****一、梯度迭代树****

**[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "http://blog.csdn.net/liulingyuan6/article/details/_blank)简介：**

        梯度提升树是一种决策树的集成算法。它通过反复迭代训练决策树来最小化损失函数。决策树类似，梯度提升树具有可处理类别特征、易扩展到多分类问题、不需特征缩放等性质。**[Spark](http://lib.csdn.net/base/spark" \o "Apache Spark知识库" \t "http://blog.csdn.net/liulingyuan6/article/details/_blank)**.ml通过使用现有[decision tree](http://spark.apache.org/docs/latest/mllib-decision-tree.html" \t "http://blog.csdn.net/liulingyuan6/article/details/_blank)工具来实现。

       梯度提升树依次迭代训练一系列的决策树。在一次迭代中，算法使用现有的集成来对每个训练实例的类别进行预测，然后将预测结果与真实的标签值进行比较。通过重新标记，来赋予预测结果不好的实例更高的权重。所以，在下次迭代中，决策树会对先前的错误进行修正。

       对实例标签进行重新标记的机制由损失函数来指定。每次迭代过程中，梯度迭代树在训练数据上进一步减少损失函数的值。spark.ml为分类问题提供一种损失函数（Log Loss），为回归问题提供两种损失函数（平方误差与绝对误差）。

       Spark.ml支持二分类以及回归的随机森林算法，适用于连续特征以及类别特征。

＊注意梯度提升树目前不支持多分类问题。

**参数：**

checkpointInterval:

类型：整数型。

含义：设置检查点间隔（>=1），或不设置检查点（-1）。

featuresCol:

类型：字符串型。

含义：特征列名。

impurity:

类型：字符串型。

含义：计算信息增益的准则（不区分大小写）。

labelCol:

类型：字符串型。

含义：标签列名。

lossType:

类型：字符串型。

含义：损失函数类型。

maxBins:

类型：整数型。

含义：连续特征离散化的最大数量，以及选择每个节点分裂特征的方式。

maxDepth:

类型：整数型。

含义：树的最大深度（>=0）。

maxIter:

类型：整数型。

含义：迭代次数（>=0）。

minInfoGain:

类型：双精度型。

含义：分裂节点时所需最小信息增益。

minInstancesPerNode:

类型：整数型。

含义：分裂后自节点最少包含的实例数量。

predictionCol:

类型：字符串型。

含义：预测结果列名。

rawPredictionCol:

类型：字符串型。

含义：原始预测。

seed:

类型：长整型。

含义：随机种子。

subsamplingRate:

类型：双精度型。

含义：学习一棵决策树使用的训练数据比例，范围[0,1]。

stepSize:

类型：双精度型。

含义：每次迭代优化步长。

1. **from** pyspark.ml **import** Pipeline
2. **from** pyspark.ml.classification **import** GBTClassifier
3. **from** pyspark.ml.feature **import** StringIndexer, VectorIndexer
4. **from** pyspark.ml.evaluation **import** MulticlassClassificationEvaluator
6. # Load and parse the data file, converting it to a DataFrame.
7. data = spark.read.format("libsvm").load("data/mllib/sample\_libsvm\_data.txt")
9. # Index labels, adding metadata to the label column.
10. # Fit on whole dataset to include all labels in index.
11. labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
12. # Automatically identify categorical features, and index them.
13. # Set maxCategories so features with > 4 distinct values are treated as continuous.
14. featureIndexer =\
15. VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)
17. # Split the data into training and test sets (30% held out for testing)
18. (trainingData, testData) = data.randomSplit([0.7, 0.3])
20. # Train a GBT model.
21. gbt = GBTClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", maxIter=10)
23. # Chain indexers and GBT in a Pipeline
24. pipeline = Pipeline(stages=[labelIndexer, featureIndexer, gbt])
26. # Train model.  This also runs the indexers.
27. model = pipeline.fit(trainingData)
29. # Make predictions.
30. predictions = model.transform(testData)
32. # Select example rows to display.
33. predictions.select("prediction", "indexedLabel", "features").show(5)
35. # Select (prediction, true label) and compute test error
36. evaluator = MulticlassClassificationEvaluator(
37. labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
38. accuracy = evaluator.evaluate(predictions)
39. **print**("Test Error = %g" % (1.0 - accuracy))
41. gbtModel = model.stages[2]
42. **print**(gbtModel)  # summary only

**二、GBDT源码分析之一 ：总览**

96 作者 cathyxlyl 关注

2017.04.24 23:20\* 字数 2013 阅读 1240评论 0喜欢 13

0x00 前言

这个系列将会对python的scikit-learn算法包中GBDT算法的源码实现做一个深入梳理和解读。本文会首先对GBDT算法做一个简单的介绍，并对其源码的结构做一个整体上的梳理。因为这里偏重的是源码分析，所以如果想对GBDT算法本身的原理进行深入了解，可以阅读参考文献中推荐的几位大牛的文章。

文章结构

本文将分为下面几个部分：

简要介绍一下GBDT算法的基本概念。

scikit-learn中GBDT算法的运行例子。

对GBDT源码结构的一个整体梳理。这里我们会通过思维导图的方式展现GBDT算法实现涉及的主要源码构成。

0x01 GBDT简介

GBDT(Gradient Boosting Decision Tree) 又称 MART(Multiple Additive Regression Tree)或GBRT(Gradient Boosting Regression Tree)，是一种基于回归决策树的Boosting集成算法。

GBDT的核心从算法命名来看一目了然，即决策树（DT）和梯度提升（GB）。

决策树

决策树是一种十分常用和基础的监督学习算法，可适用于分类和回归问题；它将决策过程表述为树状结构，树中的不同路径代表不同的决策分支。决策树的构建过程由根节点出发，根据样本的属性（特征）不断将样本集分裂生成子节点，直至满足停止条件；树结构的每个叶子节点都代表一个最终的预测结果，一般取落入该叶子节点的样本的众数/概率分布/平均值等。由于通过决策树算法生成的模型可以由一系列if-then规则表述，因此非常易于理解和实现，也是最简单的非线性算法之一。

决策树的关键技术包括分裂点的选择、分裂停止的条件以及避免过拟合的方法（如剪枝；合适的分裂停止条件也可以防止过拟合）。经典的决策树算法包括ID3、C4.5、CART等。

回归树

回归树即用来解决回归问题的决策树。在分类树中，样本标签是离散的或非有序的，我们取叶子节点样本标签的众数或概率分布作为预测结果；而在回归树中，样本标签一般是连续性的有序数据，我们取叶子结点中所有样本标签的平均值作为预测结果。

集成方法（Ensemble Method）

集成学习方法是将多个弱模型通过一定的组合方式组成一个新的强模型的方法，一般情况下集成的模型具有更强的预测和泛化能力。在机器学习问题中，这是一种非常强大的思路，也是"集体智慧"的典型例子。集成算法中的弱模型又称元算法；在GBDT中，回归树是GBDT的元算法。

我们在理解集成方法时，可以更多将其看作一些学习框架，重点在于理解这些框架的思路。各种集成算法（如GBDT、随机森林）的核心也可理解为将基本算法（如决策树）带入集成框架（如Boosting、Bagging）的产物。

Boosting与Gradient Boosting

Boosting的意思是"提升"，它关注被预测错误的样本，基于预测错误的部分构建新的弱模型并集成，是一种常用的迭代集成方法。原始的Boosting方法可以说是基于"样本"的，它会在一开始给所有样本附上相等的权重值，在每轮迭代（生成一个弱模型）后增加预测错误的样本的权重，减少预测正确的样本的权重，并在此基础上训练新的弱模型；最终通过加权或投票的形式对所有弱模型进行组合，生成强模型。

而Gradient Boosting和原始Boosting方法不同的地方在于，它在残差减少的梯度方向建立新的弱模型。直观上看，它用来训练第K轮弱模型的数据，来自于之前所有弱模型集成后的预测值和样本真实值的"差"（准确来说损失函数梯度减少的方向）。

基于上面描述的一系列概念，我们可以较为容易的理解：一个GBDT模型由多颗回归决策树组成；理论上在训练过程中的一轮迭代中，算法基于残差减少的梯度方向生成一颗决策树（scikit-learn在用GBDT解决多标签问题时，实际上在每一轮迭代中用了多棵回归树，本文中我们不对这种情况做深入说明）。在预测阶段，累加模型中所有决策树的预测值(乘上步长/学习率)，即可计算整个模型的预测结果。

GBDT算法在实际生产中运用非常广泛，表达能力也很强，通常不需要复杂的特征工程就能得到较好的预测效果，还能输出特征重要性得分；同时通过设定合理的样本和特征抽样比例，可以在训练过程中实现交叉检验（cross validation），有效地减少模型过拟合的出现。缺点则是基于Boosting集成方法的算法较难实现并行化，且基于GBDT的模型会较为复杂，深入分析和调优会有一定困难性。

0x02 运行示例

scikit-learn中ensemble包下关于GBDT的算法有两个，分别用来解决回归问题GradientBoostingRegressor和分类问题GradientBoostingClassifier，调用起来十分简单。

回归示例（波士顿房价数据集）

from sklearn.ensemble import GradientBoostingRegressor

from sklearn.datasets import load\_boston

from sklearn.metrics import mean\_squared\_error

from sklearn.model\_selection import train\_test\_split

# 导入数据

X\_train, X\_test, y\_train, y\_test = train\_test\_split(load\_boston().data, load\_boston().target, test\_size=0.2)

"""初始化算法，设置参数

一些主要参数

loss: 损失函数，GBDT回归器可选'ls', 'lad', 'huber', 'quantile'。

learning\_rate: 学习率/步长。

n\_estimators: 迭代次数，和learning\_rate存在trade-off关系。

criterion: 衡量分裂质量的公式，一般默认即可。

subsample: 样本采样比例。

max\_features: 最大特征数或比例。

决策树相关参数包括max\_depth, min\_samples\_split, min\_samples\_leaf, min\_weight\_fraction\_leaf, max\_leaf\_nodes, min\_impurity\_split, 多数用来设定决策树分裂停止条件。

verbose: 日志level。

具体说明和其它参数请参考官网API。

"""

reg\_model = GradientBoostingRegressor(

loss='ls',

learning\_rate=0.02,

n\_estimators=200,

subsample=0.8,

max\_features=0.8,

max\_depth=3,

verbose=2

)

# 训练模型

reg\_model.fit(X\_train, y\_train)

# 评估模型

prediction\_train = reg\_model.predict(X\_train)

rmse\_train = mean\_squared\_error(y\_train, prediction\_train)

prediction\_test = reg\_model.predict(X\_test)

rmse\_test = mean\_squared\_error(y\_test, prediction\_test)

print "RMSE for training dataset is %f, for testing dataset is %f." % (rmse\_train, rmse\_test)

"""Output:

RMSE for training dataset is 4.239157, for testing dataset is 10.749044.

"""

分类示例（鸢尾花分类数据集）

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.datasets import load\_iris

from sklearn.metrics import confusion\_matrix

from sklearn.model\_selection import train\_test\_split

# 导入数据

X\_train, X\_test, y\_train, y\_test = train\_test\_split(load\_iris().data, load\_iris().target, test\_size=0.2)

"""初始化算法，设置参数

一些主要参数

loss: 损失函数，GBDT分类器可选'deviance', 'exponential'。

learning\_rate: 学习率/步长。

n\_estimators: 迭代次数，和learning\_rate存在trade-off关系。

criterion: 衡量分裂质量的公式，一般默认即可。

subsample: 样本采样比例。

max\_features: 最大特征数或比例。

决策树相关参数包括max\_depth, min\_samples\_split, min\_samples\_leaf, min\_weight\_fraction\_leaf, max\_leaf\_nodes, min\_impurity\_split, 多数用来设定决策树分裂停止条件。

verbose: 日志level。

具体说明和其它参数请参考官网API。

"""

clf\_model = GradientBoostingClassifier(

loss='deviance',

learning\_rate=0.01,

n\_estimators=50,

subsample=0.8,

max\_features=1,

max\_depth=3,

verbose=2

)

# 训练模型

clf\_model.fit(X\_train, y\_train)

# 评估模型

prediction\_train = clf\_model.predict(X\_train)

cm\_train = confusion\_matrix(y\_train, prediction\_train)

prediction\_test = clf\_model.predict(X\_test)

cm\_test = confusion\_matrix(y\_test, prediction\_test)

print "Confusion matrix for training dataset is \n%s\n for testing dataset is \n%s." % (cm\_train, cm\_test)

"""Output:

Confusion matrix for training dataset is

[[40 0 0]

[ 0 40 1]

[ 0 1 38]]

for testing dataset is

[[10 0 0]

[ 0 8 1]

[ 0 0 11]].

"""

0x03 源码总览

整体介绍

Python的scikit-learn包包含了我们常用的大部分的机器学习算法和数据处理方法，我们主要分析其中实现GBDT的源码。GBDT的实现源码依然可以被分为GB和DT两部分。其中DT为决策树部分，其源码在一个名为Tree的package下；GB为gradient boosting方法，其相关源码在一个名为Ensemble的package下。总体结构见下面的思维导图。

Tree包的源码结构截图如下。里面实现了决策树算法、决策树的基本数据结构Tree、决策树构建策略以及树的可视化等内容。

Ensemble包的源码结构截图如下。Ensemble包里还包含了如bagging、随机森林等其它主题，但我们主要关注其中的base.py和grandient\_boosting.py文件。

# **[scikit-learn 梯度提升树(GBDT)调参小结](http://www.cnblogs.com/DjangoBlog/p/6201663.html)**

1. scikit-learn GBDT类库概述

　　　　在sacikit-learn中，GradientBoostingClassifier为GBDT的分类类， 而GradientBoostingRegressor为GBDT的回归类。两者的参数类型完全相同，当然有些参数比如损失函数loss的可选择项并不相同。这些参数中，类似于Adaboost，我们把重要参数分为两类，第一类是Boosting框架的重要参数，第二类是弱学习器即CART回归树的重要参数。

　　　　下面我们就从这两个方面来介绍这些参数的使用。

2. GBDT类库boosting框架参数

　　　　首先，我们来看boosting框架相关的重要参数。由于GradientBoostingClassifier和GradientBoostingRegressor的参数绝大部分相同，我们下面会一起来讲，不同点会单独指出。

　　　　1) n\_estimators: 也就是弱学习器的最大迭代次数，或者说最大的弱学习器的个数。一般来说n\_estimators太小，容易欠拟合，n\_estimators太大，又容易过拟合，一般选择一个适中的数值。默认是100。在实际调参的过程中，我们常常将n\_estimators和下面介绍的参数learning\_rate一起考虑。

　　　　2) learning\_rate: 即每个弱学习器的权重缩减系数νν，也称作步长，在原理篇的正则化章节我们也讲到了，加上了正则化项，我们的强学习器的迭代公式为fk(x)=fk−1(x)+νhk(x)fk(x)=fk−1(x)+νhk(x)。νν的取值范围为0<ν≤10<ν≤1。对于同样的训练集拟合效果，较小的νν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。所以这两个参数n\_estimators和learning\_rate要一起调参。一般来说，可以从一个小一点的νν开始调参，默认是1。

　　　　3) subsample: 即我们在原理篇的正则化章节讲到的子采样，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，默认是1.0，即不使用子采样。

　　　　4) init: 即我们的初始化的时候的弱学习器，拟合对应原理篇里面的f0(x)f0(x)，如果不输入，则用训练集样本来做样本集的初始化分类回归预测。否则用init参数提供的学习器做初始化分类回归预测。一般用在我们对数据有先验知识，或者之前做过一些拟合的时候，如果没有的话就不用管这个参数了。

　　　　5) loss: 即我们GBDT算法中的损失函数。分类模型和回归模型的损失函数是不一样的。

　　　　　　对于分类模型，有对数似然损失函数"deviance"和指数损失函数"exponential"两者输入选择。默认是对数似然损失函数"deviance"。在原理篇中对这些分类损失函数有详细的介绍。一般来说，推荐使用默认的"deviance"。它对二元分离和多元分类各自都有比较好的优化。而指数损失函数等于把我们带到了Adaboost算法。

　　　　　　对于回归模型，有均方差"ls", 绝对损失"lad", Huber损失"huber"和分位数损失“quantile”。默认是均方差"ls"。一般来说，如果数据的噪音点不多，用默认的均方差"ls"比较好。如果是噪音点较多，则推荐用抗噪音的损失函数"huber"。而如果我们需要对训练集进行分段预测的时候，则采用“quantile”。

　　　　6) alpha：这个参数只有GradientBoostingRegressor有，当我们使用Huber损失"huber"和分位数损失“quantile”时，需要指定分位数的值。默认是0.9，如果噪音点较多，可以适当降低这个分位数的值。

3. GBDT类库弱学习器参数

　　　　这里我们再对GBDT的类库弱学习器的重要参数做一个总结。由于GBDT使用了CART回归决策树，因此它的参数基本来源于决策树类，也就是说，和DecisionTreeClassifier和DecisionTreeRegressor的参数基本类似。如果你已经很熟悉决策树算法的调参，那么这一节基本可以跳过。不熟悉的朋友可以继续看下去。

　　　　1) 划分时考虑的最大特征数max\_features: 可以使用很多种类型的值，默认是"None",意味着划分时考虑所有的特征数；如果是"log2"意味着划分时最多考虑log2Nlog2N个特征；如果是"sqrt"或者"auto"意味着划分时最多考虑N−−√N个特征。如果是整数，代表考虑的特征绝对数。如果是浮点数，代表考虑特征百分比，即考虑（百分比xN）取整后的特征数。其中N为样本总特征数。一般来说，如果样本特征数不多，比如小于50，我们用默认的"None"就可以了，如果特征数非常多，我们可以灵活使用刚才描述的其他取值来控制划分时考虑的最大特征数，以控制决策树的生成时间。

　　　　2) 决策树最大深度max\_depth: 默认可以不输入，如果不输入的话，决策树在建立子树的时候不会限制子树的深度。一般来说，数据少或者特征少的时候可以不管这个值。如果模型样本量多，特征也多的情况下，推荐限制这个最大深度，具体的取值取决于数据的分布。常用的可以取值10-100之间。

　　　　3) 内部节点再划分所需最小样本数min\_samples\_split: 这个值限制了子树继续划分的条件，如果某节点的样本数少于min\_samples\_split，则不会继续再尝试选择最优特征来进行划分。 默认是2.如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。

　　　　4) 叶子节点最少样本数min\_samples\_leaf: 这个值限制了叶子节点最少的样本数，如果某叶子节点数目小于样本数，则会和兄弟节点一起被剪枝。 默认是1,可以输入最少的样本数的整数，或者最少样本数占样本总数的百分比。如果样本量不大，不需要管这个值。如果样本量数量级非常大，则推荐增大这个值。

　　　　5）叶子节点最小的样本权重和min\_weight\_fraction\_leaf：这个值限制了叶子节点所有样本权重和的最小值，如果小于这个值，则会和兄弟节点一起被剪枝。 默认是0，就是不考虑权重问题。一般来说，如果我们有较多样本有缺失值，或者分类树样本的分布类别偏差很大，就会引入样本权重，这时我们就要注意这个值了。

　　　　6) 最大叶子节点数max\_leaf\_nodes: 通过限制最大叶子节点数，可以防止过拟合，默认是"None”，即不限制最大的叶子节点数。如果加了限制，算法会建立在最大叶子节点数内最优的决策树。如果特征不多，可以不考虑这个值，但是如果特征分成多的话，可以加以限制，具体的值可以通过交叉验证得到。

　　　　7) 节点划分最小不纯度min\_impurity\_split: 这个值限制了决策树的增长，如果某节点的不纯度(基于基尼系数，均方差)小于这个阈值，则该节点不再生成子节点。即为叶子节点 。一般不推荐改动默认值1e-7。

4. GBDT调参实例

　　　　这里我们用一个二元分类的例子来讲解下GBDT的调参。这部分参考了这个Github上的数据调参过程Parameter\_Tuning\_GBM\_with\_Example。这个例子的数据有87000多行，单机跑会比较慢，下面的例子我只选择了它的前面20000行，我将其打包后，下载地址在这。

　　　　首先，我们载入需要的类库：

复制代码

复制代码

import pandas as pd

import numpy as np

from sklearn.ensemble import GradientBoostingClassifier

from sklearn import cross\_validation, metrics

from sklearn.grid\_search import GridSearchCV

import matplotlib.pylab as plt

%matplotlib inline

复制代码

复制代码

　　　　接着，我们把解压的数据用下面的代码载入，顺便看看数据的类别分布。

train = pd.read\_csv('train\_modified.csv')

target='Disbursed' # Disbursed的值就是二元分类的输出

IDcol = 'ID'

train['Disbursed'].value\_counts()

　　　　可以看到类别输出如下，也就是类别0的占大多数。

0 19680

1 320

Name: Disbursed, dtype: int64

　　　　现在我们得到我们的训练集。最后一列Disbursed是分类输出。前面的所有列（不考虑ID列）都是样本特征。

x\_columns = [x for x in train.columns if x not in [target, IDcol]]

X = train[x\_columns]

y = train['Disbursed']

　　　　不管任何参数，都用默认的，我们拟合下数据看看：

复制代码

复制代码

gbm0 = GradientBoostingClassifier(random\_state=10)

gbm0.fit(X,y)

y\_pred = gbm0.predict(X)

y\_predprob = gbm0.predict\_proba(X)[:,1]

print "Accuracy : %.4g" % metrics.accuracy\_score(y.values, y\_pred)

print "AUC Score (Train): %f" % metrics.roc\_auc\_score(y, y\_predprob)

复制代码

复制代码

　　　 输出如下，可见拟合还可以，我们下面看看怎么通过调参提高模型的泛化能力。

Accuracy : 0.9852

AUC Score (Train): 0.900531

　　　首先我们从步长(learning rate)和迭代次数(n\_estimators)入手。一般来说,开始选择一个较小的步长来网格搜索最好的迭代次数。这里，我们将步长初始值设置为0.1。对于迭代次数进行网格搜索如下：

复制代码

复制代码

param\_test1 = {'n\_estimators':range(20,81,10)}

gsearch1 = GridSearchCV(estimator = GradientBoostingClassifier(learning\_rate=0.1, min\_samples\_split=300,

min\_samples\_leaf=20,max\_depth=8,max\_features='sqrt', subsample=0.8,random\_state=10),

param\_grid = param\_test1, scoring='roc\_auc',iid=False,cv=5)

gsearch1.fit(X,y)

gsearch1.grid\_scores\_, gsearch1.best\_params\_, gsearch1.best\_score\_

复制代码

复制代码

　　　　输出如下，可见最好的迭代次数是60。

([mean: 0.81285, std: 0.01967, params: {'n\_estimators': 20},

mean: 0.81438, std: 0.01947, params: {'n\_estimators': 30},

mean: 0.81451, std: 0.01933, params: {'n\_estimators': 40},

mean: 0.81618, std: 0.01848, params: {'n\_estimators': 50},

mean: 0.81751, std: 0.01736, params: {'n\_estimators': 60},

mean: 0.81547, std: 0.01900, params: {'n\_estimators': 70},

mean: 0.81299, std: 0.01860, params: {'n\_estimators': 80}],

{'n\_estimators': 60},

0.8175146087398375)

　　　　找到了一个合适的迭代次数，现在我们开始对决策树进行调参。首先我们对决策树最大深度max\_depth和内部节点再划分所需最小样本数min\_samples\_split进行网格搜索。

复制代码

复制代码

param\_test2 = {'max\_depth':range(3,14,2), 'min\_samples\_split':range(100,801,200)}

gsearch2 = GridSearchCV(estimator = GradientBoostingClassifier(learning\_rate=0.1, n\_estimators=60, min\_samples\_leaf=20,

max\_features='sqrt', subsample=0.8, random\_state=10),

param\_grid = param\_test2, scoring='roc\_auc',iid=False, cv=5)

gsearch2.fit(X,y)

gsearch2.grid\_scores\_, gsearch2.best\_params\_, gsearch2.best\_score\_

复制代码

复制代码

　　　　输出如下，可见最好的最大树深度是7，内部节点再划分所需最小样本数是300。

([mean: 0.81199, std: 0.02073, params: {'min\_samples\_split': 100, 'max\_depth': 3},

mean: 0.81267, std: 0.01985, params: {'min\_samples\_split': 300, 'max\_depth': 3},

mean: 0.81238, std: 0.01937, params: {'min\_samples\_split': 500, 'max\_depth': 3},

mean: 0.80925, std: 0.02051, params: {'min\_samples\_split': 700, 'max\_depth': 3},

mean: 0.81846, std: 0.01843, params: {'min\_samples\_split': 100, 'max\_depth': 5},

mean: 0.81630, std: 0.01810, params: {'min\_samples\_split': 300, 'max\_depth': 5},

mean: 0.81315, std: 0.01898, params: {'min\_samples\_split': 500, 'max\_depth': 5},

mean: 0.81262, std: 0.02090, params: {'min\_samples\_split': 700, 'max\_depth': 5},

mean: 0.81807, std: 0.02004, params: {'min\_samples\_split': 100, 'max\_depth': 7},

mean: 0.82137, std: 0.01733, params: {'min\_samples\_split': 300, 'max\_depth': 7},

mean: 0.81703, std: 0.01773, params: {'min\_samples\_split': 500, 'max\_depth': 7},

mean: 0.81383, std: 0.02327, params: {'min\_samples\_split': 700, 'max\_depth': 7},

mean: 0.81107, std: 0.02178, params: {'min\_samples\_split': 100, 'max\_depth': 9},

mean: 0.80944, std: 0.02612, params: {'min\_samples\_split': 300, 'max\_depth': 9},

mean: 0.81476, std: 0.01973, params: {'min\_samples\_split': 500, 'max\_depth': 9},

mean: 0.81601, std: 0.02576, params: {'min\_samples\_split': 700, 'max\_depth': 9},

mean: 0.81091, std: 0.02227, params: {'min\_samples\_split': 100, 'max\_depth': 11},

mean: 0.81309, std: 0.02696, params: {'min\_samples\_split': 300, 'max\_depth': 11},

mean: 0.81713, std: 0.02379, params: {'min\_samples\_split': 500, 'max\_depth': 11},

mean: 0.81347, std: 0.02702, params: {'min\_samples\_split': 700, 'max\_depth': 11},

mean: 0.81444, std: 0.01813, params: {'min\_samples\_split': 100, 'max\_depth': 13},

mean: 0.80825, std: 0.02291, params: {'min\_samples\_split': 300, 'max\_depth': 13},

mean: 0.81923, std: 0.01693, params: {'min\_samples\_split': 500, 'max\_depth': 13},

mean: 0.81382, std: 0.02258, params: {'min\_samples\_split': 700, 'max\_depth': 13}],

{'max\_depth': 7, 'min\_samples\_split': 300},

0.8213724275914632)

　　　　由于决策树深度7是一个比较合理的值，我们把它定下来，对于内部节点再划分所需最小样本数min\_samples\_split，我们暂时不能一起定下来，因为这个还和决策树其他的参数存在关联。下面我们再对内部节点再划分所需最小样本数min\_samples\_split和叶子节点最少样本数min\_samples\_leaf一起调参。

复制代码

复制代码

param\_test3 = {'min\_samples\_split':range(800,1900,200), 'min\_samples\_leaf':range(60,101,10)}

gsearch3 = GridSearchCV(estimator = GradientBoostingClassifier(learning\_rate=0.1, n\_estimators=60,max\_depth=7,

max\_features='sqrt', subsample=0.8, random\_state=10),

param\_grid = param\_test3, scoring='roc\_auc',iid=False, cv=5)

gsearch3.fit(X,y)

gsearch3.grid\_scores\_, gsearch3.best\_params\_, gsearch3.best\_score\_

复制代码

复制代码

输出结果如下，可见这个min\_samples\_split在边界值，还有进一步调试小于边界60的必要。由于这里只是例子，所以大家可以自己下来用包含小于60的网格搜索来寻找合适的值。

([mean: 0.81828, std: 0.02251, params: {'min\_samples\_split': 800, 'min\_samples\_leaf': 60},

mean: 0.81731, std: 0.02344, params: {'min\_samples\_split': 1000, 'min\_samples\_leaf': 60},

mean: 0.82220, std: 0.02250, params: {'min\_samples\_split': 1200, 'min\_samples\_leaf': 60},

mean: 0.81447, std: 0.02125, params: {'min\_samples\_split': 1400, 'min\_samples\_leaf': 60},

mean: 0.81495, std: 0.01626, params: {'min\_samples\_split': 1600, 'min\_samples\_leaf': 60},

mean: 0.81528, std: 0.02140, params: {'min\_samples\_split': 1800, 'min\_samples\_leaf': 60},

mean: 0.81590, std: 0.02517, params: {'min\_samples\_split': 800, 'min\_samples\_leaf': 70},

mean: 0.81573, std: 0.02207, params: {'min\_samples\_split': 1000, 'min\_samples\_leaf': 70},

mean: 0.82021, std: 0.02521, params: {'min\_samples\_split': 1200, 'min\_samples\_leaf': 70},

mean: 0.81512, std: 0.01995, params: {'min\_samples\_split': 1400, 'min\_samples\_leaf': 70},

mean: 0.81395, std: 0.02081, params: {'min\_samples\_split': 1600, 'min\_samples\_leaf': 70},

mean: 0.81587, std: 0.02082, params: {'min\_samples\_split': 1800, 'min\_samples\_leaf': 70},

mean: 0.82064, std: 0.02698, params: {'min\_samples\_split': 800, 'min\_samples\_leaf': 80},

mean: 0.81490, std: 0.02475, params: {'min\_samples\_split': 1000, 'min\_samples\_leaf': 80},

mean: 0.82009, std: 0.02568, params: {'min\_samples\_split': 1200, 'min\_samples\_leaf': 80},

mean: 0.81850, std: 0.02226, params: {'min\_samples\_split': 1400, 'min\_samples\_leaf': 80},

mean: 0.81855, std: 0.02099, params: {'min\_samples\_split': 1600, 'min\_samples\_leaf': 80},

mean: 0.81666, std: 0.02249, params: {'min\_samples\_split': 1800, 'min\_samples\_leaf': 80},

mean: 0.81960, std: 0.02437, params: {'min\_samples\_split': 800, 'min\_samples\_leaf': 90},

mean: 0.81560, std: 0.02235, params: {'min\_samples\_split': 1000, 'min\_samples\_leaf': 90},

mean: 0.81936, std: 0.02542, params: {'min\_samples\_split': 1200, 'min\_samples\_leaf': 90},

mean: 0.81362, std: 0.02254, params: {'min\_samples\_split': 1400, 'min\_samples\_leaf': 90},

mean: 0.81429, std: 0.02417, params: {'min\_samples\_split': 1600, 'min\_samples\_leaf': 90},

mean: 0.81299, std: 0.02262, params: {'min\_samples\_split': 1800, 'min\_samples\_leaf': 90},

mean: 0.82000, std: 0.02511, params: {'min\_samples\_split': 800, 'min\_samples\_leaf': 100},

mean: 0.82209, std: 0.01816, params: {'min\_samples\_split': 1000, 'min\_samples\_leaf': 100},

mean: 0.81821, std: 0.02337, params: {'min\_samples\_split': 1200, 'min\_samples\_leaf': 100},

mean: 0.81922, std: 0.02377, params: {'min\_samples\_split': 1400, 'min\_samples\_leaf': 100},

mean: 0.81545, std: 0.02221, params: {'min\_samples\_split': 1600, 'min\_samples\_leaf': 100},

mean: 0.81704, std: 0.02509, params: {'min\_samples\_split': 1800, 'min\_samples\_leaf': 100}],

{'min\_samples\_leaf': 60, 'min\_samples\_split': 1200},

0.8222032996697154)

　　　　我们调了这么多参数了，终于可以都放到GBDT类里面去看看效果了。现在我们用新参数拟合数据：

复制代码

复制代码

gbm1 = GradientBoostingClassifier(learning\_rate=0.1, n\_estimators=60,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, max\_features='sqrt', subsample=0.8, random\_state=10)

gbm1.fit(X,y)

y\_pred = gbm1.predict(X)

y\_predprob = gbm1.predict\_proba(X)[:,1]

print "Accuracy : %.4g" % metrics.accuracy\_score(y.values, y\_pred)

print "AUC Score (Train): %f" % metrics.roc\_auc\_score(y, y\_predprob)

复制代码

复制代码

　　　　输出如下：

Accuracy : 0.984

AUC Score (Train): 0.908099

　　　　对比我们最开始完全不调参的拟合效果，可见精确度稍有下降，主要原理是我们使用了0.8的子采样，20%的数据没有参与拟合。

　　　　现在我们再对最大特征数max\_features进行网格搜索。

复制代码

复制代码

param\_test4 = {'max\_features':range(7,20,2)}

gsearch4 = GridSearchCV(estimator = GradientBoostingClassifier(learning\_rate=0.1, n\_estimators=60,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, subsample=0.8, random\_state=10),

param\_grid = param\_test4, scoring='roc\_auc',iid=False, cv=5)

gsearch4.fit(X,y)

gsearch4.grid\_scores\_, gsearch4.best\_params\_, gsearch4.best\_score\_

复制代码

复制代码

　　　　输出如下：

([mean: 0.82220, std: 0.02250, params: {'max\_features': 7},

mean: 0.82241, std: 0.02421, params: {'max\_features': 9},

mean: 0.82108, std: 0.02302, params: {'max\_features': 11},

mean: 0.82064, std: 0.01900, params: {'max\_features': 13},

mean: 0.82198, std: 0.01514, params: {'max\_features': 15},

mean: 0.81355, std: 0.02053, params: {'max\_features': 17},

mean: 0.81877, std: 0.01863, params: {'max\_features': 19}],

{'max\_features': 9},

0.822412506351626)

　　　　现在我们再对子采样的比例进行网格搜索：

复制代码

复制代码

param\_test5 = {'subsample':[0.6,0.7,0.75,0.8,0.85,0.9]}

gsearch5 = GridSearchCV(estimator = GradientBoostingClassifier(learning\_rate=0.1, n\_estimators=60,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, max\_features=9, random\_state=10),

param\_grid = param\_test5, scoring='roc\_auc',iid=False, cv=5)

gsearch5.fit(X,y)

gsearch5.grid\_scores\_, gsearch5.best\_params\_, gsearch5.best\_score\_

复制代码

复制代码

　　　　输出如下：

([mean: 0.81828, std: 0.02392, params: {'subsample': 0.6},

mean: 0.82344, std: 0.02708, params: {'subsample': 0.7},

mean: 0.81673, std: 0.02196, params: {'subsample': 0.75},

mean: 0.82241, std: 0.02421, params: {'subsample': 0.8},

mean: 0.82285, std: 0.02446, params: {'subsample': 0.85},

mean: 0.81738, std: 0.02236, params: {'subsample': 0.9}],

{'subsample': 0.7},

0.8234378969766262)

　　　　现在我们基本已经得到我们所有调优的参数结果了。这时我们可以减半步长，最大迭代次数加倍来增加我们模型的泛化能力。再次拟合我们的模型：

复制代码

复制代码

gbm2 = GradientBoostingClassifier(learning\_rate=0.05, n\_estimators=120,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, max\_features=9, subsample=0.7, random\_state=10)

gbm2.fit(X,y)

y\_pred = gbm2.predict(X)

y\_predprob = gbm2.predict\_proba(X)[:,1]

print "Accuracy : %.4g" % metrics.accuracy\_score(y.values, y\_pred)

print "AUC Score (Train): %f" % metrics.roc\_auc\_score(y, y\_predprob)

复制代码

复制代码

　　　　输出如下：

Accuracy : 0.984

AUC Score (Train): 0.905324

　　　　可以看到AUC分数比起之前的版本稍有下降，这个原因是我们为了增加模型泛化能力，为防止过拟合而减半步长，最大迭代次数加倍，同时减小了子采样的比例，从而减少了训练集的拟合程度。

　　　　下面我们继续将步长缩小5倍，最大迭代次数增加5倍，继续拟合我们的模型：

复制代码

复制代码

gbm3 = GradientBoostingClassifier(learning\_rate=0.01, n\_estimators=600,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, max\_features=9, subsample=0.7, random\_state=10)

gbm3.fit(X,y)

y\_pred = gbm3.predict(X)

y\_predprob = gbm3.predict\_proba(X)[:,1]

print "Accuracy : %.4g" % metrics.accuracy\_score(y.values, y\_pred)

print "AUC Score (Train): %f" % metrics.roc\_auc\_score(y, y\_predprob)

复制代码

复制代码

　　　　输出如下，可见减小步长增加迭代次数可以在保证泛化能力的基础上增加一些拟合程度。

Accuracy : 0.984

AUC Score (Train): 0.908581

　　　　最后我们继续步长缩小一半，最大迭代次数增加2倍，拟合我们的模型：

复制代码

复制代码

gbm4 = GradientBoostingClassifier(learning\_rate=0.005, n\_estimators=1200,max\_depth=7, min\_samples\_leaf =60,

min\_samples\_split =1200, max\_features=9, subsample=0.7, random\_state=10)

gbm4.fit(X,y)

y\_pred = gbm4.predict(X)

y\_predprob = gbm4.predict\_proba(X)[:,1]

print "Accuracy : %.4g" % metrics.accuracy\_score(y.values, y\_pred)

print "AUC Score (Train): %f" % metrics.roc\_auc\_score(y, y\_predprob)

复制代码

复制代码

　　　　输出如下，此时由于步长实在太小，导致拟合效果反而变差，也就是说，步长不能设置的过小。

Accuracy : 0.984

AUC Score (Train): 0.908232

　　　　以上就是GBDT调参的一个总结，希望可以帮到朋友们。