

XGBoost

参考文献:

1. 陈天奇PPT链接: <https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf>
2. 【机器学习笔记】xgboost陈天奇PPT逐页翻译详解: https://blog.csdn.net/weixin_45551676/article/details/106049819

Outline

- Review of key concepts of supervised learning
- Regression Tree and Ensemble (What are we Learning)
- Gradient Boosting (How do we Learn)
- Summary

Elements in Supervised Learning

- Notations: $x_i \in \mathbb{R}^d$ i-th training example
- Model: how to make prediction \hat{y}_i given x_i
 - Linear model: $\hat{y}_i = \sum_j w_j x_{ij}$ (include linear/logistic regression)
 - The prediction score \hat{y}_i can have different interpretations depending on the task
 - Linear regression: \hat{y}_i is the predicted score
 - Logistic regression: $1/(1 + \exp(-\hat{y}_i))$ is predicted the probability of the instance being positive
 - Others... for example in ranking \hat{y}_i can be the rank score
- Parameters: the things we need to learn from data
 - Linear model: $\Theta = \{w_j | j = 1, \dots, d\}$

Elements continued: Objective Function

- Objective function that is everywhere

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

Training Loss measures how well model fit on training data

Regularization, measures complexity of model

- Loss on training data: $L = \sum_{i=1}^n l(y_i, \hat{y}_i)$

- Square loss: $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$
- Logistic loss: $l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})$

- Regularization: how complicated the model is?

- L2 norm: $\Omega(w) = \lambda \|w\|^2$
- L1 norm (lasso): $\Omega(w) = \lambda \|w\|_1$

Putting known knowledge into context

- Ridge regression: $\sum_{i=1}^n (y_i - w^T x_i)^2 + \lambda \|w\|^2$
 - Linear model, square loss, L2 regularization
- Lasso: $\sum_{i=1}^n (y_i - w^T x_i)^2 + \lambda \|w\|_1$
 - Linear model, square loss, L1 regularization
- Logistic regression:
$$\sum_{i=1}^n [y_i \ln(1 + e^{-w^T x_i}) + (1 - y_i) \ln(1 + e^{w^T x_i})] + \lambda \|w\|^2$$
 - Linear model, logistic loss, L2 regularization
- The conceptual separation between model, parameter, objective also gives you **engineering benefits**.
 - Think of how you can implement SGD for both ridge regression and logistic regression

Objective and Bias Variance Trade-off

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

Training Loss measures how well model fit on training data

Regularization, measures complexity of model

- Why do we want to contain two component in the objective?
- Optimizing training loss encourages **predictive** models
 - Fitting well in training data at least get you close to training data which is hopefully close to the underlying distribution
- Optimizing regularization encourages **simple** models
 - Simpler models tends to have smaller variance in future predictions, making prediction **stable**

Outline

- Review of key concepts of supervised learning

Regression Tree and Ensemble (What are we Learning)

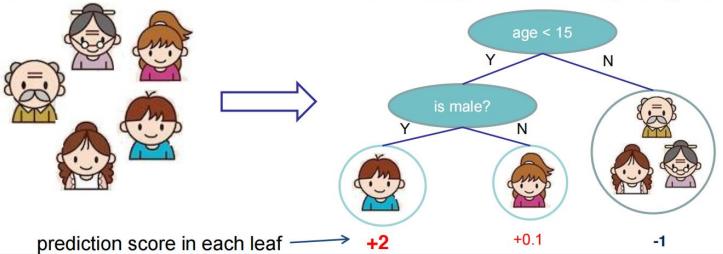
Gradient Boosting (How do we Learn)

Summary

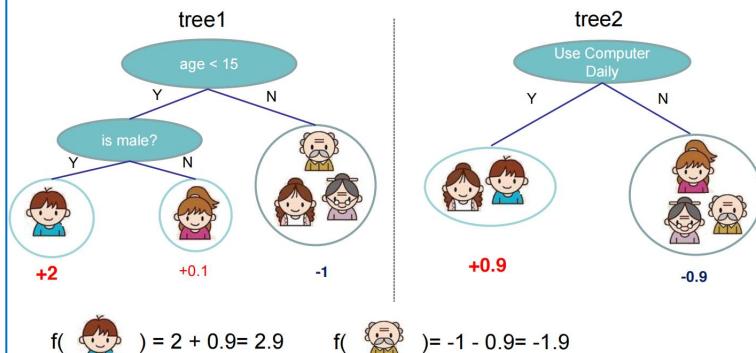
Regression Tree (CART)

- regression tree (also known as classification and regression tree):
 - Decision rules same as in decision tree
 - Contains one score in each leaf value

Input: age, gender, occupation, ... Does the person like computer games



Regression Tree Ensemble



Tree Ensemble methods

- Very widely used, look for GBM, random forest...
 - Almost half of data mining competition are won by using some variants of tree ensemble methods
- Invariant to scaling of inputs, so you do not need to do careful features normalization.
- Learn higher order interaction between features.
- Can be scalable, and are used in Industry

Put into context: Model and Parameters

- Model: assuming we have K trees

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), \quad f_k \in \mathcal{F}$$

Space of functions containing all Regression trees

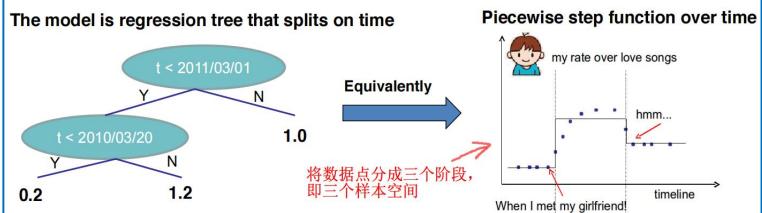
Think: regression tree is a function that maps the attributes to the score

Parameters

- Including structure of each tree, and the score in the leaf
- Or simply use function as parameters
- $\Theta = \{f_1, f_2, \dots, f_K\}$
- Instead learning weights in \mathbb{R}^d , we are learning functions(trees)

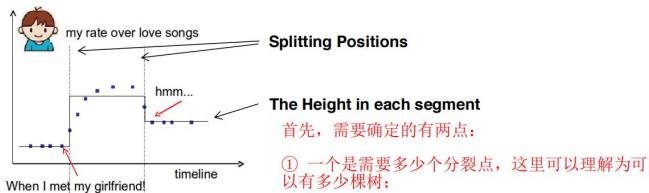
Learning a tree on single variable

- How can we learn functions?
- Define objective (loss, regularization), and optimize it!!
- Example:
 - Consider regression tree on single input t (time)
 - I want to predict whether I like romantic music at time t



Learning a step function

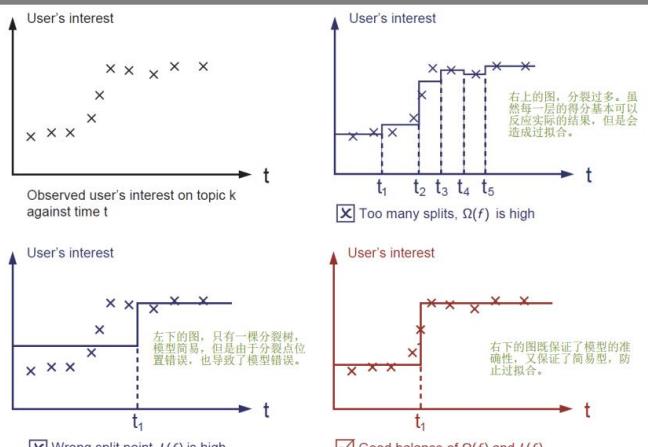
- Things we need to learn



Objective for single variable regression tree(step functions)

- Training Loss: How will the function fit on the points?
- Regularization: How do we define complexity of the function?
 - Number of splitting points, L2 norm of the height in each segment?

Learning step function (visually)



Coming back: Objective for Tree Ensemble

- Model: assuming we have K trees

$$\hat{y}_i = \sum_{k=1}^K f_k(x_i), \quad f_k \in \mathcal{F}$$

- Objective

$$Obj = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k)$$

Training loss Complexity of the Trees

- Possible ways to define Ω ?

- Number of nodes in the tree, depth
- L2 norm of the leaf weights
- ... detailed later

Objective vs Heuristic

- When you talk about (decision) trees, it is usually heuristics

- Split by information gain
- Prune the tree
- Maximum depth
- Smooth the leaf values

- Most heuristics maps well to objectives, taking the formal (objective) view let us know what we are learning

- Information gain -> training loss
- Pruning -> regularization defined by #nodes
- Max depth -> constraint on the function space
- Smoothing leaf values -> L2 regularization on leaf weights

Regression Tree is not just for regression!

- Regression tree ensemble defines how you make the prediction score, it can be used for

- Classification, Regression, Ranking....
-

- It all depends on how you define the objective function!

- So far we have learned:

- Using Square loss $l(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$ 如果用平方差作为损失函数，那就能获得一个回归树，也就是常见的GBM。
 - Will results in common gradient boosted machine
- Using Logistic loss $l(y_i, \hat{y}_i) = y_i \ln(1 + e^{-\hat{y}_i}) + (1 - y_i) \ln(1 + e^{\hat{y}_i})$ 如果选用交叉熵作为损失函数，就能得到 LogitBoost.

Outline

- Review of key concepts of supervised learning
- Regression Tree and Ensemble (What are we Learning)
- Gradient Boosting (How do we Learn)
- Summary

Take Home Message for this section

- Bias-variance tradeoff is everywhere
- The loss + regularization objective pattern applies for regression tree learning (function learning)
- We want **predictive** and **simple** functions
- This defines what we want to learn (objective, model).
- But how do we learn it?
 - Next section

So How do we Learn?

- Objective: $\sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_k \Omega(f_k), f_k \in \mathcal{F}$
- We can not use methods such as SGD, to find f (since they are trees, instead of just numerical vectors)
在这里，我们没办法使用SGD(Stochastic Gradient Decent,随机梯度下降法)。因为SGD常用来寻找的是数值解，需要有已定的目标函数；但是在这里，我们需要在确定是一个函数，而非数值解，因此无法使用。
- Solution: **Additive Training (Boosting)**
加法模型提升
 - Start from constant prediction, add a new function each time
$$\begin{aligned}\hat{y}_i^{(0)} &= 0 \\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\ &\dots \\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i) \leftarrow \text{New function}\end{aligned}$$

Model at training round t Keep functions added in previous round

- 在这里，我们没办法使用SGD(Stochastic Gradient Decent,随机梯度下降法)。因为SGD常用来寻找的是数值解，需要有已定的目标函数；但是在这里，我们需要在确定是一个函数，而非数值解，因此无法使用。
- 解决方案是使用加法模型提升。具体来说，是将每一次的预测值，转换为前一次的预测值加上一个函数 $f_t(x_i)$ 。注意这里的 $f_t(x_i)$ 和上文的 f_k 中的 f_k 不是同一个函数。

这里对GBDT在做多一些解释，方便下文的理解。

- GBDT的思想是累加多个弱学习器，以达到强学习器的效果。
- 每一棵树，就是一个弱分类器。例如大于10岁和小于10岁是一个弱分类器，性别是另一个弱分类器，把二者的得分相加起来，就能得到一个预测值 y 。
- 加法模型是GBDT的常见模型，他的思路是先训练一棵树，即弱分类器，计算弱分类器与实际值的差值。接着，将该差值传到下一个弱分类器中，再计算差值与下一个分类器的差值，以此迭代。如果换个方向讲，就是每一次迭代，预测值都等于了前一次的预测值，加上了这次的树。
- 对于加法模型，正则项也与SGD算法中的L1范式L2范式有区别。在GBDT中，常见的正则项处理方式有三种（摘自百度）：
 1. 第一种，是对步长的调节，即每一轮迭代中对新增的树乘上一个系数 v , v 属于[0,1]。对于同样的训练集学习效果，较小的步长意味着我们需要更多的弱学习器的迭代次数。通常我们用步长和迭代最大次数一起来决定算法的拟合效果。
 2. 第二种正则化的方式是通过子采样比例（subsample）。取值为[0,1]。注意这里的子采样和随机森林不一样，随机森林使用的是放回抽样，而这里是不放回抽样。如果取值为1，则全部样本都使用，等于没有使用子采样。如果取值小于1，则只有一部分样本会去做GBDT的决策树拟合。选择小于1的比例可以减少方差，即防止过拟合，但是会增加样本拟合的偏差，因此取值不能太低。推荐在[0.5, 0.8]之间。
 3. 第三种利用了决策树的剪枝，即正则项为 $\alpha * T$, T 为叶子树，显然 T 无法过大，否则目标函数会过高。

Additive Training

- How do we decide which f to add?
 - Optimize the objective!!
- The prediction at round t is $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$
This is what we need to decide in round t
$$\begin{aligned}Obj^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{k=1}^t \Omega(f_k) \\ &= \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \text{constant}\end{aligned}$$

Goal: find f_t to minimize this
- Consider square loss
$$\begin{aligned}Obj^{(t)} &= \sum_{i=1}^n \left(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i))\right)^2 + \Omega(f_t) + \text{const} \\ &= \sum_{i=1}^n \left[2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2\right] + \Omega(f_t) + \text{const}\end{aligned}$$

This is usually called **residual** from previous round

将加法公式 $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$ 代入到目标函数中，可以将目标函数的第一项——损失函数变为：
$$\sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))$$

其次，对于正则项，在原公式中，是对每一次迭代的残差函数 f_t 求和。但是假如我们使用了上一页中提到的步长控制法，这一项其实仅与本次迭代有关。因此作者这里就将求和符号去掉了，仅留下了本次迭代所增加的决策树 f_t 。

接着，作者举了一个例子，当损失函数为平方差时，即：

$$l(y_i, \hat{y}_i^{(t)}) = (y_i - \hat{y}_i^{(t)})^2 = (y_i - \hat{y}_i^{(t-1)} - f_t(x_i))^2$$

$$l(y_i, \hat{y}_i^{(t)}) = (y_i - \hat{y}_i^{(t-1)})^2 - 2(y_i - \hat{y}_i^{(t-1)})f_t(x_i) + f_t(x_i)^2$$

在这里，由于 y_i 和 $\hat{y}_i^{(t-1)}$ 都是在上一轮算出来的，并且在这个公式中，仅有 f_t 是变量，所以我们将第一项视为一个常数项，搬到最后，则原公式就变成了：

$$l(y_i, \hat{y}_i^{(t)}) = 2(\hat{y}_i^{(t-1)} - y_i)f_t(x_i) + f_t(x_i)^2$$

而 $y_i - \hat{y}_i^{(t-1)}$ 也可以视为上一轮的残差。

Taylor Expansion Approximation of Loss

- Goal $Obj^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant}$
 - Seems still complicated except for the case of square loss

Take Taylor expansion of the objective

- Recall $f(x + \Delta x) \approx f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$
- Define $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$

$$Obj^{(t)} \approx \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + \text{constant}$$

- If you are not comfortable with this, think of square loss

$$g_i = \partial_{\hat{y}^{(t-1)}} (\hat{y}^{(t-1)} - y_i)^2 = 2(\hat{y}^{(t-1)} - y_i) \quad h_i = \partial_{\hat{y}^{(t-1)}}^2 (\hat{y}^{(t-1)} - y_i)^2 = 2$$

- Compare what we get to previous slide



Our New Goal

- Objective, with constants removed

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

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$

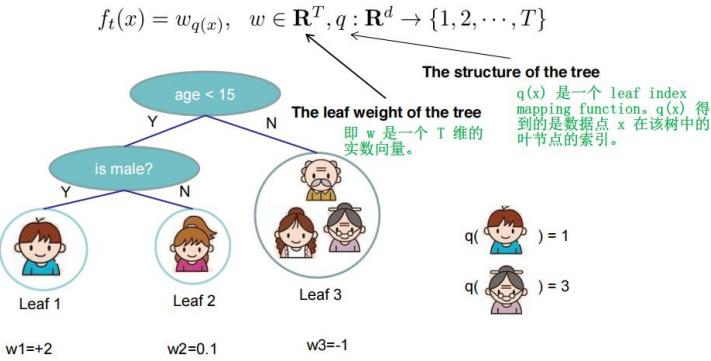
- Why spending so much efforts to derive the objective, why not just grow trees ...

- Theoretical benefit: know what we are learning, convergence
- Engineering benefit, recall the elements of supervised learning
 - g_i and h_i comes from definition of loss function
 - The learning of function only depend on the objective via g_i and h_i
 - Think of how you can separate modules of your code when you are asked to implement boosted tree for both square loss and logistic loss

经过一系列的努力，我们将目标函数简化到了三个变量上： g_i, h_i, f_t 。其中，前两项都能轻易计算得出，并且不随着树的变化而变化。

Refine the definition of tree

- We define tree by a vector of scores in leafs, and a leaf index mapping function that maps an instance to a leaf

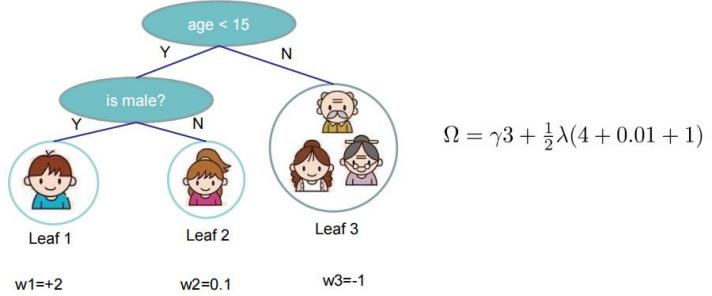


Define Complexity of a Tree (cont')

- Define complexity as (this is not the only possible definition)

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

Number of leaves L2 norm of leaf scores



Revisit the Objectives

- Define the instance set in leaf j as $I_j = \{i | q(x_i) = j\}$

- Regroup the objective by each leaf

$$\begin{aligned} Obj^{(t)} &\approx \sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \\ &= \sum_{i=1}^n \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right] + \gamma T + \lambda \frac{1}{2} \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \end{aligned}$$

- This is sum of T independent quadratic functions

The Structure Score

- Two facts about single variable quadratic function

$$\operatorname{argmin}_x Gx + \frac{1}{2} Hx^2 = -\frac{G}{H}, \quad H > 0 \quad \min_x Gx + \frac{1}{2} Hx^2 = -\frac{1}{2} \frac{G^2}{H}$$

- Let us define $G_j = \sum_{i \in I_j} g_i, H_j = \sum_{i \in I_j} h_i$

$$\begin{aligned} Obj^{(t)} &= \sum_{j=1}^T \left[(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2 \right] + \gamma T \\ &= \sum_{j=1}^T [G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2] + \gamma T \end{aligned}$$

- Assume the structure of tree ($q(x)$) is fixed, the optimal weight in each leaf, and the resulting objective value are

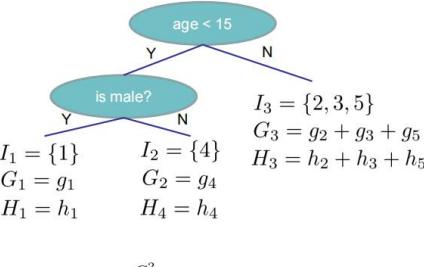
$$w_j^* = -\frac{G_j}{H_j + \lambda} \quad Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

This measures how good a tree structure is!

The Structure Score Calculation

Instance index gradient statistics

1	boy icon	g1, h1
2	girl icon	g2, h2
3	old man icon	g3, h3
4	girl icon	g4, h4
5	girl icon	g5, h5



$$Obj = -\sum_j \frac{G_j^2}{H_j + \lambda} + 3\gamma$$

The smaller the score is, the better the structure is

Searching Algorithm for Single Tree

- Enumerate the possible tree structures q
 - Calculate the structure score for the q , using the scoring eq.

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

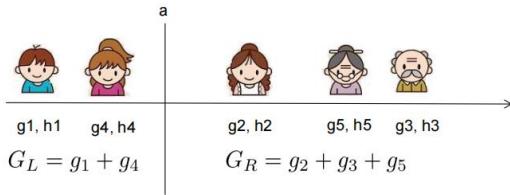
- Find the best tree structure, and use the optimal leaf weight

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

- But... there can be infinite possible tree structures..

Efficient Finding of the Best Split

- What is the gain of a split rule $x_j < a$? Say x_j is age



- All we need is sum of g and h in each side, and calculate

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

- Left to right linear scan over sorted instance is enough to decide the best split along the feature

What about Categorical Variables?

- Some tree learning algorithm handles categorical variable and continuous variable separately
 - We can easily use the scoring formula we derived to score split based on categorical variables.
 - Actually it is not necessary to handle categorical separately.
 - We can encode the categorical variables into numerical vector using **one-hot encoding**. Allocate a **#categorical length vector**
$$z_j = \begin{cases} 1 & \text{if } x \text{ is in category } j \\ 0 & \text{otherwise} \end{cases}$$
 - The vector will be sparse if there are lots of categories, the learning algorithm is preferred to handle **sparse** data

Greedy Learning of the Tree

- In practice, we grow the tree greedily
 - Start from tree with depth 0
 - For each leaf node of the tree, try to add a split. The change of objective after adding the split is ΔJ . The complexity cost is $C(\Delta J)$.

- Remaining question: how do we find the best split?

An Algorithm for Split Finding

- 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 along that feature
 - Take the best split solution along all the features
 - Time Complexity growing a tree of depth K
 - It is $O(n d K \log n)$: or each level, need $O(n \log n)$ time to sort There are d features, and we need to do it for K level
 - This can be further optimized (e.g. use approximation or caching the sorted features)
 - Can scale to very large dataset

Pruning and Regularization

- Recall the gain of split, it can be negative!
$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$
 - When **the training loss reduction** is smaller than **regularization**
 - Trade-off between simplicity and predictivness
 - Pre-stopping
 - Stop split if the best split have negative gain
 - But maybe a split can benefit future splits..
 - Post-Prunning
 - Grow a tree to maximum depth, recursively prune all the leaf splits with negative gain

Recap: Boosted Tree Algorithm

- Add a new tree in each iteration
- Beginning of each iteration, calculate
 $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$
- Use the statistics to greedily grow a tree $f_t(x)$

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

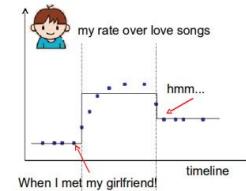
- Add $f_t(x)$ to the model $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$
 - Usually, instead we do $y^{(t)} = y^{(t-1)} + \epsilon f_t(x_i)$
 - ϵ is called **step-size or shrinkage**, usually set around **0.1**
 - This means we do not do full optimization in each step and reserve chance for future rounds, it helps prevent overfitting

Outline

- Review of key concepts of supervised learning
- Regression Tree and Ensemble (What are we Learning)
- Gradient Boosting (How do we Learn)
- Summary

Questions to check if you really get it

- How can we build a boosted tree classifier to do weighted regression problem, such that each instance have a importance weight?
- Back to the time series problem, if I want to learn step functions over time. Is there other ways to learn the time splits, other than the top down split approach?



Questions to check if you really get it

- How can we build a boosted tree classifier to do weighted regression problem, such that each instance have a importance weight?
 - Define objective, calculate g_i, h_i , feed it to the old tree learning algorithm we have for un-weighted version
- $$l(y_i, \hat{y}_i) = \frac{1}{2} a_i (\hat{y}_i - y_i)^2 \quad g_i = a_i (\hat{y}_i - y_i) \quad h_i = a_i$$
- Again think of separation of model and objective, how does the theory can help better organizing the machine learning toolkit

Questions to check if you really get it

- Time series problem
- All that is important is the structure score of the splits

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$
 - Top-down greedy, same as trees
 - Bottom-up greedy, start from individual points as each group, greedily merge neighbors
 - Dynamic programming, can find optimal solution for this case

Summary

- The separation between model, objective, parameters can be helpful for us to understand and customize learning models
- The bias-variance trade-off applies everywhere, including learning in functional space

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

- We can be formal about what we learn and how we learn. Clear understanding of theory can be used to guide cleaner implementation.

Reference

- Greedy function approximation a gradient boosting machine. *J.H. Friedman*
 - *First paper about gradient boosting*
- *Stochastic Gradient Boosting*. *J.H. Friedman*
 - *Introducing bagging trick to gradient boosting*
- *Elements of Statistical Learning*. *T. Hastie, R. Tibshirani and J.H. Friedman*
 - *Contains a chapter about gradient boosted boosting*
- Additive logistic regression a statistical view of boosting. *J.H. Friedman T. Hastie R. Tibshirani*
 - *Uses second-order statistics for tree splitting, which is closer to the view presented in this slide*
- Learning Nonlinear Functions Using Regularized Greedy Forest. *R. Johnson and T. Zhang*
 - *Proposes to do fully corrective step, as well as regularizing the tree complexity. The regularizing trick is closed related to the view present in this slide*
- Software implementing the model described in this slide: <https://github.com/tqchen/xgboost>

XGBoost

参考文献:

1. 终于有人说清楚了--XGBoost算法:

- 文章: <https://www.cnblogs.com/mantch/p/11164221.html>
- 代码: <https://github.com/NLP-LOVE/ML-NLP/blob/master/Machine%20Learning/3.3%20XGBoost/3.3%20XGBoost.ipynb>

说到 XGBoost, 不得不提 GBDT(Gradient Boosting Decision Tree)。因为 XGBoost 本质上还是一个 GBDT, 但是力争把速度和效率发挥到极致, 所以叫 X (Extreme) GBoosted。包括前面说过, 两者都是 boosting 方法。

1.1 XGBoost树的定义

..... (举例到这里) 恩, 你可能要拍案而起了, 惊呼, 这不是跟上文介绍的GBDT乃异曲同工么?

事实上, 如果不考虑工程实现、解决问题上的一些差异, XGBoost 与 GBDT 比较大的不同就是目标函数的定义。XGBoost的目标函数如下图所示:

- 目标 $Obj^{(t)} = \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + constant$
 - 用泰勒展开来近似我们原来的目标
 - 泰勒展开: $f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$
 - 定义: $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$
- $$Obj^{(t)} \simeq \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + constant$$
- 

其中:

- 红色箭头所指向的L 即为损失函数 (比如平方损失函数: $l(y_i, y^i) = (y_i - y^i)^2$)
- 红色方框所框起来的是正则项 (包括L1正则、L2正则)
- 红色圆圈所圈起来的为常数项
- 对于f(x), XGBoost利用泰勒展开三项, 做一个近似。f(x)表示的是其中一颗回归树。

泰勒展开三项, 即只到二阶导, 而不是到三阶导。

看到这里可能有些读者会头晕了, 这么多公式, 这里只做一个简要式的讲解, 具体的算法细节和公式求解请查看这篇博文, 讲得很仔细:

通俗理解kaggle比赛大杀器xgboost: https://blog.csdn.net/v_JULY_v/article/details/81410574

总结:

XGBoost的核心算法思想不难，基本就是：

- 不断地添加树，不断地进行特征分裂来生长一棵树，每次添加一个树，其实是学习一个新函数 $f(x)$ ，去拟合上次预测的残差。
- 当我们训练完成得到 k 棵树，我们要预测一个样本的分数，其实就是根据这个样本的特征，在每棵树中会落到对应的一个叶子节点，每个叶子节点就对应一个分数。
- 最后只需要将每棵树对应的分数加起来就是该样本的预测值。

显然，我们的目标是要使得树群的预测值 \hat{y}_i 尽量接近真实值 y_i ，而且有尽量大的泛化能力。类似之前 GBDT 的套路，XGBoost 也是需要将多棵树的得分累加得到最终的预测得分（每一次迭代，都在现有树的基础上，增加一棵树去拟合前面树的预测结果与真实值之间的残差）。

- Start from constant prediction, add a new function each time

$$\begin{aligned}\hat{y}_i^{(0)} &= 0 \\ \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\ \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\ &\dots \\ \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)\end{aligned}$$

← New function

Model at training round t **Keep functions added in previous round**

那接下来，我们如何选择每一轮加入一个什么样的 f 呢？答案是非常直接的，选取一个 f 来使得我们的目标函数尽量最大地降低。这里 f 可以使用泰勒展开公式近似。

$$\begin{aligned}Obj^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \text{constant}\end{aligned}$$

Goal: find f_t to minimize this

到目前为止我们讨论了目标函数中的第一个部分：训练误差。

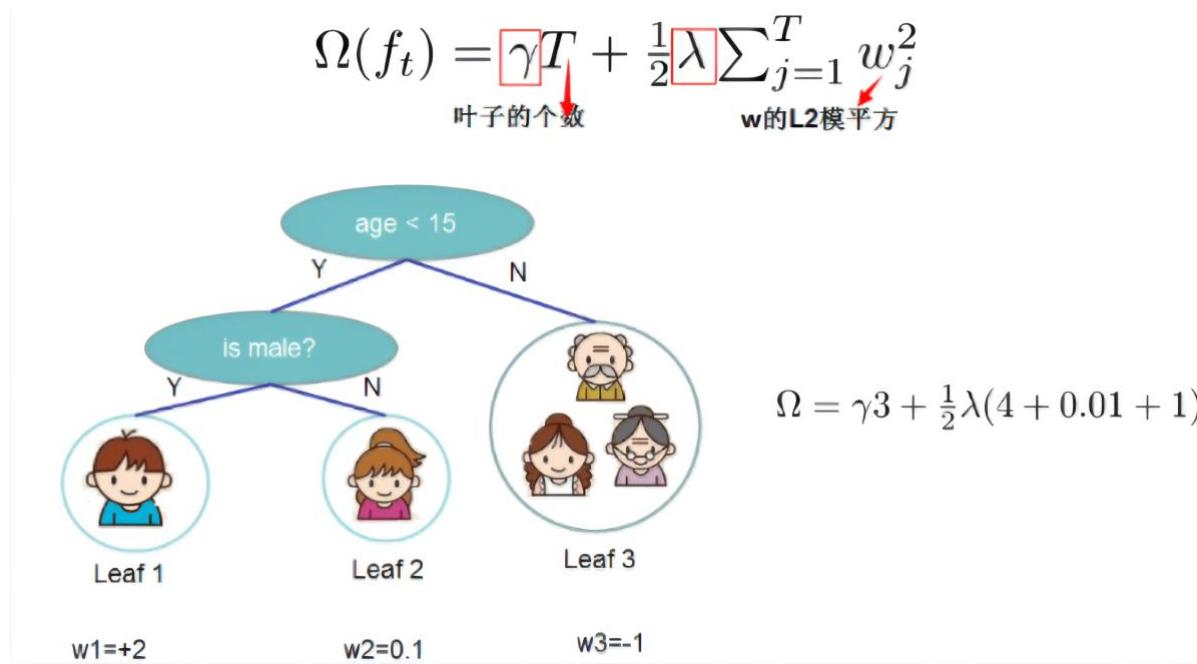
接下来我们讨论目标函数的第二个部分：正则项，即如何定义树的复杂度。

总结：

1.2 正则项

XGBoost对树的复杂度包含了两个部分：

- 一个是树里面叶子节点的个数 T 。
- 一个是树上叶子节点的得分 w 的 L2 模平方（对 w 进行 L2 正则化，相当于针对每个叶结点的得分增加 L2 平滑，目的是为了避免过拟合）。



我们再来看一下XGBoost的目标函数（损失函数揭示训练误差 + 正则化定义复杂度）：

$$L(\phi) = \sum_i l(y'_i - y_i) + \sum_k \Omega(f_t)$$

正则化公式也就是目标函数的后半部分，对于上式而言， y'_i 是整个累加模型的输出，正则化项 $\sum_k \Omega(f_t)$ 是表示树的复杂度的函数，值越小复杂度越低，泛化能力越强。

1.3 树该怎么长

很有意思的一个事是，我们从头到尾了解了xgboost 如何优化、如何计算，但树到底长啥样，我们却一直没看到。很显然，一棵树的生成是由一个节点一分为二，然后不断分裂最终形成为整棵树。那么树怎么分裂的就成为了接下来我们要探讨的关键。对于一个叶子节点如何进行分裂，XGBoost 作者在其原始论文中给出了一种分裂节点的方法：**枚举所有不同树结构的贪心法**。

不断地枚举不同树的结构，然后利用**打分函数**来寻找出一个最优结构的树，接着加入到模型中，不断重复这样的操作。这个寻找的过程使用的就是**贪心算法**。选择一个 feature 分裂，计算 loss function 最小值，然后再选一个 feature 分裂，又得到一个 loss function 最小值，枚举完，找一个效果最好的，把树给分裂，就得到了小树苗。

总而言之，XGBoost 使用了和 CART回归树一样的想法，利用**贪婪算法**，遍历所有特征的所有特征划分点，不同的是**使用的目标函数不一样**。

具体做法就是分裂后的目标函数值比单子叶子节点的目标函数的**增益**，同时为了限制树生长过深，还加了个**阈值**，只有当增益大于该阈值才进行分裂。从而继续分裂，形成一棵树，再形成一棵树，每次在上一次的预测基础上取最优进一步分裂/建树。

总结：

1.4 如何停止树的循环生成

凡是这种循环迭代的方式必定有停止条件，什么时候停止呢？简言之，**设置树的最大深度、当样本权重和小于设定阈值时停止生长以防止过拟合**。具体而言，则：

- 当引入的分裂带来的增益小于**设定阈值**的时候，我们可以忽略掉这个分裂，所以并不是每一次分裂 loss function，整体都会增加的，有点预剪枝的意思，**阈值参数为（即正则项里叶子节点数T的系数）**；
- 当树达到**最大深度**时则停止建立决策树，设置一个超参数 `max_depth`，避免树太深导致学习局部样本而过拟合；
- **样本权重和小于设定阈值**时则停止建树。即涉及到一个超参数——最小的样本权重和 `min_child_weight`，和 GBM 的 `min_child_leaf` 参数类似，但不完全一样。大意就是一个叶子节点样本太少了，也终止同样是防止过拟合。

2. XGBoost与GBDT有什么不同

除了算法上与传统的GBDT有一些不同外，XGBoost 还在工程实现上做了大量的优化。总的来说，两者之间的区别和联系可以总结成以下几个方面。

- GBDT是机器学习算法，XGBoost是该算法的工程实现。
- 在使用CART作为基分类器时，XGBoost 显式地加入了正则项来控制模型的复杂度，有利于防止过拟合，从而提高模型的泛化能力。
- GBDT在模型训练时只使用了代价函数的一阶导数信息，XGBoost对代价函数进行二阶泰勒展开，可以同时使用一阶和二阶导数。
- 传统的GBDT采用CART作为基分类器，XGBoost支持多种类型的基分类器，比如线性分类器。
- 传统的GBDT在每轮迭代时使用全部的数据，XGBoost则采用了与随机森林相似的策略，支持对数据进行采样。
- 传统的GBDT没有设计对缺失值进行处理，XGBoost能够自动学习出缺失值的处理策略。

3. 为什么XGBoost要用泰勒展开，优势在哪里？

XGBoost使用了一阶和二阶偏导，二阶导数有利于梯度下降的更快更准。

使用泰勒展开取得函数做自变量的二阶导数形式，可以在不选定损失函数具体形式的情况下，仅仅依靠输入数据的值就可以进行叶子分裂优化计算，本质上也就把损失函数的选取和模型算法优化/参数选择分开了。

这种去耦合增加了XGBoost的适用性，使得它按需选取损失函数，可以用于分类，也可以用于回归。

参考文献：

1. 通俗理解kaggle比赛大杀器xgboost: https://blog.csdn.net/v_JULY_v/article/details/81410574

1 决策树

举个例子，集训营某一期有100多名学员，假定给你一个任务，要你统计男生女生各多少人，当一个一个学员依次上台站到你面前时，你会怎么区分谁是男谁是女呢？

很快，你考虑到男生的头发一般很短，女生的头发一般比较长，所以你通过头发的长短将这个班的所有学员分为两拨，长发的为“女”，短发为“男”。

相当于你依靠一个指标“头发长短”将整个班的人进行了划分，于是形成了一个简单的决策树，而划分的依据是头发长短。这时，有的人可能有不同意见了：为什么要用“头发长短”划分呀，我可不可以用“穿的鞋子是否是高跟鞋”，“有没有喉结”等等这些来划分呢，答案当然是可以的。

但究竟根据哪个指标划分更好呢？很直接的判断是哪个分类效果更好则优先用哪个。所以，这时就需要一个评价标准来量化分类效果了。

怎么判断“头发长短”或者“是否有喉结”是最好的划分方式，效果怎么量化呢？直观上来说，如果根据某个标准分类人群后，纯度越高效果越好，比如说你分为两群，“女”那一群都是女的，“男”那一群全是男的，那这个效果是最好的。但有时实际的分类情况不是那么理想，所以只能说越接近这种情况，我们认为效果越好。

量化分类效果的方式有很多，比如**信息增益 (ID3)**、**信息增益率 (C4.5)**、**基尼系数 (CART)**等等。

信息增益的度量标准：熵

ID3算法的核心思想就是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。

什么是信息增益呢？为了精确地定义信息增益，我们先定义信息论中广泛使用的一个度量标准，称为**熵** (entropy)，它刻画了任意样例集的纯度 (purity)。给定包含关于某个目标概念的正反样例的样例集S，那么S相对这个布尔型分类的熵为：

$$\text{Entropy}(S) \equiv -p_+ \log_2 p_+ - p_- \log_2 p_-$$

上述公式中， p_+ 代表正样例，比如在本文开头第二个例子中 p_+ 则意味着去打羽毛球，而 p_- 则代表反样例，不去打球(在有关熵的所有计算中我们定义 $0\log_0$ 为0)。

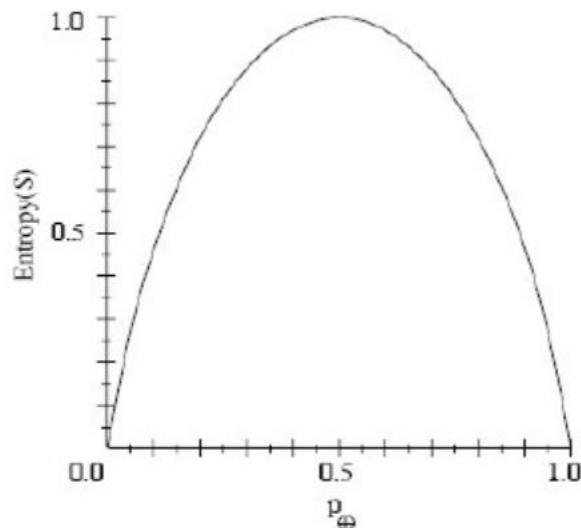
举例来说，假设S是一个关于布尔概念的有14个样例的集合，它包括9个正例和5个反例（我们采用记号[9+, 5-]来概括这样的数据样例），那么S相对于这个布尔样例的熵为：

$$\text{Entropy} ([9+, 5-]) = - (9/14) \log_2 (9/14) - (5/14) \log_2 (5/14) = 0.940.$$

So，根据上述这个公式，我们可以得到：

- 如果S的所有成员属于同一类，则 $\text{Entropy}(S)=0$ ；
- 如果S的正反样例数量相等，则 $\text{Entropy}(S)=1$ ；
- 如果S的正反样例数量不等，则熵介于0, 1之间

如下图所示：



看到没，通过 Entropy 的值，就能评估当前分类树的分类效果好坏。

更多细节如剪枝、过拟合、优缺点、可以参考此文《决策树学习》：https://blog.csdn.net/v_july_v/article/details/7577684

所以，现在决策树的灵魂已经有了，即依靠某种指标进行树的分裂达到分类/回归的目的，总是希望纯度越高越好。

2. 回归树与集成学习

如果用一句话定义 Xgboost，很简单：Xgboost 就是由很多 CART树 集成。但，什么是 CART树？

数据挖掘或机器学习中使用的决策树有两种主要类型：

1. 分类树分析是指预测结果是数据所属的类（比如某个电影去看还是不看）
2. 回归树分析是指预测结果可以被认为是实数（例如房屋的价格，或患者在医院中的逗留时间）

而术语分类回归树（CART，Classification And Regression Tree）分析是用于指代上述两种树的总称，由Breiman等人首先提出。

2.1 回归树

事实上，分类与回归是两个很接近的问题：

- 分类的目标是根据已知样本的某些特征，判断一个新的样本属于哪种已知的样本类，它的结果是离散值。
- 回归的结果是连续的值。
- 当然，本质是一样的，都是特征（feature）到结果/标签（label）之间的映射。

理清了什么是分类和回归之后，理解分类树和回归树就不难了。

分类树的样本输出（即响应值）是类的形式，比如判断这个救命药是真的还是假的，周末去看电影《风语咒》还是不去。而回归树的样本输出是数值的形式，比如给某人发放房屋贷款的数额就是具体的数值，可以是0到300万元之间的任意值。

所以，对于回归树，你没法再用分类树那套信息增益、信息增益率、基尼系数来判定树的节点分裂了，你需要采取新的方式评估效果，包括预测误差（常用的有均方误差、对数误差等）。而且节点不再是类别，是数值（预测值）。

那么怎么确定呢？有的是节点内样本均值，有的是最优化算出来的比如 Xgboost。

CART回归树是假设树为二叉树，通过不断将特征进行分裂。比如当前树结点是基于第j个特征值进行分裂的，设该特征值小于s的样本划分为左子树，大于s的样本划分为右子树。

$$R_1(j, s) = \{x | x^{(j)} \leq s\} \text{ and } R_2(j, s) = \{x | x^{(j)} > s\}$$

而CART回归树实质上就是在该特征维度对样本空间进行划分，而这种空间划分的优化是一种NP难问题，因此，在决策树模型中是使用启发式方法解决。典型CART回归树产生的目标函数为：

$$\sum_{x_i \in R_m} (y_i - f(x_i))^2$$

因此，当我们为了求解最优的切分特征j和最优的切分点s，就转化为求解这么一个目标函数：

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right]$$

所以我们只要遍历所有特征的所有切分点，就能找到最优的切分特征和切分点。最终得到一棵回归树。

2.2 boosting 集成学习

所谓集成学习，是指构建多个分类器（弱分类器）对数据集进行预测，然后用某种策略将多个分类器预测的结果集成起来，作为最终预测结果。通俗比喻就是“三个臭皮匠赛过诸葛亮”，或一个公司董事会上的各董事投票决策，**它要求每个弱分类器具备一定的“准确性”，分类器之间具备“差异性”。**

集成学习根据各个弱分类器之间有无依赖关系，分为 **Boosting** 和 **Bagging** 两大流派：

- Boosting流派，各分类器之间有依赖关系，必须串行，比如Adaboost、GBDT、Xgboost。
- Bagging流派，各分类器之间没有依赖关系，可各自并行，比如随机森林（Random Forest）。

而著名的Adaboost作为boosting流派中最具代表性的一种方法，本博客曾详细介绍它。

AdaBoost，是英文"Adaptive Boosting"（自适应增强）的缩写，由Yoav Freund和Robert Schapire在1995年提出。它的自适应在于：前一个基本分类器分错的样本会得到加强，加权后的全体样本再次被用来训练下一个基本分类器。同时，在每一轮中加入一个新的弱分类器，直到达到某个预定的足够小的错误率或达到预先指定的最大迭代次数。

具体说来，整个Adaboost迭代算法就3步：

1. 初始化训练数据的权值分布。如果有N个样本，则每一个训练样本最开始时都被赋予相同的权值： $1/N$ 。
2. 训练弱分类器。具体训练过程中，如果某个样本点已经被准确地分类，那么在构造下一个训练集中，它的权值就被降低；相反，如果某个样本点没有被准确地分类，那么它的权值就得到提高。然后，权值更新过的样本集被用于训练下一个分类器，整个训练过程如此迭代地进行下去。
3. 将各个训练得到的弱分类器组合成强分类器。各个弱分类器的训练过程结束后，加大分类误差率小的弱分类器的权重，使其在最终的分类函数中起着较大的决定作用，而降低分类误差率大的弱分类器的权重，使其在最终的分类函数中起着较小的决定作用。**换言之，误差率低的弱分类器在最终分类器中占的权重较大，否则较小。**

而另一种 boosting 方法 GBDT（Gradient Boost Decision Tree），则与 AdaBoost 不同，**GBDT 每一次的计算都是为了减少上一次的残差，进而在残差减少（负梯度）的方向上建立一个新的模型。**

boosting集成学习由多个相关联的决策树联合决策，什么叫相关联？举个例子

1. 有一个样本[数据->标签]是：[(2, 4, 5)-> 4]
2. 第一棵决策树用这个样本训练的预测为3.3
3. 那么第二棵决策树训练时的输入，这个样本就变成了：[(2, 4, 5)-> 0.7]
4. 也就是说，下一棵决策树输入样本会与前面决策树的训练和预测相关

很快你会意识到，Xgboost为何也是一个boosting的集成学习了。

总结：

而一个回归树形成的关键点在于：

- 分裂点依据什么来划分（如前面说的均方误差最小，loss）；
- 分类后的节点预测值是多少（如前面说，有一种是将叶子节点下各样本实际值得均值作为叶子节点预测误差，或者计算所得）

至于另一类集成学习方法 bagging，比如 Random Forest（随机森林）算法，各个决策树是独立的、每个决策树在样本堆里随机选一批样本，随机选一批特征进行独立训练，各个决策树之间没有啥关系。

3. GBDT

(详见原文，这里仅摘录一小部分。)

说到 Xgboost，不得不先从 GBDT 说起。因为 xgboost 本质上还是一个 GBDT，但是力争把速度和效率发挥到极致，所以叫 X (Extreme) GBoosted。

注意，为何gbdt可以用用负梯度近似残差呢？

回归任务下，GBDT 在每一轮的迭代时对每个样本都会有一个预测值，此时的损失函数为均方差损失函数，

$$l(y_i, y^i) = \frac{1}{2}(y_i - y^i)^2$$

那此时的负梯度是这样计算的

$$-\left[\frac{\partial l(y_i, y^i)}{\partial y^i}\right] = (y_i - y^i)$$

所以，当损失函数选用均方损失函数时，每一次拟合的值就是（真实值 - 当前模型预测的值），即残差。此时的变量是 y^i ，即“当前预测模型的值”，也就是对它求负梯度。

残差在数理统计中是指实际观察值与估计值（拟合值）之间的差。“残差”蕴含了有关模型基本假设的重要信息。如果回归模型正确的话，我们可以将残差看作误差的观测值。

4. Xgboost

4.1 Xgboost树的定义

(具体详见原文，这里仅摘录关键部分)

事实上，如果不考虑工程实现、解决问题上的一些差异，xgboost与gbdt比较大的不同就是目标函数的定义。Xgboost的目标函数如下图所示：

- 目标 $Obj^{(t)} = \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant$

- 用泰勒展开来近似我们原来的目标

- 泰勒展开： $f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$
- 定义： $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$

$$Obj^{(t)} \simeq \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + constant$$

其中

- 红色箭头所指向的L即为损失函数（比如平方损失函数： $l(y_i, y^i) = (y_i - y^i)^2$ ，或logistic损失函数： $l(y_i, \hat{y}_i) = y_i \ln(1/(1+e^{-\hat{y}_i})) + (1-y_i) \ln(1/(1+e^{\hat{y}_i}))$ ）
- 红色方框所框起来的是正则项（包括L1正则、L2正则）
- 红色圆圈所圈起来的为常数项
- 对于 $f(x)$ ，xgboost利用泰勒展开三项，做一个近似

我们可以很清晰地看到，最终的目标函数只依赖于每个数据点在误差函数上的一阶导数和二阶导数。

4.2 Xgboost目标函数

显然，我们的目标是要使得树群的预测值 \hat{y}_i 尽量接近真实值 y_i ，而且有尽量大的泛化能力。

所以，从数学角度看这是一个泛函最优化问题，故把目标函数简化如下：

$$L(\phi) = \sum_i l(\hat{y}_i - y_i) + \sum_k \Omega(f_k)$$

如你所见，这个目标函数分为两部分：损失函数和正则化项。且损失函数揭示训练误差（即预测分数和真实分数的差距），正则化定义复杂度。

对于上式而言， \hat{y}_i 是整个累加模型的输出，正则化项 $\sum_k \Omega(f_k)$ 是表示树的复杂度的函数，值越小复杂度越低，泛化能力越强，其表达式为

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

T 表示叶子节点的个数， w 表示叶子节点的分数。直观上看，目标要求预测误差尽量小，且叶子节点 T 尽量少（ γ 控制叶子结点的个数），节点数值 w 尽量不极端（ λ 控制叶子节点的分数不会过大），防止过拟合。

插一句，一般的目标函数都包含下面两项

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

误差函数：我们的模型有多拟合数据。

正则化项：惩罚复杂模型

https://blog.csdn.net/v_JULY_v

其中，误差/损失函数鼓励我们的模型尽量去拟合训练数据，使得最后的模型会有比较少的 bias。而正则化项则鼓励更加简单的模型。因为当模型简单之后，有限数据拟合出来结果的随机性比较小，不容易过拟合，使得最后模型的预测更加稳定。

4.2.1 模型学习与训练误差

具体来说，目标函数第一部分中的 i 表示第 i 个样本， $l(\hat{y}_i - y_i)$ 表示第 i 个样本的预测误差，我们的目标当然是误差越小越好。

类似之前GBDT的套路，xgboost也是需要将多棵树的得分累加得到最终的预测得分（每一次迭代，都在现有树的基础上，增加一棵树去拟合前面树的预测结果与真实值之间的残差）。

- Start from constant prediction, add a new function each time

$$\begin{aligned}
 \hat{y}_i^{(0)} &= 0 \\
 \hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
 \hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
 &\dots \\
 \hat{y}_i^{(t)} &= \sum_{k=1}^t f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
 \end{aligned}$$

← New function

Model at training round t

Keep functions added in previous round

总结：

但，我们如何选择每一轮加入什么 f 呢？答案是非常直接的，选取一个 f 来使得我们的目标函数尽量最大地降低。

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^t \Omega(f_i) \\ &= \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \text{constant} \end{aligned}$$

https://blog.csdn.net/v_JULY_v

再强调一下，考虑到第t轮的模型预测值 $\hat{y}_i^{(t)}$ = 前t-1轮的模型预测 $\hat{y}_i^{(t-1)}$ + $f_t(x_i)$ ，因此误差函数记为： $l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))$ ，后面一项为正则化项。

对于这个误差函数的式子而言，在第t步， y_i 是真实值，即已知， $\hat{y}_i^{(t-1)}$ 可由上一步第t-1步中的 y_i^{t-2} 加上 $f_{t-1}(x_i)$ 计算所得，某种意义上也算已知值，故模型学习的是 f 。

上面那个Obj的公式表达的可能有些过于抽象，我们可以考虑当 l 是平方误差的情况（相当于 $l(y_i, y^i) = (y_i - y^i)^2$ ），这个时候我们的目标可以被写成下面这样的二次函数（图中画圈的部分表示的就是预测值和真实值之间的残差）：

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n \left(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)) \right)^2 + \Omega(f_t) + \text{const} \\ &= \sum_{i=1}^n \left[2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2 \right] + \Omega(f_t) + \text{const} \end{aligned}$$

https://blog.csdn.net/v_JULY_v

更加一般的，损失函数不是二次函数咋办？利用泰勒展开，不是二次的想办法近似为二次（如你所见，定义了一阶导 g 和二阶导 h ）。

- Goal $Obj^{(t)} = \sum_{i=1}^n l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + \text{constant}$

- Seems still complicated except for the case of square loss

- Take Taylor expansion of the objective

- Recall $f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$
- Define $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$, $h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$



$$Obj^{(t)} \simeq \sum_{i=1}^n \left[l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) + \text{constant}$$

- If you are not comfortable with this, think of square loss

$$g_i = \partial_{\hat{y}^{(t-1)}} (\hat{y}^{(t-1)} - y_i)^2 = 2(\hat{y}^{(t-1)} - y_i) \quad h_i = \partial_{\hat{y}^{(t-1)}}^2 (\hat{y}^{(t-1)} - y_i)^2 = 2$$

呜呼，透了！不过，这个转化过程中的关键泰勒二次展开到底是哪来的呢？

在数学中，泰勒公式（英语：Taylor's Formula）是一个用函数在某点的信息描述其附近取值的公式。这个公式来自于微积分的泰勒定理（Taylor's theorem），泰勒定理描述了一个可微函数，如果函数足够光滑的话，在已知函数在某一点的各阶导数值的情况下，泰勒公式可以用这些导数值做系数构建一个多项式来近似函数在这一点的邻域中的值，这个多项式称为泰勒多项式（Taylor polynomial）。

相当于告诉我们可由利用泰勒多项式的某些次项做原函数的近似。

泰勒定理：

设 n 是一个正整数。如果定义在一个包含 a 的区间上的函数 f 在 a 点处 $n+1$ 次可导，那么对于这个区间上的任意 x ，都有：

$$f(x) = f(a) + \frac{f'(a)}{1!}(x - a) + \frac{f^{(2)}(a)}{2!}(x - a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x - a)^n + R_n(x)$$

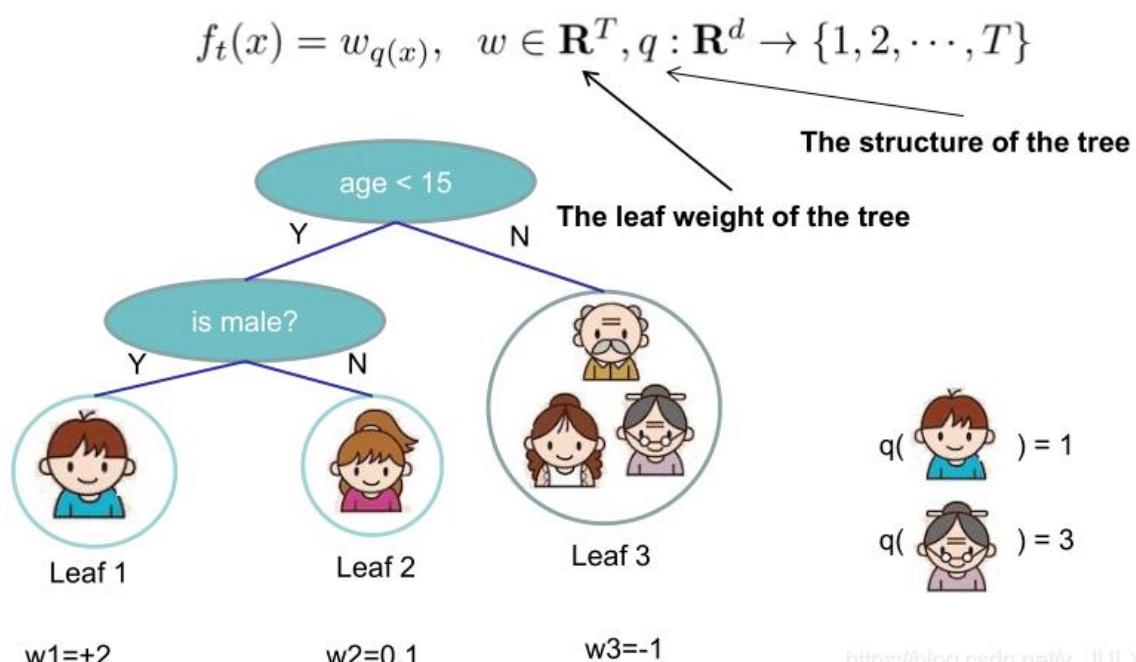
其中的多项式称为函数在 a 处的泰勒展开式，剩余的 $R_n(x)$ 是泰勒公式的余项，是 $(x - a)^n$ 的高阶无穷小。

4.2.2 正则项：树的复杂度

首先，梳理下几个规则：

- 用叶子节点集合以及叶子节点得分表示。
- 每个样本都落在一个叶子节点上。
- $q(x)$ 表示样本 x 在某个叶子节点上， $wq(x)$ 是该节点的打分，即该样本的模型预测值。

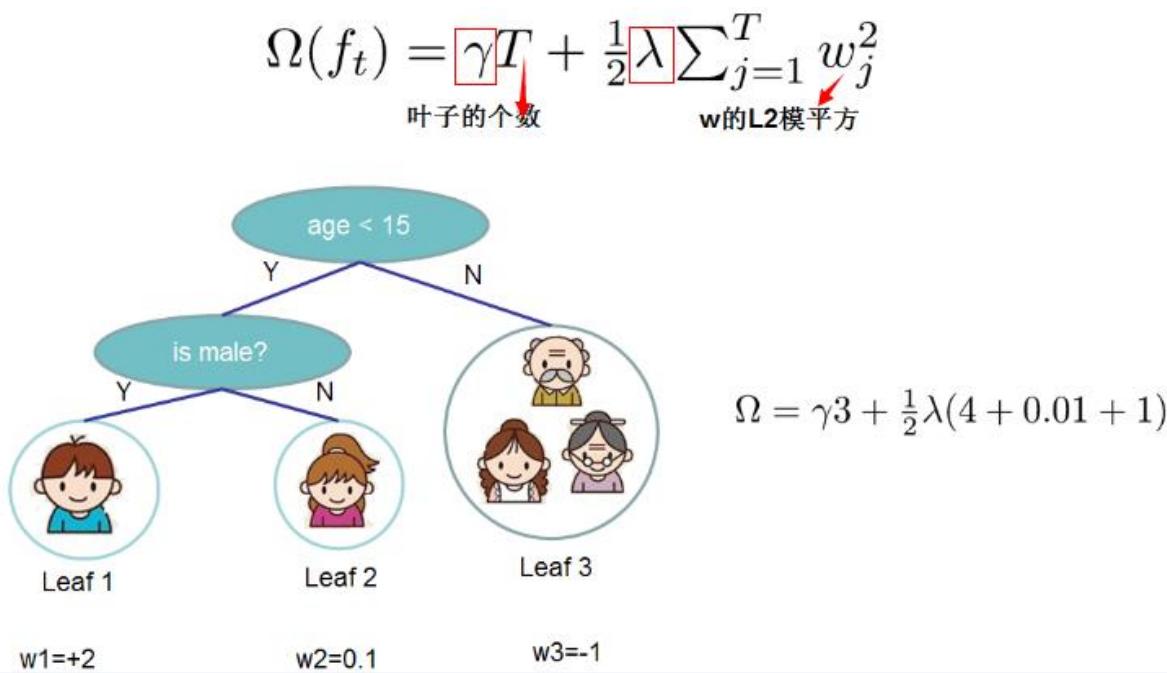
所以当我们把树分成 **结构部分 q** 和 **叶子权重部分 w** 后，结构函数 q 把输入映射到叶子的索引号上面去，而 w 给定了每个索引号对应的叶子分数是什么。



总结：

另外，如下图所示，Xgboost 对树的复杂度包含了两个部分：

- 一个是树里面叶子节点的个数 T。
- 一个是树上叶子节点的得分 w 的 L2 模平方（对 w 进行 L2 正则化，相当于针对每个叶结点的得分增加 L2 平滑，目的是为了避免过拟合）。



在这种新的定义下，我们可以把之前的目标函数进行如下变形（另，别忘了： $\Omega(f) = \gamma T + \frac{1}{2} \lambda ||w||^2$ ）

$$\begin{aligned} Obj^{(t)} &\simeq \sum_{i=1}^n \left[g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i) \right] + \Omega(f_t) \\ &= \sum_{i=1}^n \left[g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2 \right] + \gamma T + \lambda \frac{1}{2} \sum_{j=1}^T w_j^2 \\ &= \sum_{j=1}^T \left[\left(\sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left(\sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \end{aligned}$$

其中 I_j 被定义为每个叶节点 j 上面样本下标的集合 $I_j = \{i | q(x_i) = j\}$, g 是一阶导数, h 是二阶导数。这一步是由于 xgboost 目标函数第二部分加了两个正则项，一个是叶子节点个数(T), 一个是叶子节点的分数(w)。

从而，加了正则项的目标函数里就出现了两种累加

- 一种是 $i \rightarrow n$ (样本数)
- 一种是 $j \rightarrow T$ (叶子节点数)

这一个目标包含了 T 个相互独立的单变量二次函数。

理解这个推导的关键在哪呢？在和 AI lab 陈博士讨论之后，其实就在于理解这个定义： I_j 被定义为每个叶节点 j 上面样本下标的集合 $I_j = \{i | q(x_i) = j\}$ ，这个定义里的 $q(x_i)$ 要表达的是：每个样本值 x_i 都能通过函数 $q(x_i)$ 映射到树上的某个叶子节点，从而通过这个定义把两种累加统一到了一起。

总结：

$$G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$$

最终公式可以化简为

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

通过对 w_j 求导等于0，可以得到

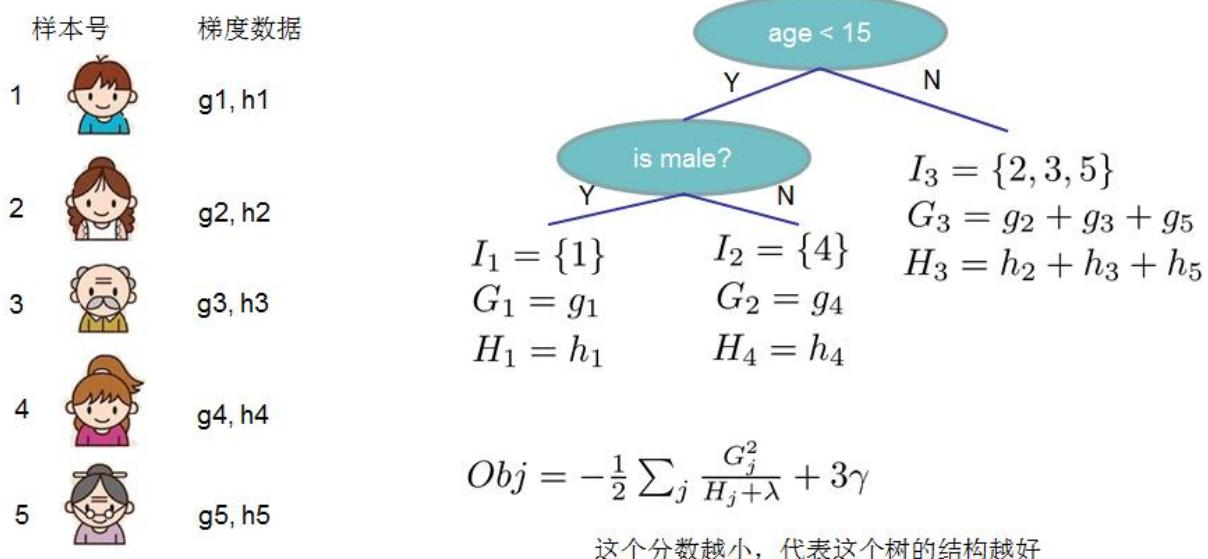
$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

然后把 w_j 最优解代入得到：

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

4.3 打分函数计算

(上式) Obj 代表了当我们指定一个树的结构的时候，在目标上面最多减少多少。可以把它叫做**结构分数**(structure score)。



4.3.1 分裂节点

我们从头到尾了解了 Xgboost 如何优化、如何计算，但树到底长啥样，我们却一直没看到。很显然，一棵树的生成是由一个节点一分为二，然后不断分裂最终形成为整棵树。那么树怎么分裂的就成为了接下来我们要探讨的关键。

对于一个叶子节点如何进行分裂，xgboost作者在其原始论文中给出了两种分裂节点的方法

- 枚举所有不同树结构的贪心法
 - 近似算法

(1) 枚举所有不同树结构的贪心法

现在的情况是只要知道树的结构，就能得到一个该结构下的最好分数，那如何确定树的结构呢？

一个想当然的方法是：不断地枚举不同树的结构，然后利用打分函数来寻找出一个最优结构的树，接着加入到模型中，不断重复这样的操作。而这样要枚举的状态太多了，基本属于无穷种，那咋办呢？

我们试下贪心法，从树深度 0 开始，每一节点都遍历所有的特征，比如年龄、性别等等，然后对于某个特征，先按照该特征里的值进行排序，然后线性扫描该特征进而确定最好的分割点，最后对所有特征进行分割后，我们选择所谓的增益Gain最高的那个特征，而 Gain 如何计算呢？

还记得 4.2 节最后，得到的计算式子吧？

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_{j+\lambda}} + \gamma T$$

换句话说，目标函数中的 $G/(H+\lambda)$ 部分，表示着每一个叶子节点对当前模型损失的贡献程度，融合一下，得到 Gain 的计算表达式，如下所示：

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

加入新叶子节点引入的复杂度代价

第一个值得注意的事情是“对于某个特征，先按照该特征里的值进行排序”，这里举个例子。

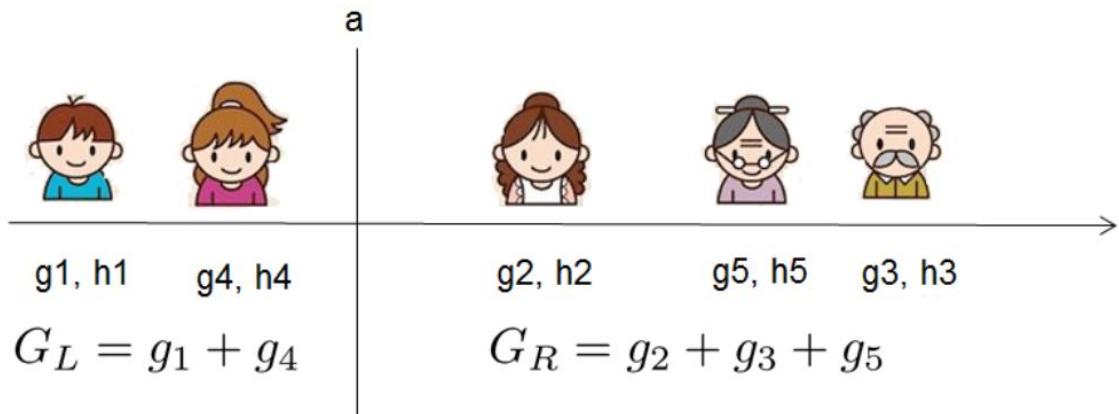
比如设置一个值 a ，然后枚举所有 $x < a$ 、 $a < x$ 这样的条件（ x 代表某个特征比如年龄age，把age从小到大排序：假定从左至右依次增大，则比 a 小的放在左边，比 a 大的放在右边），对于某个特定的分割 a ，我们要计算 a 左边和右边的导数和。

比如总共五个人，按年龄排好序后，一开始我们总共有如下4种划分方法：

1. 把第一个人和后面四个人划分开
 2. 把前两个人和后面三个人划分开
 3. 把前三个人和后面两个人划分开
 4. 把前面四个人和后面一个人划分开

接下来，把上面4种划分方法全都各自计算一下Gain，看哪种划分方法得到的Gain值最大则选取哪种划分方法，经过计算，发现把第2种划分方法“前面两个人和后面三个人划分开”得到的Gain值最大，意味着在一分为二这个第一层层面上这种划分方法是最合适的。

总结：



换句话说，对于所有的特征 x ，我们只要做一遍从左到右的扫描就可以枚举出所有分割的梯度和 GL 和 GR 。然后用计算Gain的公式计算每个分割方案的分数就可以了。

然后后续则依然按照这种划分方法继续第二层、第三层、第四层、第N层的分裂。

第二个值得注意的事情就是引入分割不一定会使得情况变好，所以我们有一个引入新叶子的惩罚项。优化这个目标对应了树的剪枝，当引入的分割带来的增益小于一个阀值 γ 的时候，则忽略这个分割。

换句话说，当引入某项分割，结果分割之后得到的分数 - 不分割得到的分数得到的值太小（比如小于我们的最低期望阀值 γ ），但却因此得到的复杂度过高，则相当于得不偿失，不如不分割。即做某个动作带来的好处比因此带来的坏处大不了太多，则为避免复杂多一事不如少一事的态度，不如不做。

相当于在我们发现“分”还不如“不分”的情况下后（得到的增益太小，小到小于阈值 γ ），会有2个叶子节点存在同一棵子树上的情况。

下面是论文中的算法

Algorithm 1: Exact Greedy Algorithm for Split Finding

```

Input:  $I$ , instance set of current node
Input:  $d$ , feature dimension
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
     $G_L \leftarrow 0$ ,  $H_L \leftarrow 0$ 
    for  $j$  in sorted( $I$ , by  $x_{jk}$ ) 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
end
Output: Split with max score

```

(2) 近似算法

主要针对数据太大，不能直接进行计算。

Algorithm 2: Approximate Algorithm for Split Finding

```
for k = 1 to m do
    | Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature k.
    | Proposal can be done per tree (global), or per split(local).
end
for k = 1 to m do
    |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_j$ 
    |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_j$ 
end
```

Follow same step as in previous section to find max score only among proposed splits.

4.4 小结：Boosted Tree Algorithm

总结一下，如图所示：

- Add a new tree in each iteration
- Beginning of each iteration, calculate

$$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$$

一阶
二阶

- Use the statistics to greedily grow a tree $f_t(x)$

$$Obj = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

贪心算法寻找切分点，生产每一轮新的树

- Add $f_t(x)$ to the model $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$

▪ Usually, instead we do $y^{(t)} = y^{(t-1)} + \epsilon f_t(x_i)$

▪ ϵ is called step-size or shrinkage, usually set around 0.1

▪ This means we do not do full optimization in each step and reserve chance for future rounds, it helps prevent overfitting

称之为：缩减因子
同样为了避免过拟合

如果某个样本 label 数值为 4，那么第一个回归树预测 3，第二个预测为 1；另外一组回归树，一个预测 2，一个预测 2，那么倾向后一种，为什么呢？前一种情况，第一棵树学的太多，太接近 4，也就意味着有较大的过拟合的风险。

听起来很美好，可是怎么实现呢，上面这个目标函数跟实际的参数怎么联系起来，我们说过，回归树的参数：

- 选取哪个 feature 分裂节点
- 节点的预测值

最终的策略就是：贪心 + 最优化（对的，二次最优化）。

通俗解释贪心策略：就是决策时刻按照当前目标最优化决定。

总结：

总而言之，XGBoost 使用了和 CART 回归树一样的想法，利用贪婪算法，遍历所有特征的所有特征划分点，不同的是使用的目标函数不一样。

具体做法就是分裂后的目标函数值比单子叶子节点的目标函数的增益，同时为了限制树生长过深，还加了个阈值，只有当增益大于该阈值才进行分裂。

以下便为设定的阈值：

$$\mathcal{L}_{split} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

参考文献：

1. python机器学习案例系列教程——GBDT算法、XGBOOST算法：
<https://blog.csdn.net/luanpeng825485697/article/details/79766455>

xgboost 对比 gbdt

1. 传统 GBDT 以CART作为基分类器，Xgboost 还支持线性分类器，这个时候 Xgboost 相当于带 L1 和 L2 正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题）。
2. 传统 GBDT 在优化时只用到一阶导数信息，Xgboost则对代价函数进行了二阶泰勒展开，同时用到了一阶和二阶导数。顺便提一下，**Xgboost工具支持自定义代价函数，只要函数可一阶和二阶求导。**
3. Xgboost 在代价函数里加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、每个叶子节点上输出的 score 的 L2模 的平方和。从 Bias-variance tradeoff 角度来讲，**正则项降低了模型variance**，使学习出来的模型更加简单，防止过拟合，这也是 Xgboost 优于传统 GBDT 的一个特性
4. Shrinkage（缩减），相当于学习速率（Xgboost 中的 eta）。Xgboost在进行完一次迭代后，会将叶子节点的权重乘上该系数，**主要是为了削弱每棵树的影响，让后面有更大的学习空间。实际应用中，一般把 eta 设置得小一点，然后迭代次数设置得大一点。**（补充：传统GBDT的实现也有学习速率）
5. 列抽样（column subsampling）。Xgboost 借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算，这也是 Xgboost 异于传统gbdt的一个特性。
6. 对缺失值的处理。对于特征的值有缺失的样本，Xgboost 可以自动学习出它的分裂方向。
7. Xgboost工具支持并行。boosting不是一种串行的结构吗？怎么并行的？注意 **Xgboost 的并行不是 tree粒度 的并行**，Xgboost 也是一次迭代完才能进行下一次迭代的（第 t 次迭代的代价函数里包含了前面 t-1 次迭代的预测值）。**Xgboost 的并行是在特征粒度上的。**决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），**Xgboost 在训练之前，预先对数据进行了排序，然后保存为 block结构，后面的迭代中重复地使用这个结构，大大减小计算量。**这个 block结构 也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，各个特征的增益计算就可以开多线程进行。
8. **可并行的近似直方图算法。**树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。**当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以 Xgboost 还提出了一种可并行的近似直方图算法，用于高效地生成候选的分割点。**