任务一：使用决策树预测隐形眼镜类型

1. **问题**：眼科医生是如何判断患者需要佩戴的镜片类型的？
   隐形眼镜数据集是非常著名的数据集，它包含了很多患者眼部状况的观察条件以及医生推荐的隐形眼镜类型。隐形眼镜类型包括硬材质（hard）、软材质（soft）以及不适合佩戴隐形眼镜（no lenses）。以下为该数据集的部分数据，包括年龄、近视 or 远视类型，是否散光，是否容易流泪，最后 1 列为应佩戴眼镜类型：

```
young   myope   no      reduced no lenses
young   myope   no      normal  soft
young   myope   yes     reduced no lenses
young   myope   yes     normal  hard
young   hyper   no      reduced no lenses
young   hyper   no      normal  soft
young   hyper   yes     reduced no lenses
young   hyper   yes     normal  hard
pre     myope   no      reduced no lenses
```

2. **代码实现：**
   a) **创建决策树：**
   按书上流程（如下图所示）创建决策树：

输入： 训练集 $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)\}$;
　　属性集 $A = \{a_1, a_2, \ldots, a_d\}$.
过程： 函数 TreeGenerate(D, A)
1: 生成结点 node;
2: **if** D 中样本全属于同一类别 C **then**
3: 　将 node 标记为 C 类叶结点; **return**
4: **end if**
5: **if** $A = \varnothing$ **OR** D 中样本在 A 上取值相同 **then**
6: 　将 node 标记为叶结点, 其类别标记为 D 中样本数最多的类; **return**
7: **end if**
8: 从 A 中选择最优划分属性 a_*;
9: **for** a_* 的每一个值 a_*^v **do**
10: 　为 node 生成一个分支; 令 D_v 表示 D 中在 a_* 上取值为 a_*^v 的样本子集;
11: 　**if** D_v 为空 **then**
12: 　　将分支结点标记为叶结点, 其类别标记为 D 中样本最多的类; **return**
13: 　**else**
14: 　　以 TreeGenerate(D_v, $A \setminus \{a_*\}$)为分支结点
15: 　**end if**
16: **end for**
输出： 以 node 为根结点的一棵决策树

```python
def createTree(dataSet, labels):
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        return classList[0]  # 结束划分 如果只有一种分类属性
if len(dataSet[0]) == 1:     # 结束划分 都为同一类属性标签了
        return majorityCnt(classList)   # 计数排序 取最大数特征
```

```
    bestFeat = chooseBestFeatureToSplit(dataSet)   # 获取最优特征索引
    bestFeatLabel = labels[bestFeat]    # 获取最优特征属性标签
    myTree = {bestFeatLabel: {}}        # 决策树初始化 嵌套字典
    del(labels[bestFeat])               # 删除已经使用的特征标签
# 取出数据集所有最优属性值
featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)        # 去重
# 开始构建决策树
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(
                  dataSet, bestFeat, value), subLabels)
    return myTree
```

b) 选择最优划分方式：

采用信息增益为指标。

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1      # 计算特征的数目
baseEntropy = calcShannonEnt(dataSet)
# 计算数据集的原始信息熵 用于与划分完的数据集的香农熵进行比较
    bestInfoGain = 0.0                  # 最佳信息增益初始化
    bestFeature = -1                    # 最佳划分特征初始化
    for i in range(numFeatures):        # 遍历所有的特征
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)      # 特征去重
        newEntropy = 0.0                # 划分后信息熵初始化
        for value in uniqueVals:        # 遍历计算每一个划分后的香农熵
            subDataSet = splitDataSet(dataSet, i, value)    # 划分
            prob = len(subDataSet)/float(len(dataSet))      # 算概率
            newEntropy += prob * calcShannonEnt(subDataSet) # 算熵
        infoGain = baseEntropy - newEntropy     # 计算信息增益
        if (infoGain > bestInfoGain):
            bestInfoGain = infoGain         # 储存最佳信息增益值
            bestFeature = i                 # 储存最佳特征值索引
    return bestFeature
```

c) 计算信息熵：

在计算信息增益时，调用该函数计算信息熵：

```
def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet:     # 为所有可能分类创建字典
        currentLabel = featVec[-1]  # 取数据集的标签
        if currentLabel not in labelCounts.keys():
```

```
            labelCounts[currentLabel] = 0    # 分类标签值初始化
        labelCounts[currentLabel] += 1          # 给标签赋值
    shannonEnt = 0.0                            # 熵初始化
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        shannonEnt -= prob * log(prob, 2)
    return shannonEnt
```

d) 划分数据集

在每个节点后划分数据集。

```
def splitDataSet(dataSet, axis, value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis+1:])
            retDataSet.append(reducedFeatVec)
    return retDataSet
```

e) 确定叶子结点类属性：

如果子节点已经没有属性可以继续划分，则成为叶节点，类属性为其中所占最多的属性。

```
def majorityCnt(classList):
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.items(),
              key=operator.itemgetter(1), reverse=True)
    return sortedClassCount[0][0]
```

f) 由训练集生成决策树：

```
"""
    函数流程:
       取出数据集
       解析由 tap 键分割的数据  去除数据左右的空格
       创造样本决策树
"""
def prodict():
    with open("lenses.txt", "rb") as fr:
        lenses = [inst.decode().strip().split('\t')for inst in fr.re
adlines()]
```

```
    lensesLabels = ['age', 'prescript', 'astigmatic', "tearRate"]
    lensesTree = createTree(lenses, lensesLabels)
    createPlot(lensesTree)
    return lensesTree
```

g) 利用决策树分类：

```
def classify(inputTree, featLabels, testVec):
    firstStr = list(inputTree.keys())[0]    # 得到首 key 值
    secondDict = inputTree[firstStr]
    # 首 key 的 value--->下一个分支
    featIndex = featLabels.index(firstStr)
    # 确定根节点是标签向量中的哪一个（索引）
    key = testVec[featIndex]
    # 确定一个条件后的类别或进入下一个分支有待继续判别
    valueOfFeat = secondDict[key]
    if isinstance(valueOfFeat, dict):
    # 判断实例函数 和 type 函数类似 但是这个更好一点
        classLabel = classify(valueOfFeat, featLabels, testVec)
    else:
        classLabel = valueOfFeat
    return classLabel
```

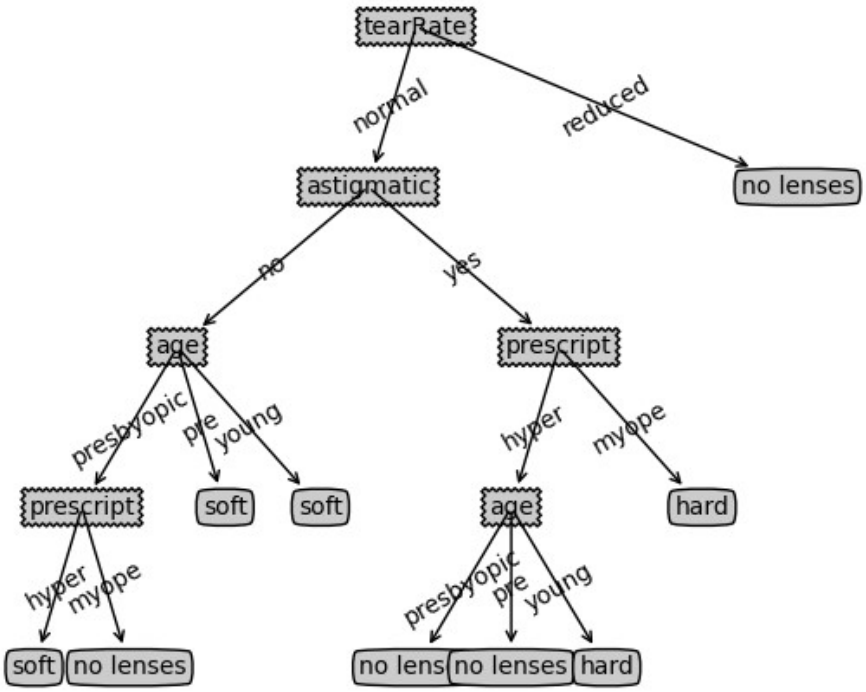
h) 绘制决策树：

```
#1: 绘制带箭头的注解
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    arrow_args = dict(arrowstyle = "<-")
    createPlot.ax1.annotate(nodeTxt, xy = parentPt, xycoords ='axes
fraction', xytext = centerPt,
                            textcoords = 'axes fraction',va="center"
,ha="center",bbox=nodeType,arrowprops=arrow_args)

#2: 创建绘图区，计算树形图的全局尺寸
def createPlot(inTree):
    fig = plt.figure(1, facecolor='white')
    fig.clf()
    axprops = dict(xticks = [], yticks = [])
    createPlot.ax1 = plt.subplot(111, frameon = False, **axprops)
    plotTree.totalW = float(getNumLeafs(inTree))
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0
    plotTree(inTree, (0.5, 1.0), '')
    plt.show()

#3: 获取叶节点的数目
```

```
def getNumLeafs(myTree):
    numLeafs = 0 #初始化叶子
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            numLeafs += getNumLeafs(secondDict[key])
        else:
            numLeafs +=1
    return numLeafs

#4: 获取树的层数
def getTreeDepth(myTree):
    maxDepth = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[key]).__name__=='dict':
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth:
            maxDepth = thisDepth
    return maxDepth

#5: 标注有向边属性
def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0] - cntrPt[0])/2.0 + cntrPt[0]
    yMid = (parentPt[1] - cntrPt[1])/2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString, va="center", ha="cent
er", rotation = 30)

#6: 绘制决策函数
def plotTree(myTree, parentPt, nodeTxt):
    decisionNode = dict(boxstyle = "sawtooth", fc = "0.8")
    leafNode = dict(boxstyle = "round4", fc = "0.8")
    numLeafs = getNumLeafs(myTree)
    defth = getTreeDepth(myTree)
    firstStr = list(myTree.keys())[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.t
otalW, plotTree.yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondeDict = myTree[firstStr]
```

```
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD
    for key in secondeDict.keys():
        if type(secondeDict[key]) is dict:
            plotTree(secondeDict[key], cntrPt, str(key))
        else:
            plotTree.xOff = plotTree.xOff + 1.0/plotTree.totalW
            plotNode(secondeDict[key], (plotTree.xOff, plotTree.yOff
), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(
key))
    plotTree.yOff = plotTree.yOff + 1.0/plotTree.totalD
```

3. 决策树可视化：



任务二：根据用户采集的 WiFi 信息采用决策树预测用户所在房间

1. 数据集：

数据集：训练集存于 TrainDT.csv 中；测试集存于 TestDT.csv 中。

BSSIDLabel： BSSID 标识符，每个 AP（接入点，如路由器）拥有 1 个

或多个不同的 BSSID，但 1 个 BSSID 只属于 1 个 AP；

RSSLabel：该 BSSID 的信号强度，单位 dbm；

RoomLabel: 该 BSSID 被采集时所属的房间号，为类标签，测试集中也含该标签，主要用于计算预测准确度；

SSIDLabel: 该 BSSID 的名称，不唯一；

finLabel：finLabel 标号相同，表示这部分 BSSID 在同一时刻被采集到；我们将在同一时刻采集的所有 BSSID 及其相应 RSS 构成的矢量称为一个指纹$f_i = [BSSID_1: RSS_1, BSSID_2: RSS_2, \dots, \text{RoomLabel}]$；由于 BSSID 的 RSS 在不同位置大小不同，因此指纹可以唯一的标识一个位置。

BSSIDLabel	RSSLabel	RoomLabel	SSIDLabel	finLabel
06:69:6c:0a:bf:02	-56	1	HUST_WIRELESS	1
20:76:93:3a:ae:78	-69	1	HC5761	1
4a:69:6c:07:a1:e7	-69	1	HUST_WIRELESS_AUTO	1
0e:69:6c:0a:bf:02	-63	1	HUST_WIRELESS_AUTO	2
4a:69:6c:07:a1:e7	-66	1	HUST_WIRELESS_AUTO	2
2a:69:6c:05:c5:25	-67	1	HUST_WIRELESS_AUTO	2
08:57:00:7b:63:16	-72	1	TL-WTR9500	2

2. 代码实现：

使用 sklearn 分类决策树完成。

a) 基本参数：

```
'''
分类决策树
'''
DecisionTreeClassifier(criterion="gini",
                       splitter="best",
                       max_depth=None,
                       min_samples_split=2,
                       min_samples_leaf=1,
                       min_weight_fraction_leaf=0.,
                       max_features=None,
                       random_state=None,
                       max_leaf_nodes=None,
                       min_impurity_decrease=0.,
                       min_impurity_split=None,
                       class_weight=None,
                       presort=False)
'''
参数含义：
1.criterion:string, optional (default="gini")
            (1).criterion='gini',分裂节点时评价准则是 Gini 指数。
            (2).criterion='entropy',分裂节点时的评价指标是信息增益。
2.max_depth:int or None, optional (default=None)。指定树的最大深度。
            如果为 None，表示树的深度不限。直到所有的叶子节点都是纯净的，
即叶子节点
```

中所有的样本点都属于同一个类别。或者每个叶子节点包含的样本数小于 min_samples_split。

3.splitter:string, optional (default="best")。指定分裂节点时的策略。

(1).splitter='best',表示选择最优的分裂策略。

(2).splitter='random',表示选择最好的随机切分策略。

4.min_samples_split:int, float, optional (default=2)。表示分裂一个内部节点需要的做少样本数。

(1).如果为整数，则 min_samples_split 就是最少样本数。

(2).如果为浮点数(0 到 1 之间)，则每次分裂最少样本数为 ceil(min_samples_split * n_samples)

5.min_samples_leaf: int, float, optional (default=1)。指定每个叶子节点需要的最少样本数。

(1).如果为整数，则 min_samples_split 就是最少样本数。

(2).如果为浮点数(0 到 1 之间)，则每个叶子节点最少样本数为 ceil(min_samples_leaf * n_samples)

6.min_weight_fraction_leaf:float, optional (default=0.)

指定叶子节点中样本的最小权重。

7.max_features:int, float, string or None, optional (default=None).

搜寻最佳划分的时候考虑的特征数量。

(1).如果为整数，每次分裂只考虑 max_features 个特征。

(2).如果为浮点数(0 到 1 之间)，每次切分只考虑 int(max_features * n_features)个特征。

(3).如果为'auto'或者'sqrt',则每次切分只考虑 sqrt(n_features)个特征

(4).如果为'log2',则每次切分只考虑 log2(n_features)个特征。

(5).如果为 None,则每次切分考虑 n_features 个特征。

(6).如果已经考虑了 max_features 个特征，但还是没有找到一个有效的切分，那么还会继续寻找

下一个特征，直到找到一个有效的切分为止。

8.random_state:int, RandomState instance or None, optional (default=None)

(1).如果为整数，则它指定了随机数生成器的种子。

(2).如果为 RandomState 实例，则指定了随机数生成器。

(3).如果为 None，则使用默认的随机数生成器。

9.max_leaf_nodes: int or None, optional (default=None)。指定了叶子节点的最大数量。

(1).如果为 None,叶子节点数量不限。

(2).如果为整数，则 max_depth 被忽略。

10.min_impurity_decrease:float, optional (default=0.)

如果节点的分裂导致不纯度的减少(分裂后样本比分裂前更加纯净)大于或等于 min_impurity_decrease，则分裂该节点。

加权不纯度的减少量计算公式为：

min_impurity_decrease=N_t / N * (impurity - N_t_R / N_t * right_impurity

- N_t_L / N_t * left_impurity)

其中 N 是样本的总数，N_t 是当前节点的样本数，N_t_L 是分裂后左子节点的样本数，

N_t_R 是分裂后右子节点的样本数。impurity 指当前节点的基尼指数，right_impurity 指

分裂后右子节点的基尼指数。left_impurity 指分裂后左子节点的基尼指数。

11.min_impurity_split:float

树生长过程中早停止的阈值。如果当前节点的不纯度高于阈值，节点将分裂，否则它是叶子节点。

这个参数已经被弃用。用 min_impurity_decrease 代替了 min_impurity_split。

12.class_weight:dict, list of dicts, "balanced" or None, default=None

类别权重的形式为{class_label: weight}

(1).如果没有给出每个类别的权重，则每个类别的权重都为 1。

(2).如果 class_weight='balanced'，则分类的权重与样本中每个类别出现的频率成反比。

计算公式为：n_samples / (n_classes * np.bincount(y))

(3).如果 sample_weight 提供了样本权重(由 fit 方法提供)，则这些权重都会乘以 sample_weight。

13.presort:bool, optional (default=False)

指定是否需要提前排序数据从而加速训练中寻找最优切分的过程。设置为 True 时，对于大数据集

会减慢总体的训练过程；但是对于一个小数据集或者设定了最大深度的情况下，会加速训练过程。

属性:

1.classes_:array of shape = [n_classes] or a list of such arrays

类别的标签值。

2.feature_importances_ : array of shape = [n_features]

特征重要性。越高，特征越重要。

特征的重要性为该特征导致的评价准则的（标准化的）总减少量。它也被称为基尼的重要性

3.max_features_ : int

max_features 的推断值。

4.n_classes_ : int or list

类别的数量

5.n_features_ : int

执行 fit 后，特征的数量

6.n_outputs_ : int

执行 fit 后，输出的数量

7.tree_ : Tree object

树对象，即底层的决策树。

b) 加载数据：

选择有用的数据项['finLabel', 'BSSIDLabel', 'RoomLabel'],

对缺失值，填入-100

```
Train_data_f = pd.read_csv('TrainDT.csv')
Test_data_f = pd.read_csv('TestDT.csv')
imputer = Imputer(missing_values=np.nan, strategy='constant', fill_v
alue=-100)
Train_data = pd.DataFrame(imputer.fit_transform(Train_data_f))
Test_data = pd.DataFrame(imputer.fit_transform(Test_data_f))
Train_data.columns = Train_data_f.columns
Test_data.columns = Test_data_f.columns
feature_train = Train_data_f[['finLabel', 'BSSIDLabel', 'RoomLabel']
]
feature_test = Test_data_f[['finLabel', 'BSSIDLabel', 'RoomLabel']]
```

c) 数据处理：

将数据按 'finLabel' 聚合。

采用所有样本 BSSID 集合的并集作为特征，如指纹f_i的 BSSID 集合为 $B_i = \{BSSID_j | BSSID_j \in f_i\}$。

并按照：

$$f_1 = [BSSID_1: 1, BSSID_2: 0, BSSID_3: 1, BSSID_4: 1,0]$$
$$f_2 = [BSSID_1: 1, BSSID_2: 1, BSSID_3: 1, BSSID_4: 0,1]$$

将输入转为向量。

```
BSSID_v = list(set(Train_data_f['BSSIDLabel']))
BSSID_l = len(BSSID_v)
tarin_bssid = feature_train.groupby('finLabel')
tarin_input = []
Train_data_classes = []
for i, v in tarin_bssid:
    tmp = np.array(v['BSSIDLabel'])
    tmpa = BSSID_l * [0]
    for bssidv in BSSID_v:
        if bssidv in tmp:
            tmpa[BSSID_v.index(bssidv)] = 1
    tarin_input.append(tmpa)
    roomid = np.array(v['RoomLabel'])
    Train_data_classes.append(roomid[1])
Train_data_inputs = np.array(tarin_input)
```

d) 建立决策树：

```
# 建立决策树
在此全部为默认参数了
decision_tree_classifier = DecisionTreeClassifier()
decision_tree_classifier.fit(Train_data_inputs, Train_data_classes)
decision_tree_output = decision_tree_classifier.predict(Test_data_in
puts)
```

e) 在测试集上测试：

```
print('真实值是：')
print(Test_data_classes)

print('预测值是:')
print(decision_tree_output)
score = accuracy_score(Test_data_classes, decision_tree_output)
print(score)
```

3. 实验结果：

在测试集上 accuracy_score 达到 1.0

```
真实值是：
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
预测值是：
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
1.0
```

任务三：IMDB 数据集电影评测分类（二分类问题）

1. 数据集讲解：

该数据集是 IMDB 电影数据集的一个子集，已经划分好了测试集和训练集，训练集包括 25000 条电影评论，测试集也有 25000 条，该数据集已经经过预处理，将每条评论的具体单词序列转化为词库里的整数序列，其中每个整数代表该单词在词库里的位置。例如，整数 104 代表该单词是词库的第 104 个单词。为实验简单，词库仅仅保留了 10000 个最常出现的单词，低频词汇被舍弃。每条评论都具有一个标签，0 表示为负面评论，1 表示为正面评论。

训练数据在 train_data.txt 文件下，每一行为一条评论，训练集标签在 train_labels.txt 文件下，每一行为一条评论的标签；测试数据在 test_data.txt 文件下，测试数据标签未给出。

2. 代码实现：

a) 取出数据集：

从 txt 中取出训练集与测试集：

```
with open("test/test_data.txt", "rb") as fr:
    test_data_n = [inst.decode().strip().split(' ') for inst in fr.r
eadlines()]
```

```
    test_data = [[int(element) for element in line] for line in test
_data_n]
test_data = np.array(test_data)
```

b) 数据处理：

对每条评论，先将其解码为英文单词，再键值颠倒，将整数索引映射为单词。

把整数序列编码为二进制序列。

最后把训练集标签向量化。

```
# 将某条评论解码为英文单词
word_index = imdb.get_word_index()  # word_index 是一个将单词映射为整
数索引的字典
reverse_word_index = dict([(value, key) for (key, value) in word_ind
ex.items()])

# 键值颠倒，将整数索引映射为单词
decode_review = ' '.join(
    [reverse_word_index.get(i - 3, '?') for i in train_data[0]]
)
# 将评论解码
# 注意，索引减去了 3，因为 0,1,2 是为 padding 填充
# "start sequence"序列开始，"unknow"未知词分别保留的索引

# 将整数序列编码为二进制矩阵
def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))  # 创建一个形状为
(len(sequences), dimension)的矩阵
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1  # 将 results[i]的指定索引设为 1
    return results


x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
# 标签向量化
y_train = np.asarray(train_labels).astype('float32')
```

c) 建立决策树：

```
decision_tree_classifier = DecisionTreeClassifier()
decision_tree_classifier.fit(x_train, y_train)
```

d) 输出测试集上的预测结果：

将结果写入 txt

```
decision_tree_output = decision_tree_classifier.predict(x_test)
des = decision_tree_output.astype(int)
np.savetxt('Text3_result.txt', des, fmt='%d', delimiter='\n')
print(decision_tree_output)
```

3. **参数调整：**

使用设置 max_depth 控制树的深度，置 random_state = 30 不变，使用 for 循环寻找，发现深度为 25 时，accuracy_score 最大。

```
In[10]: test = []
   ...: for i in range(5):
   ...:     clf = DecisionTreeClassifier(max_depth=3*i+10
   ...:                                  ,criterion="entropy"
   ...:                                  ,random_state=30
   ...:                                  ,splitter="random"
   ...:                                 )
   ...:     clf = clf.fit(x_train, y_train)
   ...:     scoref = clf.score(x_test_f, y_test)
   ...:     test.append(scoref)
   ...:
In[11]: test
Out[11]: [0.71872, 0.72688, 0.7212, 0.72736, 0.72592]
```

4. **实验结果：**

分离出一部分作为测试集，在测试集上的 accuracy_score 不太理想：

```
In[19]: scoref
Out[19]: 0.72764
```

考虑应该是决策树模型并不适合处理该问题。