



南開大學
Nankai University

计算机学院
机器学习实验报告

决策分类器实验

姓名：周重天
学号：2311082
专业：计算机科学与技术

2025 年 12 月 17 日

目录

1 基础任务：ID3 决策树构建与预测（离散属性）	3
1.1 ID3 算法简介	3
1.2 关键算法步骤	3
1.2.1 第一步：信息熵计算	3
1.2.2 第二步：信息增益计算	3
1.2.3 第三步：递归建树	4
1.2.4 代码讲解	5
1.3 实验结果	6
1.4 关键代码：预测函数	6
2 中级要求：C4.5 决策树构建与预测	6
2.1 C4.5 算法改进简介	6
2.2 连续属性处理方法	6
2.2.1 代码讲解	7
2.3 信息增益率	8
2.3.1 代码讲解	8
2.4 实验结果	9
3 C4.5 算法属性处理过程	9
3.1 属性类型自动识别	9
3.1.1 代码讲解	10
4 高级要求：CART 决策树构建与预测	10
4.1 CART 算法简介	10
4.2 基尼指数	10
4.3 CART 建树算法	10
4.3.1 代码讲解	11
4.4 实验结果	11
5 高级要求：决策树剪枝	12
5.1 剪枝算法选择：简化后剪枝（REP）	12
5.2 REP 算法流程	12
5.2.1 代码讲解	13
5.3 剪枝实施结果	14
5.3.1 ID3 树的剪枝	14
5.3.2 C4.5 树的剪枝	14
5.3.3 CART 树的剪枝	14
5.4 剪枝分析	14
6 扩展要求：三种决策树算法比较分析	14
6.1 维度一：分裂准则	14
6.2 维度二：树结构	15

6.3	维度三：属性处理能力	15
6.4	算法选择建议	15
6.5	本实验观察	15
7	附录：代码仓库	15

1 基础任务：ID3 决策树构建与预测（离散属性）

1.1 ID3 算法简介

ID3 (Iterative Dichotomiser 3) 决策树是由 Ross Quinlan 提出的经典算法，仅适用于离散属性。ID3 使用信息增益作为属性选择准则，通过递归地选择增益最大的属性进行分裂，直至满足停止条件。

ID3 决策树的核心思想是：选择能够最大程度减少数据集不纯度（熵）的属性作为分裂属性。

1.2 关键算法步骤

1.2.1 第一步：信息熵计算

对于数据集 S ，信息熵的计算公式为：

$$H(S) = - \sum_{i=1}^c p_i \log_2(p_i) \quad (1)$$

其中 p_i 表示类别 i 在数据集中的比例， c 为类别总数。

本实验中的实现如下：

Listing 1: 信息熵计算

```
1 def entropy(self, labels):
2     """计算信息熵 H(S) = - p_i * log2(p_i)"""
3     if len(labels) == 0:
4         return 0.0
5     value_counts = Counter(labels)
6     entropy_val = 0.0
7     total = len(labels)
8     for count in value_counts.values():
9         if count > 0:
10             p = count / total
11             entropy_val -= p * np.log2(p)
12     return entropy_val
```

1.2.2 第二步：信息增益计算

对于属性 A ，其信息增益定义为：

$$\text{Gain}(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \quad (2)$$

其中 S_v 表示属性 A 取值为 v 的样本子集。

实现代码：

Listing 2: 信息增益计算

```
1 def information_gain(self, parent_labels, child_labels_list):
2     """
3     计算信息增益
```

```

4 Gain(S, A) = Entropy(S) - |S_v|/|S| * Entropy(S_v)
5 """
6 parent_entropy = self.entropy(parent_labels)
7 total_samples = len(parent_labels)
8 if total_samples == 0:
9     return 0.0
10 weighted_child_entropy = 0.0
11 for child_labels in child_labels_list:
12     if len(child_labels) > 0:
13         weight = len(child_labels) / total_samples
14         weighted_child_entropy += weight * self.entropy(child_labels)
15 return parent_entropy - weighted_child_entropy

```

1.2.3 第三步: 递归建树

算法遍历所有离散属性, 计算各属性的信息增益, 选择增益最大的属性作为当前节点的分裂属性。然后按该属性的不同取值划分样本子集, 对每个子集递归调用建树算法。

递归停止条件:

1. 子集中所有样本属于同一类别
2. 没有可用属性可进行分裂
3. 子集为空

核心递归代码片段:

Listing 3: ID3 递归建树

```

1 def build_tree(self, X, y, available_features):
2     node = TreeNode()
3     node.samples = len(y)
4     node.class_distribution = dict(Counter(y))
5     node.majority_class = Counter(y).most_common(1)[0][0] if len(y) > 0 else None
6
7     # 停止条件 1: 标签全相同
8     if len(np.unique(y)) <= 1:
9         node.is_leaf = True
10        node.prediction = y.iloc[0] if len(y) > 0 else None
11        return node
12
13    # 停止条件 2 和 3: 没有属性或子集为空
14    if len(available_features) == 0 or len(X) == 0:
15        node.is_leaf = True
16        node.prediction = node.majority_class
17        return node
18
19    # 选择最优分裂属性 (信息增益最大)
20    best_feature = None
21    best_gain = -1

```

```
22     for feature in available_features:
23         feature_values = X[feature].unique()
24         child_labels_list = []
25         for value in feature_values:
26             mask = X[feature] == value
27             child_labels = y[mask]
28             child_labels_list.append(child_labels.values)
29         gain = self.information_gain(y.values, child_labels_list)
30         if gain > best_gain:
31             best_gain = gain
32             best_feature = feature
33
34     # 递归构建子树
35     remaining_features = [f for f in available_features if f != best_feature]
36     node.feature = best_feature
37     node.children = {}
38     for value in X[best_feature].unique():
39         mask = X[best_feature] == value
40         X_subset = X[mask].reset_index(drop=True)
41         y_subset = y[mask].reset_index(drop=True)
42         node.children[value] = self.build_tree(X_subset, y_subset, remaining_features)
43     return node
```

1.2.4 代码讲解

递归建树函数 `build_tree` 是 ID3 算法的核心, 其执行流程如下:

1. **节点初始化**: 创建新的树节点, 计算该节点所有样本的多数类。这用于处理无法完全分裂的情况下的默认预测。
2. **停止条件检查**:
 - 若所有样本标签相同 (`len(np.unique(y)) <= 1`), 则该节点是纯净的, 无需继续分裂, 直接创建叶子节点。
 - 若没有可用属性 (`len(available_features) == 0`) 或样本子集为空, 也无法继续分裂, 创建叶子节点并使用多数类作为预测。
3. **属性选择**: 遍历所有可用属性, 对每个属性计算其信息增益。具体步骤为:
 - 获取该属性的所有不同取值 (`X[feature].unique()`)
 - 按各取值将样本分组, 构建子集标签列表
 - 调用 `information_gain` 方法计算该属性的增益
 - 记录增益最大的属性作为最优分裂属性
4. **递归分裂**: 选定最优分裂属性后, 按该属性的每个不同取值创建子分支。对于每个分支:
 - 使用布尔掩码 `mask` 筛选出满足该属性值的样本
 - 从可用属性集合中移除已分裂的属性 (`remaining_features`)
 - 递归调用 `build_tree` 构建子树

1.3 实验结果

使用 Watermelon-train1.csv (纯离散属性) 训练 ID3 决策树, 对 Watermelon-test1.csv 进行预测。
测试集结果:

表 1: ID3 决策树在 test1 上的性能

指标	值
训练集样本数	16
测试集样本数	10
特征维数	5
分类精度	0.70 (70%)

[任务 1] ID3 决策树 (离散属性)

训练集: 16 样本, 5 特征
测试集: 10 样本, 5 特征
ID3 test1 accuracy: 0.70

图 1.1: ID3 决策树实验运行结果

1.4 关键代码: 预测函数

Listing 4: ID3 预测

```

1 def predict_one(self, x):
2     """预测单个样本"""
3     node = self.tree
4     while not node.is_leaf:
5         feature = node.feature
6         feature_value = x.get(feature, None)
7         # 如果特征值不在子节点中, 使用多数类回退
8         if feature_value not in node.children:
9             return node.majority_class
10        node = node.children[feature_value]
11    return node.prediction

```

2 中级要求: C4.5 决策树构建与预测

2.1 C4.5 算法改进简介

C4.5 是对 ID3 的重要改进, 主要区别在于:

1. 支持连续属性的处理
2. 使用信息增益率代替信息增益, 避免对多值属性的偏向

2.2 连续属性处理方法

对于连续属性, C4.5 采用如下策略:

将属性值从小到大排序，相邻两个不同值的中点作为候选分裂阈值。对每个阈值，将数据分为两部分： $A \leq t$ 和 $A > t$ 。

Listing 5: 连续属性最优阈值寻找

```

1 def find_best_continuous_split(self, X_col, y, feature_name):
2     """
3     为连续属性寻找最优分裂阈值
4     候选阈值：排序后相邻不同值的中点
5     """
6     X_col_sorted = sorted(set(X_col.values))
7     if len(X_col_sorted) <= 1:
8         return None, -1
9     thresholds = []
10    for i in range(len(X_col_sorted) - 1):
11        threshold = (X_col_sorted[i] + X_col_sorted[i + 1]) / 2
12        thresholds.append(threshold)
13    best_threshold = None
14    best_gain_ratio = -1
15    for threshold in thresholds:
16        left_mask = X_col <= threshold
17        right_mask = X_col > threshold
18        y_left = y[left_mask]
19        y_right = y[right_mask]
20        if len(y_left) == 0 or len(y_right) == 0:
21            continue
22        child_labels_list = [y_left.values, y_right.values]
23        gain_ratio = self.information_gain_ratio(y.values, child_labels_list)
24        if gain_ratio > best_gain_ratio:
25            best_gain_ratio = gain_ratio
26            best_threshold = threshold
27    return best_threshold, best_gain_ratio

```

2.2.1 代码讲解

连续属性处理是 C4.5 相比 ID3 的关键改进。`find_best_continuous_split` 函数的核心思想是：

- 候选阈值生成：**对属性值进行排序并提取唯一值。相邻两个不同值的中点被作为候选分裂阈值。例如，若排序后的唯一值为 $[0.5, 0.7, 0.9]$ ，则候选阈值为 $[0.6, 0.8]$ 。这种方式能有效减少阈值的搜索空间。
- 阈值评估：**对每个候选阈值，将数据分为两部分： $X \leq \text{threshold}$ 和 $X > \text{threshold}$ 。计算此二分裂下的信息增益率。
- 最优阈值选择：**在所有有效的二分裂中（两部分都非空），选择能最大化增益率的阈值。

相比穷举所有可能的分裂点，该方法大大降低了计算复杂度，同时保留了充分的表达能力。

2.3 信息增益率

为避免 ID3 对多值属性的偏向, C4.5 引入信息增益率:

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInfo}(S, A)} \quad (3)$$

其中分裂信息定义为:

$$\text{SplitInfo}(S, A) = - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|} \quad (4)$$

Listing 6: C4.5 信息增益率

```

1 def information_gain_ratio(self, parent_labels, child_labels_list):
2     """
3     计算信息增益率
4     GainRatio(S, A) = Gain(S, A) / SplitInfo(S, A)
5     """
6     parent_entropy = self.entropy(parent_labels)
7     total_samples = len(parent_labels)
8     if total_samples == 0:
9         return 0.0
10    weighted_child_entropy = 0.0
11    child_sizes = []
12    for child_labels in child_labels_list:
13        child_sizes.append(len(child_labels))
14        if len(child_labels) > 0:
15            weight = len(child_labels) / total_samples
16            weighted_child_entropy += weight * self.entropy(child_labels)
17    gain = parent_entropy - weighted_child_entropy
18    split_info = self.split_info(child_sizes)
19    if split_info == 0:
20        return 0.0
21    gain_ratio = gain / split_info
22    return gain_ratio

```

2.3.1 代码讲解

信息增益率是对信息增益的改进, 解决了 ID3 对多值属性的偏好问题。代码实现流程如下:

1. **基础增益计算:** 首先计算属性分裂前的熵 (`parent_entropy`), 然后计算分裂后各子集的加权熵, 二者之差即为原始的信息增益。
2. **分裂信息计算:** `split_info` 方法计算分裂信息, 其值随着分支数增加而增加。多值属性会产生较大的分裂信息。
3. **增益率归一化:** 通过将增益除以分裂信息, 使得多值属性的增益被压低, 避免算法过度偏好多值属性。这是 C4.5 相比 ID3 的根本改进。

4. **边界处理**：当分裂信息为 0（即所有分支大小都相等且只有一个分支）时，避免除以零的错误。

在实践中，信息增益率能产生更平衡的决策树，对未见过的数据有更好的泛化能力。

2.4 实验结果

使用 Watermelon-train2.csv（离散 + 连续属性混合）训练 C4.5 决策树，对 Watermelon-test2.csv 进行预测。

表 2: C4.5 决策树在 test2 上的性能

指标	值
训练集样本数	17
测试集样本数	5
特征维数	6
其中连续属性数	2
分类精度	0.60 (60%)

[任务 2] C4.5 决策树

```
训练集：17 样本，6 特征
测试集：5 样本，6 特征
C45 test2 accuracy: 0.60
```

图 2.2: C4.5 决策树实验运行结果

3 C4.5 算法属性处理过程

3.1 属性类型自动识别

C4.5 利用 pandas 的 `is_numeric_dtype` 函数自动识别连续属性：

Listing 7: 属性类型自动识别

```
1 def fit(self, X, y):
2     """训练 C4.5 决策树"""
3     self.feature_names = X.columns.tolist()
4     self.class_label = y.name if y.name else "label"
5     # 自动识别连续属性
6     from pandas.api.types import is_numeric_dtype
7     for col in self.feature_names:
8         if is_numeric_dtype(X[col]):
9             self.continuous_features.add(col)
10    available_features = [f for f in self.feature_names]
11    self.tree = self.build_tree(X.reset_index(drop=True),
12                                y.reset_index(drop=True),
13                                available_features)
14    return self
```

3.1.1 代码讲解

属性类型自动识别是 C4.5 实现的重要细节，使得算法能自动适应混合属性数据集：

1. **属性收集**：fit 方法首先提取数据集的所有特征列名，保存在 `feature_names` 中。
2. **类型识别**：使用 pandas 库的 `is_numeric_dtype` 函数检查每个特征列的数据类型。若为数值型 (int、float 等)，则标记为连续属性，加入 `continuous_features` 集合；否则视为离散属性。
3. **差异化处理**：在建树过程中，当遍历特征选择最优分裂属性时，会针对特征类型采用不同策略：
 - 连续特征：调用 `find_best_continuous_split` 寻找最优二分阈值
 - 离散特征：枚举所有不同取值，按值进行多路分裂
4. **用户友好性**：自动识别机制让用户无需手动预处理数据，直接传入混合属性数据集即可。相比 ID3 需要预先离散化连续属性，C4.5 更加易用。

4 高级要求：CART 决策树构建与预测

4.1 CART 算法简介

CART (Classification And Regression Tree) 由 Breiman 等人提出。CART 的主要特点：

1. 总是生成二叉树结构
2. 使用基尼指数 (Gini Index) 作为分裂准则
3. 支持离散和连续属性

4.2 基尼指数

对数据集 D ，基尼指数定义为：

$$\text{Gini}(D) = 1 - \sum_{k=1}^c p_k^2 \quad (5)$$

其中 p_k 是类别 k 在数据集中的比例。基尼指数反映数据集的不纯度。

4.3 CART 建树算法

CART 对连续属性和离散属性都采用二分策略。对于离散属性，每个节点分裂为“取值为 v ”和“取值不为 v ”两个分支。

Listing 8: CART 基尼指数计算

```
1 def gini_index(self, labels):
2     """计算基尼指数 Gini(D) = 1 - p_k^2"""
3     if len(labels) == 0:
4         return 0.0
5     value_counts = Counter(labels)
6     gini = 1.0
```

```

7     total = len(labels)
8     for count in value_counts.values():
9         if count > 0:
10             p = count / total
11             gini -= p * p
12     return gini
13
14 def weighted_gini(self, left_labels, right_labels):
15     """计算二分后的加权基尼指数"""
16     total = len(left_labels) + len(right_labels)
17     if total == 0:
18         return 0.0
19     left_weight = len(left_labels) / total
20     right_weight = len(right_labels) / total
21     return (left_weight * self.gini_index(left_labels) +
22            right_weight * self.gini_index(right_labels))

```

4.3.1 代码讲解

CART 使用基尼指数作为分裂准则，相比信息论方法具有计算优势。代码涵盖两个关键函数：

1. 基尼指数计算 (gini_index):

- 遍历数据集中各类别，计算每类的比例 p_k
- 基尼指数定义为 $Gini(D) = 1 - \sum_{k=1}^c p_k^2$ ，体现数据的不纯度
- 当数据集只包含一个类别时，基尼指数为 0（纯净）；当各类比例相等时，基尼指数最大

2. 加权基尼指数 (weighted_gini):

- 对于二分裂，分别计算左右两个子集的基尼指数
- 按子集大小比例进行加权求和
- 公式为: $Gini_{split} = \frac{|L|}{|D|} \cdot Gini(L) + \frac{|R|}{|D|} \cdot Gini(R)$
- 在寻找最优分裂时，选择能最小化加权基尼指数的分裂点

CART 的二分策略（每次只分裂成两个分支）使得树的结构更规则，易于剪枝和理解。虽然可能导致树深度较深，但通过剪枝可以获得更优的复杂度-精度平衡。

4.4 实验结果

使用相同的 Watermelon-train2.csv 训练 CART 决策树。

表 3: CART 决策树在 test2 上的性能

指标	值
训练集样本数	17
测试集样本数	5
特征维数	6
树结构	二叉树
分类精度	0.60 (60%)

```

[任务 3] CART 决策树
-----
训练集：17 样本，6 特征
测试集：5 样本，6 特征
CART test2 accuracy: 0.60

```

图 4.3: CART 决策树实验运行结果

5 高级要求：决策树剪枝

5.1 剪枝算法选择：简化后剪枝（REP）

本实验采用简化后剪枝（Reduced Error Pruning, REP）算法。REP 是一种后剪枝方法，通过在验证集上评估剪枝效果来决定是否进行剪枝。

5.2 REP 算法流程

1. 从训练集中随机分出 20% 作为验证集（random_state=42 保证可复现性）
2. 在较小的训练集上重新训练决策树
3. 对每个内部节点，尝试将其替换为叶子（预测为该节点的多数类）
4. 若替换后验证集精度不下降，则保留该剪枝操作
5. 自底向上重复第 3、4 步，直到无法继续剪枝

Listing 9: 简化后剪枝实现

```

1 def reduced_error_pruning(tree_model, X_valid, y_valid):
2     """
3     简化后剪枝 (Reduced Error Pruning, REP)
4     """
5     def calculate_accuracy(model, X, y):
6         predictions = model.predict(X)
7         return np.mean(predictions == y.values)
8
9     def prune_node(node, X_valid, y_valid, model):
10        """递归剪枝"""
11        if node.is_leaf:
12            return False
13        # 先对所有子节点尝试剪枝
14        any_pruned = False
15        for child in node.children.values():
16            if prune_node(child, X_valid, y_valid, model):
17                any_pruned = True
18        # 计算剪枝前的精度
19        acc_before = calculate_accuracy(model, X_valid, y_valid)
20        # 尝试将当前节点替换为叶子
21        old_is_leaf = node.is_leaf

```

```
22     old_prediction = node.prediction
23     old_feature = node.feature
24     old_children = node.children
25     node.is_leaf = True
26     node.prediction = node.majority_class
27     node.feature = None
28     node.children = {}
29     # 计算剪枝后的精度
30     acc_after = calculate_accuracy(model, X_valid, y_valid)
31     # 如果精度没有下降，保留剪枝
32     if acc_after >= acc_before:
33         return True
34     else:
35         # 恢复原状
36         node.is_leaf = old_is_leaf
37         node.prediction = old_prediction
38         node.feature = old_feature
39         node.children = old_children
40         return any_pruned
41
42 for _ in range(100): # 最多迭代 100 次
43     if not prune_node(tree_model.tree, X_valid, y_valid, tree_model):
44         break
45 return tree_model
```

5.2.1 代码讲解

简化后剪枝（REP）通过在验证集上的精度变化来指导剪枝决策。实现原理如下：

1. **验证集分离**：在调用剪枝前，通常从训练集中分出 20% 作为验证集（通过 `np.random.choice`）。验证集用于评估剪枝效果，不参与树的构建。
2. **递归剪枝流程**（`prune_node` 函数）：
 - 若节点是叶子，无需剪枝，直接返回
 - 否则，先对所有子节点递归尝试剪枝，自底向上进行
 - 对当前节点，记录原始状态（`old_*` 变量）
 - 尝试将节点替换为叶子（预测为该节点的多数类），临时改变节点属性
 - 在验证集上计算替换前后的精度（`acc_before` 和 `acc_after`）
 - 若替换后精度不下降，则保留剪枝；否则恢复原状
3. **迭代优化**：外层循环最多迭代 100 次，每次调用 `prune_node`。只要还有节点被成功剪枝，就继续迭代。当某次迭代无任何剪枝发生，算法停止。
4. **关键特性**：
 - REP 是后剪枝方法，先构建完整的树再剪枝，避免过早停止导致信息丧失

- 使用独立的验证集而非训练集评估剪枝效果，更客观地反映泛化能力
- 自底向上的递归策略确保了剪枝的系统性和一致性

5.3 剪枝实施结果

5.3.1 ID3 树的剪枝

表 4: ID3 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.50	-0.10
节点数	14	1	-13
树深度	1	0	-1

5.3.2 C4.5 树的剪枝

表 5: C4.5 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.60	0.00
节点数	3	3	0
树深度	1	1	0

5.3.3 CART 树的剪枝

表 6: CART 决策树剪枝前后对比

指标	剪枝前	剪枝后	变化
测试集精度	0.60	0.60	0.00
节点数	3	3	0
树深度	1	1	0

5.4 剪枝分析

ID3 树的剪枝导致精度下降，说明该模型在验证集上的最优结构保留了所有节点。而 C4.5 和 CART 在剪枝前已经相对简洁，没有进一步的优化空间。这表明对于 Watermelon 数据集，原始的 C4.5 和 CART 模型已经具有良好的泛化能力。

6 扩展要求：三种决策树算法比较分析

6.1 维度一：分裂准则

表 7: 三种算法的分裂准则对比

算法	分裂准则	公式	特点
ID3	信息增益	$\text{Gain}(S, A) = H(S) - \sum_v \frac{ S_v }{ S } H(S_v)$	偏向多值属性
C4.5	信息增益率	$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInfo}(S, A)}$	消除多值偏向
CART	基尼指数	$\text{Gini}(D) = 1 - \sum_k p_k^2$	计算简便

6.2 维度二：树结构

表 8: 三种算法的树结构对比

算法	树结构类型	分裂策略	优缺点
ID3	多叉树	多路分裂	结构紧凑，易理解；剪枝困难
C4.5	多叉树	多路分裂	继承 ID3；支持连续属性
CART	二叉树	二分	易剪枝，规则明确；树可能较深

6.3 维度三：属性处理能力

表 9: 三种算法的属性处理对比

算法	离散属性	连续属性	自动识别
ID3	✓原生支持	× 需预处理	×
C4.5	✓原生支持	✓自动二分	✓
CART	✓二分处理	✓自动二分	✓

6.4 算法选择建议

1. **ID3**: 适用于纯离散属性数据集，数据规模较小，属性值有限的场景。优点是算法简洁高效，缺点是不支持连续属性。
2. **C4.5**: 适用于混合属性数据集，属性数较多，类标签多的场景。通过增益率解决了 ID3 的多值偏向问题，扩展了连续属性支持。
3. **CART**: 适用于生产环境，需要回归扩展的场景。二叉树结构易于剪枝和理解，基尼指数计算更快。

6.5 本实验观察

在 Watermelon 数据集上，三种算法在 test2 上的精度都达到 0.60。C4.5 和 CART 都能有效处理连续属性（密度、含糖量等），而 ID3 仅限于离散属性。从模型复杂度看，C4.5 和 CART 生成的树都比较简洁（3 个节点），具有较好的泛化能力。

7 附录：代码仓库

本实验的全部源代码已上传至 GitHub 仓库：

GitHub 仓库地址: <https://github.com/sskystack/machinelearning.git>

源代码文件: work6/exp_6.py