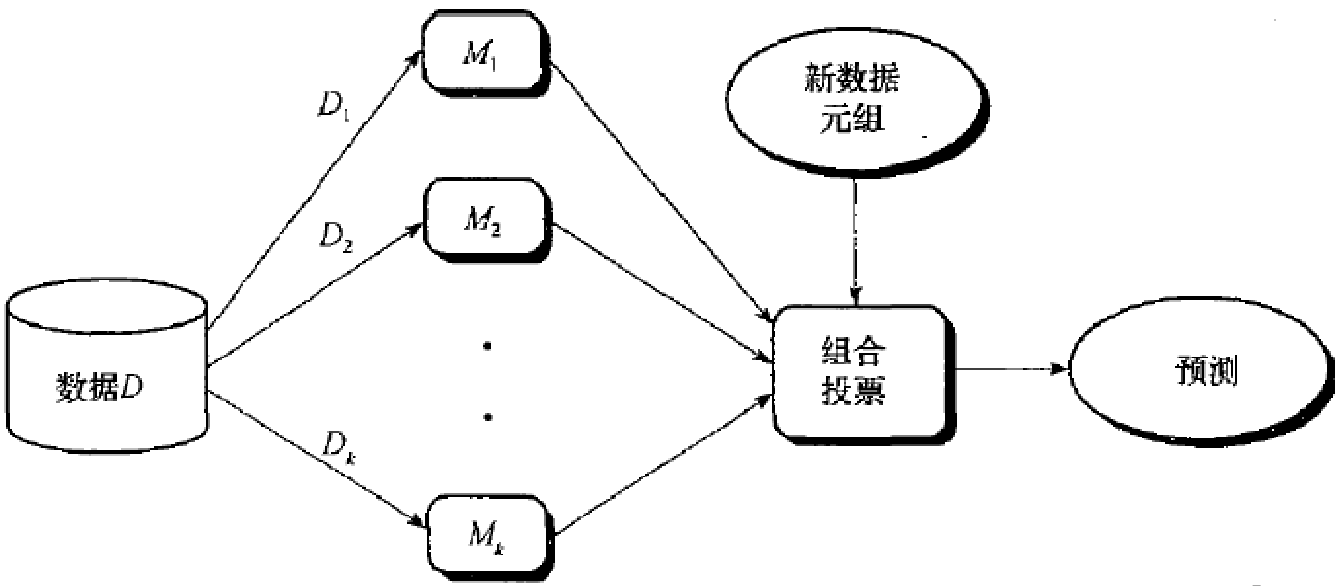
组合算法与提升

主要内容

- 组合法的基本思路
- 装袋法(Bagging)
- 提升与Adaboost算法
  - 算法描述
  - 前向分步算法+指数损失函数
- 随机森林
- 三种方法总结

提升分类器准确率的组合方法

- 组合方法包括：装袋（bagging），提升（boosting）和随机森林
- 基于学习数据集抽样产生若干训练集
- 使用训练集产生若干分类器
- 每个分类器分别进行预测，通过简单选举多数，判定最终所属分类



为什么组合方法能提高分类准确率？

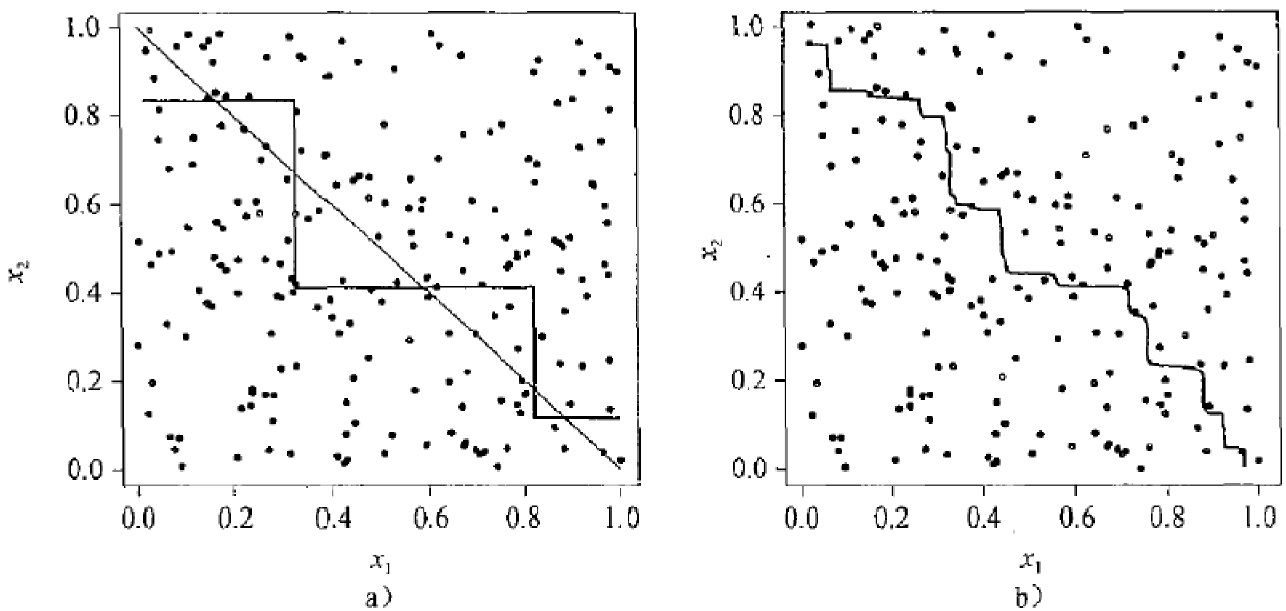


图 8.22　一个线性可分问题（即实际的决策边界是一条直线）的决策边界：a）单棵决策树；b）决策树的组合分类器。决策树努力近似线性边界。组合分类器更接近于真实的边界。取自 Seni 和 Elder[SE10]

组合算法的优势

- 能明显提升判别准确率:由弱分类器得到强分类器
- 对误差和噪音更加鲁棒性
- 一定程度抵消过度拟合
- 适合并行化计算

装袋法(Bagging)策略

- bootstrap aggregation
- 从样本集中重采样(有重复的)选出n个样本
- 在所有属性上，对这n个样本建立分类器(ID3、C4.5、CART、SVM、Logistic回归等)
- 重复以上两步m次，即获得了m个分类器
- 将数据放在这m个分类器上，最后根据这m个分类器的投票结果，决定数据属于哪一类

装袋法(Bagging)算法

算法：装袋。装袋算法——为学习方案创建组合分类模型，其中每个模型给出等权重预测。

输入：

- D：d个训练元组的集合；
- k：组合分类器中的模型数；
- 一种学习方案（例如，决策树算法、后向传播等）

输出：组合分类器——复合模型M^*。

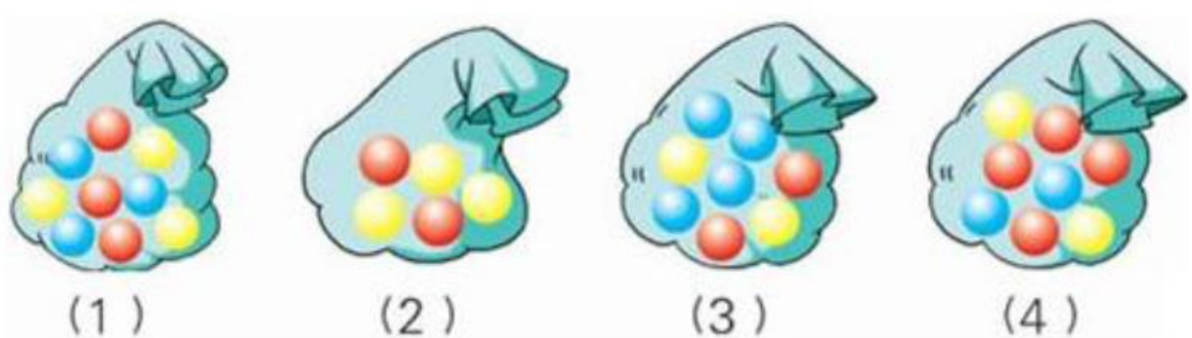
方法：

(1) **for** i = 1 to k **do** // 创建k个模型
(2) 通过对D有放回抽样，创建自助样本D_i；
(3) 使用D_i和学习方法导出模型M_i；
(4) **endfor**

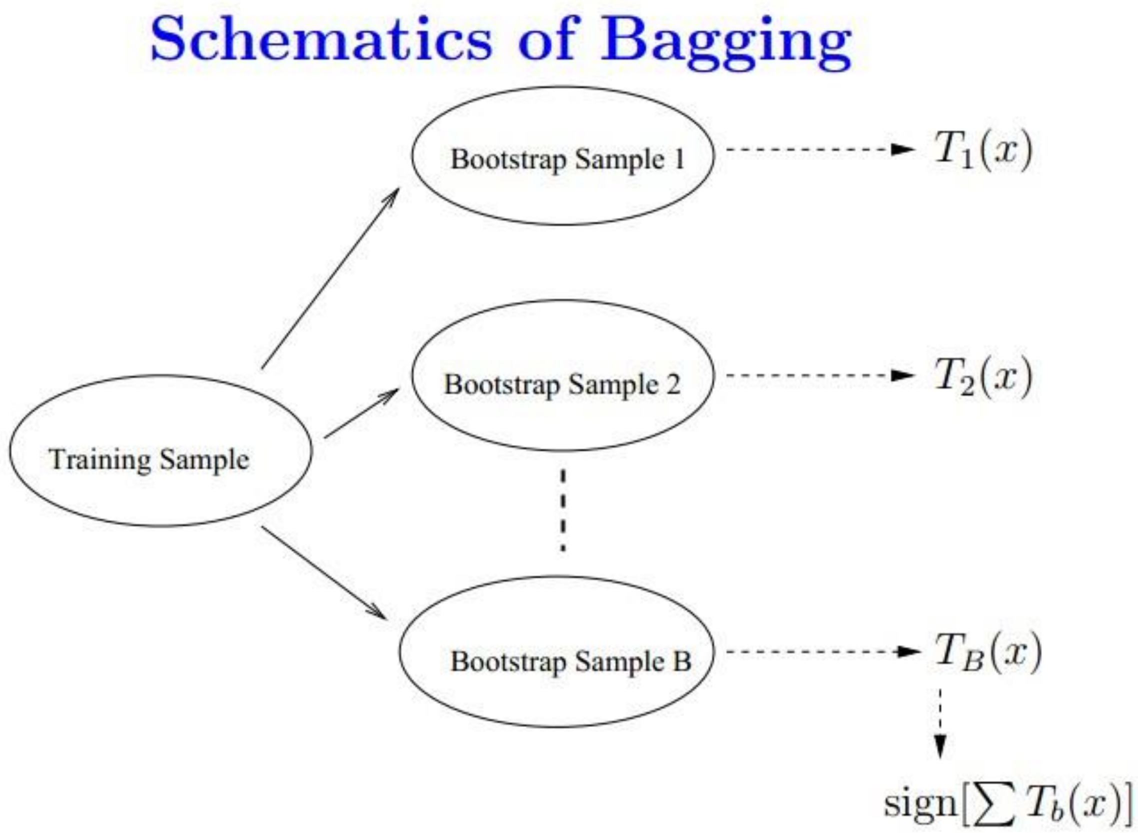
使用组合分类器对元组X分类：
让k个模型都对X分类并返回多数表决；

解释：有放回抽样自助样本

- 有放回抽样
- 自助样本（bootstrap）



Bagging

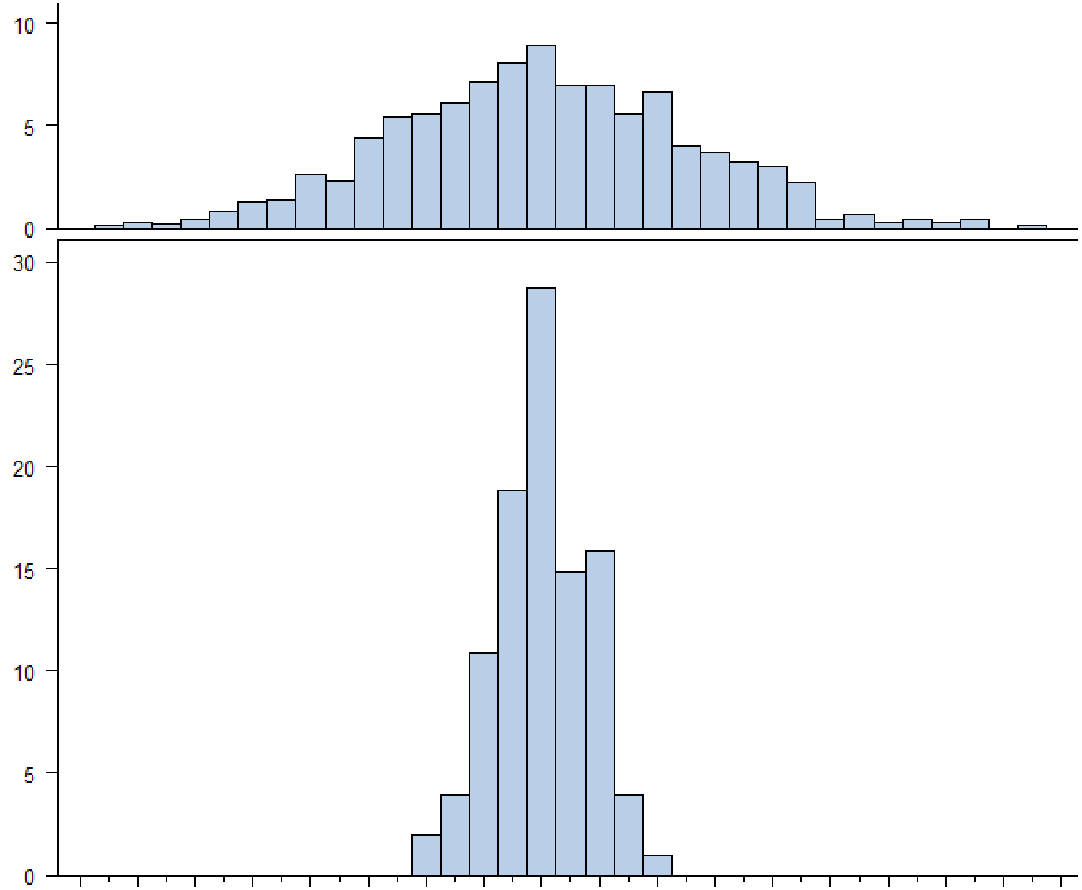

Schematics of Bagging
Training Sample
Bootstrap Sample 1
Bootstrap Sample 2
Bootstrap Sample B
$T_1(x)$
$T_2(x)$
$T_B(x)$
$\text{sign}[\sum T_b(x)]$

袋装算法的优势

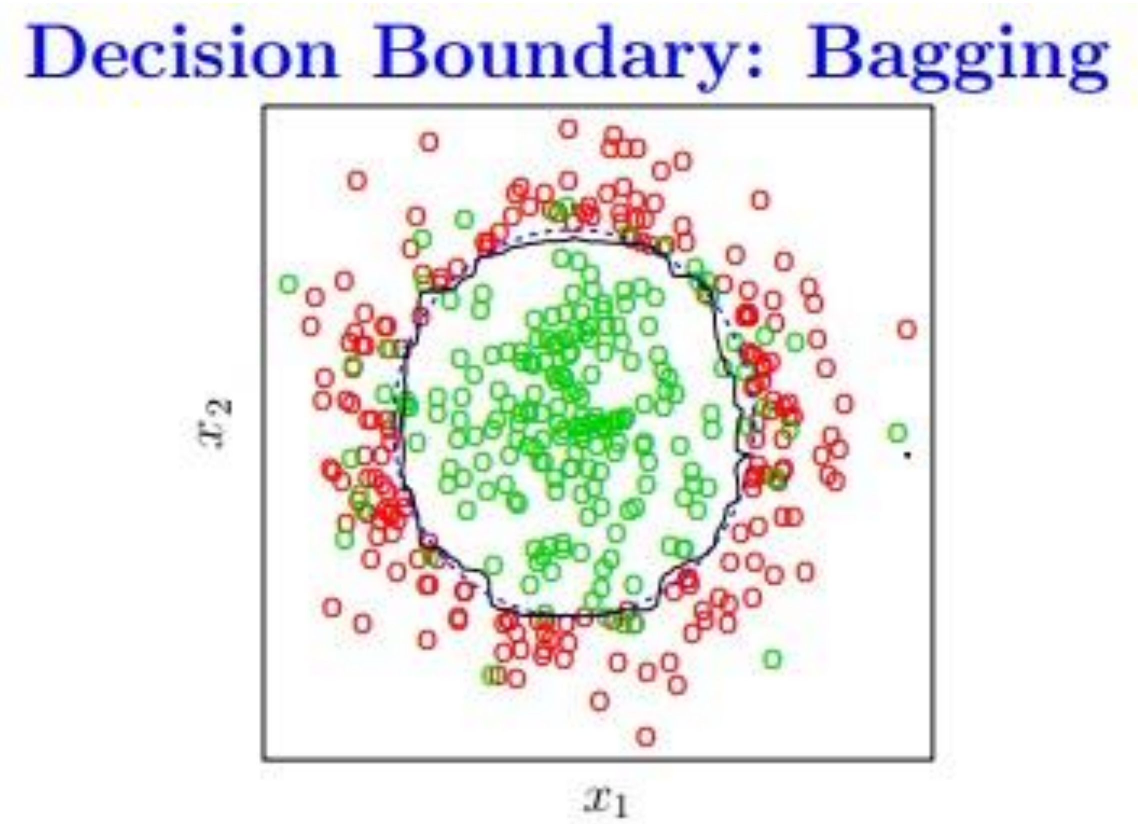
- 准确率明显高于组合中任何单个的分类器
- 对于较大的噪音，表现不至于很差，并且具有鲁棒性
- 不容易过度拟合

随机变量分布

随机变量**均值**分布



Bagging的结果



提升（boosting）算法思想

- 训练集中的元组被分配权重
- 权重影响抽样，权重越大，越可能被抽取
- 迭代训练若干个分类器，在前一个分类器中被错误分类的元组，会被提高权重，使到它在后面建立的分类器里被更加“关注”
- 最后分类也是由所有分类器一起投票，投票权重取决于分类器的准确率

Adaboost算法

算法： Adaboost.一种提升算法——创建分类器的组合。每个给出一个加权投票。

输入：

- D：类标记的训练元组集。
- k：轮数（每轮产生一个分类器）。
- 一种分类学习方案。

输出： 一个复合模型。

方法：

```
(1)  将D中每个元组的权重初始化为1/d;
(2)  for i = 1 to k do                              // 对于每一轮
(3)     根据元组的权重从D中有放回抽样,得到D_i;
(4)     使用训练集D_i导出模型M_i;
(5)     计算M_i的错误率error(M_i)（8.34式）
(6)     if error(M_i) > 0.5 then
(7)        转步骤（3）重试;
(8)     endif
(9)     for D_i的每个被正确分类的元组do
(10)       元组的权重乘以error(M_i)/(1-error(M_i));   // 更新权重
(11)    规范化每个元组的权重;
(12) endfor
```

Adaboost算法

使用组合分类器对元组x分类：

(1) 将每个类的权重初始化为0；

(2) **for** $i = 1$ to k **do** // 对于每个分类器

(3) $w_i = \log \frac{1-error(M_i)}{error(M_i)}$; // 分类器的投票权重

(4) $c = M_i(\mathbf{x})$; // 从M_i得到$\mathbf{x}$的类预测

(5) 将w_i加到类c的权重；

(6) **endfor**

(7) 返回具有最大权重的类；

提升算法的优缺点

- 可以获得比bagging更高的准确率
- 容易过度拟合

随机森林（Random Forest）算法

- 由很多决策树分类器组合而成（因而称为“森林”）
- 单个的决策树分类器用随机方法构成。首先，学习集是从原训练集中通过有放回抽样得到的自助样本。其次，参与构建该决策树的变量也是随机抽出，参与变量数通常大大小于可用变量数。
- 单个决策树在产生学习集和确定参与变量后，使用CART算法计算，不剪枝
- 最后分类结果取决于各个决策树分类器简单多数选举

随机森林算法优点

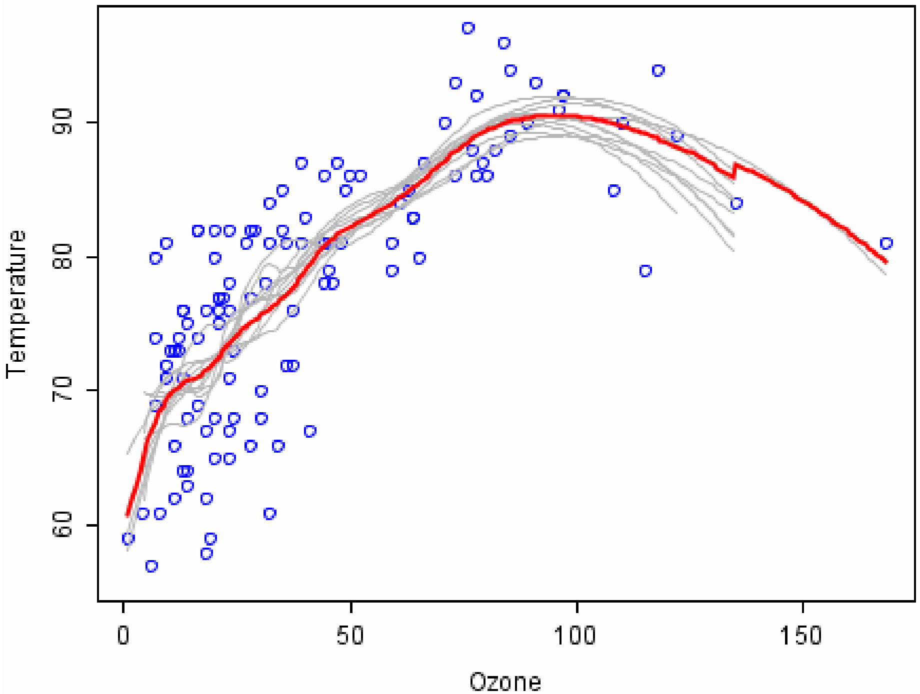
- 准确率可以和Adaboost媲美
- 对错误和离群点更加鲁棒性
- 决策树容易过度拟合的问题会随着森林规模而削弱
- 在大数据情况下速度快，性能好

随机森林/Bagging和决策树的关系

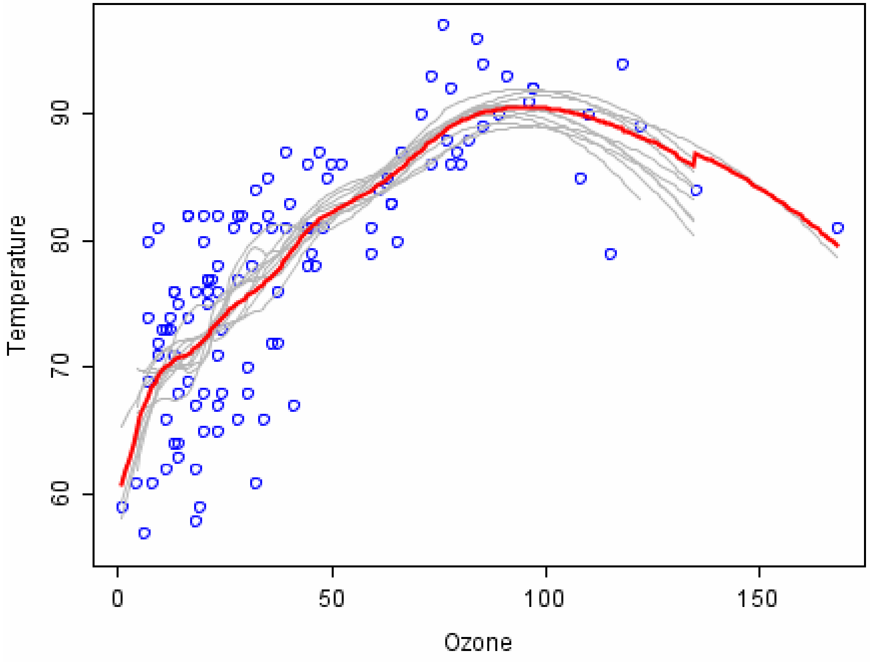
- 当然可以使用决策树作为基本分类器
- 但也可以使用SVM、Logistic回归等其他分类器，习惯上，这些分类器组成的“总分类器”，仍然叫做随机森林。
- 举例
  - 回归问题

回归问题

- 离散点是样本集合，描述了臭氧(横轴)和温度(纵轴)的关系
- 试拟合二者的变化曲线



使用Bagging



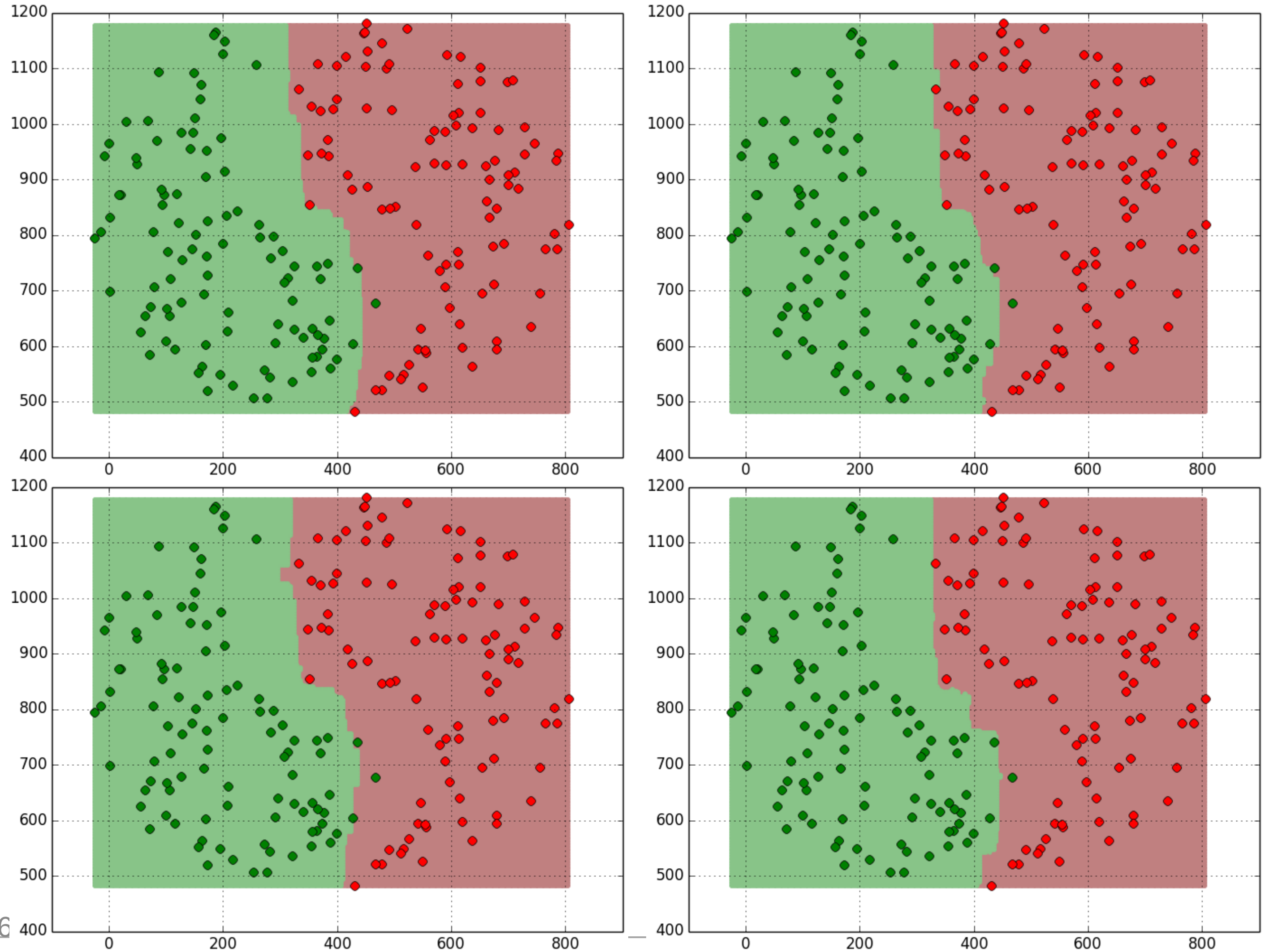
记原始数据为D，长度为N(即图中有N个离散点)

- 算法过程
  - 做100次bootstrap，每次得到的数据Di，Di的长度为N
  - 对于每一个Di，使用局部回归(LOESS)拟合一条曲线(图中灰色线是其中的10条曲线)
  - 将这些曲线取平均，即得到红色的最终拟合曲线
  - 显然，红色的曲线更加稳定，并且没有过拟合明显减弱

随机森林：30，

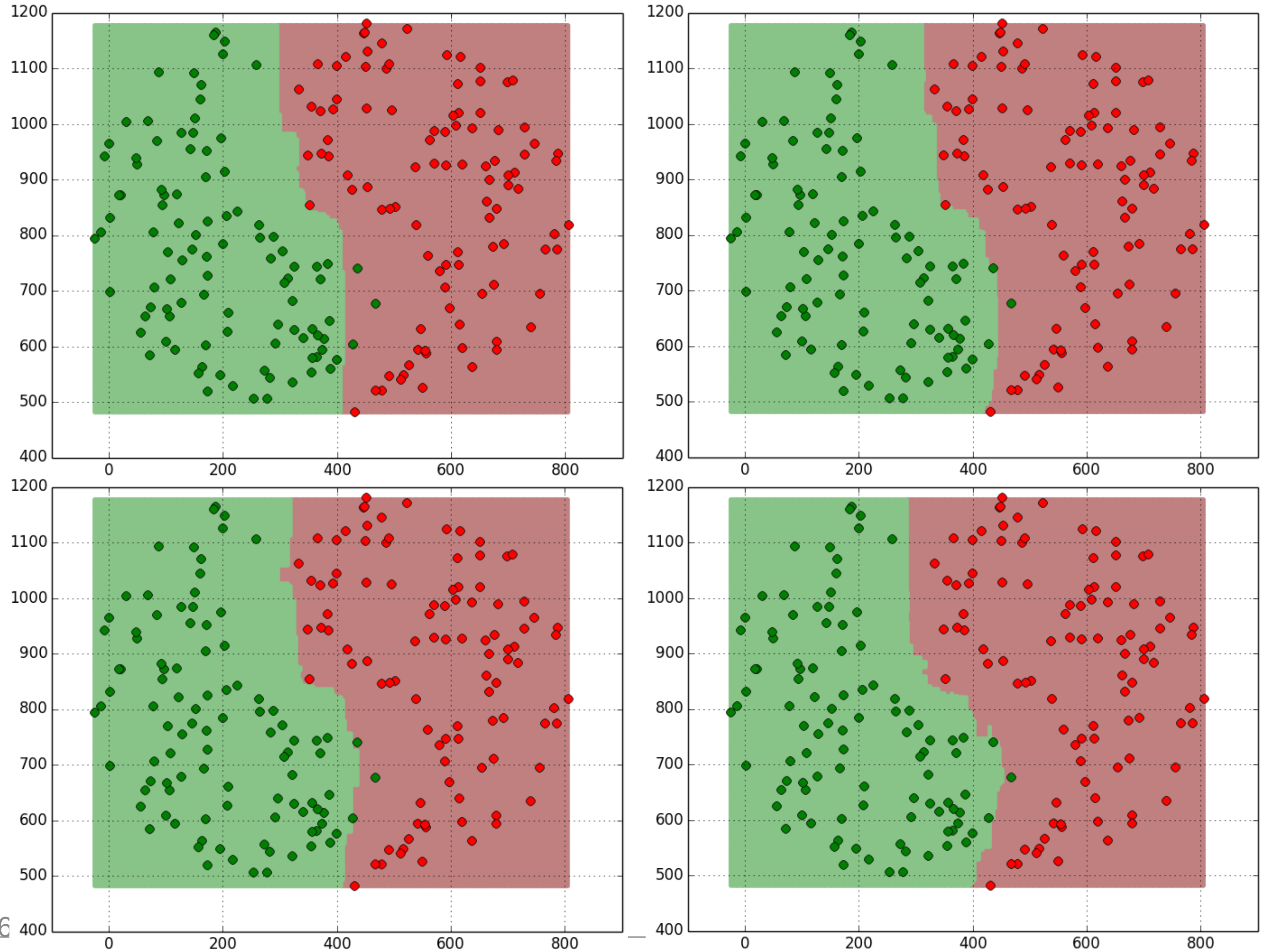
4 | 4

5 | 5

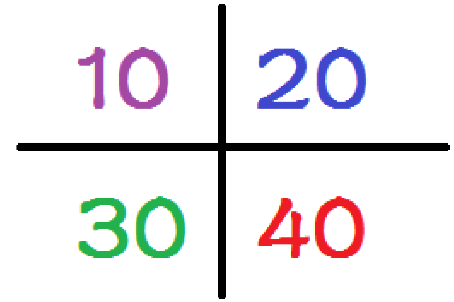


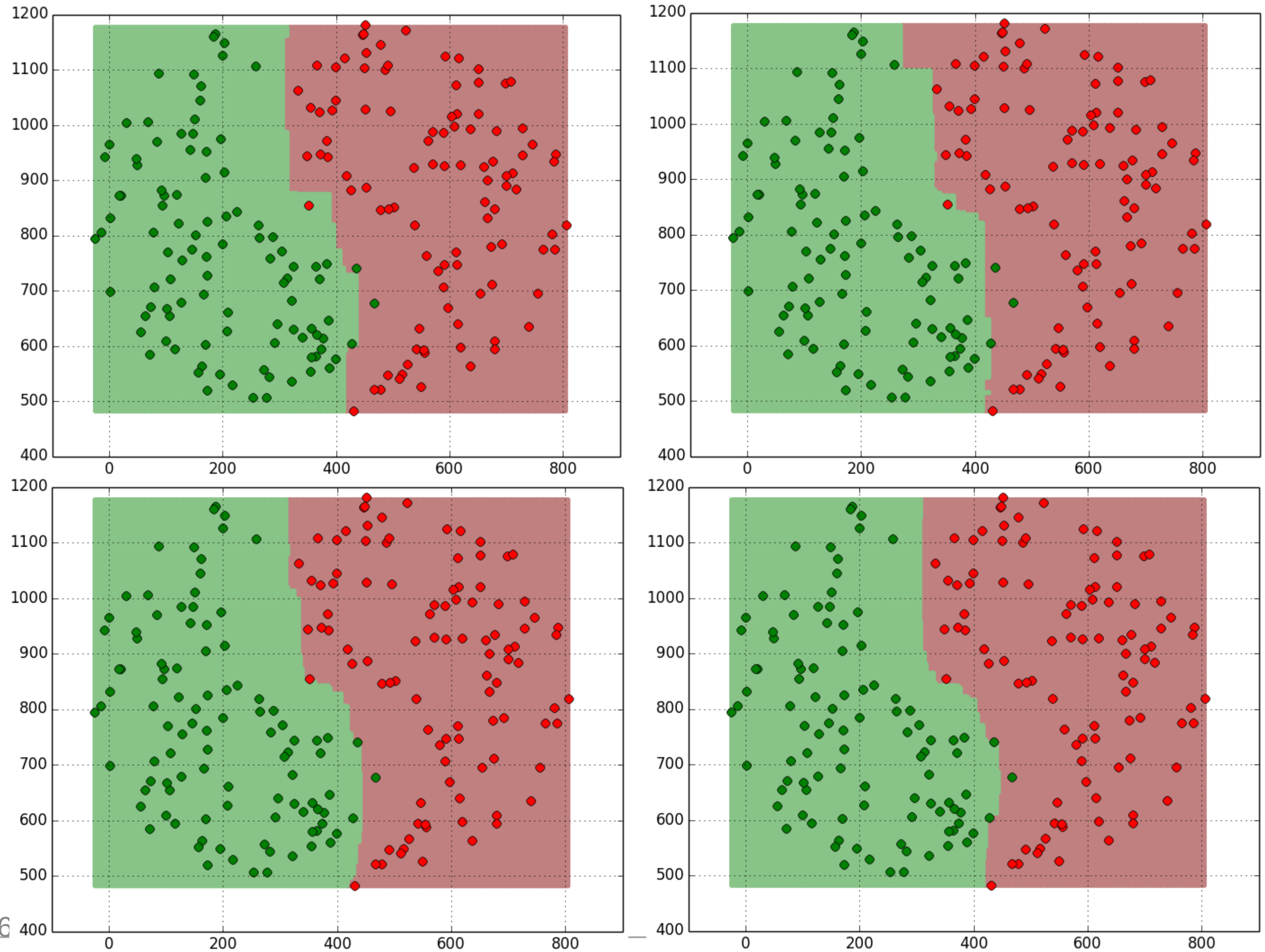
随机森林：30，

3	4
5	6

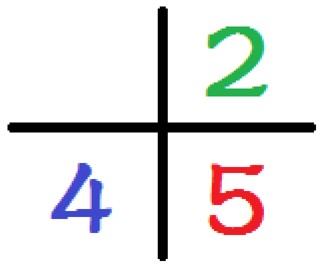


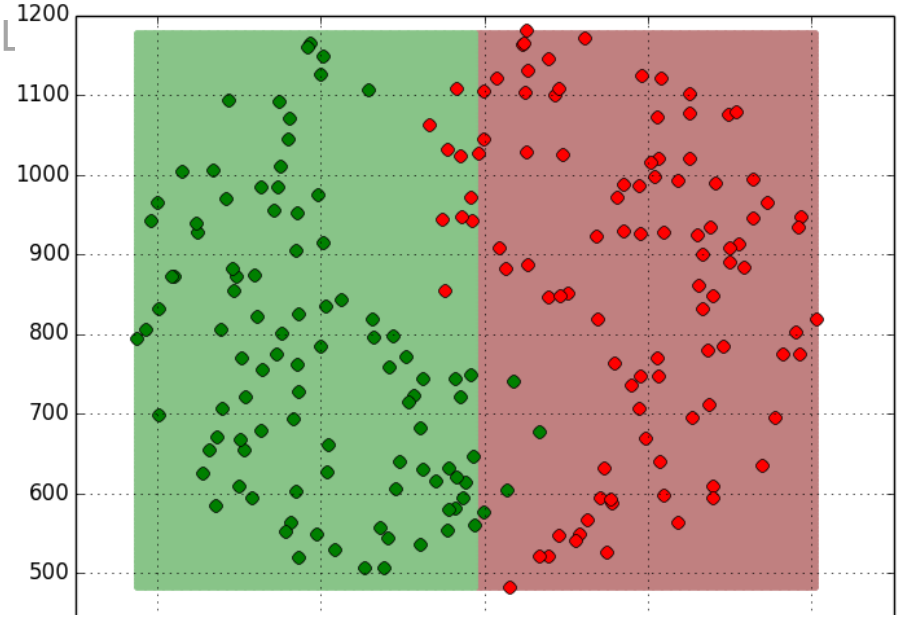
随机森林：4，

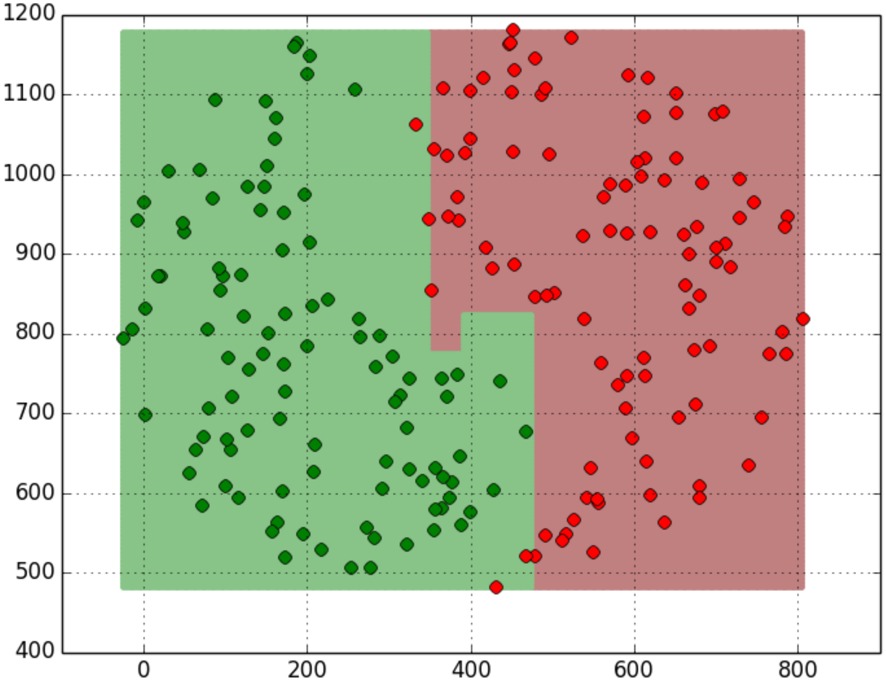

10
20
30
40

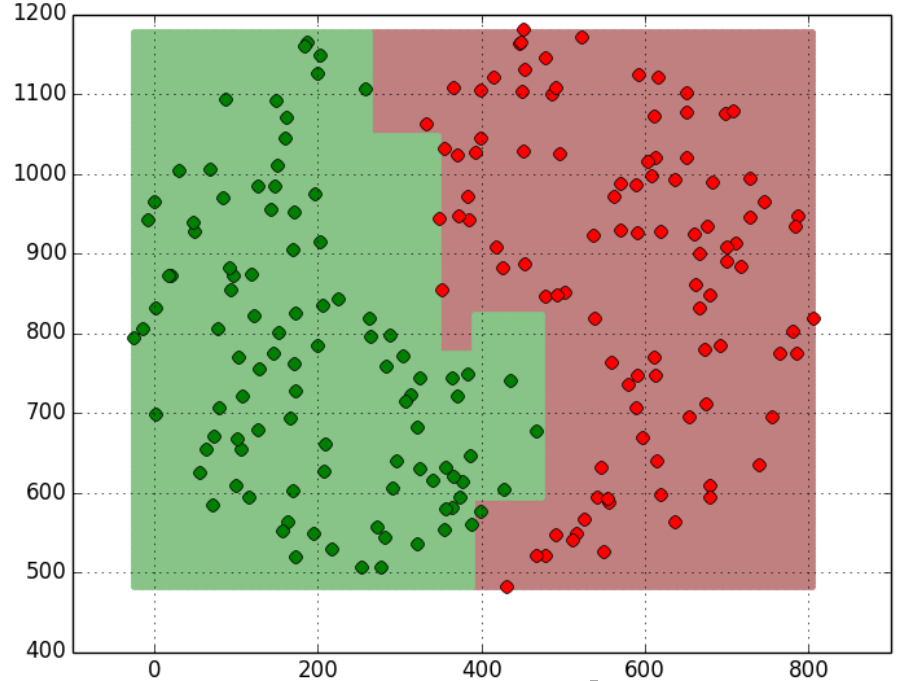

1200
1100
1000
900
800
700
600
500
400
0
200
400
600
800

决策树： Level


2
4
5


1200
1100
1000
900
800
700
600
500


1200
1100
1000
900
800
700
600
500
400
0
200
400
600
800


1200
1100
1000
900
800
700
600
500
400
0
200
400
600
800

Code

```python
def split(self, tree):
    f = self.select_feature()
    self.choose_value(f, tree)

def select_feature(self):   # 返回当前
    n = len(data[0])
    if rf:
        return random.randint(0,n-2)
    gini_f = 1          # gini指数最大是
    f = -1              # gini指数最小的
    for i in range(n-1):
        g = self.gini_feature(i)
        if gini_f > g:
            gini_f = g
            f = i
    return f
```

```python
class TreeNode:
    def __init__(self):
        self.sample = []     # 该结点拥有哪些样本
        self.feature = -1    # 用几号特征划分
        self.value = 0       # 该特征的取值
        self.type = -1       # 该结点的类型
        self.left = -1       # 该节点的左孩子
        self.right = -1      # 该节点的右孩子
        self.gini = 0

    def gini_coefficient(self):
        types = {}
        for i in self.sample:
            type = data[i][-1]
            if types.has_key(type):
                types[type] += 1
            else:
                types[type] = 1
        pp = 0
        m = float(len(self.sample))
        for t in types:
            pp += (float(types[t]) / m) ** 2
        self.gini = 1-pp
        max_type = 0
        for t in types:
            if max_type < types[t]:
                max_type = types[t]
                self.type = t
```

```python
def choose_value(self, f, tree):
    f_max = self.calc_max(f)
    f_min = self.calc_min(f)
    step = (f_max - f_min) / granularity
    if step == 0:
        return f_min
    x_split = 0
    g_split = 1
    for x in numpy.arange(f_min+step, f_max, step):
        if rf:
            x = random.uniform(f_min, f_max)
        g = self.gini_coefficient2(f, x)
        if g_split > g:
            g_split = g
            x_split = x
    if g_split < self.gini: # 分割后gini系数要变小才有意义
        self.value = x_split
        self.feature = f
        t = TreeNode()
        t.sample = self.choose_sample(f, x_split, True)
        t.gini_coefficient()
        self.left = len(tree)
        tree.append(t)
        t = TreeNode()
        t.sample = self.choose_sample(f, x_split, False)
        t.gini_coefficient()
        self.right = len(tree)
        tree.append(t)
```

```python
def predict_tree(d, tree):
    node = tree[0]
    while node.left != -1 and node.right != -1:
        if d[node.feature] < node.value:
            node = tree[node.left]
        else:
            node = tree[node.right]
    return node.type
```

```python
def decision_tree():
    m = len(data)
    n = len(data[0])
    tree = []
    root = TreeNode()
    if rf:
        root.sample = random_select(alpha)
    else:
        root.sample = [x for x in range(m)]
    root.gini_coefficient()
    tree.append(root)
    first = 0
    last = 1
    for level in range(max_level):
        for node in range(first, last):
            tree[node].split(tree)
        first = last
        last = len(tree)
        print level+1, len(tree)
    return tree
```

```python
def predict(d, forest):
    pd = {}
    for tree in forest:
        type = predict_tree(d, tree)
        if pd.has_key(type):
            pd[type] += 1
        else:
            pd[type] = 1
    number = 0
    type = 0.0
    for p in pd:
        if number < pd[p]:
            number = pd[p]
            type = p
    return type
```

参考文献

- Thomas M. Cover, Joy A. Thomas, Elements of Information Theory,2006
- Christopher M. Bishop, Pattern Recognition and Machine Learning, Springer-Verlag, 2006
- 李航，统计学习方法，清华大学出版社，2012
- Jamie Shotton, Andrew Fitzgibbon, etc, Real-Time Human Pose Recognition in Parts from Single Depth Images, 2011