集成方法

课前准备

- 下载Anaconda软件，请点击这里进行下载。

本节要点

- 集成方法的概念，分类与原理。
- Bagging算法。
- 随机森林算法与特征重要性。

集成方法介绍

集成方法（集成学习）是一种解决问题的思想（不是具体的算法）。操作为将若干个基本评估器(分类器&回归器)进行组合，然后使用这些基本评估器来综合对未知样本进行预测。通过这种"集思广益"的行为，比起使用单个基本评估器进行预测，集成学习具有更好的泛化能力或稳健性。

偏差与方差

偏差

模型预测值与真实值之间的差异，称为偏差。偏差衡量的是模型的拟合效果，偏差越小，说明模型拟合的越好。

方差

模型在不同数据集上预测结果的差异，称为方差。方差衡量的是模型的泛化能力，方差越小，说明模型的泛化能力越好。

集成方法分类

集成方法可以分为以下两类：

- 平均方法。
- 增强方法。

平均方法

平均方法训练多个独立的基本评估器（评估器之间没有关联），然后对多个评估器的预测结果进行平均化。对于分类任务，使用每个评估器预测结果中，类别最多（或加权最多）的作为预测结果。对于回归任务，使用多个评估器预测结果的均值作为预测结果。平均方法通过综合考量的行为，可以有效的减少方差，因此，其预测结果通常可以优于任何一个基本评估器。

增强方法

在增强方法中，多个基本评估器是按顺序训练的，然后将这些基本评估器（通常是弱评估器）进行组合，进而产生一个预测能力强的评估器。与平均方法不同，增强方法的多个基本评估器不是独立的，后续评估器需要依赖于之前的评估器，训练过程中，会试图减少组合之后评估器的偏差。

集成方法效果

我们以二分类为例，如果存在n个分类器，每个分类器的错误率都为e且各个分类器之间是独立的。因此，多个分类器集成之后的错误率服从二项分布，其中，k个分类器出错的概率密度可表示为：

$$P(y=k)=C_n^k e^k(1-e)^{n-k}$$

假设现有11个分类器，单个分类器的错误率为0.25，则如果集成分类器出错，则至少需要6个（或6个以上）的分类器出错，集成后分类器出错的概率密度为：

$$P(y \geqslant k)=\sum_{k=6}^{n} C_{11}^k 0.25^k * 0.75^{11-k}=0.034$$

可见，集成后分类器的出错率要远小于单个分类器的出错率。

集成评估器效果是否一定会好于基本评估器呢?
A 一定好于。
B 不一定好于，考虑所有可能，集成评估器的综合效果更好。
C 不一定好于，考虑所有可能，基本评估器的综合效果更好。
D 不一定好于，考虑所有可能，集成评估器与基本评估器的综合效果相同。

效果示例

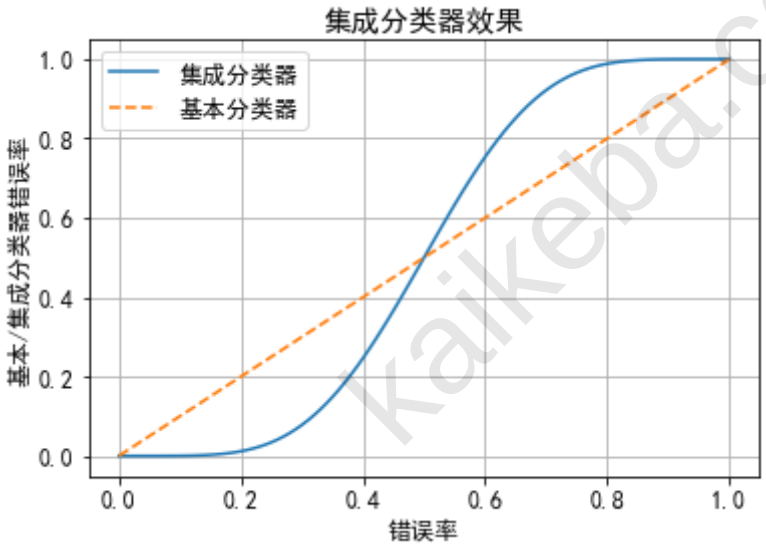
接下来，我们通过程序来演示集成的效果。

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
# 计算组合值。
from scipy.special import comb

mpl.rcParams["font.family"] = "SimHei"
mpl.rcParams["axes.unicode_minus"] = False
mpl.rcParams["font.size"] = 12


def ensemble_error(n, error):
    """用来计算集成分类器发生错误的概率密度值。

    如果要使集成评估器预测错误，则需要半数以上的基本评估器发生错误。
    当n为偶数时，达到半数错误，即认为集成评估器预测错误。

    Parameters
    ----------
    n : int
        基本评估器的数量。
```

```
    error : array-like, shape=(num, 1)
        每一个基本评估器发生错误的概率。

    """

    # 确保error为二维的ndarray数组类型，以用于后面的广播运算。
    error = np.asarray(error)
    # 计算半数错误的评估器个数。
    start = np.ceil(n / 2.0)
    # 计算当集成评估器发生错误，基本评估器错误个数的可能区间。
    k = np.arange(start, n + 1)
    # error的形状为(n1, 1)，k的形状为(n2,)，经过广播计算后，
    # v的形状为(n1, n2)。每一行为一个不同错误率，每一列为在不同的k值
    #（预测错误的评估器个数）下得到的错误概率。
    v = comb(n, k) * error ** k * (1-error) ** (n - k)
    # 将每一行的概率相加，就是最终在每个错误率下，集成评估器的错误率。
    return np.sum(v, axis=1)


ensemble_error(11, [[0.25], [0.3]])
```

```
array([0.03432751, 0.07822479])
```

```
# 定义基本评估器发生错误的概率区间。
error = np.arange(0.0, 1.01, 0.01)
# 计算在不同error取值的情况下，集成评估器发生错误的概率。
ens_errors = ensemble_error(n=11, error=error[:, np.newaxis])
plt.plot(error, ens_errors, label="集成分类器")
plt.plot(error, error, linestyle="--", label="基本分类器")
plt.xlabel("错误率")
plt.ylabel("基本/集成分类器错误率")
plt.legend(loc="best")
plt.title("集成分类器效果")
plt.grid()
```

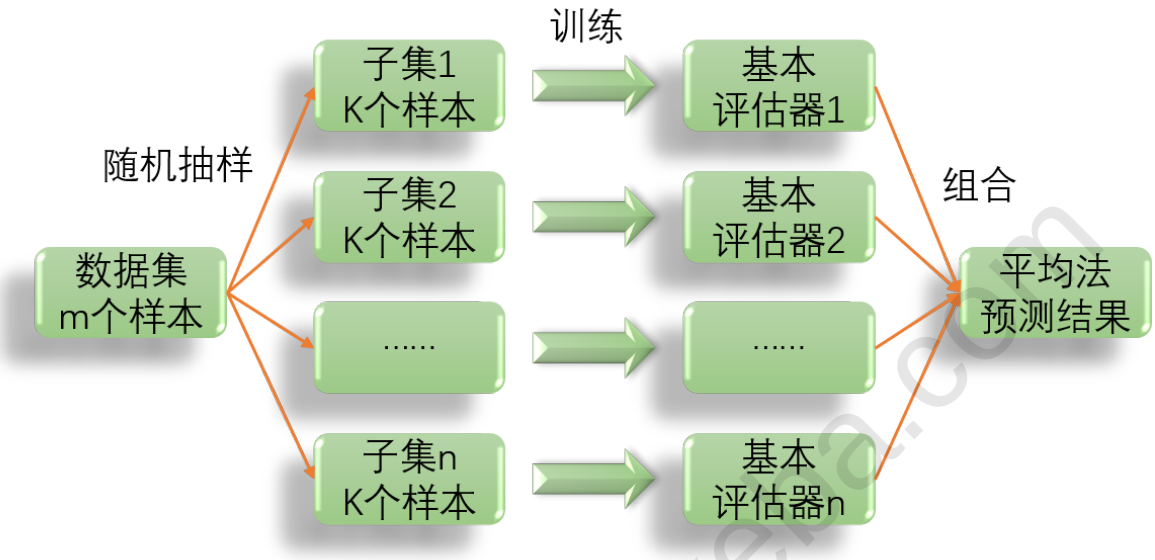


Bagging

步骤

Bagging算法采用平均方法的思想，也称为汇聚法(Bootstrap Aggregating)。算法过程如下：

1. 在原始数据集上进行随机抽样（抽样可以是放回抽样与不放回抽样）。
2. 使用得到的随机子集来训练评估器（基本评估器）。
3. 重复步骤1与步骤2若干次，会得到n个基本评估器。
4. 将n个评估器进行组合，根据多数投票（分类）或者求均值（回归）的方式来统计最终的结果（平均方法）。



优势

bagging方法通过随机抽样来构建原始数据集的子集，来训练不同的基本评估器，然后再将多个基本评估器进行组合来预测结果，这样可以有效减小基本评估器的方差。因此，通过bagging方法，就可以非常便捷的对基本评估器进行改进，而无需去修改基本评估器底层的实现。
因为bagging方法可以有效的降低过拟合，因此，bagging方法适用于强大而复杂的模型（例如，完全生长的决策树）。

bagging程序示例

分类示例

```
from sklearn.datasets import make_classification
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split

# flip_y：类别随机分配的比例。值越大，则噪声越大，分类难度越大。
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, n_classes=3, random_state=0,
        flip_y=0.2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
# 定义一颗不限生长的决策树（强学习器）。
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)
print("决策树准确率：")
print(tree.score(X_train, y_train))
print(tree.score(X_test, y_test))
print("bagging准确率：")
# base_estimator：指定基本评估器。即bagging算法所组合的评估器。
# n_estimators：基本评估器的数量。有多少个评估器，就会进行多少次随机采样，产生多少个原始数据集的子集。
# max_samples：每次随机采样的样本数量。该参数可以是int类型或float类型。如果是int类型，则指定采样的样本数量。
#               如果是float类型，则指定采样占原始数据集的比例。
# max_features：每次随机采样的特征数量。可以是int类型或float类型。
# bootstrap：指定是否进行放回抽样。默认为True。
# bootstrap_features：指定对特征是否进行重复抽取。默认为False。
bag = BaggingClassifier(base_estimator=tree, n_estimators=100, max_samples=0.8, max_features=0.8)
bag.fit(X_train, y_train)
print(bag.score(X_train, y_train))
print(bag.score(X_test, y_test))
```

```
决策树准确率：
1.0
0.5
bagging准确率：
1.0
0.6633333333333333
```

我们运行上面的程序多次，决策树的运行结果可能是不同的，这是什么原因呢？【作业】

回归示例

```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import BaggingRegressor
from sklearn.model_selection import train_test_split

X, y = load_diabetes(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)

lr = LinearRegression()
lr.fit(X_train, y_train)
print("线性回归R^2值：")
print(lr.score(X_train, y_train))
print(lr.score(X_test, y_test))
bag = BaggingRegressor(lr, n_estimators=100, max_samples=0.9,
max_features=0.9)
bag.fit(X, y)
print("bagging R^2值：")
print(bag.score(X_train, y_train))
print(bag.score(X_test, y_test))
```

```
线性回归R^2值：
0.5539411781927148
0.39289398450747565
bagging R^2值：
0.5439035790971334
0.4277552490477452
```

课堂练习

关于bagging，以下说法正确的是（ ）。【不定项】
A bagging能够减少基本评估器的偏差。
B bagging能够减少基本评估器的方差。
C bagging采用的是平均方法的思想。
D bagging组合之后的效果一定会优于基本评估器。

随机森林(Random Forest)

随机森林(Random Forest)是一种元评估器，其使用原始数据集的子集来训练多棵决策树，并使用平均方法来计算预测结果。其实现为：

1. 从原始数据集中选出m个样本用于训练。
2. 使用这m个样本来构建一棵决策树。
    - 从所有特征中随机选择K个特征（特征不重复）。
    - 根据目标函数的要求（如最大信息增益），使用选定的特征对节点进行划分。
3. 重复以上两步n次，即建立n棵决策树。
4. 这n棵决策树形成随机森林，通过投票表决结果（概率）或均值决定最终的预测值。

关于随机森林，具有如下的说明：

- 随机森林的基本评估器，固定为决策树。
- 因为随机森林就是组合多棵决策树，因此，决策树的很多参数，也适用于随机森林。
- 由于这种随机性，随机森林的偏差通常会略微增加（相对于单棵决策树），但由于使用多棵决策树平均预测，其方差也会减小，从而从整体上来讲，模型更加优秀。
- 在分类预测时，scikit-learn中使用概率进行投票，而不是加1进行投票。
- 对于回归任务，通常设置max_features=n_features，对于分类任务，通常设置max_features=sqrt(n_features)。这也正是scikit-learn随机森林的默认值。
- max_depth=None结合min_samples_split=2，通常可以获得很好的结果，但是，这往往会消耗大量的内存。

分类程序示例

```
# 葡萄酒数据集
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

X, y = load_wine(return_X_y=True)
# 为了可视化方便，简化操作，我们只取两个特征。
X = X[:, [0, 10]]
# 我们过滤掉0的类别，只取两个类别。
X = X[y != 0]
y = y[y != 0]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=0)
```

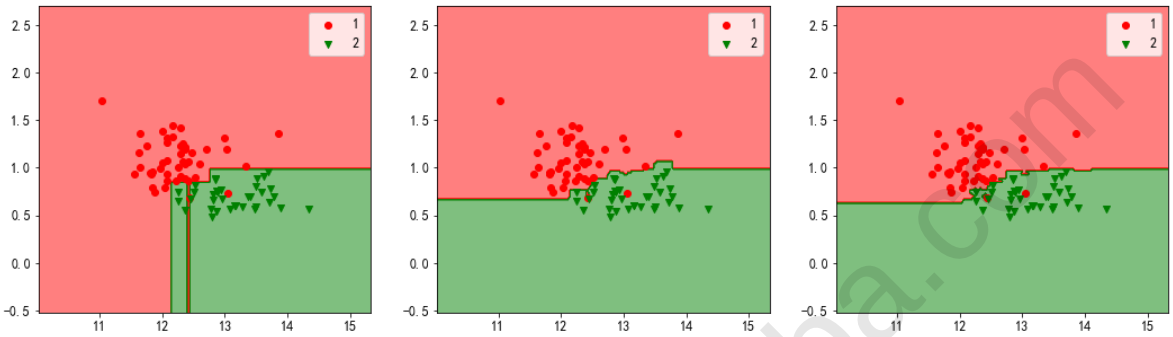
```
14
15  tree = DecisionTreeClassifier(criterion="gini")
16  tree = tree.fit(X_train, y_train)
17  print("决策树分类准确率：")
18  print(tree.score(X_train, y_train))
19  print(tree.score(X_test, y_test))
20  # n_jobs 开辟进程的数量。如果指定-1，则表示利用现有的所有CPU来实现并行化。
21  bag = BaggingClassifier(base_estimator=tree, n_estimators=100,
    max_samples=0.9, max_features=1.0,
22          bootstrap=True, bootstrap_features=False, n_jobs=-1,
    random_state=0)
23  bag = bag.fit(X_train, y_train)
24  print("bagging准确率：")
25  print(bag.score(X_train, y_train))
26  print(bag.score(X_test, y_test))
27  # n_estimators：基本评估器（决策树）的数量。
28  # max_samples：每次抽样用于训练基本评估器的样本数量。
29  # bootstrap：如果为True（默认值），抽样允许重复（放回抽样）。如果为False，则使用所有的
    样本训练每棵决策树。
30  #             注意该参数的含义与bagging的参数有所不同。
31  # max_samples：当bootstrap为True时，训练每棵决策树所使用的样本数。默认为None（使用所
    有的样本）。
32  rf = RandomForestClassifier(n_estimators=100, criterion="gini",
    random_state=0)
33  rf.fit(X_train, y_train)
34  print("随机森林准确率：")
35  print(rf.score(X_train, y_train))
36  print(rf.score(X_test, y_test))
```

```
1  决策树分类准确率：
2  1.0
3  0.8611111111111112
4  bagging准确率：
5  1.0
6  0.8888888888888888
7  随机森林准确率：
8  1.0
9  0.8888888888888888
```

```
1  from matplotlib.colors import ListedColormap
2
3  def plot_decision_boundary(model, X, y):
4      """绘制model模型的决策边界。
5
6      绘制决策边界，同时绘制样本数据X与对应的类别y，
7      用于可视化模型的分类效果。
8
9      Parameters
10     ----------
11     model : object
12         模型对象。
13     X : array-like
14         需要绘制的样本数据。
15     y : array-like
```

```
        每个样本数据对应的类别（标签）。

    """
    # 定义不同类别的颜色与符号。可以用于二分类与三分类。
    color = ["r", "g", "b"]
    marker = ["o", "v", "x"]
    # 获取数据中不重复的标签。
    class_label = np.unique(y)
    # 定义颜色图，在绘制等高线的时候使用，不同的值使用不同的颜色来绘制。
    cmap = ListedColormap(color[: len(class_label)])
    # 获取每个特征的最小值与最大值。
    x1_min, x2_min = np.min(X, axis=0)
    x1_max, x2_max = np.max(X, axis=0)
    # 定义每个特征的取值范围。
    x1 = np.arange(x1_min - 1, x1_max + 1, 0.02)
    x2 = np.arange(x2_min - 1, x2_max + 1, 0.02)
    # 对数组x1,x2进行扩展，获取二者的笛卡尔积组合，用于送入模型中，进行预测。
    X1, X2 = np.meshgrid(x1, x2)
    # 将之前两个特征的笛卡尔积组合送入模型中，预测结果。
    Z = model.predict(np.array([X1.ravel(),
X2.ravel()]).T).reshape(X1.shape)
    # 根据Z值的不同，绘制等高线（不同的值使用不同的颜色表示）。
    plt.contourf(X1, X2, Z, cmap=cmap, alpha=0.5)
    # 绘制样本数据X。
    for i, class_ in enumerate(class_label):
        plt.scatter(x=X[y == class_, 0], y=X[y == class_, 1],
                    c=cmap.colors[i], label=class_, marker=marker[i])
    plt.legend()
```

```
plt.figure(figsize=(18, 5))
name = ["决策树", "bagging", "随机森林"]
for index, estimator in enumerate([tree, bag, rf], start=1):
    plt.subplot(1, 3, index)
    plot_decision_boundary(estimator, X_train, y_train)
```



特征重要度

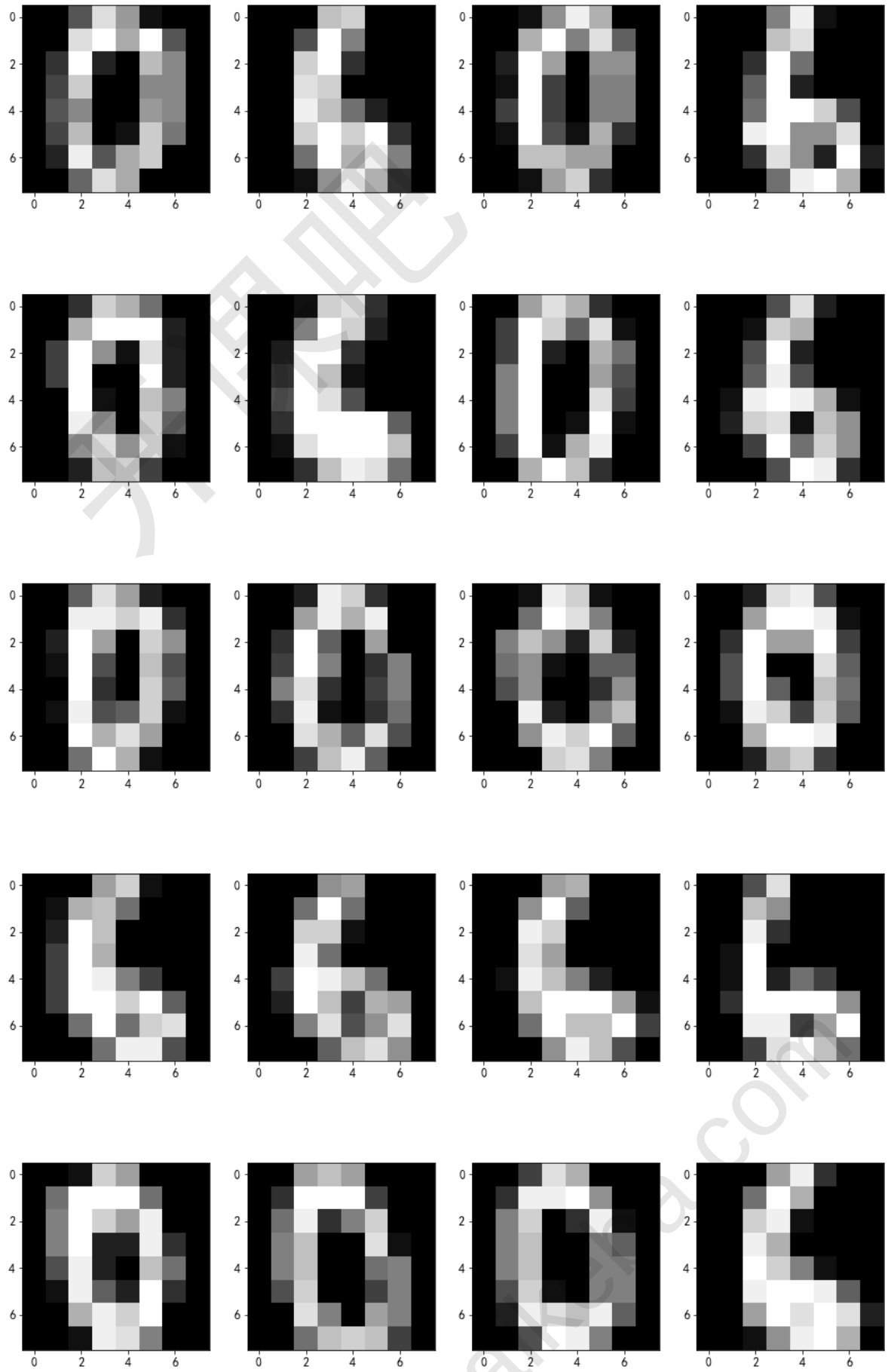
在决策树（或随机森林）中，每个特征对目标变量（y）的预测性帮助可能都是不同的，如果一个特征对目标变量的可预测性帮助越大，则该特征的重要度就越大，否则，特征的重要度就越小。

我们可以通过决策树（或随机森林）对象的feature_importances_属性来返回每个特征的重要度，该值是进行归一化之后的结果。

```
from sklearn.datasets import load_digits
from sklearn.ensemble import RandomForestClassifier
```

```
3
4  digits = load_digits()
5  X, y = digits.data, digits.target
6  mask = (y == 0) | (y == 6)
7  X = X[mask]
8  y = y[mask]
9  print(X.shape, y.shape)
10 row, col = 5, 4
11 fig, ax = plt.subplots(row, col)
12 ax = ax.ravel()
13 fig.set_size_inches(15, row * 5)
14 for i in range(row * col):
15     ax[i].imshow(X[i].reshape(8, 8), cmap="gray")
```

```
1 (359, 64) (359,)
```



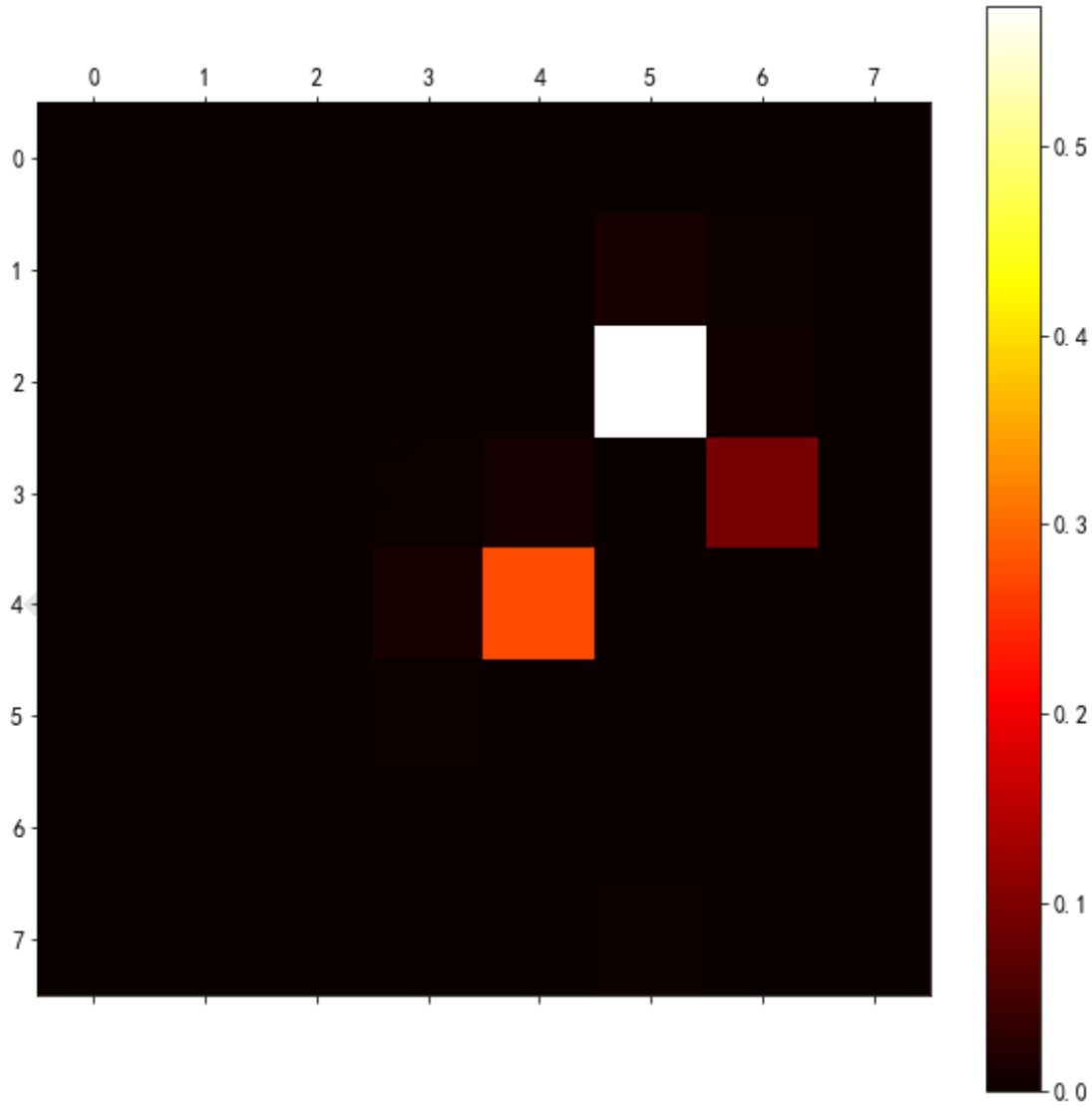
```
rf = RandomForestClassifier(n_estimators=100, max_features=0.8,
random_state=0)
rf.fit(X, y)
print(rf.score(X, y))
```

```
1.0
```

```
# 返回特征的重要度。特征的重要度根据该特征对目标变量（y）的可预测性来度量。
# 特征对目标变量的可预测性越有帮助，则重要度越大，否则越小。
importances = rf.feature_importances_
print(importances)
# 特征重要度的权重之和为1。
print(np.sum(importances))
importances = importances.reshape(8, 8)
plt.figure(figsize=(10, 10))
# 绘制特征的重要度。
plt.matshow(importances, cmap=plt.cm.hot, fignum=0)
plt.colorbar()
```

```
[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 1.12981904e-04 1.11002333e-04 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 3.31831458e-04 0.00000000e+00
 7.59283078e-04 1.28214747e-02 2.50816352e-03 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 5.74172641e-01 4.61586144e-03 0.00000000e+00
 0.00000000e+00 5.57141970e-05 0.00000000e+00 2.69864695e-03
 9.25326713e-03 1.87840875e-03 9.48801255e-02 0.00000000e+00
 0.00000000e+00 0.00000000e+00 1.14899529e-04 1.26538510e-02
 2.75078592e-01 1.38217325e-03 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 1.01708937e-04 2.69819236e-03
 4.38275419e-04 0.00000000e+00 8.92535273e-05 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00
 0.00000000e+00 0.00000000e+00 0.00000000e+00 1.10805889e-04
 0.00000000e+00 0.00000000e+00 1.12033345e-04 2.17925127e-04
 2.20251758e-04 2.58263493e-03 0.00000000e+00 0.00000000e+00]
1.0
```

```
<matplotlib.colorbar.Colorbar at 0x252c08daac8>
```



AdaBoost

AdaBoost也是一种集成方法算法，该算法使用增强方法的思想，通过调整样本的权重来实现分类或回归任务。关于该算法的细节，老梁提供辅助视频，供大家参考。

拓展点

- AdaBoost算法。

作业

1. 使用RFECV对随机森林进行进行特征选择，查看结果。
2. 当决策树的splitter参数设置为best，max_features设置为None（选择所有特征），多次运行程序，拟合效果也可能会不一致，这是为什么呢？参考Bagging分类的程序，给出解释。