Ensemble Learning

An ensemble method tries to make a strong learner by combining the predictions of many weak learners (**whose accuracy is a little bit higher than random learner**). Actually, for binary classification, we could prove that the majority vote makes correct decision if the weak classifies are **independent** to each other. It's called condorcet's rule. Two core elements have to be considered in ensemble learning: accuracy and diversity.

Ensemble learning is a method which could be used in both regression and classification and now a lot of ensemble algorithms are tree-based. There are two strategies to ensemble weak learners: boosting and bagging.

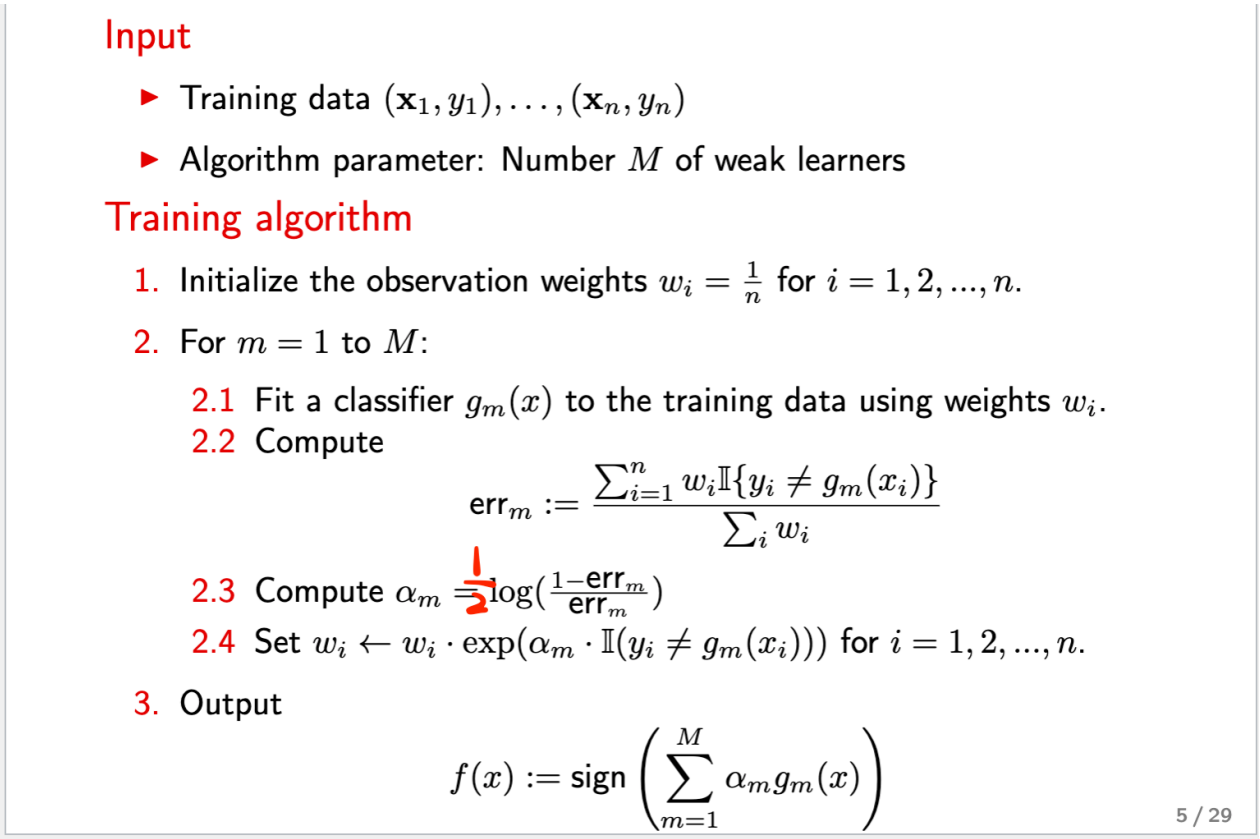
Boosting

Boosting is composed of:

- additive model $F(x) = \Sigma_{m=1}^{M} \alpha_m g_m(x)$, where $g_m(x)$ is a weak learner.
- reweighted samples after training each weak learner.

The most famous algorithm is Adaboost.

1. Algorithm - Adaboost



Note: 1. standard adaboost could only be used in binary classification problem; 2. the weak learn whose error rate is larger than 0.5 would be discarded in iteration.

The idea of adaboost is to make the sample which is classified into a wrong class currently have a larger weight. In this case, if we want to minimize our loss, it's better to classify these wrong-classified samples correctly in later iterations.

2. Loss Function - Adaboost

The loss function used in Adaboost is exponential loss function:

$$L(y, f(x)) = e^{-yf(x)}$$

Actually we could derivate the updating policy for parameters in Adaboost from the perspective of minimizing loss function. The final policy is the same as the first part stated. The specific derivation could be found a chinese version in here.

3. Additionals

- How to use it in multi-classification problem?

  As we stated before, the standard adaboost could only be used in binary classification problem. What if we want to apply this algorithm for multi-classes problem? We could transform the standard algorithm in three ways:

  - choose weak learners that can handle multiple classification.
  - still use binary weak learners and combine the output and features into new sample set. Then train a multi-classifier based on the new sample set.
  - encode label by using m bits binary code and then train m weak learners. The final output is the label we predict.

Bagging

As we stated before, we want the weak learners are independent to each other so as to get a strong learner with high ability of generalization. In order to do this, bagging samples dataset randomly by **boostrap** (sample with replacement) in each iteration of training. The most famous algorithm is random forest.

1. Algorithm - Random Forest

- For t = 1, 2, 3, ... , T repeat
  - Randomly sample a subset of dataset
  - **Randomly sample a subset of features**
  - Train a **decision tree** $h_t(x)$on the subset
- Output: if it's for classification, we could use majority vote (*Note: if there are two classes with same votes, we could just pick one as the prediction randomly*) to predict while for regression, we could compute the average.

Random forest is an variation of bagging. It not only sample dataset randomly, but also make the features in each iteration to be different. This stragegy makes the weak learners in random forest more diverse/independent to each other.

2. Additionals

- Unlike adaboost, random forest could be used in both regression and classification (even multi-classification).

- Bagging is a stragety of ensemble learning which means the weak learners do not have to be decision trees. However, **for random forest, it is an bagging-idea-based algorithm with decision trees as weak learners**.
- Disadvantage: since each time we fit a decision tree to a boostrap sample, we will get a different tree. It would decrease the interpretability of our model.
- What is **out of bag estimate** in bagging?

  Since boostrap is a sample method with replacement, there always are some data which are not used in each training (*It could be shown that on average around 37% of the observations would not be drawned in each boostrap.*). Naturally, we could use these unsampled dataset to do cross validation so as to estimate the test error of a bagging estimate which is called out-of-bag estimate.
- What is **importance of feature** in random forest?

  We would still find the best splitting feature even though only a random subset of features would be considered in each iteration. This actually could help us evaluate the importance of each variable because for each predictor, we could add up the total amount by which the loss decreases every time we use the predictor in the decision tree. Finally, we would get the importance of feature by averaging over all weak learners.
- A trick of choosing the number of trees in random forest: In practise, we could use $\sqrt{p}$ to be the number of decision trees. But of course, we could also use cross validation to tune this parameter.

Variance & Bias Analysis

- Boosting focus on decreasing bias while bagging focus on decreasing variance.
- Classical bagging algorithms use all features every time and find the best split feature. It could lead that all trees are similar to each other. However, random forest randomly choose features every time also so that it could make variance lower (better generalization) than classical bagging algorithms.
- Why do bagging's weak learners always have larger depth than boosting?

  As we stated that bagging focus on decreasing variance but we could make the bias not too big by make the weak learner deeper. However, for boosting, bias is not the main issue but the variance, so in order to make the variance smaller (avoid overfitting), we would choose to prune the trees.

Stragies to Ensemble Learners

- Averaging: usually for regression problem
  - Simple averaging
  - Weighted averaging
- Voting: usually for classification problem
  - Majority vote
  - Weighted vote
- Stacking
  - Step 1: randomly choose a subset of dataset with replacement

- Step 2: train T weak learners and then get predictions for unused samples
- Step 3: combine predictions in step 2 and unused samples' labels to train a meta learner

Stragies to Increase Diversity: Adding randomness

For ensemble idea, we always want the weak learners to be more diverse/independent. Basically, there are four ways could help us add diversity.

- Add randomness on dataset, such as bagging
- Add randomness on features, such as random forest
- Add randomness on output, such as flipping output (make part of 1s label to be 0) or output smearing(make classification labels to be numerical and then train a regression model)
- Add randomness on parameters of learners

Advanced Algorithm 1: Gradient Boosting Model

Ref: https://www.gormanalysis.com/blog/gradient-boosting-explained/ & http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/slides/gradient_boosting.pdf

Gradient boosting model = boosting + gradient decent. It could be used in both classification and regression.

1. Algorithm

Algorithm 1 Friedman's Gradient Boost algorithm

Inputs:

- input data $(x, y)_{i=1}^{N}$
- number of iterations M
- choice of the loss-function $\Psi(y, f)$
- choice of the base-learner model $h(x, \theta)$

Algorithm:

1: initialize $\widehat{f}_0$ with a constant
2: **for** $t = 1$ to M **do**
3: compute the negative gradient $g_t(x)$
4: fit a new base-learner function $h(x, \theta_t)$
5: find the best gradient descent step-size ρ_t:

$$\rho_t = \arg\min_{\rho} \sum_{i=1}^{N} \Psi\left[y_i, \widehat{f}_{t-1}(x_i) + \rho h(x_i, \theta_t)\right]$$

6: update the function estimate:

$$\widehat{f}_t \leftarrow \widehat{f}_{t-1} + \rho_t h(x, \theta_t)$$

7: **end for**

*Note: in step 5 we could use **line search** to find the best step size ρ_t but in practise we could also make it to be 1 or a small value like learning rate.*

The most famous algorithm in GBM is GBDT (gradient boosting with decision trees).

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 $\hat{f}(x) = f_M(x)$.

Note: No matter regression or classification, we use regression tree to fit the negative gradient. Actually, it could be proved that if we use cross entropy in classification problem, the final term of negative gradient is similar to regression with mean square loss.

TABLE 10.2. *Gradients for commonly used loss functions.*

Setting	Loss Function	$-\partial L(y_i, f(x_i))/\partial f(x_i)$
Regression	$\frac{1}{2}[y_i - f(x_i)]^2$	$y_i - f(x_i)$
Regression	$\lvert y_i - f(x_i)\rvert$	$\text{sign}[y_i - f(x_i)]$
Regression	Huber	$y_i - f(x_i)$ for $\lvert y_i - f(x_i)\rvert \le \delta_m$ $\delta_m \text{sign}[y_i - f(x_i)]$ for $\lvert y_i - f(x_i)\rvert > \delta_m$ where $\delta_m = \alpha\text{th-quantile}\{\lvert y_i - f(x_i)\rvert\}$
Classification	Deviance	kth component: $I(y_i = \mathcal{G}_k) - p_k(x_i)$

Note: for classification, if we define $p_k(x_i) = \frac{1}{1+e^{-\hat{y}_i}}$, we could definitely use the regression tree to fit $\hat{y}_i$ and then calculate the negative gradient for next iteration. (*Ref:* https://zhuanlan.zhihu.com/p/46445201)

2. Additionals

- Negative gradient $\neq$ Residuals

  If we take mean square error as the loss function, of course we would get the negative gradient which is exactly the residual we define for regression problem. However, it is not the whole story. For GBM, we use **negative gradient** to approximate the truth rather than residuals.

- What's the difference between GBM and Adaboost?

  Basically, GBM is a general idea of algorithm. Only if you can define a differentiable loss function, you could try to use GBM to train a strong learner. In this perspective, GBM is a generic algorithm to find approximate solutions to the additive modeling problem, while AdaBoost can be seen as a special case with a particular loss function. Hence, gradient boosting is much more flexible.

  What's more, AdaBoost can be interepted from a much more intuitive perspective and can be implemented without the reference to gradients by reweighting the training samples based on classifications from previous learners.

  One thing needs to be state: **Adaboost does not belong to GBM!**

Advanced Algorithm 2: XGBOOST

XGBOOST (Extreme Gradient Boosting) is a optimized version of gradient boosting algorithm. For most of tree-based algorithms, we have two main challenges to consider: **the speed of training on large dataset and overfitting**. In order to solve these two problems, XGBOOST makes some improvements on classical GBDT.

1. Algorithm

Algorithm 1: XGBoost Algorithm

Input: the image feature: $feat_z,\ z \in \{1, \ldots, n\},\ y_i \subseteq C, C = \{c_1,\ c_2,\ \cdots,\ c_l\}$, the loss function: $Loss_{XGBoost}(y, f(x))$, the total number of sub-tree: M;

Output: the estimated probability of image feature $feat_z$

(1) Repeat

(2) Initialize the m-th tree $f_m(x_i)$

(3) Compute $g_i = \partial_{\hat{y}_i^{(m-1)}} Loss_{XGBoost}\left(y_i, \hat{y}_i^{(m-1)}\right)$

(4) Compute $h_i = \partial^2_{\hat{y}_i^{(m-1)}} Loss_{XGBoost}\left(y_i, \hat{y}_i^{(m-1)}\right)$

(5) Use the statistics to greedily grow a new tree $f_m(x_i)$:

$$obj^{(m)} = -\frac{1}{2}\sum_{j=1}^{M}\frac{G_j^2}{H_j + \lambda} + \gamma M$$

(6) As shown in Equation (13), add the best tree $f_m(x_i)$ into the current model

(7) Until all M sub-trees are processed

(8) Obtain a strong regression tree based on all weak regression sub-trees

(9) Output the estimated probability based on the strong regression tree

- Why do we define the loss function like (5) ?

  Ref: https://hanszhuang.github.io/2017/09/01/GBDT%20%E7%AE%97%E6%B3%95%E5%8E%9F%E7%90%86 and https://speakerdeck.com/datasciencela/tianqi-chen-xgboost-overview-and-latest-news-la-meetup-talk?slide=16

Intuitively, we usually define our loss function as

$$L = \Sigma_i l(\hat{y}_i, y_i) + \Sigma_k \Omega(f_k)$$

where $\Omega(f_k)$ is the regularization term. For tree-based algorithms, we often use the number of leaves(T) or the score of leaf node(w) to evaluate model complexity. Therefore, it's natural to define $\Omega(f) = \gamma T + \frac{1}{2}\lambda||\omega||^2$.

And since we know $\hat{y}_i^t = \hat{y}_i^{t-1} + f_t(x_i)$, we could get the loss:

$$L^t = \Sigma_{i=1}^n l(\hat{y}_i^{t-1} + f_t(x_i), y_i) + \Omega(f_t)$$

Then if we use the 2nd order Taylor expansion to approximate the loss, we would get

$$L^t \approx \Sigma_{i=1}^n [l(\hat{y}_i^{t-1}, y_i) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

where $g_i = \partial_{\hat{y}^{t-1}} l(\hat{y}_i^{t-1}, y_i)$ and $h_i = \partial^2_{\hat{y}^{t-1}} l(\hat{y}_i^{t-1}, y_i)$. And then remove the constant term $l(\hat{y}_i^{t-1}, y_i)$ which does not matter to $f_t(x_i)$, we coule get

$$L^t \approx \Sigma_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

Since $f(x) = w_{q(x)}$ in terms of tree structure, then

$$L^t \approx \Sigma_{i=1}^n [g_i w_{q(x_i)} + \frac{1}{2} h_i w^2_{q(x_i)}] + \gamma T + \lambda \frac{1}{2} \Sigma_{j=1}^T w_j^2$$

Then we define $I_j = i$ and $q(x_i) = j$, we could unify the loss

$$\begin{aligned} L^t &= \Sigma_{j=1}^T [(\Sigma_{i \in I_j} g_i) w_j + \frac{1}{2} (\Sigma_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \\ &= \Sigma_{j=1}^T [G_j w_j + \frac{1}{2} (H_i + \lambda) w_j^2] + \gamma T \end{aligned}$$

When we know the tree structure $q(x)$, we could minimize the L_t by making its derivative to be 0. Finally we would get

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

In this case, the minimal loss is

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

- How to get the tree structure $q(x)$?

  In the first question, we supposed that we've already know the tree structure. But how to get the tree structure? Of course, we could not go over all possible tree structures because it is a NP complex problem. As we used in decision trees, we could try greedy algorithm to find the best feature and value to split in each step. Here, two problems need to be considered:

  - What metric should be used to evaluate the best?

    In decision trees, we've introduced information gain, gain ratio or gini index as splitting metrics. What about XGBOOST?

    Let's look at the loss function again. Minimizing the loss function means maximizing $\frac{G_j^2}{H_j + \lambda}$. A natural idea is to define the gain like this:

$$gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$

The larger gain, the better splitting point.

- Is it good to try all values in one feature and then find the best splitting point?

  Of course, if we go over all values in the feature, then we would definitely get the best! However, this actually the main reason for why tree-based algorithms take a long time on training. XGBOOST proposed **an approximate algoritm** for split finding.

Algorithm 2: Approximate Algorithm for Split Finding

for $k = 1$ ***to*** m **do**
  Propose $S_k = \{s_{k1}, s_{k2}, \cdots s_{kl}\}$ by percentiles on feature k.
  Proposal can be done per tree (global), or per split(local).
end
for $k = 1$ ***to*** m **do**
  $G_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$
  $H_{kv} \leftarrow= \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$
end
Follow same step as in previous section to find max score only among proposed splits.

  The approximate algorithm only takes percentiles into account which reduces the computation a lot. As the algorithm stated that, we could propose the split candidates before building tree(global) or each split(local). Global proposal is fast but its precision is not so good because it decides the splitting candidates before building tree which means some value would never have the chance to be used as the splitting point. In practise, if we choose to use global proposal, we would make the number of percentiles to be larger while if we choose local proposal, the number could be a little bit smaller.

Actually, XGBOOST uses **weighted quantile sketch** to propose the candidates for splitting. It used h_i to weigh and then choose the corresponding quantiles to be candidates. Why h_i? That's because we could rewrite the loss function like

$$\Sigma_{i=1}^{n} \frac{1}{2} h_i (f_t(x_i) - \frac{g_i}{h_i})^2 + \Omega(f_t) + constant$$

Therefore, we would like to use h_i as the weights.

- How to find split candidates when the dataset is sparse?

  XGBOOST also provides an algorithm to solve sparse dataset.

Algorithm 3: Sparsity-aware Split Finding

Input: I, instance set of current node
Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$
Input: d, feature dimension
Also applies to the approximate setting, only collect statistics of non-missing entries into buckets
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I}, g_i, H \leftarrow \sum_{i \in I} h_i$
for $k = 1$ ***to*** m **do**
  // enumerate missing value goto right
  $G_L \leftarrow 0,\ H_L \leftarrow 0$
  for j *in sorted*$(I_k,$ *ascent order by* $\mathbf{x}_{jk})$ **do**
    $G_L \leftarrow G_L + g_j,\ H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L,\ H_R \leftarrow H - H_L$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
  // enumerate missing value goto left
  $G_R \leftarrow 0,\ H_R \leftarrow 0$
  for j *in sorted*$(I_k,$ *descent order by* $\mathbf{x}_{jk})$ **do**
    $G_R \leftarrow G_R + g_j,\ H_R \leftarrow H_R + h_j$
    $G_L \leftarrow G - G_R,\ H_L \leftarrow H - H_R$
    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
  end
end
Output: Split and default directions with max gain

Note: classify the missing value into left and right and then compare the gain.

2. Additionals

Till now, we've introduced some improvements on XGBOOST compared with GBM:

- Model improvement: regularized loss for better model (avoid overfitting)
- Algorithm improvements: weighted quantile sketch and sparse aware algorithm.(speeding up)

Other highlights of XGBOOST focus on speeding up computation via system design.

- Column block for parallel learning (sort in order)
- Out-of-core computation: block compression and block sharding
- Cache optimization
- Distributed computing