GBDT算法及其应用

目录

提升树之Adaboost算法的介绍

模型介绍

提升树算法与线性回归模型的思想类似，所不同的是该算法实现了多棵基础决策树$f(x)$的加权运算，最具代表的提升树为AdaBoost算法，即：

$$F(x) = \sum_{m=1}^{M} \alpha_m f_m(x) = F_{m-1}(x) + \alpha_m f_m(x)$$

其中，$F(x)$是由M棵基础决策树构成的最终提升树，$F_{m-1}(x)$表示经过$m-1$轮迭代后的提升树，α_m为第m棵基础决策树所对应的权重，$f_m(x)$为第m棵基础决策树。

提升树之Adaboost算法的介绍

损失函数的介绍

对于Adaboost算法而言，每一棵基础决策树都是基于前一棵基础决策树的分类结果对样本点设置不同的权重，如果在前一棵基础决策树中将某样本点预测错误，就会增大该样本点的权重，否则会相应降低样本点的权重，进而再构建下一棵基础决策树，更加关注权重大的样本点。

所以，AdaBoost算法需要解决三大难题，即样本点的权重w_{mi}如何确定、基础决策树$f(x)$如何选择以及每一棵基础决策树所对应的权重α_m如何计算。

提升树之Adaboost算法的介绍

损失函数的介绍

$$L(y,F(x)) = exp(-yF(x))$$

$$= exp\left(-y\sum_{m=1}^{M}\alpha_m f_m(x)\right)$$

$$= exp\left(-y(F_{m-1}(x) + \alpha_m f_m(x))\right)$$

如果提升树$F_{m-1}(x)$还能够继续提升，就说明损失函数还能够继续降低，换句话说，如果将所有训练样本点带入损失函数中，一定存在一个最佳的α_m和$f_m(x)$，使得损失函数尽量最大化地降低，即：

$$(\alpha_m, f_m(x)) = argmin_{\alpha, f(x)}\sum_{i=1}^{N} exp\left(-y_i(F_{m-1}(x_i) + \alpha_m f_m(x_i))\right)$$

提升树之Adaboost算法的介绍

损失函数的介绍

进一步，还可以将最小化的目标函数改写成下式：

$$(\alpha_m, f_m(x)) = argmin_{\alpha, f(x)} \sum_{i=1}^{N} p_{mi} ex\, p(-y_i \alpha_m f_m(x_i))$$

其中，$p_{mi} = exp[-y_i F_{m-1}(x_i)]$，由于$p_{mi}$与损失函数中的$\alpha_m$和$f_m(x)$无关，因此在求解最小化的问题时只需重点关注$\sum_{i=1}^{N} exp(-y_i \alpha_m f_m(x_i))$部分。

提升树之Adaboost算法的介绍

损失函数的介绍

对于$\sum_{i=1}^{N} exp(-y_i \alpha_m f_m(x_i))$而言，当第$m$棵基础决策树能够准确预测时，$y_i$与$f_m(x_i)$的乘积为1，否则为-1，于是$exp(-y_i \alpha_m f_m(x_i))$的结果为$exp(-\alpha_m)$或$exp(\alpha_m)$，对于某个固定的$\alpha_m$而言，损失函数中的和式仅仅是关于$\alpha_m$的式子。所以，要想求得损失函数的最小值，首先得找到最佳的$f_m(x)$，使得所有训练样本点x_i带入$f_m(x)$后，误判结果越少越好，即最佳的$f_m(x)$可以表示为：

$$f_m(x)^* = argmin_f \sum_{i=1}^{N} p_{mi} I(y_i \neq f_m(x))$$

其中，f表示所有可用的基础决策树空间，$f_m(x)^*$就是从f空间中寻找到的第m轮基础决策树，它能够使加权训练样本点的分类错误率最小，$I(y_i \neq f_m(x))$表示当第m棵基础决策树预测结果与实际值不相等时返回1。

提升树之Adaboost算法的介绍

损失函数的介绍

$$L(y, F(x)) = exp\left(-y\left(F_{m-1}(x) + \alpha_m f_m(x)\right)\right)$$

$$= \sum_{i=1}^{N} p_{mi}\, exp\left(-y_i \alpha_m f_m(x_i)\right)$$

$$= \sum_{y_i = f_m(x_i)} p_{mi} exp(-\alpha_m) + \sum_{y_i \neq f_m(x_i)} p_{mi} exp(\alpha_m)$$

$$= \left(exp(\alpha_m) - exp(-\alpha_m)\right) \sum_{i=1}^{N} p_{mi} I\left(y_i \neq f_m(x)\right)$$

$$+ exp(-\alpha_m) \sum_{i=1}^{N} p_{mi}$$

提升树之Adaboost算法的介绍

损失函数的介绍

$$\sum_{i=1}^{N} p_{mi} = \sum_{i=1}^{N} p_{mi} I\left(y_i \neq f_m(x)\right) + \sum_{i=1}^{N} p_{mi} I\left(y_i = f_m(x)\right)$$

$$= \sum_{y_i \neq f_m(x_i)} p_{mi} + \sum_{y_i = f_m(x_i)} p_{mi}$$

求偏导，令导函数为0

$$\frac{\partial L\left(y, F(x)\right)}{\partial \alpha_m} = \left(\alpha_m e^{\alpha_m} + \alpha_m e^{-\alpha_m}\right) \sum_{i=1}^{N} p_{mi} I\left(y_i \neq f_m(x)\right) - \alpha_m e^{-\alpha_m} \sum_{i=1}^{N} p_{mi}$$

最终令 $\frac{\partial L\left(y, F(x)\right)}{\partial \alpha_m} = 0$

$$\therefore \quad {\alpha_m}^* = \frac{1}{2} log \frac{1 - e_m}{e_m}$$

其中，$e_m = \frac{\sum_{i=1}^{N} p_{mi} I\left(y_i \neq f_m(x)\right)}{\sum_{i=1}^{N} p_{mi}} = \sum_{i=1}^{N} w_{mi} I\left(y_i \neq f_m(x)\right)$，表示基础决策树$m$的错误率。

提升树之Adaboost算法的介绍

Adaboost分类算法步骤

- 在第一轮基础决策树$f_1(x)$的构建中，会设置每一个样本点的权重w_{1i}均为$1/N$。
- 计算基础决策树$f_m(x)$在训练数据集上的误判率$e_m = \sum_{i=1}^{N} w_{mi}{}^{*} I\left(y_i \neq f_m(x_i)\right)$。
- 计算基础决策树$f_m(x)$所对应的权重 $\alpha_m{}^{*} = \frac{1}{2} log \frac{1-e_m}{e_m}$。
- 根据基础决策树 $f_m(x)$ 的预测结果，计算下一轮用于构建基础决策树的样本点权重 $w_{m+1,i}{}^{*}$ ：

$$w_{m+1,i}{}^{*} = \begin{cases} \dfrac{w_{mi} exp(-\alpha_m{}^{*})}{\sum_{i=1}^{N} w_{mi} exp(-\alpha_m{}^{*})}, & f_m(x_i)^{*} = y_i \\ \dfrac{w_{mi} exp(\alpha_m{}^{*})}{\sum_{i=1}^{N} w_{mi} exp(\alpha_m{}^{*})}, & f_m(x_i)^{*} \neq y_i \end{cases}$$

提升树之Adaboost算法的介绍

Adaboost回归算法步骤

- 初始化一棵仅包含根节点的树，并寻找到一个常数能够使损失函数达到极小值。
- 计算第m轮基础树的残差值$r_{mi} = y_i - f_m(x_i)$。
- 将残差值r_{mi}视为因变量，再利用自变量的值对其构造第$m+1$轮基础树$f_{m+1}(x)$。
- 重复步骤（2）和（3），得到多棵基础树，最终将这些基础树相加得到回归提升树 $F(x) = \sum_{m=1}^{M} f_m(x)$。

提升树之Adaboost算法的应用

Adaboost函数语法

```
AdaBoostClassifier(base_estimator=None, n_estimators=50,
                   learning_rate=1.0, algorithm='SAMME.R', random_state=None)

AdaBoostRegressor(base_estimator=None, n_estimators=50,
                  learning_rate=1.0, loss='linear', random_state=None)
```

base_estimator：用于指定提升算法所应用的基础分类器，默认为分类决策树（CART），也可以是其他基础分类器，但分类器必须支持带样本权重的学习，如神经网络。
n_estimators：用于指定基础分类器的数量，默认为50个，当模型在训练数据集中得到完美的拟合后，可以提前结束算法，不一定非得构建完指定个数的基础分类器。
learning_rate：用于指定模型迭代的学习率或步长，即对应的提升模型$F(x)$可以表示为$F(x) = F_{m-1}(x) + \upsilon\alpha_m f_m(x)$，其中的$\upsilon$就是该参数的指定值，默认值为1；对于较小的学习率$\upsilon$而言，则需要迭代更多次的基础分类器，通常情况下需要利用交叉验证法确定合理的基础分类器个数和学习率。

提升树之Adaboost算法的应用

Adaboost函数语法

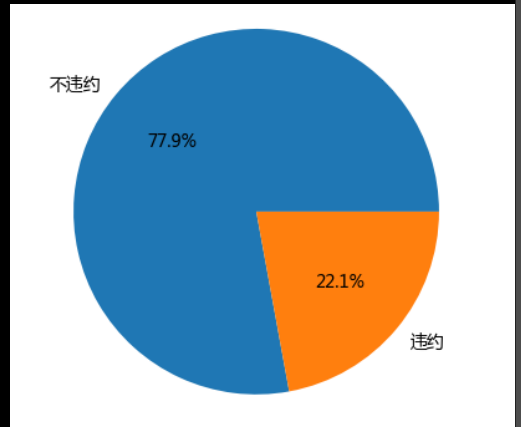
```
AdaBoostClassifier(base_estimator=None, n_estimators=50,
                   learning_rate=1.0, algorithm='SAMME.R', random_state=None)

AdaBoostRegressor(base_estimator=None, n_estimators=50,
                  learning_rate=1.0, loss='linear', random_state=None)
```

algorithm：用于指定AdaBoostClassifier分类器的算法，默认为'SAMME.R'，也可以使用'SAMME'；使用'SAMME.R'时，基础模型必须能够计算类别的概率值；一般而言，'SAMME.R'算法相比于'SAMME'算法，收敛更快、误差更小、迭代数量更少。
loss：用于指定AdaBoostRegressor回归提升树的损失函数，可以是'linear'，表示使用线性损失函数；也可以是'square'，表示使用平方损失函数；还可以是'exponential'，表示使用指数损失函数；该参数的默认值为'linear'。
r**andom_state**：用于指定随机数生成器的种子。

提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 读入数据
default = pd.read_excel(r'C:\Users\Administrator\Desktop\default of credit card.xls')
# 为确保绘制的饼图为圆形，需执行如下代码
plt.axes(aspect = 'equal')
# 统计客户是否违约的频数
counts = default.y.value_counts()
# 绘制饼图
plt.pie(x = counts, # 绘图数据
        labels=pd.Series(counts.index).map({0:'不违约',1:'违约'}),
        autopct='%.1f%%' # 设置百分比的格式，这里保留一位小数
       )
# 显示图形
plt.show()
```



提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 排除数据集中的ID变量和因变量，剩余的数据用作自变量X
X = default.drop(['ID','y'], axis = 1)
y = default.y
# 数据拆分
X_train,X_test,y_train,y_test = model_selection.train_test_split(X,y,test_size = 0.25,
                                                                 random_state = 1234)

# 构建AdaBoost算法的类
AdaBoost1 = ensemble.AdaBoostClassifier()
# 算法在训练数据集上的拟合
AdaBoost1.fit(X_train,y_train)
# 算法在测试数据集上的预测
pred1 = AdaBoost1.predict(X_test)
```

提升树之Adaboost算法的应用

信用卡是否违约的识别

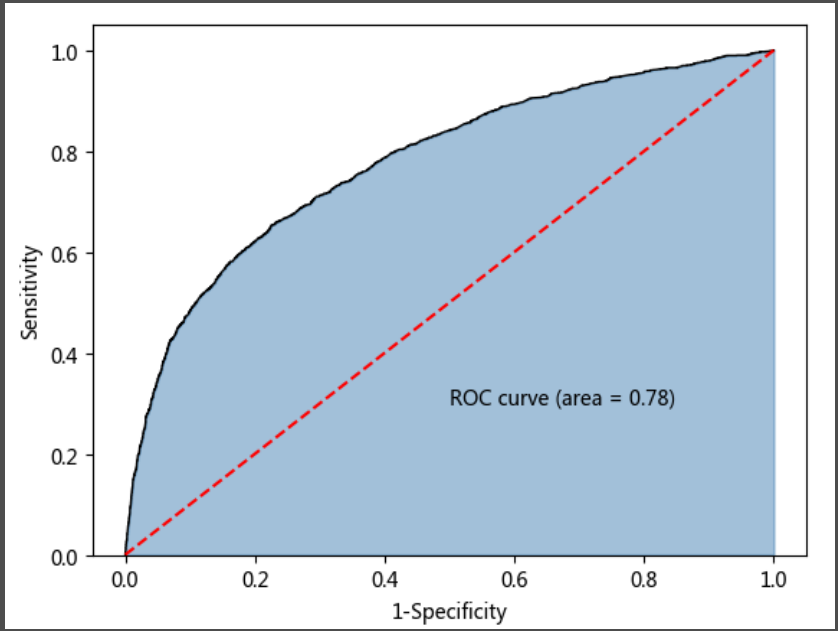
```
# 返回模型的预测效果
print('模型的准确率为：\n',metrics.accuracy_score(y_test, pred1))
print('模型的评估报告：\n',metrics.classification_report(y_test, pred1))
```

out:

```
模型的准确率为：
 0.812533333333
模型的评估报告：
                precision    recall  f1-score   support
          0       0.83      0.96      0.89      5800
          1       0.68      0.32      0.44      1700
avg / total       0.80      0.81      0.79      7500
```

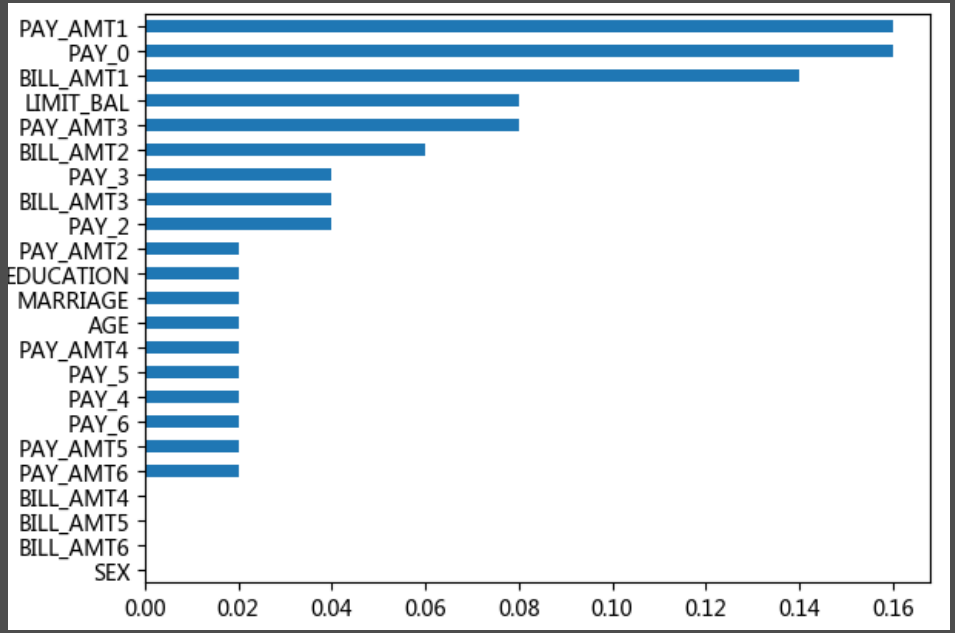
提升树之Adaboost算法的应用

信用卡是否违约的识别



提升树之Adaboost算法的应用

信用卡是否违约的识别



提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 取出重要性比较高的自变量建模
predictors = list(importance[importance>0.02].index)
Predictors
# 通过网格搜索法选择基础模型所对应的合理参数组合
max_depth = [3,4,5,6]
params1 = {'base_estimator__max_depth':max_depth}
base_model = GridSearchCV(estimator = ensemble.AdaBoostClassifier(base_estimator =
                                        DecisionTreeClassifier()),param_grid= params1,
                          scoring = 'roc_auc', cv = 5, n_jobs = 4, verbose = 1)
base_model.fit(X_train[predictors],y_train)
# 返回参数的最佳组合和对应AUC值
base_model.best_params_, base_model.best_score_

out:
{'base_estimator__max_depth': 3}, 0.744250467979361 45
```

提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 通过网格搜索法选择合理的Adaboost算法参数
n_estimators = [100,200,300]
learning_rate = [0.01,0.05,0.1,0.2]
params2 = {'n_estimators':n_estimators,'learning_rate':learning_rate}
adaboost = GridSearchCV(estimator = ensemble.AdaBoostClassifier(base_estimator =
                                DecisionTreeClassifier(max_depth = 3)),param_grid= params2,
                                scoring = 'roc_auc', cv = 5, n_jobs = 4, verbose = 1)
adaboost.fit(X_train[predictors] ,y_train)
# 返回参数的最佳组合和对应AUC值
adaboost.best_params_, adaboost.best_score_
```

out:

{'learning_rate': 0.01, 'n_estimators': 300}, 0.76866547085583281

提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 使用最佳的参数组合构建AdaBoost模型
AdaBoost2 = ensemble.AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth =
3),n_estimators = 300, learning_rate = 0.01)
# 算法在训练数据集上的拟合
AdaBoost2.fit(X_train[predictors],y_train)
# 算法在测试数据集上的预测
pred2 = AdaBoost2.predict(X_test[predictors])
```

提升树之Adaboost算法的应用

信用卡是否违约的识别

```
# 返回模型的预测效果
print('模型的准确率为：\n',metrics.accuracy_score(y_test, pred2))
print('模型的评估报告：\n',metrics.classification_report(y_test, pred2))

out:
模型的准确率为：
 0.816
模型的评估报告：
                precision    recall    f1-score   support
          0          0.83      0.96      0.89      5800
          1          0.69      0.34      0.45      1700
avg / total          0.80      0.82      0.79      7500
```

梯度提升树之GBDT算法的介绍

模型介绍

梯度提升树算法实际上是提升算法的扩展版，在原始的提升算法中，如果损失函数为平方损失或指数损失，求解损失函数的最小值问题会非常简单，但如果损失函数为更一般的函数（如绝对值损失函数或Huber损失函数等），目标值的求解就会相对复杂很多。

梯度提升算法，是在第m轮基础模型中，利用损失函数的负梯度值作为该轮基础模型损失值的近似，并利用这个近似值构建下一轮基础模型。利用损失函数的负梯度值近似残差的计算就是梯度提升算法在提升算法上的扩展，这样的扩展使得目标函数的求解更为方便。

梯度提升树之GBDT算法的介绍

GBDT算法步骤

- 初始化一棵仅包含根节点的树，并寻找到一个常数$Const$能够使损失函数达到极小值

- 计算损失函数的负梯度值，用作残差的估计值，即：

$$r_{mi} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

- 利用数据集(x_i, r_{mi})拟合下一轮基础模型，得到对应的J个叶子节点R_{mj}，$j = 1,2,\cdots,J$；计算每个叶子节点R_{mj}的最佳拟合值，用以估计残差r_{mi}：

$$c_{mj} = argmin_c \sum_{x_i \in R_{mj}} L(y_i, f_{m-1}(x_i) + c)$$

梯度提升树之GBDT算法的介绍

GBDT算法步骤

进而得到第m轮的基础模型$f_m(x)$，再结合前$m-1$轮的基础模型，得到最终的梯度提升模型：

$$
\begin{aligned}
F_M(x) &= F_{M-1}(x) + f_m(x) \\
&= F_{M-1}(x) + \sum_{j=1}^{J} c_{mj} I\left(x_i \in R_{mj}\right) \\
&= \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I\left(x_i \in R_{mj}\right)
\end{aligned}
$$

其中，c_{mj}表示第m个基础模型$f_m(x)$在叶节点j上的预测值；$F_M(x)$表示由M个基础模型构成的梯度提升树，它是每一个基础模型在样本点x_i处的输出值c_{mj}之和。

梯度提升树之GBDT算法的介绍

GBDT算法在分类问题上的操作步骤

- 初始化一个弱分类器

$$f_0(x) = argmin_c \sum_{i=1}^{n} L(y_i, c)$$

- 计算损失函数的负梯度值

$$r_{mi} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

$$= -\left[\frac{\partial log\left(1 + exp(-y_i f(x_i))\right)}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

$$= \frac{y_i}{1 + exp(-y_i f(x_i))}$$

梯度提升树之GBDT算法的介绍

GBDT算法在分类问题上的操作步骤

利用数据集(x_i, r_{mi})拟合下一轮基础模型

$$f_m(x) = \sum_{j=1}^{J} c_{mj} I(x_i \in R_{mj})$$

$$\text{其中，} c_{mj} = argmin_c \sum_{x_i \in R_{mj}} log\left(1 + exp\left(-y_i(f_{m-1}(x_i) + c)\right)\right)$$

重复（2）和（3），并利用m个基础模型，构建梯度提升模型

$$F_M(x) = F_{M-1}(x) + f_m(x)$$

$$= \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I(x_i \in R_{mj})$$

梯度提升树之GBDT算法的介绍

GBDT算法在预测问题上的操作步骤

- 初始化一个弱分类器

$$f_0(x) = argmin_c \sum_{i=1}^{n} L(y_i, c)$$

- 计算损失函数的负梯度值

$$r_{mi} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

$$= -\left[\frac{\partial \frac{1}{2}(y_i - f(x_i))^2}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

$$= y_i - f(x_i)$$

梯度提升树之GBDT算法的介绍

GBDT算法在预测问题上的操作步骤

利用数据集(x_i, r_{mi})拟合下一轮基础模型

$$f_m(x) = \sum_{j=1}^{J} c_{mj} I\left(x_i \in R_{mj}\right)$$

其中，$c_{mj} = argmin_c \sum_{x_i \in R_{mj}} \frac{1}{2}\left(y_i - \left(f_{m-1}(x_i) + c\right)\right)^2$。

重复（2）和（3），并利用m个基础模型，构建梯度提升模型

$$F_M(x) = F_{M-1}(x) + f_m(x) = \sum_{m=1}^{M} \sum_{j=1}^{J} c_{mj} I\left(x_i \in R_{mj}\right)$$

梯度提升树之GBDT算法的应用

GBDT函数介绍

GradientBoostingRegressor(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0,
criterion='friedman_mse', min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3,
min_impurity_decrease=0.0, min_impurity_split=None, init=None,
random_state=None, max_features=None, alpha=0.9, verbose=0,
max_leaf_nodes=None, warm_start=False, presort='auto')

loss：用于指定GBDT算法的损失函数，对于分类的GBDT，可以选择'deviance'和'exponential'，分别表示对数似然损失函数和指数损失函数；对于预测的GBDT，可以选择'ls' 'lad' 'huber'和'quantile'，分别表示平方损失函数、绝对值损失函数、Huber损失函数（前两种损失函数的结合，当误差较小时，使用平方损失，否则使用绝对值损失，误差大小的度量可使用alpha参数指定 ）和分位数回归损失函数（需通过alpha参数设定分位数）。

learning_rate：用于指定模型迭代的学习率或步长，即对应的梯度提升模型F(x)可以表示为$F_M(x) = F_{M-1}(x) + \upsilon f_m(x)$：，其中的υ就是该参数的指定值，默认值为0.1；对于较小的学习率υ而言，则需要迭代更多次的基础分类器，通常情况下需要利用交叉验证法确定合理的基础模型的个数和学习率。

梯度提升树之GBDT算法的应用

GBDT函数介绍

GradientBoostingRegressor(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0,
criterion='friedman_mse', min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3,
min_impurity_decrease=0.0, min_impurity_split=None, init=None,
random_state=None, max_features=None, alpha=0.9, verbose=0,
max_leaf_nodes=None, warm_start=False, presort='auto’)

n_estimators：用于指定基础模型的数量，默认为100个。
subsample：用于指定构建基础模型所使用的抽样比例，默认为1，表示使用原始数据构建每一个基础模型；当抽样比例小于1时，表示构建随机梯度提升树模型，通常会导致模型的方差降低，偏差提高。
criterion：用于指定分割质量的度量，默认为'friedman_mse'，表示使用Friedman均方误差，还可以使用'mse'和'mae'，分别表示均方误差和绝对误差。
min_samples_split：用于指定每个基础模型的根节点或中间节点能够继续分割的最小样本量， 默认为2。
min_samples_leaf：用于指定每个基础模型的叶节点所包含的最小样本量，默认为1。
min_weight_fraction_leaf：用于指定每个基础模型叶节点最小的样本权重，默认为0，表示不考虑叶节点的样本权值。

梯度提升树之GBDT算法的应用

GBDT函数介绍

```
GradientBoostingRegressor(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0,
                          criterion='friedman_mse', min_samples_split=2,
                          min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3,
                          min_impurity_decrease=0.0, min_impurity_split=None, init=None,
                          random_state=None, max_features=None, alpha=0.9, verbose=0,
                          max_leaf_nodes=None, warm_start=False, presort='auto’ )
```

max_depth：用于指定每个基础模型所包含的最大深度，默认为3层。
min_impurity_decrease：用于指定每个基础模型的节点是否继续分割的最小不纯度值，默认为0；如果不纯度超过指定的阈值，则节点需要分割，否则不分割。
min_impurity_split：该参数同min_impurity_decrease参数，在sklearn的0.21版本及之后版本将删除。
init：用于指定初始的基础模型，用于执行初始的分类或预测。
random_state：用于指定随机数生成器的种子，默认为None，表示使用默认的随机数生成器。
verbose：用于指定GBDT算法在计算过程中是否输出日志信息，默认为0，表示不输出。
alpha：当loss参数为'huber'或'quantile'时，该参数有效，分别用于指定误差的阈值和分位数。

梯度提升树之GBDT算法的应用

GBDT函数介绍

GradientBoostingRegressor(loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0,
criterion='friedman_mse', min_samples_split=2,
min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3,
min_impurity_decrease=0.0, min_impurity_split=None, init=None,
random_state=None, max_features=None, alpha=0.9, verbose=0,
max_leaf_nodes=None, warm_start=False, presort='auto’)

max_features：用于指定每个基础模型所包含的最多分割字段数，默认为None，表示分割时使用所有的字段；如果为具体的整数，则考虑使用对应的分割字段数；如果为0~1的浮点数，则考虑对应百分比的字段个数；如果为'sqrt'，则表示最多考虑$\sqrt{P}$个字段，与指定'auto'效果一致；如果为'log2'，则表示最多使用$log_2 P$个字段。其中，P表示数据集所有自变量的个数。
max_leaf_nodes：用于指定每个基础模型最大的叶节点个数，默认为None，表示对叶节点个数不做任何限制。
warm_start：bool类型参数，表示是否使用上一轮的训练结果，默认为False。

梯度提升树之GBDT算法的应用

信用卡是否违约的识别

```
# 运用网格搜索法选择梯度提升树的合理参数组合
learning_rate = [0.01,0.05,0.1,0.2]
n_estimators = [100,200,300]
max_depth = [3,4,5,6]
params = {'learning_rate':learning_rate,'n_estimators':n_estimators,'max_depth':max_depth}
gbdt_grid = GridSearchCV(estimator = ensemble.GradientBoostingClassifier(),
                         param_grid= params, scoring = 'roc_auc', cv = 5, n_jobs = 4, verbose = 1)
gbdt_grid.fit(X_train[predictors],y_train)
# 返回参数的最佳组合和对应AUC值
gbdt_grid.best_params_, gbdt_grid.best_score_
```

out:

```
{'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 100},  0.77397780446802755
```

梯度提升树之GBDT算法的应用

信用卡是否违约的识别

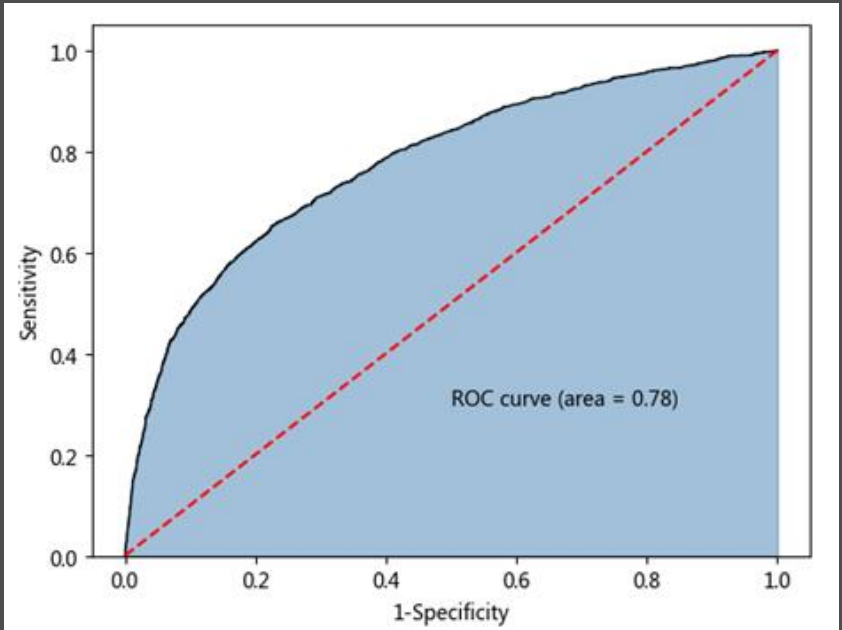
```
# 基于最佳参数组合的GBDT模型，对测试数据集进行预测
pred = gbdt_grid.predict(X_test[predictors])
# 返回模型的预测效果
print('模型的准确率为：\n',metrics.accuracy_score(y_test, pred))
print('模型的评估报告：\n',metrics.classification_report(y_test, pred))
```

out:

```
模型的准确率为：
 0.814266666667
模型的评估报告：
                precision    recall    f1-score   support
          0        0.83       0.95      0.89       5800
          1        0.68       0.35      0.46       1700
avg / total        0.80       0.81      0.79       7500
```

梯度提升树之GBDT算法的应用

信用卡是否违约的识别



GBDT模型在测试数据集上的预测效果与AdaBoost算法基本一致，进而可以说明GBDT算法采用一阶导函数的值近似残差是合理的，并且这种近似功能也提升了AdaBoost算法求解目标函数时的便捷性。

非平衡数据的处理

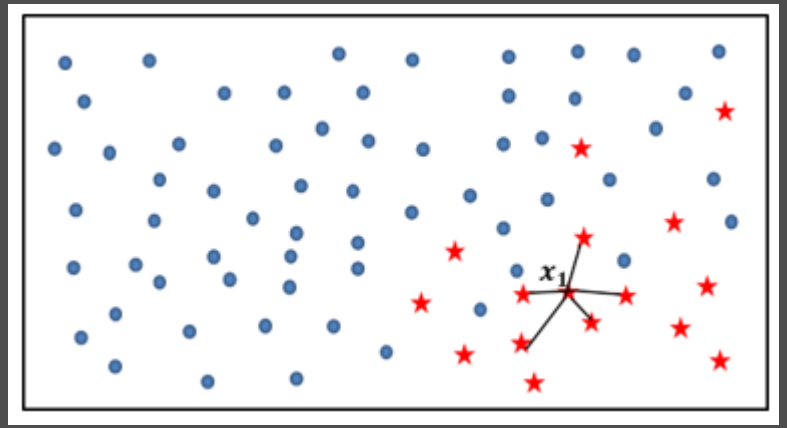
非平衡数据的特征

在实际应用中，类别型的因变量可能存在严重的偏倚，即类别之间的比例严重失调。如欺诈问题中，欺诈类观测在样本集中毕竟占少数；客户流失问题中，忠实的客户往往也是占很少一部分；在某营销活动的响应问题中，真正参与活动的客户也同样只是少部分。

如果数据存在严重的不平衡，预测得出的结论往往也是有偏的，即分类结果会偏向于较多观测的类。为了解决数据的非平衡问题，2002年Chawla提出了SMOTE算法，即合成少数过采样技术，它是基于随机过采样算法的一种改进方案。

非平衡数据的处理

SMOTE算法的思想



SMOTE算法的基本思想就是对少数类别样本进行分析和模拟，并将人工模拟的新样本添加到数据集中，进而使原始数据中的类别不再严重失衡。

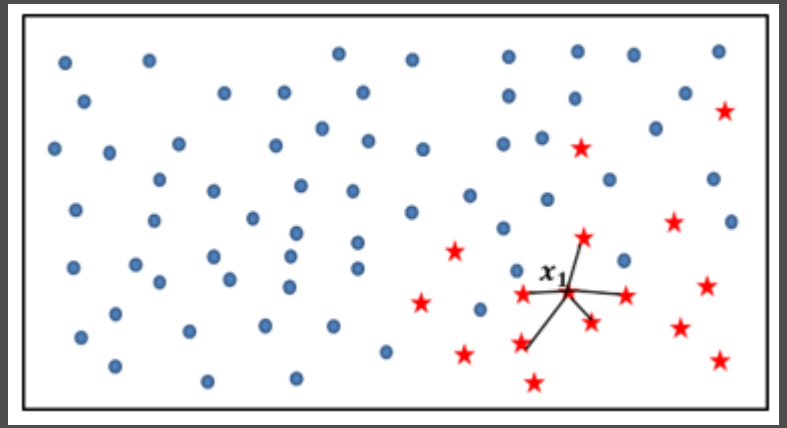
非平衡数据的处理

SMOTE算法的步骤

- 采样最邻近算法，计算出每个少数类样本的K个近邻。
- 从K个近邻中随机挑选N个样本进行随机线性插值。
- 构造新的少数类样本。
- 将新样本与原数据合成，产生新的训练集。

非平衡数据的处理

SMOTE算法的手工案例

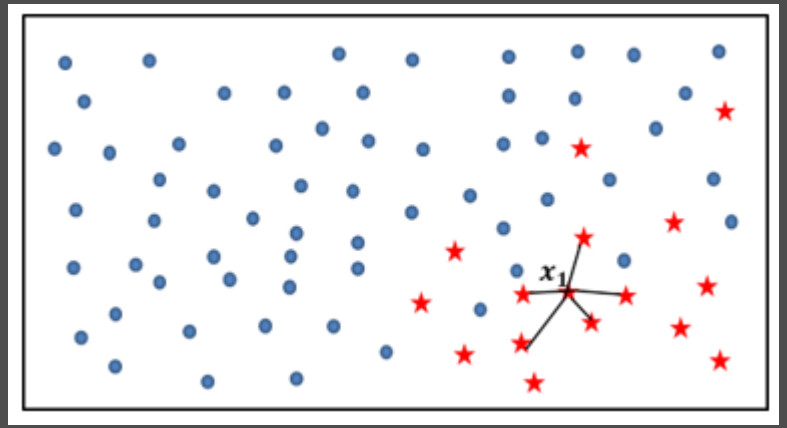


（1）利用第11章所介绍的KNN算法，选择离样本点x_1最近的K个同类样本点（不妨最近邻为5）。

（2）从最近的K个同类样本点中，随机挑选M个样本点（不妨设M为2），M的选择依赖于最终所希望的平衡率。

非平衡数据的处理

SMOTE算法的手工案例



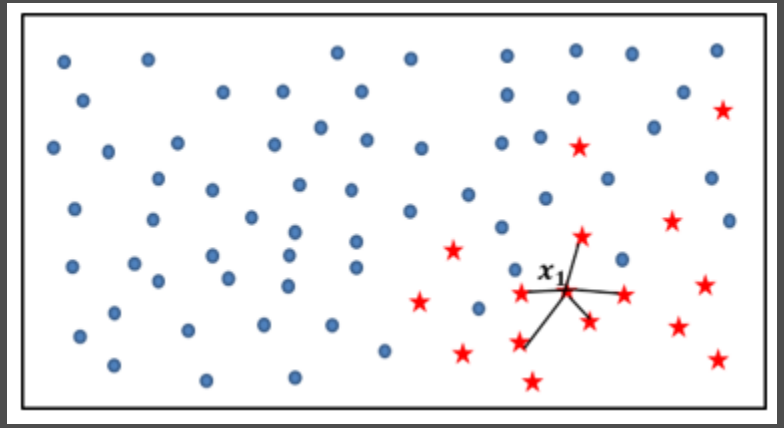
（3）对于每一个随机选中的样本点，构造新的样本点。新样本点的构造需要使用下方的公式：

$$x_{new} = x_i + rand(0,1) \times (x_j - x_i), \qquad j = 1,2,\cdots,M$$

其中，x_i表示少数类别中的一个样本点（如图中五角星所代表的x_1样本）；x_j表示从K近邻中随机挑选的样本点j；$rand(0,1)$表示生成0~1的随机数。

非平衡数据的处理

SMOTE算法的手工案例



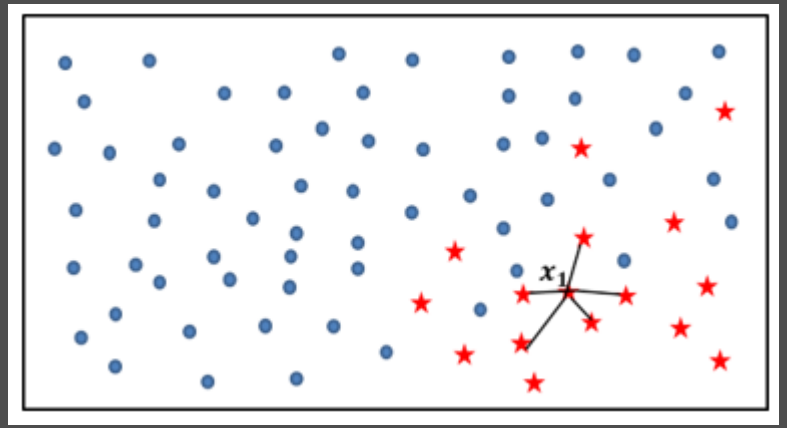
假设图中样本点x_1的观测值为(2,3,10,7)，从图中的5个近邻随机挑选两个样本点，它们的观测值分别为(1,1,5,8)和(2,1,7,6)，由此得到的两个新样本点为：

$$x_{new1} = (2,3,10,7) + 0.3 \times \big((1,1,5,8) - (2,3,10,7)\big) = (1.7,2.4,8.5,7.3)$$

$$x_{new2} = (2,3,10,7) + 0.26 \times \big((2,1,7,6) - (2,3,10,7)\big) = (2,2.48,9.22,6.74)$$

非平衡数据的处理

SMOTE算法的手工案例



（4）重复步骤（1）、（2）和（3），通过迭代少数类别中的每一个样本x_i，最终将原始的少数类别样本量扩大为理想的比例。

非平衡数据的处理

SMOTE算法的函数介绍

SMOTE(ratio='auto', random_state=None, k_neighbors=5, m_neighbors=10,
out_step=0.5, kind='regular', svm_estimator=None, n_jobs=1)

ratio：用于指定重抽样的比例，如果指定字符型的值，可以是'minority'（表示对少数类别的样本进行抽样）、'majority'（表示对多数类别的样本进行抽样）、'not minority'（表示采用欠采样方法）、'all'（表示采用过采样方法），默认为'auto'，等同于'all'和'not minority'。如果指定字典型的值，其中键为各个类别标签，值为类别下的样本量。
random_state：用于指定随机数生成器的种子，默认为None，表示使用默认的随机数生成器。
k_neighbors：指定近邻个数，默认为5个。
m_neighbors：指定从近邻样本中随机挑选的样本个数，默认为10个。
kind：用于指定SMOTE算法在生成新样本时所使用的选项，默认为'regular'，表示对少数类别的样本进行随机采样，也可以是'borderline1' 'borderline2'和'svm'。
svm_estimator：用于指定SVM分类器，默认为sklearn.svm.SVC，该参数的目的是利用支持向量机分类器生成支持向量，然后生成新的少数类别的样本。
n_jobs：用于指定SMOTE算法在过采样时所需的CPU数量，默认为1表示仅使用1个CPU运行算法，即不使用并行运算功能。

DBDT的改进之XGBoost算法介绍

XGBoost算法的介绍

XGBoost是由传统的GBDT模型发展而来的，GBDT模型在求解最优化问题时应用了一阶导技术，而XGBoost则使用损失函数的一阶和二阶导，而且可以自定义损失函数，只要损失函数可一阶和二阶求导。

XGBoost算法相比于GBDT算法还有其他优点，例如支持并行计算，大大提高算法的运行效率；XGBoost在损失函数中加入了正则项，用来控制模型的复杂度，进而可以防止模型的过拟合；XGBoost除了支持CART基础模型，还支持线性基础模型；XGBoost采用了随机森林的思想，对字段进行抽样，既可以防止过拟合，也可以降低模型的计算量。

DBDT的改进之XGBoost算法介绍

XGBoost算法的损失函数

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

其中，$\hat{y}_i^{(t)}$表示经第t轮迭代后的模型预测值，$\hat{y}_i^{(t-1)}$表示已知$t-1$个基础模型的预测值，$f_t(x_i)$表示第t个基础模型。

对于集成树，关键点就是第t个基础模型f_t的选择。所以，只需要寻找一个能够使目标函数尽可能最大化降低的f_t即可，故构造的目标函数如下：

$$Obj^{(t)} = \sum_{i=1}^{n} L\left(y_i, \hat{y}_i^{(t)}\right) + \sum_{j=1}^{t} \Omega(f_j)$$

$$= \sum_{i=1}^{n} L\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \sum_{j=1}^{t} \Omega(f_j)$$

DBDT的改进之XGBoost算法介绍

XGBoost算法的损失函数

损失函数中的$\Omega(f_j)$为第j个基础模型的正则项，用于控制模型的复杂度。为了简单起见，不妨将损失函数L表示为平方损失，则如上的目标函数可以表示为：

$$Obj^{(t)} = \sum_{i=1}^{n}\left(y_i - \left(\hat{y_i}^{(t-1)} + f_t(x_i)\right)\right)^2 + \sum_{j=1}^{t}\Omega(f_j)$$

$$= \sum_{i=1}^{n}\left({y_i}^2 + \left(\hat{y_i}^{(t-1)} + f_t(x_i)\right)^2 - 2y_i\left(\hat{y_i}^{(t-1)} + f_t(x_i)\right)\right) + \sum_{j=1}^{t}\Omega(f_j)$$

$$= \sum_{i=1}^{n}\left(2f_t(x_i)\left(\hat{y_i}^{(t-1)} - y_i\right) + f_t(x_i)^2 + \left(y_i - \hat{y_i}^{(t-1)}\right)^2\right) + \sum_{j=1}^{t}\Omega(f_j)$$

DBDT的改进之XGBoost算法介绍

XGBoost算法的损失函数

由于前$t-1$个基础模型是已知的，故$\hat{y}_i^{(t-1)}$的预测值也是已知的，同时前$t-1$个基础模型的复杂度也是已知的，故不妨将所有的已知项设为常数 $constant$ ，则目标函数可以重新表达为：

$$Obj^{(t)} = \sum_{i=1}^{n} \left(2 f_t(x_i) \left(\hat{y}_i^{(t-1)} - y_i\right) + f_t(x_i)^2\right) + \Omega(f_t) + constant$$

其中，$\left(\hat{y}_i^{(t-1)} - y_i\right)$项就是前$t-1$个基础模型所产生的残差，说明目标函数的选择与前$t-1$个基础模型的残差相关，这一点与GBDT是相同的。如上是假设损失函数为平方损失，对于更一般的损失函数来说，可以使用泰勒展开对损失函数值做近似估计。

DBDT的改进之XGBoost算法介绍

泰勒展开式

$$f(x+\Delta x) \approx f(x) + f(x)'\Delta x + f(x)''\Delta x^2$$

其中，$f(x)$是一个具有二阶可导的函数，$f(x)'$为$f(x)$的一阶导函数，$f(x)''$为$f(x)$的二阶导函数，Δx为$f(x)$在某点处的变化量。假设令损失函数L为泰勒公式中的f，令损失函数中$\hat{y}_i^{(t-1)}$项为泰勒公式中的x，令损失函数中$f_t(x_i)$项为泰勒公式中的Δx，则目标函数$Obj^{(t)}$可以近似表示为：

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

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

DBDT的改进之XGBoost算法介绍

泰勒展开式

在上式中，g_i和h_i分别是损失函数$L\left(y_i,\hat{y_i}^{(t-1)}\right)$关于$\hat{y_i}^{(t-1)}$的一阶导函数值和二阶导函数值，即它们可以表示为：

$$\begin{cases} g_i = \dfrac{\partial L\left(y_i,\hat{y_i}^{(t-1)}\right)}{\partial \hat{y_i}^{(t-1)}} \\ h_i = \dfrac{\partial^2 L\left(y_i,\hat{y_i}^{(t-1)}\right)}{\partial \hat{y_i}^{(t-1)}} \end{cases}$$

DBDT的改进之XGBoost算法介绍

损失函数的演变

假设基础模型f_t由CART树构成，对于一棵树来说，它可以被拆分为结构部分q，以及叶子节点所对应的输出值w。可以利用这两部分反映树的复杂度，即复杂度由树的叶子节点个数（反映树的结构）和叶子节点输出值的平方构成：

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

其中，T表示叶子节点的个数，w_j^2表示输出值向量的平方。CART树生长得越复杂，对应的T越大，$\Omega(f_t)$也越大。

DBDT的改进之XGBoost算法介绍

损失函数的演变

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

$$\approx \sum_{i=1}^{n}\left(g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + constant$$

$$\approx \sum_{i=1}^{n}\left(g_i w_{q(x_i)} + \frac{1}{2} h_i {w_{q(x_i)}}^2\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + constant$$

$$\approx \sum_{j=1}^{T}\left(\left(\sum_{i\in I_j} g_i\right) w_j + \frac{1}{2}\left(\sum_{i\in I_j} h_i\right) w_j^2\right) + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 + constant$$

$$\approx \sum_{j=1}^{T}\left(\left(\sum_{i\in I_j} g_i\right) w_j + \frac{1}{2}\left(\sum_{i\in I_j} (h_i + \lambda)\right) w_j^2\right) + \gamma T + constant$$

DBDT的改进之XGBoost算法介绍

损失函数的演变

$$Obj^{(t)} \approx \sum_{j=1}^{T}\left(\left(\sum_{i\in I_j} g_i\right)w_j + \frac{1}{2}\left(\sum_{i\in I_j}(h_i+\lambda)\right)w_j^2\right) + \gamma T + constant$$

$$\approx \sum_{j=1}^{T}\left(\left(\sum_{i\in I_j} g_i\right)w_j + \frac{1}{2}\left(\sum_{i\in I_j}(h_i+\lambda)\right)w_j^2\right) + \gamma T$$

$$\approx \sum_{j=1}^{T}\left(G_j w_j + \frac{1}{2}(H_j+\lambda)w_j^2\right) + \gamma T$$

其中，$G_j = \sum_{i\in I_j} g_i$；$H_j = \sum_{i\in I_j} h_i$。它们分别表示所有属于叶子节点$j$的样本点对应的$g_i$之和以及$h_i$之和。所以，最终是寻找一个合理的$f_t$，使得式子$\sum_{j=1}^{T}\left(G_j w_j + \frac{1}{2}(H_j+\lambda)w_j^2\right) + \gamma T$尽可能大地减小。

DBDT的改进之XGBoost算法介绍

损失函数的演变

求偏导，令导函数为0

$$\frac{\partial Obj^{(t)}}{\partial w_j} = G_j + (H_j + \lambda)w_j = 0$$

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

所以，将w_j的值导入到目标函数$Obj^{(t)}$中，可得：

$$J(f_t) = \sum_{j=1}^{T}\left(G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2\right) + \gamma T$$

$$= -\frac{1}{2}\sum_{j=1}^{T}\left(\frac{{G_j}^2}{H_j + \lambda}\right) + \gamma T$$

DBDT的改进之XGBoost算法介绍

损失函数的演变

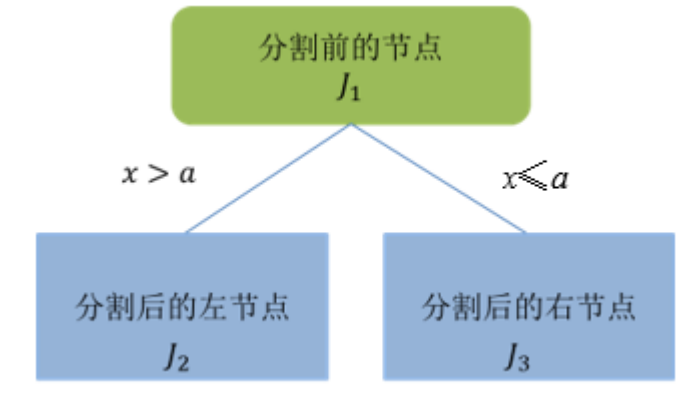
为了能够寻找到最佳的树结构q（即最佳的基础模型f_t），通常会选择贪心法，即在某个已有的可划分节点中加入一个分割，并通过计算分割前后的增益值决定是否剪枝。有关增益值的计算如下：

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

其中，G_L和H_L为某节点分割后对应的左支样本点的导函数值，G_R和H_R为某节点分割后对应的右支样本点的导函数值。这里的增益值$Gain$其实就是将某节点分割为另外两个节点后对应的目标值$J(f_t)$的减少量。

DBDT的改进之XGBoost算法介绍

损失函数的演变



其中，J_1表示某个可分割节点在分割前的目标函数值，J_2和J_3则代表该节点按照某个变量x在a处的分割后对应的目标函数值。按照目标函数的公式，可以将这三个值表示为下方的式子：

$$\begin{cases} J_1 = -\frac{1}{2}\left(\frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right) + \gamma \\ J_2 = -\frac{1}{2}\left(\frac{{G_L}^2}{H_L + \lambda}\right) + \gamma \\ J_3 = -\frac{1}{2}\left(\frac{{G_R}^2}{H_R + \lambda}\right) + \gamma \end{cases}$$

XGBoost算法的应用实战

XGBoost函数介绍

XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
objective='binary:logistic', booster='gbtree', n_jobs=1, nthread=None,
gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
scale_pos_weight=1, base_score=0.5, random_state=0, seed=None, missing=None)

max_depth：用于指定每个基础模型所包含的最大深度，默认为3层。
learning_rate：用于指定模型迭代的学习率或步长，默认为0.1，即对应的梯度提升模型$F_T(x)$可以表示为$F_T(x) = F_{T-1}(x) + \upsilon f_t(x)$：，其中的$\upsilon$就是该参数的指定值，默认值为1；对于较小的学习率$\upsilon$而言，则需要迭代更多次的基础分类器，通常情况下需要利用交叉验证法确定合理的基础模型的个数和学习率。
n_estimators：用于指定基础模型的数量，默认为100个。
silent：bool类型参数，是否输出算法运行过程中的日志信息，默认为True。
booster：用于指定基础模型的类型，默认为'gbtree'，即CART模型，也可以是'gblinear'，表示基础模型为线性模型。

XGBoost算法的应用实战

XGBoost函数介绍

```
XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
              objective='binary:logistic', booster='gbtree', n_jobs=1, nthread=None,
              gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
              colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
              scale_pos_weight=1, base_score=0.5, random_state=0, seed=None, missing=None)
```

objective：用于指定目标函数中的损失函数类型，对于分类型的XGBoost算法，默认的损失函数为二分类的Logistic损失（模型返回概率值），也可以是'multi:softmax',表示用于处理多分类的损失函数（模型返回类别值），还可以是'multi:softprob'，与'multi:softmax'相同，所不同的是模型返回各类别对应的概率值；对于预测型的XGBoost算法，默认的损失函数为线性回归损失。
n_jobs：用于指定XGBoost算法在并行计算时所需的CPU数量，默认为1表示仅使用1个CPU运行算法，即不使用并行运算功能。
nthread：用于指定XGBoost算法在运行时所使用的线程数，默认为None，表示使用计算机最大可能的线程数。
gamma：用于指定节点分割所需的最小损失函数下降值，即增益值Gain的阈值，默认为0。

XGBoost算法的应用实战

XGBoost函数介绍

```
XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
              objective='binary:logistic', booster='gbtree', n_jobs=1, nthread=None,
              gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
              colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
              scale_pos_weight=1, base_score=0.5, random_state=0, seed=None, missing=None)
```

min_child_weight：用于指定叶子节点中各样本点二阶导之和的最小值，即H_j的最小值，默认为1，该参数的值越小，模型越容易过拟合。
max_delta_step：用于指定模型在更新过程中的步长，如果为0，表示没有约束；如果取值为某个较小的正数，就会导致模型更加保守。
subsample：用于指定构建基础模型所使用的抽样比例，默认为1，表示使用原始数据构建每一个基础模型；当抽样比例小于1时，表示构建随机梯度提升树模型，通常会导致模型的方差降低，偏差提高。
colsample_bytree：用于指定每个基础模型所需的采样字段比例，默认为1，表示使用原始数据的所有字段。
colsample_bylevel：用于指定每个基础模型在节点分割时所需的采样字段比例，默认为1，表示使用原始数据的所有字段。

XGBoost算法的应用实战

XGBoost函数介绍

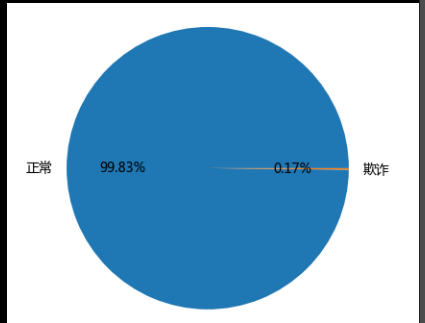
```
XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, silent=True,
              objective='binary:logistic', booster='gbtree', n_jobs=1, nthread=None,
              gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
              colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
              scale_pos_weight=1, base_score=0.5, random_state=0, seed=None, missing=None)
```

reg_alpha：用于指定L1正则项的系数，默认为0。
reg_lambda：用于指定L2正则项的系数，默认为1。
scale_pos_weight：当各类别样本的比例十分不平衡时，通过设定该参数设定为一个正值，可以使算法更快收敛。
base_score：用于指定所有样本的初始化预测得分，默认为0.5。
random_state：用于指定随机数生成器的种子，默认为0，表示使用默认的随机数生成器。
seed：同random_state参数。
missing：用于指定缺失值的表示方法，默认为None，表示NaN即为默认值。

算法的应用实战

信用卡欺诈行为的识别

```
# 读入数据
creditcard = pd.read_csv(r'C:\Users\Administrator\Desktop\creditcard.csv')
# 为确保绘制的饼图为圆形，需执行如下代码
plt.axes(aspect = 'equal')
# 统计交易是否为欺诈的频数
counts = creditcard.Class.value_counts()
# 绘制饼图
plt.pie(x = counts, # 绘图数据
        labels=pd.Series(counts.index).map({0:'正常',1:'欺诈'}), # 添加文字标签
        autopct='%.2f%%' # 设置百分比的格式，这里保留两位小数
       )
# 显示图形
plt.show()
```



算法的应用实战

信用卡欺诈行为的识别

```
# 运用SMOTE算法实现训练数据集的平衡
over_samples = SMOTE(random_state=1234)
over_samples_X,over_samples_y = over_samples.fit_sample(X_train, y_train)
# 重抽样前的类别比例
print(y_train.value_counts()/len(y_train))
# 重抽样后的类别比例
print(pd.Series(over_samples_y).value_counts()/len(over_samples_y))
```

out:

```
0    0.998249
1    0.001751
Name: Class, dtype: float64
1    0.5
0    0.5
```

算法的应用实战

信用卡欺诈行为的识别

```
# 构建XGBoost分类器
xgboost = xgboost.XGBClassifier()
# 使用重抽样后的数据，对其建模
xgboost.fit(over_samples_X,over_samples_y)
# 将模型运用到测试数据集中
resample_pred = xgboost.predict(np.array(X_test))
```

算法的应用实战

信用卡欺诈行为的识别

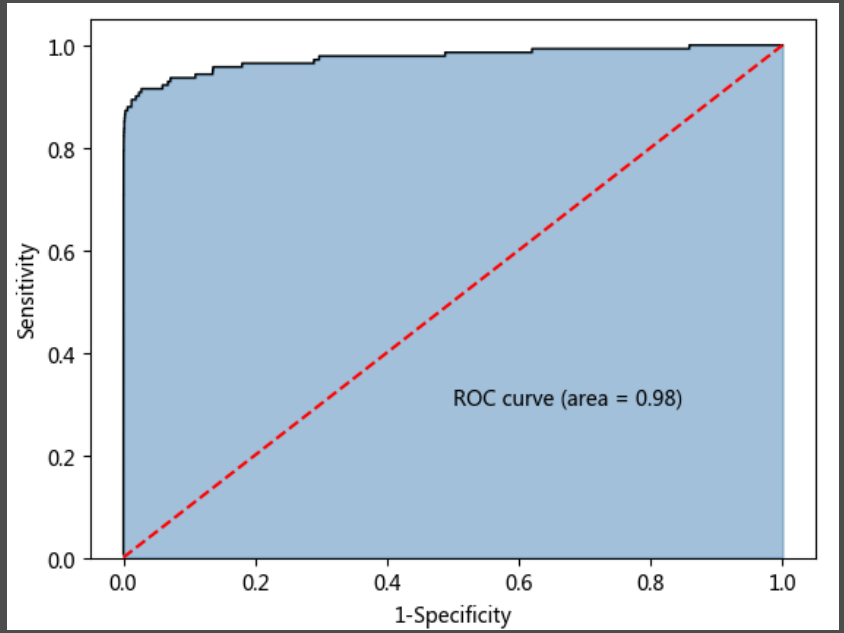
```
# 返回模型的预测效果
print('模型的准确率为：\n',metrics.accuracy_score(y_test, resample_pred))
print('模型的评估报告：\n',metrics.classification_report(y_test, resample_pred))
```

out:

```
模型的准确率为：
 0.990145477102
模型的评估报告：
                precision    recall      f1-score   support
          0          1.00      0.99      1.00       85302
          1          0.13      0.88      0.23       141
  avg / total        1.00      0.99      0.99       85443
```

算法的应用实战

信用卡欺诈行为的识别



算法的应用实战

信用卡欺诈行为的识别

```
# 构建XGBoost分类器
xgboost2 = xgboost.XGBClassifier()
# 使用非平衡的训练数据集拟合模型
xgboost2.fit(X_train,y_train)
# 基于拟合的模型对测试数据集进行预测
pred2 = xgboost2.predict(X_test)
```

算法的应用实战

信用卡欺诈行为的识别

```
# 返回模型的预测效果
print('模型的准确率为：\n',metrics.accuracy_score(y_test, pred2))
print('模型的评估报告：\n',metrics.classification_report(y_test, pred2))
```

out:

```
模型的准确率为：
 0.999403110846
模型的评估报告：
                precision    recall  f1-score   support
          0       1.00      1.00      1.00     85302
          1       0.88      0.74      0.80       141
avg / total       1.00      1.00      1.00     85443
```

算法的应用实战

信用卡欺诈行为的识别