**NLP\_chapter6**

**第六章 学习分类文本**

**6.1 有监督分类**

**分类**是为给定的输入选择正确的**类标签**的任务。在基本的分类任务中，每个输入被认为是与所有其它输入隔离的，并且标签集是预先定义的。

基本的分类任务有许多有趣的变种。例如:在多类分类中，每个实例可以分配多个标签;在开放性分类中，标签集是事先没有定义的;在序列分类中，一个输入链表作为一个整体分类。

如果分类的建立基于包含每个输入的正确标签的训练语料，被称为**有监督分类**。其基本框架：

(a)在训练过程中，特征提取器用来将每一个输入值转换为特征集。 这些特征集捕捉每个输入中应被用于对其分类的基本信息，我们将在下一节中讨论它。特征 集与标签的配对被送入机器学习算法，生成模型。(b)在预测过程中，相同的特征提取器被 用来将未见过的输入转换为特征集。之后，这些特征集被送入模型产生预测标签。

**性别鉴定**

创建一个分类器的第一步是决定输入的什么样的**特征**是相关的，以及如何为那些特征**编码**。在这个例子中，我们一开始只是寻找一个给定的名称的最后一个字母。以下**特征提取器**函数建立一个字典，包含有关给定名称的相关信息:

def gender\_features(word):

return {'last\_letter':word[-1]}

print(gender\_features('Shrek'))

这个函数返回的字典被称为**特征集**，映射特征名称到它们的值。特征名称是区分大小写的字符串，通常提供一个简短的人可读的特征描述。特征值是简单类型的值，如布尔、数字和字符串。

准备一个例子和对应类标签的链表:

from nltk.corpus import names

import random

names = ([(name,'male') for name in names.words('male.txt')]+

[(name,'female') for name in names.words('female.txt')])

random.shuffle(names)

使用特征提取器处理名称数据，并划分特征集的结果链表为一个**训练集**和一个**测试集**。训练集用于训练一个新的“朴素贝叶斯”分类器:

featuresets = [(gender\_features(n),g) for (n,g) in names]

train\_set,test\_set = featuresets[500:],featuresets[:500]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

在上面测试一些没有出现在训练数据中的名字:

print(classifier.classify(gender\_features('Neo')))

print(classifier.classify(gender\_features('Trinity')))

在大数据量的未见过的数据上系统 地评估这个分类器:

print(nltk.classify.accuracy(classifier,test\_set))

检查分类器，确定哪些特征对于区分名字的性别是最有效的:

classifier.show\_most\_informative\_features(5)

对于不同特征的男性女性的这些比率称为**似然比**，可以用于比较不同特征-结果关系。

在处理大型语料库时，构建一个包含每一个实例的特征的单独的链表会使用大量的内存。在这些情况下，使用函数 nltk.classify.apply\_features，返回一个行为像一个链表而不会在内存中存储所有特征集的对象:

from nltk.classify import apply\_features

train\_set = apply\_features(gender\_features,names[500:])

test\_set = apply\_features(gender\_features,names[:500])

**选择正确的特征**

选择相关的特征，并决定如何为一个学习方法编码它们，这对学习方法提取一个好的模型可以产生巨大的影响。虽然使用相当简单而明显的特征集往往可以得到像样的性能，但是使用精心构建的基于对当前任务的透彻理解的特征，通常会显著提高收益。

典型地，特征提取是通过反复试验和错误的过程建立的，由哪些信息是与问题相关的直觉 指引的。它通常以“厨房水槽”的方法开始，包括你能想到的所有特征，然后检查哪些特征 是实际有用的。

一个特征提取器，过拟合性别特征。这个特征提取器返回的特征集包括大量指定的特征，从而导致对于相对较小的名字语料库过拟合:

def gender\_features2(name):

features = {}

features["firstletter"] = name[0].lower()

features["lastleter"] = name[-1].lower()

for letter in 'abcdefghijklmnopqrstuvwxyz':

features["count(%s)"%letter] = name.lower().count(letter)

features["has(%s)"%letter] = (letter in name.lower())

return features

print(list(gender\_features2('John').items())[:10])

然而，用于一个给定的学习算法的特征的数目是有限的——如果提供太多的特征，那么该算法将高度依赖你的训练数据的特性而一般化到新的例子的效果不会很好。这个问题被称为**过拟合**，当运作在小训练集上时尤其会有问题。显然这对分类器的性能是不利的。

一旦初始特征集被选定，完善特征集的一个非常有成效的方法是**错误分析**。首先，选择一个**开发集**，包含用于创建模型的语料数据。然后将这种开发集分为**训练集**和**开发测试集**。

train\_names = names[1500:]

devtest\_names = names[500:1500]

test\_names = names[:500]

训练集用于训练模型，开发测试集用于进行错误分析，测试集用于系统的最终评估。

已经将语料分为适当的数据集，使用训练集训练一个模型，然后在开发测试集上运行:

train\_set = [(gender\_features(n),g) for (n,g) in train\_names]

devtest\_set = [(gender\_features(n),g) for (n,g) in devtest\_names]

test\_set = [(gender\_features(n),g) for (n,g) in test\_names]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,devtest\_set))

使用开发测试集，可以生成一个分类器预测名字性别时的错误列表:

errors = []

for (name,tag) in devtest\_names:

guess = classifier.classify(gender\_features(name))

if guess != tag:

errors.append((tag,guess,name))

然后，可以检查个别错误案例，在那里该模型预测了错误的标签，尝试确定什么额外信息将使其能够作出正确的决定(或者现有的哪部分信息导致其做出错误的决定)。然后可以相应的调整特征集：

for (tag,guess,name) in sorted(errors)[:10]:

print('correct={:<8} guess={:<8} name={:<30}'.format(tag,guess,name))

这个错误列表，它明确指出一些多个字母的后缀可以指示名字性别。调整特征提取器包括两个字母后缀的特征:

def gender\_features(word):

return {'suffix1':word[-1],

'suffix2':word[-2:]}

使用新的特征提取器重建分类器，可以看到测试数据集上的性能明显提高

这个错误分析过程可以不断重复，检查存在于由新改进的分类器产生的错误中的模式，每一次错误分析过程被重复，应该选择一个不同的开发测试/训练分割，以确保该分类器不会开始反映开发测试集的特质。

但是，一旦已经使用了开发测试集帮助我们开发模型，关于这个模型在新数据会表现多好，我们将不能再相信它会给我们一个准确地结果!因此，保持测试集分离、未使用过，直到模型开发完毕是很重要的。在这一点上，我们可以使用测试集评估模型在新的输入值上执行的有多好。

**文档分类**

使用nltk中已按类别标记的语料库，可以建立分类器，自动给新文档添加适当的类别标签。构造一个标记了相应类别的文档清单。对于这个例子，我们选择电影评论语料库，将每个评论归类为正面或负面：

documents = [(list(movie\_reviews.words(fileid)),category)

for category in movie\_reviews.categories()

for fileid in movie\_reviews.fileids(category)]

random.shuffle(documents)

接下来，为文档定义一个特征提取器。对于文档主题识别，我们可以为每个词定义一个特性表示该文档是否包含这个词。为了限制分类器需要处理的特征的数目，一开始构建一个整个语料库中前 2000 个最频繁词的链表。然后，定义一个特征提取器，简单地检查这些词是否在一个给定的文档中：

all\_words = nltk.FreqDist(w.lower() for w in movie\_reviews.words())

word\_features = [w for (w,\_) in all\_words.most\_common(2000)]

def document\_features(document):

document\_words = set(document)

features = {}

for word in word\_features:

features['contains({})'.format(word)] = (word in document\_words)

return features

print(list(document\_features(movie\_reviews.words('pos/cv957\_8737.txt')).items())[:20])

我们计算文档中的所有词的集合，而不仅仅检查if word in document，因为检查一个词是否在一个集合中出现比检查它是否在一个链表中出现要快的多。

用上面的特征提取器来训练一个分类器，为新的电影评论加标签。为了检查产生的分类器可靠性如何，我们在测试集上计算其准确性。再一次的，我们可以使用 show\_most\_informative\_features()来找出哪些特征是分类器发现最有信息量的：

featuresets = [(document\_features(d),c) for (d,c) in documents]

train\_set,test\_set = featuresets[100:],featuresets[:100]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,test\_set))

classifier.show\_most\_informative\_features(5)

show\_most\_informative\_features()同样可以用于多个类标签的分类器

**词性标注**

训练一个分类器来算出哪个后缀最有信息量，进而替代手工制作的正则表达式标注器。首先，找出最常见的后缀:

suffix\_fdist = nltk.FreqDist()

for word in brown.words():

word = word.lower()

suffix\_fdist[word[-1]]+=1

suffix\_fdist[word[-2:]]+=1

suffix\_fdist[word[-3:]]+=1

common\_suffixes = [suffix for (suffix,count) in suffix\_fdist.most\_common(100)]

print(common\_suffixes)

接下来，定义一个特征提取器函数，检查给定的单词的这些后缀:

def pos\_features(word):

features = {}

for suffix in common\_suffixes:

features['endswith({})'.format(suffix)] = word.lower().endswith(suffix)

return features

特征提取函数的行为就像有色眼镜一样，强调我们的数据中的某些属性(颜色)，并使其无法看到其他属性。分类器在决定如何标记输入时，将完全依赖它们强调的属性。在这种情况下，分类器将只基于一个给定的词拥有(如果有)哪个常见后缀的信息来做决定。

用我们定义的特征提取器来训练一个新的“决策树”的分类器(将在 6.4 节讨论)（决策树分类器的训练时间比朴素贝叶斯分类器要长的多）:

tagged\_words = brown.tagged\_words(categories='news')

featuresets = [(pos\_features(n),g) for (n,g) in tagged\_words]

size = int(len(featuresets)\*0.1)

train\_set,test\_set = featuresets[size:],featuresets[:size]

classifier = nltk.DecisionTreeClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,test\_set))

print(classifier.classify(pos\_featuresets('cats')))

决策树模型的一个很好的性质是它们往往很容易解释。我们甚至可以指示 NLTK 将它们以伪代码形式输出:

print(classifier.pseudocode(depth=4))

实际的分类器包含这里显示的 if-then 语句下面进一步的嵌套，参数 depth=4 只显示决定树的顶端部分。

**探索上下文语境**

通过增加特征提取函数，我们可以修改这个词性标注器来利用各种词内部的其他特征。然而，只要特征提取器仅仅看着目标词，就没法添加依赖词出现的上下文语境特征。然而上下文语境特征往往提供关于正确标记的强大线索。

为了采取基于词的上下文的特征，我们必须修改以前为我们的特征提取器定义的模式。不是只传递已标注的词，而是传递整个(未标注的)句子，以及目标词的索引：

def pos\_features(sentence,i):

features = {"suffix(1)":sentence[i][-1:],

"suffix(2)":sentence[i][-2:],

"suffix(3)":sentence[i][-3:]}

if i == 0:

features["prev-word"] = "<START>"

else:

features["prev-word"] = sentence[i-1]

return features

print(pos\_features(brown.sents()[0],8))

tagged\_sents = brown.tagged\_sents(categories='news')

featuresets = []

for tagged\_sent in tagged\_sents:

untagged\_sent = nltk.tag.untag(tagged\_sent)

for i,(word,tag) in enumerate(tagged\_sent):

featuresets.append(

(pos\_features(untagged\_sent,i),tag))

size = int(len(featuresets)\*0.1)

train\_set,test\_set = featuresets[size:],featuresets[:size]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,test\_set))

在一般情况下，简单的分类器总是将每一个输入与所有其他输入独立对待。

**序列分类**

为了捕捉相关的分类任务之间的依赖关系，可以使用**联合分类器**模型，收集有关输入，选择适当的标签。在词性标注的例子中，各种不同的**序列分类器**模型可以被用来为一个给定的句子中的所有的词共同选择词性标签。

一种序列分类器策略，称为**连续分类**或**贪婪序列分类**，是为第一个输入找到最有可能的类标签，然后使用这个问题的答案帮助找到下一个输入的最佳的标签。这个过程可以不断重复直到所有的输入都被贴上标签。

**其他序列分类方法**

上述方法的一个缺点是决定一旦做出无法更改。

这个问题的一个解决方案是采取**转型策略**。转型联合分类的工作原理是为输入的标签创建一个初始值，然后反复提炼那个值，尝试修复相关输入之间的不一致。Brill 标注器，5.6 节描述的，是这种策略的一个很好的例子。

另一种方案是为词性标记所有可能的序列打分，选择总得分最高的序列。隐马尔可夫模型就采取这种方法。**隐马尔可夫模型**类似于连续分类器，它不光看输入也看已预测标记的历史。然而，不是简单地找出一个给定的词的单个最好的标签，而是为标记产生一个概率分布。 然后将这些概率结合起来计算标记序列的概率得分，最高概率的标记序列会被选中。采用对每个连续的词索引 i，每个可能的当前及以前的标记都被计算得分的方法的两种模型，被称为**最大熵马尔可夫模型**和**线性链条件随机场模型** ;但为标记序列打分用的是不同的算法。

**6.2 有监督分类的更多例子**

**句子分割**

句子分割可以看作是一个标点符号的分类任务:每当我们遇到一个可能会结束一个句子的符号，如句号或问号，我们必须决定它是否终止了当前句子。

首先，获得一些已被分割成句子的数据，将它转换成一种适合提取特征的形式:

sents = nltk.corpus.treebank\_raw.sents()

tokens = []

boundaries = set()

offset = 0

for sent in sents:

tokens.extend(sent)

offset += len(sent)

boundaries.add(offset-1)

在这里，tokens 是单独句子标识符的合并链表，boundaries 是一个包含所有句子边界标识符索引的集合。下一步，需要指定用于决定标点是否表示句子边界的数据特征:

def punct\_features(tokens,i):

return {'next-word-capitalized':tokens[i+1][0].isupper(),

'prevword':tokens[i-1].lower(),

'punct':tokens[i],

'prec-word-is-one-char':len(tokens[i-1]) == 1}

基于这一特征提取器，可以通过选择所有的标点符号创建一个加标签的特征集的链表，然后标注它们是否是边界标识符:

featuresets = [(punct\_features(tokens,i),(i in boundaries))

for i in range(1,len(tokens)-1)

if tokens[i] in '.?!']

使用这些特征集，可以训练和评估一个标点符号分类器:

size = int(len(featuresets)\*0.1)

train\_set,test\_set = featuresets[size:],featuresets[:size]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,test\_set))

使用这种分类器进行断句，我们只需检查每个标点符号，看它是否是作为一个边界标识符，在边界标识符处分割词链表:

def segments\_sentences(words):

start = 0

sents = []

for i,word in words:

if word in '.?!' and classifier.classify(words,i) == True:

sents.append(words[start:i+1])

start = i+1

if start < len(words):

sents.append(words[start:])

**识别对话行为类型**

处理对话时，将对话看作说话者执行的动作是很有用的。对于表述行为的陈述句这种解释是最简单的。但是问候、问题、回答、断言和说明都可以被认为是基于语言的行动类型。识别对话中言语下的**对话行为**是理解谈话的重要的第一步。

NPS 聊天语料库中的帖子都已经被贴上 15 种对话行为类型中的一种标签，例如:“陈述”，“情感”，“yn 问题”，“Continuer”。因此，我们可以利用这些数据建立一个分类器，识别新的即时消息帖子的对话行为类型。第一步是提取基本的消息数据。调用 xml\_posts()来得到一个数据结构，表示每个帖子的 XML 注释:

posts = nltk.corpus.nps\_chat.xml\_posts()[:10000]

下一步，定义一个简单的特征提取器，检查帖子包含什么词:

def dialogue\_act\_features(post):

features = {}

for word in nltk.word\_tokenize(post):

features['contains({})'.format(word.lower())] = True

return features

通过为每个帖子提取特征(使用 post.get('class') 获得一个帖子的对话行为类型)构造训练和测试数据，并创建一个新的分类器:

featuresets = [(dialogue\_act\_features(post.text),post.get('class'))

for post in posts]

size = int(len(featuresets)\*0.1)

train\_set,test\_set = featuresets[size:],featuresets[:size]

classifier = nltk.NaiveBayesClassifier.train(train\_set)

print(nltk.classify.accuracy(classifier,test\_set))

**识别文字蕴含**

识别文字蕴含(Recognizing textual entailment(RTE))是判断文本 T 的一个给定片段是否蕴含着另一个叫做“假设”的文本。

注意，文字和假设之间的关系并不一定是逻辑蕴涵，而是一个人是否会得出结论:文本提供了合理的证据证明假设是真实的。

我们可以把 RTE 当作一个分类任务，尝试为每一对预测真/假标签。虽然这项任务的成功做法似乎看上去涉及语法分析、语义和现实世界的知识的组合，但是RTE 的许多早期的尝试 使用**基于文字和假设之间的在词级别的相似性的**粗浅的分析取得了相当不错的结果。在理想情况下，我们希望如果有一个蕴涵那么假设所表示的所有信息也应该在文本中表示。相反，如果假设中有的资料文本中没有，那么就没有蕴涵。

让词(即词类型)作为信息的代理，寻找特征计数词重叠的程度和假设中有而文本中没有的词的程度(由 hyp\_extra()方法获取)。不是所有的词都是同样重要的——命名实体，如人、组织和地方的名称，可能会更为重要，这促使我们分别为 words 和 nes(命名实体)提取不同的信息。此外，一些高频虚词作为“停用词”被过滤掉。

RTE特征提取器。RTEFeatureExtractor 类建立了一个除去一些停用词后在文本和假设中都有的词汇包，然后计算重叠和差异：

def rte\_features(rtepair):

extractor = nltk.RTEFeatureExtractor(rtepair)

features = {}

features['word\_overlap'] = len(extractor.overlap('word'))

features['word\_hyp\_extra'] = len(extractor.hyp\_extra('word'))

features['ne\_overlap'] = len(extractor.overlap('ne'))

features['ne\_hyp\_extra'] = len(extractor.hyp\_extra('ne'))

return features

检查文本/假设对 34 的一些属性以便于理解这些特征的内容:

rtepair = nltk.corpus.rte.pairs(['rte3\_dev.xml'])[33]

extractor = nltk.RTEFeatureExtractor(rtepair)

print(extractor.text\_words)

print(extractor.hyp\_words)

print(extractor.overlap('word'))

print(extractor.overlap('ne'))

print(extractor.hyp\_extra('word'))

这些特征表明假设中所有重要的词都包含在文本中，因此有一些证据支持标记这个为 True。

**扩展到大型数据集**

Python 提供了一个良好的环境进行基本的文本处理和特征提取。然而，它处理机器学习方法需要的密集数值计算不能够如 C 语言那样的低级语言那么快。因此，如果在大型数据集使用纯 Python 的机器学习实现(如 nltk.NaiveBayesClassifier)，学习算法会花费大量的时间和内存。

如果打算用大量训练数据或大量特征来训练分类器，建议探索 NLTK 与外部机器学习包的接口。只要这些软件包已安装，NLTK 可以透明地调用它们(通过系统调用来训练分类模型，明显比纯 Python 的分类实现快。

**6.3 评估**

**测试集**

大多数评估技术为模型计算一个得分，通过比较它在**测试集**(或**评估集**)中为输入生成的标签与那些输入的正确标签。该测试集通常与训练集具有相同的格式。但是测试集与训练语料必须不同：只记住了输入而没有学会如何推广到新的例子的模型会得到误导人的高分。

选择测试集时应注意的因素：

权衡数据量，在不同的情况下有不同的标准，一般要保证出现次数最少的标签至少出现 50 次。当有大量已标注数据可用时，只使用整体数据的 10%进行评估常常会在安全方面犯错。

测试集中实例与开发集中的实例的相似程度。越相似，评估结果推广到其他数据集的信心就越小。尽量避免测试集中包含来自训练使用过的相同的文档的句子（因为在特定文档中可能出现一个给定的词与特定词性标记一起出现特别频繁的情况），一个稍好的做法是确保训练集和测试集来自不同的文件：

file\_ids = brown.fileids(categories='news')

size = int(len(file\_ids)\*0.1)

train\_set = brown.tagged\_sents(file\_ids[size:])

test\_set = brown.tagged\_sents(file\_ids[:size])

如果要执行更令人信服的评估，应该从与训练集中文档联系更少的文档中获取测试集：

train\_set = brown.tagged\_sents(categories='news')

test\_set = brown.tagged\_sents(categories='fiction')

**准确度**

用于评估一个分类最简单的度量是**准确度**，测量测试集上分类器正确标注的输入的比例。

nltk.classify.accuracy()函数会在给定的测试集上计算分类器模型的准确度。

解释一个分类器的准确性得分，考虑测试集中单个类标签的频率是很重要的。

**精确度和召回率**

在“搜索”任务中，准确度分数可能会产生误导，由于不相关的文档的数量远远多于相关文档的数量，一个将每一个文档都标记为无关的模型的准确度分数将非常接近 100%。

对搜索任务使用不同的测量集：

**• 真阳性(TP)**是相关项目中我们正确识别为相关的。

**• 真阴性(TN)**是不相关项目中我们正确识别为不相关的。

**• 假阳性(FP)**(或 **I 型错误**)是不相关项目中我们错误识别为相关的。

**• 假阴性(FN)**(或 **II 型错误**)是相关项目中我们错误识别为不相关的。

基于这四个类别的每一个中的项目的数量，定义以下指标:

**• 精确度**(Precision)，表示我们发现的项目中有多少是相关的，TP/(TP+ FP)。

**• 召回率**(Recall)，表示相关的项目中我们发现了多少，TP/(TP+FN)。

**• F-度量值**(F-Measure)(或 F-得分，F-Score)，组合精确度和召回率为一个单独的得分，被定义为精确度和召回率的调和平均数(2 × Precision × Recall)/(Precision + Recall )。

**混淆矩阵**

当处理有 3 个或更多的标签的分类任务时，基于模型错误类型细分模型的错误是有信息量的。一个混淆矩阵是一个表，其中每个 cells[i,j]表示正确的标签 i 被预测为标签 j 的次数。 因此，对角线项目(即 cells[i,i])表示正确预测的标签，非对角线项目表示错误。

为unigram 标注器(注意先用pickle模块输入已保存的t2标注器)生成一个混淆矩阵:

def tag\_list(tagged\_sents):

return [tag for sent in tagged\_sents for (word,tag) in sent]

def apply\_tagger(tagger,corpus):

return [tagger.tag(nltk.tag.untag(sent)) for sent in corpus]

gold = tag\_list(brown.tagged\_sents(categories='editorial'))

test = tag\_list(apply\_tagger(t2,brown.tagged\_sents(categories='editorial')))

cm = nltk.ConfusionMatrix(gold,test)

print(cm.pretty\_format(sort\_by\_count=True,show\_percents=True,truncate=9))

**交叉验证**

在不同的测试集上执行多个评估，然后组合这些评估的得分，这种技术被称为**交叉验证**。将原始语料细分为 N 个子集称为**折叠**(folds)。 对于每一个这样的折叠，使用除这个折叠中的数据外其他所有数据训练模型，然后在这个折叠上测试模型。即使个别的折叠可能是太小了而不能在其上给出准确的评价分数，综合 评估得分是基于大量的数据，因此是相当可靠的。

采用交叉验证的另一个优势是，它可以让我们研究不同的训练集上性能变化有多大。如果从所有 N 个训练集得到非常相似的分数，就可以相当有信心，得分是准确的。另一方面，如果 N 个训练集上分数很大不同，那么，应该对评估得分的准确性持怀疑态度。

**6.4 决策树**

**决策树**是一个简单的为输入值选择标签的流程图。这个流程图由检查特征值的**决策节点**和分配标签的**叶节点**组成。为输入值选择标签，我们以流程图的初始决策节点(称为其**根节点**)开始。此节点包含一个条件，检查输入值的特征之一，基于该特征的值选择一个分支。沿着这个描述我们输入值的分支，我们到达了一个新的决策节点，有一个关于输入值的特征的新的条件。我们继续沿着每个节点的条件选择的分支，直到到达叶节点，它为输入值提供了一个标签。

**建立决策树的学习算法**

思考一个简单的任务:为语料库选择最好的“决策树桩”。**决策树桩**是只有一个节点的决策树，基于一个特征决定如何为输入分类。每个可能的特征值是一个叶子，为特征有那个值的输入指定类标签。要建立决策树桩，首先必须决定哪些特征应该使用。最简单的方法是为每个可能的特征建立一个决策树桩，看哪一个在训练数据上得到最高的准确度，也有其他的替代方案，将在下面讨论。一旦选择了一个特征，就可以通过分配一个标签给每个叶子，基于在训练集中所选的例子的最频繁的标签，建立决策树桩(即选择特征具有那个值的例子)。

给出了选择决策树桩的算法，生长出较大的决策树的算法就很简单了。首先，选择分类任务的整体最佳的决策树桩。然后，在训练集上检查每个叶子的准确度。没有达到足够的准确度的叶片被新的决策树桩替换，新决策树桩是在根据到叶子的路径选择的训练语料的子集上训练的。

**熵和信息增益**

也有其他几种方法来为决策树桩确定最有信息量的特征。一种流行的替代方法，被称为**信息增益**，当使用给定的特征分割输入值时，衡量它们变得更有序的程度。要衡量原始输入值集合如何无序，要计算它们的标签的墒，如果输入值的标签非常不同，墒就高;如果输入值的标签都相同，墒就低。特别是，熵被定义为每个标签的概率乘以那个标签的 log 概率的总和。

(1) H = Σl ∈ labelsP(l) × log2P(l).

低频率的标签不会贡献多少给墒(因为 P(l)很小)，高频率的标签对熵也没有多大帮助(因为 log2P(l)很小)。另一方面，如果输入值的标签变化很多，那么有很多“中等”频率的标签，它们的 P(l)和 log2P(l)都不小，所以墒很高。

计算标签链表的墒：

import math

def entropy(labels):

freqdist = nltk.FreqDist(labels)

probs = [freqdist.freq(l) for l in freqdist]

return -sum(p\*math.log(p,2) for p in probs)

一旦已经计算了原始输入值的标签集的墒，就可以判断应用了决策树桩之后标签会变得多么有序。为了这样做，计算每个决策树桩的叶子的熵，利用这些叶子熵值的平均值(加权每片叶子的样本数量)。信息增益等于原来的熵减去这个新的减少的熵。信息增益越高，将输入值分为相关组的决策树桩就越好，于是可以通过选择具有最高信息增益的决策树桩来建立决策树。

决策树的另一个考虑因素是效率。为了提高效率，开发了一些算法通过存储和重用先前评估的例子的信息减少训练时间。

决策树有一些有用的性质：

首先，它们简单明了，容易理解。决策树顶部附近尤其如此，这通常使学习算法可以找到非常有用的特征。决策树特别适合有很多层次的分类区别的情况。

决策树也有一些缺点：

一个问题是，由于决策树的每个分支会划分训练数据，在训练树的低节点，可用的训练数据量可能会变得非常小。因此，这些较低的决策节点可能**过拟合**训练集，学习模式反映训练集的特质而不是问题底层显著的语言学模式。对这个问题的一个解决方案是当训练数据量变得太小时停止分裂节点。另一种方案是长出一个完整的决策树，但随后进行**剪枝剪去**在开发测试集上不能提高性能的决策节点。

决策树的第二个问题是，它们强迫特征按照一个特定的顺序进行检查，即使特征可能是相对独立的。由于决定树顶部附近的空间有限，大部分这些特征将需要在树中的许多不同的分支中重复。因为越往树的下方，分支的数量成指数倍增长，重复量可能非常大。

一个相关的问题是决策树不善于利用对正确的标签具有较弱预测能力的特征。由于这些特征的影响相对较小，它们往往出现在决策树非常低的地方。决策树学习的时间远远不够用到这些特征，也不能留下足够的训练数据来可靠地确定它们应该有什么样的影响。如果能够在整个训练集中看看这些特征的影响，那么也许能够做出一些关于它们是如何影响标签的选择的结论，。

决策树需要按一个特定的顺序检查特征的事实，限制了它们的利用相对独立的特征的能力。

下面讨论的朴素贝叶斯分类方法克服了这一限制，允许所有特征“并行”的起作用。

**6.5 朴素贝叶斯分类器**

在**朴素贝叶斯**分类器中，每个特征都得到发言权，来确定哪个标签应该被分配到一个给定的输入值。为一个输入值选择标签，朴素贝叶斯分类器以计算每个标签的**先验概率**开始，它由在训练集上检查**每个标签的频率**来确定。之后，每个特征的贡献与它的先验概率组合，得到每个标签的似然估计。似然估计最高的标签会分配给输入值。

个别特征对整体决策作出自己的贡献，通过“投票反对”那些不经常出现的特征的标签。 特别是，每个标签的似然得分由于与输入值具有此特征的标签的概率相乘而减小。似然得分结果可以认为是从具有给定的标签和特征集的训练集中随机选取的值的概率的估计，假设所有特征概率是独立的。

**潜在概率模型**

每个输入值是通过首先为那个输入值选择一个类标签，然后产生每个特征的方式产生的，每个特征与其他特征完全独立。这简化的假设，称为**朴素贝叶斯假设**(或**独立性假设**)，使得它更容易组合不同特征的贡献，因为我们不必担心它们相互影响。

即概率论中的**贝叶斯公式**的推导过程。

P(label|features) = P(features, label)/P(features)

P(features) = Σlabel ∈ labels P(features, label)

P(features, label) = P(label) × P(features|label)

因为特征都是独立的(给定标签)，可以分离每个独立特征的概率:

P(features, label) = P(label) × ∏f ∈ featuresP(f|label)

**零计数和平滑**

最简单的方法计算 P(f|label)，特征 f 对标签 label 的标签可能性的贡献，是取得具有给定特征和给定标签的训练实例的百分比:

P(f|label) = count(f, label)/count(label)

然而，当训练集中有特征从来没有和给定标签一起出现时，这种简单的方法会产生一个问题。在这种情况下，P(f|label)计算值将是 0，这将导致给定标签的标签可能性为 0。 从而，输入将永远不会被分配给这个标签，不管其他特征有多么适合这个标签。

这里的基本问题与计算 P(f|label)有关，对于给定标签输入将具有一个特征的概率。 特别的，仅仅因为在训练集中没有看到特征/标签组合出现，并不意味着该组合不会出现。

虽然 count(f,label)/count(label)当 count(f,label)相对高时是 P(f|label)的好的估计，但是当 count(f) 变小时这个估计变得不那么可靠。因此，建立朴素贝叶斯模型时，通常采用更复杂的技术，被称为**平滑**技术，用于计算 P(f|label)，给定标签的特征的概率。例如:给定标签的 一个特征的概率的**期望似然估计**基本上给每个 count(f,label)值加 0.5，**Heldout 估计**使用一个heldout 语料库计算特征频率与特征概率之间的关系。nltk.probability 模块提供了多种平滑技术的支持。

**非二元特征**

前面的推导基于假设每个特征是二元的，即每个输入要么有这个特征要么没有。标签值特征可通过用二元特征替换它们，将它们转换为二元特征。数字特征可以通过**装箱**转换为二元特征，装箱是用特征，如“4<X <6”，替换它们。

另一种方法是使用回归方法模拟数字特征的概率。计算每个标签的输入的均值和方差。

**独立的朴素**

朴素贝叶斯分类器被称为“nai ve(天真、朴素)”的原因是它不切实际地假设所有特征相互独立(给定标签)。事实上，几乎所有现实世界的问题含有的特征都不同程度的彼此依赖。

解决问题的一个方法是，分类器“双重计数”高度相关的特征的影响，将分类器推向更接近给定的标签而不是合理的标签。

另一种方法是，建立包含相互依赖特征的分类器。对于这些功能，重复的信息可能会被训练集赋予比合理的更多的比重。

**双重计数**

考虑在训练中特征的贡献之间可能的相互作用，然后可以使用这些相互作用调整独立特征所作出的贡献。

重写计算标签的可能性的方程，分离出每个功能(或标签) 所作出的贡献:

P(features, label) = w[label] × ∏f ∈ features w[f, label]

在这里，w[label]是一个给定标签的“初始分数”，w[f, label]是给定特征对一个标签的可能性所作的贡献。称这些值 w[label]和 w[f, label]为模型的**参数**或**权重**。使用朴素贝叶斯算法，单独设置这些参数:

w[label] = P(label)

w[f, label] = P(f|label)

**6.6 最大熵分类器**

**最大熵分类器**使用了一个与朴素贝叶斯分类器使用的模型非常相似的模型。不是使用概率设置模型的参数，它使用搜索技术找出一组将最大限度地提高分类器性能的参数。特别的， 它查找使训练语料的**整体可能性**最大的参数组：

P(features) = Σx ∈ corpus P(label(x)|features(x))

其中 P(label|features)，一个特征为 features 将有类标签 label的输入的概率，被定义为:

P(label|features) = P(label, features)/Σlabel P(label, features)

由于相关特征之间潜在的复杂的相互作用，最大熵分类器采用**迭代优化**技术选择模型参数，该技术 用随机值初始化模型的参数，然后反复优化这些参数，使它们更接近最优解，但这个技术不一定提供方法来确定是否已经达到最佳值。因为迭代优化技术使得最大熵分类器需要花费很长时间来学习，应尽量选择比较快的共轭梯度(Conjugate Gradient，CG) 和 BFGS 优化方法而不是广义迭代缩放(Generalized Iterative Scaling，GIS)或改进的迭代缩放(Improv ed Iterative Scaling，IIS)。

**最大熵模型**

最大熵分类器模型是一朴素贝叶斯分类器模型的泛化。它同样是为给定的输入值计算每个标签的可能性，通过将适合于输入值和标签的参数乘在一起，但是它留给用户来决定什么样的标签和特征组合应该得到自己的参数。特别的，它可以使用一个单独的参数关联一个特征与一个以上的标签;或者关联一个以上的特征与一个给定的标签。这有时会允许模型“概括”相关的标签或特征之间的一些差异。

每个接收它自己的参数的标签和特征的组合被称为一个**联合特征**。请注意，**联合特征**是有**标签的值**的属性，而(简单)特征是未加标签的值的属性。

描述和讨论最大熵模型的文字中，术语“特征 features”往往指联合特征;术语“上下文 contexts”指我们一直说的(简单)特征。

通常情况下，用来构建最大熵模型的联合特征完全镜像朴素贝叶斯模型使用的联合特征。特别的，每个标签定义的联合特征对应于 w[label]，每个(简单)特征和标签组合定义的联合特征对应于w[f, label]。给定一个最大熵模型的联合特征，分配到一个给定输入的标签的得分与适用于该输入和标签的联合特征相关联的参数的简单的乘积。

P(input, label) = ∏joint-features(input,label)w[joint-feature]

**熵的最大化**

进行最大熵分类的直觉是建立一个模型，捕捉单独的联合特征的频率

**最大熵原理**是说在与我们所知道的一致的的分布中，我们会选择熵最高的（最少无根据的）（均匀分布）。

特别的，对于每个联合特征，最大熵模型计算该特征的“经验频率”——即它出现在训练集中的频率。然后，它搜索熵最大的分布，同时也预测每个联合特征正确的频率。

**生成式分类器对比条件式分类器**

朴素贝叶斯分类器是一个**生成式**分类器的例子，建立一个模型，预测P(input, label)， 即(input, label)对的联合概率。因此，生成式模型可以用来回答下列问题:

1. 一个给定输入的最可能的标签是什么?

2. 对于一个给定输入，一个给定标签有多大可能性?

3. 最有可能的输入值是什么?

4. 一个给定输入值的可能性有多大?

5. 一个给定输入具有一个给定标签的可能性有多大?

6. 对于一个可能有两个值中的一个值(但我们不知道是哪个)的输入，最可能的标签是什么?

另一方面，最大熵分类器是**条件式**分类器的一个例子。条件式分类器建立模型，预测 P(label|input)——一个给定输入值的标签的概率。因此，条件式模型仍然可以被用来回答问题 1 和 2。然而，条件式模型不能用来回答剩下的问题 3-6。

一般情况下，生成式模型确实比条件式模型强大，因为我们可以从联合概率 P(input, label)计算出条件概率 P(label|input)，但反过来不行。然而，这种额外的能力是要付出代价的。由于该模型更强大，它也有更多的“自由参数”需要学习。而训练集的大小是固定的。因此，使用一个更强大的模型时，可用来训练每个参数的值的数据也更少，使其难以找到最佳参数值。结果是一个生成式模型回答问题 1 和 2 可能不会与条件式模型一样好，因为条件式模型可以集中精力在这两个问题上。然而，如果确实需要像 3-6 问题的答案，那么只能使用生成式模型。

**6.7 为语言模式建模**

在分类器中建立的这些用来捕捉语言模式的明确的模型有两个重要目的:它们帮助我们了解语言模式，它们可以被用来预测新的语言数据。

**模型告诉我们什么?**

处理语言模型时一个重要的考虑因素是描述性模型与解释性模型之间的区别。描述性模型捕获数据中的模式，但它们并不提供任何有关数据包含这些模式的原因的信息。与此相反，解释性模型试图捕捉造成语言模式的属性和关系。描述性模型提供数据内相关性的信息，而解释性模型再进一步假设因果关系。

大多数从语料库自动构建的模型是描述性模型;换句话说，它们可以告诉我们哪些特征与一个给定的模式或结构有关，但它们不一定能告诉我们这些特征和模式之间如何关联。如果我们的目标是理解语言模式，那么我们就可以使用哪些特征是相关的这一信息作为出发点，设计进一步的实验弄清特征与模式之间的关系。另一方面，如果我们只是对利用该模型进行预测，例如:作为一种语言处理系统的一部分，感兴趣，那么我们可以使用该模型预测新的数据，而不用担心潜在的因果关系的细节。