实验五：决策树算法 ID3

一、实验目的

编程实现决策树算法 ID3，理解算法原理；利用给定数据训练构造决策树，并对测试数据进行分类，输出分类准确率。

二、实验原理

ID3 算法核心是以信息增益度量属性选择，选择分裂后信息增益最大的属性进行分裂。相关概念如下：

- **熵（Entropy）**：设 D 为用类别对训练元组进行的划分，D 的熵表示为：

  $$H(D) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

  其中，(pi)表示第i个类别在整个训练元组中出现的概率。熵表示 D 中元组的类标号所需要的平均信息量。

- **期望信息**：假设将训练元组 D 按属性 A 进行划分，A 对 D 划分的期望信息为：

  $$H_A(D) = \sum_{v=1}^{V} \frac{|D_v|}{|D|} H(D_v)$$

  其中，(Dv)表示按属性 A 的第v个取值划分后的子集，V为属性 A 的取值个数。

- **信息增益**：信息增益为熵与期望信息的差值，即：

  $$\text{gain}(A) = H(D) - H_A(D)$$

  ID3 算法在每次分裂时，计算每个属性的信息增益，选择信息增益最大的属性进行分裂。对于连续值特征属性，先将 D 中元素按特征属性排序，每两个相邻元素的中间点作为潜在分裂点，计算每个潜在分裂点分裂后的期望信息，具有最小期望信息的点为最佳分裂点，其信息期望作为此属性的信息期望。

代码如下：

```python
import numpy as np

class DecisionTreeID3:
    def __init__(self):
        self.tree = {}

    def calculate_entropy(self, labels):
        label_counts = np.bincount(labels.astype(int))
        probabilities = label_counts / len(labels)
        entropy = -np.sum([p * np.log2(p) for p in probabilities if p > 0])
        return entropy

```

```python
    def get_continuous_split(self, data, feature_index):
        sorted_data = data[data[:, feature_index].argsort()]
        features = sorted_data[:, feature_index]
        potential_splits = [(features[i] + features[i+1])/2 for i in
range(len(features)-1)]
        best_split = None
        min_entropy = float('inf')
        for split in potential_splits:
            left = sorted_data[sorted_data[:, feature_index] <= split]
            right = sorted_data[sorted_data[:, feature_index] > split]
            if len(left) == 0 or len(right) == 0:
                continue
            entropy = (len(left)/len(sorted_data)) *
self.calculate_entropy(left[:, -1]) + (len(right)/len(sorted_data)) *
self.calculate_entropy(right[:, -1])
            if entropy < min_entropy:
                min_entropy = entropy
                best_split = split
        return best_split, min_entropy

    def choose_best_feature(self, data):
        num_features = data.shape[1] - 1
        base_entropy = self.calculate_entropy(data[:, -1])
        best_info_gain = 0.0
        best_feature = -1
        best_split = None
        for feature_index in range(num_features):
            feature_values = data[:, feature_index]
            if len(np.unique(feature_values)) > 10:
                split, entropy = self.get_continuous_split(data, feature_index)
                if split is None:
                    continue
                info_gain = base_entropy - entropy
            else:
                unique_values = np.unique(feature_values)
                entropy = 0.0
                for value in unique_values:
                    sub_data = data[data[:, feature_index] == value]
                    prob = len(sub_data) / len(data)
                    entropy += prob * self.calculate_entropy(sub_data[:, -1])
                info_gain = base_entropy - entropy
            if info_gain > best_info_gain:
                best_info_gain = info_gain
                best_feature = feature_index
                if len(np.unique(feature_values)) > 10:
                    best_split = split
                else:
                    best_split = None
        return best_feature, best_split

    def majority_vote(self, labels):
        label_counts = np.bincount(labels.astype(int))
        return np.argmax(label_counts)

    def build_tree(self, data):
        labels = data[:, -1]
```

```python
        if len(np.unique(labels)) == 1:
            return int(labels[0])
        if data.shape[1] == 1:
            return self.majority_vote(labels)
        best_feature, best_split = self.choose_best_feature(data)
        tree = {'feature': best_feature, 'split': best_split}
        if best_split is not None:
            left_data = data[data[:, best_feature] <= best_split]
            right_data = data[data[:, best_feature] > best_split]
            tree['left'] = self.build_tree(left_data)
            tree['right'] = self.build_tree(right_data)
        else:
            unique_values = np.unique(data[:, best_feature])
            for value in unique_values:
                sub_data = data[data[:, best_feature] == value]
                tree[value] = self.build_tree(sub_data)
        return tree

    def fit(self, X, y):
        data = np.hstack((X, y.reshape(-1, 1)))
        self.tree = self.build_tree(data)

    def predict_sample(self, sample, tree):
        if isinstance(tree, int):
            return tree
        feature = tree['feature']
        split = tree['split']
        if split is not None:
            if sample[feature] <= split:
                return self.predict_sample(sample, tree['left'])
            else:
                return self.predict_sample(sample, tree['right'])
        else:
            value = sample[feature]
            return self.predict_sample(sample, tree[value])

    def predict(self, X):
        predictions = []
        for sample in X:
            pred = self.predict_sample(sample, self.tree)
            predictions.append(pred)
        return np.array(predictions)


def load_data(file_path):
    data = np.loadtxt(file_path)
    X = data[:, :-1]
    y = data[:, -1]
    return X, y


if __name__ == "__main__":
    X_train, y_train = load_data('traindata.txt')
    X_test, y_test = load_data('testdata.txt')

    clf = DecisionTreeID3()
```

```
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)
    accuracy = np.mean(y_pred == y_test)
    print(f"分类准确率: {accuracy * 100:.2f}%")

    print("决策树结构:")
    print(clf.tree)
```