关于集成学习

集成学习

如果硬要把集成学习进一步分类，可以分为两类，一种是把强分类器进行强强联合，使得融合后的模型效果更强，称为**模型融合**。另一种是将弱分类器通过学习算法集成起来变为很强的分类器，称为**机器学习元算法**。

这里我们把用来进行融合的学习器称为**个体学习器（教材里的基学习器）**。

模型融合的代表有：投票法(Voting)、线性混合(Linear Blending)、Stacking。

而机器学习元算法又可以根据个体学习器之间是否存在依赖关系分为两类，称为Bagging和Boosting:

Bagging: 个体学习器不存在依赖关系，可同时对样本随机采样并行化生成个体学习器。代表作为随机森林(Random Forest)
Boosting: 个体学习器存在依赖关系,基于前面模型的训练结果误差生成新的模型，必须串行化生成。代表的算法有：Adaboost、GBDT、XGBoost

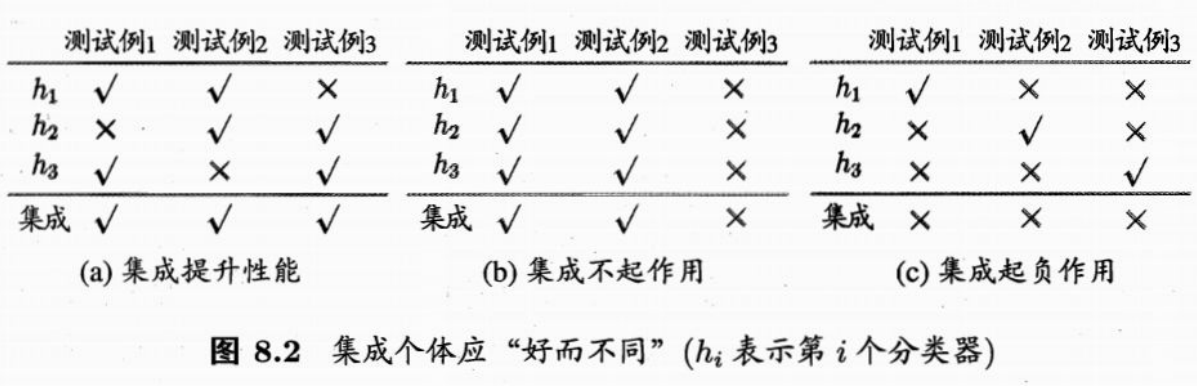
模型融合

上面提到，模型融合是把强分类器进行强强联合，变得更强。

在进行模型融合的时候，也不是说随意的融合就能达到好的效果。进行融合时，所需的集成个体（就是用来集成的模型）应该**好而不同**。好指的是个体学习器的性能要好，不同指的是个体模型的类别不同。

这里举个西瓜书的例子，在介绍例子之前，首先提前介绍简单投票法，以分类

问题为例，就是每个分类器对样例进行投票，哪个类别得到的票数最多的就是融合后模型的结果。



	测试例1	测试例2	测试例3
h_1	√	√	×
h_2	×	√	√
h_3	√	×	√
集成	√	√	√

(a) 集成提升性能

	测试例1	测试例2	测试例3
h_1	√	√	×
h_2	√	√	×
h_3	√	√	×
集成	√	√	×

(b) 集成不起作用

	测试例1	测试例2	测试例3
h_1	√	×	×
h_2	×	√	×
h_3	×	×	√
集成	×	×	×

(c) 集成起负作用

图 8.2 集成个体应“好而不同”(h_i 表示第 i 个分类器)

在上面的例子中，采用的就是简单的投票法。中间的图b各个模型输出都一样，因此没有什么效果。第三个图c每个分类器的精度只有33%，融合后反而更糟。也就是说，想要模型融合有效果，个体学习器要有一定的准确率，并且要有多样性，学习器之间具有差异，即”好而不同“。

如何做到好而不同呢？可以由下面几个方面：

针对输入数据：使用采样的方法得到不同的样本（比如bagging方法采用自助法进行抽样）
针对特征：对特征进行抽样
针对算法本身：
个体学习器 h_t 来自不**同的模型集合**
个体学习器 h_t 来自于同一个模型集合的**不同超参数**，例如学习率η不同
算法本身具有随机性，例如用不同的随机种子来得到不同的模型
针对输出：对输出表示进行操纵以增强多样性
如将多分类转化为多个二分类任务来训练单模型
将分类输出转化为回归输出等

投票和平均 Voting and Average

分类

对于**分类任务**来说，可以使用**投票**的方法：

简单投票法： $H(\mathbf{x}) = c_{\underset{x}{\operatorname{argmin}}} \sum_{i=1}^{T} h_i^j(\mathbf{x})$
即各个分类器输出其预测的类别，取最高票对应的类别作为结果。若有多个类别都是最高票，那么随机选取一个。
加权投票法： $H(\mathbf{x}) = c_{\underset{x}{\operatorname{argmin}}} \sum_{i=1}^{T} \alpha_i \cdot h_i^j(\mathbf{x})$
和上面的简单投票法类似，不过多了权重αi，这样可以区分分类器的重要程度，通常 $\alpha_i \geq 0;\ \sum_{i=1}^{T} \alpha_i = 1$

此外，个体学习器可能产生不同的 $h_i^j(\mathbf{x})$ 的值，比如类标记和类概率。

类标记 $h_i^j(\mathbf{x}) \in \{0,1\}$ ，若hi将样本**x**预测为类别 c_j 取值为1，否则为0。使用类标记的投票亦称“硬投票”。(其实就是多分类的输出)，使用类标记的称为**硬投票**
类概率 $h_i^j(\mathbf{x}) \in [0,1]$ ，即输出类别为 c_j 的概率。使用类概率的投票称为**软投票。对应sklearn中的VotingClassifier中voting参数设为soft。**

PS：**使用类概率进行结合往往比直接基于类标记的效果好，即使分类器估计出的概率值一般都不太准确。**

回归

对于**回归任务**来说，采用的为**平均法(Average)**：

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

加权平均： $H(\mathbf{x}) = \frac{1}{T}\sum_{i=1}^{T} \alpha_i \cdot h_i(\mathbf{x});\ \ \alpha_i \geq 0;\ \ \sum_{i=1}^{T} \alpha_i = 1$

> **首先我们回顾两个概念：**

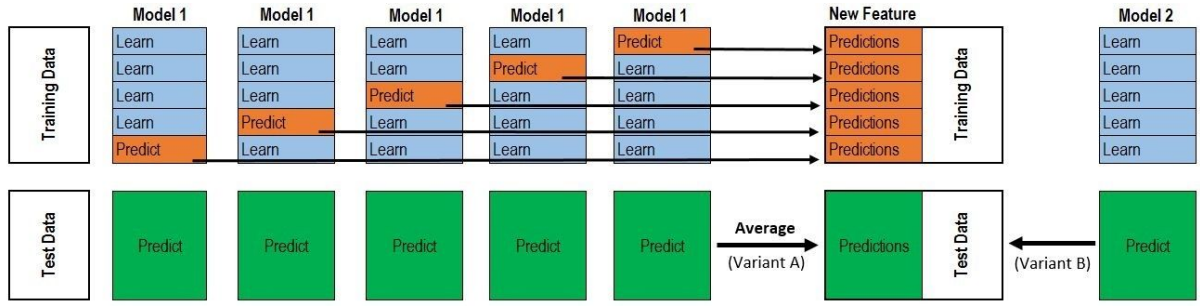
> **holdout**和**交叉验证**是两个常用的评估分类器预测准确率的技术，它们均是在给定数据集中随机取样划分数据。 holdout：将所给定的数据集随机划分成两个独立部分：一个作为训练数据集，而另一个作为测试数据集，通常训练数据集包含初始数据集中的三分之二的数据，而其余的三分之一则作为测试数据集的内容。利用训练集数据学习获得一个分类器，然后使用测试数据集对该分类器预测准确率进行评估，由于仅使用初始数据集中的一部分进行学习，因此对所得分类器预测准确性的估计应该是悲观的估计。 随机取样是holdout方法的一种变化，在随机取样方法中，重复利用holdout方法进行预测准确率估计k次，最后对这k次所获得的预测准确率求平均，以便获得最终的预测准确率。 k-交叉验证：将初始数据集随机分为k个互不相交的子集，S1，S2，...,Sk,每个子集大小基本相同。学习和测试分别进行k次，在第i次循环中，子集Si作为测试集，其他子集则合并到一起构成一个大训练数据集并通过学习获得相应的分类器，也就是第一次循环，使用S2....Sk作为训练数据集，S1作为测试数据集；而在第二次循环时，使用S1，S3，...,Sk作为训练数据集，S2作为测试数据集；如此下去等等。而对整个初始数据所得分类器的准确率估计则可用k次循环中所获得的正确分类数目之和除以初始数据集的大小来获得。在分层交叉验证中，将所划分的子集层次化以确保每个子集中的各类别分布与初始数据集中的类别分布基本相同。

Stacking

Stacking相比Linear Blending来说，更加强大，然而也更容易过拟合。

Stacking做法和Linear Blending类似，首先从数据集中训练出初级学习器，然后”生成“一个新的数据集用于训练次级学习器。为了防止过拟合，采用K折交叉验证法求解。

一个直观的图如下：



假设采用5折交叉验证，每个模型都要做满5次训练和预测，对于每一次：

从80%的数据训练得到一个模型ht，然后预测训练集剩下的那20%，同时也要预测测试集。
每次有20%的训练数据被预测，5次后正好每个训练样本都被预测过了。
每次都要预测测试集，因此最后测试集被预测5次，最终结果取5次的平均。

stacking例子：

> 对于每一轮的 5-fold，Model 1都要做满5次的训练和预测。
> Titanic 栗子：
> Train Data有890行。(请对应图中的上层部分)
> 每1次的fold，都会生成 713行 小train， 178行 小test。我们用Model 1来训练713行的小train，然后预测 178行 小test。预测的结果是长度为 178 的预测值。
> 这样的动作走5次！ 长度为178 的预测值 X 5 = 890 预测值，刚好和Train

> data长度吻合。这个890预测值是Model 1产生的，我们先存着，因为，一会让它将是第二层模型的训练来源。
> 重点：这一步产生的预测值我们可以转成 890 X 1 （890 行，1列），记作 P1 (大写P)
> 接着说 Test Data 有 418 行。(请对应图中的下层部分，对对对，绿绿的那些框框)
> 每1次的fold，713行 小train训练出来的Model 1要去预测我们全部的Test Data（全部！因为Test Data没有加入5-fold，所以每次都是全部！）。此时，Model 1的预测结果是长度为418的预测值。
> 这样的动作走5次！我们可以得到一个 5 X 418 的预测值矩阵。然后我们根据行来就平均值，最后得到一个 1 X 418 的平均预测值。
> 重点：这一步产生的预测值我们可以转成 418 X 1 （418行，1列），记作 p1 (小写p)
> 走到这里，你的第一层的Model 1完成了它的使命。
> 第一层还会有其他Model的，比如Model 2，同样的走一遍， 我们有可以得到 890 X 1 (P2) 和 418 X 1 (p2) 列预测值。
> 这样吧，假设你第一层有3个模型，这样你就会得到：
> 来自5-fold的预测值矩阵 890 X 3， （P1，P2， P3） 和 来自Test Data预测值矩阵 418 X 3， （p1, p2, p3）。
> ----------
> 到第二层了.....
> 来自5-fold的预测值矩阵 890 X 3 作为你的Train Data，训练第二层的模型
> 来自Test Data预测值矩阵 418 X 3 就是你的Test Data，用训练好的模型来预测他们吧。

回归问题，代码如下（get_oof就是上图的过程）：

```
_N_FOLDS = 5  # 采用5折交叉验证
kf = KFold(n_splits=_N_FOLDS, random_state=42)  # sklearn的交叉验证

def get_oof(clf, X_train, y_train, X_test):
    # X_train: 1000 * 10
    # y_train: 1 * 1000
```

```
    # X_test : 500 * 10
    oof_train = np.zeros((X_train.shape[0], 1))  # 1000 * 1  Stack
    oof_test_skf = np.empty((_N_FOLDS, X_test.shape[0], 1))  # 5 *

    for i, (train_index, test_index) in enumerate(kf.split(X_train
        kf_X_train = X_train[train_index]  # 800 * 10 训练集
        kf_y_train = y_train[train_index]  # 1 * 800 训练集对应的输出
        kf_X_val = X_train[test_index]  # 200 * 10  验证集

        clf.fit(kf_X_train, kf_y_train)  # 当前模型进行训练

        oof_train[test_index] = clf.predict(kf_X_val).reshape(-1,
        oof_test_skf[i, :] = clf.predict(X_test).reshape(-1, 1)  #

    oof_test = oof_test_skf.mean(axis=0)  # 对每一则交叉验证的结果取平
    return oof_train, oof_test  # 返回当前分类器对训练集和测试集的预测结


# 将数据换成你的数据
X_train = np.random.random((1000, 10))  # 1000 * 10
y_train = np.random.random_integers(0, 1, (1000,))  # 1000
X_test = np.random.random((500, 10))  # 500 * 10

# 将你的每个分类器都调用get_oof函数，并把它们的结果合并，就得到了新的训练和测
new_train, new_test = [], []
for clf in [LinearRegression(), RandomForestRegressor()]:
    oof_train, oof_test = get_oof(clf, X_train, y_train, X_test)
    new_train.append(oof_train)
    new_test.append(oof_test)

new_train = np.concatenate(new_train, axis=1)
new_test = np.concatenate(new_test, axis=1)

# 用新的训练数据new_train作为新的模型的输入，stacking第二层
clf = RandomForestRegressor()
clf.fit(new_train, y_train)
clf.predict(new_test)
```

如果是分类问题，我们对测试集的结果就不能像回归问题一样直接取平均，而是分类器输出所有类别的概率，最后取平均。每个分类器都贡献了_N_CLASS(类别数)的维度。

修改get_oof函数如下即可：

```
_N_CLASS = 2
def get_oof(clf, X_train, y_train, X_test):
    # X_train: 1000 * 10
    # y_train: 1 * 1000
    # X_test : 500 * 10
    oof_train = np.zeros((X_train.shape[0], _N_CLASS))  # 1000 * _
    oof_test = np.empty((X_test.shape[0], _N_CLASS))  # 500 * _N_C

    for i, (train_index, test_index) in enumerate(kf.split(X_train
        kf_X_train = X_train[train_index]  # 800 * 10 交叉验证划分此
        kf_y_train = y_train[train_index]  # 1 * 800
        kf_X_test = X_train[test_index]  # 200 * 10 验证集

        clf.fit(kf_X_train, kf_y_train)  # 当前模型进行训练

        oof_train[test_index] = clf.predict_proba(kf_X_test)  # 当
        oof_test += clf.predict_proba(X_test)  # 对测试集概率预测 oo

    oof_test /= _N_FOLDS  # 对每一则交叉验证的结果取平均
    return oof_train, oof_test  # 返回当前分类器对训练集和测试集的预测结
```

上面的代码只做了两层，你想的话还可以在加几层，因此这个方法叫做stacking，堆叠。。

线性混合 Linear Blending

前面提到过加权平均法，每个个体学习器的权重不再相等，看起来就像**是对每**

个个体学习器做一个线性组合，这也是线性混合法名字的由来。那么最优的权重是什么呢？一个直接的想法就是最好的αi使得error最小，即对应了优化问题：

$$\min_{\alpha_t \geq 0} \frac{1}{M} \sum_{i=1}^{M} \left(y_i - \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}_i) \right)^2$$

这里有T个个体学习器，每个学习器用 h_t 表示，而αt就是对应的权重。

这里我们首先用训练数据训练出所有的h，然后再做**线性回归**求出 α_t 。注意到这里要求 $\alpha_t \geq 0$ ，来个拉格朗日函数？其实不用，通常我们可以忽略这个条件。以二分类为例如果αi小于0，相当于把模型反过来用。（假如给你个错误率99%的模型，你反过来用正确率不就99%了么！）

如何得到 h_t 呢？这里我们将个体学习器称为**初级学习器**，用于结合的学习器称为**次级学习器**。**首先从数据集中训练出初级学习器，然后”生成“一个新的数据集用于训练次级学习器**。注意为了防止过拟合，我们需要在训练集上做训练得到初级学习器ht，而在验证集上比较不同α的好坏。最终模型则在所有的数据上进行训练（数据量多可能使得模型效果更好）

步骤如下：

从**训练集**Dtrain中训练得到 $h_1^-, h_2^-, \cdots, h_t^-$ ，并对**验证集** D_{val} 中的数据 $(\mathbf{x_i}, y_i)$ 做转换为新的数据集 $(\Phi^-(\mathbf{x_i}), y_i)$ ，其中
$\Phi^-(\mathbf{x_i}) = (h_1^-(\mathbf{x_i}), h_2^-(\mathbf{x_i}), \cdots, h_t^-(\mathbf{x_i}))$
用线性回归求解 $\alpha = Lin(\{(z_i, y_i)\})$
最后，用所有的数据D求解得到 $h_1, h_2, \cdots, h_t$ ，组成特征变换向量
$\Phi(\mathbf{x}) = (h_1(\mathbf{x}), h_2(\mathbf{x}), \cdots, h_t(\mathbf{x}))$
对于新数据x， $f(\mathbf{x}) = \frac{1}{T} \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$

Blending与Stacking大致相同，只是Blending的主要区别在于训练集不是通过K-

Fold的CV策略来获得预测值从而生成第二阶段模型的特征，而是建立一个Holdout集，例如10%的训练数据，第二阶段的stacker模型就基于第一阶段模型对这10%训练数据的预测值进行拟合。说白了，就是把Stacking流程中的K-Fold CV 改成 HoldOut CV。

Blending的优点在于：

比stacking简单（因为不用进行k次的交叉验证来获得stacker feature）
避开了一个信息泄露问题：generlizers和stacker使用了不一样的数据集
在团队建模过程中，不需要给队友分享自己的随机种子

而缺点在于：

使用了很少的数据
blender可能会过拟合（其实大概率是第一点导致的）
stacking使用多次的CV会比较稳健

mlxtend库

一.StackingClassifier

下面我们介绍一款功能强大的stacking利器，mlxtend库，它可以很快地完成对sklearn模型地stacking。

主要有以下几种使用方法吧：

I. 最基本的使用方法，即使用前面分类器产生的特征输出作为最后总的meta-classifier的输入数据

```
from sklearn import datasets

iris = datasets.load_iris()
```

```
X, y = iris.data[:, 1:3], iris.target

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier
import numpy as np

clf1 = KNeighborsClassifier(n_neighbors=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                          meta_classifier=lr)

print('3-fold cross validation:\n')

for clf, label in zip([clf1, clf2, clf3, sclf],
                      ['KNN',
                       'Random Forest',
                       'Naive Bayes',
                       'StackingClassifier']):

    scores = model_selection.cross_val_score(clf, X, y,
                                             cv=3, scoring='accu
    print("Accuracy: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

II. 另一种使用第一层基本分类器产生的类别概率值作为meta-classfier的输入，这种情况下需要将StackingClassifier的参数设置为 use_probas=True。如果将参数设置为 average_probas=True，那么这些基分类器对每一个类别产生的概率值会被平均，否则会拼接。

例如有两个基分类器产生的概率输出为：

```
classifier 1: [0.2, 0.5, 0.3]
classifier 2: [0.3, 0.4, 0.4]
   1) average = True :
产生的meta-feature 为: [0.25, 0.45, 0.35]
   2) average = False:
产生的meta-feature为: [0.2, 0.5, 0.3, 0.3, 0.4, 0.4]
```

```
from sklearn import datasets

iris = datasets.load_iris()
X, y = iris.data[:, 1:3], iris.target

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from mlxtend.classifier import StackingClassifier
import numpy as np

clf1 = KNeighborsClassifier(n_neighbors=1)
clf2 = RandomForestClassifier(random_state=1)
clf3 = GaussianNB()
lr = LogisticRegression()
sclf = StackingClassifier(classifiers=[clf1, clf2, clf3],
                          use_probas=True,
                          average_probas=False,
                          meta_classifier=lr)

print('3-fold cross validation:\n')

for clf, label in zip([clf1, clf2, clf3, sclf],
                      ['KNN',
                       'Random Forest',
                       'Naive Bayes',
                       'StackingClassifier']):

    scores = model_selection.cross_val_score(clf, X, y,
```

```
                                                cv=3, scoring='accu
    print("Accuracy: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
```

III. 另外一种方法是对训练基中的特征维度进行操作的，这次不是给每一个基分类器全部的特征，而是给不同的基分类器分不同的特征，即比如基分类器1训练前半部分特征，基分类器2训练后半部分特征（可以通过sklearn 的pipelines 实现）。最终通过StackingClassifier组合起来。

```
from sklearn.datasets import load_iris
from mlxtend.classifier import StackingClassifier
from mlxtend.feature_selection import ColumnSelector
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

iris = load_iris()
X = iris.data
y = iris.target

pipe1 = make_pipeline(ColumnSelector(cols=(0, 2)),
                      LogisticRegression())
pipe2 = make_pipeline(ColumnSelector(cols=(1, 2, 3)),
                      LogisticRegression())

sclf = StackingClassifier(classifiers=[pipe1, pipe2],
                          meta_classifier=LogisticRegression())

sclf.fit(X, y)
```

```
StackingClassifier 使用API及参数解析:

StackingClassifier(classifiers, meta_classifier, use_probas=False,
```

参数：

```
classifiers : 基分类器，数组形式，[cl1, cl2, cl3]. 每个基分类器的属性被存
meta_classifier : 目标分类器，即将前面分类器合起来的分类器
use_probas : bool (default: False) ，如果设置为True， 那么目标分类器的输
average_probas : bool (default: False)，用来设置上一个参数当使用概率值输
verbose : int, optional (default=0)。用来控制使用过程中的日志输出，当 ve
use_features_in_secondary : bool (default: False). 如果设置为True，那
```

属性：

```
clfs_ : 每个基分类器的属性，list, shape 为 [n_classifiers]。
meta_clf_ : 最终目标分类器的属性
```

方法：

```
fit(X, y)
fit_transform(X, y=None, fit_params)
get_params(deep=True)，如果是使用sklearn的GridSearch方法，那么返回分类器
predict(X)
predict_proba(X)
score(X, y, sample_weight=None)， 对于给定数据集和给定label，返回评价ac
set_params(params)，设置分类器的参数，params的设置方法和sklearn的格式一样
```