



分类与预测

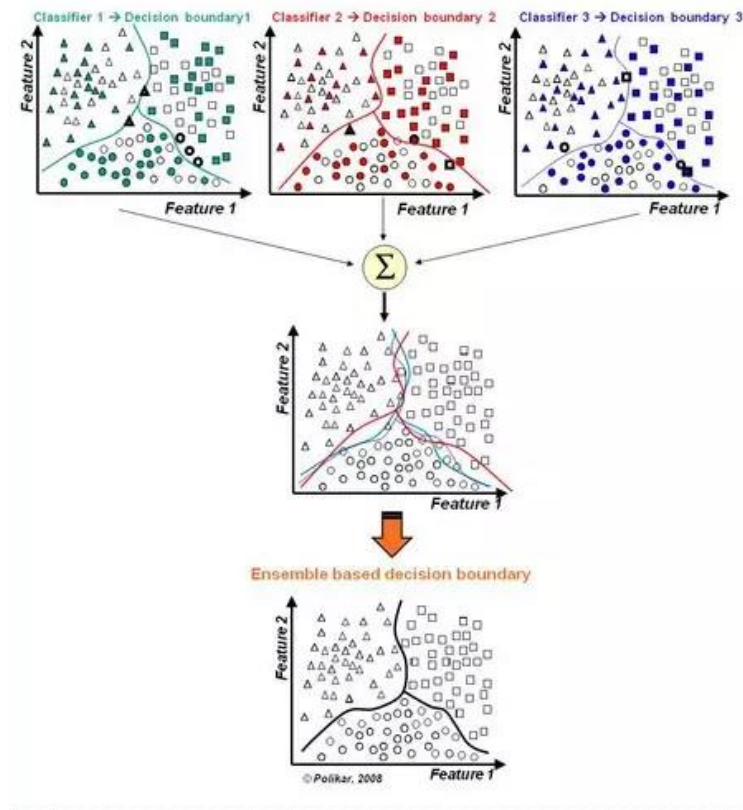
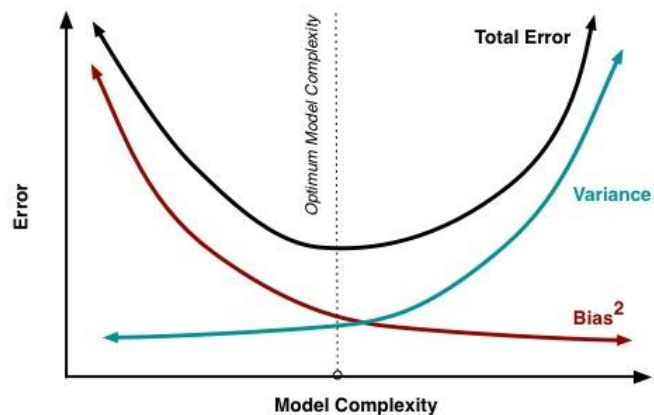
2021/11/8

目录

1	背景
2	决策树
3	回归
4	集成学习
5	神经网络
6	深度学习

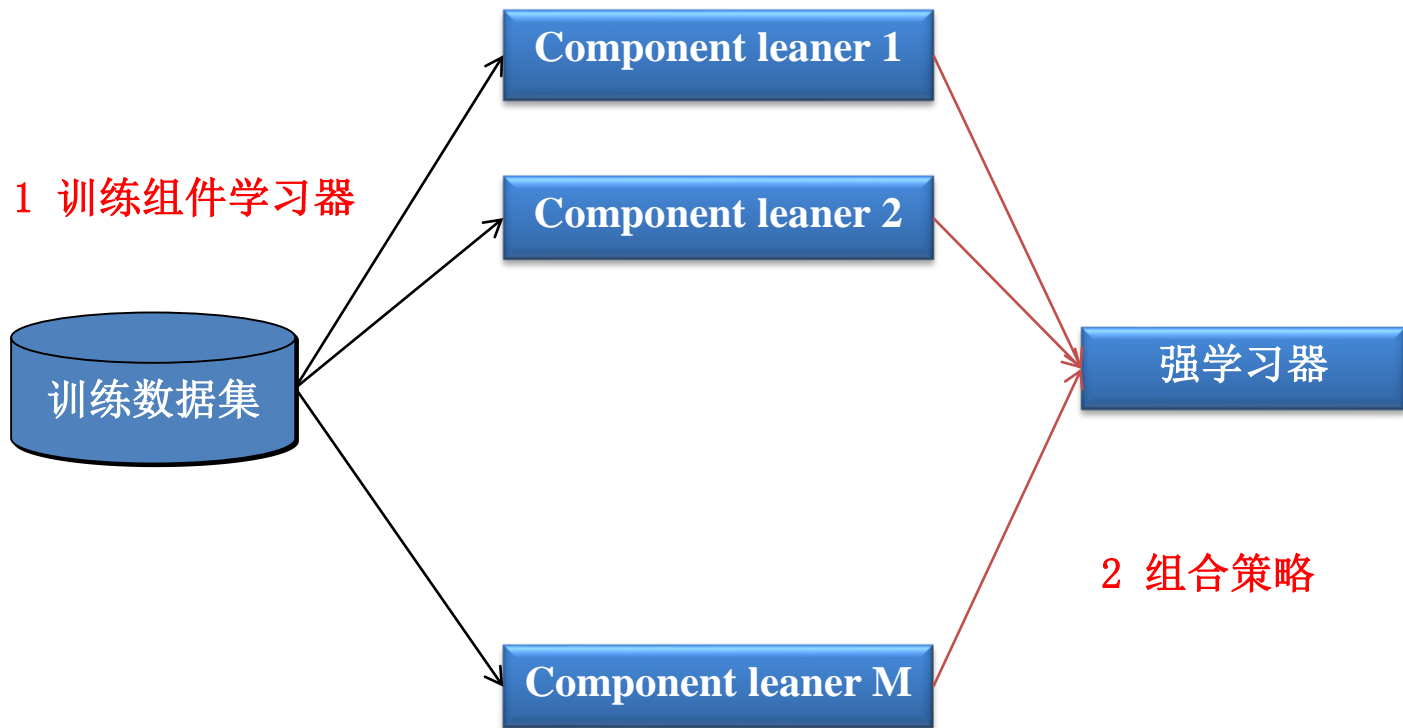
集成学习的动机

误差的期望值 = 噪音的方差 + 模型预测值的方差 + 预测值相对真实值的偏差的平方



单模型的局限性: **High bias or high variance**

集成学习



集成学习

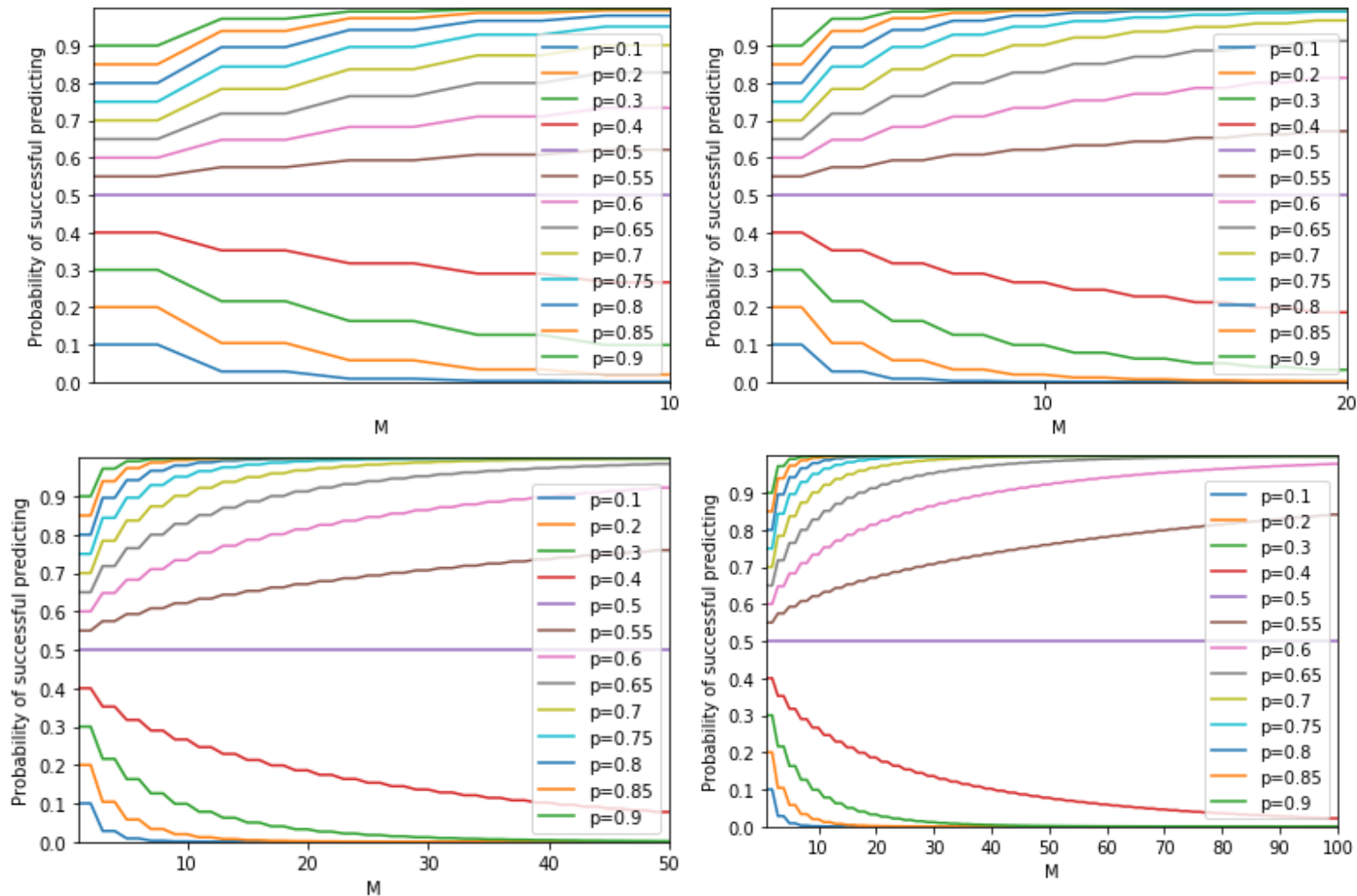
集成学习：基于训练集数据训练若干个不同的组件学习器，通过一定的集成策略形成一个强学习器。

两个问题

- ◆ 第一是如何得到若干个不同的组件学习器？
- ◆ 第二是如何选择一种组合策略，将这些组件学习器集成为一个强学习器？

集成学习的意义

假设有 M 个正确率为 p 的二分类器，可以将这些分类器视为伪随机数产生器，以 p 的概率产生“1”， $(1-p)$ 的概率产生“0”。将 M 个结果用投票法组合得到最后的结果，正确率如下。



$p > 0.5$ ，随着 M 的增加，投票后的正确概率迅速增大。

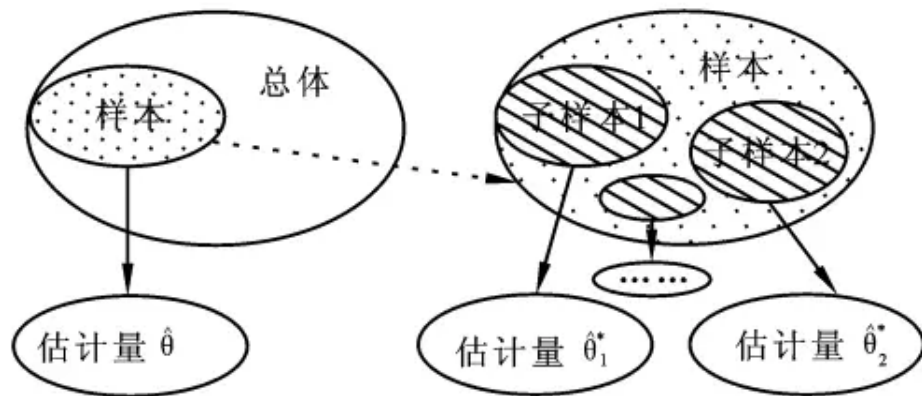
Bootstrap抽样方法

◆ 栗子:

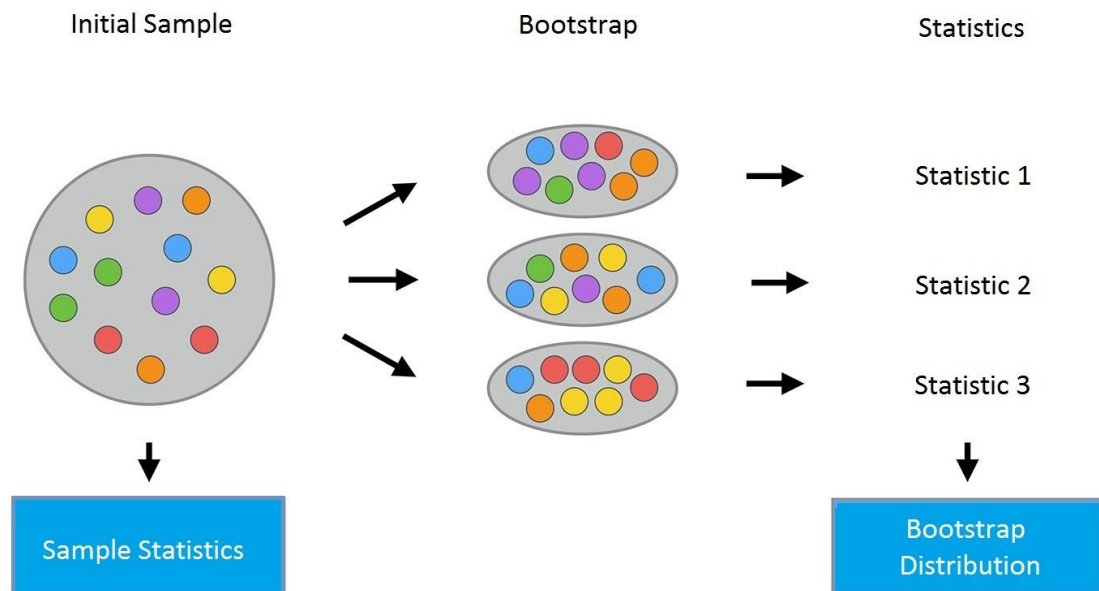
鱼塘里有许多鱼，不知道某类鱼有多少条，也不知道某类鱼占鱼的总数量占比是多少，怎么办？

◆ Bootstrap(自助法)

1. 承包鱼塘，不让别人捞鱼(规定总体分布不变)。
2. 自己捞鱼，捞某类鱼**100**条，都打上标签(构造样本)
3. 把鱼放回鱼塘，休息一晚(使之混入整个鱼群，确保之后 抽样随机)
4. 开始捞鱼，每次捞**1000**条，数一下某类鱼有多少条，自己昨天标记的某类鱼有多少条，占比多少(一次重采样取分布)。
5. 重复3，4步骤**n**次。建立分布。



Bootstrap抽样方法



对于包含 n 个样本的样本集，其总体的概率分布中的参数 θ 是未知的，现在想要利用这些有限的样本得到一个估测值 $\hat{\theta}$ 。

Step1: 根据自己的需要选择确定重采样的次数 i ，1000或者2000等；

Step2: 从 n 个样本中有放回的抽取 n 次；

Step3: 重复Step2，一共 i 次（Step1中的 i ），对每次抽取后的结果都计算出一个 $\hat{\theta}$ 值；

Step4: 利用得到的 $\hat{\theta}_1, \hat{\theta}_2, \dots, \hat{\theta}_i$ 计算估计量的均值和方差。

Bootstrap特点

- 在小样本时效果很好。但对超大规模样本存在计算量大的问题（要求抽样的样本数不能过小）。
- 就是一个在自身样本重采样的方法来估计真实分布的问题
- 当我们不知道样本分布的时候，bootstrap方法最有用。
- 可用于整合多个弱分类器，成为一个强大的分类器。

集成方法-组件学习器

1 组件学习器类型？

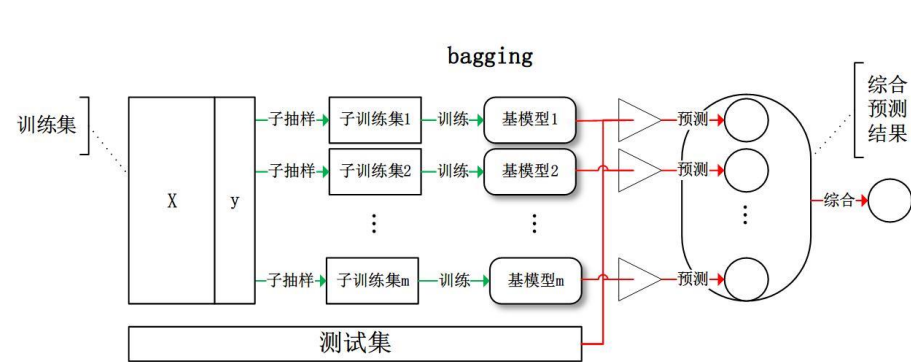


同质，如bagging, boosting

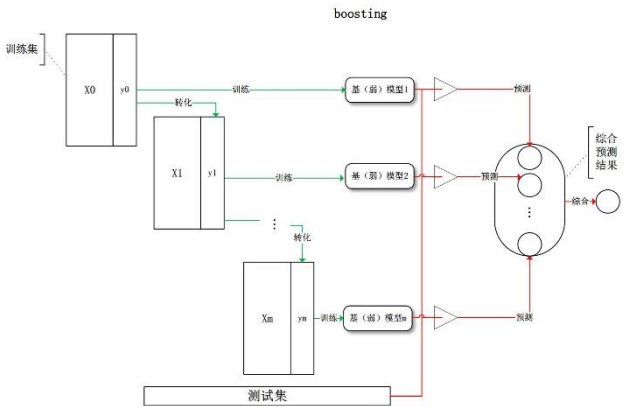


异质，如stacking

2 组件学习器生成？

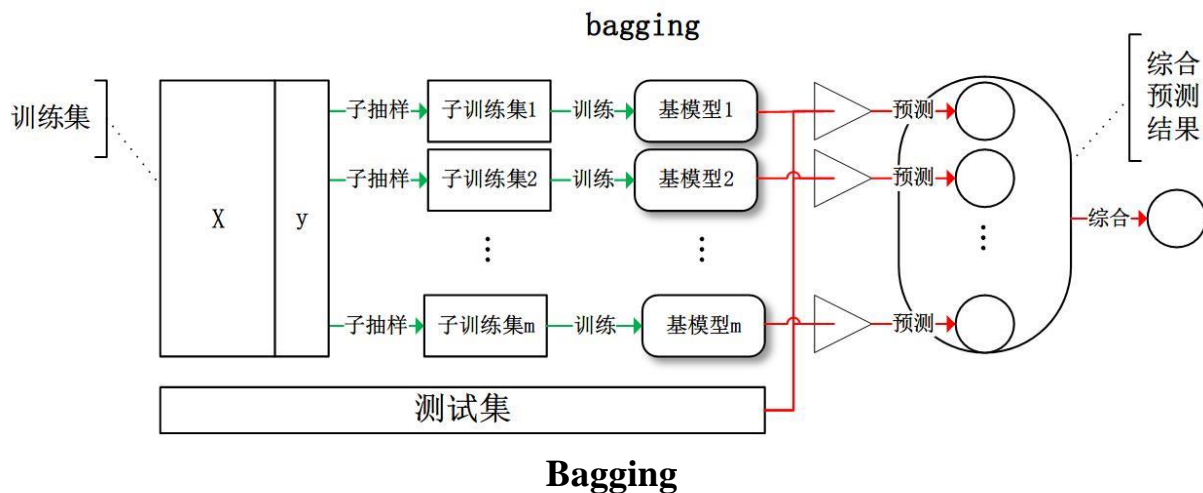


并行生成，如bagging



串行生成，如boosting

Bagging



步骤

1. 从原始样本集中抽取训练集。每轮从原始样本集中使用Bootstrap（有放回）方法抽取 n 个训练样本（在训练集中，有些样本可能被多次抽取到，而有些样本可能一次都没有被抽中）。共进行 m 轮抽取，得到 m 个独立的训练集。
2. 每次使用一个训练集得到一个模型， m 个训练集共得到 m 个模型。
3. 对分类问题，将上步得到的 m 个模型采用投票的方式得到分类结果；对回归问题，计算上述模型的均值作为最后的结果。

Bagging误差分析

让我们考虑一个回归问题，并且设计每个基算法为 $h_1(x), h_2(x), \dots, h_m(x)$ 。假设每个输入对应的输出为 y ，并且数据的分布为 $p(x)$ 。那么我们可以表达每个回归函数的误差、均方误差期望及平均误差如下所示：

$$\varepsilon_i(x) = h_i(x) - y \quad E_p[(h_i(x) - y)^2] = E_p[\varepsilon_i^2(x)]$$

现在构造一个新的回归函数

$$a(x) = \frac{1}{m} \sum_{i=1}^m h_i(x)$$

则有

$$\begin{aligned} \text{Bias}[\hat{f}(x)] &= f(x) - a(x) \\ &= f(x) - \bar{h}(x) \end{aligned}$$

$$\begin{aligned} \text{Var} \left(\frac{1}{m} \sum_{i=1}^m h_i(x) \right) &= E_p \left[\frac{1}{m} \sum_{i=1}^m h_i(x) \right]^2 \\ &= \frac{1}{m^2} E_p \left[\sum_{i=1}^m h_i(x) \right]^2 \\ &= \frac{1}{m^2} \left(\sum_{i=1}^m E_p[h_i^2(x)] + \sum_{i,j=1, i \neq j}^m E_p[h_i(x)h_j(x)] \right) \end{aligned}$$

假设模型是不相关的，即：

$$E_p[h_i(x)h_j(x)] = 0, i \neq j$$

$$\begin{aligned} \text{Var} \left(\frac{1}{m} \sum_{i=1}^m h_i(x) \right) &= \frac{1}{m} \left(\frac{1}{m} \sum_{i=1}^m E_p[h_i^2(x)] \right) \end{aligned}$$

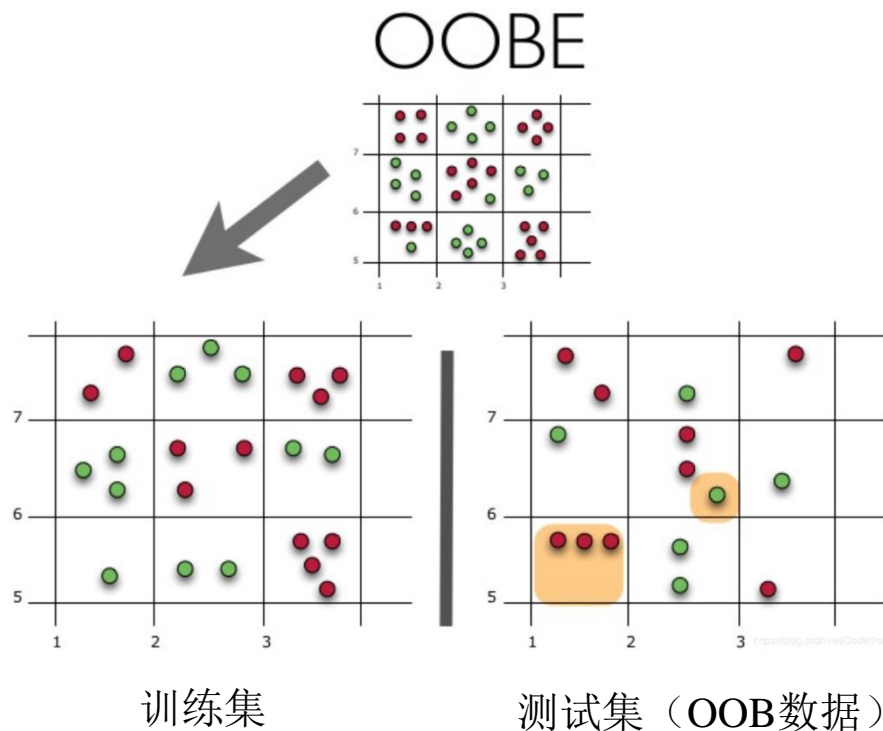
当我们在不同数据集上面进行训练模型时，Bagging 通过减小误差的方差来降低误差。换句话说，bagging 模型能降低过拟合。Bagging不能减小偏差。

OOB 错误(out of bag error)

假设我们的数据集中有 n 个数据，在每个步骤中，每个数据点都有相同的概率被取到，概率为 $\frac{1}{n}$ 。
没有包含这种元素的概率为

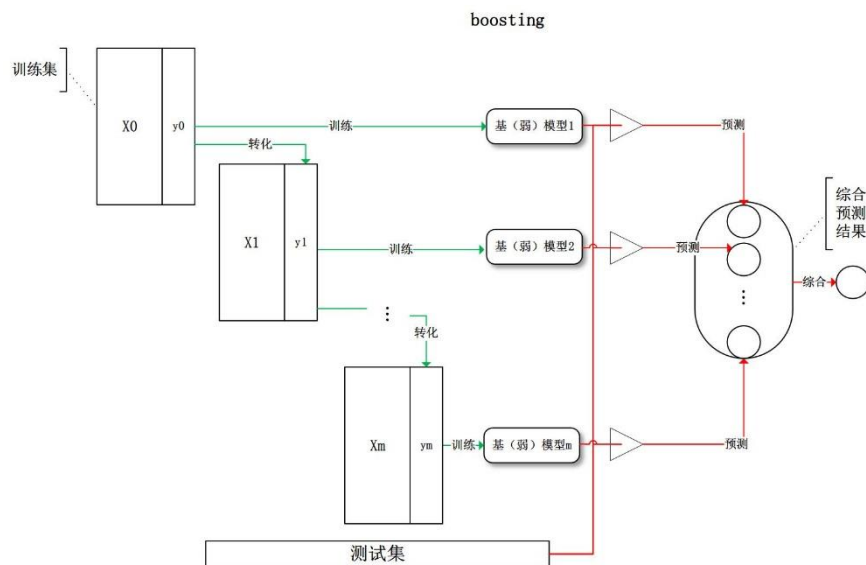
$$\lim_{n \rightarrow +\infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}$$

选择特定数据的概率为 $1 - 1/e \approx 0.63$ ，有37%的输入数据不会被选到，称为OOB（out of bag）数据，可以用来测试模型。



袋外误差（OOBE）为 $4/15 = 26.67\%$

Boosting



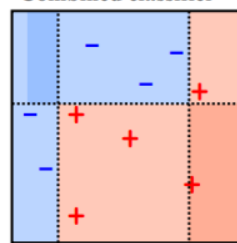
Boosting

Weight each classifier and combine them:

$$.33 * \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} + .57 * \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} + .42 * \begin{array}{|c|} \hline \text{blue} \\ \hline \text{red} \\ \hline \end{array} \geq 0$$



Combined classifier



1-node decision trees
"decision stumps"
very simple classifiers

Boosting主要思想是将弱分类器组装成一个强分类器。在PAC（probably approximately correct，概率近似正确）学习框架下，则一定可以将弱分类器组装成一个强分类器。

Boosting的两个核心问题：

1. 在每一轮如何改变训练数据的权值或概率分布？

通过提高那些在前一轮被弱分类器分错样例的权值，减小前一轮分对样例的权值，来使得分类器对误分的数据有较好的效果。

➤ 通过什么方式来组合弱分类器？

通过加法模型将弱分类器进行线性组合

Bagging与Boosting对比

	Bagging	Boosting
抽样	均匀取样	概率抽样，概率大小与误差成反比
放回情况	有放回	每一轮抽样是不放回抽样，不同轮之间抽样没关系
训练集	独立	依赖
结构	并行	串行
测试	可并行	串行
作用	减小方差	减小偏差

组合策略

1 平均法

$$H(\mathbf{x}) = \frac{1}{T} \sum_{i=1}^T h_i(\mathbf{x})$$

简单平均法

$$H(\mathbf{x}) = \sum_{i=1}^T w_i h_i(\mathbf{x})$$

加权平均法，w为权重

2 投票法

$$H(\mathbf{x}) = \begin{cases} c_j, & \text{if } \sum_{i=1}^T h_i^j(\mathbf{x}) > 0.5 \sum_{k=1}^N \sum_{i=1}^T h_i^k(\mathbf{x}) \\ \text{reject}, & \text{otherwise.} \end{cases}$$

绝对多数投票法：得票最多的类别的票数要超过50%，否则拒绝预测

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T h_i^j(\mathbf{x})}$$

相对多数投票法，无需超过50%

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T w_i h_i^j(\mathbf{x})}$$

加权投票法，w为权重

$$H(\mathbf{x}) = c_{\arg \max_j \sum_{i=1}^T w_i h_i^j(\mathbf{x})}$$

集成学习的错误率

考虑二分类问题 $y \in \{-1, 1\}$ 和真实函数 f ，每个基分类器 $f_l(x)$ 的错误率为 ϵ ，即

$$P(f_l(x) \neq f(x)) = \epsilon$$

假设通过投票的方法将 M 个分类器集成在一起，若超过一半的基分类器结果正确的话，那么集成分类器就正确

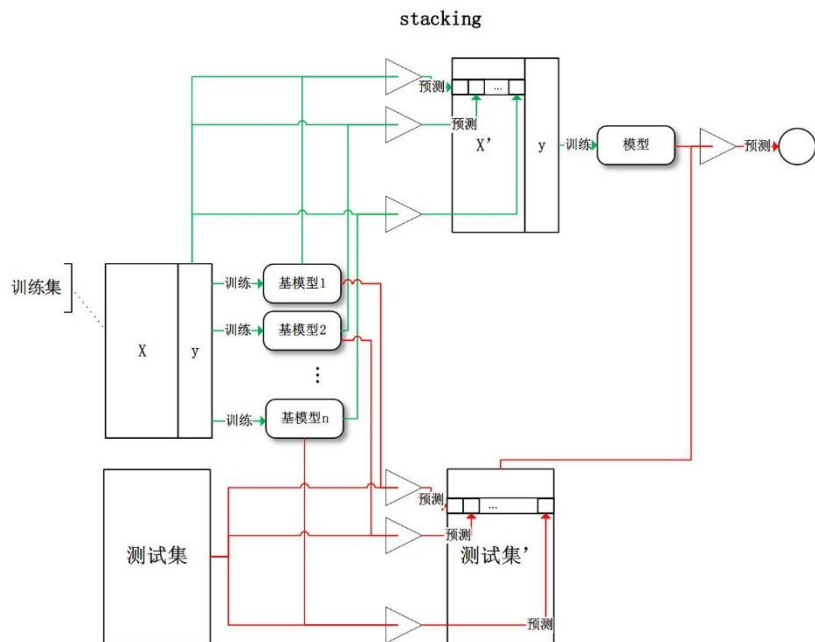
$$F(x) = \text{sgn}\left(\sum_{l=1}^m f_l(x)\right)$$

假设基分类器的错误率相互独立，则由Hoeffding不等式可知，集成分类器的错误率为

$$P(F(x) \neq f(x)) = \sum_{k=0}^{\lfloor M/2 \rfloor} \binom{m}{k} (1-\epsilon)^k \epsilon^{m-k} \leq \exp\left(-\frac{1}{2}m(1-2\epsilon)^2\right)$$

由上式可知，随着基分类器数目 M 的增加，集成分类器的错误率将指数级的下降。

组合策略-Stacking



输入: 训练集 $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$;
初级学习算法 $\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_T$;
次级学习算法 \mathcal{L} .

过程:

```
1: for  $t = 1, 2, \dots, T$  do
2:    $h_t = \mathcal{L}_t(D)$ ;
3: end for
4:  $D' = \emptyset$ ;
5: for  $i = 1, 2, \dots, m$  do
6:   for  $t = 1, 2, \dots, T$  do
7:      $z_{it} = h_t(x_i)$ ;
8:   end for
9:    $D' = D' \cup ((z_{i1}, z_{i2}, \dots, z_{iT}), y_i)$ ;
10: end for
11:  $h' = \mathcal{L}(D')$ ;
```

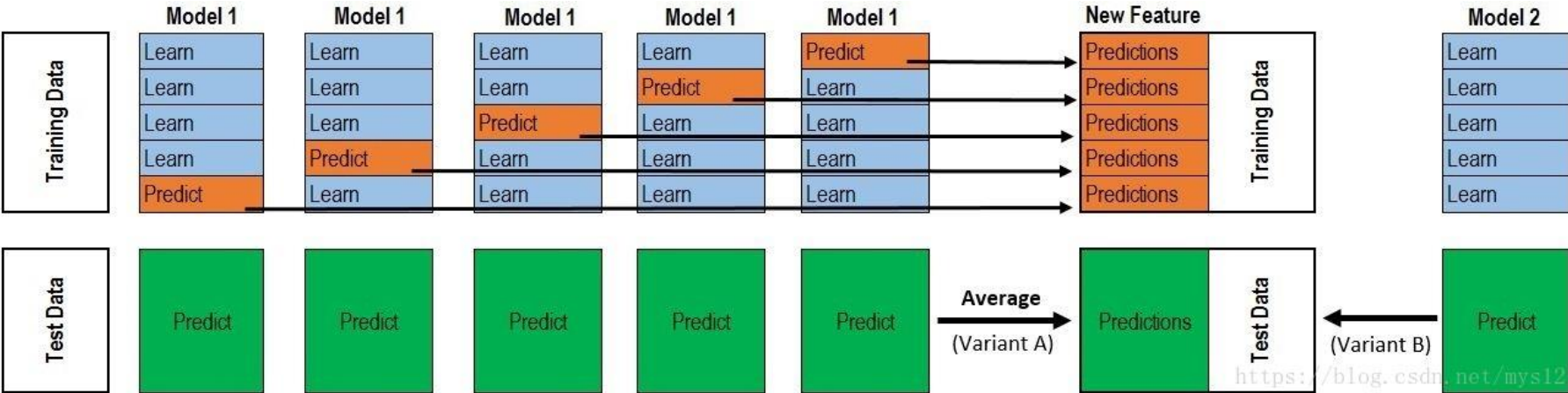
输出: $H(x) = h'(h_1(x), h_2(x), \dots, h_T(x))$

Stacking

步骤

1. 首先用不同的算法形成 T 个弱分类器
2. 将训练好的所有基模型对训练集进行预测, 第 j 个基模型对第 i 个训练样本的预测值将作为新的训练集中第 i 个样本的第 j 个特征值, 最后基于新的训练集训练新模型。同理, 预测的过程也要先经过所有基模型的预测形成新的测试集, 最后再对测试集进行预测。

Stacking



Stacking

```
import csv
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cross_validation import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import LabelEncoder
from sklearn import metrics
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier, GradientBoostingClassifier
```

```
def run(data):
    X = np.array([ i[:-1] for i in data ], dtype=float)
    Y = np.array([ i[-1] for i in data ])

    # We need to transform the string output to numeric
    label_encoder = LabelEncoder()
    label_encoder.fit(Y)
    Y = label_encoder.transform(Y)

    # The DEV SET will be used for all training and validation purposes
    # The TEST SET will never be used for training, it is the unseen set.
    dev_cutoff = len(Y) * 4/5
    X_dev = X[:dev_cutoff]
    Y_dev = Y[:dev_cutoff]
    X_test = X[dev_cutoff:]
    Y_test = Y[dev_cutoff:]

    n_trees = 10
    n_folds = 5

    # Our level 0 classifiers
    clfs = [
        RandomForestClassifier(n_estimators = n_trees, criterion = 'gini'),
        ExtraTreesClassifier(n_estimators = n_trees * 2, criterion = 'gini'),
        GradientBoostingClassifier(n_estimators = n_trees),
    ]

    # Ready for cross validation
    skf = list(StratifiedKFold(Y_dev, n_folds))

    # Pre-allocate the data
    blend_train = np.zeros((X_dev.shape[0], len(clfs))) # Number of training data x Number of classifiers
    blend_test = np.zeros((X_test.shape[0], len(clfs))) # Number of testing data x Number of classifiers

    print 'X_test.shape = %s' % (str(X_test.shape))
    print 'blend_train.shape = %s' % (str(blend_train.shape))
    print 'blend_test.shape = %s' % (str(blend_test.shape))
```

```
# For each classifier, we train the number of fold times (=len(skf))
for j, clf in enumerate(clfs):
    print 'Training classifier [%s]' % (j)
    blend_test_j = np.zeros((X_test.shape[0], len(skf))) # Number of testing data x Number of folds
    for i, (train_index, cv_index) in enumerate(skf):
        print 'Fold [%s]' % (i)

        # This is the training and validation set
        X_train = X_dev[train_index]
        Y_train = Y_dev[train_index]
        X_cv = X_dev[cv_index]
        Y_cv = Y_dev[cv_index]

        clf.fit(X_train, Y_train)

        # This output will be the basis for our blended classifier to train against,
        # which is also the output of our classifiers
        blend_train[cv_index, j] = clf.predict(X_cv)
        blend_test_j[:, i] = clf.predict(X_test)
        # Take the mean of the predictions of the cross validation set
        blend_test[:, j] = blend_test_j.mean(1)

    print 'Y_dev.shape = %s' % (Y_dev.shape)

# Start blending!
bcf = LogisticRegression()
bcf.fit(blend_train, Y_dev)

# Predict now
Y_test_predict = bcf.predict(blend_test)
score = metrics.accuracy_score(Y_test, Y_test_predict)
print 'Accuracy = %s' % (score)

return score

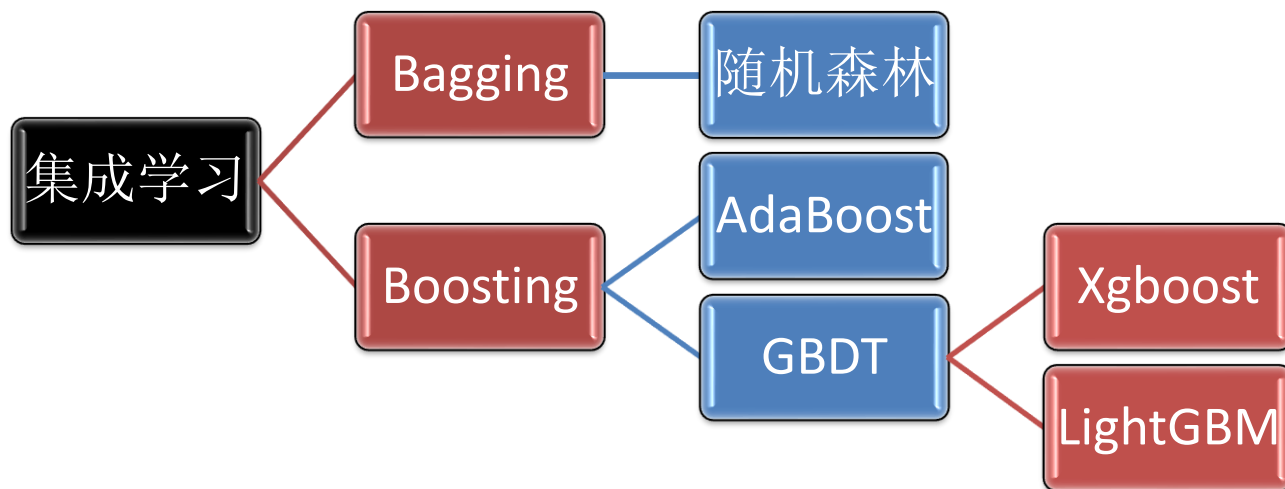
if __name__ == '__main__':
    train_file = 'data/column_3C.dat'

    data = [ i for i in csv.reader(file(train_file, 'rb'), delimiter=',') ]
    data = data[1:] # remove header

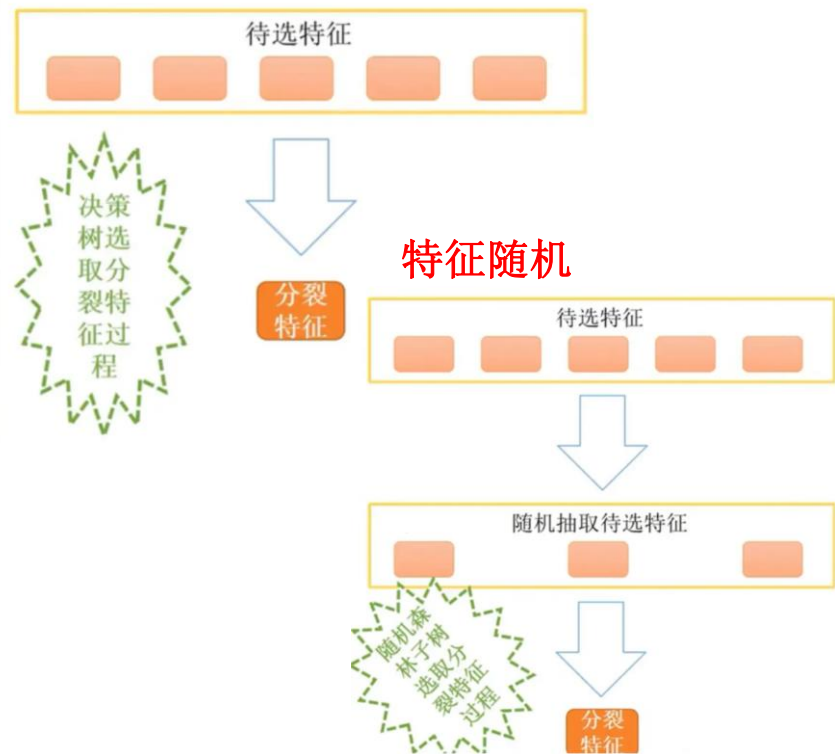
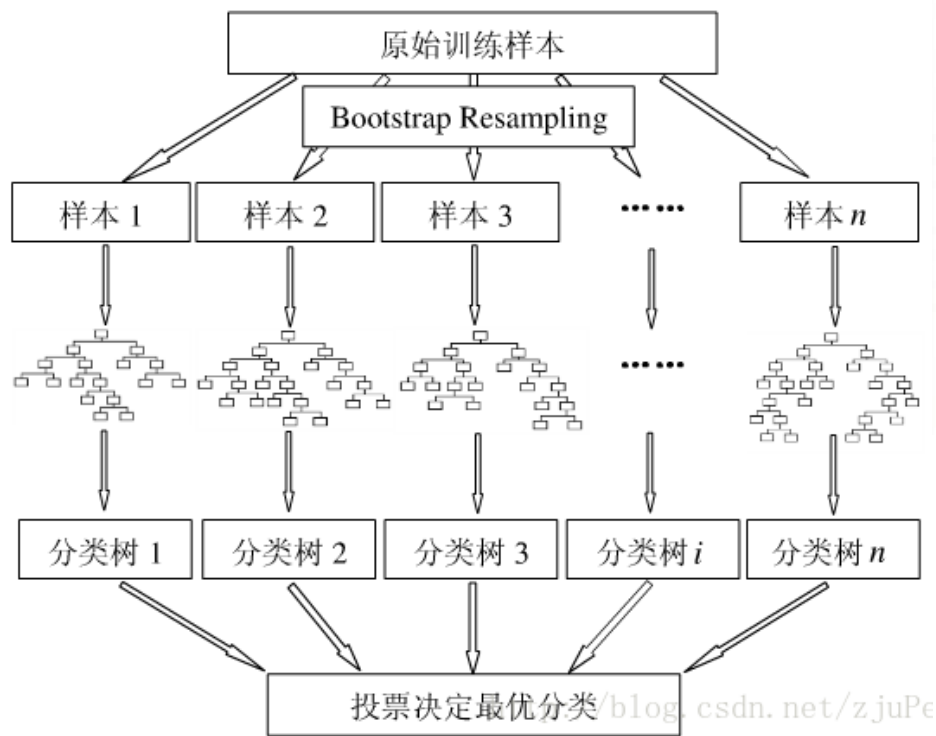
    best_score = 0.0

    # run many times to get a better result, it's not quite stable.
    for i in xrange(1):
        print 'Iteration [%s]' % (i)
        random.shuffle(data)
        score = run(data)
        best_score = max(best_score, score)
        print

    print 'Best score = %s' % (best_score)
```



随机森林



1. 用有抽样放回的方法 (bootstrap) 从样本集中选取 n 个样本作为一个训练集 **样本随机**
2. 用抽样得到的样本集生成一棵决策树。在生成的每一个结点：随机不重复地选择 d 个特征（子空间），利用这 d 个特征分别对样本集进行划分，找到最佳的划分特征（可用基尼系数、增益率或者信息增益判别） **特征随机**
3. 重复步骤1到步骤2共 T 次， T 即为随机森林中决策树的个数。
4. 用训练得到的随机森林对测试样本进行预测，并用投票法决定预测结果。

随机森林泛化误差

随机森林泛化误差界为

$$PE^* \leq \frac{\bar{\rho}(1 - s^2)}{s^2}$$

s 为单个决策树的分类强度

$\bar{\rho}$ 为决策树之间的相关性

单个决策树强度越大，相关性越小，则随机森林分类泛化误差越小

随机森林代码

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.datasets import load_iris
RF = RandomForestClassifier(n_estimators=100,n_jobs=4,oob_score=True)
#ET = ExtraTreesClassifier(n_estimators=100,n_jobs=4,oob_score=True,bootstrap=True)
iris = load_iris()
x = iris.data[:, :2]
y = iris.target
RF.fit(x, y)
#ET.fit(x, y)

h = .02 # step size in the mesh

# Create color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])

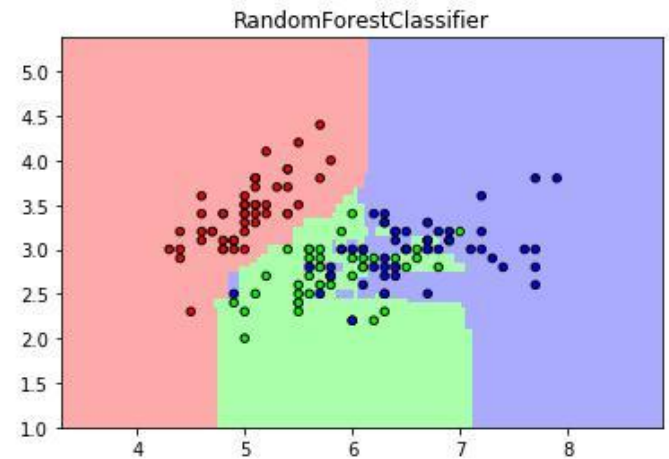
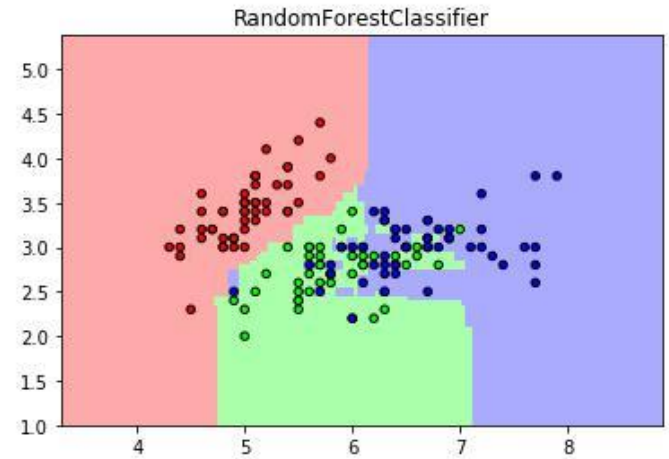
for weights in ['uniform', 'distance']:
    # we create an instance of Neighbours Classifier and fit the data.
    #clf = neighbors.KNeighborsClassifier(n_neighbors, weights=weights)
    #clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = x[:, 0].min() - 1, x[:, 0].max() + 1
    y_min, y_max = x[:, 1].min() - 1, x[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                        np.arange(y_min, y_max, h))
    Z = RF.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure()
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

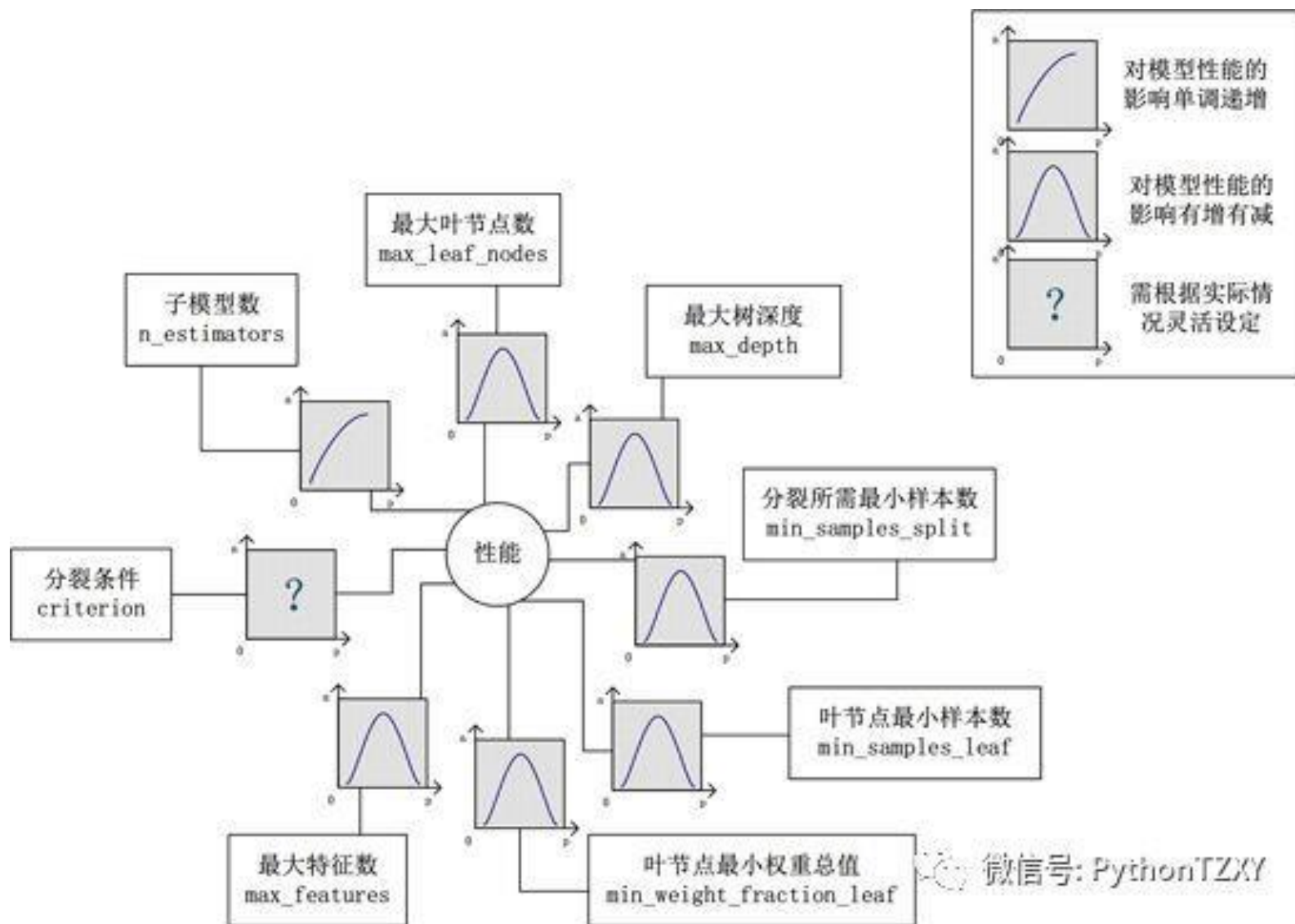
    # Plot also the training points
    plt.scatter(x[:, 0], x[:, 1], c=y, cmap=cmap_bold,
                edgecolor='k', s=20)
    plt.xlim(xx.min(), xx.max())
    plt.title("RandomForestClassifier")

plt.show()
print('RandomForestClassifier:', RF.score(x, y))
```



RandomForestClassifier: 0.9266666666666666

随机森林影响因素



随机森林优缺点分析

◆优点

- 由于采用了集成算法，本身精度比大多数单个算法要好，所以准确性高
- 由于两个随机性的引入（**样本随机**，**特征随机**），使得随机森林不容易陷入过拟合，并具有一定的抗噪声能力，对比其他算法具有一定优势
- 由于树的组合，使得随机森林可以处理非线性数据，本身属于非线性分类（拟合）模型
- 它能够处理很高维度（**feature**很多）的数据，并且不用做特征选择，对数据集的适应能力强：既能处理离散型数据，也能处理连续型数据，数据集无需规范化
- 训练速度快，可以运用在大规模数据集上
- 可以处理缺省值（单独作为一类），不用额外处理
- 由于有袋外数据（**OOB**），可以在模型生成过程中取得真实误差的无偏估计，且不损失训练数据量
- 在训练过程中，能够检测到**feature**间的互相影响，且可以得出**feature**的重要性，具有一定参考意义
- 由于每棵树可以独立、同时生成，容易做成并行化方法
- 由于实现简单、精度高、抗过拟合能力强，当面对非线性数据时，适于作为基准模型

◆缺点

- 当随机森林中的决策树个数很多时，训练时需要的空间和时间会比较大
- 随机森林中还有许多不好解释的地方，有点算是黑盒模型
- 在某些噪音比较大的样本集上，**RF**的模型容易陷入过拟合
- 对少量数据集和低维数据集的分类不一定可以得到很好的效果。

AdaBoost

输入：训练数据集 $T = \{(x_1, y_1), (x_2, y_2), (x_N, y_N)\}$ ，其中， $x_i \in X \subseteq R^n$ ， $y_i \in Y = -1, 1$ ，迭代次数 M

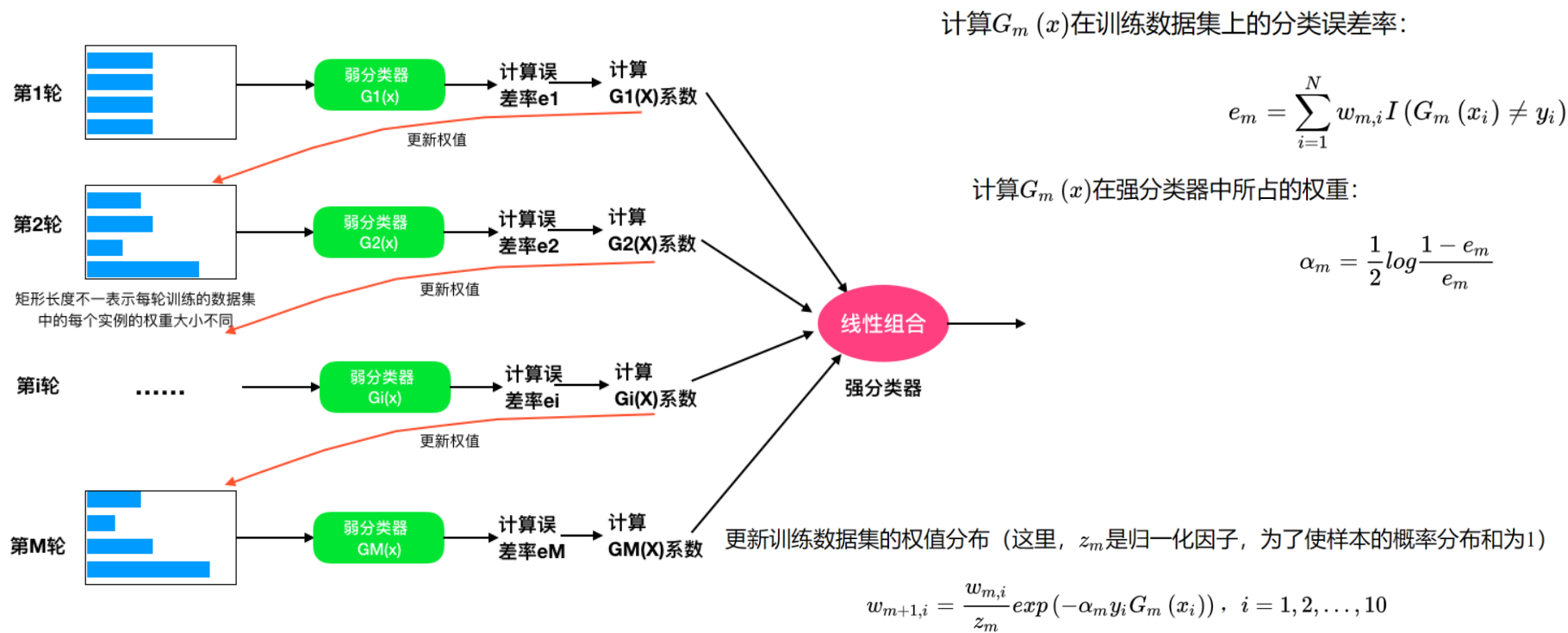


图2. AdaBoost算法示意图

得到最终分类器：

$$F(x) = \text{sign}\left(\sum_{i=1}^N \alpha_m G_m(x)\right)$$

AdaBoost算法解释

AdaBoost算法的最终模型表达式: $f(x) = \sum_{m=1}^M \alpha_m G_m(x)$ 加性模型(additive model)

我们希望这个模型在训练集上的经验误差最小, 即

$$\min \sum_{i=1}^N L(y_i, f(x_i)) \Leftrightarrow \min \sum_{i=1}^N L\left(y_i, \sum_{m=1}^M \alpha_m G_m(x_i)\right)$$

前向分步算法: 因为最终模型是一个加性模型, 如果能从前往后, 每一步只学习一个基学习及其权重, 不断迭代得到最终的模型, 那么就可以简化问题复杂度。

具体的, 当我们经过 $m-1$ 轮迭代得到了最优模型 $f_{m-1}(x)$ 时, 此轮优化目标为

$$\min \sum_{i=1}^N L(y_i, f_{m-1}(x_i) + \alpha_m G_m(x_i))$$

选取目标函数 $L(y, f(x)) = \exp(-yf(x))$

分步优化变量, 则有

$$w_{m,i} = \exp(-y_i f_{m-1}(x_i))$$

$$\hat{G}_m(x) = \operatorname{argmin}_{G_m(x)} \sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))$$

$$\hat{\alpha}_m = \frac{1}{2} \log \frac{1 - e_m}{e_m} \quad e_m = \frac{\sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_{m,i}} \quad \text{分类误差率}$$

AdaBoost代码

```
def my_adaboost_clf(Y_train, X_train, Y_test, X_test, M=20,
                    weak_clf=DecisionTreeClassifier(max_depth = 1)):
    n_train, n_test = len(X_train), len(X_test)
    # Initialize weights
    w = np.ones(n_train) / n_train
    pred_train, pred_test = [np.zeros(n_train), np.zeros(n_test)]
    for i in range(M):
        # Fit a classifier with the specific weights
        weak_clf.fit(X_train, Y_train, sample_weight = w)
        pred_train_i = weak_clf.predict(X_train)
        pred_test_i = weak_clf.predict(X_test)

        # Indicator function
        miss = [int(x) for x in (pred_train_i != Y_train)]
        print("weak_clf_%02d train acc: %.4f" % (i + 1, 1 - sum(miss) / n_train))
        # Error
        err_m = np.dot(w, miss)
        # Alpha
        alpha_m = 0.5 * np.log((1 - err_m) / float(err_m))
        # New weights
        miss2 = [x if x==1 else -1 for x in miss] # -1 * y_i * G(x_i): 1 / -1
        w = np.multiply(w, np.exp([float(x) * alpha_m for x in miss2]))
        w = w / sum(w)
        # Add to prediction
        pred_train_i = [1 if x == 1 else -1 for x in pred_train_i]
        pred_test_i = [1 if x == 1 else -1 for x in pred_test_i]
        pred_train = pred_train + np.multiply(alpha_m, pred_train_i)
        pred_test = pred_test + np.multiply(alpha_m, pred_test_i)

    pred_train = (pred_train > 0) * 1
    pred_test = (pred_test > 0) * 1
    print("My AdaBoost clf train accuracy: %.4f" % (sum(pred_train == Y_train) / n_train))
    print("My AdaBoost clf test accuracy: %.4f" % (sum(pred_test == Y_test) / n_test))
```

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features=4,
...                           n_informative=2, n_redundant=0,
...                           random_state=0, shuffle=False)
>>> clf = AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.fit(X, y)
AdaBoostClassifier(n_estimators=100, random_state=0)
>>> clf.feature_importances_
array([0.28..., 0.42..., 0.14..., 0.16...])
>>> clf.predict([[0, 0, 0, 0]])
array([1])
>>> clf.score(X, y)
0.983...
```

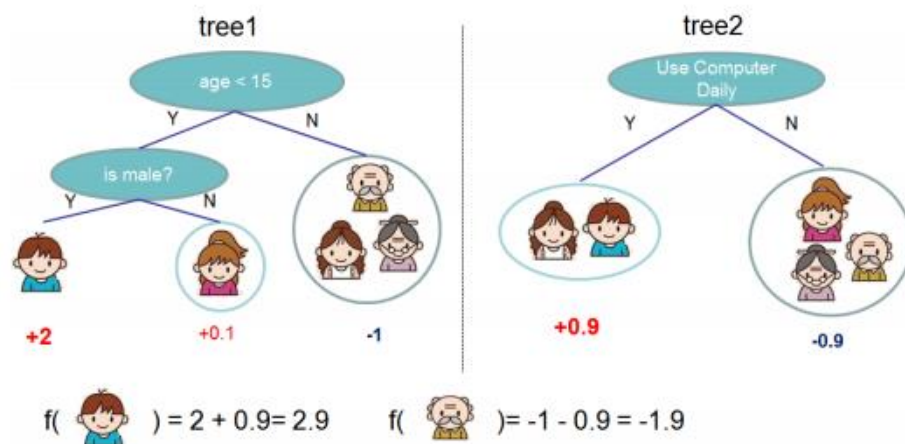
sklearn实现

GBDT (Gradient Boosting Decision Tree)

加性模型 $f(x) = \sum_{m=1}^M h_m(x; \beta_m)$

区别:

- AdaBoost模型每个基分类器的损失函数优化目标是相同的且独立的，都是最优化当前样本（样本权重）的指数损失。
- GBDT虽然也是一个加性模型，但其是通过不断迭代拟合样本真实值与当前分类器的残差来逼近真实值的。



注意GBDT中的DT本质上并不是决策树（ID3、C4.5、CART等等）而是回归树，为什么呢？

因为最终的预测结果是多个弱分类器的预测结果线性相加，回归树预测结果是连续值，最后相加才有意义。

其次需要注意的就是在构建回归树的过程中分支节点的选取及取值（对比分类树来看）

- 回归树不是像分类树那样采用最大熵来作为划分标准，而是采用**均方差**
- 回归树的每个叶子节点得到的不是像分类树那样的样本计数，而是属于这个叶子节点所有样本的平均值（根据**loss function**变化）。

GBDT梯度提升决策树算法是在决策树的基础上引入GB（逐步提升）和shrinkage（小幅缩进）两种思想，从而提升普通决策树的泛化能力。

GBDT (Gradient Boosting Decision Tree)

使用前向分步算法逐个训练学习器，按照泰勒一阶展开式将损失函数展开

$$L(y_i, f_{m-1}(x_i) + h(x_i; \beta_m)) = L(y_i, f_{m-1}(x_i)) + \frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)} h(x_i; \beta_m)$$

$$\text{令 } h(x_i; \beta_m) = - \frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}$$

$$\text{则 } L(y_i, f_{m-1}(x_i) + h(x_i; \beta_m)) = L(y_i, f_{m-1}(x_i)) - \left(\frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)} \right)^2$$

1 Initialize $f_0(x) = \operatorname{argmin}_c \sum_{i=1}^N L(y_i, c)$

2 For $m=1$ to M :

a. For $i=1$ to N compute

$$r_i^m = - \left[\frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)} \right]$$

b. Fit a regression tree $h_m(x; \beta_m)$ to the targets

r^m giving terminal regions R_{jm} , $j=1, \dots, J_m$.

c. Solve
$$\rho_{jm} = \operatorname{argmin}_{\rho} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \rho h_m(x_i; \beta_m))$$

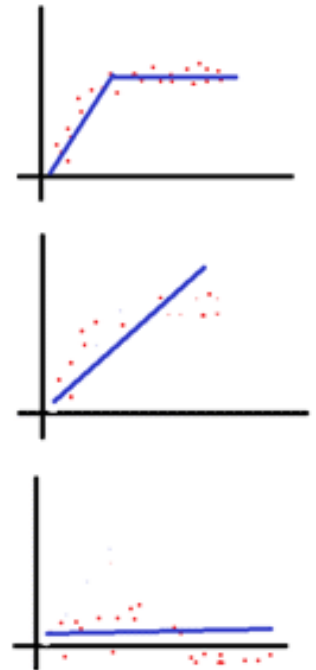
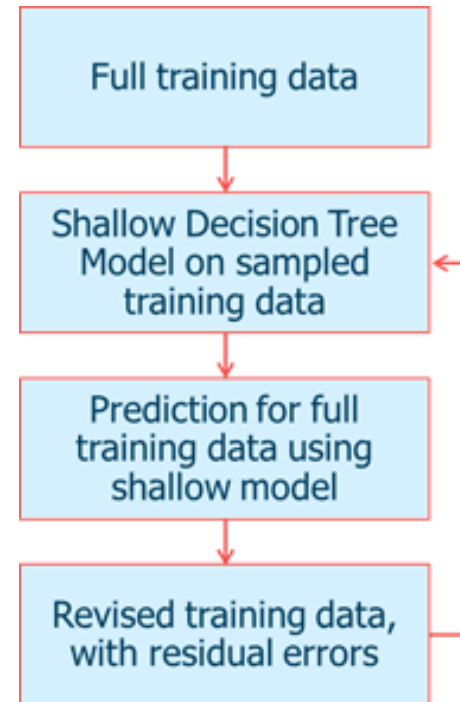
d. Update

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

3 Output

$$\hat{f}(x) = f_M(x)$$

GBDT算法 伪代码



常用损失函数-回归

1 平方损失

$$L(y, f(x)) = \frac{1}{2} (y - f(x))^2 \quad r_{mi} = (y_i - f_{m-1}(x_i)) \quad \text{残差}$$

2 绝对损失

$$L(y, f(x)) = |y - f(x)| \quad r_{mi} = \text{sign}(y_i - f_{m-1}(x_i))$$

3 Huber损失：平方损失和绝对损失的折衷产物，对于远离中心的异常点，采用绝对损失，而中心附近的点采用均方差。

$$L(y, f(x)) = \begin{cases} \frac{1}{2} (y - f(x))^2 & |y - f(x)| \leq \delta \\ \delta \left(|y - f(x)| - \frac{\delta}{2} \right) & |y - f(x)| > \delta \end{cases}$$
$$r_{mi} = \begin{cases} y_i - f(x_i) & |y_i - f(x_i)| \leq \delta \\ \delta (\text{sign}(y_i - f(x_i)) - 1) & |y_i - f(x_i)| > \delta \end{cases}$$

4 分位数损失

$$L(y, f(x)) = \begin{cases} \theta |y - f(x)| & y \geq f(x) \\ (1 - \theta) |y - f(x)| & y < f(x) \end{cases}$$

$$r_{mi} = \begin{cases} \theta & y_i \geq f(x_i) \\ (\theta - 1) & y_i < f(x_i) \end{cases}$$

Huber损失和分位数损失，主要用于减少异常点对损失函数的影响

GBDT常用损失函数-分类

1) 指数损失函数

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

$$r_{mi} = -y_i \exp(-y_i f(x_i))$$

2) 对数损失函数

二分类 $L(y, f(x)) = \log[1 + \exp(-yf(x))]$

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

多分类

$$L(y, f(x)) = - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log p_k(x_i)$$

$$r_{mil} = y_{il} - p_l^{m-1}(x_i)$$

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

GBDT正则化

1) 步长

$$f_m(x) = f_{m-1}(x) + \nu h_m(x; \beta_m) \quad \nu \in (0,1]$$

对于同样的训练集学习效果，较小的意味着我们需要更多的弱学习器的迭代次数。

2) 调整子采样比例(Stochastic Gradient Boosting Tree)，推荐[0.5, 0.8]

3) 对于弱学习器即CART回归树进行正则化剪枝。

GBDT优缺点

◆ 1) 优点

- 可以灵活处理各种类型的数据，包括连续值和离散值。
- 在相对少的调参时间情况下，预测的准确率也可以比较高。这个是相对SVM来说的。
- 使用一些健壮的损失函数，对异常值的鲁棒性非常强。比如 Huber 损失函数和 Quantile 损失函数。

◆ 缺点

- 由于弱学习器之间存在依赖关系，难以并行训练数据。不过可以通过自采样的SGBT来达到部分并行。

损失函数

$$Obj = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{j=1}^T \Omega(f_j)$$

$\Omega(f_t)$ 为第j棵树 f_j 的复杂度，防止过拟合

由于

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

则

$$\begin{aligned} Obj^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j) \\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \sum_{j=1}^t \Omega(f_j) \\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + const \end{aligned}$$

只考虑第t棵树

泰勒公式展开

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2$$

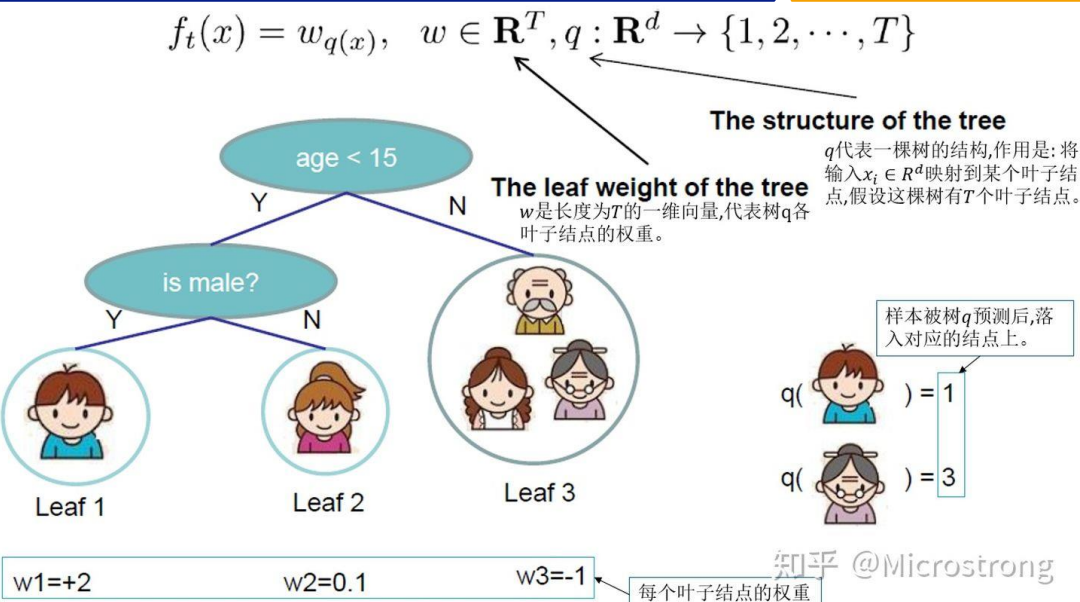
$$g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$$

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

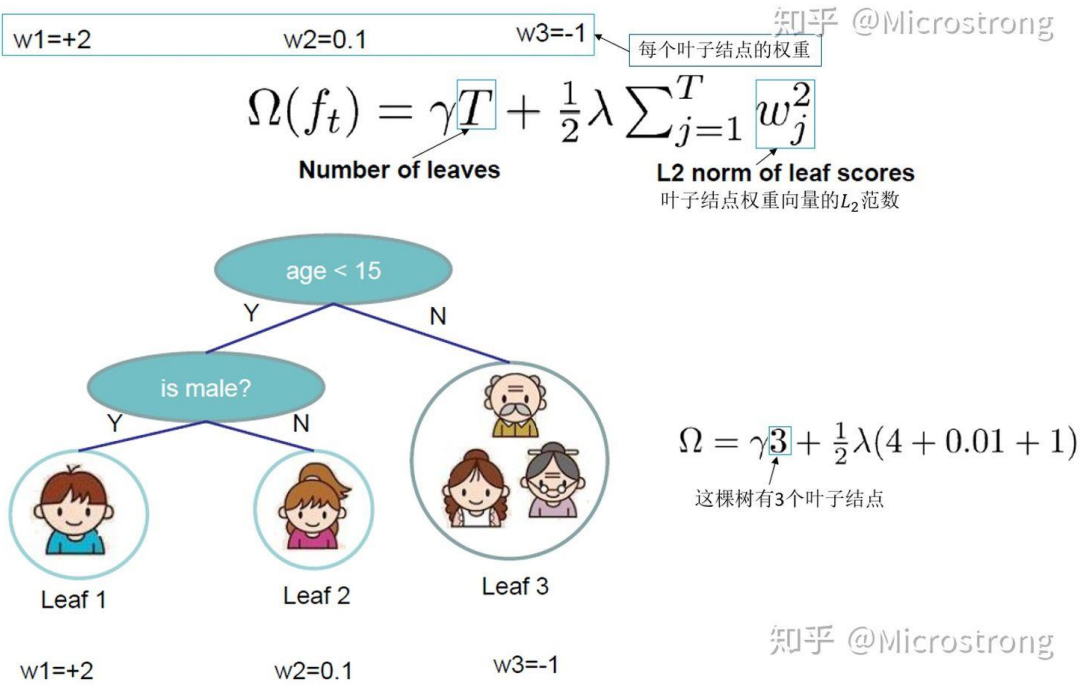
去掉常数项

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

决策树重新定义



定义树的复杂度



XGBOOST

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

第二行是遍历所有的样本后求每个样本的损失函数

由于样本最终会落在叶子节点上，所以也可以遍历叶子节点，然后获取叶子节点上的样本集合，最后再求损失函数。

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

则
$$Obj^{(t)} = \sum_{j=1}^T \left[G_j \omega_j + \frac{1}{2} (H_j + \lambda) \omega_j^2 \right] + \gamma T$$






求出最优解

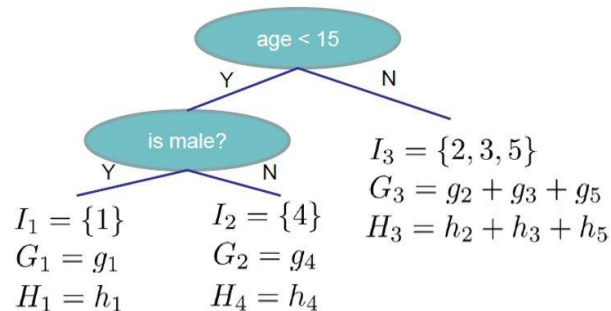
$$\omega_j^* = - \frac{G_j}{H_j + \lambda}$$

则目标函数

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

Instance index gradient statistics

1		g_1, h_1
2		g_2, h_2
3		g_3, h_3
4		g_4, h_4
5		g_5, h_5



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

The smaller the score is, the better the structure is
这个分数越小,代表这个树的结构越好。

如何计算每个特征的分裂收益

假设我们在某一节点完成特征分裂，则分裂前的目标函数可以写为：

$$Obj_1 = -\frac{1}{2} \left[\frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] + \gamma$$

分裂后的目标函数为：

$$Obj_2 = -\frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} \right] + 2\gamma$$

则对于目标函数来说，分裂后的收益为：

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

XGBOOST代码

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
import xgboost as xgb
from sklearn.metrics import mean_absolute_error

# 1.读文件
data = pd.read_csv('./dataset/train.csv')
data.dropna(axis=0, subset=['SalePrice'], inplace=True)

# 2.切分数据输入：特征 输出：预测目标变量
y = data.SalePrice
X = data.drop(['SalePrice'], axis=1).select_dtypes(exclude=['object'])

# 3.切分训练集、测试集,切分比例7.5 : 2.5
train_X, test_X, train_y, test_y = train_test_split(X.values, y.values, test_size=0.25)

# 4.空值处理，默认方法：使用特征列的平均值进行填充
my_imputer = SimpleImputer()
train_X = my_imputer.fit_transform(train_X)
test_X = my_imputer.transform(test_X)

# 5.调用XGBoost模型，使用训练集数据进行训练（拟合）
# Add verbosity=2 to print messages while running boosting
my_model = xgb.XGBRegressor(objective='reg:squarederror', verbosity=2) # xgb.XGBClassifier() XGBoost分类模型
my_model.fit(train_X, train_y, verbose=False)

# 6.使用模型对测试集数据进行预测
predictions = my_model.predict(test_X)

# 7.对模型的预测结果进行评判（平均绝对误差）
print("Mean Absolute Error : " + str(mean_absolute_error(predictions, test_y)))
```

XGBOOST与GBDT的关系

- (1) GBDT是机器学习算法，XGBoost是该算法的工程实现。
- (2) **正则项：**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。
- (3) **导数信息：**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。
- (4) **基分类器：**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。
- (5) **子采样：**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。
- (6) **缺失值处理：**传统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。
- (7) **并行化：**传统GBDT没有进行并行化设计，注意不是tree维度的并行，而是特征维度的并行。XGBoost预先将每个特征按特征值排好序，存储为块结构，分裂结点时可以采用多线程并行查找每个特征的最佳分割点，极大提升训练速度。

LightGBM

XGBOOST缺点:

- 虽然利用预排序和近似算法可以降低寻找最佳分裂点的计算量，但在节点分裂过程中仍需要遍历数据集；
- 预排序过程的空间复杂度过高，不仅需要存储特征值，还需要存储特征对应样本的梯度统计值的索引，相当于消耗了两倍的内存。

LightGBM方案

- 1.单边梯度抽样算法；
- 2.直方图算法；
- 3.互斥特征捆绑算法；
- 4.基于最大深度的 Leaf-wise 的垂直生长算法；
- 5.类别特征最优分割；
- 6.特征并行和数据并行；
- 7.缓存优化。

Gradient-based One-Side Sampling (GOSS): GBDT虽然没有数据权重，但每个数据实例有不同的梯度，根据计算信息增益的定义，梯度大的实例对信息增益有更大的影响，因此在下采样时，我们应该尽量保留梯度大的样本（预先设定阈值，或者最高百分位间），随机去掉梯度小的样本。证明此措施在相同的采样率下比随机采样获得更准确的结果，尤其是在信息增益范围较大时。

Exclusive Feature Bundling (EFB): 通常真实应用中，虽然特征量比较多，但是由于特征空间十分稀疏；特别在稀疏特征空间上，许多特征几乎是互斥的（例如许多特征不会同时为非零值，像one-hot），可以将一些特征进行融合绑定，降低特征数量。最后，将捆绑问题归约到图着色问题，通过贪心算法求得近似解。

LightGBM: Gradient-based One-Side Sampling (GOSS)

定义：0表示某个固定节点的训练集，分割特征j的分割点d定义为：

$$V_{J|O}(d) = \frac{1}{n_O} \left[\frac{\left(\sum_{\{x_i \in O: x_{ij} \leq d\}} g_i \right)^2}{n_{l|O}^j(d)} + \frac{\left(\sum_{\{x_i \in O: x_{ij} > d\}} g_i \right)^2}{n_{r|O}^j(d)} \right]$$

$$n_O = \sum I(x_i \in O) \quad n_{l|O}^j(d) = \sum I(\{x_i \in O: x_{ij} \leq d\}) \quad n_{r|O}^j(d) = \sum I(\{x_i \in O: x_{ij} > d\})$$

GOSS

1. 首先根据数据的梯度将训练降序排序;
2. 保留top a个数据实例，作为数据子集A;
3. 对于剩下的数据的实例，随机采样获得大小为b的数据子集B;
4. 遍历每个特征的每个分裂点，找到 $d_j^* = \underset{-d}{\operatorname{argmax}} V_j(d)$ 并计算最大的信息增益 $V_j(d_j^*)$ ，然后，将数据根据特征j的分裂点 d_j^* 将数据分到左右子节点。

LightGBM: Exclusive Feature Bundling

需要解决两个问题:

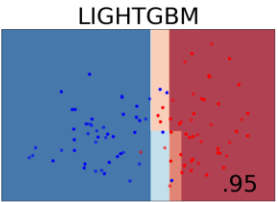
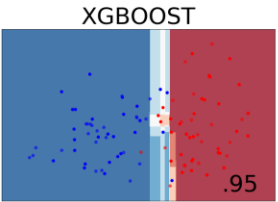
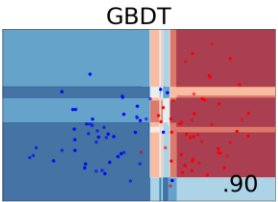
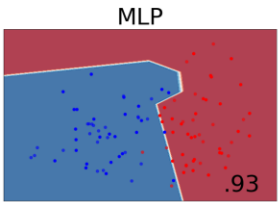
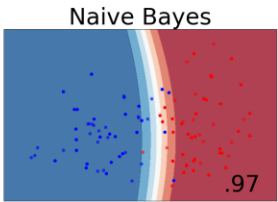
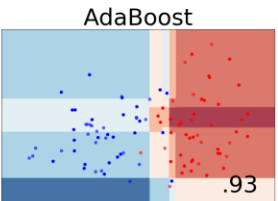
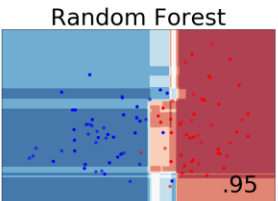
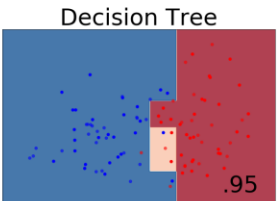
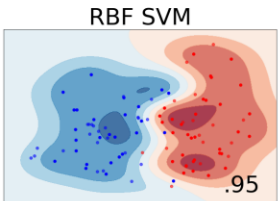
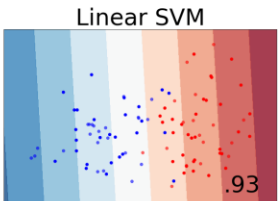
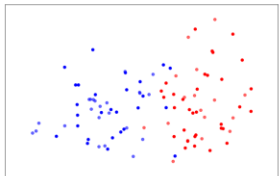
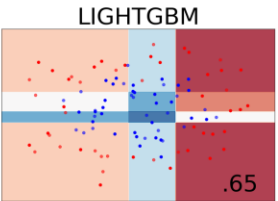
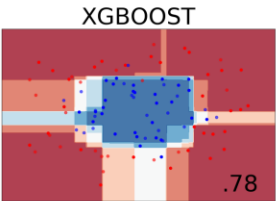
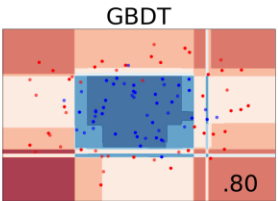
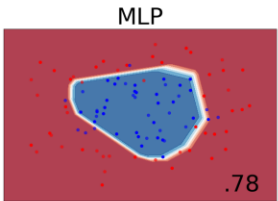
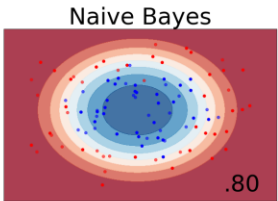
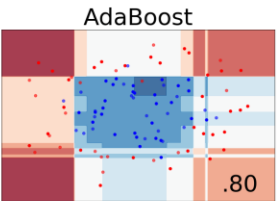
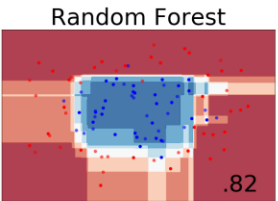
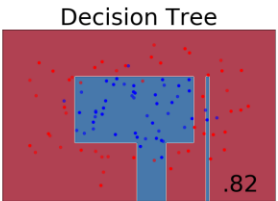
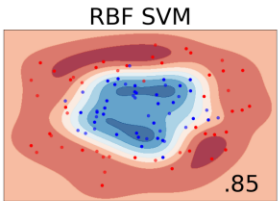
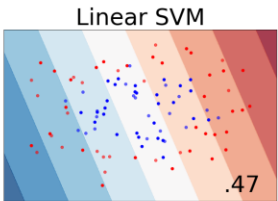
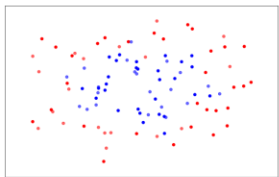
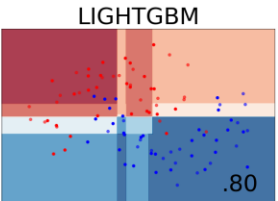
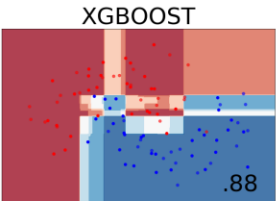
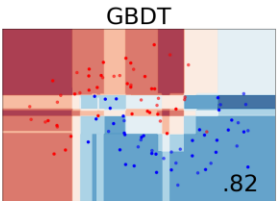
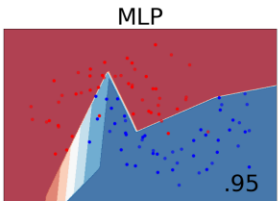
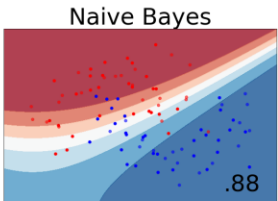
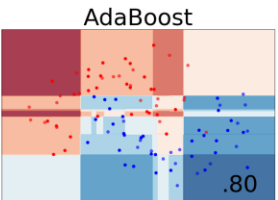
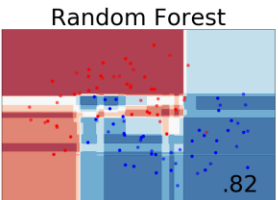
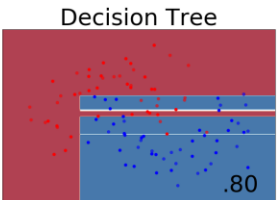
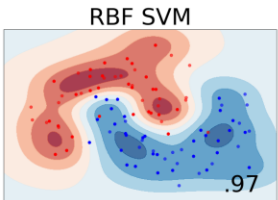
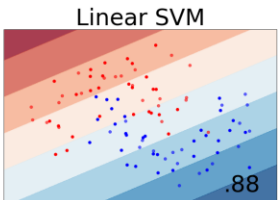
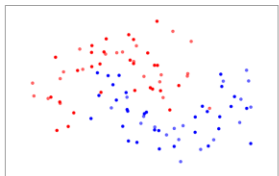
- 怎么判定那些特征应该绑在一起 (build bundled) ?
- 怎么把特征绑为一个 (merge feature) ?

问题一： 利用特征和特征间的关系构造一个加权无向图，并将其转换为图着色算法。我们知道图着色是个 **NP-Hard** 问题，故采用贪婪算法得到近似解，具体步骤如下：

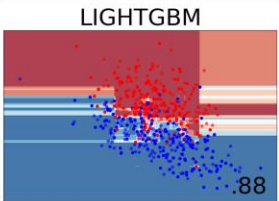
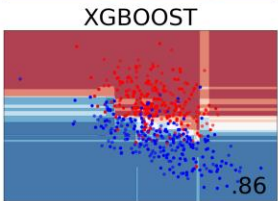
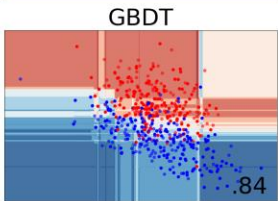
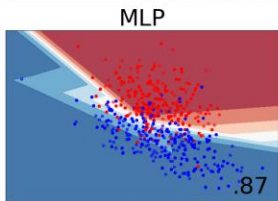
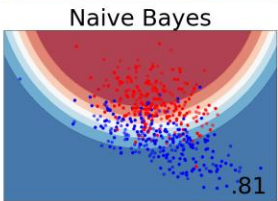
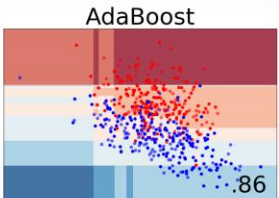
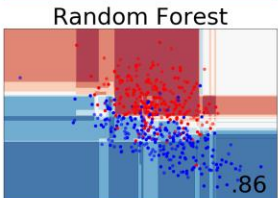
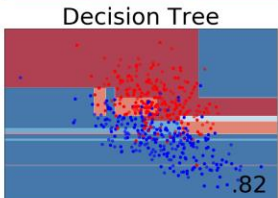
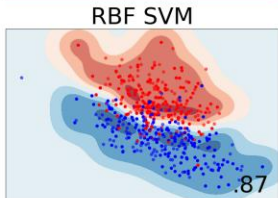
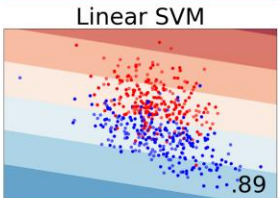
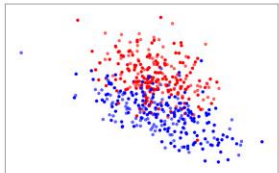
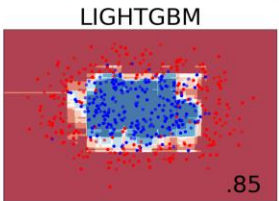
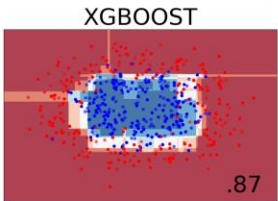
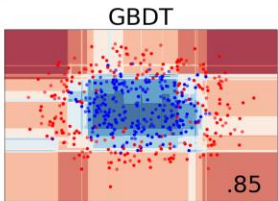
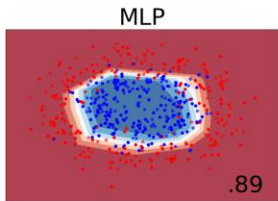
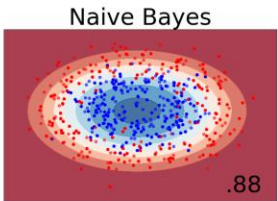
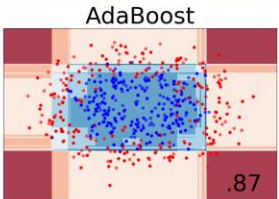
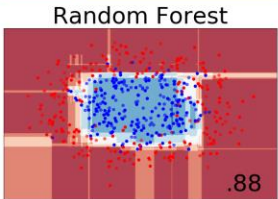
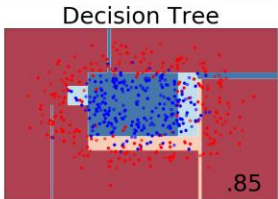
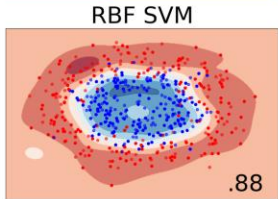
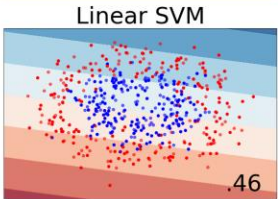
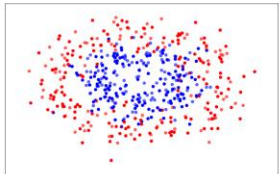
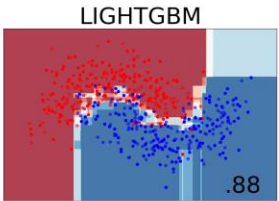
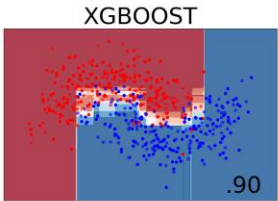
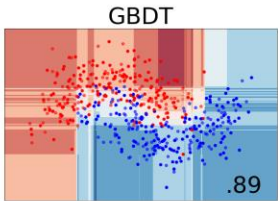
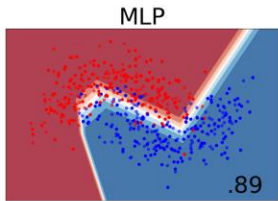
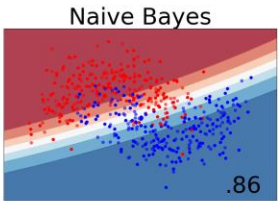
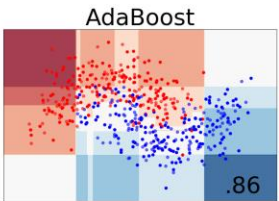
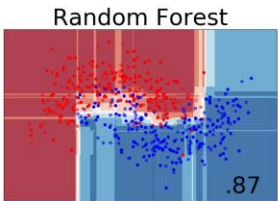
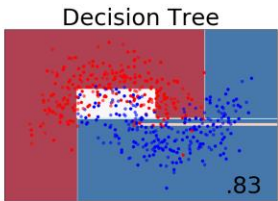
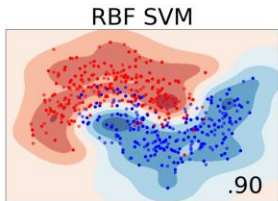
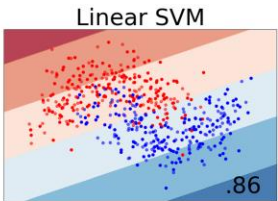
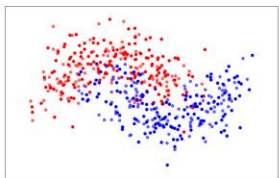
1. 构造一个加权无向图，顶点是特征，边是两个特征间互斥程度；
2. 根据节点的度进行降序排序，度越大，与其他特征的冲突越大；
3. 遍历每个特征，将它分配给现有特征包，或者新建一个特征包，保证总体冲突最小。

问题二： 如何合并同一个bundle的特征来降低训练时间复杂度。关键在于原始特征值可以从bundle中区分出来。鉴于直方图算法存储离散值而不是连续特征值，我们通过将互斥特征放在不同的箱中来构建bundle。这可以通过将偏移量添加到特征原始值中实现，例如，假设bundle中有两个特征，原始特征A取值 $[0, 10]$ ，B取值 $[0, 20]$ 。我们添加偏移量10到B中，因此B取值 $[10, 30]$ 。

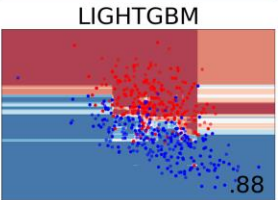
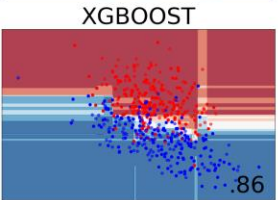
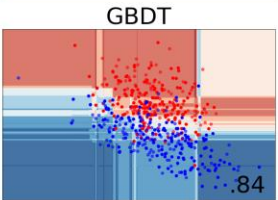
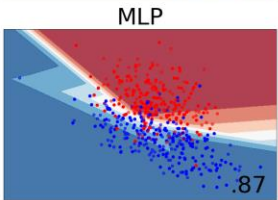
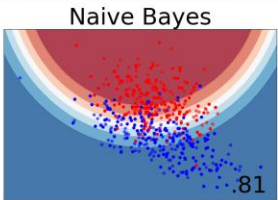
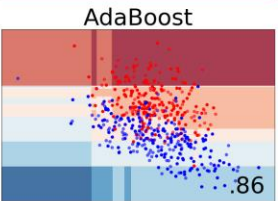
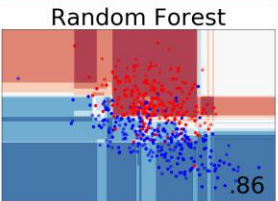
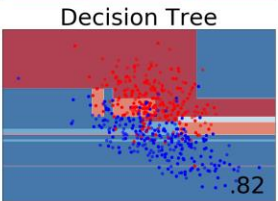
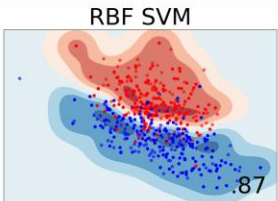
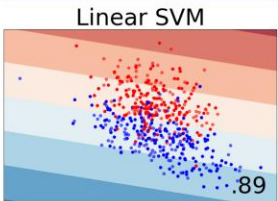
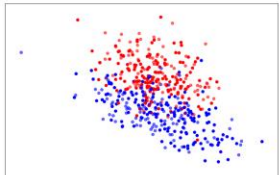
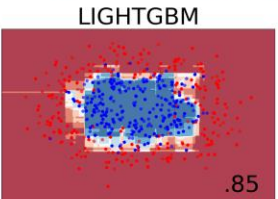
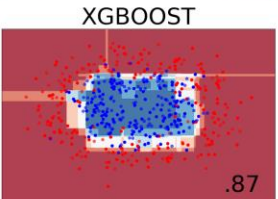
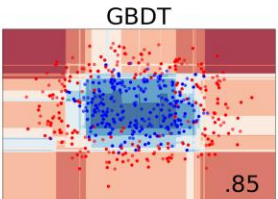
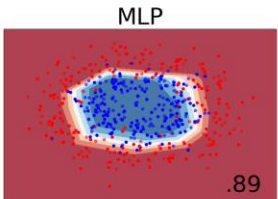
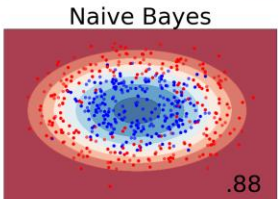
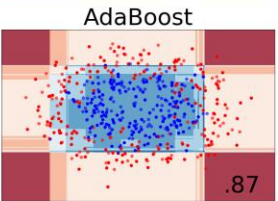
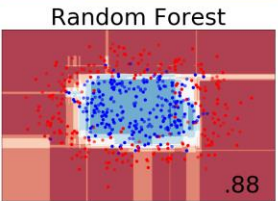
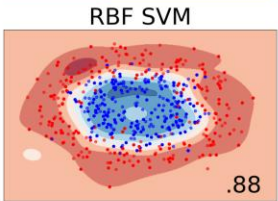
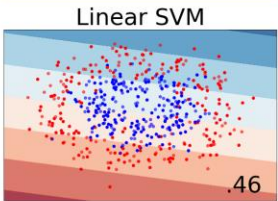
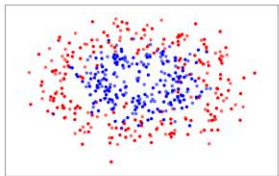
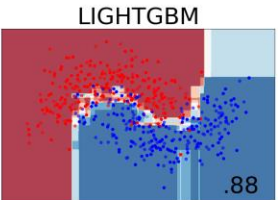
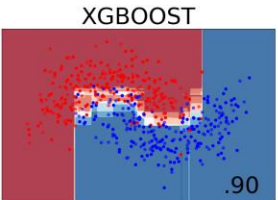
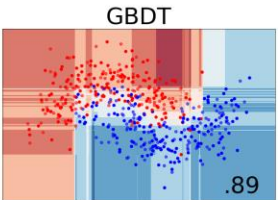
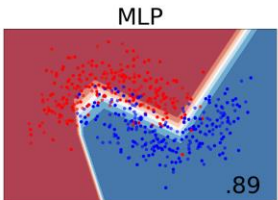
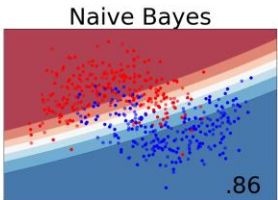
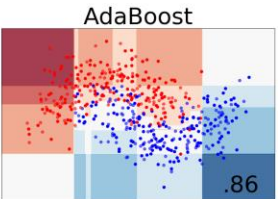
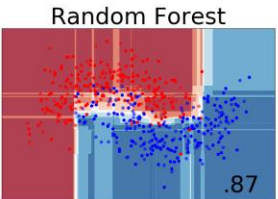
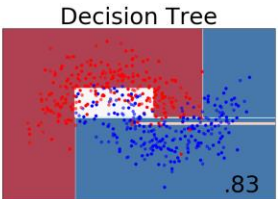
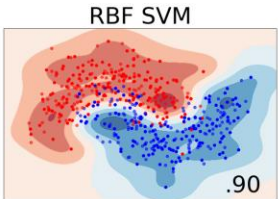
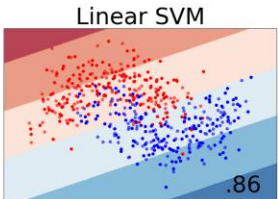
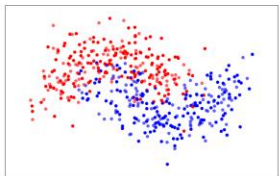
多算法效果对比-100样本



多算法效果对比-500样本



多算法效果对比-1000样本



参考资料

<https://www.kaggle.com/arthurtok/introduction-to-ensembling-stacking-in-python>

Thank You!