一、利用 sklearn 库和 iris 数据集实现分类

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection  import train_test_split
import matplotlib.pyplot as plt

# data
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', \
                  'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, [0, 1, -1]])
    for i in range(len(data)):
        if data[i,-1] == 0:
            data[i,-1] = -1
    # print(data)
    return data[:,:2], data[:,-1]

X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

plt.scatter(X[:50,0],X[:50,1], label='0')
plt.scatter(X[50:,0],X[50:,1], label='1')
plt.legend()

from sklearn.ensemble import AdaBoostClassifier
clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5)
clf.fit(X_train, y_train)

print (clf.score(X_test, y_test))
```

二、自己实现 adaboost 算法

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection  import train_test_split
import matplotlib.pyplot as plt

# data
def create_data():
    iris = load_iris()
    df = pd.DataFrame(iris.data, columns=iris.feature_names)
    df['label'] = iris.target
    df.columns = ['sepal length', 'sepal width', \
                  'petal length', 'petal width', 'label']
    data = np.array(df.iloc[:100, [0, 1, -1]])
    for i in range(len(data)):
        if data[i,-1] == 0:
            data[i,-1] = -1
    # print(data)
    return data[:,:2], data[:,-1]
```

```
X, y = create_data()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

plt.scatter(X[:50,0],X[:50,1], label='0')
plt.scatter(X[50:,0],X[50:,1], label='1')
plt.legend()

class AdaBoost:
    def __init__(self, n_estimators=50, learning_rate=1.0):
        self.clf_num = n_estimators
        self.learning_rate = learning_rate

    def init_args(self, datasets, labels):
        self.X = datasets
        self.Y = labels
        self.M, self.N = datasets.shape

        # 弱分类器数目和集合
        self.clf_sets = []

        # 初始化weights
        self.weights = [1.0/self.M]*self.M

        # G(x)系数 alpha
        self.alpha = []
```

```
    def _G(self, features, labels, weights):
        m = len(features)
        error = 100000.0 # 无穷大
        best_v = 0.0
        # 单维features
        features_min = min(features)
        features_max = max(features)
        n_step = (features_max - features_min + self.learning_rate) // self.learning_rate
        # print('n_step:{}'.format(n_step))
        direct, compare_array = None, None
        for i in range(1, int(n_step)):
            v = features_min + self.learning_rate * i

            if v not in features:
                # 误分类计算
                compare_array_positive = np.array([1 if features[k] > v else -1 for k in range(m)])
                weight_error_positive = sum([weights[k] for k in range(m) \
                if compare_array_positive[k] != labels[k]])

                compare_array_nagetive = np.array([-1 if features[k] > v else 1 for k in range(m)])
                weight_error_nagetive = sum([weights[k] for k in range(m) \
                if compare_array_nagetive[k] != labels[k]])

                if weight_error_positive < weight_error_nagetive:
                    weight_error = weight_error_positive
                    _compare_array = compare_array_positive
                    direct = 'positive'
                else:
                    weight_error = weight_error_nagetive
                    _compare_array = compare_array_nagetive
                    direct = 'nagetive'
```

```
                # print('v:{} error:{}'.format(v, weight_error))
                if weight_error < error:
                    error = weight_error
                    compare_array = _compare_array
                    best_v = v
        return best_v, direct, error, compare_array

    # 计算alpha
    def _alpha(self, error):
        return 0.5 * np.log((1-error)/error)

    # 规范化因子
    def _Z(self, weights, a, clf):
        return sum([weights[i]*np.exp(-1*a*self.Y[i]*clf[i]) for i in range(self.M)])

    # 权值更新
    def _w(self, a, clf, Z):
        for i in range(self.M):
            self.weights[i] = self.weights[i]*np.exp(-1*a*self.Y[i]*clf[i])/ Z

    # G(x)的线性组合
    def _f(self, alpha, clf_sets):
        pass

    def G(self, x, v, direct):
        if direct == 'positive':
            return 1 if x > v else -1
        else:
            return -1 if x > v else 1
```

```
    def fit(self, X, y):
        self.init_args(X, y)
        for epoch in range(self.clf_num):
            best_clf_error, best_v, clf_result = 100000, None, None
            # 根据特征维度, 选择误差最小的
            for j in range(self.N):
                features = self.X[:, j]
                # 分类阈值, 分类误差, 分类结果
                v, direct, error, compare_array = self._G(features, self.Y, self.weights)

                if error < best_clf_error:
                    best_clf_error = error
                    best_v = v
                    final_direct = direct
                    clf_result = compare_array
                    axis = j

                # print('epoch:{}/{} feature:{} error:{} v:{}'.\
                #format(epoch, self.clf_num, j, error, best_v))
                if best_clf_error == 0:
                    break

            # 计算G(x)系数a
            a = self._alpha(best_clf_error)
            self.alpha.append(a)
            # 记录分类器
            self.clf_sets.append((axis, best_v, final_direct))
            # 规范化因子
            Z = self._Z(self.weights, a, clf_result)
            # 权值更新
            self._w(a, clf_result, Z)
```

```
#             print('classifier:{}/{} error:{:.3f} v:{} direct:{} a:{:.5f}'. \
              #format(epoch+1, self.clf_num, error, best_v, final_direct, a))
#             print('weight:{}'.format(self.weights))
#             print('\n')

    def predict(self, feature):
        result = 0.0
        for i in range(len(self.clf_sets)):
            axis, clf_v, direct = self.clf_sets[i]
            f_input = feature[axis]
            result += self.alpha[i] * self.G(f_input, clf_v, direct)
        # sign
        return 1 if result > 0 else -1

    def score(self, X_test, y_test):
        right_count = 0
        for i in range(len(X_test)):
            feature = X_test[i]
            if self.predict(feature) == y_test[i]:
                right_count += 1

        return right_count / len(X_test)

# 100次结果
result = []
for i in range(1, 101):
    X, y = create_data()
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
    clf = AdaBoost(n_estimators=100, learning_rate=0.2)
    clf.fit(X_train, y_train)
    r = clf.score(X_test, y_test)
    print('{}/100 score: {}'.format(i, r))
    result.append(r)

print('average score:{:.3f}%'.format(sum(result)))
```

三、实现提升树算法

```
from collections import defaultdict
import numpy as np


class BoostingTree:
    def __init__(self, epsilon=1e-2):
        self.epsilon = epsilon
        self.cand_splits = []  # 候选切分点
        self.split_index = defaultdict(tuple)  # 由于要多次切分数据集，故预先存储，切分后数据点的索引
        self.split_list  = []  # 最终各个基本回归树的切分点
        self.c1_list     = []  # 切分点左区域取值
        self.c2_list     = []  # 切分点右区域取值
        self.N           = None
        self.n_split     = None

    def init_param(self, X_data):
        # 初始化参数
        self.N = X_data.shape[0]
        for i in range(1, self.N):
            self.cand_splits.append((X_data[i][0] + X_data[i - 1][0]) / 2)
        self.n_split = len(self.cand_splits)
        for split in self.cand_splits:
            left_index = np.where(X_data[:, 0]<= split)[0]
            right_index = list(set(range(self.N))-set(left_index))
            self.split_index[split] = (left_index, right_index)
        return
```

```
    def _cal_err(self, split, y_res):
        # 计算每个切分点的误差
        inds  = self.split_index[split]
        left  = y_res[inds[0]]
        right = y_res[inds[1]]

        c1 = np.sum(left) / len(left)
        c2 = np.sum(right) / len(right)
        y_res_left  = left - c1
        y_res_right = right - c2
        res = np.hstack([y_res_left, y_res_right])
        res_square = np.apply_along_axis(lambda x: x ** 2, 0, res).sum()
        return res_square, c1, c2

    def best_split(self,y_res):
        # 获取最佳切分点，并返回对应的残差
        best_split = self.cand_splits[0]
        min_res_square, best_c1, best_c2 = self._cal_err(best_split, y_res)

        for i in range(1, self.n_split):
            res_square, c1, c2 = self._cal_err(self.cand_splits[i], y_res)
            if res_square < min_res_square:
                best_split = self.cand_splits[i]
                min_res_square = res_square
                best_c1 = c1
                best_c2 = c2

        self.split_list.append(best_split)
        self.c1_list.append(best_c1)
        self.c2_list.append(best_c2)
        return
```

```
    def _fx(self, X):
        # 基于当前组合树，预测X的输出值
        s = 0
        for split, c1, c2 in zip(self.split_list, self.c1_list, self.c2_list):
            if X < split:
                s += c1
            else:
                s += c2
        return s

    def update_y(self, X_data, y_data):
        # 每添加一颗回归树，就要更新y，即基于当前组合回归树的预测残差
        y_res = []
        for X, y in zip(X_data, y_data):
            y_res.append(y - self._fx(X[0]))
        y_res = np.array(y_res)
        res_square = np.apply_along_axis(lambda x: x ** 2, 0, y_res).sum()
        return y_res, res_square

    def fit(self, X_data, y_data):
        self.init_param(X_data)
        y_res = y_data
        while True:
            self.best_split(y_res)
            y_res, res_square = self.update_y(X_data, y_data)
            if res_square < self.epsilon:
                break
        return

    def predict(self, X):
        return self._fx(X)
```

```
if __name__ == '__main__':
    data = np.array(
        [[1, 5.56], [2, 5.70], [3, 5.91], [4, 6.40], [5, 6.80],\
         [6, 7.05], [7, 8.90], [8, 8.70], [9, 9.00], [10, 9.05]])
    X_data = data[:, :-1]
    y_data = data[:, -1]
    BT = BoostingTree(epsilon=0.18)
    BT.fit(X_data, y_data)
    print(BT.split_list)
    print(BT.c1_list)
    print(BT.c2_list)
    '''
    X_data_raw = np.linspace(-5, 5, 100)
    X_data = np.transpose([X_data_raw])
    y_data = np.sin(X_data_raw)
    BT = BoostingTree(epsilon=0.1)
    BT.fit(X_data, y_data)
    y_pred = [BT.predict(X) for X in X_data]

    import matplotlib.pyplot as plt

    p1 = plt.scatter(X_data_raw, y_data, color='r')
    p2 = plt.scatter(X_data_raw, y_pred, color='b')
    plt.legend([p1, p2], ['real', 'pred'])
    plt.show()
    '''
```