集成学习 - AdaBoost（B站公开课）

```
import numpy as np
import pandas as pd
import sklearn
import matplotlib as mlp
import seaborn as sns
import re, pip, conda
```

```
for package in [sklearn,mlp,np,pd,sns,pip,conda]:
    print(re.findall("([^']*)",str(package))[2],package.__version__)
```

```
sklearn 1.0.1
matplotlib 3.4.3
numpy 1.21.4
pandas 1.3.4
seaborn 0.11.2
pip 21.3.1
conda 4.11.0
```

```
#pip install --upgrade scikit-learn
#conda update scikit-learn
```

目录

【公开课】一 Boosting方法的基本思想

【公开课】 1 Bagging pk Boosting

【公开课】 2 Boosting算法的基本元素与基本流程

【公开课】 3 sklearn中的boosting算法

【公开课】二 AdaBoost

【公开课】 1 AdaBoost的基本参数与损失函数

【公开课】 1.1 n_estimators

【公开课】 1.2 learning_rate

【完整版】 1.3 algorithm与loss

【完整版】 2 原理进阶：Adaboost回归的求解流程

（1.3节+原理进阶部分才是重头部分，占AdaBoost课时的4/5）

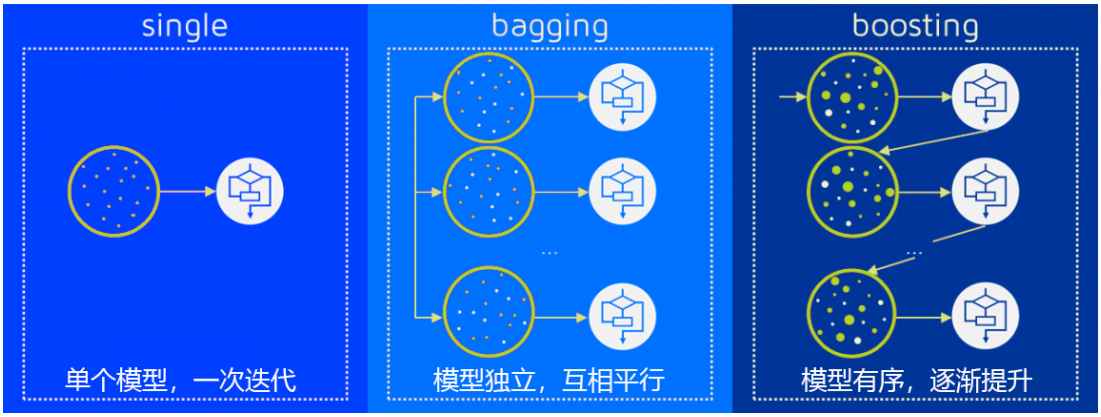
一 Boosting方法的基本思想

在集成学习的"弱分类器集成"领域，除了降低方差来降低整体泛化误差的装袋法Bagging，还有专注于**降低整体偏差**来降低泛化误差的提升法Boosting。相比起操作简单、大道至简的Bagging算法，Boosting算法在操作和原理上的难度都更大，但由于专注于偏差降低，Boosting算法们在模型效果方面的突出表现制霸整个弱分类器集成的领域。当代知名的Boosting算法当中，Xgboost，LightGBM与Catboost都是机器学习领域最强大的强学习器，Boosting毫无疑问是当代机器学习领域最具统治力的算法领域。

- **Boosting PK Bagging**

	装袋法 Bagging	**提升法 Boosting**
弱评估器	**相互独立**，并行构建	**相互关联**，按顺序依次构建 先建弱分类器的预测效果影响后续模型的建立
建树前的抽样方式	样本有放回抽样 特征无放回抽样	样本有放回抽样 特征无放回抽样 先建弱分类器的预测效果可能影响抽样细节

	装袋法 Bagging	提升法 Boosting
集成的结果	回归平均 分类众数	每个算法**具有自己独特的规则**，一般来说： (1) 表现为某种分数的加权平均 (2) 使用输出函数
目标	**降低方差** 提高模型整体的稳定性来提升泛化能力 本质是从“平均”这一数学行为中获利	**降低偏差** 提高模型整体的精确度来提升泛化能力 相信众多弱分类器叠加后可以等同于强学习器
单个评估器容易 过拟合的时候	具有一定的抗过拟合能力	具有一定的抗过拟合能力
单个评估器的效力 比较弱的时候	可能失效	大概率会提升模型表现
代表算法	随机森林	梯度提升树，Adaboost



在以随机森林为代表的Bagging算法中，我们一次性建立多个平行独立的弱评估器，并让所有评估器并行运算。在Boosting集成算法当中，我们逐一建立多个弱评估器（基本是决策树），并且下一个弱评估器的建立方式依赖于上一个弱评估器的评估结果，最终综合多个弱评估器的结果进行输出，因此Boosting算法中的弱评估器之间不仅不是相互独立的、反而是强相关的，同时Boosting算法也不依赖于弱分类器之间的独立性来提升结果，这是Boosting与Bagging的一大差别。如果说Bagging不同算法之间的核心区别在于靠以不同方式实现“独立性”（随机性），**那Boosting的不同算法之间的核心区别就在于上一个弱评估器的评估结果具体如何影响下一个弱评估器的建立过程。**

与Bagging算法中统一的回归求平均、分类少数服从多数的输出不同，Boosting算法在结果输出方面表现得十分多样。早期的Boosting算法的输出一般是最后一个弱评估器的输出，当代Boosting算法的输出都会考虑整个集成模型中全部的弱评估器。**一般来说，每个Boosting算法会其以独特的规则自定义集成输出的具体形式**，但对大部分算法而言，集成算法的输出结果往往是**关于弱评估器的某种结果的加权平均**，其中权重的求解是boosting领域中非常关键的步骤。

- **Boosting算法的基本元素与基本流程**

基于上面所明确的“降低偏差”、“逐一建树”、以及“以独特规则输出结果”的三大特色，我们可以确立任意boosting算法的三大基本元素以及boosting算法自适应建模的基本流程：

- 损失函数$L(x,y)$：用以衡量模型预测结果与真实结果的差异
- 弱评估器$f(x)$：（一般为）决策树，不同的boosting算法使用不同的建树过程
- 综合集成结果$H(x)$：即集成算法具体如何输出集成结果

这三大元素将会贯穿所有我们即将学习的boosting算法，我们会发现几乎所有boosting算法的原理都围绕这三大元素构建。在此三大要素基础上，所有boosting算法都遵循以下流程进行建模：

**

依据上一个弱评估器$f(x)_{t-1}$的结果，计算损失函数$L(x,y)$，

并使用$L(x,y)$自适应地影响下一个弱评估器$f(x)*t$的构建。*
集成模型输出的结果，受到整体所有弱评估器$f(x)*0$ ~ $f(x)_T$的影响。

**

正如之前所言，Boosting算法之间的不同之处就在于使用不同的方式来**影响**后续评估器的构建。无论boosting算法表现出复杂或简单的流程，其核心思想一定是围绕上面这个流程不变的。

- **sklearn中的boosting算法**

在sklearn当中，我们可以接触到数个Boosting集成算法，包括Boosting入门算法**AdaBoost**，性能最稳定、奠定了整个Boosting效果基础的梯度提升树**GBDT**（Gradient Boosting Decision Tree），以及近几年才逐渐被验证有效的**直方提升树**（Hist Gradient Boosting Tree）。

在过去5年之间，除了sklearn，研究者们还创造了大量基于GBDT进行改造的提升类算法，这些算法大多需要从第三方库进行调用，例如极限提升树**XGBoost**（Extreme Gradient Boosting Tree），轻量梯度提升树**LightGBM**（Light Gradiant Boosting Machine），以及离散提升树**CatBoost**（Categorial Boosting Tree）。

Boosting算法	库	集成类
ADB分类	sklearn	AdaBoostClassifer
ADB回归	sklearn	AdaBoostRegressor
梯度提升树分类	sklearn	GradientBoostingClassifier
梯度提升树回归	sklearn	GradientBoostingRegressor
直方提升树分类	sklearn	HistGraidientBoostingClassifier
直方提升树回归	sklearn	HistGraidientBoostingRegressor
极限提升树	第三方库xgboost	xgboost.train()
轻量梯度提升树	第三方库lightgbm	lightgbm.train()
离散提升树	第三方库catboost	catboost.train()

在课程当中，我们会一一介绍以上所有算法的原理与用法。另外需要注意的是，周志华老师于2017年提出的深度森林算法既不是boosting也不是bagging，而是以深度学习的思路重新集成决策树之后得到的独特算法，可以算是模型融合的一部分。

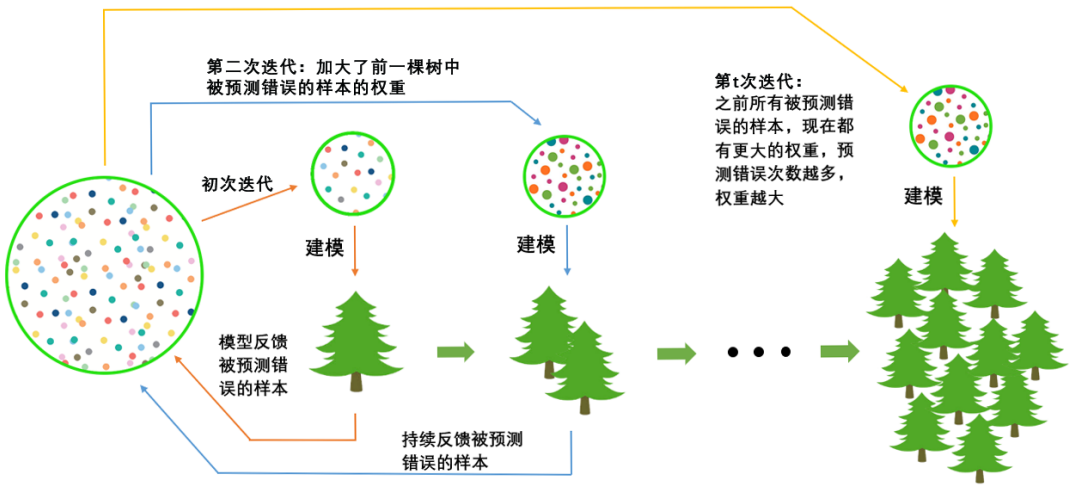
二 AdaBoost

AdaBoost（Adaptive Boosting，自适应提升法）是当代boosting领域的开山鼻祖，它虽然不是首个实践boosting思想算法，却是首个成功将boosting思想发扬光大的算法。它的主要贡献在于实现了两个变化：

1、首次实现根据之前弱评估器的结果**自适应地**影响后续建模过程

2、在Boosting算法中，首次实现考虑全部弱评估器结果的输出方式

作为开山算法，AdaBoost的构筑过程非常简单：**首先，在全样本上建立一棵决策树，根据该决策树预测的结果和损失函数值，增加被预测错误的样本在数据集中的样本权重，并让加权后的数据集被用于训练下一棵决策树。**这个过程相当于有意地加重“难以被分类正确的样本”的权重，同时降低“容易被分类正确的样本”的权重，而将后续要建立的弱评估器的注意力引导到难以被分类正确的样本上。



在该过程中，上一棵决策树的的结果通过影响样本权重、即影响数据分布来影响下一棵决策树的建立，整个过程是自适应的。当全部弱评估器都被建立后，集成算法的输出$H(x)$等于所有弱评估器输出值的加权平均，加权所用的权重也是在建树过程中被自适应地计算出来的。

需要注意的是，虽然最初的原理较为简单，但近年来AdaBoost在已经发展出多个升级的版本（比如，**在建立每棵树之前，允许随机抽样特征，这使得Boosting中的决策树行为更加接近Bagging中的决策树**），而sklearn中使用了这些升级后的版本进行实现。幸运的是，这些实现并不影响我们对sklearn中的类的使用，对这些实现的具体过程感兴趣的小伙伴，可以在章节《2 原理进阶：AdaBoost的求解流程》中查看具体原理。

在sklearn中，AdaBoost既可以实现分类也可以实现回归，我们使用如下两个类来调用它们：

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

class `sklearn.ensemble.AdaBoostRegressor`(base_estimator=None, *, n_estimators=50, learning_rate=1.0, loss='linear', random_state=None)

不难发现，AdaBoost的参数非常非常少，在调用AdaBoost时我们甚至无需理解AdaBoost的具体求解过程。同时，ADB分类器与ADB回归器的参数也高度一致。在课程当中，我们将重点Boosting算法独有的参数，以及ADB分类与ADB回归中表现不一致的参数。

参数	参数含义
base_estimator	弱评估器
n_estimators	集成算法中弱评估器的数量
learning_rate	迭代中所使用的学习率
algorithm（分类器专属）	用于指定分类ADB中使用的具体实现方法
loss（回归器专属）	用于指定回归ADB中使用的损失函数
random_state	用于控制每次建树之前随机抽样过程的随机数种子

1 AdaBoost的基本参数与损失函数

```
from sklearn.ensemble import AdaBoostClassifier as ABC
from sklearn.ensemble import AdaBoostRegressor as ABR
from sklearn.tree import DecisionTreeClassifier as DTC
from sklearn.tree import DecisionTreeRegressor as DTR
from sklearn.datasets import load_digits
```

```
#用于分类的数据
data_c = load_digits()
X_c = data_c.data
y_c = data_c.target
```

```
X_c.shape
```

```
(1797, 64)
```

```
X_c
```

```
array([[ 0.,  0.,  5., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ..., 10.,  0.,  0.],
       [ 0.,  0.,  0., ..., 16.,  9.,  0.],
       ...,
       [ 0.,  0.,  1., ...,  6.,  0.,  0.],
       [ 0.,  0.,  2., ..., 12.,  0.,  0.],
       [ 0.,  0., 10., ..., 12.,  1.,  0.]])
```

```
np.unique(y_c) #手写数字数据集，10分类
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
#用于回归的数据
data_r = pd.read_csv(r"D:\Pythonwork\2021ML\PART 2 Ensembles\datasets\House
Price\train_encode.csv",index_col=0)
X_g = data_r.iloc[:,:-1]
y_g = data_r.iloc[:,-1]
```

```
X_g.shape
```

```
(1460, 80)
```

```
X_g.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Id	住宅类型	住宅区域	街道接触面积(英尺)	住宅面积	街道路面状况	巷子路面状况	住宅形状(大概)	住宅现状	水电气	...	半开放式门廊面积	泳池面积	泳池质量	篱笆质量	其他配置	其他配置的价值	销售月份	销售年份
0	0.0	5.0	3.0	36.0	327.0	1.0	0.0	3.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	2.0
1	1.0	0.0	3.0	51.0	498.0	1.0	0.0	3.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	4.0	1.0
2	2.0	5.0	3.0	39.0	702.0	1.0	0.0	0.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	8.0	2.0
3	3.0	6.0	3.0	31.0	489.0	1.0	0.0	0.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
4	4.0	5.0	3.0	55.0	925.0	1.0	0.0	0.0	3.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	11.0	2.0

5 rows × 80 columns

- 参数`base_estimator`，属性`base_estimator_`与`estimators_`

`base_estimator`是规定AdaBoost中使用弱评估器的参数。与对弱评估器有严格要求的Bagging算法不同，boosting算法通过降低偏差来降低整体泛化误差，因此可以使用任意弱评估器，且这些弱评估器往往被假设成非常弱小的评估器。当然了，默认的弱评估器还是决策树。在sklearn中，**ADB分类器的默认弱评估器是最大深度为1的"树桩"，ADB回归器的默认评估器是最大深度为3的"树苗"**，弱评估器本身基本不具备判断能力。而回归器中树深更深是因为boosting算法中回归任务往往更加复杂。在传统ADB理论当中，一般认为AdaBoost中的弱分类器为最大深度为1的树桩，但现在我们也可以自定义某种弱评估器来进行输入。

当模型建好之后，我们可以使用属性`base_estimator_`来查看当前弱评估器，同时也可以使用`estimators_`来查看当前集成模型中所有弱评估器的情况：

- 建立集成算法，调用其中的弱评估器

```
#建立ADB回归器和分类器
clf = ABC(n_estimators=3).fit(X_c,y_c)
reg = ABR(n_estimators=3).fit(X_g,y_g)
```

```
clf.base_estimator_
```

```
DecisionTreeClassifier(max_depth=1)
```

```
reg.base_estimator_
```

```
DecisionTreeRegressor(max_depth=3)
```

```
reg.estimators_
```

```
[DecisionTreeRegressor(max_depth=3, random_state=765348147),
 DecisionTreeRegressor(max_depth=3, random_state=850911835),
 DecisionTreeRegressor(max_depth=3, random_state=1434155639)]
```

当AdaBoost完成分类任务时，弱评估器是分类树，当AdaBoost完成回归任务时，弱评估器是回归树，这一点与之后的Boosting算法们有较大的区别。

- 自建弱评估器

```
base_estimator = DTC(max_depth=10,max_features=30)
```

```
clf = ABC(base_estimator = base_estimator, n_estimators=3).fit(X_c,y_c)
```

```
clf.base_estimator_
```

```
DecisionTreeClassifier(max_depth=10, max_features=30)
```

```
clf.estimators_
```

```
[DecisionTreeClassifier(max_depth=10, max_features=30, random_state=814836020),
 DecisionTreeClassifier(max_depth=10, max_features=30, random_state=880262373),
 DecisionTreeClassifier(max_depth=10, max_features=30, random_state=925249775)]
```

注意，为了保证集成算法中的树不一致，AdaBoost会默认消除我们填写在弱评估器中的random_state：

```
base_estimator = DTC(max_depth=10,max_features=30,random_state=1412)
```

```
clf = ABC(base_estimator = base_estimator, n_estimators=3).fit(X_c,y_c)
```

```
clf.estimators_
```

```
[DecisionTreeClassifier(max_depth=10, max_features=30, random_state=677195652),
 DecisionTreeClassifier(max_depth=10, max_features=30, random_state=1650391099),
 DecisionTreeClassifier(max_depth=10, max_features=30, random_state=672741048)]
```

- 参数`learning_rate`

在Boosting集成方法中，集成算法的输出$H(x)$往往都是多个弱评估器的输出结果的加权平均结果。但$H(x)$并不是在所有树建好之后才统一加权求解的，而是在算法逐渐建树的过程当中就随着迭代不断计算出来的。例如，对于样本x_i，集成算法当中一共有T棵树（也就是参数`n_estimators`的取值），现在正在建立第t个弱评估器，则第t个弱评估器上x_i的结果可以表示为$f_t(x_i)$。假设整个Boosting算法对样本x_i输出的结果为$H(x_i)$，则该结果一般可以被表示为t=1~t=T过程当中，所有弱评估器结果的加权求和：

$$H(x_i) = \sum_{t=1}^T\phi_tf_t(x_i)$$

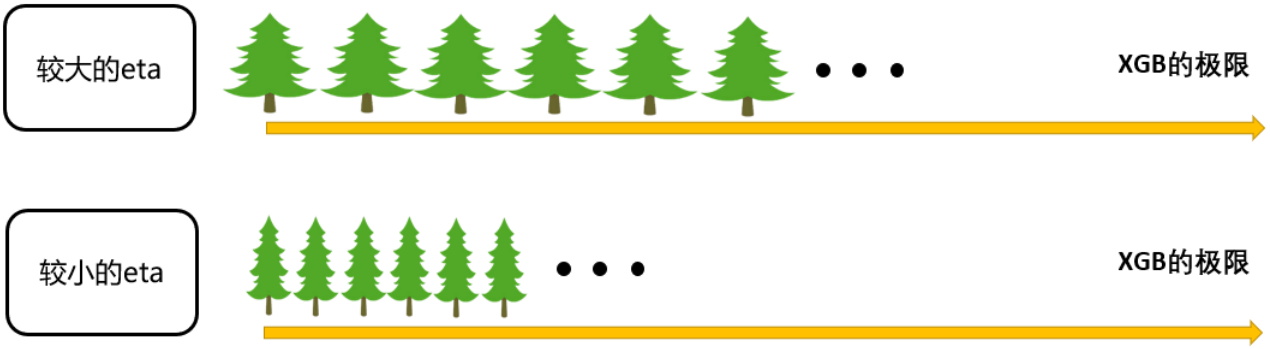
其中，ϕ_t为第t棵树的权重。对于第t次迭代来说，则有：

$$H_t(x_i) = H_{t-1}(x_i) + \phi_tf_t(x_i)$$

在这个一般过程中，每次将本轮建好的决策树加入之前的建树结果时，可以在权重ϕ前面增加参数$\color{red}\eta$，表示为第t棵树加入整体集成算法时的学习率，对标参数`learning_rate`。

$$H_t(x_i) = H_{t-1}(x_i) + \boldsymbol{\color{red}\eta} \phi_tf_t(x_i)$$

该学习率参数控制Boosting集成过程中$H(x_i)$的增长速度，是相当关键的参数。当学习率很大时，$H(x_i)$增长得更快，我们所需的n_estimators更少，当学习率较小时，$H(x_i)$增长较慢，我们所需的n_estimators就更多，因此boosting算法往往会需要在n_estimators与learning_rate当中做出权衡（以XGBoost算法为例）。



需要注意的是，**以上式子为boosting算法中计算方式的一般规则，并不是具体到AdaBoost或任意Boosting集成算法的具体公式。**