机器学习算法 day04_决策树分类算法及应用

课程大纲

决策树分类算法原理	决策树算法概述
	决策树算法思想
	决策树构造
	算法要点
决策树分类算法案例	案例需求
	Python 实现
	决策树的持久化保存

课程目标：

1、理解决策树算法的核心思想

2、理解决策树算法的代码实现

3、掌握决策树算法的应用步骤：数据处理、建模、运算和结果判定

1. 决策树分类算法原理

1.1 概述

决策树（decision tree）——是一种被广泛使用的分类算法。
相比贝叶斯算法，决策树的优势在于构造过程不需要任何领域知识或参数设置
在实际应用中，**对于探测式的知识发现，决策树更加适用**

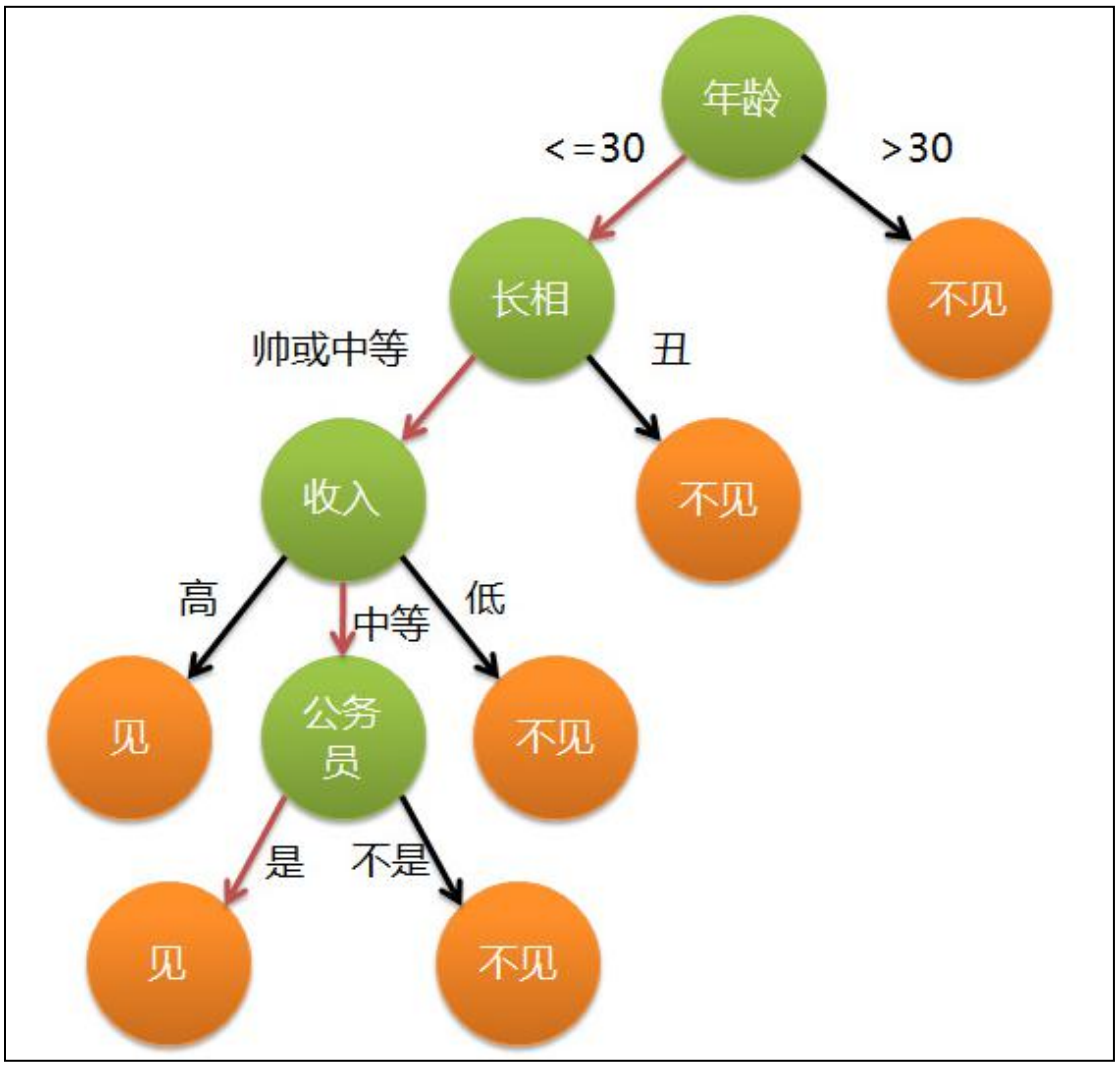
1.2 算法思想

通俗来说，决策树分类的思想类似于找对象。现想象一个女孩的母亲要给这个女孩介绍男朋友，于是有了下面的对话：

女儿：多大年纪了？
母亲：26。
女儿：长的帅不帅？
母亲：挺帅的。
女儿：收入高不？
母亲：不算很高，中等情况。
女儿：是公务员吗？
母亲：是，公务员，在税务局上班呢。
女儿：那好，我去见见。

这个女孩的决策过程就是典型的分类树决策。
实质：通过年龄、长相、收入和是否公务员对将男人分为两个类别：见和不见

假设这个女孩对男人的要求是：30 岁以下、长相中等以上并且是高收入者或中等以上收入的公务员，那么这个可以用下图表示女孩的决策逻辑



上图完整表达了这个女孩决定是否见一个约会对象的策略，其中：

- 绿色节点表示判断条件
- 橙色节点表示决策结果
- 箭头表示在一个判断条件在不同情况下的决策路径

图中红色箭头表示了上面例子中女孩的决策过程。

这幅图基本可以算是一颗决策树，说它“基本可以算”是因为图中的判定条件没有量化，如收入高中低等等，还不能算是严格意义上的决策树，如果将所有条件量化，则就变成真正的决策树了。

决策树分类算法的关键就是根据“先验数据”构造一棵最佳的决策树，用以预测未知数据的类别

决策树：是一个树结构（可以是二叉树或非二叉树）。其每个非叶节点表示一个特征属性上的测试，每个分支代表这个特征属性在某个值域上的输出，而每个叶节点存放一个类别。使用决策树进行决策的过程就是从根节点开始，测试待分类项中相应的特征属性，并按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为决策结果。

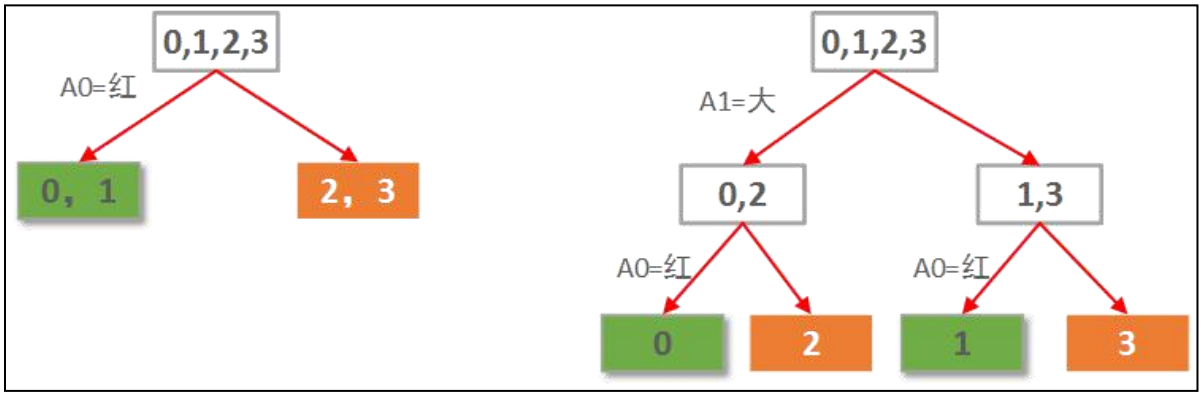
1.3 决策树构造

1.3.1 决策树构造样例

假如有以下判断苹果好坏的数据样本：

样本	红	大	好苹果
0	1	1	1
1	1	0	1
2	0	1	0
3	0	0	0

样本中有 2 个属性，A0 表示是否红苹果。A1 表示是否大苹果。假如要根据这个数据样本构建一棵自动判断苹果好坏的决策树。
由于本例中的数据只有 2 个属性，因此，我们可以穷举所有可能构造出来的决策树，就 2 棵，如下图所示：



显然左边先使用 A0（红色）做划分依据的决策树要优于右边用 A1（大小）做划分依据的决策树。
当然这是直觉的认知。而直觉显然不适合转化成程序的实现，所以需要有一种定量的考察来评价这两棵树的性能好坏。
决策树的评价所用的定量考察方法为计算每种划分情况的信息熵增益：
如果经过某个选定的属性进行数据划分后的信息熵下降最多，则这个划分属性是最优选择

1.3.2 属性划分选择（即构造决策树）的依据

熵：信息论的奠基人香农定义的用来信息量的单位。简单来说，熵就是“无序，混乱”的程度。
通过计算来理解：
1、原始样本数据的熵：
样例总数：4
好苹果：2
坏苹果：2
熵： -(1/2 * log(1/2) + 1/2 * log(1/2)) = 1

信息熵为 1 表示当前处于最混乱，最无序的状态。

2、两颗决策树的划分结果熵增益计算

- 树 1 先选 A0 作划分，各子节点信息熵计算如下：

0，1 叶子节点有 2 个正例，0 个负例。信息熵为：e1 = -(2/2 * log(2/2) + 0/2 * log(0/2)) = 0。

2，3 叶子节点有 0 个正例，2 个负例。信息熵为：e2 = -(0/2 * log(0/2) + 2/2 * log(2/2)) = 0。

因此选择 A0 划分后的信息熵为每个子节点的信息熵所占比重的加权和：E = e1*2/4 + e2*2/4 = 0。

选择 A0 做划分的信息熵增益 G（S, A0）=S - E = 1 - 0 = 1.

事实上，决策树叶子节点表示已经都属于相同类别，因此信息熵一定为 0。

- 树 2 先选 A1 作划分，各子节点信息熵计算如下：

0，2 子节点有 1 个正例，1 个负例。信息熵为：e1 = -(1/2 * log(1/2) + 1/2 * log(1/2)) = 1。

1，3 子节点有 1 个正例，1 个负例。信息熵为：e2 = -(1/2 * log(1/2) + 1/2 * log(1/2)) = 1。

因此选择 A1 划分后的信息熵为每个子节点的信息熵所占比重的加权和：E = e1*2/4 + e2*2/4 = 1。也就是说分了跟没分一样！

选择 A1 做划分的信息熵增益 G（S, A1）=S - E = 1 - 1 = 0.

因此，每次划分之前，我们只需要计算出信息熵增益最大的那种划分即可。

1.4 算法要点

1.4.1、指导思想

经过决策属性的划分后，数据的无序度越来越低，也就是信息熵越来越小

1.4.2 算法实现

梳理出数据中的属性
比较按照某特定属性划分后的数据的信息熵增益，选择信息熵增益最大的那个属性作为第一划分依据，然后继续选择第二属性，以此类推

2. 决策树分类算法 Python 实战

2.1 案例需求

我们的任务就是训练一个决策树分类器，输入身高和体重，分类器能给出这个人是胖子还是瘦子。
所用的训练数据如下，这个数据一共有 10 个样本，每个样本有 2 个属性，分别为身高和体重，第三列为类别标签，表示“胖”或“瘦”。该数据保存在 1.txt 中。

```
1.5 50 thin
1.5 60 fat
1.6 40 thin
1.6 60 fat
1.7 60 thin
1.7 80 fat
1.8 60 thin
1.8 90 fat
1.9 70 thin
1.9 80 fat
```

2.2 模型分析

决策树对于“是非”的二值逻辑的分枝相当自然。而在本数据集中，身高与体重是连续值怎么办呢？

虽然麻烦一点，不过这也不是问题，只需要找到将这些连续值划分为不同区间的中间点，就转换成了二值逻辑问题。

本例决策树的任务是找到身高、体重中的一些临界值，按照大于或者小于这些临界值的逻辑

将其样本两两分类，自顶向下构建决策树。

2.3 python 实现

使用 python 的机器学习库，实现起来相当简单和优雅

```
# -*- coding: utf-8 -*-
import numpy as np
import scipy as sp
from sklearn import tree
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import classification_report
from sklearn.cross_validation import train_test_split


''' 数据读入 '''
data &amp;nbsp; = []
labels = []
with open("d:\\python\\ml\\data\\1.txt") as ifile:
        for line in ifile:
                tokens = line.strip().split(' ')
                data.append([float(tk) for tk in tokens[:-1]])
                labels.append(tokens[-1])
x = np.array(data)
labels = np.array(labels)
y = np.zeros(labels.shape)


''' 标签转换为 0/1 '''
y[labels=='fat']=1

''' 拆分训练数据与测试数据 '''
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2)

''' 使用信息熵作为划分标准，对决策树进行训练 '''
clf = tree.DecisionTreeClassifier(criterion='entropy')
print(clf)
clf.fit(x_train, y_train)

''' 把决策树结构写入文件 '''
with open("tree.dot", 'w') as f:
```

```
        f = tree.export_graphviz(clf, out_file=f)

''' 系数反映每个特征的影响力。越大表示该特征在分类中起到的作用越大 '''
print(clf.feature_importances_)

'''测试结果的打印'''
answer = clf.predict(x_train)
print(x_train)
print(answer)
print(y_train)
print(np.mean( answer == y_train))

'''准确率与召回率'''
precision, recall, thresholds = precision_recall_curve(y_train, clf.predict(x_train))
answer = clf.predict_proba(x)[:,1]
print(classification_report(y, answer, target_names = ['thin', 'fat']))
```

这时候会输出

```
[ 0.2488562   0.7511438]
array([[   1.6,   60. ],
       [   1.7,   60. ],
       [   1.9,   80. ],
       [   1.5,   50. ],
       [   1.6,   40. ],
       [   1.7,   80. ],
       [   1.8,   90. ],
       [   1.5,   60. ]])
array([ 1.,   0.,   1.,   0.,   0.,   1.,   1.,   1.])
array([ 1.,   0.,   1.,   0.,   0.,   1.,   1.,   1.])
1.0

             precision    recall  f1-score   support

       thin       0.83      1.00      0.91         5
        fat       1.00      0.80      0.89         5

avg / total       1.00      1.00      1.00         8

array([ 0.,   1.,   0.,   1.,   0.,   1.,   0.,   1.,   0.,   0.])
array([ 0.,   1.,   0.,   1.,   0.,   1.,   0.,   1.,   0.,   1.])
```

可以看到，对训练过的数据做测试，准确率是 100%。但是最后将所有数据进行测试，会出现 1 个测试样本分类错误。

说明本例的决策树对训练集的规则吸收的很好，但是预测性稍微差点。

2.4 决策树的保存

一棵决策树的学习训练是非常耗费运算时间的，因此，决策树训练出来后，可进行保存，以便在预测新数据时只需要直接加载训练好的决策树即可
本案例的代码中已经决策树的结构写入了 tree.dot 中。打开该文件，很容易画出决策树，还可以看到决策树的更多分类信息。

本例的 tree.dot 如下所示：

```
digraph Tree {
0 [label="X[1] <= 55.0000\nentropy = 0.954434002925\nsamples = 8", shape="box"] ;
1 [label="entropy = 0.0000\nsamples = 2\nvalue = [ 2.    0.]", shape="box"] ;
0 -> 1 ;
2 [label="X[1] <= 70.0000\nentropy = 0.650022421648\nsamples = 6", shape="box"] ;
0 -> 2 ;
3 [label="X[0] <= 1.6500\nentropy = 0.918295834054\nsamples = 3", shape="box"] ;
2 -> 3 ;
4 [label="entropy = 0.0000\nsamples = 2\nvalue = [ 0.    2.]", shape="box"] ;
3 -> 4 ;
5 [label="entropy = 0.0000\nsamples = 1\nvalue = [ 1.    0.]", shape="box"] ;
3 -> 5 ;
6 [label="entropy = 0.0000\nsamples = 3\nvalue = [ 0.    3.]", shape="box"] ;
2 -> 6 ;
}
```

根据这个信息，决策树应该长的如下这个样子：