x_i, y_i) is the negative cross entropy given by
$$L_i = -y_i \log p_i - (1-y_i)\log(1-p_i)$$
where $p_i = \frac{1}{1+e^{-f(x_i)}}$ We can also write the L_i as
$$\begin{aligned} L_i &= -y_i \log \frac{p_i}{1-p_i} - \log(1-p_i) \\ &= \log(1+e^{f(x_i)}) - y_i f(x_i) \end{aligned}$$

Because

$$\alpha_j = \arg\min_{\alpha} \sum_{x_i \in R_j} L(y_i, f_{m-1}(x_i) + \alpha)$$

does not have an analytical solution. We use gradient descent to approximate α_j.

Note that the gradient L_i w.r.t. $f(x_i)$ is given by

$$\frac{\partial L_i}{\partial f(x_i)} = \frac{e^{y_i}}{1 + e^{y_i}} - y_i = p_i - y_i.$$

And the hessian is given by

$$\frac{\partial^2 L_i}{\partial f(x_i)^2} = \frac{\partial^2 \left(p_i - y_i\right)}{\partial f(x_i)^2} = \frac{\partial^2 p_i}{\partial f(x_i)^2} = p_i \left(1 - p_i\right)$$

We can approximate α be taking a Newton step

$$\alpha = -(\sum_{x_i \in R_j} \frac{\partial^2 L_i}{\partial f_{m-1}(x_i)^2})^{-1} (\sum_{x_i \in R_j} \frac{\partial L_i}{\partial f_{m-1}(x_i)}).$$

In other cases where Hessian is difficult or computational prohibitive to compute, we can take a gradient step

$$\alpha = -\gamma(\sum_{x_i \in R_j} \frac{\partial L_i}{\partial f_{m-1}(x_i)}).$$

where $\gamma > 0$ is the learning rate (usually small).

Remark 28.4.1 (hyperparameters in the algorithm). There are two categories of hyperparameter governing the training process of a gradient boosting machine.

- **Tree-specific parameters** controlling overfitting of individual base tree estimators [Remark 28.2.1].
- **Boosting parameters** that affect the boosting operation in the model. For example, **learning rate** α determines the impact of each tree on the final outcome. The learning parameter controls the magnitude of this change in the estimates during each step. Lower values are generally preferred as th make the model robust to the specific characteristics of tree and thus allowing it to generalize well. Lower values would require higher number of trees to model all the relations and will be computationally expensive.
  The **fraction of samples** to be selected to train each tree is another boosting parameter. Sample selection can be achieved by random sampling. Fraction less than 1 can make the model robust by reducing the variance via a mechanism similar to bagging and tree de-correlation.

Remark 28.4.2 (robustness of deviance and logistic loss over exponential loss). Deviance loss or logistic loss is usually more robust to outliers than exponential loss. To see this, consider $yf(x) < 0$, then

$$\log(1 + \exp(-2Yf(x))) \approx \log(\exp(-2Yf(x))) = -2Yf(x)$$

exponential loss allows big influence of observations with big negative margin, whereas deviance is less sensitive.

28.5 XGBoost

XGBoost[8] is one of most successful ensemble algorithms applied in applications with real-world complexities because of its speed, performance, and various features coping with real-world data issues (e.g., missing data).

Key elements in XGBoost aiming to speed up computational bottleneck. promote accuracy. and enhance robustness include

- Penalties are added to tree depth and coefficients in leaf node to regularize the tree growth and estimation process. We can view XGBoost as performing 'regularized boosting'.
- Using Newton step (involving both gradients and Hessian) in solving optimization subproblem to achieve better estimation of the coefficients in tree leaf nodes.
- Using quantiles and histograms to efficiently, approximately search best split points for continuous features.
- Using ideas from bagging and Random forest to perform data and feature down-sampling in training base trees, with an objective of reducing the correlation between the trees.
- Additional shrinkage parameters to downplay newly added tree and leaves space for future trees to improve the model.

In the following, we will mainly focus on the regularization and the Newton step. For a detailed account of comprehensive algorithmic parameters, see the documentation of XGBoost software.

Given n training examples $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, XGBoost has the following objective function for m step iteration

$$L = \sum_{i=1}^{n} l\left(y_i, f_i^{(m-1)} + b_m(x_i)\right) + \Omega(b_m)$$

where L is a regular loss function (e.g., mean squared error for regression tasks or logistic loss for classification tasks) used in gradient boosting tree, and the model complexity associated with a tree with J leaf nodes and output coefficients $\alpha_j, j = 1, ..., J$ are given by

$$\Omega(b_m) = \gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} \alpha_j^2.$$

To approximate the solution, we expand the above objective function to its quadratic form

$$\begin{aligned} L(y, f(x)) &= \sum_{i=1}^{n} l\left(y_i, f_i^{(m-1)} + b_m\left(x_i\right)\right) + \Omega\left(b_m\right) \\ &\approx \sum_{i=1}^{n}\left[l\left(y_i, f_{m-1}(x_i)\right) + g_i f_m\left(x_i\right) + \frac{1}{2} h_i f_m^2\left(x_i\right)\right] + \Omega\left(b_m\right) \end{aligned}$$

where

$$g_i = \frac{\partial l(y_i, f(x_i))}{\partial f(x_i)}_{f=f_{m-1}}, h_i = \frac{\partial^2 l(y_i, f(x_i))}{[\partial f(x_i)]^2}_{f=f_{m-1}}$$

Given a tree with J_m terminal regions, $R_{jm}, j = 1, 2, ..., J_m$, the objective function without the constant $l\left(y_i, f_{m-1}(x_i)\right)$ becomes

$$\begin{aligned} L'(y, f(x)) &\simeq \sum_{i=1}^{n}\left[g_i f_m\left(x_i\right) + \frac{1}{2} h_i f_t^2\left(x_i\right)\right] + \gamma J_m + \lambda \frac{1}{2} \sum_{j=1}^{J_m} \alpha_{jm}^2 \\ &= \sum_{j=1}^{J_m}\left[\left(\sum_{x_i \in R_{jm}} g_i\right) \alpha_{jm} + \frac{1}{2}\left(\sum_{x_i \in I_j} h_i + \lambda\right) \alpha_{jm}^2\right] + \gamma J_m \end{aligned}$$

Denote

$$G_j = \sum_{x_i \in R_{jm}} g_i, H_j = \sum_{x_i \in R_{jm}} h_i.$$

The optimal coefficients α_{jm} and optimal objective function are given by

$$\alpha_{jm}^* = \arg\min_{\alpha} G_j \alpha + \frac{1}{2}(H_j + \lambda)\alpha^2 = -\frac{G_j}{H_j + \lambda},$$

Plug in α_{jm}^*, and the optimal objective function value becomes

$$L^* = -\frac{1}{2} \sum_{j=1}^{J_m} \frac{G_j^2}{H_j + \lambda} + \gamma J_m$$

where γJ_m reflects the penalty on the structure complexity.

We further define a Gain quantity to determine **where to split and whether to continue splitting**. Gain is given by

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{\left(G_L + G_R\right)^2}{H_L + H_R + \lambda}\right] - \gamma$$

where G_L, H_L are gradients and Hessians on the left child, G_R, H_R are gradients and Hessians on the right child,

We use following method to determine the split point.

Methodology 28.5.1 (search best split point in XGBoost). *For each node, enumerate over all features*

- *For each feature, sorted the instances by feature value.*
- *Use a linear scan to decide the best split (maximum Gain) associated with that feature.*
- *Use the best split solution along all the features as the best split point.*

The Gain quantity can also be used to decide whether to stop growing or post-pruning a grown tree. For example

- We can stop splitting if the best split has negative gain and the gain within a threshold.
- We can also grow a tree to its maximum depth, then recursively prune splits that has negative gain.

The complete XGBoost algorithm is given by algorithm 47.

Algorithm 47: XGBoost algorithm

```
Input: Initial guess $f_0(x) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$.
1 Set $m = 1$.
2 repeat
3     For $i = 1, 2, ..., N$ compute graident and Hessian
          $$g_{im} = -\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\bigg|_{f=f_{m-1}}, g_{im} = \frac{\partial^2 L(y_i, f(x_i))}{[\partial f(x_i)]^2}\bigg|_{f=f_{m-1}}$$
4     Grow a regression tree to based on gain criterion.
5     Suppose the tree ends up with $J_m$ terminal regions $R_{jm}, j = 1, 2, ..., J_m$, then
        the output coefficients are
          $$\alpha^*_{jm} = \arg\min_\alpha G_j\alpha + \frac{1}{2}(H_j + \lambda)\alpha^2 = -\frac{G_j}{H_j + \lambda},$$
        where
          $$G_j = \sum_{x_i \in R_{jm}} g_i, H_j = \sum_{x_i \in R_{jm}} h_i.$$
6     Update $f_m(x) = f_{m-1}(x) + \epsilon \sum_{j=1}^{J_m} \alpha_{jm} I(x \in R_{jm})$.
7 until $m = M$;
8 Construct the boosted model,
          $$f(x) = f_M$$
Output: the boosted model
```

Remark 28.5.1 (computational time complexity).

- A rough estimate of time complexity of growing a tree of contains K level on N data examples with d features is $(dKN\log(N))$: on each level, sorting the data for splitting on each feature requires $O(N\log N)$ and we have d features.
- Further optimization can be realized by caching pre-sorted examples.

Remark 28.5.2 (recent improvements over XGBoost). LightGBM[9] is a recent improved GBM and offers significantly faster speeds over XGBoost in large scale applications. In face of large training data size and feature space, lightGBM introduces two techniques called Gradient based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS is down sampling technique that preserve most informative examples, characterized by large gradients, and randomly sample less informatative examples, characterized by small gradients. EFB exploits the fact that large-scale datasets used in real applications are usually quite sparse and group exclusive features (they are rarely taking non-zero value at the same time) in a near lossless way to reduce dimensionality.

28.6 Notes on Bibliography

For ensemble learning methods, see [1].

For boosting, see [10].

For gradient boost machine, see [6].

BIBLIOGRAPHY

1. Seni, G. & Elder, J. F. Ensemble methods in data mining: improving accuracy through combining predictions. *Synthesis Lectures on Data Mining and Knowledge Discovery* **2,** 1–126 (2010).
2. Wolpert, D. H. & Macready, W. G. An efficient method to estimate bagging's generalization error. *Machine Learning* **35,** 41–55 (1999).
3. James, G., Witten, D., Hastie, T. & Tibshirani, R. *An introduction to statistical learning* (Springer, 2013).
4. Li, H. *Statistical Learning Methods* (Tsinghua University, 2011).
5. Drucker, H. Improving regressors using boosting techniques. *ICML* **97,** 107–115 (1997).
6. Friedman, J. H. Greedy function approximation: a gradient boosting machine. *Annals of statistics,* 1189–1232 (2001).
7. Friedman, J., Hastie, T. & Tibshirani, R. *The elements of statistical learning (2017 corrected version)* (Springer series in statistics Springer, Berlin, 2007).
8. Chen, T. & Guestrin, C. *Xgboost: A scalable tree boosting system* in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining* (2016), 785–794.
9. Ke, G. *et al. Lightgbm: A highly efficient gradient boosting decision tree* in *Advances in neural information processing systems* (2017), 3146–3154.
10. Schapire, R. E. & Freund, Y. *Boosting: Foundations and algorithms* (MIT press, 2012).