菊安酱的机器学习第8期

菊安酱的直播间：https://live.bilibili.com/14988341

每周一晚8:00 菊安酱和你不见不散哦~(^o^)/~

更新日期：2018-12-24

作者：菊安酱

课件内容说明：

- 本文为作者原创，转载请注明作者和出处
- 如果想获得此课件及录播视频，可扫描左边二维码，回复"k"进群
- 如果想获得2小时完整版视频，可扫描右边二维码或点击如下链接
- 若有任何疑问，请给作者留言。

交流群二维码

完整版视频及课件

直播视频及课件：http://www.peixun.net/view/1278.html

完整版视频及课件：http://edu.cda.cn/course/966

12期完整版课纲

直播时间：每周一晚8:00

直播内容：

时间	期数	算法
2018/11/05	第1期	k-近邻算法
2018/11/12	第2期	决策树
2018/11/19	第3期	朴素贝叶斯
2018/11/26	第4期	Logistic回归
2018/12/03	第5期	支持向量机
2018/12/10	第6期	AdaBoost 算法
2018/12/17	第7期	线性回归
2018/12/24	第8期	树回归
2018/12/31	第9期	K-均值聚类算法
2019/01/07	第10期	Apriori 算法
2019/01/14	第11期	FP-growth 算法
2019/01/21	第12期	奇异值分解SVD

树回归

上一期，介绍的线性回归包含了一些强大的方法，但这些方法创建的模型需要拟合所有的样本点（局部加权线性回归除外）。当数据集特征很多并且特征之间关系复杂时，构建全局模型就十分困难了，也略显笨拙。而且，现实生活中，很多问题其实都是非线性的，不可能使用全局线性模型来拟合任何数据。

全局线性建模困难？想简单点？那就把数据集切分呗~

这个大而化小的切分思想与SMO算法(序列最小优化)的思想有点类似，我们在讲解SVM的时候有讲解过SMO算法，它的核心思想就是：把难以求解的大优化问题分解成多个易于求解的小优化问题，然后将小优化问题按照一定的顺序求解，结果与整体求解结果完全一致。

把数据集切分成很多份易于建模的数据，然后再用线性模型来建模应该就会容易多了。如果首次切分后仍然难以拟合线性模型，那就继续切分。在这样的切分方式下，树结构和回归法就相当有用了。

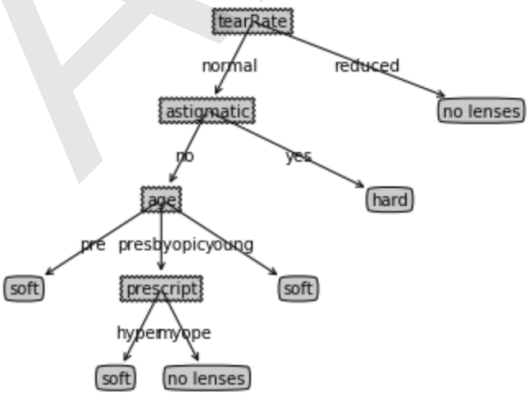
一、回顾决策树（分类）

在第2期，我们讲了如何使用python来构建分类决策树，并利用分类决策树来进行分类。

我们先来回顾一下当时给大家讲的几个的概念：

节点	说明
根节点	没有进边，有出边
中间节点	既有进边也有出边，但进边有且仅有一条，出边也可以有很多条
叶节点	只有进边，没有出边，进边有且仅有一条。**每个叶节点都是一个类别标签**
*父节点和子节点	在两个相连的节点中，更靠近根节点的是父节点，另一个则是子节点。**两者是相对的。**

我们也用了很长的篇幅（6个函数）来讲解决策树的可视化，最后出来的结果是这样的：



当时我们使用的是ID3算法来构建树模型。ID3的做法是：每次选取当前最佳的特征来分割数据，并按照该特征的所有可能取值来切分。也就是说，如果一个特征有4种取值，那么数据将被切分成4份。一旦按某特征切分后，该特征在之后的算法执行过程中将不会再起作用，所以有观点认为这种切分方式过于迅速。

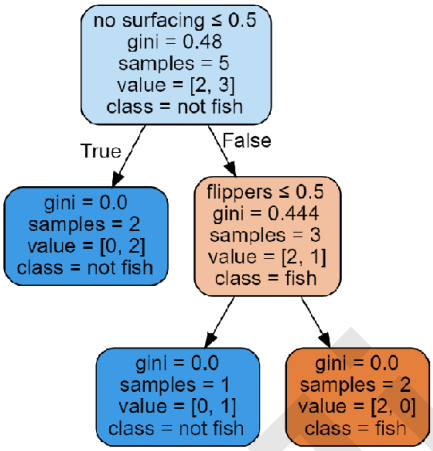
除了切分过于迅速外，ID3算法还存在另一个问题，它不能直接处理连续型特征。只有事先将连续型特征离散化，才能在ID3算法中使用。但这种转换过程会破坏连续型变量的内在特性，也会损失一部分信息。

总结ID3算法缺点：

- 每个特性只能参与一次切分，对后续切分不再起作用
- 不能直接处理连续型特征

构建分类树还有另外一种方法是**二元切分法**，即每次把数据集切成两份。如果数据的某值等于切分所要求的值，那么这些数据就进入树的左子树，反之进入树的右子树。

sklearn就是用这种二元切分法来构建决策树的：



二、CART算法

1. 什么是CART?

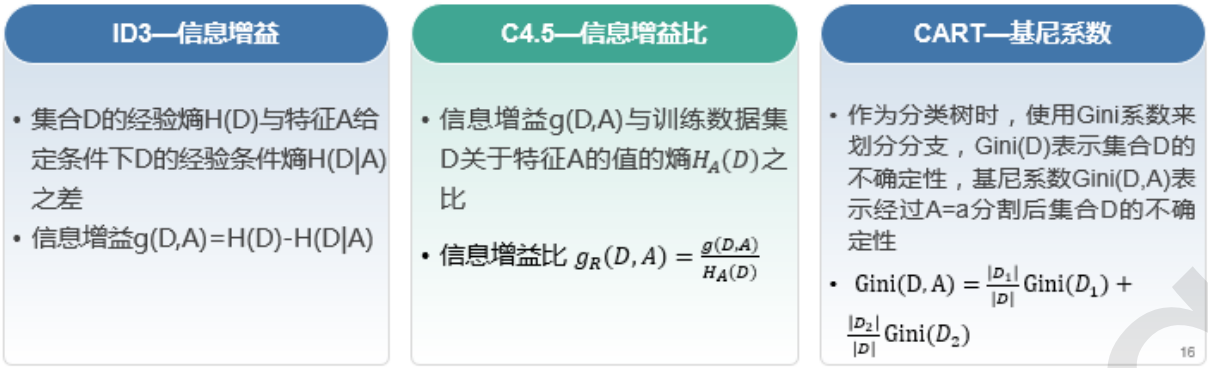
CART是英文**Classification And Regression Tree**的简写，又称为**分类回归树。**从它的名字我们就可以看出，它是一个很强大的算法，既可以用于分类还可以用于回归，所以非常值得我们来学习。

CART算法使用的就是二元切分法，这种方法可以通过调整树的构建过程，使其能够处理连续型变量。具体的处理方法是：如果特征值大于给定值就走左子树，否则就走右子树。

CART算法有两步：

- 决策树生成：递归地构建二叉决策树的过程，基于训练数据集生成决策树，生成的决策树要尽量大；自上而下从根开始建立节点，在每个节点处要选择一个**最好**的属性来分裂，使得子节点中的训练集尽量的纯。
- 决策树剪枝：用验证数据集对已生成的树进行剪枝并选择最优子树，这时损失函数最小作为剪枝的标准。

不同的算法使用不同的指标来定义"**最好**"：



2. 衡量指标

我们先构建一个DataFrame数据集，最后一列为标签。方便后续测试函数。

```
#导入相关包
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt

#构造数据集
def createDataSet():
    group = np.array([[1, 2],
                      [1, 0],
                      [2, 1],
                      [0, 1],
                      [0, 0]])
    labels = np.array([0, 1, 0, 1, 0])
    dataSet = pd.concat([pd.DataFrame(group), pd.DataFrame(labels)],axis=1,
ignore_index=True)
    return dataSet
```

查看数据集：

```
dataSet = createDataSet()
dataSet
```

2.1 误差率计算函数

对于分类树来说，不管数据不纯度的度量方式如何，都是由误差率衍生而来，其计算公式如下：

$$Classification\ error(t)=1-\max_{i=1}[p(i|t)]$$

i为某类别，t为该节点数据集总样本数。

python实现代码如下：

```
def clError(dataSet):
    m = dataSet.shape[0]
    iMax = dataSet.iloc[:, -1].value_counts()[0]
    error = 1 - iMax / m
    return error
```

查看执行结果：

```
clError(dataSet) #0.4
```

2.2 信息熵函数

这里的信息熵，也就是香农熵，计算公式为：

$$Entropy(t) = -\sum_{i=0}^{c-1} p(i|t)log_2 p(i|t)$$

其中c为分类变量总分类数，i为某类别，t为该节点数据集总样本数。这里需要注意的是在信息熵计算过程中令$log_2 0 = 0$,但实际上如果采用DF数据结构并且使用value_counts()函数的话，类别计数为0的结果不会出现在最终统计结果中。在这个式子中$p(i|t)$一定是一个（0,1）之间的数，取对数后一定是一个负数，所以要在式子前面加负号，使其整体变为正数。

python实现如下：

```
def Ent(dataSet):
    m = dataSet.shape[0]
    iSet = dataSet.iloc[:, -1].value_counts()
    p = iSet / m
    ent = (-p * np.log2(p)).sum()
    return ent
```

查看执行结果：

```
Ent(dataSet) #0.97
```

由于原数据集混乱程度较高，因此信息熵趋近于1。

2.3 Gini指数

Gini指数主要用于CART树的纯度判定，计算公式如下：

$$Gini = 1 - \sum_{i=0}^{c-1} [p(i|t)]^2$$

其中c为分类变量总分类数，i为某类别，t为该节点数据集总样本数。

python代码实现如下：

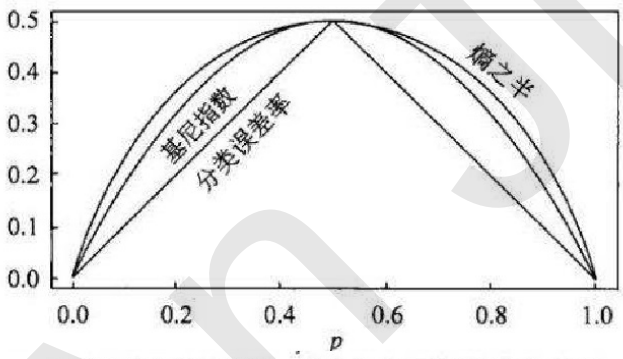
```
def Gini(dataSet):
    m = dataSet.shape[0]
    iSet = dataSet.iloc[:, -1].value_counts()
    p = iSet / m
    gini = 1 - (np.power(p, 2)).sum()
    return gini
```

运行结果如下：

```
Gini(dataSet) #0.48
```

同样由于原始数据集混乱程度比较高，因此gini指数趋近于0.5。

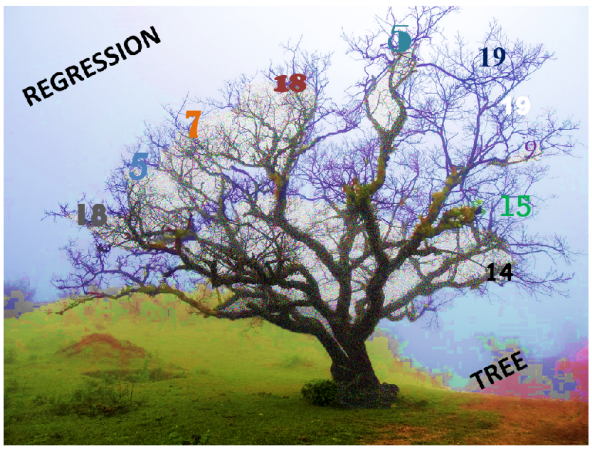
能够看出，三种方法本质上都相同，在类分布均衡时（即当p=0.5时）达到最大值，而当所有记录都属于同一个类时（p等于1或0）达到最小值。换而言之，在纯度较高时三个指数均较低，而当纯度较低时，三个指数都比较大，且可以计算得出，熵在0-1区间内分布，而Gini指数和分类误差均在0-0.5区间内分布，三个指数随某变量占比增加而变化的曲线如下所示：



二类分类中基尼指数、熵之半和分类误差率的关系

、

三、CART回归树的python实现



CART树的构建过程：首先找到最佳的列来切分数据集，每次都执行二元切分法，如果特征值大于给定值就走左子树，否则就走右子树，当节点不能再分时就将该节点保存为叶节点。

这里需要大家思考：

什么样的切分方式才叫最佳？也就是衡量'最佳'的指标什么？

叶节点里面存放的是什么？

在这里，我们使用的数据集储存在 ex00.txt 这样一份文本文件中，先将数据集导进来，方便后续使用。

导入数据集，并查看数据分布

```
ex00= pd.read_table('ex00.txt',header=None)
plt.scatter(ex00.iloc[:,0].values,ex00.iloc[:,1].values);
```

1. 找到最佳切分列

如何使用CART算法选择最佳切分特征？我们先来看一下寻找最佳切分列函数的伪代码：

```
对每个特征：
    对每个特征值：
        将数据切分成两份（辅助函数1）
        计算切分的误差（辅助函数2）
        如果当前误差小于最小误差，则将当前切分设定为最佳切分并更新最小误差
 返回最佳切分的特征和阈值
```

这里需要两个辅助函数，我们先来构建辅助函数

辅助函数1：切分数据集函数

在给定特征和特征值的情况下，该函数通过数组过滤的方式将数据集切分得到两个子集并返回。

这里需要注意的是，我们的数据接口使用的是DataFrame的数据形式，所以每次切分完数据后，要更新数据集的索引。更新索引，更新索引，更新索引！！！重要的事情要说三遍~

```
"""
函数说明:根据特征切分数据集合
参数说明:
    dataSet：原始数据集
    feature：待切分的特征索引
    value：该特征的值
返回:
    mat0：切分的数据集合0
    mat1：切分的数据集合1
"""
def binSplitDataSet(dataSet, feature, value):
    mat0 = dataSet.loc[dataSet.iloc[:,feature] > value,:]
    mat0.index = range(mat0.shape[0])
    mat1 = dataSet.loc[dataSet.iloc[:,feature] <= value,:]
```

```
    mat1.index = range(mat1.shape[0])
    return mat0, mat1
```

使用上面构造的的简单数据集验证函数运行效果：

```
mat0, mat1 = binSplitDataSet(dataSet, 0, 1)
```

辅助函数2：计算切分误差函数

该函数的功能是，在给定数据集上计算目标变量的平方误差。这里使用均方差函数var()来计算目标变量的均方误差，由于我们需要的是总方差，所以要用均方误差乘以样本个数。

```
#计算总方差：均方差*样本数
def errType(dataSet):
    var= dataSet.iloc[:,-1].var() *dataSet.shape[0]
    return var
```

用ex00数据集测试函数：

```
errType(ex00)
```

辅助函数3：生成叶节点函数

当我们的最佳切分函数确定不再对数据进行切分时，将调用该函数来得到叶节点的模型。在回归树中，该模型其实就是目标变量的均值。

```
#生成叶节点
def leafType(dataSet):
    leaf = dataSet.iloc[:,-1].mean()
    return leaf
```

同样用ex00数据集测试函数：

```
leafType(ex00)
```

我们的辅助函数都构建好了，然后我们就可以来构建我们的主函数——最佳寻找最佳切分列函数

```
"""
函数说明:找到数据的最佳二元切分方式函数
参数说明:
    dataSet：原始数据集
    leafType：生成叶结点函数
    errType：误差估计函数
    ops：用户定义的参数构成的元组
返回:
    bestIndex：最佳切分特征
```

```
    bestValue: 最佳特征值
"""
def chooseBestSplit(dataSet, leafType=leafType, errType=errType, ops = (1,4)):
    #tolS允许的误差下降值,tolN切分的最少样本数
    tolS = ops[0]; tolN = ops[1]
    #如果当前所有值相等,则退出。(根据set的特性)
    if len(set(dataSet.iloc[:,-1].values)) == 1:
        return None, leafType(dataSet)
    #统计数据集合的行m和列n
    m, n = dataSet.shape
    #默认最后一个特征为最佳切分特征,计算其误差估计
    S = errType(dataSet)
    #分别为最佳误差,最佳特征切分的索引值,最佳特征值
    bestS = np.inf; bestIndex = 0; bestValue = 0
    #遍历所有特征列
    for featIndex in range(n - 1):
        colval= set(dataSet.iloc[:,featIndex].values)
        #遍历所有特征值
        for splitVal in colval:
            #根据特征和特征值切分数据集
            mat0, mat1 = binSplitDataSet(dataSet, featIndex, splitVal)
            #如果数据少于tolN,则退出
            if (mat0.shape[0] < tolN) or (mat1.shape[0] < tolN): continue
            #计算误差估计
            newS = errType(mat0) + errType(mat1)
            #如果误差估计更小,则更新特征索引值和特征值
            if newS < bestS:
                bestIndex = featIndex
                bestValue = splitVal
                bestS = newS
    #如果误差减少不大则退出
    if (S - bestS) < tolS:
        return None, leafType(dataSet)
    #根据最佳的切分特征和特征值切分数据集合
    mat0, mat1 = binSplitDataSet(dataSet, bestIndex, bestValue)
    #如果切分出的数据集很小则退出
    if (mat0.shape[0] < tolN) or (mat1.shape[0] < tolN):
        return None, leafType(dataSet)
    #返回最佳切分特征和特征值
    return bestIndex, bestValue
```

运行函数，查看结果：

```
chooseBestSplit(exOO)
```

2. CART算法实现代码

创建函数 createTree() 的伪代码如下：

```
找到最佳的待切分特征：
    如果该节点不能再分，将该节点保存为叶节点
    执行二元切分
    在右子树调用createTree() 方法
    在左子树调用createTree() 方法
```

```
"""
函数功能:树构建函数
参数说明:
    dataSet：原始数据集
    leafType：建立叶结点的函数
    errType：误差计算函数
    ops：包含树构建所有其他参数的元组
返回:
    retTree：构建的回归树
"""
def createTree(dataSet, leafType = leafType, errType = errType, ops = (1, 4)):
    #选择最佳切分特征和特征值
    col, value = chooseBestSplit(dataSet, leafType, errType, ops)
    #如果没有特征,则返回特征值
    if col == None: return value
    #回归树
    retTree = {}
    retTree['spInd'] = col
    retTree['spVal'] = value
    #分成左数据集和右数据集
    lSet, rSet = binSplitDataSet(dataSet, col, value)
    #创建左子树和右子树
    retTree['left'] = createTree(lSet, leafType, errType, ops)
    retTree['right'] = createTree(rSet, leafType, errType, ops)
    return retTree
```

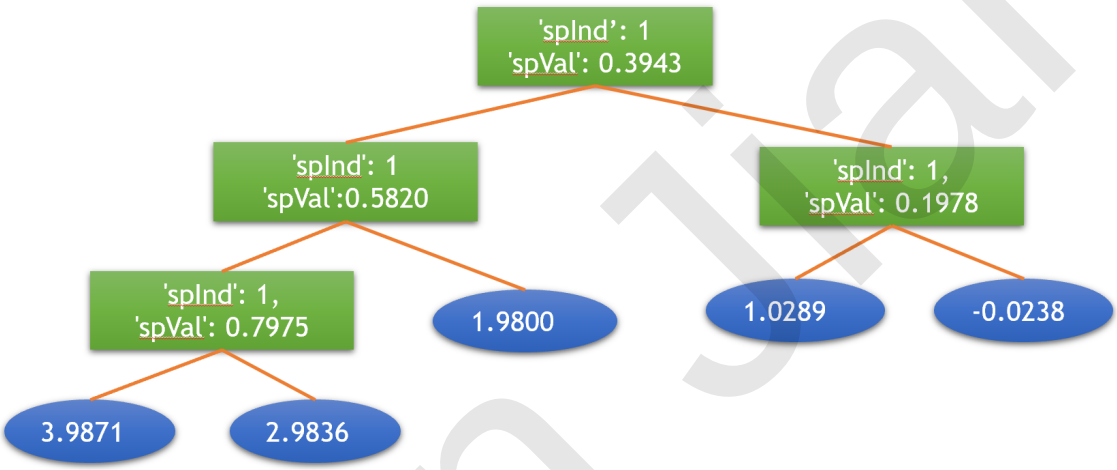
运行函数，查看函数返回结果：

```
createTree(ex00)
```

从结果中可以看出，这个树只分了一次就结束了。那是因为我们使用的这个ex00数据集比较简单，它只有一个特征。我们换一个稍微复杂点的数据集，再跑一遍看看结果。

```
#导入数据集
ex0 = pd.read_table('ex0.txt',header=None)
ex0.head()
ex0.shape
ex0.describe()

#数据可视化
plt.scatter(ex0.iloc[:,1].values,ex0.iloc[:,2].values);

#创建回归树
ex0tree = createTree(ex0,ops = (1, 4))
ex0tree
```

从结果中可以看出，这里生成了5个叶节点。



这里只是简单将字典形式的树可视化出来，帮助大家理解。实际上回归树的形态不是这样子的。下面我们一起来看一下SKlearn实现的回归树是怎样的。

四、回归树的SKlearn实现

前面讲了回归树的建模以及预测等一系列过程，那么最后形成的结果到底是怎么样的呢？

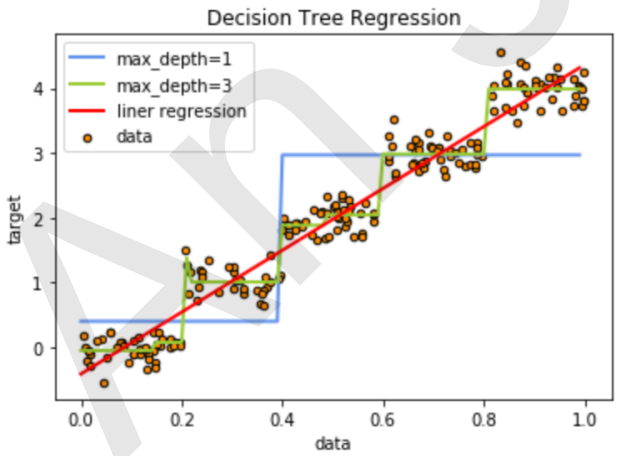
接下来，用SKlearn调库来给大家实现回归树并做可视化，让大家能够更直观的感受到回归树。

Sklearn调库实现回归树：

```
from sklearn.tree import DecisionTreeRegressor
from sklearn import linear_model

#用来训练的数据
x = (ex0.iloc[:,1].values).reshape(-1,1)
y = (ex0.iloc[:,-1].values).reshape(-1,1)

# 训练模型
model1 = DecisionTreeRegressor(max_depth=1)
model2 = DecisionTreeRegressor(max_depth=3)
```

```
model3 = linear_model.LinearRegression()
model1.fit(x, y)
model2.fit(x, y)
model3.fit(x, y)

# 预测
X_test = np.arange(0, 1, 0.01)[:, np.newaxis]
y_1 = model1.predict(X_test)
y_2 = model2.predict(X_test)
y_3 = model3.predict(X_test)

# 可视化结果
plt.figure()
plt.scatter(x, y, s=20, edgecolor="black",c="darkorange", label="data")
plt.plot(X_test, y_1, color="cornflowerblue",label="max_depth=1", linewidth=2)
plt.plot(X_test, y_2, color="yellowgreen", label="max_depth=3", linewidth=2)
plt.plot(X_test, y_3, color='red', label='liner regression', linewidth=2)
plt.xlabel("data")
plt.ylabel("target")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

运行结果如下：



从可视化图中可以看出，线性回归只能捕捉线性趋势，数据中其他微小的波动或者规律很难捕捉到，但树回归就可以做到。树回归也有自己的缺陷，如果不进行剪枝的话，树很容易过拟合。

五、树剪枝

一棵树如果节点过多，表明该模型很有可能“过拟合”了。通过降低决策树的复杂度来避免过拟合的过程称为**剪枝**（pruning）。剪枝的方法有**预剪枝**（prepruning）和**后剪枝**（postpruning）。

1. 预剪枝

预剪枝就是通过调节参数tolS和tolN修改终止条件来达到剪枝的目的。下面我们调节这两个参数，来看看树有何变化

```
for i in range(2):
    for j in range(25):
        tree = createTree(ex0,ops = (i, j))
        print('-'*10,f'容差tolS={i},',f'最少样本数tolN={j}','-'*60)
        print(tree)
```

从运行就结果中可以看出，当tolS和tolN都为0的时候，生成了一棵非常臃肿的树，当tolS达到1的时候，树才收敛至5个叶节点，而当最少样本数tolN达到22的时候，树才能收敛至5个叶节点。当然这个范围也是我们不断调整才得到的。

下面我们换一个数据集再次进行测试

```
#导入数据集
ex2 = pd.read_table('ex2.txt',header=None)
ex2.head()
ex2.shape
ex2.describe()

#可视化
plt.scatter(ex2.iloc[:,0].values,ex2.iloc[:,1].values);

#绘制树
ex2tree = createTree(ex2,ops = (1, 4))
ex2tree

#探索tolS
for i  in np.arange(0,5000,500):
    ex2tree = createTree(ex2,ops = (i, 4))
    print('-'*10,f'容差tolS={i}','-'*60)
    print(ex2tree)

#探索tolN
for j in np.arange(0,100,10):
    ex2tree = createTree(ex2,ops = (3000, j))
    print('-'*10,f'最少样本数tolN={j}','-'*60)
    print(ex2tree)
```

从散点图中，大家可以看到这个ex2数据集和ex00数据集非常相似，ex00数据集构建出来的树只有两个叶节点，但是用同样的参数（ops = (1, 4)），ex2数据集构建出来的新树则有很多的叶节点。为什么会出现这样的情况呢？大家仔细观察这两个数据集，可以发现对于y，ex2的数量级是ex00的100倍。

从上述探索过程可以得出两个结论：

1. 相比于最小样本数tolN，容差tolS的剪枝效果更明显
2. 容差tolS对误差的数量级十分敏感

然而，这种通过不断修改停止条件来得到合理结果并不是很好的办法。首先，调参的过程非常耗时间，其次，在实际应用中，我们常常不确定到底需要寻找什么样的结果。

2. 后剪枝

使用后剪枝的方法需要将数据集分为训练集和测试集。首先，我们根据训练集数据构建出一棵足够庞大的树，然后从上而下找到叶节点，用测试集来判断将这些叶节点合并是否能降低测试误差，如果是的话就合并。

剪枝的伪代码如下：

```
基于已有的树切分测试数据：
    如果存在任一子集是一棵树，则在该子集递归剪枝
    计算将当前两个叶节点合并后的误差
    计算不合并的误差
    如果合并会降低误差的话，就将叶节点合并
```

切分训练集和测试集：

```
#训练集
train = ex2.iloc[:140,:]
train.shape
#测试集
test = ex2.iloc[140:,:]
test.index = range(test.shape[0])
test.shape

#可视化
plt.scatter(train.iloc[:,0].values,train.iloc[:,1].values);
plt.scatter(test.iloc[:,0].values,test.iloc[:,1].values);
```

辅助函数1：该函数主要用来判断当前节点是否是叶节点。前面我们用将树放在字典里面，如果返回的类型是dict，则说明该节点不是叶节点。

```
"""
函数功能：测试输入变量是否是字典类型，返回布尔类型的结果
"""
def isTree(obj):
    return type(obj).__name__=='dict'
```

辅助函数2：该函数是一个递归函数，从上到下遍历树直到叶节点为止，如果找到两个叶节点，则计算它们的平均值。该函数对树进行了塌陷处理（即返回树平均值）。

```
def getMean(tree):
    if isTree(tree['right']):
        tree['right']=getMean(tree['right'])
    if isTree(tree['left']):
        tree['left']=getMean(tree['left'])
    mean = (tree['right']+tree['left'])/2.0
    return mean
```

主函数：回归树剪枝函数

```
def prune(tree,testData):
    #如果没有测试数据，则对树进行塌陷处理
    if testData.shape[0]==0:
        return getMean(tree)
    #递归调用函数prune()对测试集进行切分
    if (isTree(tree['right']) or isTree(tree['left'])):
        lSet,rSet = binSplitDataSet(testData,tree['spInd'],tree['spVal'])
    #对左子树进行剪枝
    if isTree(tree['left']):
        tree['left'] = prune(tree['left'],lSet)
    #对右子树进行剪枝
    if isTree(tree['right']):
        tree['right'] = prune(tree['right'],rSet)
    #对叶节点进行合并
    if not isTree(tree['left']) and not isTree(tree['right']):
        lSet,rSet = binSplitDataSet(testData,tree['spInd'],tree['spVal'])
        errorNoMerge = sum(np.power(lSet.iloc[:,-1]-
tree['left'],2))+sum(np.power(rSet.iloc[:,-1]-tree['right'],2))
        treeMean = (tree['left']+tree['right'])/2.0
        errorMerge = sum((testData.iloc[:,-1]-treeMean)**2)
        if errorMerge < errorNoMerge:
            print('Merging')
            return treeMean
        else: return tree
    else: return tree
```

测试函数，查看运行结果：

```
tree = createTree(train,ops = (0, 1))
prune(tree,test)
```

根据结果可以看到，有一些节点被合并了（即剪枝），但是并没有出现预期的那样剪枝成两部分，这说明后剪枝可能不如预剪枝有效。有时候，为了寻求最佳模型，可以同时使用预剪枝和后剪枝两种剪枝技术。

六、模型树

用树来对数据进行建模，除了把叶节点简单地设定成常数值之外，还有一种方法是把叶节点设定为分段线性函数，这里的**分段线性函数**（piecewise linear）指的是模型由多个线性片段组成。

在前面讲的回归树中，每个叶节点中包含的是单个值；下面我们要讲的这种模型树，每个叶节点中包含的就是一个线性方程。

这里我们以一个简单数据集为例，帮助我们建模。数据集存放在exp2.txt文件中。

```
#导入数据集
exp2 = pd.read_table('exp2.txt',header=None)
exp2.describe()

#可视化，探索数据分布
plt.scatter(exp2.iloc[:,0],exp2.iloc[:,1]);
```

从运行结果可以看出，该数据集由两部分组成，建立模型树的时候可以分成两段线性函数。

1. 模型树叶节点生成函数

函数1：将数据集格式化成特征矩阵X和标签矩阵Y，并计算回归系数

```
"""
函数功能：计算特征矩阵、标签矩阵、回归系数
参数说明：
    dataSet：原始数据集
返回：
    ws：回归系数
    X：特征矩阵（第一列增加x0=1）
    Y：标签矩阵
"""
def linearSolve(dataSet):
    m,n = dataSet.shape
    con = pd.DataFrame(np.ones((m,1)))
    conX = pd.concat([con,dataSet.iloc[:,:-1]],axis=1,ignore_index=True)
    X = np.mat(conX)
    Y = np.mat(dataSet.iloc[:,-1].values).T
    xTx = X.T*X
    if np.linalg.det(xTx) == 0:
        raise NameError('奇异矩阵无法求逆，请尝试增大tolN,即ops第二个值')
    ws = xTx.I*(X.T*Y)
    return ws,X,Y
```

运行函数，查看结果：

```
ws,X,Y = linearSolve(exp2)
ws
X
Y
```

函数2：生成模型树的叶节点（即线性方程），这里返回的是回归系数

```
def modelLeaf(dataSet):
    ws,X,Y = linearSolve(dataSet)
    return ws
```

函数3：计算给定数据集的误差（误差平方和）

```
def modelErr(dataSet):
    ws,X,Y = linearSolve(dataSet)
    yHat = X*ws
    err = sum(np.power(Y-yHat,2))
    return err
```

2. 模型树实现

调用createTree()函数，构建模型树，将生成叶节点函数和计算误差函数全部换成模型树的辅助函数即可：

```
createTree(exp2,modelLeaf,modelErr,(1, 10))
```

从运行结果中可以看出，该模型树以0.285477为界，生成了两个线性模型：$y = 0.00169 + 11.96x$ 和 $y = 3.46 + 1.18x$ 。该数据集实际是由模型$y = 0 + 12x$ 和 $y = 3.5 + x$ 再加上高斯噪声生成的。可以看出我们生成的模型树与数据集的真实模型是非常接近的。

七、用树回归进行预测

我们前面生成的回归树主要是用来预测的，下面我们尝试建立树回归的预测模型。

1. 导入数据集

此处我们使用的数据集涉及人的智力水平和自行车速度的关系，当然此数据集纯属虚构啦。数据分为训练集和测试集分别保存在bikeSpeedVsIq_train.txt 和 bikeSpeedVsIq_test.txt 两个文件中。

```
#导入训练集
biketrain = pd.read_table('bikeSpeedVsIq_train.txt',header=None)
biketrain.head()
#探索训练集
biketrain.shape
biketrain.describe()
plt.scatter(biketrain.iloc[:,0],biketrain.iloc[:,1]);

#导入测试集
biketest = pd.read_table('bikeSpeedVsIq_test.txt',header=None)
biketest.head()
#探索测试集
biketest.shape
biketest.describe()
plt.scatter(biketest.iloc[:,0],biketest.iloc[:,1]);
```

从上述运行结果中可以看出，数据集大致分为两段。

2. 构建预测函数的辅助函数

函数1：回归树叶节点预测函数，由于回归树的叶节点中是均值，所以可以直接返回该值。为了与下面模型树的预测函数保持一致，这里仍保留两个输入参数，虽然我们只用了一个参数。

```
def regTreeEval(model,inData):
    return model
```

函数2：模型树叶节点预测函数。

```
"""
函数功能：返回模型树的叶节点预测结果
参数说明：
    model：模型树的叶节点，即回归系数
    inData：不带标签列的单个测试数据
"""
def modelTreeEval(model,inData):
    n = len(inData)
    X = np.mat(np.ones((1,n+1)))  #增加一列常数项x0=1,放在第一列
    X[:,1:n+1] = inData
    return X*model
```

函数3：自顶向下遍历整棵树，直到找到叶节点，然后返回单条测试数据的预测结果

```
"""
函数功能：返回单个测试数据的预测结果
参数说明：
    tree：字典形式的树
    inData：单条测试数据
    modelEval：叶节点预测函数
"""
def treeForeCast(tree,inData,modelEval = regTreeEval):
    #先判断是不是叶节点，如果是叶节点直接返回预测结果
    if not isTree(tree):
        return modelEval(tree,inData)
    #根据索引找到左右子树
    if inData[tree['spInd']] > tree['spVal']:
        #如果左子树不是叶节点，则递归找到叶节点
        if isTree(tree['left']):
            return treeForeCast(tree['left'],inData,modelEval)
        else:
            return modelEval(tree['left'],inData)
    else:
        if isTree(tree['right']):
            return treeForeCast(tree['right'],inData,modelEval)
        else:
            return modelEval(tree['right'],inData)
```

函数4：返回整个测试集的预测结果

```
"""
函数功能：返回整个测试集的预测结果
参数说明：
    tree:字典形式的树
    testData:测试集
    modelEval：叶节点预测函数
返回：
    yHat:每条数据的预测结果
"""
def createForeCast(tree, testData, modelEval = regTreeEval):
```

```
    m = testData.shape[0]
    yHat = np.mat(np.zeros((m,1)))
    for i in range(m):
        inData = testData.iloc[i,:-1].values
        yHat[i,0]= treeForeCast(tree,inData,modelEval)
    return yHat
```

3. 回归树的预测结果

先初步设定一个ops值来创建一棵回归树，并进行预测以及计算其相关系数R^2 。

```
#创建回归树
regTree = createTree(biketrain,ops=(1,20))
regTree

#回归树预测结果
yHat = createForeCast(regTree,biketest, regTreeEval)
yHat

#计算相关系数R2
np.corrcoef(yHat.T,biketest.iloc[:,-1].values)[0,1]

#计算均方误差SSE
sum((yHat.A.flatten()-biketest.iloc[:,-1].values)**2)
```

改变ops的值，找到最大的相关系数R^2 和最小的SSE。

```
tolS = []
tolN = []
R2 = []
SSE = []
for i in range(5):
    for j in np.arange(1,100,10):
        regtree = createTree(biketrain,ops=(i,j))
        yHat = createForeCast(regtree,biketest,regTreeEval)
        r2 = np.corrcoef(yHat.T,biketest.iloc[:,-1].values)[0,1]
        sse = sum((yHat.A.flatten()-biketest.iloc[:,-1].values)**2)
        tolS.append(i)
        tolN.append(j)
        R2.append(r2)
        SSE.append(sse)
df = pd.DataFrame([tolS,tolN,R2,SSE],index=['tolS','tolN','R2','SSE']).T

df.head()

#找到最大的相关系数R2和最小的SSE
df.loc[df['R2']==df['R2'].max(),:]
```

从运行结果可知，回归树的最大相关系数为0.977，最小均方误差为19648。

4. 模型树的预测结果

同样，先初步设定一个ops值创建一棵模型树，然后进行预测并计算相关系数R^2 。

```
#创建模型树
modelTree = createTree(biketrain, modelLeaf, modelErr, ops=(1,20))
modelTree

#模型树预测结果
yHat1 = createForeCast( modelTree, biketest, modelTreeEval)
yHat1

#计算相关系数R2
np.corrcoef(yHat1.T,biketest.iloc[:,-1].values)[0,1]

#计算均方误差SSE
sum((yHat1.A.flatten()-biketest.iloc[:,-1].values)**2)
```

改变ops值，找到模型树的最大相关系数R^2 和最小的SSE .

```
tolS_1 = []
tolN_1 = []
R2_1 = []
SSE_1 = []
for i in range(5):
    #此处j≤16，则矩阵为奇异矩阵
    for j in np.arange(20,100,10):
        modeltree = createTree(biketrain,modelLeaf, modelErr,ops=(i,j))
        yHat_1 = createForeCast(modeltree,biketest,modelTreeEval)
        r2_1 = np.corrcoef(yHat_1.T,biketest.iloc[:,-1].values)[0,1]
        sse_1 = sum((yHat_1.A.flatten()-biketest.iloc[:,-1].values)**2)
        tolS_1.append(i)
        tolN_1.append(j)
        R2_1.append(r2_1)
        SSE_1.append(sse_1)
df1 = pd.DataFrame([tolS_1,tolN_1,R2_1,SSE_1],index=['tolS','tolN','R2','SSE']).T

df1.head()

#找到最大相关系数R2和最小的SSE
df1.loc[df1['R2']==df1['R2'].max(),:]
```

从运行结果可以看出，模型树的最大相关系数为0.976，最小的SSE为21234。

5. 标准线性回归的预测结果

```
#标准线性回归
ws,X,Y = linearSolve(biketrain)
ws

#在第一列增加常数项1,构建特征矩阵
testX = pd.concat([pd.DataFrame(np.ones((biketest.shape[0],1))),biketest.iloc[:,:-1]],
                  axis=1,ignore_index = True)
testMat = np.mat(testX)
testMat

#标准线性回归预测结果
yHat_2 = testMat*ws

#相关系数R2
R2_2 = np.corrcoef(yHat_2.T,biketest.iloc[:,-1].values)[0,1]
R2_2

#均方误差SSE
SSE_2 = sum((yHat_2.A.flatten()-biketest.iloc[:,-1].values)**2)
SSE_2
```

从运行结果可以看出，标准线性回归的相关系数R2为0.943，最小SSE为50727。

从三种模型的预测结果来看，树回归的两种模型会比标准线性回归的效果要好一些。

八、使用python的Tkinter库创建GUI

GUI（Graphical User Interface）就是**图形用户界面**，它能够同时支持数据呈现和用户交互。

1. 用Tkinter创建GUI

python有很多GUI框架，其中一个易于使用的Tkinter是随python的标准编译版本发布的。Tkinter可以在Windows、Mac OS和大多数的Linux平台上使用。

我们先从最简单的Hello World开始。

```
from tkinter import * #注意：在python3中使用的库名为 tkinter,即首写字母 T 为小写

root = Tk()                                 #实例化
myLabel = Label(root,text='hello world') #设置Label部件
myLabel.grid()                              #布局管理器
root.mainloop()                             #启动事件循环
```

这里需要说明的是，.grid()方法是一种布局管理器，这种方法是把部件安排在一个二维的网格中，用户可以设定每个部件所在的行和列，如果不做任何设定的话，默认显示在0行0列，也可以设定rowspan 和 columnspan 来实现部件的跨行和跨列。

其实，Tkinter 的GUI是由一些小部件(Widget)组成。所谓小部件，指的是文本框(TextBox)、按钮(Button)、标签(Label)、复选按钮(CheckButton)和按钮整数值(IntVar)等对象。目前有15种Tkinter的部件：http://www.runoob.com/python/python-gui-tkinter.html

2. 集成Matplotlib和Tkinter创建GUI

下面我们尝试集成Matplotlib和Tkinter来创建GUI

步骤一：导入相应的包

```
import pandas as pd
import numpy as np
import matplotlib
#导入渲染器，功能是执行绘画等动作
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
#导入画布
from matplotlib.figure import Figure
```

步骤二：创建画布，并根据用户输入值绘制树

该函数假定用户输入值都是合法的，首先创建一个画布，然后清空画布，使得前后两个图像不会重叠。然后初始化渲染器，该渲染器的作用主要是用来执行绘画等动作，.get_tk_widget()用来返回实现绘画的Tk小部件。接下来，重新创建一个子画布（由于前面清空了整个画布，图像的各个子图也会被清除）。根据用户是否点击复选框来判定执行回归树还是模型树，并进行预测。最后将原始数据集绘制成散点图，将预测结果绘制成连续曲线图。

```
#绘制树
def reDraw(tolS,tolN):
    reDraw.f = Figure(figsize = (5,4),dpi= 100)   #创建画布
    reDraw.f.clf()  #清空画布
    reDraw.canvas = FigureCanvasTkAgg(reDraw.f,master = root) #初始化渲染器
    reDraw.canvas.get_tk_widget().grid(row=0,columnspan=3) #返回用于实现FigureCanvasTkAgg
的Tk小部件
    reDraw.a = reDraw.f.add_subplot(111) #创建子画布
    if chkBtnVar.get():
        if tolN<2: tolN=2  #我们使用的数据集tolN<2时，矩阵是奇异矩阵
        myTree = createTree(reDraw.rawDat,modelLeaf,modelErr,ops = (tolS,tolN))
        yHat = createForeCast(myTree,reDraw.testDat,modelTreeEval)
    else:
        myTree = createTree(reDraw.rawDat,ops=(tolS,tolN))
        yHat = createForeCast(myTree,reDraw.testDat)

 reDraw.a.scatter(reDraw.rawDat.iloc[:,0],reDraw.rawDat.iloc[:,1],s=5,c='darkorange')
    reDraw.a.plot(reDraw.testDat.iloc[:,0],yHat,linewidth=2.0,c = 'yellowgreen')
    reDraw.canvas.draw()
```

步骤三：获取用户输入值。

该函数主要作用是尝试理解用户的输入并防止程序崩溃。其中tolS代表的是容差，期望的输入是浮点数，而tolN代表的是叶节点最小样本数，期望输入的是整数。为了得到用户输入的文本，在Entry部件上调用 .get() 方法 ，然后尝试将其转换成期望的数据类型，如果转换失败则清除用户输入，并重新输入默认值。这一做法虽然会比较耗费时间，但是对于用户体验来说是必不可少的。

```
#获取用户输入值
def getInputs():
    try:
        tolS = float(tolSentry.get())
    except:
        tolS = 1.0
        print('请输入浮点数')
        tolSentry.delete(0,END)
        tolSentry.insert(0,'1.0')
    try:
        tolN = int(tolNentry.get())
    except:
        tolN = 10
        print('请输入整数')
        tolNentry.delete(0,END)
        tolNentry.insert(0,'10')
    return tolS,tolN
```

步骤四：利用用户输入值绘制树

该函数实现了两个功能：第一，调用getInputs()函数捕捉用户输入值，第二，根据输入值调用reDraw()函数绘制树模型。

```
#利用用户输入值绘制树
def drawNewTree():
    tolS,tolN = getInputs()
    reDraw(tolS,tolN)
```

步骤五：构建GUI

```
#实例化一个窗口对象
root = Tk()
#窗口的标题
root.title('回归树调参')
#tolS
Label(root,text = 'tolS').grid(row = 1, column = 0)
tolSentry = Entry(root)
tolSentry.grid(row=1, column = 1)
tolSentry.insert(0,'1.0') #默认值为1.0
#tolN
Label(root,text = 'tolN').grid(row = 2, column = 0)
tolNentry = Entry(root) #Entry：单行文本输入框
tolNentry.grid(row=2, column = 1)
tolNentry.insert(0,'10') #默认值为10
#按钮
Button(root,text = 'ReDraw',command = drawNewTree).grid(row = 1, column = 2,rowspan =3)
#按钮整数值
chkBtnVar = IntVar()
#复选按钮
```

```
chkBtn = Checkbutton(root,text = 'Model Tree',variable = chkBtnVar)
chkBtn.grid(row = 3, column = 0,columnspan = 2)
#导入数据
dataSet = pd.read_table('sine.txt',header=None)
train = dataSet[:160]
test = dataSet[160:].sort_values(by=dataSet[160:].columns[0]) #按特征从小到大排序
test.index = range(test.shape[0]) #更新索引
reDraw.rawDat = train
reDraw.testDat = test

reDraw(1.0,10)

root.mainloop()
```

用户可以在这个GUI中改变参数来实现不同树回归的绘制并及时看到可视化结果。这里只是简单的实现了一个GUI，Matplotlib 和 Tkinter 的集成可以构建出更强大的GUI，大家可以以更自然的方式来探索机器学习算法的奥妙。

其他

- 菊安酱的直播间：https://live.bilibili.com/14988341
- 下周一（2018/12/31）将讲解**Kmeans聚类算法**，欢迎各位进入菊安酱的直播间观看直播
- 如有问题，可以给我留言哦~