GBDT概念

GBDT也是集成学习Boosting家族的成员，但是却和传统的Adaboost有很大的不同。回顾下Adaboost，我们是利用前一轮迭代弱学习器的误差率来更新训练集的权重，这样一轮轮的迭代下去。GBDT也是迭代，使用了前向分布算法，但是弱学习器限定了只能使用CART回归树模型，同时迭代思路和Adaboost也有所不同。

在GBDT的迭代中，假设我们前一轮迭代得到的强学习器是ft−1(x), 损失函数是L(y,ft−1(x)), 我们本轮迭代的目标是找到一个CART回归树模型的弱学习器ht(x)，让本轮的损失函数L(y,ft(x)=L(y,ft−1(x)+ht(x))最小。也就是说，本轮迭代找到决策树，要让样本的损失尽量变得更小。

GBDT的思想可以用一个通俗的例子解释，假如有个人30岁，我们首先用20岁去拟合，发现损失有10岁，这时我们用6岁去拟合剩下的损失，发现差距还有4岁，第三轮我们用3岁拟合剩下的差距，差距就只有一岁了。如果我们的迭代轮数还没有完，可以继续迭代下面，每一轮迭代，拟合的岁数误差都会减小。

从上面的例子看这个思想还是蛮简单的，但是有个问题是这个损失的拟合不好度量，损失函数各种各样，怎么找到一种通用的拟合方法呢？

前向分布算法

加法模型和前向分布算法

如下图所示的便是一个加法模型

$$f(x)=\sum_{m=1}^{M}\beta_m b(x;\gamma_m)$$

其中，$b(x;\gamma_m)$称为基函数，γ_m称为基函数的参数，β_m称为基函数的系数。

在给定训练数据及损失函数$L(y,f(x))$的条件下，学习加法模型$f(x)$成为经验风险极小化问题，即损失函数极小化问题：

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

随后，该问题可以作如此简化：从前向后，每一步只学习一个基函数及其系数，逐步逼近上式，即：每步只优化如下损失函数：

$$\min_{\beta,\gamma}\sum_{i=1}^{N}L(y_i,\beta b(x_i;\gamma))$$

这个优化方法便就是所谓的前向分布算法。

下面，咱们来具体看下前向分步算法的算法流程：

```
输入：训练数据集
损失函数：
基函数集：
输出：加法模型
算法步骤：
    1. 初始化
    2. 对于m=1,2,..M
    a)极小化损失函数
```

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{M} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma))$$

得到参数β_m和γ_m。

b)更新

$$f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$$

3．最终得到加法模型

$$f(x) = f_M(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m)$$

就这样，前向分步算法将同时求解从m=1到M的所有参数（β_m、γ_m）的优化问题简化为逐次求解各个β_m、γ_m（1≤m≤M）的优化问题。

GBDT负梯度拟合

我们介绍了GBDT的基本思路，但是没有解决损失函数拟合方法的问题。针对这个问题，大牛Freidman提出了用损失函数的负梯度来拟合本轮损失的近似值，进而拟合一个CART回归树。第t轮的第i个样本的损失函数的负梯度表示为

$$r_{ti} = -[\frac{\partial L(y_i, f(x_i)))}{\partial f(x_i)}]_{f(x)=f_{t-1}(x)}$$

利用(x_i,r_{ti})(i=1,2,..m),我们可以拟合一颗CART回归树，得到了第t颗回归树，其对应的叶节点区域R_{tj},j=1,2,...,J。其中J为叶子节点的个数。针对每一个叶子节点里的样本，我们求出使损失函数最小，也就是拟合叶子节点最好的的输出值c_{tj} 如下：

$$c_{tj} = \arg\min_{c} \sum_{xi \in R_{tj}} L(y_i, f_{t-1}(x_i) + c)$$

这样我们就得到了本轮的决策树拟合函数如下：

$$h_t(x) = \sum_{j=1}^{J} c_{tj} I(x \in R_{tj})$$

从而本轮最终得到的强学习器的表达式如下：

$$f_t(x) = f_{t-1}(x) + \sum_{j=1}^{J} c_{tj} I(x \in R_{tj})$$

通过损失函数的负梯度来拟合，我们找到了一种通用的拟合损失误差的办法，这样无轮是分类问题还是回归问题，我们通过其损失函数的负梯度的拟合，就可以用GBDT来解决我们的分类回归问题。区别仅仅在于损失函数不同导致的负梯度不同而已。

GBDT回归算法

好了，有了上面的思路，下面我们总结下GBDT的回归算法。为什么没有加上分类算法一起？那是因为分类算法的输

出是不连续的类别值，需要一些处理才能使用负梯度，我们在下一节讲。

输入是训练集样本T={(x,y1),(x2,y2),...(xm,ym)}

， 最大迭代次数T, 损失函数L。

输出是强学习器f(x)

1) 初始化弱学习器

$$f_0(x) = \arg\min_c \sum_{i=1}^{m} L(y_i, c)$$

2)对迭代轮数t=1,2,...T有：

a)对样本i=1,2，...m，计算负梯度

$$r_{ti} = -[\frac{\partial L(yi, f(xi)))}{\partial f(xi)}]_{f(x)=f_{t-1}(x)}$$

b)利用(xi,rti)(i=1,2,..m) , 拟合一颗CART回归树,得到第t颗回归树，其对应的叶子节点区域为Rtj,j=1,2,...,J

。其中J为回归树t的叶子节点的个数。

c) 对叶子区域j =1,2,..J,计算最佳拟合值

$$ctj = \arg\min_c \sum_{xi \in R_{tj}} L(y_i, f_{t-1}(x_i) + c)$$

d) 更新强学习器

$$f_t(x) = f_{t-1}(x) + \sum_{j=1}^{J} c_{tj} I(x \in R_{tj})$$

3) 得到强学习器f(x)的表达式

$$f(x) = f_T(x) = f_0(x) + \sum_{t=1}^{T} \sum_{j=1}^{J} c_{tj} I(x \in R_{tj})$$

GBDT分类算法

这里我们再看看GBDT分类算法，GBDT的分类算法从思想上和GBDT的回归算法没有区别，但是由于样本输出不是连续的值，而是离散的类别，导致我们无法直接从输出类别去拟合类别输出的误差。

为了解决这个问题，主要有两个方法，一个是用指数损失函数，此时GBDT退化为Adaboost算法。另一种方法是用类似于逻辑回归的对数似然损失函数的方法。也就是说，我们用的是类别的预测概率值和真实概率值的差来拟合损失。本文仅讨论用对数似然损失函数的GBDT分类。而对于对数似然损失函数，我们又有二元分类和多元分类的区别。

二元GBDT分类算法

对于二元GBDT，如果用类似于逻辑回归的对数似然损失函数，则损失函数为：

$$L(y, f(x)) = \log(1 + \exp(-yf(x)))$$

其中y∈{−1,+1}。则此时的负梯度误差为

$$r_{ti} = -[\frac{\partial L(y, f(x_i))}{\partial f(x_i)}]_{f(x)=f_{t-1}(x)} = y_i/(1 + exp(y_i f(x_i)))$$

对于生成的决策树，我们各个叶子节点的最佳负梯度拟合值为

$$c_{tj} = \arg\min_{c} \sum_{x_i \in R_{tj}} log(1+exp(-y_i(f_{t-1}(x_i)+c)))$$

由于上式比较难优化，我们一般使用近似值代替

$$c_{tj} = \sum x_i \in R_{tj} / \sum_{x_i \in R_{tj}} |r_{ti}|(1-|r_{ti}|)$$

除了负梯度计算和叶子节点的最佳负梯度拟合的线性搜索，二元GBDT分类和GBDT回归算法过程相同。

多元GBDT分类算法

多元GBDT要比二元GBDT复杂一些，对应的是多元逻辑回归和二元逻辑回归的复杂度差别。假设类别数为K，则此时我们的对数似然损失函数为： $$L(y,f(x))=−∑_{k=1}^Ky_klogp_k(x)$$ 其中如果样本输出类别为k，则yk=1。第k类的概率pk(x)的表达式为：

$$p_k(x) = exp(f_k(x)) / \sum_{l=1}^{K} exp(f_l(x))$$

集合上两式，我们可以计算出第t轮的第i个样本对应类别l的负梯度误差为

$$r_{til} = -[\frac{\partial L(y_i, f(x_i)))}{\partial f(x_i)}]_{f_k(x)=f_{l,t-1}(x)} = y_{il} - p_{l,t-1}(xi)$$

观察上式可以看出，其实这里的误差就是样本i对应类别l的真实概率和t−1轮预测概率的差值。

对于生成的决策树，我们各个叶子节点的最佳负梯度拟合值为

$$ctjl = \arg\min_{c_{jl}} \sum_{i=0}^{m} \sum_{k=1}^{K} L(y_k, f_{t-1,l}(x) + \sum_{j=0}^{J} c_{jl} I(x_i \in R_{tj}))$$

由于上式比较难优化，我们一般使用近似值代替

$$c_{tjl} = \frac{K-1}{K} \frac{\sum_{xi \in R_{tjl}} r_{til}}{\sum_{x_i \in R_{til}} |r_{til}|(1-|r_{til}|)}$$

除了负梯度计算和叶子节点的最佳负梯度拟合的线性搜索，多元GBDT分类和二元GBDT分类以及GBDT回归算法过程相同。

GBDT的正则化

和Adaboost一样，我们也需要对GBDT进行正则化，防止过拟合。GBDT的正则化主要有三种方式。 第一种是和Adaboost类似的正则化项，即步长(learning rate)。定义为ν,对于前面的弱学习器的迭代

$$f_k(x) = f_{k-1}(x) + h_k(x)$$

如果我们加上了正则化项，则有

$$f_k(x) = f_{k-1}(x) + \nu h_k(x)$$

ν的取值范围为0<ν≤1。对于同样的训练集学习效果，较小的ν

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。 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。 使用了子采样的GBDT有时也称作随机梯度提升树(Stochastic

Gradient Boosting Tree, SGBT)。由于使用了子采样，程序可以通过采样分发到不同的任务去做boosting的迭代过程，最后形成新树，从而减少弱学习器难以并行学习的弱点。

```
class GradientBoostingClassifier(BaseGradientBoosting, ClassifierMixin):
    """Gradient Boosting for classification.

    GB builds an additive model in a
    forward stage-wise fashion; it allows for the optimization of
    arbitrary differentiable loss functions. In each stage ``n_classes_``
    regression trees are fit on the negative gradient of the
    binomial or multinomial deviance loss function. Binary classification
    is a special case where only a single regression tree is induced.

    Read more in the :ref:`User Guide <gradient_boosting>`.

    Parameters
    ----------
    loss : {'deviance', 'exponential'}, optional (default='deviance')
        loss function to be optimized. 'deviance' refers to
        deviance (= logistic regression) for classification
        with probabilistic outputs. For loss 'exponential' gradient
        boosting recovers the AdaBoost algorithm.

    learning_rate : float, optional (default=0.1)
        learning rate shrinks the contribution of each tree by `learning_rate`.
        There is a trade-off between learning_rate and n_estimators.

    n_estimators : int (default=100)
        The number of boosting stages to perform. Gradient boosting
        is fairly robust to over-fitting so a large number usually
        results in better performance.

    subsample : float, optional (default=1.0)
        The fraction of samples to be used for fitting the individual base
        learners. If smaller than 1.0 this results in Stochastic Gradient
        Boosting. `subsample` interacts with the parameter `n_estimators`.
        Choosing `subsample < 1.0` leads to a reduction of variance
        and an increase in bias.

    criterion : string, optional (default="friedman_mse")
        The function to measure the quality of a split. Supported criteria
        are "friedman_mse" for the mean squared error with improvement
        score by Friedman, "mse" for mean squared error, and "mae" for
        the mean absolute error. The default value of "friedman_mse" is
        generally the best as it can provide a better approximation in
        some cases.

        .. versionadded:: 0.18

    min_samples_split : int, float, optional (default=2)
        The minimum number of samples required to split an internal node:

        - If int, then consider `min_samples_split` as the minimum number.
        - If float, then `min_samples_split` is a fraction and
          `ceil(min_samples_split * n_samples)` are the minimum
          number of samples for each split.

        .. versionchanged:: 0.18
```

```
           Added float values for fractions.

    min_samples_leaf : int, float, optional (default=1)
        The minimum number of samples required to be at 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.  This may have the effect of smoothing the model,
        especially in regression.

        - If int, then consider `min_samples_leaf` as the minimum number.
        - If float, then `min_samples_leaf` is a fraction and
          `ceil(min_samples_leaf * n_samples)` are the minimum
          number of samples for each node.

        .. versionchanged:: 0.18
           Added float values for fractions.

    min_weight_fraction_leaf : float, optional (default=0.)
        The minimum weighted fraction of the sum total of weights (of all
        the input samples) required to be at a leaf node. Samples have
        equal weight when sample_weight is not provided.

    max_depth : integer, optional (default=3)
        maximum depth of the individual regression estimators. The maximum
        depth limits the number of nodes in the tree. Tune this parameter
        for best performance; the best value depends on the interaction
        of the input variables.

    min_impurity_decrease : float, optional (default=0.)
        A node will be split if this split induces a decrease of the impurity
        greater than or equal to this value.

        The weighted impurity decrease equation is the following::

            N_t / N * (impurity - N_t_R / N_t * right_impurity
                                - N_t_L / N_t * left_impurity)

        where ``N`` is the total number of samples, ``N_t`` is the number of
        samples at the current node, ``N_t_L`` is the number of samples in the
        left child, and ``N_t_R`` is the number of samples in the right child.

        ``N``, ``N_t``, ``N_t_R`` and ``N_t_L`` all refer to the weighted sum,
        if ``sample_weight`` is passed.

        .. versionadded:: 0.19

    min_impurity_split : float, (default=1e-7)
        Threshold for early stopping in tree growth. A node will split
        if its impurity is above the threshold, otherwise it is a leaf.

        .. deprecated:: 0.19
           ``min_impurity_split`` has been deprecated in favor of
           ``min_impurity_decrease`` in 0.19. The default value of
           ``min_impurity_split`` will change from 1e-7 to 0 in 0.23 and it
           will be removed in 0.25. Use ``min_impurity_decrease`` instead.

    init : estimator, optional
        An estimator object that is used to compute the initial
        predictions. ``init`` has to provide ``fit`` and ``predict``.
    If None it uses ``loss.init_estimator``.

random_state : int, RandomState instance or None, optional (default=None)
    If int, random_state is the seed used by the random number generator;
    If RandomState instance, random_state is the random number generator;
    If None, the random number generator is the RandomState instance used
    by `np.random`.

max_features : int, float, string or None, optional (default=None)
    The number of features to consider when looking for the best split:

    - If int, then consider `max_features` features at each split.
    - If float, then `max_features` is a fraction and
      `int(max_features * n_features)` features are considered at each
      split.
    - If "auto", then `max_features=sqrt(n_features)`.
    - If "sqrt", then `max_features=sqrt(n_features)`.
    - If "log2", then `max_features=log2(n_features)`.
    - If None, then `max_features=n_features`.

    Choosing `max_features < n_features` leads to a reduction of variance
    and an increase in bias.

    Note: the search for a split does not stop until at least one
    valid partition of the node samples is found, even if it requires to
    effectively inspect more than ``max_features`` features.

verbose : int, default: 0
    Enable verbose output. If 1 then it prints progress and performance
    once in a while (the more trees the lower the frequency). If greater
    than 1 then it prints progress and performance for every tree.

max_leaf_nodes : int or None, optional (default=None)
    Grow trees with ``max_leaf_nodes`` in best-first fashion.
    Best nodes are defined as relative reduction in impurity.
    If None then unlimited number of leaf nodes.

warm_start : bool, default: False
    When set to ``True``, reuse the solution of the previous call to fit
    and add more estimators to the ensemble, otherwise, just erase the
    previous solution. See :term:`the Glossary <warm_start>`.

presort : bool or 'auto', optional (default='auto')
    Whether to presort the data to speed up the finding of best splits in
    fitting. Auto mode by default will use presorting on dense data and
    default to normal sorting on sparse data. Setting presort to true on
    sparse data will raise an error.

    .. versionadded:: 0.17
       *presort* parameter.

validation_fraction : float, optional, default 0.1
    The proportion of training data to set aside as validation set for
    early stopping. Must be between 0 and 1.
    Only used if ``n_iter_no_change`` is set to an integer.

    .. versionadded:: 0.20

n_iter_no_change : int, default None
```

```
        ``n_iter_no_change`` is used to decide if early stopping will be used
        to terminate training when validation score is not improving. By
        default it is set to None to disable early stopping. If set to a
        number, it will set aside ``validation_fraction`` size of the training
        data as validation and terminate training when validation score is not
        improving in all of the previous ``n_iter_no_change`` numbers of
        iterations.

        .. versionadded:: 0.20

    tol : float, optional, default 1e-4
        Tolerance for the early stopping. When the loss is not improving
        by at least tol for ``n_iter_no_change`` iterations (if set to a
        number), the training stops.

        .. versionadded:: 0.20

    Attributes
    ----------
    n_estimators_ : int
        The number of estimators as selected by early stopping (if
        ``n_iter_no_change`` is specified). Otherwise it is set to
        ``n_estimators``.

        .. versionadded:: 0.20

    feature_importances_ : array, shape (n_features,)
        The feature importances (the higher, the more important the feature).

    oob_improvement_ : array, shape (n_estimators,)
        The improvement in loss (= deviance) on the out-of-bag samples
        relative to the previous iteration.
        ``oob_improvement_[0]`` is the improvement in
        loss of the first stage over the ``init`` estimator.

    train_score_ : array, shape (n_estimators,)
        The i-th score ``train_score_[i]`` is the deviance (= loss) of the
        model at iteration ``i`` on the in-bag sample.
        If ``subsample == 1`` this is the deviance on the training data.

    loss_ : LossFunction
        The concrete ``LossFunction`` object.

    init_ : estimator
        The estimator that provides the initial predictions.
        Set via the ``init`` argument or ``loss.init_estimator``.

    estimators_ : ndarray of DecisionTreeRegressor,\
 shape (n_estimators, ``loss_.K``)
        The collection of fitted sub-estimators. ``loss_.K`` is 1 for binary
        classification, otherwise n_classes.

    Notes
    -----
    The features are always randomly permuted at each split. Therefore,
    the best found split may vary, even with the same training data and
    ``max_features=n_features``, if the improvement of the criterion is
    identical for several splits enumerated during the search of the best
    split. To obtain a deterministic behaviour during fitting,
    ``random_state`` has to be fixed.

    See also
    --------
    sklearn.tree.DecisionTreeClassifier, RandomForestClassifier
    AdaBoostClassifier

    References
    ----------
    J. Friedman, Greedy Function Approximation: A Gradient Boosting
    Machine, The Annals of Statistics, Vol. 29, No. 5, 2001.

    J. Friedman, Stochastic Gradient Boosting, 1999

    T. Hastie, R. Tibshirani and J. Friedman.
    Elements of Statistical Learning Ed. 2, Springer, 2009.
    """
```