



AACSB
ACCREDITED

CAMEA
中国高质量MBA教育认证



Iris 及 Income 数据集分类 实验报告

Experimental Report on Classification of IRIS and INCOME Datasets

学 院： 经济管理学院

课程名称： 机器学习

团队组号： 14

团队成员： 李佳颖 201805035

徐安琪 201805020

郑雅璇 201805059

周洁颖 201805017

任课教师： 李先能

完成日期 2021 年 5 月 12 日

目 录

1	算法介绍及优缺点	1
1.1	决策树	1
1.1.1	决策树介绍	1
1.1.2	决策树优缺点	2
1.2	神经网络	2
1.2.1	神经网络介绍	2
1.2.2	神经网络优缺点	3
2	iris 数据集	5
2.1	数据描述	5
2.1.1	特征统计	5
2.1.2	数据分布	6
2.2	决策树	8
2.2.1	代码实现	8
2.2.2	结果分析	23
2.3	神经网络	25
2.3.1	理论基础	25
2.3.2	代码实现	27
2.3.3	结果分析	33
3	income 数据集	35
3.1	数据描述	35
3.1.1	特征统计	35
3.1.2	数据分布	37
3.2	决策树	41
3.2.1	数据处理	41
3.2.2	代码实现	41
3.3.3	结果分析	42
3.3.4	随机森林	44
3.3	神经网络	48
3.3.1	理论基础	48
3.3.2	代码实现	49
3.3.3	结果分析	56

4	算法对比	59
4.1	准确率	59
4.2	运算时间	60
4.3	可解释性	61

1 算法介绍及优缺点

1.1 决策树

1.1.1 决策树介绍

决策树是一类常见的机器学习方法，用于样本的分类的任务。它可以认为是定义在特征空间与类空间的条件概率分布，实际上就是寻找最纯净的划分方法，实现把数据按照对应的类标签进行分类。决策树实质上就是一颗树，但其中的元素所代表的意思却与传统的树有所不同，它的每个非叶子节点表示一个属性，每个分枝代表一个输出，每个叶子节点代表一个类。树的最顶层为根节点。决策树是对单个特征进行处理，每一步寻找一个最优特征进行划分，直到叶节点。

决策树算法的核心就是划分属性选择。选择的目标就是决策树的分支结点尽可能属于同一类别，即结点的“纯度”(purity)越高越好。常见的属性选择标准有信息增益(information gain)、信息增益率(information gain ratio)、Gini 指数和卡方检验等。不同建树算法的属性划分标准可能不同，如常见的 ID3 算法和 C4.5 算法分别是基于信息增益和信息增益率构建决策树，而 CART 决策树使用“基尼指数”(Gini index)来选择划分属性。

建立决策模型之后，通过测试集采用一定的算法对树进行剪枝也是核心之一。决策树剪枝的基本策略有“预剪枝”(prepruning)和“后剪枝”(post-pruning)。预剪枝是指在决策树生产过程中，对每个结点在划分前先进行估计，若当前结点的划分不能带来决策树泛化性能的提升，则停止划分并将当前结点标记为叶结点。后剪枝则是先从训练集中生成一棵完整的决策树，然后自底向上地对非叶结点进行考察，若将该结点对应的子树替换为叶结点能带来决策树泛化性能的提升，则将该子树替换为叶结点。两种剪枝方法相比，预剪枝显著减少了决策树的训练和测试时间开销，但会给决策树带来欠拟合的风险；后剪枝的欠拟合风险很小，但训练和测试时间开销大很多。

构造决策树基本流程一般如下：

- ① 根据用户需求对数据训练集做进一步处理，开始为空树，之后根据每个节点的属性测试，将其划分到合适的类别；
- ② 通过数据训练集获取知识，通过自上而下的递归方式建立决策模型；
- ③ 通过最优化测量算法，计算出每个样本集合的可能划分并通过测试数据集采用特定的算法对数进行剪枝处理；
- ④ 通过后期的剪枝处理消除可能存在的异常，最终形成一颗完整的决策树。

1.1.2 决策树优缺点

1 优点

- ① 决策树可解释性强，树的结构可视化，容易提取出规则；
- ② 可以同时处理标称型和数值型数据，可以进行多分类划分；
- ③ 比较适合处理有缺失属性的样本；
- ④ 能够处理不相关的样本特征；
- ⑤ 模型的训练对数据量要求不高；
- ⑥ 运行速度比较快，在相对短的时间内能够对大型数据源做出可行且效果良好的结果。

2 缺点

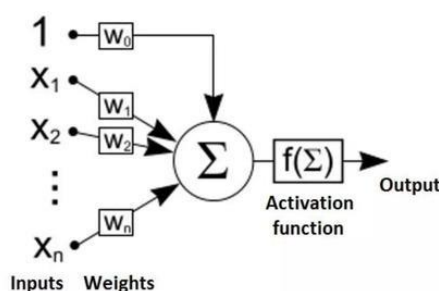
- ① 有模型过拟合的风险，尤其是对于生成复杂的决策树，导致泛化性能低；
- ② 容易忽略数据集中属性之间的关联性；
- ③ 进行属性划分时，不同的判定准则会带来不同的属性选择倾向。信息增益准则对可取值数目较多的属性有所偏好，信息增益率准则对可取值数目较少的属性有所偏好；
- ④ 决策树可能是不稳定的，因为在数据中的微小变化可能会导致完全不同的树生成；
- ⑤ 有些概念很难被决策树学习到，因为决策树很难清楚的表述那些概念，例如 XOR，奇偶或者复用器问题。

1.2 神经网络

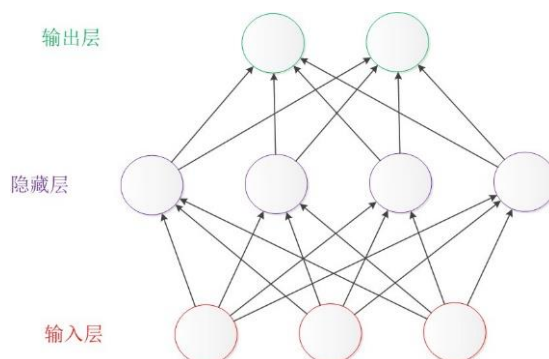
1.2.1 神经网络介绍

人工神经网络(Artificial Neural Networks, 简称为 ANNs)也简称为神经网络(NNs)，它是一种模仿动物神经网络行为特征，进行分布式并行信息处理的算法数学模型，由大量的节点(或称神经元)之间相互联接构成。每个节点代表一种特定的输出函数，称为激励函数(activation function)；每两个节点间的连接都代表一个对于通过该连接信号的加权值，称之为权重，相当于人工神经网络的记忆。网络的输出依据网络的连接方式、权重值和激励函数的不同而不同。而网络自身通常都是对自然界某种算法或者函数的逼近，也可能是对一种逻辑策略的表达。这种网络依靠系统的复杂程度，通过调整内部大量节点之间相互连接的关系，从而达到处理信息的目的。经过几十年的发展，神经网络理论在模式识别、自动控制、信号处理、辅助决策、人工智能等众多研究领域取得了广泛的成功。

神经网络分类算法由多层神经元结构组成，每一层神经元拥有输入和输出。作为神经网络的基本组成，每个神经元模型都包含两部分——线性计算部分和激活部分，如下图所示：



神经网络主要由输入层，隐藏层，输出层构成。其中输入层神经元可以接受多种类型的数据输入(文字、声音和图像等)；输出层输出分类或其他决策信息；隐藏层包含在输入层和输出层之间，可有多层，负责实现信息传递转换。当隐藏层只有一层时，该网络为两层神经网络，由于输入层未做任何变换，可以不看作单独的一层。实际中，网络输入层的每个神经元代表了一个特征，输出层个数代表了分类标签的个数，而隐藏层层数以及隐藏层神经元个数是由人工设定。



1.2.2 神经网络优缺点

1 优点

- ① 数据依赖，数据越多，神经网络训练效果越好，在现在的大数据时代，海量的数据使神经网络性能大大增强；
- ② 拟合能力强，可以逼近任何复杂的函数，而且神经网络的维度可以达到无穷维，这样其对数据的拟合能力是相当强大的。往往已有的传统的机器学习方法，在某一定程度上属于神经网络的特例。例如 SVM, Logistic regression 等均可以由神经网络来完成；

- ③ 神经网络由于包含了许多隐藏层，而隐藏层又具有许多隐藏结点，这样便使得神经网络的表达能力十分强大，这在贝叶斯理论中有很好的体现；
- ④ 容错能力强，局部的或者部分的神经元受到破坏后对全局的训练结果不会造成很大的影响，也就是说即使系统在受到局部损伤时还是可以正常工作的。

2 缺点

- ① 可解释性差，无法解释自己的推理过程和推理依据，这意味着你不知道神经网络如何以及为何会得出一定的输出；
- ② 开发时间长，虽然像 Keras 这样的库让神经网络的开发变得简单，但有时需要更多地控制算法的细节，尤其是试图解决机器学习中之前没人做过的难题时；
- ③ 计算时间长，通常在计算方面，神经网络比传统算法更昂贵。先进的深度学习算法，若想成功完成训练，可能需要几周的时间。而大多数传统机器学习只需花费少于几分钟，几个小时或者几天。

2 iris 数据集

2.1 数据描述

2.1.1 特征统计

1 特征统计

我们导入 Iris 数据集，查看其基本信息。

代码：

```
1. import pandas as pdd # abc
2. dataSet = pdd.read_csv(r'C:\Users\86176\Downloads\Video\iris.txt',header
=None)
3. dataSet.columns=['sepal_length','sepal+width_cm','petal_length_cm','petal
l_width_cm','class']
4. dataSet.head()
```

输出结果：

	sepal_length	sepal+width_cm	petal_length_cm	petal_width_cm	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

2 统计学特征

输出数据集的实例数、平均值、标准差、最大最小值、百分位点值：

代码：

```
1. dataSet.describe()
```

输出结果：

	sepal_length	sepal+width_cm	petal_length_cm	petal_width_cm
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

2.1.2 数据分布

1 数据是否平衡

Iris 数据集中有三个鸢尾花种类，统计各种类个数：

代码：

```
1. iris["Species"].value_counts()
```

输出结果：

setosa 50

virginica 50

versicolor 50

Name: Species, dtype: int64

可知各种类均有 50 个，数据平衡。

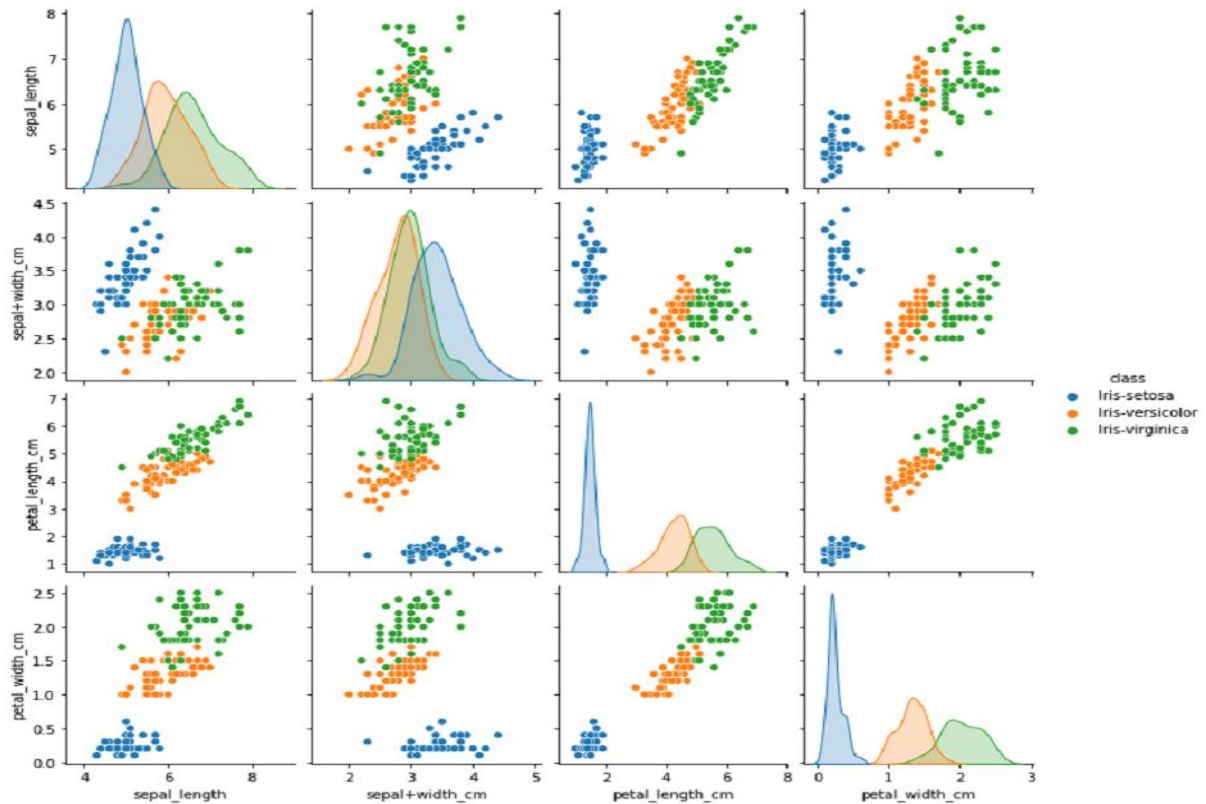
2 特征的散点图

绘制数据集两两特征的散点图：

代码：

```
1. %matplotlib inline
2. import matplotlib.pyplot as plt
3. import seaborn as sea
4.
5. sea.pairplot(dataSet.dropna(), hue='class')
```

输出结果：



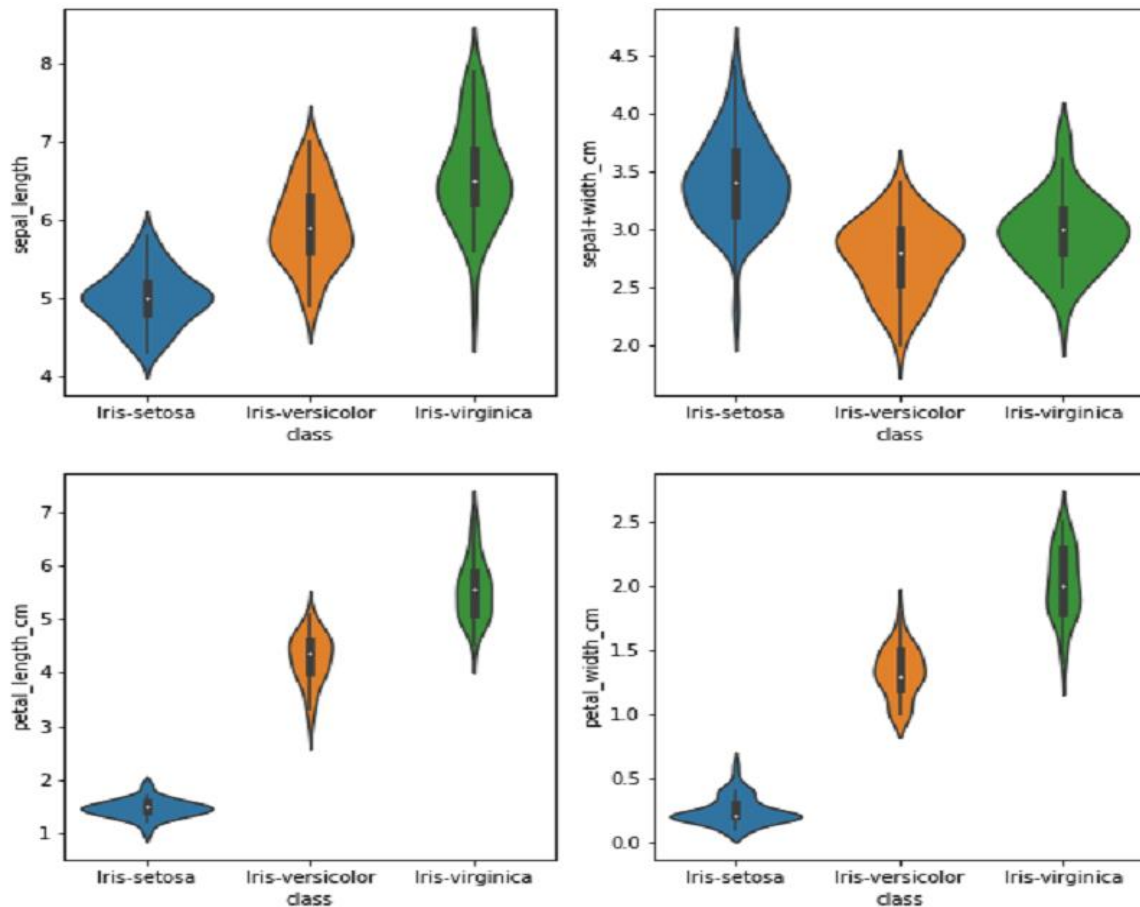
3 特征值分布图

更密集的数据区域更宽，更稀疏的数据区域更窄，根据此图可以更直观地看出不同种类鸢尾花的特征。

代码：

```
1. plt.figure(figsize = (10,10))
2. for index, row in enumerate(dataSet.columns):
3.     if row == 'class':
4.         continue
5.     plt.subplot(2, 2, index + 1)
6.     sea.violinplot(x = 'class', y = row, data = dataSet)
```

输出结果:



2.2 决策树

2.2.1 代码实现

1. 函数 calcShannonEnt()

函数名称 (参数)	calcShannonEnt (dataSet, labelIndex)
函数功能	计算信息熵
输入	数据集 dataSet, 类别索引 labelIndex
输出	信息熵 shannonEnt

```

1. def calcShannonEnt(dataSet, labelIndex):
2.     numEntries = 0 # 样本数(按权重计算)
3.     labelCounts = {}
4.     for featVec in dataSet: # 遍历每个样本
5.         if featVec[labelIndex] != 'N':

```

```

6.         weight = float(featVec[-2])
7.         numEntries += weight
8.         currentLabel = featVec[-1] # 当前样本的类别
9.         if currentLabel not in labelCounts.keys(): # 生成类别字典
10.             labelCounts[currentLabel] = 0
11.             labelCounts[currentLabel] += weight # 数据集的倒数第二个值用
            来标记样本权重
12.         shannonEnt = 0.0
13.         for key in labelCounts: # 计算信息熵
14.             prob = float(labelCounts[key]) / numEntries
15.             shannonEnt = shannonEnt - prob * log(prob, 2)
16.         return shannonEnt

```

2. 函数 splitDataSet()

函数名称（参数）	splitDataSet(dataSet, axis, value, LorR='N')
函数功能	划分训练集
输入	数据集 dataSet, 按第几个特征划分 axis, 划分特征的值 value, 离散属性 LorR='N'
输出	新数据集 retDataSet

```

1. def splitDataSet(dataSet, axis, value, LorR='N'):
2.     retDataSet = []
3.     featVec = []
4.     if LorR == 'N': # 离散属性
5.         for featVec in dataSet:
6.             if featVec[axis] == value:
7.                 reducedFeatVec = featVec[:axis]
8.                 reducedFeatVec.extend(featVec[axis + 1:])
9.                 retDataSet.append(reducedFeatVec)
10.    elif LorR == 'L':
11.        for featVec in dataSet:
12.            if featVec[axis] != 'N':
13.                if float(featVec[axis]) < value:
14.                    retDataSet.append(featVec)
15.    elif LorR == 'R':
16.        for featVec in dataSet:
17.            if featVec[axis] != 'N':
18.                if float(featVec[axis]) > value:
19.                    retDataSet.append(featVec)
20.    return retDataSet

```

3. 函数 splitDataSetWithNull()

函数名称（参数）	splitDataSetWithNull(dataSet, axis, value, LorR='N')
函数功能	划分训练集
输入	数据集 dataSet, 按第几个特征划分 axis, 划分特征的值 value, 离散属性 LorR='N'
输出	新数据集 retDataSet

```

1. def splitDataSetWithNull(dataSet, axis, value, LorR='N'):
2.     retDataSet = []
3.     nullDataSet = []
4.     featVec = []
5.     totalWeightV = calcTotalWeight(dataSet, axis, False) # 非空样本权重
6.     totalWeightSub = 0.0
7.     if LorR == 'N': # 离散属性
8.         for featVec in dataSet:
9.             if featVec[axis] == value:
10.                 reducedFeatVec = featVec[:axis]
11.                 reducedFeatVec.extend(featVec[axis + 1:])
12.                 retDataSet.append(reducedFeatVec)
13.             elif featVec[axis] == 'N':
14.                 reducedNullVec = featVec[:axis]
15.                 reducedNullVec.extend(featVec[axis + 1:])
16.                 nullDataSet.append(reducedNullVec)
17.     elif LorR == 'L':
18.         for featVec in dataSet:
19.             if featVec[axis] != 'N':
20.                 if float(featVec[axis]) < value:
21.                     retDataSet.append(featVec)
22.             elif featVec[axis] == 'N':
23.                 nullDataSet.append(featVec)
24.     elif LorR == 'R':
25.         for featVec in dataSet:
26.             if featVec[axis] != 'N':
27.                 if float(featVec[axis]) > value:
28.                     retDataSet.append(featVec)
29.             elif featVec[axis] == 'N':
30.                 nullDataSet.append(featVec)
31.     totalWeightSub = calcTotalWeight(retDataSet, -1, True) # 计算此分支
                        中非空样本的总权重
32.     for nullVec in nullDataSet: # 把缺失值样本按权值比例划分到分支中
33.         nullVec[-2] = float(nullVec[-2]) * totalWeightSub / totalWeightV

```

```

34.         retDataSet.append(nullVec)
35.     return retDataSet

```

4. 函数 calcTotalWeight()

函数名称（参数）	calcTotalWeight(dataSet, labelIndex, isContainNull)
函数功能	计算样本集对某个特征值的总样本数
输入	数据集 dataSet, 特征索引 labelIndex, 是否包含空值 isContainNull
输出	总数 totalWeight

```

1. def calcTotalWeight(dataSet, labelIndex, isContainNull):
2.     totalWeight = 0.0
3.     for featVec in dataSet: # 遍历每个样本
4.         weight = float(featVec[-2])
5.         if isContainNull is False and featVec[labelIndex] != 'N':
6.             totalWeight += weight # 非空样本树, 按权重计算
7.         if isContainNull is True:
8.             totalWeight += weight # 总样本数, 按权重计算
9.     return totalWeight

```

5. 函数 calcGain()

函数名称（参数）	calcGain(dataSet, labelIndex, labelPropertyi)
函数功能	计算信息增益
输入	数据集 dataSet, 特征索引 labelIndex, 特征类型 labelPropertyi
输出	信息增益 gain

```

1. def calcGain(dataSet, labelIndex, labelPropertyi):
2.     baseEntropy = calcShannonEnt(dataSet, labelIndex) # 计算根节点的信息熵
3.     featList = [example[labelIndex] for example in dataSet] # 特征值列表
4.     uniqueVals = set(featList) # 该特征包含的所有值
5.     newEntropy = 0.0
6.     totalWeight = 0.0
7.     totalWeightV = 0.0
8.     totalWeight = calcTotalWeight(dataSet, labelIndex, True) # 总样本权重
9.     totalWeightV = calcTotalWeight(dataSet, labelIndex, False) # 非空样本权重

```

```

10.     if labelPropertyi == 0: # 对离散的特征
11.         for value in uniqueVals: # 对每个特征值, 划分数据集, 计算各子集的信
            息熵
12.             if value != 'N':
13.                 subDataSet = splitDataSet(dataSet, labelIndex, value)
14.                 totalWeightSub = 0.0
15.                 totalWeightSub = calcTotalWeight(subDataSet, labelIndex,
                    True)
16.                 prob = totalWeightSub / totalWeightV
17.                 newEntropy += prob * calcShannonEnt(subDataSet, labelInd
                    ex)
18.     else: # 对连续的特征
19.         uniqueValsList = list(uniqueVals)
20.         if 'N' in uniqueValsList:
21.             uniqueValsList.remove('N')
22.         sortedUniqueVals = sorted(uniqueValsList) # 对特征值排序
23.         listPartition = []
24.         minEntropy = inf
25.         if len(sortedUniqueVals) == 1: # 如果只有一个值, 可以看作只有左子
            集, 没有右子集
26.             totalWeightLeft = calcTotalWeight(dataSet, labelIndex, True)
27.             probLeft = totalWeightLeft / totalWeightV
28.             minEntropy = probLeft * calcShannonEnt(dataSet, labelIndex)
29.         else:
30.             for j in range(len(sortedUniqueVals) - 1): # 计算划分点
31.                 partValue = (float(sortedUniqueVals[j]) + float(
32.                     sortedUniqueVals[j + 1])) / 2
33.                 # 对每个划分点, 计算信息熵
34.                 dataSetLeft = splitDataSet(dataSet, labelIndex, partValu
                    e, 'L')
35.                 dataSetRight = splitDataSet(dataSet, labelIndex, partVal
                    ue, 'R')
36.                 totalWeightLeft = 0.0
37.                 totalWeightLeft = calcTotalWeight(dataSetLeft, labelInde
                    x, True)
38.                 totalWeightRight = 0.0
39.                 totalWeightRight = calcTotalWeight(dataSetRight, labelIn
                    dex, True)
40.                 probLeft = totalWeightLeft / totalWeightV
41.                 probRight = totalWeightRight / totalWeightV
42.                 Entropy = probLeft * calcShannonEnt(dataSetLeft, labelIn
                    dex) + probRight * calcShannonEnt(dataSetRight, labelIndex)
43.                 if Entropy < minEntropy: # 取最小的信息熵
44.                     minEntropy = Entropy
45.                 newEntropy = minEntropy
46.         gain = totalWeightV / totalWeight * (baseEntropy - newEntropy)

```

```
47.     return gain
```

6. 函数 calcGainRatio()

函数名称（参数）	calcGainRatio(dataSet, labelIndex, labelPropertyi)
函数功能	计算信息增益率
输入	数据集 dataSet, 特征索引 labelIndex, 特征类型 labelPropertyi
输出	信息增益率 gainRatio, 最好的划分点 bestPartValuei

```
1. def calcGainRatio(dataSet, labelIndex, labelPropertyi):
2.     baseEntropy = calcShannonEnt(dataSet, labelIndex) # 计算根节点的信息熵
3.     featList = [example[labelIndex] for example in dataSet] # 特征值列表
4.     uniqueVals = set(featList) # 该特征包含的所有值
5.     newEntropy = 0.0
6.     bestPartValuei = None
7.     IV = 0.0
8.     totalWeight = 0.0
9.     totalWeightV = 0.0
10.    totalWeight = calcTotalWeight(dataSet, labelIndex, True) # 总样本权重
11.    totalWeightV = calcTotalWeight(dataSet, labelIndex, False) # 非空样本权重
12.    if labelPropertyi == 0: # 对离散的特征
13.        for value in uniqueVals: # 对每个特征值, 划分数据集, 计算各子集的信息熵
14.            subDataSet = splitDataSet(dataSet, labelIndex, value)
15.            totalWeightSub = 0.0
16.            totalWeightSub = calcTotalWeight(subDataSet, labelIndex, True)
17.            if value != 'N':
18.                prob1 = totalWeightSub / totalWeightV
19.                newEntropy += prob1 * calcShannonEnt(subDataSet, labelIndex)
20.                prob1 = totalWeightSub / totalWeight
21.                IV -= prob1 * log(prob1, 2)
22.        else: # 对连续的特征
23.            uniqueValsList = list(uniqueVals)
24.            if 'N' in uniqueValsList:
25.                uniqueValsList.remove('N')
26.                # 计算空值样本的总权重, 用于计算 IV
27.                totalWeightN = 0.0
28.                dataSetNull = splitDataSet(dataSet, labelIndex, 'N')
```



```

29.         totalWeightN = calcTotalWeight(dataSetNull, labelIndex, True
    )
30.         probNull = totalWeightN / totalWeight
31.         if probNull > 0.0:
32.             IV += -1 * probNull * log(probNull, 2)
33.         sortedUniqueVals = sorted(uniqueValsList) # 对特征值排序
34.         listPartition = []
35.         minEntropy = inf
36.         if len(sortedUniqueVals) == 1: # 如果只有一个值, 可以看作只有左子
            集, 没有右子集
37.             totalWeightLeft = calcTotalWeight(dataSet, labelIndex, True)
38.             probLeft = totalWeightLeft / totalWeightV
39.             minEntropy = probLeft * calcShannonEnt(dataSet, labelIndex)
40.             IV = -1 * probLeft * log(probLeft, 2)
41.         else:
42.             for j in range(len(sortedUniqueVals) - 1): # 计算划分点
43.                 partValue = (float(sortedUniqueVals[j]) + float(sortedUn
            iqueVals[j + 1])) / 2
44.                 # 对每个划分点, 计算信息熵
45.                 dataSetLeft = splitDataSet(dataSet, labelIndex, partValu
            e, 'L')
46.                 dataSetRight = splitDataSet(dataSet, labelIndex, partVal
            ue, 'R')
47.                 totalWeightLeft = 0.0
48.                 totalWeightLeft = calcTotalWeight(dataSetLeft, labelInde
            x, True)
49.                 totalWeightRight = 0.0
50.                 totalWeightRight = calcTotalWeight(dataSetRight, labelIn
            dex, True)
51.                 probLeft = totalWeightLeft / totalWeightV
52.                 probRight = totalWeightRight / totalWeightV
53.                 Entropy = probLeft * calcShannonEnt(dataSetLeft, labelIn
            dex) + probRight * calcShannonEnt(dataSetRight, labelIndex)
54.                 if Entropy < minEntropy: # 取最小的信息熵
55.                     minEntropy = Entropy
56.                     bestPartValuei = partValue
57.                     probLeft1 = totalWeightLeft / totalWeight
58.                     probRight1 = totalWeightRight / totalWeight
59.                     IV += -1 * (probLeft1 * log(probLeft1, 2) + probRigh
            t1 * log(probRight1, 2))
60.                 newEntropy = minEntropy
61.                 gain = totalWeightV / totalWeight * (baseEntropy - newEntropy)
62.                 if IV == 0.0: # 如果属性只有一个值, IV 为 0, 为避免除数为 0, 给个很小的值
63.                     IV = 0.0000000001
64.                 gainRatio = gain / IV
65.                 return gainRatio, bestPartValuei

```

7. 函数 chooseBestFeatureToSplit()

函数名称（参数）	chooseBestFeatureToSplit(dataSet, labelProperty)
函数功能	选择最好的数据集划分方式
输入	数据集 dataSet, 特征类型 labelProperty
输出	最好的特征 bestFeature, 最好特征值 bestPartValue

```

1. def chooseBestFeatureToSplit(dataSet, labelProperty):
2.     numFeatures = len(labelProperty) # 特征数
3.     bestInfoGainRatio = 0.0
4.     bestFeature = -1
5.     bestPartValue = None # 连续的特征值, 最佳划分值
6.     gainSum = 0.0
7.     gainAvg = 0.0
8.     for i in range(numFeatures): # 对每个特征循环
9.         infoGain = calcGain(dataSet, i, labelProperty[i])
10.        gainSum += infoGain
11.    gainAvg = gainSum / numFeatures
12.    for i in range(numFeatures): # 对每个特征循环
13.        infoGainRatio, bestPartValuei = calcGainRatio(dataSet, i, labelProperty[i])
14.        infoGain = calcGain(dataSet, i, labelProperty[i])
15.        if infoGainRatio > bestInfoGainRatio and infoGain > gainAvg: #
            取信息增益高于平均增益且信息增益率最大的特征
16.            bestInfoGainRatio = infoGainRatio
17.            bestFeature = i
18.            bestPartValue = bestPartValuei
19.    return bestFeature, bestPartValue

```

8. 函数 majorityCnt()

函数名称（参数）	majorityCnt(classList, weightList)
函数功能	通过排序返回出现次数最多的类别
输入	类别 classList, 权重 weightList
输出	类型 sortedClassCount[0][0], sortedClassCount[0][1], sortedClassCount[1][1]

```

1. def majorityCnt(classList, weightList):
2.     classCount = {}
3.     for i in range(len(classList)):
4.         if classList[i] not in classCount.keys():
5.             classCount[classList[i]] = 0.0

```

```

6.         classCount[classList[i]] += round(float(weightList[i]),1)
7.     sortedClassCount = sorted(classCount.items(),key=operator.itemgetter
    (1), reverse=True)
8.     if len(sortedClassCount) == 1:
9.         return (sortedClassCount[0][0],sortedClassCount[0][1],0.0)
10.    return (sortedClassCount[0][0], sortedClassCount[0][1], sortedClassC
    ount[1][1])

```

9. 函数 createTree()

函数名称（参数）	createTree(dataSet, labels, labelProperty)
函数功能	创建树
输入	数据集 dataSet, 特征 labels, 特征属性 labelProperty
输出	树 myTree

```

1. def createTree(dataSet, labels, labelProperty):
2.     classList = [example[-1] for example in dataSet] # 类别向量
3.     weightList = [example[-2] for example in dataSet] # 权重向量
4.     if classList.count(classList[0]) == len(classList): # 如果只有一个类
        别, 返回
5.         totalWeiht = calcTotalWeight(dataSet,0,True)
6.         return (classList[0], round(totalWeiht,1),0.0)
7.     if len(dataSet[0]) == 1: # 如果所有特征都被遍历完了, 返回出现次数最多的
        类别
8.         return majorityCnt(classList)
9.     bestFeat, bestPartValue = chooseBestFeatureToSplit(dataSet,labelProp
        erty) # 最优分类特征的索引
10.    if bestFeat == -1: # 如果无法选出最优分类特征, 返回出现次数最多的类别
11.        return majorityCnt(classList, weightList)
12.    if labelProperty[bestFeat] == 0: # 对离散的特征
13.        bestFeatLabel = labels[bestFeat]
14.        myTree = {bestFeatLabel: {}}
15.        labelsNew = copy.copy(labels)
16.        labelPropertyNew = copy.copy(labelProperty)
17.        del (labelsNew[bestFeat]) # 已经选择的特征不再参与分类
18.        del (labelPropertyNew[bestFeat])
19.        featValues = [example[bestFeat] for example in dataSet]
20.        uniqueValue = set(featValues) # 该特征包含的所有值
21.        uniqueValue.discard('N')
22.        for value in uniqueValue: # 对每个特征值, 递归构建树
23.            subLabels = labelsNew[:]
24.            subLabelProperty = labelPropertyNew[:]
25.            myTree[bestFeatLabel][value] = createTree(splitDataSetWithNu
                ll(dataSet, bestFeat, value), subLabels,subLabelProperty)
26.    else: # 对连续的特征, 不删除该特征, 分别构建左子树和右子树

```

```

27.     bestFeatLabel = labels[bestFeat] + '<' + str(bestPartValue)
28.     myTree = {bestFeatLabel: {}}
29.     subLabels = labels[:]
30.     subLabelProperty = labelProperty[:]
31.     # 构建左子树
32.     valueLeft = 'Y'
33.     myTree[bestFeatLabel][valueLeft] = createTree(splitDataSetWithNull(dataSet, bestFeat, bestPartValue, 'L'), subLabels, subLabelProperty)
34.     # 构建右子树
35.     valueRight = 'N'
36.     myTree[bestFeatLabel][valueRight] = createTree(splitDataSetWithNull(dataSet, bestFeat, bestPartValue, 'R'), subLabels, subLabelProperty)
37.     return myTree

```

10. 函数 classify ()

函数名称（参数）	classify(inputTree, classList, featLabels, featLabelProperties, testVec)
函数功能	分类
输入	树 inputTree, 类 classList, 特征 featLabels, 特征属性 featLabelProperties, 测试数据 testVec
输出	类别标签 classLabel

```

1. def classify(inputTree, classList, featLabels, featLabelProperties, testVec):
2.     firstStr = list(inputTree.keys())[0] # 根节点
3.     firstLabel = firstStr
4.     lessIndex = str(firstStr).find('<')
5.     if lessIndex > -1: # 如果是连续型的特征
6.         firstLabel = str(firstStr)[:lessIndex]
7.     secondDict = inputTree[firstStr]
8.     featIndex = featLabels.index(firstLabel) # 跟节点对应的特征
9.     classLabel = {}
10.    for classI in classList:
11.        classLabel[classI] = 0.0
12.    for key in secondDict.keys(): # 对每个分支循环
13.        if featLabelProperties[featIndex] == 0: # 离散的特征
14.            if testVec[featIndex] == key: # 测试样本进入某个分支
15.                if type(secondDict[key]).__name__ == 'dict': # 该分支不是叶子节点, 递归
16.                    classLabelSub = classify(secondDict[key], classList, featLabels, featLabelProperties, testVec)
17.                    for classKey in classLabel.keys():
18.                        classLabel[classKey] += classLabelSub[classKey]

```

```

19.         else: # 如果是叶子, 返回结果
20.             for classKey in classLabel.keys():
21.                 if classKey == secondDict[key][0]:
22.                     classLabel[classKey] += secondDict[key][1]
23.                 else:
24.                     classLabel[classKey] += secondDict[key][2]
25.         elif testVec[featIndex] == 'N': # 如果测试样本的属性值缺失, 则
            进入每个分支
26.             if type(secondDict[key]).__name__ == 'dict': # 该分支不
                是叶子节点, 递归
27.                 classLabelSub = classify(secondDict[key], classList,
                    featLabels, featLabelProperties, testVec)
28.                 for classKey in classLabel.keys():
29.                     classLabel[classKey] += classLabelSub[key]
30.             else: # 如果是叶子, 返回结果
31.                 for classKey in classLabel.keys():
32.                     if classKey == secondDict[key][0]:
33.                         classLabel[classKey] += secondDict[key][1]
34.                     else:
35.                         classLabel[classKey] += secondDict[key][2]
36.         else:
37.             partValue = float(str(firstStr)[lessIndex + 1:])
38.             if testVec[featIndex] == 'N': # 如果测试样本的属性值缺失, 则对
                每个分支的结果加和
39.                 # 进入左子树
40.                 if type(secondDict[key]).__name__ == 'dict': # 该分支不
                    是叶子节点, 递归
41.                     classLabelSub = classify(secondDict[key], classList,
                        featLabels, featLabelProperties, testVec)
42.                     for classKey in classLabel.keys():
43.                         classLabel[classKey] += classLabelSub[classKey]
44.                 else: # 如果是叶子, 返回结果
45.                     for classKey in classLabel.keys():
46.                         if classKey == secondDict[key][0]:
47.                             classLabel[classKey] += secondDict[key][1]
48.                         else:
49.                             classLabel[classKey] += secondDict[key][2]
50.             elif float(testVec[featIndex]) <= partValue and key == 'Y':
                # 进入左子树
51.                 if type(secondDict['Y']).__name__ == 'dict': # 该分支不
                    是叶子节点, 递归
52.                     classLabelSub = classify(secondDict['Y'], classList,
                        featLabels, featLabelProperties, testVec)
53.                     for classKey in classLabel.keys():
54.                         classLabel[classKey] += classLabelSub[classKey]
55.                 else: # 如果是叶子, 返回结果

```

```

56.         for classKey in classLabel.keys():
57.             if classKey == secondDict[key][0]:
58.                 classLabel[classKey] += secondDict['Y'][1]
59.             else:
60.                 classLabel[classKey] += secondDict['Y'][2]
61.         elif float(testVec[featIndex]) > partValue and key == 'N':
62.             if type(secondDict['N']).__name__ == 'dict': # 该分支不
                是叶子节点，递归
63.                 classLabelSub = classify(secondDict['N'], classList,
                    featLabels, featLabelProperties, testVec)
64.                 for classKey in classLabel.keys():
65.                     classLabel[classKey] += classLabelSub[classKey]
66.             else: # 如果是叶子，返回结果
67.                 for classKey in classLabel.keys():
68.                     if classKey == secondDict[key][0]:
69.                         classLabel[classKey] += secondDict['N'][1]
70.                     else:
71.                         classLabel[classKey] += secondDict['N'][2]
72.         return classLabel

```

11. 函数 storeTree()

函数名称（参数）	storeTree(inputTree, filename)
函数功能	存储决策树
输入	树 inputTree, 文件名 filename
输出	-

```

1. def storeTree(inputTree, filename):
2.     import pickle
3.     fw = open(filename, 'w')
4.     pickle.dump(inputTree, fw)
5.     fw.close()

```

12. 函数 grabTree()

函数名称（参数）	grabTree(filename)
函数功能	读取决策树
输入	文件名 filename
输出	文件不存在返回 None

```

1. def grabTree(filename):
2.     import pickle
3.     if os.path.isfile(filename):
4.         fr = open(filename)

```

```

5.         return pickle.load(fr)
6.     else:
7.         return None

```

13. 函数 acctestng()

函数名称（参数）	acctestng(myTree, classList, data_test, labels, labelProperties)
函数功能	测试决策树正确率
输入	树 myTree, 属性 classList, 测试集 data_test, 类别 labels, 属性类型 labelProperties
输出	错误率 error

```

1. def acctestng(myTree, classList, data_test, labels, labelProperties):
2.     error = 0.0
3.     acc=0
4.     for i in range(len(data_test)):
5.         classkey = data_test[i][-1]
6.         classLabelSet = classify(myTree, classList, labels, labelPropert
ies, data_test[i]) # classlabel
7.         if classLabelSet[classkey] != 0:
8.             acc += 1
9.     acc=acc/len(data_test)
10.    print(acc)
11.    return float(error)

```

14. 函数 testingMajor()

函数名称（参数）	testingMajor(major, data_test)
函数功能	测试投票节点正确率
输入	测试节点 major, 测试集 data_test
输出	错误率 error

```

1. def testingMajor(major, data_test):
2.     error = 0.0
3.     for i in range(len(data_test)):
4.         if major[0] != data_test[i][-1]:
5.             error += 1
6.     return float(error)

```

15. 函数 postPruningTree()

函数名称（参	postPruningTree(inputTree, classSet, dataSet, data_test, labels,
--------	------------------------------------------------------------------

数)	labelProperties)
函数功能	划分训练集和测试集
输入	树 inputTree, 类 classSet, 训练集 dataSet, 测试集 data_test, 特征 labels, 特征属性 labelProperties
输出	类型 sortedClassCount[0][0], sortedClassCount[0][1], sortedClassCount[1][1]

```

1. def postPruningTree(inputTree, classSet, dataSet, data_test, labels, labelProperties):
2.     firstStr = list(inputTree.keys())[0]
3.     secondDict = inputTree[firstStr]
4.     classList = [example[-1] for example in dataSet]
5.     weightList = [example[-2] for example in dataSet]
6.     featkey = copy.deepcopy(firstStr)
7.     if '<' in firstStr: # 对连续的特征值, 使用正则表达式获得特征标签和 value
8.         featkey = re.compile("(.<)").search(firstStr).group()[:-1]
9.         featvalue = float(re.compile("<.+").search(firstStr).group()[1:])
10.    labelIndex = labels.index(featkey)
11.    temp_labels = copy.deepcopy(labels)
12.    temp_labelProperties = copy.deepcopy(labelProperties)
13.    if labelProperties[labelIndex] == 0: # 离散特征
14.        del (labels[labelIndex])
15.        del (labelProperties[labelIndex])
16.    for key in secondDict.keys(): # 对每个分支
17.        if type(secondDict[key]).__name__ == 'dict': # 如果不是叶子节点
18.            if temp_labelProperties[labelIndex] == 0: # 离散的
19.                subDataSet = splitDataSet(dataSet, labelIndex, key)
20.                subDataTest = splitDataSet(data_test, labelIndex, key)
21.            else:
22.                if key == 'Y':
23.                    subDataSet = splitDataSet(dataSet, labelIndex, featvalue, 'L')
24.                    subDataTest = splitDataSet(data_test, labelIndex, featvalue, 'L')
25.                else:
26.                    subDataSet = splitDataSet(dataSet, labelIndex, featvalue, 'R')
27.                    subDataTest = splitDataSet(data_test, labelIndex, featvalue, 'R')
28.                if len(subDataTest) > 0:

```



```

29.         inputTree[firstStr][key] = postPruningTree(secondDict[key], classSet, subDataSet, subDataTest, copy.deepcopy(labels), copy.deepcopy(labelProperties))
30.     if testing(inputTree, classSet, data_test, temp_labels, temp_labelProperties) <= testingMajor(majorityCnt(classList, weightList), data_test):
31.         return inputTree
32.     return majorityCnt(classList, weightList)

```

16. 函数 split_test_train()

函数名称（参数）	split_test_train(dataset, p)
函数功能	划分训练集和测试集
输入	数据集 dataset, 测试集比例 p
输出	测试集 test_data, 训练集 training_data

```

1. def split_test_train(dataset, p): # 按照 1: p-1 分测试集
2.     n = len(dataset)
3.     fore = int(n*p)
4.     rear = n-fore
5.     test_data = dataset[:fore] # 少的部分
6.     training_data = dataset[-rear:]
7.     return test_data, training_data

```

17. 程序入口

```

1. import C45
2. import treePlotter
3. import random
4. fr = open(r'D:\Download\C4.5\C4.5 决策树\iris.txt') # 读取数据文件
5. list_n = [inst.strip().split(',') for inst in fr.readlines()] # 生成数据集
6. random.shuffle(list_n)
7. labels = ['f1', 'f2', 'f3', 'f4', 'class'] # 样本特征标签
8. test_data, training_data = split_test_train(list_n, 0.2)
9. validation_data, training_data2 = split_test_train(training_data, 0.3)
10. labelProperties = [1, 1, 1, 1] # 样本特征类型, 0 为离散, 1 为连续
11. classList = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
12. trees = C45.createTree(training_data2, labels, labelProperties) # 构建决策树
13. treePlotter.createPlot(trees) # 绘制决策树
14. C45.acctesting(trees, classList, test_data, labels, labelProperties)
15. C45.postPruningTree(trees, classList, training_data2, validation_data, labels, labelProperties) # 剪枝
16. treePlotter.createPlot(trees)
17. C45.acctesting(trees, classList, test_data, labels, labelProperties)

```

2.2.2 结果分析

1 训练集和测试集划分

周志华在《机器学习》中提出，常见划分样本集的做法是将大约 $2/3 \sim 4/5$ 的样本用于训练，剩余样本用于测试。为了得到泛化能力较强的决策树模型，我们依据模型的测试准确率越高越好的原则，将 Iris 样本集划分为训练集和测试集。

我们编写 `split_test_train` 函数，以训练样本：测试样本=4:1 划分 Iris 样本集。之后使用 `random.shuffle` 函数打乱数据列表顺序，以实现随机性。

2 信息增益和信息增益比的选择

决策树的生成过程中，对于在每一个节点处，按照哪一个属性来进行划分，我们可以采用信息增益或信息增益比来决定。前文介绍中提到，ID3 算法采用的是信息增益，C4.5 决策树采用的是信息增益比。信息增益比是在信息增益的基础上对特征取值的个数进行惩罚，使信息增益不再倾向于取值多的特征。两种分支方法的选择通过设置生成决策树的参数来实现。

首先，创建 `calcShannonEnt` 函数计算信息熵。接着，创建 `chooseBestFeatureToSplit` 函数，通过计算得到不同属性的信息熵，进而得到相应属性划分前后的信息增益，选择最大的信息增益或信息增益率对应的属性进行结点划分。

分别使用信息增益和信息增益率作为划分依据，计算模型的预测准确率。我们发现，由于鸢尾花数据集的特征数据是连续型数据，不存在一个特征的取值比另一个特征多很多的情况，所以在鸢尾花数据集上，使用信息增益或是信息增益比的结果差异不大。测试情况也是如此。

```
In [17]: runfile('C:/Users/mac/Desktop/机器学习/ratio.py',
wdir='C:/Users/mac/Desktop/机器学习')
Reloaded modules: tree_plotter
['f1', 'f2', 'f3', 'f4', 'class']
测试集占比0.2时，模型预测准确率为0.9666666666666667

In [18]: runfile('C:/Users/mac/Desktop/机器学习/ratio.py',
wdir='C:/Users/mac/Desktop/机器学习')
Reloaded modules: tree_plotter
['f1', 'f2', 'f3', 'f4', 'class']
测试集占比0.2时，模型预测准确率为0.9666666666666667
```

3 决策树剪枝

剪枝是减少决策树“过拟合”的主要手段，目的是用最简洁的结构训练出泛化能力最好的模型。《机器学习》中提到，后剪枝决策树的欠拟合风险很小，泛化性能优于预剪枝决策树，因此我们选择对生成的决策树进行后剪枝。

在划分完成的训练集和测试集的基础上，再次预留 30% 的训练集中的样本用作验证集，以进行后剪枝的评估。

我们创建 `postPruningTree` 函数，利用划分好的验证集对决策树进行后剪枝。并得出模型预测准确率。运行多次发现，训练出来的决策树模型的预测准确率总体有所提升。因此，我们最终采用后剪枝操作后的决策树作为最终的训练模型。

```
In [19]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
0.9666666666666667
1.0

In [20]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9
0.9333333333333333

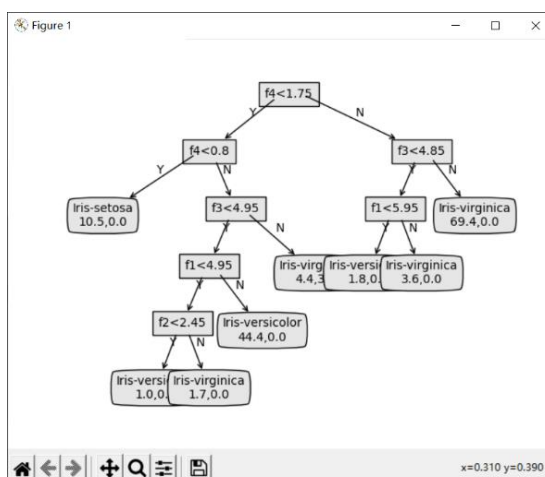
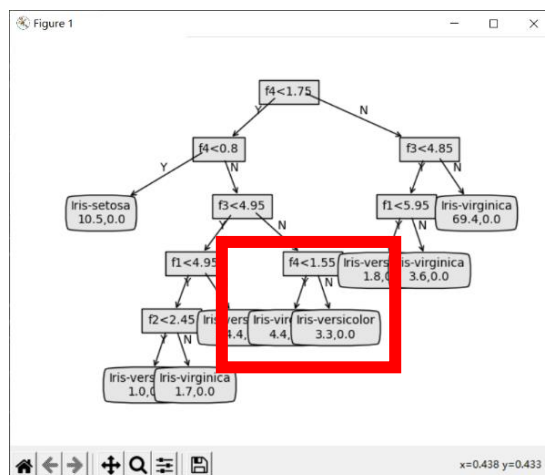
In [21]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9666666666666667
0.9666666666666667

In [22]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9333333333333333
1.0
```

4 决策树可视化

决策树模型具有较强的可解释性，为了更直观地展示模型的树形结构，我们对生成的决策树进行可视化。

以其中一次训练中剪枝前后决策树为例作图如下



2.3 神经网络

2.3.1 理论基础

1 softmax 函数

softmax 函数，又称归一化指数函数。它是二分类函数 sigmoid 在多元分类上的推广，目的是将多元分类的结果以概率的形式展现出来。

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{c=1}^C e^{z_c}}$$

其中 z_i 为第 i 个节点的输出值， C 为输出节点的个数，即分类的类别个数。

softmax 第一步是将模型的预测结果转化到指数函数上，这样保证了概率的非负性。第二步是计算转化后结果占总数的百分比，即对转换后的结果进行归一化处理，确保各个预测结果的概率之和等于 1。这样就将多元分类的输出值转换为范围在 $[0, 1]$ 且和为 1

的概率分布。

2 cross-entropy 函数

交叉熵主要刻画的是实际输出（概率）与期望输出（概率）的距离，也就是交叉熵的值越小，两个概率分布就越接近。在多分类问题中，交叉熵通常采取如下形式：

$$L_{\log}(Y, P) = -\log \Pr(Y|P) = -\frac{1}{N} \sum_{i=0}^{N-1} \sum_{k=0}^{K-1} y_{i,k} \log p_{i,k}$$

其中 N 为样本数， K 为类别数， y 为样本真实类别， p 为预测概率分布。

softmax 搭配 cross-entropy 通常是解决分类问题的通用方案。作为回归问题的常见损失函数，均方误差 (MSE) 公式为：

$$loss_{MSE}(y, t) = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2$$

但它并不适合分类问题。在分类问题中，样本的真实分布用 one-hot 编码来表示，一个样本的表示向量中有且仅有一个维度为 1，其余为 0。若使用 MSE，即使与真实类别所对应下标的预测值是正确的，其他项预测值的分布也会影响损失的大小，这不符合我们对于分类问题损失函数的预期。

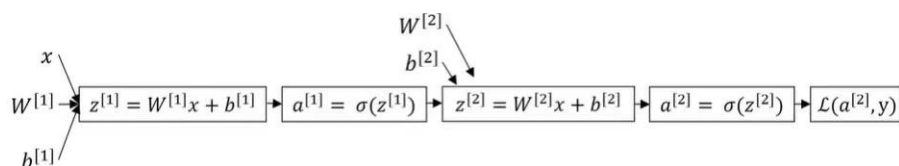
而 cross-entropy 适合用于分类问题，因为其值只与真实类别所对应下标的预测值有关，且该预测值越大，损失函数越小。

3 softmax 与 cross-entropy 求导

$$\begin{aligned} \frac{\partial H(\hat{y}, y)}{\partial z_j} &= \sum_{i=1}^m \frac{\partial H(\hat{y}, y)}{\partial y_i} \frac{\partial y_i}{\partial z_j} \\ &= \sum_{i=1}^m -\frac{\hat{y}_i}{y_i} \cdot \frac{\partial y_i}{\partial z_j} \\ &= \left(-\frac{\hat{y}_j}{y_j} \cdot \frac{\partial y_j}{\partial z_j} \right)_{i=j} + \sum_{i=1, i \neq j}^m -\frac{\hat{y}_i}{y_i} \cdot \frac{\partial y_i}{\partial z_j} \\ &= -\frac{\hat{y}_j}{y_j} \cdot y_j (1 - y_j) + \sum_{i=1, i \neq j}^m -\frac{\hat{y}_i}{y_i} \cdot -y_i y_j \\ &= -\hat{y}_j + \hat{y}_j y_j + \sum_{i=1, i \neq j}^m \hat{y}_i y_j \\ &= -\hat{y}_j + y_j \sum_{i=1}^m \hat{y}_i \\ &= y_j - \hat{y}_j \end{aligned}$$

4 梯度计算

在 Iris 分类问题中使用单隐层神经网络，其正向传播模型如下：



各参数梯度下降公式如下：

$$dz^{[2]} = a^{[2]} - y$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

在有多多个样本的情况下，只需将上面的变量向量化后，即可借助 Python 中的 numpy 库方便地计算。

2.3.2 代码实现

1. 函数 split_data()

函数名称（参数）	split_data(X, Y)
函数功能	划分训练集和测试集
输入	特征矩阵 X，标签矩阵 Y
输出	训练集和测试集各自的 X 和 Y

```
def split_data(X, Y):
    data = np.hstack((X, Y))    #将X, Y矩阵合并
    np.random.shuffle(data)    #随机打乱
    data_train = data[0:120, :] #80%作为训练集
    data_test = data[120:150, :] #20%作为测试集
    X_train = data_train[:, 0:4].T
    Y_train = data_train[:, 4].reshape((1, 120))
    X_test = data_test[:, 0:4].T
    Y_test = data_test[:, 4].reshape((1, 30))
    return X_train, Y_train, X_test, Y_test
```

2. 函数 initialize()

函数名称（参数）	initialize(input_num, output_num, hidden_num)
函数功能	随机初始化参数
输入	输入层、输出层和隐含层的节点个数
输出	初始参数

```
def initialize(input_num, output_num, hidden_num):
    W1 = np.random.randn(hidden_num, input_num) * 0.01
    b1 = np.zeros((hidden_num, 1))
    W2 = np.random.randn(output_num, hidden_num) * 0.01
    b2 = np.zeros((output_num, 1))

    params = {'W1': W1,
              'b1': b1,
              'W2': W2,
              'b2': b2}

    return params
```

- *0.01 的原因在于避免 W 的初始值过大，计算出来的值落在 sigmoid 函数饱和区域，学习速度过慢。
- 将四个参数矩阵存储在名为 params 的字典内

3. 函数 sigmoid()

函数名称（参数）	sigmoid(Z)
函数功能	激活函数 sigmoid
输入	线性计算得到的矩阵
输出	激活值构成的矩阵

```
def sigmoid(Z):
    A = 1. / (1 + np.exp(-Z))
    return A
```

4. 函数 softmax()

函数名称（参数）	softmax(Z)
函数功能	输出层激活函数 softmax
输入	线性计算得到的矩阵
输出	激活值构成的矩阵

```
def softmax(Z):
    A = np.exp(Z) / np.sum(np.exp(Z), axis=0)
    return A
```

5. 函数 forward_propagation()

函数名称（参数）	forward_propagation(X, params, output_num)
函数功能	神经网络正向传播
输入	特征矩阵，参数，输出层节点个数
输出	输出矩阵，缓冲值

```
def forward_propagation(X, params, output_num):
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']
    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = softmax(Z2)

    assert (A2.shape == (output_num, X.shape[1]))

    cache = {'Z1': Z1,
             'A1': A1,
             'Z2': Z2,
             'A2': A2}
    return A2, cache
```

- 在隐含层线性计算后使用 sigmoid 函数，在输出层线性计算后使用 softmax 函数
- 缓冲值 cache 包括在隐含层计算得到的 Z1、A1 和在输出层得到的 Z2、A2，存储在一个字典内

6. 函数 compute_cost_reg()

函数名称（参数）	compute_cost_reg(A2, Y_one_hot, params, lamb)
函数功能	计算训练集上的误差
输入	输出值矩阵，one-hot 编码形式的标签矩阵，参数，正则项系数 lambda
输出	误差

```
def compute_cost_reg(A2, Y_one_hot, params, lamb):
    m = Y_one_hot.shape[1]
    A2 = np.clip(A2, 1e-12, 1. - 1e-12)
    W1 = params['W1']
    W2 = params['W2']
    cost = -1. / m * np.sum(Y_one_hot * np.log(A2 + 1e-9)) + lamb / (2*m) * (np.sum(W1*W1) + np.sum(W2*W2))
    return cost
```

- 在代价函数中加入了 L2 正则项，以降低过拟合风险，即：

$$J(w, b) = \frac{1}{m} \sum_{i=1}^m L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

7. 函数 backward_propagation()

函数名称（参数）	backward_propagation(X, Y_one_hot, params, cache)
函数功能	神经网络的反向传播
输入	特征矩阵，one-hot 编码的标签矩阵，参数，缓冲值
输出	各参数的梯度


```
def backward_propagation(X, Y_one_hot, params, cache):
    m = X.shape[1]  #样本个数
    W1 = params['W1']
    W2 = params['W2']
    A1 = cache['A1']
    A2 = cache['A2']

    dZ2 = A2 - Y_one_hot
    dW2 = 1. / m * np.dot(dZ2, A1.T) + lamb / m * W2
    db2 = 1. / m * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (A1 * (1 - A1))
    dW1 = 1. / m * np.dot(dZ1, X.T) + lamb / m * W1
    db1 = 1. / m * np.sum(dZ1, axis=1, keepdims=True)

    grads = {'dW1': dW1,
              'db1': db1,
              'dW2': dW2,
              'db2': db2}

    return grads
```

- 因为在代价函数中加入了 L2 正则项,所以在反向传播计算梯度的时候也要加上相应项
- 将四个梯度存储在一个名为 grads 的字典内

8. 函数 update_parameters()

函数名称 (参数)	update_parameters(params, grads, learning_rate)
函数功能	根据梯度更新参数
输入	参数, 梯度, 学习率
输出	更新后的参数

```
def update_parameters(params, grads, learning_rate):
    W1 = params['W1']
    b1 = params['b1']
    W2 = params['W2']
    b2 = params['b2']

    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    params = {'W1': W1,
              'b1': b1,
              'W2': W2,
              'b2': b2}

    return params
```

9. 函数 `neural_network()`

函数名称（参数）	<code>neural_network(X, Y, hidden_num, output_num, num_iterations, learning_rate, lamb)</code>
函数功能	训练神经网络
输入	特征矩阵，标签矩阵，隐含层和输出层的节点数，迭代次数，学习率，正则项系数
输出	训练得到的参数

```
def neural_network(X, Y, hidden_num, output_num, num_iterations, learning_rate, lamb):
    input_num = X.shape[0]
    m = X.shape[1]

    params = initialize(input_num, output_num, hidden_num)  #初始化

    Y_one_hot = np.zeros((output_num, m))
    for i in range(m):
        Y_one_hot[int(Y[0, i]), i] = 1  #将Y转换成one-hot编码形式

    #迭代
    cost_list = []
    for i in range(num_iterations):
        A2, cache = forward_propagation(X, params, output_num)  #正向传播
        cost = compute_cost_reg(A2, Y_one_hot, params, lamb)  #计算误差
        cost_list = np.append(cost_list, cost)
        grads = backward_propagation(X, Y_one_hot, params, cache)  #反向传播
        params = update_parameters(params, grads, learning_rate)  #更新参数

    #打印误差
    if (i == 0):
        print("最初误差为: %f" % cost)
    if ((i+1) % 100 == 0):
        print("经过%i次迭代，误差为: %f" % ((i+1), cost))

    #画图
    iteration = np.arange(0, num_iterations, 1)
    plt.figure(figsize=(4, 2.5))
    plt.plot(iteration, cost_list)
    plt.xlabel('number of iterations')
    plt.ylabel('error')
    plt.show()

    return params
```

10 程序入口

```

if __name__ == "__main__":
    #读取数据文件
    X = np.loadtxt('iris.txt', delimiter=',', usecols=(0,1,2,3), dtype=float)
    Y = np.loadtxt('iris.txt', delimiter=',', usecols=(4), dtype=str)

    #将三个品种分别用0,1,2表示
    for i in range(Y.shape[0]):
        if Y[i] == 'Iris-setosa':
            Y[i] = 0
        elif Y[i] == 'Iris-versicolor':
            Y[i] = 1
        elif Y[i] == 'Iris-virginica':
            Y[i] = 2
    Y = Y.astype(np.int)
    Y = Y[:, np.newaxis]

    X_train, Y_train, X_test, Y_test = split_data(X, Y) #划分数据集

    #一系列参数
    input_num = 4
    output_num = 3
    hidden_num = 5
    num_iterations = 1000
    learning_rate = 0.1
    lamb = 0.001

    print("隐藏层神经元个数: %d" % hidden_num)
    print("学习率: %s" % str(learning_rate))
    print("迭代次数: %d" % num_iterations)
    print("正则化系数: %s" % str(lamb))
    print("\n")

    #训练
    params = neural_network(X_train, Y_train, hidden_num, output_num, num_iterations, learning_rate, lamb)

    #测试
    test_num = X_test.shape[1]
    A2_test, cache_test = forward_propagation(X_test, params, output_num)
    predictions = np.argmax(A2_test, axis=0).reshape((1, test_num))
    correct = 0
    for i in range(test_num):
        if (predictions[0, i] == Y_test[0, i]):
            correct += 1
    precision = correct / test_num
    print("在测试集上的分类正确率为: %.2f%%" % (precision * 100))

```

- 将鸢尾花的三个品种分别用数字 0、1、2 表示
- 一系列神经网络的参数包括各层的节点数、迭代次数、学习率和正则项系数都放在主函数中以便调试
- 划分训练集和测试集，将训练集投入 `neural_network()` 函数训练，获得各个参数，以在测试集上完成分类任务，并且计算和打印分类正确率

2.3.3 结果分析

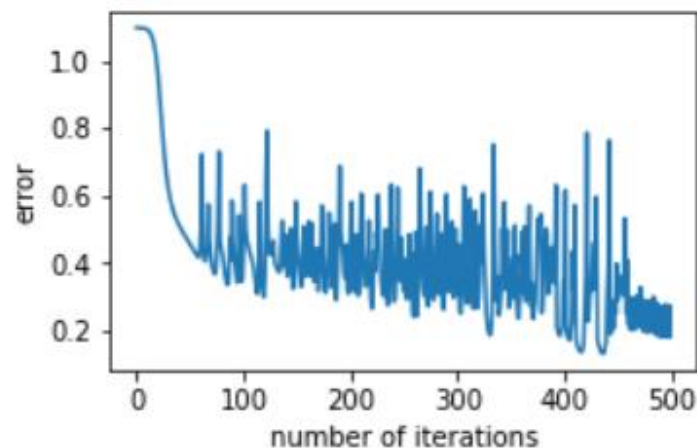
Iris 数据集内一共有 150 个样本，考虑到数据集不大，同时多次调试隐藏层节点数、学习率、迭代次数这些参数，发现隐藏层 3 个节点已经足够训练出较准确的模型。

学习率若设得较大比如为 1 时，尽管损失函数最后收敛到较小值，但出现严重的振荡问题（如左图），降低了学习效率；设得较小比如为 0.03 时，学习速度非常慢，为了达到同样的性能，需要更多次的迭代；最后发现学习率设为 0.1 时比较合适。

迭代次数在 1000 左右时训练误差已经很低，加大迭代次数误差减少得非常缓慢，趋于饱和，甚至可能有过拟合的风险，因此让迭代次数停止在 1000。运行结果如下所示：

隐藏层神经元个数：3
学习率：1
迭代次数：500
正则化系数：0.001

最初误差为：1.098656
经过100次迭代，误差为：0.351056
经过200次迭代，误差为：0.352951
经过300次迭代，误差为：0.306565
经过400次迭代，误差为：0.344158
经过500次迭代，误差为：0.270596



在测试集上的分类正确率为：90.00%

隐藏层神经元个数：3

学习率：0.1

迭代次数：1000

正则化系数：0.001

最初误差为：1.098913

经过100次迭代，误差为：1.096019

经过200次迭代，误差为：1.056948

经过300次迭代，误差为：0.779105

经过400次迭代，误差为：0.587887

经过500次迭代，误差为：0.516871

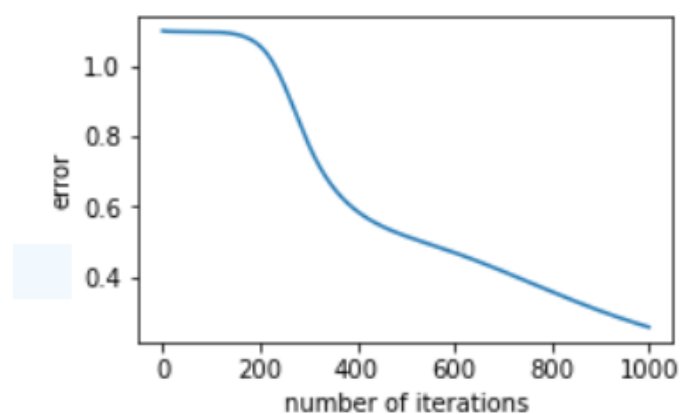
经过600次迭代，误差为：0.470078

经过700次迭代，误差为：0.416850

经过800次迭代，误差为：0.359178

经过900次迭代，误差为：0.304504

经过1000次迭代，误差为：0.258437



在测试集上的分类正确率为：100.00%

因此，3 个隐藏层神经元、0.1 的学习率、1000 次迭代，已经能使该分类器在测试集上的准确率达到 96.67% 甚至 100%（如右图），有较好的性能。

3 income 数据集

3.1 数据描述

3.1.1 特征统计

1 特征描述

我们导入 Income 数据集，查看数据集的基本信息，对数据集进行特征整理，输出所有的变量信息。

代码：

```
1. import pandas as pd
2. import numpy as np
3. df = pd.read_csv('train.csv')
4. df.head
5. df.info()
```

输出结果：

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 54256 entries, 0 to 54255
```

```
Data columns (total 42 columns):
```

#	Column	Non-Null Count	Dtype
0	id	54256 non-null	float64
1	age	54256 non-null	float64
2	class of worker	54256 non-null	object
3	detailed industry recode	54256 non-null	float64
4	detailed occupation recode	54256 non-null	float64
5	education	54256 non-null	object
6	wage per hour	54256 non-null	float64
7	enroll in edu inst last wk	54256 non-null	object
8	marital stat	54256 non-null	object
9	major industry code	54256 non-null	object
10	major occupation code	54256 non-null	object
11	race	54256 non-null	object
12	hispanic origin	54256 non-null	object
13	sex	54256 non-null	object
14	member of a labor union	54256 non-null	object

15	reason for unemployment	54256 non-null	object
16	full or part time employment stat	54256 non-null	object
17	capital gains	54256 non-null	float64
18	capital losses	54256 non-null	float64
19	dividends from stocks	54256 non-null	float64
20	tax filer stat	54256 non-null	object
21	region of previous residence	54256 non-null	object
22	state of previous residence	54256 non-null	object
23	detailed household and family stat	54256 non-null	object
24	detailed household summary in household	54256 non-null	object
25	migration code-change in msa	54256 non-null	object
26	migration code-change in reg	54256 non-null	object
27	migration code-move within reg	54256 non-null	object
28	live in this house 1 year ago	54256 non-null	object
29	migration prev res in sunbelt	54256 non-null	object
30	num persons worked for employer	54256 non-null	float64
31	family members under 18	54256 non-null	object
32	country of birth father	54256 non-null	object
33	country of birth mother	54256 non-null	object
34	country of birth self	54256 non-null	object
35	citizenship	54256 non-null	object
36	own business or self employed	54256 non-null	float64
37	fill inc questionnaire for veteran's admin	54256 non-null	object
38	veterans benefits	54256 non-null	float64
39	weeks worked in year	54256 non-null	float64
40	year	54256 non-null	int64
41	y	54256 non-null	object

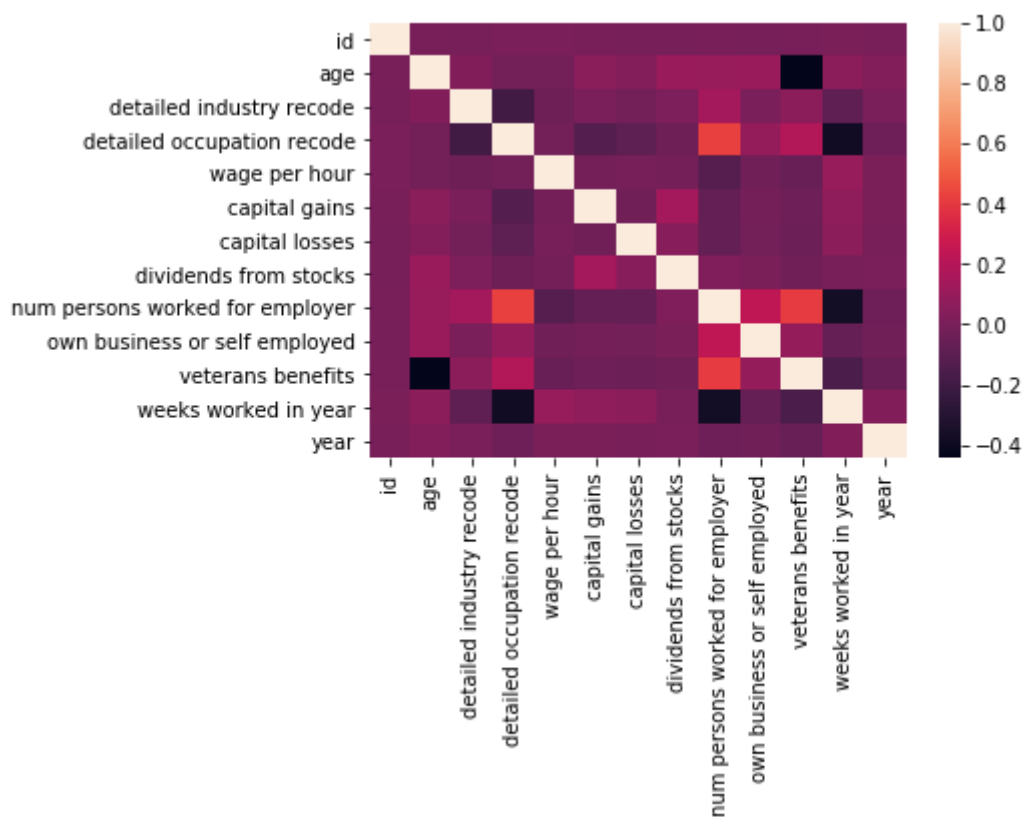
dtypes: float64(12), int64(1), object(29)

memory usage: 17.4+ MB

可以发现，Income 数据集中有 54256 个样本，除去分类特征 y，一共有 40 个特征，样本和特征数量较多。

2 特征相关性

我们利用 Python 中的 seaborn 库绘制热力图，以直观反映属性之间的两两相似度。



3.1.2 数据分布

1 数据是否平衡

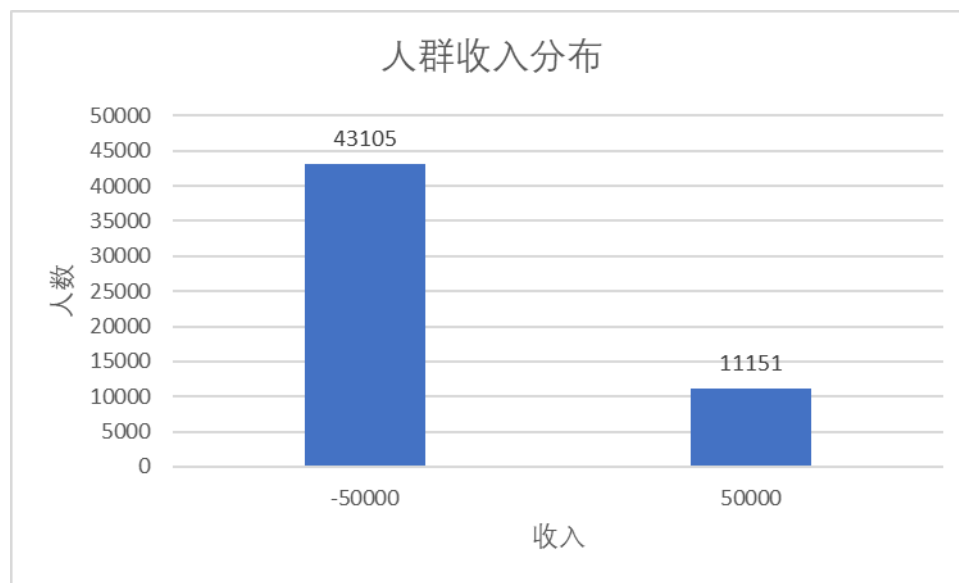
我们查看 Income 数据集的分布情况，判断是否存在分布不均。

代码：

```
1. from collections import Counter
2. print('income ', Counter(df['y']))
```

输出：

```
income Counter({-50000: 43105, 50000: 11151})
```

可见，收入低于 50000 的数据有 43105 条，收入超过 50000 的数据有 11151 条，比较不平衡。

2 缺失数据统计

我们进一步查看数据的缺失情况。

代码：

```
1. (df==0).sum()
```

结果：

id	0
age	160
class of worker	15979
detailed industry recode	16092
detailed occupation recode	16092
education	0
wage per hour	50289
enroll in edu inst last wk	50978
marital stat	0
major industry code	0
major occupation code	16092
race	0
hispanic origin	384
sex	0
member of a labor union	46817
reason for unemployment	52201
full or part time employment stat	0
capital gains	50474

capital losses	52359
dividends from stocks	44418
tax filer stat	0
region of previous residence	48956
state of previous residence	49191
detailed household and family stat	0
detailed household summary in household	0
migration code-change in msa	26945
migration code-change in reg	26802
migration code-move within reg	26802
live in this house 1 year ago	26802
migration prev res in sunbelt	48956
num persons worked for employer	14446
family members under 18	49514
country of birth father	2313
country of birth mother	2114
country of birth self	1242
citizenship	0
own business or self employed	47043
fill inc questionnaire for veteran's admin	53505
veterans benefits	3175
weeks worked in year	14446
year	0
y	0

其中 9 个特征的缺失数据量过万，在后续的实现中，需要对数据集进行进一步处理。

可以发现，数据集中存在缺失值的属性很多，属性的缺失值也较多，有些特征甚至只有不到 10000 条数据。40 个特征中，存在 29 个特征有缺失数据，其中 17 个特征的缺失数据量超过两万，在后续的实现中，需要对数据集进行进一步处理。

3 其他可视化

为了直观了解 income 数据集背后的特征如数据的集中趋势、离散趋势、数据形状和变量间的关系等，我们对数据集做简单的可视化分析。

以种族和性别为例，对比相应人群的收入情况。

代码：

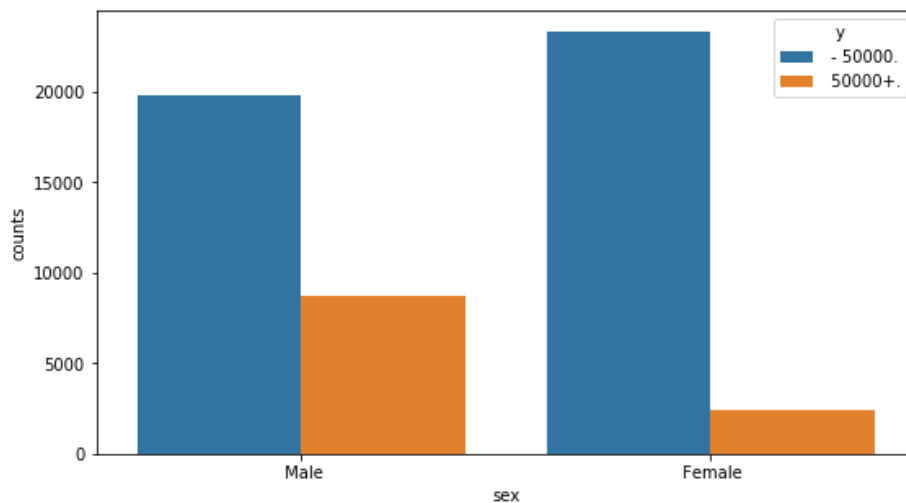
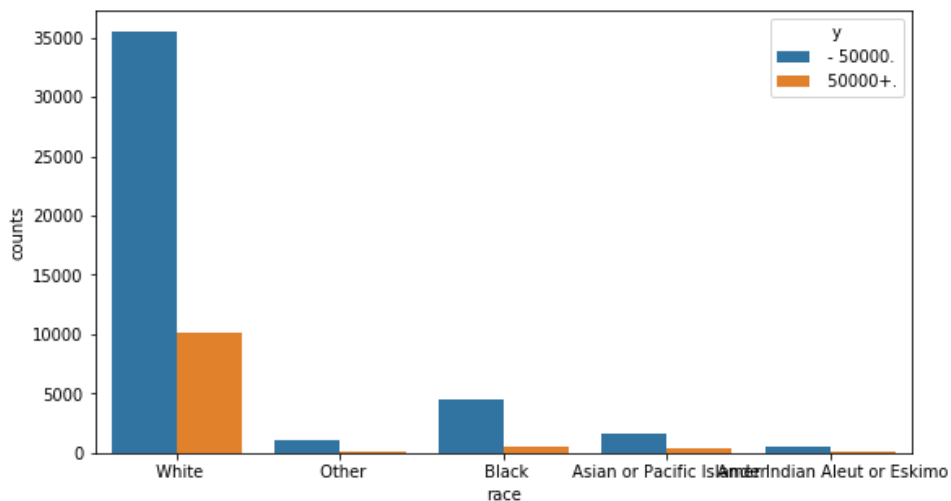
```

1. # 构造不同收入水平下各种族人数的数据
2. race = pd.DataFrame(df.groupby(by = ['race', 'y']).aggregate(np.size).loc[:, 'age'])
3. # 重设行索引
4. race = race.reset_index()
5. # 变量重命名
6. race.rename(columns={'age': 'counts'}, inplace=True)
```

```

7. # 排序
8. race.sort_values(by = ['race', 'counts'], ascending=False, inplace=True)
9. # 构造不同收入水平下各男女人数的数据
10. sex = pd.DataFrame(df.groupby(by = ['sex', 'y']).aggregate(np.size).loc[:, 'age'])
11. sex = sex.reset_index()
12. sex.rename(columns={'age': 'counts'}, inplace=True)
13. sex.sort_values(by = ['sex', 'counts'], ascending=False, inplace=True)
14. # 设置图框比例，并绘图
15. plt.figure(figsize=(9,5))
16. sns.barplot(x="race", y="counts", hue = 'y', data=race)
17. plt.show()
18. plt.figure(figsize=(9,5))
19. sns.barplot(x="sex", y="counts", hue = 'y', data=sex)
20. plt.show()

```



3.2 决策树

3.2.1 数据处理

因为数据集缺失值较多，为了提高数据质量，我们对缺失值进行处理。针对离散型变量，对缺失值进行众数填补；针对连续型变量，进行中位数填补。为了方便，我们使用 `fillna()` 函数填补。

代码：

```
1. df[df==0]=np.nan
2. null_columns1=['age','detailed industry recode','detailed occupation
   recode','num persons worked for employer','own business or self empl
   oyed','veterans benefits']
3. null_columns2=['wage per hour','capital gains','capital losses','divi
   dends from stocks','weeks worked in year']
4. for i in null_columns1:
5.     df.fillna(df[i].mode()[0], inplace=True)
6. for i in null_columns2:
7.     df.fillna(df[i].median(),inplace=True)
```

3.2.2 代码实现

对 Income 数据集，决策树的实现和 Iris 数据集一致，即 2.2.1 部分的内容，二者的区别程序入口的部分，程序入口代码如下：

```
1. import C45
2. import treePlotter
3. import datetime
4. starttime = datetime.datetime.now()
5. fr = open(r'D:\Download\C4.5\C4.5 决策树\train3333.csv')
6. list_n = [inst.strip().split(',') for inst in fr.readlines()]
7. labels = list_n[0]
8. lDataSet=list_n[1:]
9. test_data,training_data=split_test_train(lDataSet, 0.2)
10.validation_data,training_data=split_test_train(training_data,0.3)
11.labelProperties = [1,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,
   ,0,0,1,0,0,0,0,0,1,0,0,1,1]
12.classList = ['1', '-1']
13.trees = C45.createTree(training_data, labels, labelProperties)
14.treePlotter.createPlot(trees)
15.C45.acctestng(trees,classList,test_data,labels,labelProperties)
16.C45.postPruningTree(trees, classList, training_data, validation_data, la
   bels, labelProperties)
17.treePlotter.createPlot(trees)
```

```

18.C45.acctesting(trees,classList,test_data,labels,labelProperties)
19.endtime = datetime.datetime.now()
20.print (endtime - starttime)

```

3.3.3 结果分析

1 划分训练集和测试集

和 Iris 数据集一样，我们编写 `split_test_train` 函数，以训练样本：测试样本=4:1 划分 income 样本集。之后使用 `random.shuffle` 函数打乱数据列表顺序，以实现随机性。具体原理和代码本节不再赘述。

2 特征处理

缺失值超过两万条的特征（如下表），我们将其从数据集中删除。我们认为严重的数据缺失会导致该特征对分类模型的训练起到反作用。

fill inc questionnaire for veteran's admin	53505
capital losses	52359
reason for unemployment	52201
enroll in edu inst last wk	50978
capital gains	50474
wage per hour	50289
state of previous residence	49191
region of previous residence	48956
migration prev res in sunbelt	48956
own business or self employed	47043
member of a labor union	46817
dividends from stocks	44418
migration code-change in msa	26945
migration code-move within reg	26802
migration code-change in reg	26802
live in this house 1 year ago	26802

```

D:\python\python.exe "D:/Download/C4
0.7838908856326606

Process finished with exit code 0

```

特征处理后得到的决策树模型预测准确率约为 0.784.

3 决策树剪枝

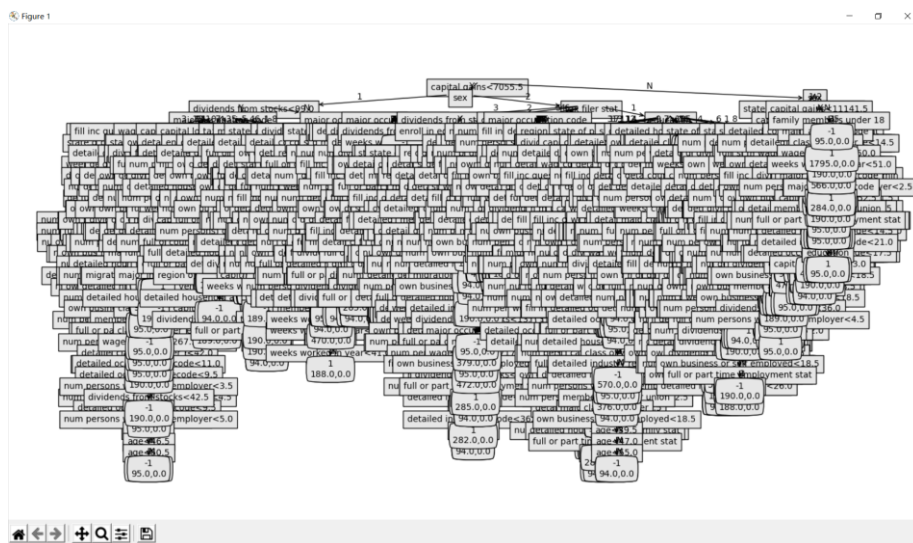
和 Iris 数据集一样，在划分完成的训练集和测试集的基础上，再次预留30%的训练集中的样本用作验证集，以进行后剪枝的评估。具体原理本节不再赘述。

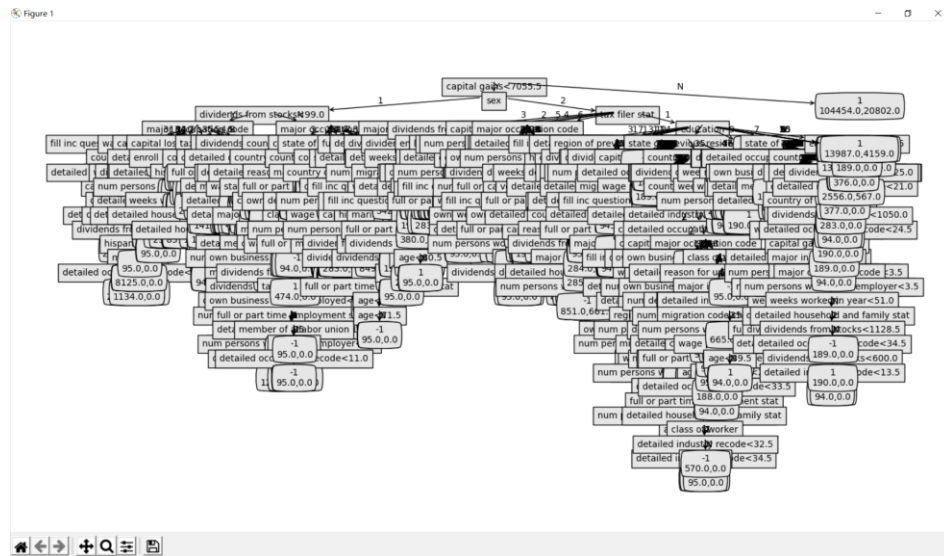
运行多次发现，决策树模型剪枝前后的预测准确率从 0.785 提升到了 0.964，效果提升显著。因此，我们最终采用后剪枝操作后的决策树作为最终的训练模型。

```
D:\python\python.exe D:/Download/C4.5/C4.5决策树/income.py
0.7849046170859829
0.9639664547046355
0:32:22.780974
```

4 决策树可视化

和 Iris 数据集实验一样，我们通过对生成的决策树进行可视化。以其中一次训练中剪枝前后决策树为例作图如下：





由图可见，虽然模型预测准确率较高，但决策树的结构冗杂。

3.3.4 随机森林

1 使用随机森林的原因

通过对数据描述和对数据集的处理可以发现，此数据集存在以下特征：

- ①特征数目多，Income 数据集除去 y ，一共有 40 个特征，容易产生过于复杂的模型。
- ②存在特征关联性比较强的数据，通过热力图可以看出，家庭成员信息、地理位置等特征之间具有关联性。
- ③有时间顺序的数据，存在 Year 这样的时间项，难以处理。
- ④存在缺失值的属性很多，缺失值也较多。
- ⑤数据不平衡，收入超过 50000 和不足 50000 的比例接近 4:1。

以上特征和随机森林擅长处理的数据集有相同之处。考虑到随机森林模型的特有优势（如下表），我们另外使用随机森林模型对 Income 数据集进行建模和预测。

2 随机森林介绍

随机森林的基本原理是把多个决策树模型组合起来形成一个新的大模型，这个大模型最终给出的预测结果由多个决策树综合决定，决定方式为少数服从多数（majority voting）。在决策树模型的建立过程中，随机选择特征子集来使各个树不同，树的每个节点都可以从这些特征中随机的选择子集，来决定怎样更好地分裂数据，每个后来的节点都获得新的、随机选择的特征子集，最终将这些很好的学习了训练数据，但存在很多不同点的树放在一起，采用多数表决进行分类。

3 随机森林优势

随机森林优势点	具体特性
针对特征值	①随机森林能处理很高维度的数据，并且不用做特征选择。 ②在训练过程中，随机森林能够检测到特征之间的影响。在训练完之后，随机森林能给出哪些特征比较重要。 ③训练速度快，容易做成并行化方法。 ④不需要进行剪枝操作。
针对抗干扰能力	①对于不平衡数据集来说，随机森林可以平衡误差。当存在分类不平衡的情况时，随机森林能提供平衡数据集误差的有效方法。 ②当数据存在大量的数据缺失，用随机森林算法仍然可以维持准确度，对错误和离群点更有鲁棒性。
针对准确率	①随机森林抗过拟合能力比较强，虽然理论上说随机森林不会产生过拟合现象，但是在现实中噪声是不能忽略的，没有办法完全消除过拟合，总体上说，决策树容易过度拟合的问题会随着森林的规模增加而削弱。 ②在创建随机森林时候，对泛化误差使用的是无偏估计模型，泛化能力强。

4 随机森林实现

代码：

```

1.  #随机森林建模
2.  param_grid = {
3.      'criterion':['entropy','gini'],
4.      'max_depth':[5, 6, 7, 8],    # 深度: 这里是森林中每棵决策树的深度
5.      'n_estimators':[11,13,15],  # 决策树个数-随机森林特有参数
6.      'max_features':[0.3,0.4,0.5], # 每棵决策树使用的变量占比-随机森林特有参数 (结合原理)
7.      'min_samples_split':[4,8,12,16] # 叶子的最小拆分样本量
8.  }
9.  import sklearn.ensemble as ensemble
10. rfc = ensemble.RandomForestClassifier()
11. rfc_cv = GridSearchCV(estimator=rfc, param_grid=param_grid,
12.                        scoring='roc_auc', cv=4)
13. rfc_cv.fit(X_train, y_train)
14. # 使用随机森林对测试集进行预测
15. test_est = rfc_cv.predict(X_test)
16. print('随机森林精确度...')
17. print(metrics.classification_report(test_est, y_test))
18. print('随机森林 AUC...')
19. fpr_test, tpr_test, th_test = metrics.roc_curve(test_est, y_test) #
    构造 roc 曲线
  
```



```
20. print('AUC = %.4f' %metrics.auc(fpr_test, tpr_test))
```

输出:

随机森林精确度...

	precision	recall	f1-score	support
-1	0.95	0.90	0.93	18325
1	0.58	0.77	0.66	3378
accuracy			0.88	21703
macro avg	0.77	0.83	0.79	21703
weighted avg	0.90	0.88	0.89	21703

随机森林 AUC...

AUC = 0.8341

调整参数:

```
1. param_grid = {
2.     'criterion':['entropy','gini'],
3.     'max_depth':[15, 17, 19, 21],
4.     'n_estimators':[25, 27, 29,32,36], # 决策树个数- 随机森林特有参数
5.     'max_features':[0.02,0.05,0.1, 0.2],# 每棵决策树使用的变量占比- 随机森林
    特有参数
6.     'min_samples_split':[2, 3, 4, 8, 12, 16] # 叶子的最小拆分样本量
7. }
8. rfc_cv = GridSearchCV(estimator=rfc, param_grid=param_grid,
9.     scoring='roc_auc', cv=4)
10. rfc_cv.fit(X_train, y_train)
11. test_est = rfc_cv.predict(X_test)
12. print('随机森林精确度...')
13. print(metrics.classification_report(test_est, y_test))
14. print('随机森林 AUC...')
15. fpr_test, tpr_test, th_test = metrics.roc_curve(test_est, y_test) # 构造 roc 曲线
16. print('AUC = %.4f' %metrics.auc(fpr_test, tpr_test))
17. rfc_cv.best_params_
```

输出:

随机森林精确度...

	precision	recall	f1-score	support
-1	0.96	0.91	0.93	18248
1	0.61	0.79	0.69	3455
accuracy			0.89	21703

macro avg	0.78	0.85	0.81	21703
weighted avg	0.90	0.89	0.89	21703

随机森林 AUC...

AUC = 0.8472

使用随机森林模型训练，得到的模型预测准确率为 0.89，相比于剪枝前的决策树模型，具有更好的预测效果。

5 特征重要性排序

由随机森林对特征随机采样的特点可知，随机森林可以应用于判断各个属性的重要性。重要程度高的属性对模型的性能起到较大作用，也是我们在现实问题中需要重点关注的部分。

我们尝试使用随机森林对 income 数据集的特征进行重要性排序，代码如下：

```

1.  from collections import Counter
2.  from sklearn.ensemble import RandomForestRegressor
3.  from sklearn.model_selection import ShuffleSplit
4.  from sklearn.metrics import r2_score
5.  from collections import defaultdict
6.  from sklearn.model_selection import train_test_split
7.  for line in df:
8.      line = line.strip("")
9.      line = line.strip('"')
10.     line = line.split(",")
11.     #line = [float(x) for x in line]
12. y = df['y']
13. X = df.iloc[:, 1:-1]
14. rf = RandomForestRegressor()
15. scores = defaultdict(list)
16. names = list(df.columns.values)
17. for i in range(1,100):
18.     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=12345)
19.     r = rf.fit(X_train, y_train)
20.     acc = r2_score(y_test, rf.predict(X_test))
21.     for i in range(X.shape[1]):
22.         X_t = X_test.copy()
23.         X_t=np.array(X_t)
24.         np.random.shuffle(X_t[:, i])
25.         shuff_acc = r2_score(y_test, rf.predict(X_t))
26.         scores[names[i]].append((acc-shuff_acc)/acc)
27. print ("Features sorted by their score:")
28. print( sorted([(round(np.mean(score), 4), feat) for

```

```
29.         feat, score in scores.items()], reverse=True))
```

输出结果:

Features sorted by their score: [(0.3045, 'veterans benefits'), (0.2601, 'detailed industry recode'), (0.2197, 'full or part time employment stat'), (0.1617, 'capital losses'), (0.1542, 'hispanic origin'), (0.1505, 'id'), (0.0685, 'detailed occupation recode'), (0.0459, 'capital gains'), (0.0323, 'class of worker'), (0.0314, 'migration prev res in sunbelt'), (0.0176, 'marital stat'), (0.0114, 'education'), (0.0106, 'major industry code'), (0.0079, 'state of previous residence'), (0.0061, 'dividends from stocks'), (0.0046, 'age'), (0.0045, 'race'), (0.0045, 'citizenship'), (0.0042, 'major occupation code'), (0.0039, 'detailed household and family stat'), (0.0022, 'sex'), (0.0021, 'country of birth father'), (0.001, 'family members under 18'), (0.0008, 'country of birth mother'), (0.0002, 'fill inc questionnaire for veteran's admin'), (0.0002, 'enroll in edu inst last wk'), (0.0001, 'num persons worked for employer'), (0.0, 'region of previous residence'), (-0.0, 'member of a labor union'), (-0.0001, 'wage per hour'), (-0.0001, 'live in this house 1 year ago'), (-0.0001, 'country of birth self'), (-0.0002, 'tax filer stat'), (-0.0004, 'own business or self employed'), (-0.001, 'detailed household summary in household'), (-0.0011, 'weeks worked in year'), (-0.0012, 'migration code-move within reg'), (-0.0014, 'migration code-change in reg'), (-0.0015, 'migration code-change in msa'), (-0.002, 'reason for unemployment')]

可见这 40 个属性当中，并不是所有属性的加入都能使模型有更好的效果，我们应当关注重要性高的特征在影响收入水平中起到的作用。

3.3 神经网络

3.3.1 理论基础

1 数据预处理

Income 数据集包含多个特征，每个特征又有很多不同的特征值，并且是离散、无序的，因此需要转换成 one-hot 编码形式，保证每个样本中的单个特征只有 1 位处于状态 1，其他的都是 0。并且把收入大于 5 万的标签记作 1，小于 5 万的标签记作 0。

2 sigmoid 函数和 ReLU 函数

sigmoid 是二分类问题的输出层激活函数，输出范围 (0, 1)，输出的是事件概率，满足某一概率条件我们将其划分正类。

$$S(x) = \frac{1}{1 + e^{-x}}$$

ReLU 函数用作中间隐藏层的激活函数，把所有的负值都变为 0，而正值不变，只有部分神经元会被激活，从而使得网络很稀疏，可以提升计算效率；并且不像 sigmoid 函数，ReLU 没有饱和区，不存在梯度消失问题。

$$f(x) = \max(0, x)$$

3.3.2 代码实现

1. 函数 split_data()

➤ 此函数与 Iris 实验中的同名函数大致相同，依然采用留出法，此处不再重复。

2. 函数 initialize()

函数名称（参数）	initialize(layers_dims)
函数功能	随机初始化参数
输入	各层的节点数
输出	初始参数

```
def initialize(layers_dims):
    L = len(layers_dims)
    params = {}
    for l in range(1, L):
        params['W' + str(l)] = np.random.randn(layers_dims[l], layers_dims[l-1]) * 0.01
        params['b' + str(l)] = np.zeros((layers_dims[l], 1))

    assert (params['W' + str(1)].shape == (layers_dims[1], layers_dims[l-1]))
    assert (params['b' + str(1)].shape == (layers_dims[1], 1))

    return params
```

➤ assert 函数是为了确保矩阵形状正确，若不正确会报错，方便检查问题

3. 函数 sigmoid()

➤ 此函数与 Iris 实验中的同名函数大致相同，此处不再重复。

4. 函数 sigmoid_backward()

函数名称（参数）	sigmoid_backward(dA, cache)
函数功能	计算 sigmoid 上的反向传播
输入	最终输出值的梯度 dA，缓冲值
输出	激活前函数值的梯度 dZ

```
def sigmoid_backward(dA, cache):
    Z = cache
    A = 1. / (1 + np.exp(-Z))
    dZ = dA * A * (1-A)
    assert (dZ.shape == Z.shape)
    return dZ
```

5. 函数 relu()

函数名称（参数）	relu(Z)
函数功能	激活函数 ReLU
输入	线性计算得到的值 Z
输出	激活值 A，缓冲值

```
def relu(Z):
    A = np.maximum(0, Z)
    assert (A.shape == Z.shape)
    cache = Z
    return A, cache
```

6. 函数 relu_backward()

函数名称（参数）	relu_backward(dA, cache)
函数功能	计算在 ReLU 上的反向传播
输入	激活值的梯度 dA，缓冲值
输出	激活前函数值的梯度 dZ

```
def relu_backward(dA, cache):
    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0 #当Z值小于0时，导数皆为0
    assert (dZ.shape == Z.shape)
    return dZ
```

7. 函数 linear_forward()

函数名称（参数）	linear_forward(A, W, b)
函数功能	每一层正向传播的线性计算部分
输入	前一层的输出值 A，该层线性计算所需参数 W 和 b
输出	线性计算所得值，缓冲值（包括 A，W，b）

```
def linear_forward(A, W, b):
    Z = np.dot(W, A) + b
    assert (Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)
    return Z, cache
```

8. 函数 linear_activation_forward()

函数名称（参数）	linear_activation_forward(A_prev, W, b, activation)
函数功能	每一层的正向传播
输入	前一层的输出值 A_prev，参数 W 和 b，激活函数名称

输出	该层输出值 A，所有缓冲值（包括线性计算和激活部分）
----	----------------------------

```
def linear_activation_forward(A_prev, W, b, activation):
    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

- 根据传入的激活函数名称，选择不同的激活函数进行正向传播
- 在该函数中需要调用上一个函数 linear_forward()

9. 函数 L_model_forward ()

函数名称（参数）	L_model_forward(X, params)
函数功能	整个神经网络的正向传播
输入	特征矩阵，参数
输出	输出层的输出 AL，所有缓冲值

```
def L_model_forward(X, params):
    caches = []
    A = X
    L = len(params) // 2
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, params['W' + str(l)], params['b' + str(l)], "relu")
        caches.append(cache)

    AL, cache = linear_activation_forward(A, params['W' + str(L)], params['b' + str(L)], "sigmoid")
    caches.append(cache)

    assert (AL.shape == (1, X.shape[1]))

    return AL, caches
```

- 在该函数中通过循环调用上一个函数 linear_activation_forward()来完成整个神经网络所有层的正向传播

10. 函数 compute_cost()

函数名称（参数）	compute_cost(AL, Y)
函数功能	计算在训练集上的误差
输入	神经网络输出，真实类别标签
输出	输出层的输出 AL，所有缓冲值

```
def compute_cost(AL, Y):
    m = Y.shape[1]
    cost = - np.sum(np.multiply(np.log(AL), Y) + np.multiply(np.log(1 - AL), 1 - Y)) / m
    cost = np.squeeze(cost)
    assert (cost.shape == ())
    return cost
```

- 代价函数也为交叉熵，但是在本实验中用的是在二分类问题中常用的形式，与 Iris 数据集中用的一般形式不同。

11. 函数 linear_backward()

函数名称（参数）	linear_backward(dZ, cache)
函数功能	每一层反向传播的线性计算部分
输入	该层梯度 dZ, 缓冲值
输出	上一层输出的梯度 dA_prev, 参数梯度 dW 和 db

```
def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]
    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis=1, keepdims=True) / m
    dA_prev = np.dot(W.T, dZ)

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db
```

12. 函数 linear_activation_backward()

函数名称（参数）	linear_activation_backward(dA, cache, activation)
函数功能	每一层的反向传播
输入	该层梯度 dA, 缓冲值, 激活函数名称
输出	上一层输出的梯度 dA_prev, 参数梯度 dW 和 db

```
def linear_activation_backward(dA, cache, activation):
    linear_cache, activation_cache = cache
    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

- 在该函数中需要调用上一个函数 `linear_backward()`，以及两个不同激活函数求梯度的函数

13. 函数 `L_model_backward()`

函数名称（参数）	<code>L_model_backward(AL, Y, caches)</code>
函数功能	整个神经网络的反向传播
输入	该层梯度 <code>dA</code> , 缓冲值, 激活函数名称
输出	上一层输出的梯度 <code>dA_prev</code> , 参数梯度 <code>dW</code> 和 <code>db</code>

```
def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches)
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)
    dAL = -(np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    current_cache = caches[L - 1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache, "sigmoid")

    for l in reversed(range(L - 1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" + str(l + 2)], current_cache, "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

- 在该函数中通过循环调用上一个函数 `linear_activation_backward()`来完成整个神经网络所有层的反向传播，并且在输出层传入激活函数名称 `sigmoid`，剩余隐藏层则传入名称 `relu`

- 求 `dAL` 时用到公式 $\frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$

14. 函数 `update_parameters()`

函数名称（参数）	<code>update_parameters(params, grads, learning_rate)</code>
函数功能	更新参数
输入	当前参数, 梯度, 学习率
输出	更新后的参数

```
def update_parameters(params, grads, learning_rate):
    L = len(params) // 2
    for l in range(L):
        params["W" + str(l + 1)] = params["W" + str(l + 1)] - learning_rate * grads["dW" + str(l + 1)]
        params["b" + str(l + 1)] = params["b" + str(l + 1)] - learning_rate * grads["db" + str(l + 1)]
    return params
```

- 与前面单隐层神经网络代码中该函数不同之处在于加入循环，因为不仅仅只有两层参数

15. 函数 L_layer_NN()

函数名称（参数）	L_layer_NN(X, Y, layers_dims, learning_rate, num_iterations, mini_batch)
函数功能	训练神经网络
输入	特征矩阵，标签，各层节点数，学习率，迭代次数，用 mini BGD 时每次用训练集样本的多少
输出	参数

```
def L_layer_NN(X, Y, layers_dims, learning_rate, num_iterations, mini_batch):
    cost_list = []
    params = initialize(layers_dims) #初始化参数
    m = int(X.shape[1] / mini_batch) #用mini BGD时每一次用到的训练集中样本数量

    #迭代
    for i in range(num_iterations):
        #mini BGD每次使用一部分训练集
        for j in range(mini_batch):
            AL, caches = L_model_forward(X[:, m*j: m*(j+1)], params) #正向传播
            grads = L_model_backward(AL, Y[:, m*j: m*(j+1)], caches) #反向传播
            params = update_parameters(params, grads, learning_rate) #更新参数

        cost = compute_cost(AL, Y[:, m*j: m*(j+1)]) #计算误差
        cost_list = np.append(cost_list, cost) #将误差存储起来便于画图
        #打印误差
        if (i == 0):
            costs.append(cost)
            print ("最初误差为: %f" % cost)
        if ((i+1) % 100 == 0):
            costs.append(cost)
            print ("经过%i次迭代, 误差为: %f" % ((i+1), cost))

    #画图
    iteration = np.arange(0, num_iterations, 1)
    plt.figure(figsize=(4, 2.5))
    plt.plot(iteration, cost_list)
    plt.xlabel('number of iterations')
    plt.ylabel('error')
    plt.show()

    return params
```

16 程序入口

```

if __name__ == "__main__":
    #导入数据文件
    X = np.loadtxt('features.txt', delimiter=',', skiprows=1, dtype=float)
    Y = np.loadtxt('labels.txt', delimiter=',', skiprows=1, usecols=(1))

    X_train, Y_train, X_test, Y_test = split_data(X, Y)    #划分数据集

    #一系列参数
    layers_dims = [511, 20, 15, 8, 1]
    num_iterations = 1000
    learning_rate = 0.001
    mini_batch = 12

    print(f"神经网络共{len(layers_dims)}层，各层节点数分别为：")
    for i in layers_dims:
        print(i, end = ' ')
    print("\n")
    print("学习率：%s" % str(learning_rate))
    print("每次学习用训练集的1/", str(mini_batch))
    print("迭代次数：%d" % num_iterations)
    print("\n")

    #训练
    params = L_layer_NN(X_train, Y_train, layers_dims, learning_rate, num_iterations, mini_batch)

    #测试
    test_num = X_test.shape[1]
    AL_test, caches_test = L_model_forward(X_test, params)
    AL_test[(AL_test < 0.5) | (AL_test == 0.5)] = 0
    AL_test[AL_test > 0.5] = 1
    correct = 0
    for i in range(test_num):
        if (AL_test[0, i] == Y_test[0, i]):
            correct += 1
    precision = correct / test_num
    print("在测试集上的分类正确率为：%.2f%%" % (precision * 100))

```

- 在主函数中需要赋值的参数包括神经网络各层的节点数、迭代次数、学习率、每次学习用到训练集样本的多少（1/mini_batch）
- 神经网络最终输出值若 ≥ 0.5 ，则被划分为类别 1，若 < 0.5 则被划分为类别 0

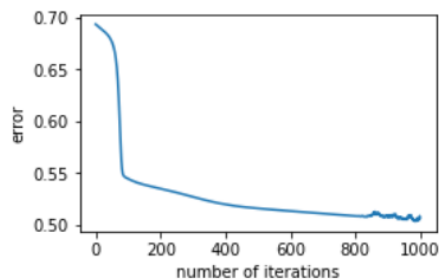
3.3.3 结果分析

Income 数据集包含五万多个样本，特征维度也较高，数据量比较大。构建了一个 5 层的深层神经网络，多次实验发现学习率取值为 0.002 时较为合理，迭代次数在 1000 左右损失函数已经基本收敛。在使用批量梯度下降（BGD）时，运行结果如下图所示。然而批量梯度下降在训练集较大时，累计误差下降到一定程度后，进一步下降变得很慢，结果中可以看出最后误差在 0.5 左右，多次实验发现该神经网络在测试集的分类正确率在 80% 左右，最高可能达到 82%。

神经网络共5层，各层节点数分别为：
511 20 15 8 1

学习率：0.002
迭代次数：1000

最初误差为：0.693708
经过100次迭代，误差为：0.544601
经过200次迭代，误差为：0.534849
经过300次迭代，误差为：0.526991
经过400次迭代，误差为：0.519652
经过500次迭代，误差为：0.515859
经过600次迭代，误差为：0.513316
经过700次迭代，误差为：0.510923
经过800次迭代，误差为：0.508312
经过900次迭代，误差为：0.507983
经过1000次迭代，误差为：0.506541



在测试集上的分类正确率为：82.98%

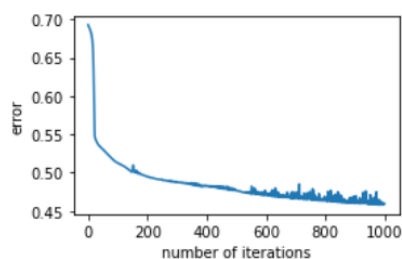
于是后续采用 mini BGD 的方式，每次只让神经网络学习训练集的一部分，循环直到学完所有训练集样本。当每次学习的样本量占训练集总样本数分别为 1/4、1/6、1/12 时，运行结果分别如下图所示：

Iris 及 Income 数据集分类实验报告

神经网络共5层，各层节点数分别为：
511 20 15 8 1

学习率：0.002
每次学习用训练集的1/ 4
迭代次数：1000

最初误差为：0.692637
经过100次迭代，误差为：0.512675
经过200次迭代，误差为：0.494623
经过300次迭代，误差为：0.487866
经过400次迭代，误差为：0.483045
经过500次迭代，误差为：0.476631
经过600次迭代，误差为：0.474619
经过700次迭代，误差为：0.468926
经过800次迭代，误差为：0.468124
经过900次迭代，误差为：0.459880
经过1000次迭代，误差为：0.459798

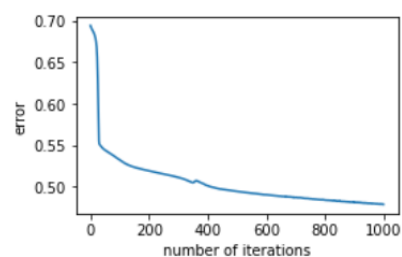


在测试集上的分类正确率为：82.52%
程序运行时间：288.0466380119324

神经网络共5层，各层节点数分别为：
511 20 15 8 1

学习率：0.001
每次学习用训练集的1/ 6
迭代次数：1000

最初误差为：0.693819
经过100次迭代，误差为：0.532163
经过200次迭代，误差为：0.518872
经过300次迭代，误差为：0.511157
经过400次迭代，误差为：0.501106
经过500次迭代，误差为：0.494190
经过600次迭代，误差为：0.490178
经过700次迭代，误差为：0.486878
经过800次迭代，误差为：0.483756
经过900次迭代，误差为：0.481398
经过1000次迭代，误差为：0.478440

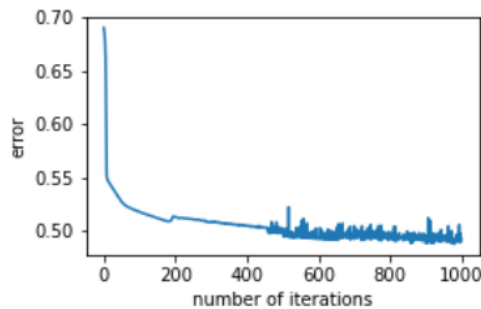


在测试集上的分类正确率为：83.16%
程序运行时间：291.13668966293335

神经网络共5层，各层节点数分别为：
5 11 20 15 8 1

学习率：0.001
每次学习用训练集的1/ 12
迭代次数：1000

最初误差为：0.689989
经过100次迭代，误差为：0.517810
经过200次迭代，误差为：0.512901
经过300次迭代，误差为：0.508272
经过400次迭代，误差为：0.504776
经过500次迭代，误差为：0.498708
经过600次迭代，误差为：0.494349
经过700次迭代，误差为：0.504575
经过800次迭代，误差为：0.491927
经过900次迭代，误差为：0.488545
经过1000次迭代，误差为：0.492639



在测试集上的分类正确率为：79.67%
程序运行时间： 298.4998526573181

可以发现采用 mini BGD 后，在训练集上的误差比 BGD 可以收敛到更低。占比为 1/4、1/6 时，多次实验发现平均分类正确率相比 BGD 稍有提升。而占比更小为 1/12 时，平均正确率有所下降，可能是因为每次所用训练样本数相对较少，数据更不均匀，并且过拟合风险更大。

4 算法对比

4.1 准确率

Iris 数据集特征简单、数据量小，决策树和神经网络在该数据集上都有良好的表现，准确率较高，都达到了 90% 以上。

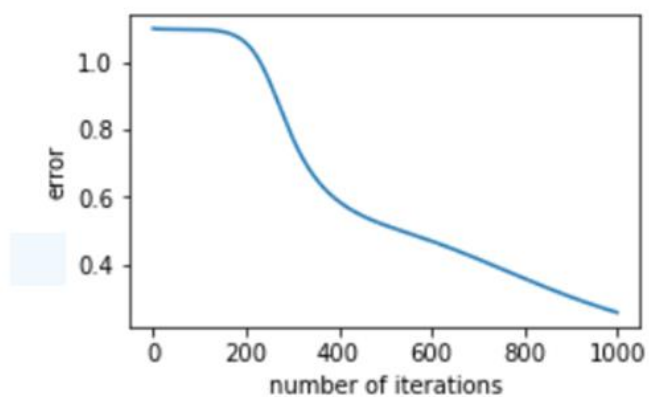
特别是决策树模型在采用剪枝后，准确率提升，从统计学角度看，可以认为二者有同样好的效果。

```
In [19]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
0.9666666666666667
1.0

In [20]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9
0.9333333333333333

In [21]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9666666666666667
0.9666666666666667

In [22]: runfile('C:/Users/mac/Desktop/机器学习/剪枝/iris.py', wdir='C:/Users/mac/Desktop/机器学习/剪枝')
Reloaded modules: C45, treePlotter
0.9333333333333333
1.0
```

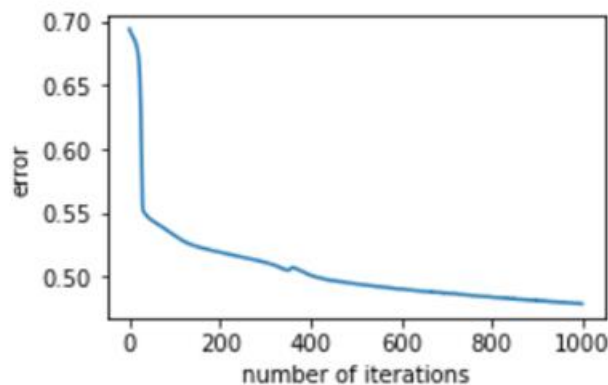


在测试集上的分类正确率为：100.00%

Income 数据集特征多、数据量大，整体数据集较为复杂，决策树和神经网络都达到较为优秀的结果。决策树在进行剪枝前，准确率为 78% 左右，进行剪枝后分类性能提高，准确率达到 96%。决策树的衍生方法——随机森林可以达到 85% 左右的准确率，而神经网络的准确率约为 84%。

在这个数据集中，具有许多相关的特征，虽然神经网络擅长处理具有潜在关联的特征，但是 Income 数据集缺失数据过多，数据质量较差，对模型的建立起到了干扰作用，数据集难以完整覆盖整个收入判断问题，容易产生过拟合的结果。神经网络模型的性能优于决策树剪枝前的结果，但是当决策树通过剪枝弱化过拟合的影响后，决策树的性能就优于神经网络了。此外，应用集成思想的随机森林不需要进行剪枝就可以拥有较为卓越的性能。

```
D:\python\python.exe D:/Download/C4.5/C4.5决策树/income.py
0.7849046170859829
0.9639664547046355
0:32:22.780974
```



在测试集上的分类正确率为：83.16%
 程序运行时间： 291.13668966293335

随机森林精确度...

	precision	recall	f1-score	support
-1	0.96	0.91	0.93	18248
1	0.61	0.79	0.69	3455
accuracy			0.89	21703
macro avg	0.78	0.85	0.81	21703
weighted avg	0.90	0.89	0.89	21703

随机森林 AUC...
 AUC = 0.8472

4.2 运算时间

在 Iris 数据集实验中，由于较小的数据量，模型的训练和测试时间都在很短时间内完成，决策树耗时 5s，神经网络耗时 0.3s，虽然相差的绝对值较小，但已经可以看出二者在时间上的显著差异，神经网络明显快于决策树。

在 Income 数据集实验中，数据量较大、特征较多，二者时间的差距就更加明显。由于决策树算法需要多次迭代、递归，单次训练和测试时间达到了 32 分钟，非常耗时。反观神经网络，训练和测试时间不到 5 分钟。神经网络在面对较大的数据集时，运算速度比决策树要快很多。

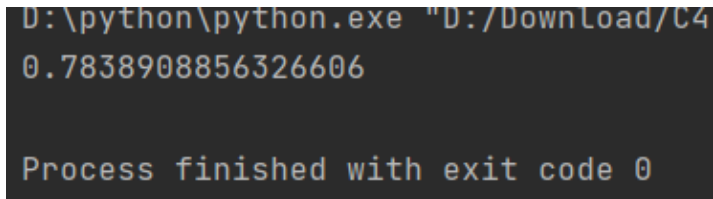
决策树 Iris 运行时间：

```
D:\python\python.exe D:/Downlo
0.9333333333333333
0.9333333333333333
0:00:05.756630
```

神经网络 Iris 运行时间：

程序运行时间： 0.3399980068206787

决策树 Income 运行时间：



```
D:\python\python.exe "D:/Download/C4
0.7838908856326606

Process finished with exit code 0
```

神经网络 Income 运行时间：

程序运行时间： 298.4998526573181

在现实生活中，如果更加注重算法的运算效率，缩短建立模型的时间，神经网络在此方面能有比决策树更好的效果。

4.3 可解释性

决策树作为强可解释性的模型，在 Iris 数据集上的表现非常出色。通过最后可视化的树形结构，自顶向下，可以清晰看出数据集中哪些属性比较关键，以及算法的判断逻辑。而在 Income 数据集中，由于属性（尤其是连续型属性）数量较多，决策树为了尽可能拟合训练集，形成了许多的分支。即使进行了后剪枝操作，和 Iris 数据集相比，决策树的结构还是非常复杂。但总体而言，决策树模型对自己擅长的数据集的可解释性还是高的。

和决策树相比，神经网络由于其算法特点，其可解释性远不如决策树。作为高度非线性黑箱模型，神经网络无法解释自己的推理过程和推理依据。在两个数据集的训练中，难以得出明确而又针对性的优化方案，比如调参具有一定的随机性。