作业6：决策树与集成学习

1. Bagging

1.假设所有分类器的误差均值为零，而且互不相关，请证明：$E_{h_B} = \frac{1}{m}E_h$

已知E_h的表达式为：

$$E_h = \frac{1}{m}\frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{n}\epsilon_i(x_j)^2$$

E_{h_B}可表示为：

$$\begin{aligned}
E_{h_B} =& \frac{1}{n}[\sum_{j=1}^{n}\epsilon_B(x_j)]^2 \\
=& \frac{1}{n}\sum_{j=1}^{n}[h_B(x_j) - y(x_j)]^2 \\
=& \frac{1}{n}\sum_{j=1}^{n}[\frac{1}{m}\sum_{i=1}^{m}h_i(x_j) - \frac{1}{m}\cdot m \cdot y(x_j)]^2 \\
=& \frac{1}{n}\sum_{j=1}^{n}[\frac{1}{m}\sum_{i=1}^{m}(h_i(x_j) - y(x_j))]^2 \\
=& \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}[\sum_{i=1}^{m}\epsilon_i(x_j)]^2 \\
=& \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}\sum_{i=1}^{m}\epsilon_i(x_j)^2 + \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}\sum_{k_1=1}^{m}\sum_{k_2=1,k_1\neq k_2}^{m}\epsilon_{k_1}(x_j)\epsilon_{k_2}(x_j) \\
=& \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}\sum_{i=1}^{m}\epsilon_i(x_j)^2 + \frac{1}{m^2}\sum_{k_1=1}^{m}\sum_{k_2=1,k_1\neq k_2}^{m}[\frac{1}{n}\sum_{j=1}^{n}\epsilon_{k_1}(x_j)\epsilon_{k_2}(x_j)]
\end{aligned}$$

由于所有分类器误差均值为零，且不相关

$$\frac{1}{n}\sum_{j=1}^{n}\epsilon_{k_1}(x_j)\epsilon_{k_2}(x_j) = 0(k_1, k_2 \in \{1, 2, \ldots, m\})$$

代入后有:

$$E_{h_B} = \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}\sum_{i=1}^{m}\epsilon_i(x_j)^2 = \frac{1}{m}E_h$$

2.实际情况中，它们的误差往往高度相关，请在（1）条件不满足的情况下证明：$E_{h_B} \leq E_h$

当（1）中条件不成立时

$$E_h = \frac{1}{m}\frac{1}{n}\sum_{i=1}^{m}\sum_{j=1}^{n}\epsilon_i(x_j)^2 = \frac{1}{m}\frac{1}{n}\sum_{j=1}^{n}[\sum_{i=1}^{m}\epsilon_i(x_j)^2]$$

$$E_{h_B} = \frac{1}{n}\frac{1}{m^2}\sum_{j=1}^{n}[\sum_{i=1}^{m}\epsilon_i(x_j)]^2 = \frac{1}{n}\frac{1}{m}\sum_{j=1}^{n}[\frac{1}{m}(\sum_{i=1}^{m}\epsilon_i(x_j))^2]$$

若证$E_{h_B} \leq E_h$，即证：

$$\frac{1}{m}(\sum_{i=1}^{m}\epsilon_i(x_j))^2 \leq \sum_{i=1}^{m}\epsilon_i(x_j)^2$$

由柯西不等式知：

$$\sum_{i=1}^{m}\epsilon_i(x_j)^2 \cdot \sum_{i=1}^{m}1^2 \geq [\sum_{i=1}^{m}\epsilon_i(x_j)\cdot 1]^2$$

$$\Rightarrow m\cdot\sum_{i=1}^{m}\epsilon_i(x_j)^2 \geq (\sum_{i=1}^{m}\epsilon_i(x_j))^2$$

$$\Rightarrow \frac{1}{m}\cdot(\sum_{i=1}^{m}\epsilon_i(x_j))^2 \leq \sum_{i=1}^{m}\epsilon_i(x_j)^2$$

原结论得证，$E_{h_B} \leq E_h$成立。

2. 决策树

0.数据集划分

```
import numpy as np
from sklearn.model_selection import train_test_split
#选取20%的数据作为测试集
x_train, x_test, y_train, y_test = train_test_split(feature, label, test_size=0.2,
random_state=3)
#将特征与标签合并为较大的矩阵，便于处理
data_train=np.concatenate((x_train,y_train),axis=1)
data_test=np.concatenate((x_test,y_test),axis=1)
#选取80%的数据进行四折交叉验证
from sklearn.model_selection import KFold
kf = KFold(n_splits=4)
for train_index, test_index in kf.split(data_train):
            train_cv=data_train[train_index]
            test_cv=data_train[test_index]
            x_test_cv=test_cv[:,:-1]
            y_test_cv=test_cv[:,-1]
```

1.决策树算法编写

```
# 计算数据的熵(entropy)
def Impurtity(samples):
    numEntries=len(samples)
    labelCounts={}
    for featVec in samples:
        if len(featVec)==0:
            return 0
        currentLabel=featVec[-1] # 当前数据的类别
        if currentLabel not in labelCounts.keys():
            labelCounts[currentLabel]=0
        labelCounts[currentLabel]+=1 # 统计有多少个类以及每类的数量
    shannonEnt=0
    for key in labelCounts:
        prob=float(labelCounts[key])/numEntries # 计算单类的熵值
        shannonEnt-=prob*math.log(prob,2) # 累加每个类的熵值
    return shannonEnt
```

构建决策树时选用信息熵作为不纯度度量，公式为$I=-\sum_{i=1}^{k}P_ilog_2Pi$

```
# 对当前节点下样本选择待分特征
def SelectFeature(SamplesUnderThisNode):
     numFeatures = len(SamplesUnderThisNode[0]) - 1#该节点下的特征数
     baseEntropy = Impurtity(SamplesUnderThisNode)
     bestInfoGain = 0.0
     bestFeature = -1
     for i in range(numFeatures):
         featList = [example[i] for example in SamplesUnderThisNode]
         uniqueVals = set(featList)#统计该特征有多少种取值
         newEntropy = 0.0
         for value in uniqueVals:
             subSamplesUnderThisNode = SplitNode(SamplesUnderThisNode, i, value)
             prob = len(subSamplesUnderThisNode) / float(len(SamplesUnderThisNode))
             newEntropy += prob * Impurtity(subSamplesUnderThisNode)
         infoGain = baseEntropy -newEntropy#未计算信息增益
         if infoGain > bestInfoGain:
             bestInfoGain = infoGain
             bestFeature = i
     return bestFeature
#对当前节点进行分支
def SplitNode(SamplesUnderThisNode, axis, value):
     retDataSet = []
     reducedFeatVec=[]
     for featVec in SamplesUnderThisNode:
         if type(featVec)!=type([]):
             featVec=featVec.tolist()
         if featVec[axis] == value:
             reducedFeatVec = featVec[:axis]
             reducedFeatVec.extend(featVec[axis+1:])#把当前选中的特征从其中删掉
             retDataSet.append(reducedFeatVec)
     return retDataSet
```

为了程序编写方便，实验中选择将停止分支的判断放在构建决策树的GenerateTree函数中进行实现。

本实验构建决策树时停止分支条件如下：

- 节点不纯度已小于阈值
- 所有特征已经用完
- 已经达到决策树最大深度

当分支条件至少有一个满足时，分支即停止。需要注意的是，在构建决策树时，若某节点的不纯度已经小于阈值，则该节点不再进行划分，即该节点成为叶子节点 。

```
#递归方法构建树
def GenerateTree(dataSet, labels,thre,max_depth):
     classList = [example[-1] for example in dataSet]
     #类别相同则停止划分
     if classList.count(classList[0]) ==len(classList):
         return classList[0]
     #不纯度已小于阈值
     if Impurtity(dataSet)<thre:
         return majorityCnt(classList)
     #所有特征已经用完
     if len(dataSet[0]) == 1:
         return majorityCnt(classList)
     #已经达到树的深度
     if max_depth <= 0:
         return majorityCnt(classList)
     bestFeat = SelectFeature(dataSet)
     bestFeatLabel = labels[bestFeat]
     print(bestFeatLabel)
     myTree = {bestFeatLabel:{}}
     del(labels[bestFeat])
     featValues = [example[bestFeat] for example in dataSet]
     uniqueVals = set(featValues)
     for value in uniqueVals:
         subLabels = labels[:]
         myTree[bestFeatLabel][value] = GenerateTree(SplitNode(dataSet,
                                         bestFeat, value),subLabels,thre,max_depth-1)
     return myTree
```

递归构建决策树是选取每次选取最优特征进行划分，由于存在所有特征已经用完、节点不纯度已小于阈值、 或已经达到决策树最大深度等情况，而此时节点不纯度不为0，因而需要采用多数表决的方式计算节点分类。

```
#多数表决的方式计算节点分类
def majorityCnt(classList):
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        classCount[vote] += 1
    return max(classCount)
```

影响决策树的超参数主要有：

- max_depth（树的最大深度）

- max_leaf_nodes（叶子结点的最大数目）
- max_features（最大特征数）
- min_samples_leaf（叶子结点的最小样本数）
- min_samples_split（中间结点的最小样本数）
- min_weight_fraction_leaf（叶子节点的样本权重占总权重的最小比例）
- min_impurity_split（节点最小不纯度）

由于实验中特征数很多，样本数也很多， 每次计算耗时很长，因而实验中仅选取决策树最大深度与节点最小不纯度进行探究，超参数的具体选择将在交叉验证部分详述。

```
#使用生成的树GeneratedTree，对单个样本进行预测。
def Decision(GeneratedTree,SampleToBePredicted,featLabels):
    currentFeat = list(GeneratedTree.keys())[0]
    secondTree = GeneratedTree[currentFeat]
    featureIndex = featLabels.index(currentFeat)
    classLabel=1
    for value in secondTree.keys():
        if value == SampleToBePredicted[featureIndex]:
            if type(secondTree[value]).__name__ == 'dict':
                classLabel = Decision(secondTree[value],SampleToBePredicted,featLabels)
            else:
                classLabel = secondTree[value]
    return classLabel
```

对测试集进行预测时，需要利用已生成的决策树对测试样本进行逐一预测，通过统计正确预测的样本数得到测试集正确率。

2.交叉验证与超参数选择

选取80%的数据进行四折交叉验证，由于单次构建决策树并在验证集上预测耗时近40min，每完成一次四折交叉验证耗时即在2小时左右，因而实验中仅选取节点不纯度最小值为0，0.05，0.1和决策树最大深度为30，50，80时进行探究。将四折交叉验证所得的平均正确率作为该超参数组合的正确率,可得到最佳超参数组合

```
#超参数取值
threshold=[0,0.05,0.1]
max_depth=[30,80,50]
kf = KFold(n_splits=4)
  for thre in threshold:
    for depth in max_depth:
        accuracy_list=[]
        #四折交叉验证
        for train_index, test_index in kf.split(data_train):
            train_cv=data_train[train_index]
            test_cv=data_train[test_index]
            x_test_cv=test_cv[:,:-1]
            y_test_cv=test_cv[:,-1]
            #利用训练集构建决策树
            t1 = time.clock()
            myTree = GenerateTree(train_cv,label,thre,depth)
            t2 = time.clock()
            print ('Complete establishing! Execute for ',t2-t1)
            #预测验证集准确率
```

```
            y_pred=[]
            for test in x_test_cv:
                temp=Decision(myTree,test,labels)
                y_pred.append(temp)
            right=0
            for i in range(len(y_test_cv)):
                 if y_pred[i]==y_test_cv[i]:
                     right = right+1
            accuracy=right/ float(len(y_test_cv))
            print("accuracy:",accuracy)
            accuracy_list.append(accuracy)
        accuracy_list=np.array(accuracy_list)
        #输出超参数组合与平均正确率
        print("Threshold:",thre,"  Max_deoth:",depth)
        print("Average accuracy:",np.mean(accuracy_list))
```

以超参数组合为节点最小不纯度为0，决策树最大深度为80为例，构建决策树时由输出可以发现，依次选取的最优特征维为368，263, 473，384, 768，13，1114，1177，222，304，966，557，1000...

此时四折交叉验证得到的正确率分别为69.94%，72.50%，70.89%，69.44%，即平均正确率为70.69%

利用类似方法可得各超参数组合下的训练集、交叉验证集的分类正确率如下：

最小不纯度	**0.1**	**0.05**	**0**	**0.1**	**0.05**	**0**	**0.1**	**0.05**	**0**
最大深度	30	30	30	50	50	50	80	80	80
平均正确率	48.61%	52.97%	64.93%	48.92%	54.44%	66.38%	48.75%	55.65%	70.69%

可以看出当节点最小不纯度不为0时，决策树构建会明显过早截止，特征利用不充分，决策树性能受到很大影响；决策树深度过浅时，决策树也会出现特征利用不够充分的情况，但总体而言模型性能受决策树深度影响更小。

可以从上表看出，节点最小不纯度min_impurity_split为0，决策树最大深度max_depth为80时，是最佳的超参数设置。在测试集进行进一步预测，得到测试集正确率为**72.53%**

根据课程所学知识可知，采用训练集进行交叉验证时，由于数据划分后训练样本数量变少、模型训练没有用到所有样本、训练不充分等原因，验证集可能高估测试错误率。实际使用全部80%的训练数据构建决策树时，测试集准确率比交叉验证的平均准确率有所提高。

进一步选取超参数为节点不纯度为0，最大树深为100时进行探究，此时交叉验证的平均准确率为66.39%。说明决策树并非越深越好。查找资料发现决策树深度在10~100范围内时较为合适，决策树过深时除导致计算量过大外，还会使模型出现一定的过拟合现象，导致模型性能下降，预测结果变差。

3.DecisionTreeClassifier

使用sklearn中的DecisionTreeClassifier函数构建决策树，可利用网格搜索GridSearchCV进行超参数选择

```
from sklearn.cross_validation import train_test_split
x_train, x_test, y_train, y_test = train_test_split(feature, label, test_size=0.2,
random_state=3)
parameters={
            'criterion':['gini','entropy'],
            'max_depth':[50,80,100,120],
            'min_impurity_decrease':[0,0.05,0.1]
```

```
            }
dtree=tree.DecisionTreeClassifier()
grid_search=GridSearchCV(dtree,parameters,scoring='accuracy',cv=4)
grid_search.fit(x_train,y_train)
grid_search.best_estimator_    #查看grid_search方法
grid_search.best_score_        #正确率
grid_search.best_params_       #最佳超参数组合

print("Best: %f using %s" % (grid_search.best_score_, grid_search.best_params_))
```

由于sklearn中以节点分裂导致的不纯度最小减小量min_impurity_decrease代替节点最小不纯度min_impurity_split，可以得到最佳超参数组合为选用 gini不纯度, 最大树深为80, 不纯度最小减小量为0，交叉验证得到的最高正确率为74.07%。

选用最佳超参数组合进行预测：

```
dtree=tree.DecisionTreeClassifier(criterion='gini',max_depth=80,
                                  min_impurity_decrease=0)
dtree.fit(x_train,y_train)
pred=dtree.predict(x_test)
print(classification_report(y_test,pred))
```

得到测试集准确率为75%，具体分类情况如下：

```
             precision    recall  f1-score   support

          1       0.73      0.78      0.75       318
          2       0.63      0.66      0.64       321
          3       0.66      0.64      0.65       317
          4       0.83      0.91      0.86       335
          5       0.75      0.71      0.73       328
          6       0.79      0.70      0.74       340
          7       0.89      0.90      0.90       309
          8       0.62      0.61      0.61       305
          9       0.81      0.80      0.81       307

avg / total       0.75      0.75      0.75      2880
```

和自己编写的决策树进行对比，可以发现，除不纯度度量选择不同外，DecisionTreeClassifier得到的最佳超参数组合和手工编写的决策树模型相一致；测试集上正确率也十分接近，sklearn自身的函数得到的测试集预测准确率比手工编写的决策树模型略高。

其中最明显的差别在于，sklearn自带函数的运行速度远快于手工，从构建决策树至完成预测仅耗时3min左右。考虑到sklearn本身可能较多地利用向量化编程，gini不纯度计算时不用求取对数、计算复杂度本身低于信息熵等因素，时间差异能得到较好的解释。总而言之，通过对比可以验证自己编写的决策树模型有着较良好的性能。

4.RandomForestClassifier

使用sklearn中的DecisionTreeClassifier函数构建决策树，可利用网格搜索GridSearchCV进行超参数选择

```
from sklearn.cross_validation import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(feature, label, test_size=0.2,
random_state=3)
parameters={
            'criterion':['gini','entropy'],
            'max_depth':[50,80,100,120],
            'min_impurity_decrease':[0,0.05,0.1]
            }
dtree=RandomForestClassifier()
grid_search=GridSearchCV(dtree,parameters,scoring='accuracy',cv=4)
grid_search.fit(x_train,y_train)
grid_search.best_estimator_    #查看grid_search方法
grid_search.best_score_        #正确率
grid_search.best_params_       #最佳超参数组合

print("Best: %f using %s" % (grid_search.best_score_, grid_search.best_params_))
```

sklearn中森林里决策树的数目的缺省值为10。通过可以GridSearchCV得到最佳超参数组合为：选用 gini不纯度， 最大树深为120，不纯度最小减小量为0。四折交叉验证得到的最佳超参数下的正确率为80.22%。

选用最佳超参数组合进行预测：

```
dtree=tree.DecisionTreeClassifier(criterion='gini',max_depth=120,
                                  min_impurity_decrease=0)
dtree.fit(x_train,y_train)
pred=dtree.predict(x_test)
print(classification_report(y_test,pred))
```

得到测试集准确率为81%，具体分类情况如下：

```
                precision    recall  f1-score   support

          1       0.76      0.81      0.79       318
          2       0.70      0.71      0.70       321
          3       0.74      0.79      0.76       317
          4       0.89      0.95      0.92       335
          5       0.82      0.84      0.83       328
          6       0.88      0.75      0.81       340
          7       0.90      0.92      0.91       309
          8       0.73      0.65      0.69       305
          9       0.91      0.88      0.89       307

avg / total       0.81      0.81      0.81      2880
```

和自己编写的决策树与sklearn的DecisionTreeClassifier函数相比，RandomForestClassifier函数得到的模型预测准确率明显有所提高，速度更是远快于自己编写的决策树。

随机森林算法属于集成学习算法，其包含多个决策树的分类器。通过对各个树的预测结果进行汇总投票以作为最终预测，可以有效提升模型准确性；此外随机森林利用bootstrapping的方法，训练过程并行，因而训练速度可以很快。可以看出，在解决大规模数据与高维度数据时，随机森林可以取得更好的效果。