重 庆 大 学

学 生 实 验 报 告

实验课程名称　　　　　人工智能导论

开课实验室　　　　　　DS1502

学　　　　院 软件学院 年级 2021 专业班 软工 X 班

学 生 姓 名 XXX 学　号 2021XXXX

开 课 时 间 2023 至 2024 学年第 1 学期

总 成 绩	
教师签名	

大数据与软件学院制

《人工智能导论》实验报告

开课实验室：DS1502 **2023 年 12 月 9 日**

<table>
<tr><td>学院</td><td>大数据与软件学院</td><td>年级、专业、班</td><td colspan="2">21 软件工程 X 班</td><td>姓名</td><td>XXX</td><td>成绩</td><td></td></tr>
<tr><td>课程名称</td><td colspan="2">人工智能导论</td><td>实验项目名　　称</td><td colspan="3">基于决策树的企鹅分类</td><td>指导教师</td><td>XX</td></tr>
<tr><td>教师评语</td><td colspan="8">教师签名：
2023 年　月　日</td></tr>
</table>

一、 实验目的

实验目的是通过决策树对企鹅进行分类，并评估模型的性能，以了解模型在解决企鹅分类问题上的有效性。

二、实验内容

① 数据准备： 使用了一个包含企鹅相关特征的数据集，例如岛屿、喙长度、喙深度、鳍长度、体重、性别和年龄。

② 决策树构建： 利用数据集和标签，以及标签属性（是分类属性还是连续属性）创建了一个决策树。

③ 数据集划分： 将数据集分为训练集和测试集，其中训练集用于构建决策树，测试集用于评估模型的性能。

④ 模型训练： 使用训练集对决策树进行训练。

⑤ 模型测试： 使用测试集进行分类预测，评估模型在新数据上的性能。

⑥ 性能评估： 计算分类准确率，并生成混淆矩阵以更详细地了解模型的性能。

⑦ 可视化： 绘制混淆矩阵的热力图，以及决策树的结构图，以便更直观地理解模型的工作方式。

三、使用仪器、材料

1. 操作系统：Windows 11
2. 开发设备：Lenovo Legion R9000P2021H
3. 开发平台：PyCharm 2023.1

四、实验过程原始记录(数据、图表、计算等)：

（一）源代码

```python
from math import log
import pandas as pd
import numpy as np
import operator
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import copy

# 获取决策树叶子节点数量
3 用法
def get_num_leafs(tree):
    num_leafs = 0
    first_str = next(iter(tree))
    second_dict = tree[first_str]
    for key in second_dict.keys():
        if type(second_dict[key]) is dict:
            num_leafs += get_num_leafs(second_dict[key])
        else:
            num_leafs += 1
    return num_leafs

# 获取决策树深度
2 用法
def get_tree_depth(tree):
    max_depth = 0
    first_str = next(iter(tree))
    second_dict = tree[first_str]
    for key in second_dict.keys():
        if type(second_dict[key]) is dict:
            this_depth = 1 + get_tree_depth(second_dict[key])
        else:
            this_depth = 1
        if this_depth > max_depth:
            max_depth = this_depth
    return max_depth

# 绘制决策树节点
2 用法
def plot_node(node_txt, center_pt, parent_pt, node_type):
    arrow_args = dict(arrowstyle="<-")
    create_plot.ax1.annotate(
        node_txt,
        xy=parent_pt,
        xycoords='axes fraction',
        xytext=center_pt,
        textcoords='axes fraction',
        va="center",
        ha="center",
        bbox=node_type,
        arrowprops=arrow_args
    )

```

```python
# 在父节点和子节点之间绘制文本
2 用法
def plot_mid_text(cntr_pt, parent_pt, txt_string):
    x_mid = (parent_pt[0] - cntr_pt[0]) / 2.0 + cntr_pt[0]
    y_mid = (parent_pt[1] - cntr_pt[1]) / 2.0 + cntr_pt[1]
    create_plot.ax1.text(x_mid, y_mid, txt_string, va="center", ha="center", rotation=30)

# 绘制决策树
23 用法
def plot_tree(tree, parent_pt, node_txt):
    decision_node = dict(boxstyle="sawtooth", fc="0.8")
    leaf_node = dict(boxstyle="round4", fc="0.8")
    num_leafs = get_num_leafs(tree)
    first_str = next(iter(tree))
    cntr_pt = (plot_tree.x_off + (1.0 + float(num_leafs)) / 2.0 / plot_tree.total_w,
               plot_tree.y_off)
    plot_mid_text(cntr_pt, parent_pt, node_txt)
    plot_node(first_str, cntr_pt, parent_pt, decision_node)
    second_dict = tree[first_str]
    plot_tree.y_off = plot_tree.y_off - 1.0 / plot_tree.total_d
    for key in second_dict.keys():
        if type(second_dict[key]).__name__ == 'dict':
            plot_tree(second_dict[key], cntr_pt, str(key))
        else:
            plot_tree.x_off = plot_tree.x_off + 1.0 / plot_tree.total_w
            plot_node(second_dict[key], (plot_tree.x_off, plot_tree.y_off), cntr_pt, leaf_node)
            plot_mid_text((plot_tree.x_off, plot_tree.y_off), cntr_pt, str(key))
    plot_tree.y_off = plot_tree.y_off + 1.0 / plot_tree.total_d

# 创建决策树绘图
4 用法
def create_plot(tree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks=[], yticks=[])
    create_plot.ax1 = plt.subplot(111, frameon=False, **axprops)
    plot_tree.total_w = float(get_num_leafs(tree))
    plot_tree.total_d = float(get_tree_depth(tree))
    plot_tree.x_off = -0.5 / plot_tree.total_w
    plot_tree.y_off = 1.0
    plot_tree(tree, (0.5, 1.0), '')
    plt.show()

```

```python
# 将字符串标签转换为数值
3 用法
def transition(x):
    if x == data['Species'].unique()[0]:
        return 0
    if x == data['Species'].unique()[1]:
        return 1
    if x == data['Species'].unique()[2]:
        return 2
    if x == data['Island'].unique()[0]:
        return 0
    if x == data['Island'].unique()[1]:
        return 1
    if x == data['Island'].unique()[2]:
        return 2
    if x == data['Sex'].unique()[0]:
        return 0
    if x == data['Sex'].unique()[1]:
        return 1
    if x == data['Sex'].unique()[2]:
        return -1.0

```

```python
# 创建数据集
1 个用法
def create_data_set():
    global data
    data = data[[
        'Island',
        'Culmen Length (mm)',
        'Culmen Depth (mm)',
        'Flipper Length (mm)',
        'Body Mass (g)',
        'Sex',
        'Age',
        'Species',
    ]]
    data = data.fillna(-1)
    data['Species'] = data['Species'].apply(transition)
    data['Island'] = data['Island'].apply(transition)
    data['Sex'] = data['Sex'].apply(transition)
    data_set = []
    for i in range(344):
        data_set.append(list(data.iloc[i, :]))
    labels = [
        'Island', 'Culmen Length (mm)', 'Culmen Depth (mm)',
        'Flipper Length (mm)', 'Body Mass (g)', 'Sex', 'Age'
    ]
    return data_set, labels

# 计算信息熵
4 用法
def calc_ent(data_set):
    num_entries = len(data_set)
    label_counts = {}
    for feat_vec in data_set:
        current_label = feat_vec[-1]
        label_counts[current_label] = label_counts.get(current_label, 0) + 1
    info_ent = 0.0
    for key in label_counts:
        prob = float(label_counts[key]) / num_entries
        info_ent -= prob * log(prob, 2)
    return info_ent

# 根据特征和特征值划分数据集
2 用法
def split_data_set(data_set, axis, value):
    ret_data_set = []
    for feat_vec in data_set:
        if feat_vec[axis] == value:
            reduced_feat_vec = feat_vec[:axis]
            reduced_feat_vec.extend(feat_vec[axis + 1:])
            ret_data_set.append(reduced_feat_vec)
    return ret_data_set

```

```python
# 根据数值型特征和划分值划分数据集
4 用法
def split_data_set_c(data_set, axis, value, lor_r='L'):
    ret_data_set = []
    if lor_r == 'L':
        for feat_vec in data_set:
            if float(feat_vec[axis]) < value:
                ret_data_set.append(feat_vec)
    else:
        for feat_vec in data_set:
            if float(feat_vec[axis]) > value:
                ret_data_set.append(feat_vec)
    return ret_data_set

```

```python
# 选择最佳划分特征
1 个用法
def choose_best_feature_to_split(data_set, label_property):
    num_features = len(label_property)
    base_entropy = calc_ent(data_set)
    best_info_gain = 0.0
    best_feature = -1
    best_part_value = None
    for i in range(num_features):
        feat_list = [example[i] for example in data_set]
        unique_vals = set(feat_list)
        new_entropy = 0.0
        best_part_value_i = None
        if label_property[i] == 0:
            for value in unique_vals:
                sub_data_set = split_data_set(data_set, i, value)
                prob = len(sub_data_set) / float(len(data_set))
                new_entropy += prob * calc_ent(sub_data_set)
        else:
            sorted_unique_vals = list(unique_vals)
            sorted_unique_vals.sort()
            min_entropy = float('inf')
            for j in range(len(sorted_unique_vals) - 1):
                part_value = (float(sorted_unique_vals[j]) +
                              float(sorted_unique_vals[j + 1])) / 2
                data_set_left = split_data_set_c(data_set, i, part_value, 'L')
                data_set_right = split_data_set_c(data_set, i, part_value, 'R')
                prob_left = len(data_set_left) / float(len(data_set))
                prob_right = len(data_set_right) / float(len(data_set))
                entropy = prob_left * calc_ent(data_set_left) + prob_right * calc_ent(data_set_right)
                if entropy < min_entropy:
                    min_entropy = entropy
                    best_part_value_i = part_value
            new_entropy = min_entropy
        info_gain = base_entropy - new_entropy
        if info_gain > best_info_gain:
            best_info_gain = info_gain
            best_feature = i
            best_part_value = best_part_value_i
    return best_feature, best_part_value

# 统计类别标签出现频率，返回出现频率最高的标签
2 用法
def majority_count(class_list):
    class_count = {}
    for vote in class_list:
        class_count[vote] = class_count.get(vote, 0) + 1
    sorted_class_count = sorted(class_count.items(),
                                key=operator.itemgetter(1),
                                reverse=True)
    return sorted_class_count[0][0]
```

```python
# 递归构建决策树
4 用法
def create_tree(data_set, labels, label_property):
    class_list = [example[-1] for example in data_set]
    if class_list.count(class_list[0]) == len(class_list):
        return class_list[0]
    if len(data_set[0]) == 1:
        return majority_count(class_list)
    best_feat, best_part_value = choose_best_feature_to_split(
        data_set, label_property)
    if best_feat == -1:
        return majority_count(class_list)
    if label_property[best_feat] == 0:
        best_feat_label = labels[best_feat]
        my_tree = {best_feat_label: {}}
        labels_new = copy.copy(labels)
        label_property_new = copy.copy(label_property)
        del labels_new[best_feat]
        del label_property_new[best_feat]
        feat_values = [example[best_feat] for example in data_set]
        unique_value = set(feat_values)
        for value in unique_value:
            sub_labels = labels_new[:]
            sub_label_property = label_property_new[:]
            my_tree[best_feat_label][value] = create_tree(
                split_data_set(data_set, best_feat, value), sub_labels,
                sub_label_property
            )
    else:
        best_feat_label = labels[best_feat] + '<' + str(best_part_value)
        my_tree = {best_feat_label: {}}
        sub_labels = labels[:]
        sub_label_property = label_property[:]
        value_left = 'Yes'
        my_tree[best_feat_label][value_left] = create_tree(
            split_data_set_c(data_set, best_feat, best_part_value, 'L'), sub_labels,
            sub_label_property
        )
        value_right = 'No'
        my_tree[best_feat_label][value_right] = create_tree(
            split_data_set_c(data_set, best_feat, best_part_value, 'R'), sub_labels,
            sub_label_property
        )
    return my_tree
```

```python
# 对测试样本进行分类
4 用法
def classify(input_tree, feat_labels, feat_label_properties, test_vec):
    first_str = list(input_tree.keys())[0]
    first_label = first_str
    less_index = str(first_str).find('<')
    if less_index > -1:
        first_label = str(first_str)[:less_index]
    second_dict = input_tree[first_str]
    feat_index = feat_labels.index(first_label)
    class_label = None
    for key in second_dict.keys():
        if feat_label_properties[feat_index] == 0:
            if test_vec[feat_index] == key:
                if type(second_dict[key]).__name__ == 'dict':
                    class_label = classify(second_dict[key], feat_labels,
                                           feat_label_properties, test_vec)
                else:
                    class_label = second_dict[key]
        else:
            part_value = float(str(first_str)[less_index + 1:])
            if test_vec[feat_index] < part_value:
                if type(second_dict['Yes']).__name__ == 'dict':
                    class_label = classify(second_dict['Yes'], feat_labels,
                                           feat_label_properties, test_vec)
                else:
                    class_label = second_dict['Yes']
            else:
                if type(second_dict['No']).__name__ == 'dict':
                    class_label = classify(second_dict['No'], feat_labels,
                                           feat_label_properties, test_vec)
                else:
                    class_label = second_dict['No']
    return class_label
```

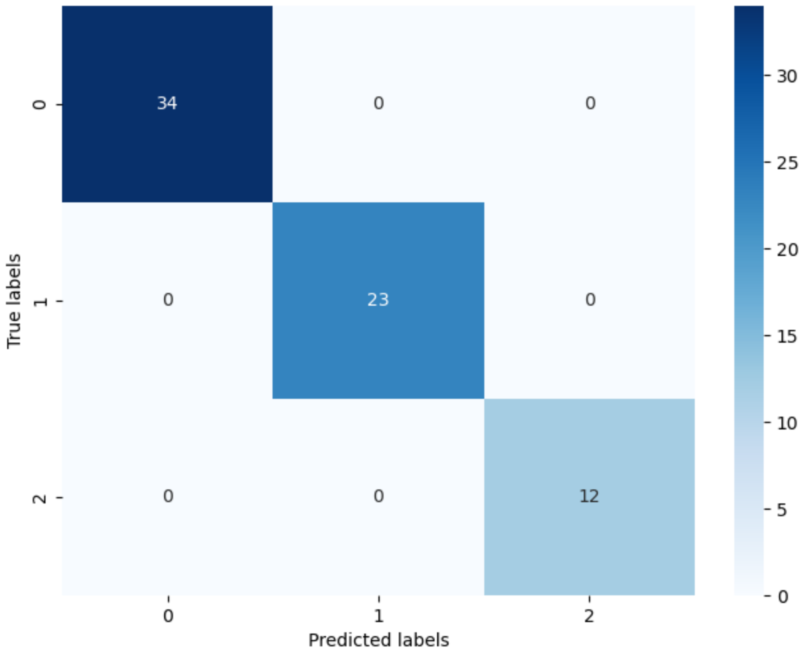
```python
# 读取数据集
data = pd.read_csv('penguins_data.csv')

# 特征名称
feature_name = [
    'Island', 'Culmen Length (mm)', 'Culmen Depth (mm)', 'Flipper Length (mm)', 'Body Mass (g)', 'Sex', 'Age'
]

# 创建数据集
data_set, labels = create_data_set()

# 定义一个标签属性列表，表示每个属性是分类属性（1）还是连续属性（0）
label_properties = [0, 1, 1, 1, 1, 0, 1]

# 使用数据集、标签和标签属性创建决策树
my_tree = create_tree(data_set, labels, label_properties)

# 从数据集中选择特征列
feature = data[[
    'Island', 'Culmen Length (mm)', 'Culmen Depth (mm)', 'Flipper Length (mm)', 'Body Mass (g)', 'Sex', 'Age'
]]

# 设置目标变量为物种类别
goal = data['Species']

# 划分数据集为训练集和测试集
x_train, x_test, y_train, y_test = train_test_split(feature,goal,test_size=0.2)

# 将测试集的特征转换为NumPy数组，并再次转换为列表
x_test = np.array(x_test)
x_test = x_test.tolist()

# 用训练好的决策树进行测试集的分类预测
test_pre = []
for i in range(len(x_test)):
    result = classify(my_tree, labels, label_properties, x_test[i])
    test_pre.append(result)

# 打印分类准确率
print("预测准确率为:", accuracy_score(y_test, test_pre))

# 生成混淆矩阵并用热力图进行可视化
confu_matrix = confusion_matrix(test_pre, y_test)
plt.figure(figsize=(8, 6))
sns.heatmap(confu_matrix, annot=True, cmap='Blues')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.show()

# 绘制决策树的图形
create_plot(my_tree)

# 打印决策树结构
print(my_tree)
```

（二）实现效果

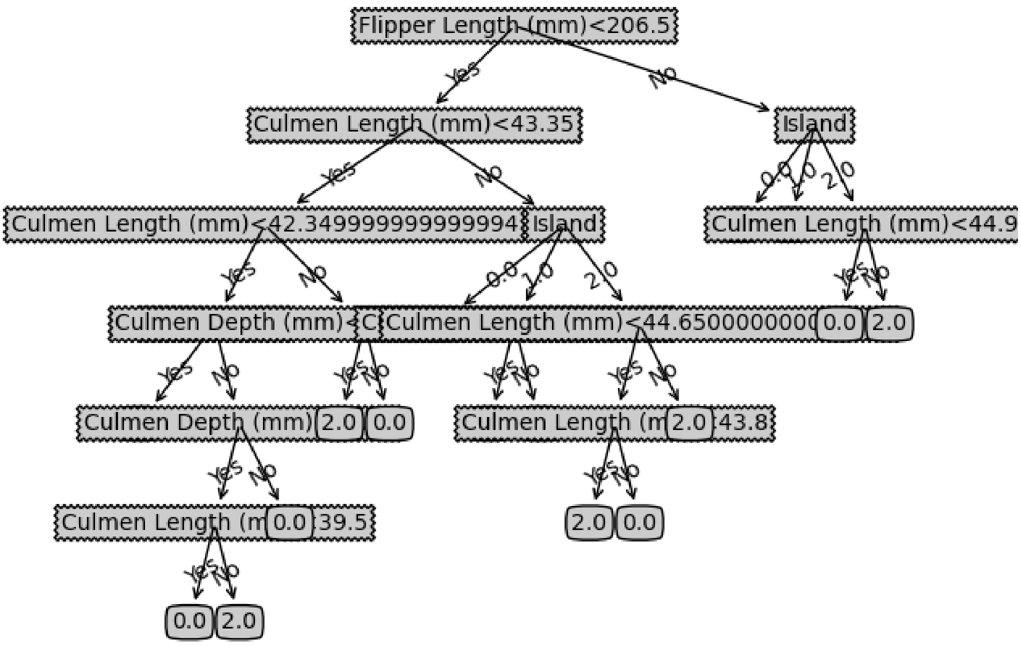
1. 预测准确率

```
预测准确率为: 1.0
```

2. 混淆矩阵



3. 决策树



4. 总结

本实验实现了一个基于决策树的企鹅分类模型，主要包括以下关键步骤：

（1） 决策树构建和可视化：

① 利用信息熵和信息增益选择最佳特征进行数据集划分，支持处理连续型特征。

② 通过递归方式绘制决策树的节点，使用矩形框表示决策节点和叶子节点，并通过箭头表示决策流向。

（2） 数据处理： 将原始数据集中的字符串标签转换为数值，处理缺失值。

（3） 模型训练和测试： 将数据集划分为训练集和测试集，使用训练集训练决策树模型。利用测试集评估模型的分类准确率。

（4） 混淆矩阵可视化：生成混淆矩阵，通过热力图进行可视化，直观展示模型在不同类别上的性能表现。

（5） 决策树结构展示：打印构建好的决策树结构，以更详细地了解模型的构建。

（6） 代码组织：通过函数进行模块化设计，提高了代码的可读性和可维护性。

总体而言，这个实验通过决策树对企鹅进行分类，并通过可视化和混淆矩阵的方式对模型进行了评估，展示了决策树在分类问题上的应用和性能评估过程。