决策树与随机森林实验报告

姓名：_____ 张韫译萱_____　学号：_____08023214_____

一、 实验题目

1 实验名称

使用决策树与随机森林实现 LIBSVM Data/dna 数据集分类。

2 实验任务

1. 使用决策树算法实现分类。
2. 使用随机森林算法实现分类。
3. 比较不同算法在 DNA 数据集上的性能。

3 实验要求

1. Python 实现。
2. 不使用现成决策树和随机森林库函数。

二、 实验原理

1 决策树

决策树是一种基于树形结构的监督学习算法，通过对特征空间进行递归划分来实现分类或回归任务。决策树由内部节点、分支和叶节点组成，其中内部节点表示特征测试，分支表示测试结果，叶节点表示类别标签或预测值。

1.1 基本原理

决策树的构建过程是一个递归的特征选择和数据划分过程。在每个内部节点上，算法选择一个最优特征进行数据划分，使得划分后的子集在某种度量标准下达到最优。这一过程持续进行，直到满足停止条件（所有划分后的节点中的样本均属于同一类别，属性集为空或决策树达到一定深度）。

本次实验使用信息增益作为特征选择的标准，下面是一些基本概念：

1.2 信息熵

信息熵用于度量数据集的不确定性或混乱程度。对于包含 K 个类别的数据集 D，其信息熵定义为：

$$H(D) = -\sum_{k=1}^{K} p_k \log_2 p_k$$

其中，K 为类别数，p_k 为第 k 个类别的样本占比。信息熵越大，表示数据集的不确定性越高；当所有样本属于同一类别时，熵为 0。

1.3 信息增益

信息增益是 ID3 算法的核心，用于衡量特征对数据集分类能力的贡献。对于特征 A，其信息增益定义为划分前后熵的差值：

$$\text{Gain}(D, A) = H(D) - \sum_{v=1}^{V} \frac{|D^v|}{|D|} H(D^v)$$

其中，V 为特征 A 的取值数量，D^v 为特征 A 取值为 v 的样本子集，$|D|$ 为数据集 D 的样本总数。信息增益越大，表明该特征对分类的贡献越大，越适合作为当前节点的分裂特征。

2 随机森林

随机森林是一种集成学习方法，通过构建多个决策树并综合其预测结果来提高模型的泛化能力和稳定性。随机森林结合了 Bagging 方法和随机特征选择，有效降低了模型的方差，避免了单一决策树的过拟合问题。

2.1 基本原理

随机森林通过构建多个弱分类器（决策树），通过投票机制综合其预测结果，可以获得比单一分类器更强的泛化能力。随机森林的构建过程包括两个关键的随机化步骤：

1. **Bootstrap 抽样（样本随机）**：从原始训练集中有放回地随机抽取 n 个样本，构成一个新的训练子集。构造 T 个这样的子集来训练 T 棵树，这种方法称为 Bagging。
2. **随机特征选择**：在每个节点分裂时，不考虑全部特征，而是从所有特征中随机选择 k 个特征（通常 $k = \log_2 d$，其中 d 为总特征数），然后在这 k 个特征中选择最优特征进行分裂。

三、 过程与结果

1 决策树的算法流程

训练

1. 计算当前树节点数据集 D 的信息熵 $H(D)$。
2. 对 DNA 数据集中的每个 DNA 特征 A，计算按该特征做分支后的信息增益 $\text{Gain}(D, A)$。

3. 选择信息增益最大的特征作为分支特征。
4. 按选定特征的不同取值将数据集划分为两类（该特征值为 0 的划分到左节点，为 1 的划分到右节点）。
5. 对每个子节点递归执行上述步骤，直到满足停止条件。

预测

1. 在经过分支节点时关注样本对应特征下的值。
2. 如样本特征的值为 0，则预测为左子树，为 1 则预测为右子树，重复执行该步骤直到遇到叶子节点。
3. 取叶子节点上的分类结果为样本预测结果。

2 随机森林的算法流程

训练

1. 设定森林中树的数量 T。
2. 对于每棵树 $t = 1, 2, \ldots, T$：
   - 从训练集中进行 t 次有放回抽样，生成训练子集 D_t。
   - 使用 D_t 构建决策树 h_t，在每个节点分裂时随机选择 k 个特征进行最优划分。
   - 树生长至满足停止条件。
3. 得到森林 $\{h_1, h_2, \ldots, h_T\}$。

预测

- 对于新样本 x，将其输入到每棵树 h_t 中进行预测，得到预测结果 y_t。
- 对于分类任务，采用相对多数投票机制。

$$\hat{y} = \arg\max_c \sum_{t=1}^{T} \mathbf{I} h_t^k(x)$$

2.1 优势分析

随机森林相比单一决策树具有以下优势：

- **降低过拟合风险**：通过集成多个树的预测结果，减少了单一树的方差，提高了模型的泛化能力。
- **鲁棒性强**：对噪声和异常值具有较好的容忍性，不易受到单一样本的影响。
- **特征重要性评估**：可以通过计算每个特征在所有树中的平均贡献，评估特征的重要性。
- **并行化能力强**：每棵树的构建相互独立，可以并行训练，提高训练效率。

2.2 参数选择

随机森林的关键参数包括：

- **树的数量** T：通常设置为 100-500 棵。树的数量越多，模型越稳定，但训练时间也越长。
- **随机特征数** k：对于分类任务，通常设置为 $k = \log_2 d$。
- **最大深度**：控制每棵树的深度，防止树过度生长导致过拟合。
- **最小样本数**：节点分裂所需的最小样本数，控制树的生长。

按照前文所述的算法流程构造决策树和随机森林，在数据集上进行训练和验证，得到以下的数据分析结果：

3 决策树的分类性能

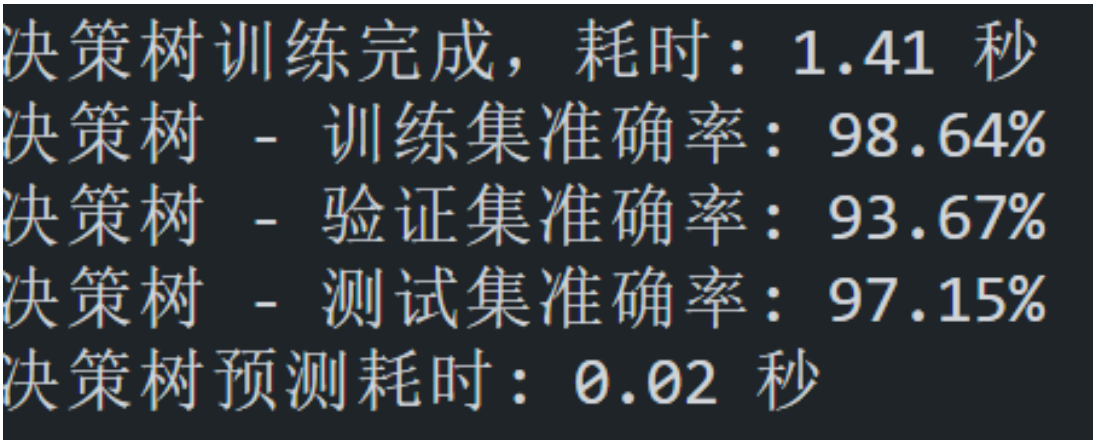

图 1: 决策树性能评估结果截图

数据集	准确率
训练集	98.64%
验证集	93.67%
测试集	97.15%

表 1: 决策树在 DNA 数据集上的分类准确率

从实验结果可以看出，单棵决策树在训练集上可能达到较高的准确率，但在测试集上的表现可能受到过拟合的影响，导致泛化能力不足。

4 随机森林的分类性能

构建包含 100 棵决策树的随机森林，在 DNA 数据集上进行评估。实验结果如下：

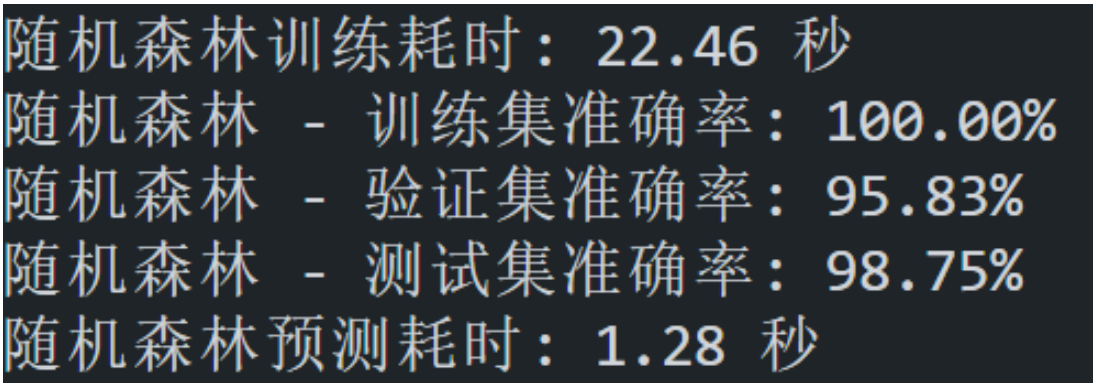

图 2: 随机森林性能评估结果截图

数据集	准确率
训练集	100%
验证集	95.83%
测试集	98.75%

表 2: 随机森林在 DNA 数据集上的分类准确率

5 性能对比

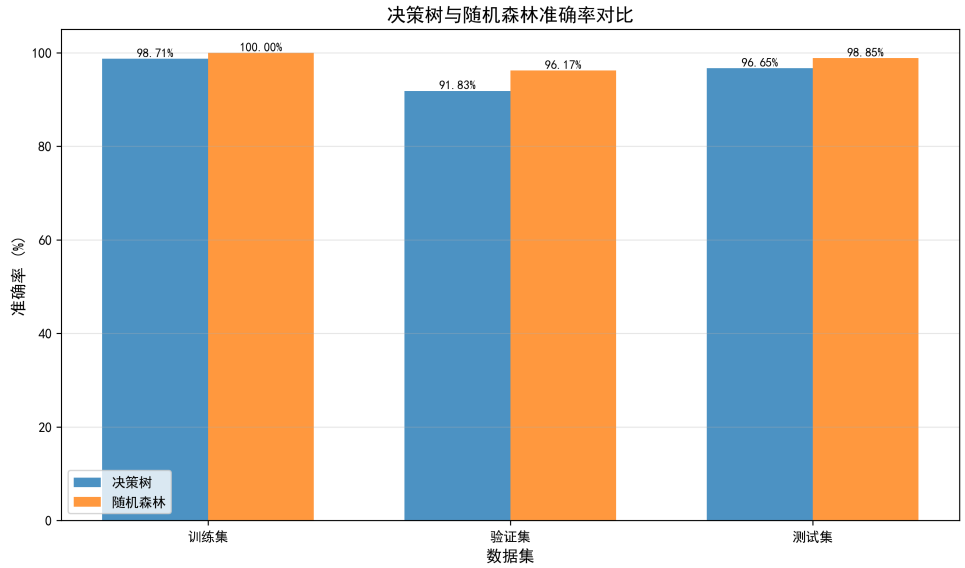

图 3: 决策树与随机森林在不同数据集上的准确率对比

算法	测试集准确率	训练时间（秒）	预测时间（秒）
决策树	97.15%	1.41	0.02
随机森林（100 trees）	98.75%	22.46	1.28

表 3: 综合性能对比

可以看出，随机森林在测试集上的分类准确率高于决策树（模型泛化能力强），但训练时间和预测时间明显长于决策树（训练多棵树，并进行投票预测）。

随机森林通过集成多棵树的预测结果，有效提高了模型的稳定性和泛化能力。在测试集上的表现明显优于单棵决策树。但树的数量并非越多越好，下面进行一个实验来进行说明。

6 树数量对随机森林性能的影响

实验中，分别构建包含 10、20、50、100、200 棵树的随机森林，并记录它们在测试集上的准确率。

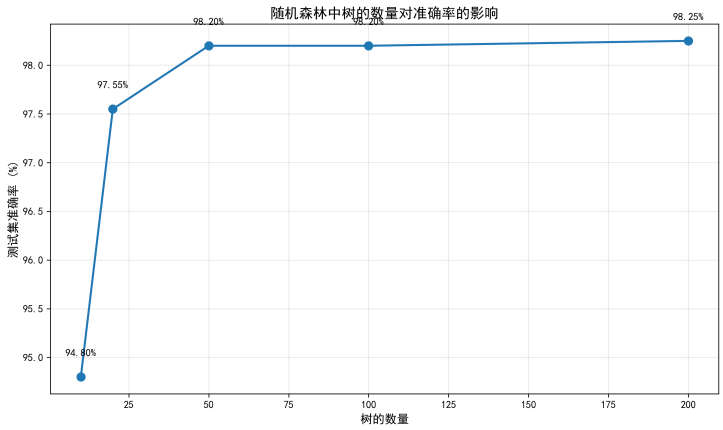

图 4: 随机森林中树的数量对测试集准确率的影响

实验结果表明，随着树的数量增加，随机森林的准确率逐渐提高并趋于稳定。当树的数量达到一定值后，继续增加树的数量对准确率的提升十分微弱，甚至会起到反作用，而且会增加训练时间和内存消耗。

7 特征重要性分析

随机森林的可解释性比较强。通过运行随机森林算法还可以评估各个特征对分类的重要性。通过计算每个特征在所有树中作为分支节点的平均次数，并归一化处理后转化成分数，得到下面的图像：

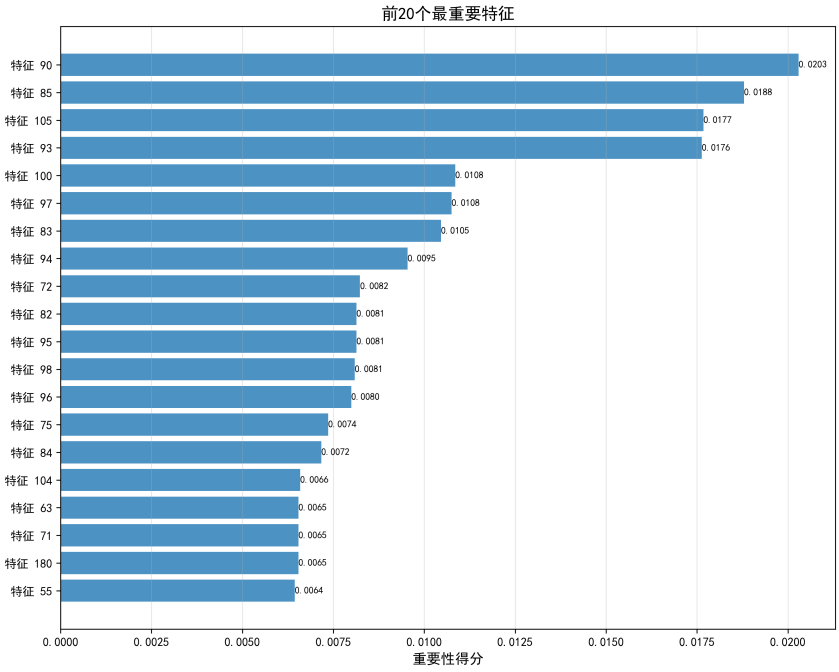

图 5: DNA 数据集中前 20 个最重要的特征

通过上面的分析，可以大致判断出对 DNA 分类贡献最大的特征，有助于理解数据的现实意义，提高模型的可解释性。

8 混淆矩阵分析

混淆矩阵可以清晰显示模型的分类性能，使用 matplotlib 库可以绘出随机森林在测试集上预测结果的混淆矩阵：

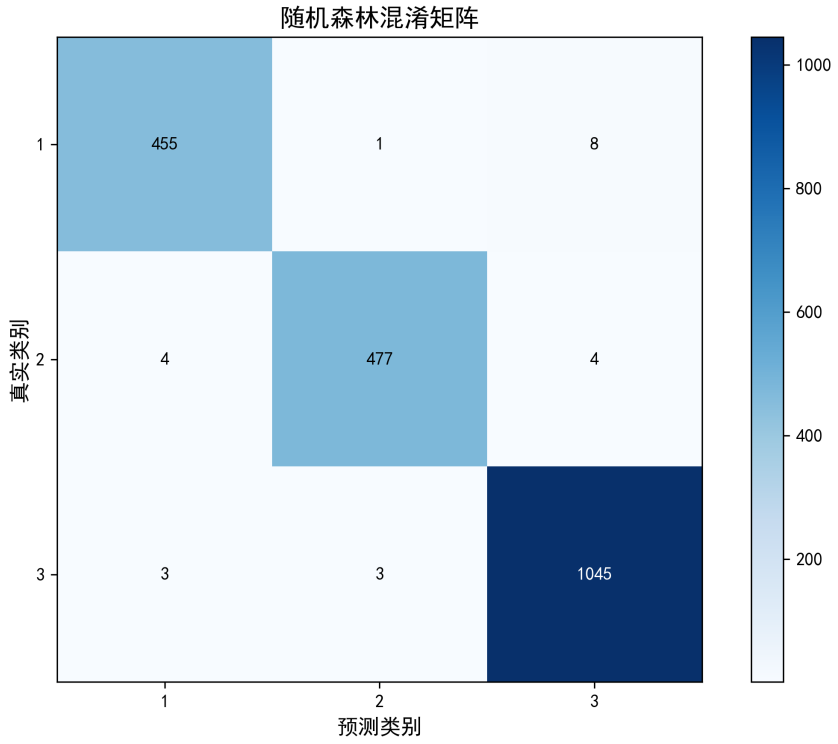

图 6: 随机森林在 DNA 测试集上的混淆矩阵

结果显示，大量预测结果位于矩阵对角线上，说明模型的分类性能很好。

四、 实验总结

通过此次实验，我更加深刻地理解了决策树和随机森林算法的原理，掌握了决策树的构建算法和其评价指标的计算方法，能够正确应用 python 对训练数据进行提取与分析，并通过多种方法对两种算法的性能进行评估和对比。

五、 代码

```python
import numpy as np
from collections import Counter
import time

def load_libsvm_data(filename):#加载数据
    """
    加载LIBSVM格式的数据
    参数:
        filename: 文件路径
    返回:
        X: 特征矩阵 (n_samples, n_features)
        y: 标签向量 (n_samples,)
```

```
    """
    X = []
    y = []
    max_feature_idx = 0

    with open(filename, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) == 0:
                continue

            # 第一个元素是标签
            label = int(float(parts[0]))
            y.append(label)

            # 解析特征 (格式: index:value)
            features = {}
            for item in parts[1:]:
                if ':' in item:
                    idx, value = item.split(':')
                    idx = int(idx)
                    value = float(value)
                    features[idx] = value
                    max_feature_idx = max(max_feature_idx, idx)

            X.append(features)

    # 将稀疏表示转换为密集矩阵
    X_dense = np.zeros((len(X), max_feature_idx))
    for i, features in enumerate(X):
        for idx, value in features.items():
            X_dense[i, idx - 1] = value  # LIBSVM的索引从1开始

    return X_dense, np.array(y)


class TreeNode:#树的节点
    def __init__(self, is_leaf=False, label=None, feature=None,
                 threshold=None, left=None, right=None,
                 samples=0, entropy=0):
        self.is_leaf = is_leaf # 是否为叶节点
        self.label = label # 叶节点的类别标签
        self.feature = feature # 用于分支的特征索引
        self.threshold = threshold  # 分支节点的信息增益（看走左还是右）
        self.left = left  # 左子树
        self.right = right # 右子树
        self.samples = samples # 样本数
        self.entropy = entropy # 信息熵
```

```
61
62
63  class DecisionTree:
64      def __init__(self, max_depth=10, min_samples_split=2,
65                   random_features=None):
66          """
67          参数:
68              max_depth: 最大深度
69              min_samples_split: 分裂所需的最小样本数
70              random_features: 每次分裂时随机选择的特征数（用于随机森林）
71          """
72          self.max_depth = max_depth
73          self.min_samples_split = min_samples_split
74          self.random_features = random_features
75          self.root = None
76          self.n_features = None
77
78      def calculate_entropy(self, y):
79          """计算信息熵"""
80          if len(y) == 0:
81              return 0
82          counter = Counter(y)
83          entropy = 0.0
84          for count in counter.values():
85              prob = count / len(y)
86              if prob > 0:
87                  entropy -= prob * np.log2(prob)
88          return entropy
89
90      def split_data(self, X, y, feature, threshold):
91          """根据特征和阈值分裂数据"""
92          left_mask = X[:, feature] <= threshold
93          right_mask = ~left_mask
94          return (X[left_mask], y[left_mask],
95                  X[right_mask], y[right_mask])
96
97      def find_best_split(self, X, y):#用信息增益
98          n_samples, n_features = X.shape
99
100         # 如果设置了随机特征数，则随机选择特征子集
101         if self.random_features is not None:
102             features_to_try = np.random.choice(
103                 n_features,
104                 min(self.random_features, n_features),
105                 replace=False)
106         else:
107             features_to_try = range(n_features)
108
```

```
        best_gain = -float('inf')
        best_feature = None
        best_threshold = None
        parent_entropy = self.calculate_entropy(y)

        for feature in features_to_try:
            # 获取该特征的所有唯一值作为候选阈值
            thresholds = np.unique(X[:, feature])

            for threshold in thresholds:
                # 分裂数据
                X_left, y_left, X_right, y_right = self.split_data(
                    X, y, feature, threshold)

                if len(y_left) == 0 or len(y_right) == 0:
                    continue

                # 计算信息增益
                n_left, n_right = len(y_left), len(y_right)
                entropy_left = self.calculate_entropy(y_left)
                entropy_right = self.calculate_entropy(y_right)
                weighted_entropy = (n_left * entropy_left +
                                    n_right * entropy_right) / n_samples
                gain = parent_entropy - weighted_entropy

                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold

        return best_feature, best_threshold

    def build_tree(self, X, y, depth=0):
        """递归构建决策树"""
        n_samples = len(y)
        n_classes = len(np.unique(y))

        if (depth >= self.max_depth or
            n_samples < self.min_samples_split or
            n_classes == 1):#到最大深度或者每个节点上的样本同属一类（不用再分类）或能作为分支的特征数减少到阈值
            # 创建叶节点
            label = Counter(y).most_common(1)[0][0]
            return TreeNode(is_leaf=True, label=label,
                            samples=n_samples,
                            entropy=self.calculate_entropy(y))

        # 寻找最佳分裂
```

```
        best_feature, best_threshold = self.find_best_split(X, y)

        if best_feature is None:
            # 无法分裂，创建叶节点
            label = Counter(y).most_common(1)[0][0]
            return TreeNode(is_leaf=True, label=label,
                            samples=n_samples,
                            entropy=self.calculate_entropy(y))

        # 分裂数据
        X_left, y_left, X_right, y_right = self.split_data(
            X, y, best_feature, best_threshold)

        # 递归构建左右子树
        left = self.build_tree(X_left, y_left, depth + 1)
        right = self.build_tree(X_right, y_right, depth + 1)

        return TreeNode(is_leaf=False, feature=best_feature,
                        threshold=best_threshold,
                        left=left, right=right, samples=n_samples,
                        entropy=self.calculate_entropy(y))

    def fit(self, X, y):
        """训练决策树"""
        self.n_features = X.shape[1]
        self.root = self.build_tree(X, y)
        return self

    def predict_single(self, x, node):
        """预测单个样本"""
        if node.is_leaf:
            return node.label

        if x[node.feature] <= node.threshold:
            return self.predict_single(x, node.left)
        else:
            return self.predict_single(x, node.right)

    def predict(self, X):
        return np.array([self.predict_single(x, self.root)
                         for x in X])


class RandomForest:
    def __init__(self, n_trees=100, max_depth=15,
                 min_samples_split=2, max_features='log2'):
        """
        参数:
```

```
204             n_trees: 树的数量
205             max_depth: 每棵树的最大深度
206             min_samples_split: 分裂所需的最小样本数
207             max_features: 每次分裂时随机选择的特征数
208         """
209         self.n_trees = n_trees
210         self.max_depth = max_depth
211         self.min_samples_split = min_samples_split
212         self.max_features = max_features
213         self.trees = []
214         self.feature_importances_ = None
215 
216     def bootstrap_sample(self, X, y):#抽样
217         n_samples = X.shape[0]
218         indices = np.random.choice(n_samples, n_samples,
219                                    replace=True)
220         return X[indices], y[indices]
221 
222     def get_n_features(self, n_features_total):#用log2
223         if self.max_features == 'sqrt':
224             return int(np.sqrt(n_features_total))
225         elif self.max_features == 'log2':
226             return int(np.log2(n_features_total))
227         elif isinstance(self.max_features, int):
228             return self.max_features
229         elif isinstance(self.max_features, float):
230             return int(self.max_features * n_features_total)
231         else:
232             return n_features_total
233 
234     def fit(self, X, y):
235         self.trees = []
236         n_features = X.shape[1]
237         random_features = self.get_n_features(n_features)
238 
239         print(f"开始训练随机森林（{self.n_trees}棵树，"
240               f"每次分裂选择{random_features}个特征）...")
241 
242         for i in range(self.n_trees):
243             # 有放回抽样
244             X_sample, y_sample = self.bootstrap_sample(X, y)
245 
246             #训练若干棵树
247             tree = DecisionTree(
248                 max_depth=self.max_depth,
249                 min_samples_split=self.min_samples_split,
250                 random_features=random_features)
251             tree.fit(X_sample, y_sample)
```

```
252             self.trees.append(tree)
253
254             if (i + 1) % 10 == 0:
255                 print(f"已完成 {i + 1}/{self.n_trees} 棵树")
256
257         print("随机森林训练完成！")
258         return self
259
260     def predict(self, X):#相对多数投票法预测
261         tree_predictions = np.array([tree.predict(X)
262                                      for tree in self.trees])
263
264         predictions = []
265         for i in range(X.shape[0]):
266             votes = tree_predictions[:, i]
267             most_common = Counter(votes).most_common(1)[0][0]
268             predictions.append(most_common)
269
270         return np.array(predictions)
271
272     def calculate_feature_importance(self, X, y):
273         # 特征在森林中被用作分支的次数越多就越重要
274         feature_counts = np.zeros(X.shape[1])
275
276         def count_features(node):
277             if node.is_leaf:
278                 return
279             feature_counts[node.feature] += 1
280             count_features(node.left)
281             count_features(node.right)
282
283         for tree in self.trees:
284             count_features(tree.root)
285
286         # 归一化
287         self.feature_importances_ = (feature_counts /
288                                      feature_counts.sum())
289         return self.feature_importances_
290
291
292 def calculate_accuracy(y_true, y_pred):#算准确率
293     return np.mean(y_true == y_pred) * 100
294
295
296 def create_confusion_matrix(y_true, y_pred):#混淆矩阵
297     classes = np.unique(y_true)
298     n_classes = len(classes)
299     matrix = np.zeros((n_classes, n_classes), dtype=int)
```

```
300
301     for true_label, pred_label in zip(y_true, y_pred):
302         true_idx = np.where(classes == true_label)[0][0]
303         pred_idx = np.where(classes == pred_label)[0][0]
304         matrix[true_idx, pred_idx] += 1
305
306     return matrix, classes
307
308 #加载数据集
309 X_train, y_train = load_libsvm_data('dna.scale.tr.txt')
310 X_val, y_val = load_libsvm_data('dna.scale.val.txt')
311 X_test, y_test = load_libsvm_data('dna.scale.txt')
312
313 #训练决策树
314 print("\n[2] 训练决策树...")
315 start_time = time.time()
316 dt = DecisionTree(max_depth=10, min_samples_split=2)
317 dt.fit(X_train, y_train)
318 dt_train_time = time.time() - start_time
319 print(f"决策树训练完成，耗时: {dt_train_time:.2f} 秒")
320
321 #预测
322 start_time = time.time()
323 dt_train_pred = dt.predict(X_train)
324 dt_val_pred = dt.predict(X_val)
325 dt_test_pred = dt.predict(X_test)
326 dt_pred_time = time.time() - start_time
327
328 dt_train_acc = calculate_accuracy(y_train, dt_train_pred)
329 dt_val_acc = calculate_accuracy(y_val, dt_val_pred)
330 dt_test_acc = calculate_accuracy(y_test, dt_test_pred)
331
332 print(f"决策树 - 训练集准确率: {dt_train_acc:.2f}%")
333 print(f"决策树 - 验证集准确率: {dt_val_acc:.2f}%")
334 print(f"决策树 - 测试集准确率: {dt_test_acc:.2f}%")
335 print(f"决策树预测耗时: {dt_pred_time:.2f} 秒")
336
337 #训练随机森林
338 print("\n[3] 训练随机森林...")
339 start_time = time.time()
340 rf = RandomForest(n_trees=100, max_depth=15, min_samples_split=2, max_features='sqrt')
341 rf.fit(X_train, y_train)
342 rf_train_time = time.time() - start_time
343 print(f"随机森林训练耗时: {rf_train_time:.2f} 秒")
344
345 #预测
346 start_time = time.time()
347 rf_train_pred = rf.predict(X_train)
```

```
rf_val_pred = rf.predict(X_val)
rf_test_pred = rf.predict(X_test)
rf_pred_time = time.time() - start_time

rf_train_acc = calculate_accuracy(y_train, rf_train_pred)
rf_val_acc = calculate_accuracy(y_val, rf_val_pred)
rf_test_acc = calculate_accuracy(y_test, rf_test_pred)

print(f"随机森林 - 训练集准确率: {rf_train_acc:.2f}%")
print(f"随机森林 - 验证集准确率: {rf_val_acc:.2f}%")
print(f"随机森林 - 测试集准确率: {rf_test_acc:.2f}%")
print(f"随机森林预测耗时: {rf_pred_time:.2f} 秒")
```