第一部分 分类

第2章 K近邻算法

k-近邻算法采用测量不同特征值之间的距离的方法进行分类。

* 2.1 k-近邻算法概述

* 算法伪代码

对未知类别的属性的数据集中的每个点以此执行以下操作：

计算已知类别数据集中的每个点与当前点之间的距离；

按照距离递增的次序排序；

选取与当前点距离最小的k个点；

确定前k个点所在类别的出现频率；

返回前k个点出现频率最高的类别作为当前点的预测分类；

* 欧式距离公式

$$d = \sqrt{(xA_0 - xB_0)^2 + (xA_1 - xB_1)^2}$$

* 数据归一化

$$newValue = (oldValue - min)/(max - min)$$

可以将所有数字转化到0到1的区间。

* 2.2 小结

优点：精度高、对异常值不敏感、无数据输入假定。

缺点：计算复杂度高、空间复杂度高。

第3章 决策树

* 3.1 决策树的构造

首先根据信息论划分数据集，将理论应用到具体的数据集上，最后编写代码构造树。必须判断当前数据中的那个特征在划分数据时起决定性作用，为此必须去评估每个特征，创建分支的伪代码函数createBranch()如下所示：

检查数据集中的每个子项是否属于同一分类：

If so return 类标签；

else:

寻找划分数据集的最好特征；

划分数据集

创建分支节点

for 每个划分的子集:

调用函数createBranch并增加返回结果到分支节点中

return 分支节点

* 3.1.1 信息增益

划分数据集的原则就是将无序的数据变得更加有序。集合信息的度量方式称为香农熵，熵的定义为信息的期望值，如果待分类的事务可能划分在多个分类中，则符号x_i的信息定义为：

$$l(x_i) = -log_2 p(x_i)$$

其中$p(x_i)$是选择该分类的概率。为了计算所有类别中包含信息值的期望，可以通过下面的公式得到：

$$H = -\sum_{i=1}^{n} p(x_i) log_2 p(x_i)$$

下面给出计算香浓熵的代码：

```
def calcShannonEnt(dataSet):
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet:
        currentLabel = featVec[-1]
        if currentLabel not in labelCount.keys():
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1
    shannonEnt = 0.0
    for key in labelCounts:
        prob = float(labelCounts[key]) / numEntries
        shannonEnt -= prob * log(prob,2)
    return shannonEnt
```

* 3.1.2 划分数据集

按照给定的特征划分数据集：

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

选择最好的数据集划分：

```
def chooseBestFeatureToSplit(dataSet):
    numFeatures = len(dataSet[0]) - 1
    baseEntropy = calcShannonEnt(dataSet)
    bestInfoGain = 0.0; bestFeature = -1
    for i in range(numFeatures):
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)
        newEntropy = 0.0
        for value in uniqueVals:
            subDataSet = splitDataSet(dataSet,i,value)
            prob = len(subDataSet) / float(len(dataSet))
            newEntropy += prob * calcShannonEnt(subDataSet)
        infoGain = baseEntropy - newEntropy
        if (infoGain > bestInfoGrain):
            bestInfoGain = infoGain
            bestFeature = i
    return beatFeature
```

* 3.1.3 递归构建决策树

创建树的函数代码

```
def createTree(dataSet,labels):
    classList = [example[-1] for example in dataSet]
    # 如果类别完全相同，就停止划分
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)
    bestFeatLabel = labels[bestFeat]
    myTree = {bestFeatLable:{}}
    def(labels[bestFeat])
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet,bestFeat,value),subLabels
    return myTree
```

* 3.2 补充

定义5.2(信息增益) 特征A对训练数据集D的信息增益$g(D,A)$,定义为集合D的经验熵$H(D)$与特征A在给定条件下D的经验条件熵$H(D|A)$之差，即:

$$g(D,A)=H(D)-H(D|A)$$

信息增益算法:
输入：训练数据集D和特征A;
输出：特征A对训练数据集D的信息增益$g(D,A)$.
(1)计算数据集D和经验熵$H(D)$

$$H(D)=-\sum_{k=1}^{K}\frac{C_k}{D}log_2\frac{C_k}{D}$$

(2)计算特征A对数据集D的经验条件熵$H(D|A)$

$$H(D|A)=\sum_{i=1}^{n}\frac{D_i}{D}H(D_i)=-\sum_{i=1}^{n}\frac{D_i}{D}\sum_{k=1}^{K}\frac{D_{ik}}{D_i}log_2\frac{D_{ik}}{D_i}$$

(3)计算信息增益:

$$g(D,A)=H(D)-H(D|A)$$

缺点：算法容易过拟合。

第4章 朴素贝叶斯

本章给出了基于概率论知识的分类方法，主要介绍朴素贝叶斯，之所以称为“朴素”，是因为在形式化的过程中，只考虑最简单的假设。算法在数据较少的时候仍然有效，可以处理多分类问题，但是对于数据的准备方式较为敏感。

* 4.1 基于贝叶斯决策理论的分类方法

给一个新的数据点(x,y)，可以通过以下的规则判断它的类别：
* 如果$p1(x,y) > p2(x,y)$，那么类别为1.
* 如果$p1(x,y) < p2(x,y)$，那么类别为2.
也就是说，我们会选择高概率对应的类别，这也是贝叶斯决策理论的核心思想。

* 4.2 条件概率

下面给出一种计算条件概率的方法：贝叶斯准则。即如果已知$p(x|c)$，要求$p(c|x)$，那么可以使用下面的计算方法：

$$p(c|x) = \frac{p(x|c)p(c)}{p(x)}$$

* 4.3 使用条件概率分类

贝叶斯准则：

$$p(c_i|x,y) = \frac{p(x,y|c_i)p(c_i)}{p(x,y)}$$

使用这些定义，可以定义贝叶斯分类准则为：
如果$p(c_1|x,y) > p(c_2|x,y)$，那么属于类别c_1。
如果$p(c_1|x,y) < p(c_2|x,y)$，那么属于类别c_2。

* 4.4 使用朴素贝叶斯和python进行文档分类

我们假设一句话中的单词在统计意义上是独立的，也就是说某一个单词的出现和其他相邻的单词没有关系。

* 4.5.1 准备数据

首先需要构建一个可以将词语转换为向量的函数，本例中采用的是$one-hot$编码。

* 4.5.2 从词向量计算概率

这里将之前的x,y替换为w，这里的w是一个向量，由多个数值组成。

$$p(c_i|w) = \frac{p(w|c_i)p(c_i)}{p(w)}$$

首先通过类别i(侮辱性留言或者非侮辱性留言)中文档数除以总的文档数来计算概率$p(c_i)$.然后计算$p(w|c_i)$,如果将$p(w|c_i)$展开，可以写成$p(w_0, w_1, w_2, \ldots, w_N|c_i)$.这里假设所有词都是相互独立的，那么上述概率可以写成$p(x_0|c_i)p(x_1|c_i)p(x_2|c_i)\ldots p(x_N|c_i)$.

- 算法的伪代码：
  计算每个类别中的文档数目。
  对每篇训练文档：
    对每个类别：
      如果词条出现在文档中→增加该词条的计数值。
      增加所有词条的计数值。
  对每个类别：
    对每个词条：
      将该词条的数目除以总词条数目得到条件概率。
  返回每个类别的条件概率。

补充：在算法实现时，可以不用去考虑计算$p(w)$，因为算法只关心比较$p(c_i|w)$之间的大小。

第5章 Logistic回归

第6章 支持向量机

第7章 AdaBoost

2.回归

第8章 回归

第9章 树回归

第9章介绍的是一种分类回归树CART(Classification And Regression Trees),该算法可以用于回归和分类。

* 9.1 复杂数据的局部建模

和第3章的决策树不同，决策树是一种贪心算法，需要在规定的时间内给出合适的决策，并不能考虑到全局最优。并且决策树使用的算法是ID3，每次选取当前的最佳特征去切割，特征使用后，在之后的划分过程中将不会再起作用，并且ID3不可以处理连续性的数据。CART可以通过二元切割来划分数据，并且稍作修改就可以处理回归问题。

* 9.2 连续和离散型特征数的构建

首先构建数据类型，新建一个字典，里面包含如下的数据：待切分的特征；待切分的特征值；右子树(不需要切分时是单值)；左子树。函数createTree()的伪代码如下：

```
找到最佳的待切分特征:
    如果该节点不能划分，将该节点存为叶节点
    执行二元划分
    在右子树调用createTree()
    在左子树调用createTree()
```

首先给出了三个函数$loadDataSet()$，$binSplitDataSet()$，$createTree()$,我们主要介绍函数$createTree()$.

```
def createTree(dataSet,leafType=regLeaf,errType=regErr,ops=(1,4)):
    feat,val = chooseBestSplit(dataSet,leafType,errType,ops)
    if feat == None: return val
    # 创建一个空字典，'spInd'记录的当前的特征，'spVal'记录的是特征值val
    retTree()
    retTree['spInd'] = feat
    retTree['spVal'] = val
    #通过二分切割，得到两棵不同的子树
    lSet,rSet = binSplitDataSet(dataSet,feat,val)
    retTree['left'] = create(lSet,leafType,errType,ops)
    retTree['right'] = create(rSet,leafType,errType,ops)
    return retTree
```

* 9.3 将CART用于算法

* 9.3.1 构建数

为了让$createTree()$函数运行，需要实现$chooseBestSplit()$函数，给定某个误差的计算方法，该函数会找到数据集的最佳二元切分方式。$chooseBestSplit()$只需要做两件事：切分数据集、生成相应的叶节点。其中，$leafType$是对创建叶节点的函数的引用，errType是对计算总方差函数的引用，ops是用户自定义的参数。

```
def chooseBestSplit(dataSet,leafType=regLeaf,errType=regErr,ops=(1,4)):
    # tolS,tolN这两个参数用来控制函数的停止时机。tolS是容许误差的下降值，tolN是切分的最少样本数
    tolS = ops[0]; tolN = ops[1]
    # 如果数据集中所有的值相等，就推出
    if len(set(dataSet[:,-1].T.tolist()[0])) == 1:
        return None,leafType=(dataSet)
    m,n = shape(dataSet)
    # S是当前计算得到的总方差
    S = errType(dataSet)
    bestS = inf; bestIndex = 0; bestValue = 0
    for featIndex in range(n-1):
        for splitVal in set(dataSet[:,featIndex]):
            mat0,mat1 = binSplitDataSet(dataSet,featIndex,splitVal)
            # 如果当前的划分不满足条件，就跳过当前步，直接进行下一步
            if(shape(mat0)[0] < tolN) or (shape(mat1)[0] < tolN):continue
            if newS < bestS:
                bestIndex = featIndex
                beatValue = splitVal
                bestS = newS
    # 如果误差的减少的不够大，就直接退出
    if (S - bestS) < tolS:
        return None,leafType(dataSet)
    mat0,mat1 = binSplitDataSet(dataSet,bestIndex,bestValue)
    # 如果切分的数据集很小，就直接推出
    if (shape(mat0)[0] < tolN) or (shape(mat1)[0] < tolN):
        return None,leafType(dataSet)
    return bestIndex,bestValue
```

* 9.4 树剪枝

如果一棵树的节点过多，那么该模型可能会出现‘过拟合’现象，为了应对这种情况，需要对模型进行剪枝(pruning)，有两种剪枝方式，预剪枝(prepruning)和后剪枝(postpruning)，后剪枝需要使用测试集和训练集。

* 9.4.1 预剪枝

预剪枝就是在函数中提前结束对模型的划分，但是对参数ops较为敏感。

* 9.4.2 后剪枝

算法伪代码

基于已有的树切分测试数据：

若存在任意一个子集是一棵树，则在该子集递归剪枝过程；

计算将两个叶子节点合并后的误差；

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；

```
def prune(tree,testData):
    # 首先判断当前节点是否是叶节点，是则直接返回数值类型数据
    if shape(testData)[0] == 0: return getMean(tree)

    # 判断左右两个数是否为子树
    if(isTree(tree['right'])) or (isTree(tree['left'])):
        lSet,rSet = binSplitDataSet(testData,tree['spInd'],tree['spVal'])

    if isTree(tree['left']): tree['left'] = prune(tree['left'],lSet)
    if isTree(tree['right']):tree['right'] = prune(tree['right'],rSet)

    # 最后判断是否将两个子节点合并，根据误差来判断
    if not isTree(tree['left']) and not isTree(tree['right']):
        lSet,rSet = binSplitDataSet(testDatad,tree['spInd'],tree['spVal'])
        errorNoMerge = sum(power(lSet[:,-1] - tree['left'],2) + power(rSet[:,-1] - tree['right']
        treeMean = (tree['left'] + tree['right']) / 2.0
        errorMerge = sum(power(testData[:,-1] - treeMean,2))
        if errorMerge < errorNoMerge:
            print "merging"
            return treeMean
        else: return tree
    else: return tree
```

* 9.5 模型树

用树来对模型进行建模，除了将叶节点设置为常数值之外，还可以将叶节点设置为分段函数，也就是所为的分段线性函数。

```
# 用于执行简单的线性回归
def linearSolve(dataSet):
    m,n = shape(dataSet)
    X = mat(ones(m,n)); Y = mat(ones((m,1)))
    X[:,1:n] = dataSet[:,0:n-1]; Y = mat(ones((:,-1)))
    xTx = X.T * X
    # 如果矩阵可逆
    if linalg.det(xTx) == 0.0:
        raise NameError('This matrix is singular,cannot do inverse,try increasing the second val
    ws = xTx.I * (X.T * Y)
    return ws,X,Y

# 当节点不需要切分时，负责生成叶节点
def modelLeaf(dataSet):
    ws,X,Y = linearSolve(dataSet)
    return ws


# 计算误差
def modelErr(dataSet):
    ws,X,Y = linearSolve(dataSet)
    yHat = X * ws
    return sum(power(Y-yHat),2)
```

* 9.6 示例

下面给出用树回归进行预测的代码。

```
# 要对回归树叶节点进行预测，就要调用regTreeVal()
def regTreeEval(model,inDat):
    return float(model)

# 要对模型树节点进行预测，就要调用modelTreeEval()
def modelTreeEval(model,inDat):
    n = shape(inDat)[1] #返回输入数据的列数
    X = mat(ones((1,n+1)))
    X[:,1:n+1] = inDat
    return float(X*model)

#对于输入的单个数据点或者行向量，函数createForeCast()会返回一个浮点值，对于单个数据点，函数会返回一个预
def treeForeCast(tree,inData,model=regTreeEval):
    # 如果输入的是一个数据，不是一棵树，就返回值
    if not isTree(tree): return modelEval(tree,inData)
    if inData[tree['spInd']] > tree['inData']:
        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)

def createForeCast(tree,testData,modelEval=regTreeEval):
    m = len(testData)
    yHat = mat(zeros((m,1)))
    for i in range(m):
        yHat[i,0] = treeForeCast(tree,mat(testData[i]),modelEval)
    return yHat
```

* 9.7 小结

优点：可以对复杂和非线性的数据建模
缺点：结果不易理解

3.无监督学习

第10章 kMeans

聚类是一种无监督学习，将相似的对象归为一类.kMeans是一种聚类算法，它可以发现K个聚类中心，每个聚类中心就是簇中所有值的均值。聚类和分类最大的不同在于，分类的目标实先已知，但是聚

类的类别没有预先定义，所有聚类有时候称为无监督学习。

* 10.1 kMeans聚类算法

kMeans是发现数据中k个簇的算法，簇的个数是用户一开始给定的，算法的伪代码如下：

创建k个初始聚类中心(一般是随机选择);

当任意一个点的簇分配结果发生改变时：

对数据中的每一个数据点：

对每一个聚类中心：

计算聚类中心和数据点之间的距离

将数据点分配到距离其最近的簇

对每一个簇，计算簇中所有点的均值，并将其作为聚类中心。

* 算法优点：容易实现；缺点：可能会收敛到局部最小值，在大规模数据上收敛较慢。

* 10.2 使用后处理提高聚类性能

有一种用于度量聚类效果的指标是SSE(Sum of Squared Error,误差平方和)。SSE值越小表示数据点越接近他们的质心，聚类效果也越好。降低SSE值的方法就是增加聚类中心的数量，但是这违背了聚类的目标。聚类的目标就是在保持簇数目不变的条件下提高簇的质量。

有两种可以量化的方法：合并最近的聚类中心，或者合并两个使得SSE增幅最小的聚类中心。第一中思路通过计算所有聚类中心之间的距离，然后合并距离最近的两个点来实现；第二种方法需要合并两个簇然后计算SSE值，必须在所有可能的两个簇上重复上述过程。

* 10.3 二分kMeans均值算法

该算法首先将所有点作为一个簇，然后将该簇一分为二，之后选择其中之一继续划分，选择哪一个簇取决于能否最大程度降低SSE的值。上述基于SSE的划分过程不断重复，直到得到用户指定的数目k为止。

伪代码的形式如下：

将所有点看出一个簇：

当簇的数目小于k时：

对于每一个簇：

计算总误差

在给定的簇上面进行kMeans聚类(k=2)

计算将该簇一分为二后的总误差

选择使得误差最小的那个簇进行划分操纵

4.其他

10.27 begin:,,,