

# 第 4 章 利用sklearn构建朴素贝叶斯分类器

在scikit-learn中, 一共有3个朴素贝叶斯的分类算法类. 分别是GaussianNB, MultinomialNB和BernoulliNB. 其中GaussianNB是高斯分布的朴素贝叶斯, MultinomialNB是多项式分布的朴素贝叶斯, BernoulliNB是为伯努利分布的朴素贝叶斯, 而ComplementNB是互补朴素贝叶斯.

## sklearn.naive\_bayes : Naive Bayes

The `sklearn.naive_bayes` module implements Naive Bayes algorithms. These are supervised learning methods based on applying Bayes' theorem with strong (naive) feature independence assumptions.

**User guide:** See the [Naive Bayes](#) section for further details.

<code>naive_bayes.BernoulliNB</code> ([alpha, binarize, ...])	Naive Bayes classifier for multivariate Bernoulli models.
<code>naive_bayes.GaussianNB</code> ([priors, var_smoothing])	Gaussian Naive Bayes (GaussianNB)
<code>naive_bayes.MultinomialNB</code> ([alpha, ...])	Naive Bayes classifier for multinomial models
<code>naive_bayes.ComplementNB</code> ([alpha, fit_prior, ...])	The Complement Naive Bayes classifier described in Rennie et al.

前面的实战中, 主要是属于多项式分布的朴素贝叶斯.

## — naive\_bayes.MultinomialNB

### sklearn.naive\_bayes.ComplementNB

```
class sklearn.naive_bayes. ComplementNB (alpha=1.0, fit_prior=True, class_prior=None, norm=False) \[source\]
```

The Complement Naive Bayes classifier described in Rennie et al. (2003).

The Complement Naive Bayes classifier was designed to correct the “severe assumptions” made by the standard Multinomial Naive Bayes classifier. It is particularly suited for imbalanced data sets.

Read more in the [User Guide](#).

<b>Parameters:</b>	<b>alpha : float, optional (default=1.0)</b> Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
	<b>fit_prior : boolean, optional (default=True)</b> Only used in edge case with a single class in the training set.
	<b>class_prior : array-like, size (n_classes,), optional (default=None)</b> Prior probabilities of the classes. Not used.
	<b>norm : boolean, optional (default=False)</b> Whether or not a second normalization of the weights is performed. The default behavior mirrors the implementations found in Mahout and Weka, which do not follow the full algorithm described in Table 9 of the paper.

### 参数说明

ComplementNB中主要有四个参数:

- **alpha**: 浮点型可选参数, 默认为1.0, 其实就是添加拉普拉斯平, 如果这个参数设置为0, 就是不添加平滑

- **fit\_prior**: 布尔型可选参数, 默认为True。布尔参数fit\_prior表示是否要考虑先验概率, 如果是false,则所有的样本类别输出都有相同的类别先验概率。否则可以自己用第三个参数class\_prior输入先验概率, 或者不输入第三个参数class\_prior让MultinomialNB自己从训练集样本来计算先验概率, 此时的先验概率为 $P(Y=C_k)=m_k/m$ 。其中m为训练集样本总数量,  $m_k$ 为输出为第k类别的训练集样本数。
- **class\_prior**: 可选参数, 默认为None。
- **norm**: 布尔型参数, 默认为False。是否执行权重的第二次标准化

## 主要方法

同时, MultinomialNB还有一些方法, 如下:

Methods	
<code>fit (self, X, y[, sample_weight])</code>	Fit Naive Bayes classifier according to X, y
<code>get_params (self[, deep])</code>	Get parameters for this estimator.
<code>partial_fit (self, X, y[, classes, sample_weight])</code>	Incremental fit on a batch of samples.
<code>predict (self, X)</code>	Perform classification on an array of test vectors X.
<code>predict_log_proba (self, X)</code>	Return log-probability estimates for the test vector X.
<code>predict_proba (self, X)</code>	Return probability estimates for the test vector X.
<code>score (self, X, y[, sample_weight])</code>	Returns the mean accuracy on the given test data and labels.
<code>set_params (self, **params)</code>	Set the parameters of this estimator.

下面是项目三: 使用朴素贝叶斯分类器从个人广告中获取区域倾向, 分别用**自行构建的分类过程**, 和**利用sklearn进行分类**.

# 二 用自行构建的分类器对Rss数据分类

## (一) 项目概述

广告商往往想知道关于一个人的一些特定人口统计信息, 以便能更好地定向推销广告。

我们将分别从美国的两个城市选取一些人, 通过分析这些人发布的信息, 来比较这两个城市的人们在广告用词上是否不同。如果结论确实不同, 那么他们各自常用的词是哪些, 从人们的用词当中, 我们能否对不同城市的人所关心的内容有所了解。

## (二) 项目流程

收集数据: 从RSS源收集内容, 这里需要对RSS源构建一个接口  
 准备数据: 将文本文件解析成词条向量  
 分析数据: 检查词条确保解析的正确性  
 训练算法: 使用我们之前建立的 `trainNB0()` 函数  
 测试算法: 观察错误率, 确保分类器可用。可以修改切分程序, 以降低错误率, 提高分类结果  
 使用算法: 构建一个完整的程序, 封装所有内容。给定两个RSS源, 改程序会显示最常用的公共词

## (三) 具体实现过程

### 1 安装feedparse包

先通过pip安装feedparse包, 用来解析RSS源.

如:

```
import feedparser
ny = feedparser.parse('http://www.nasa.gov/rss/dyn/image_of_the_day.rss')
print(ny['entries'])
print(len(ny['entries']))
```

```
feature/surround-sound-orion-service-module-for-artemis-1-undergoes-acoustic-tests'}, {'length': '1996530', 'type': 'image/
jpeg', 'href': 'http://www.nasa.gov/sites/default/files/thumbnails/image/orion_service_module_acoustic_test_co.jpg', 'rel':
'enclosure'}], 'link': 'http://www.nasa.gov/image-feature/surround-sound-orion-service-module-for-artemis-1-undergoes-acoustic-
tests', 'summary': 'Surround Sound - Orion service module for Artemis 1 undergoes acoustic tests', 'summary_detail': {'type':
'text/html', 'language': 'en', 'base': 'http://www.nasa.gov/', 'value': 'Surround Sound - Orion service module for Artemis 1
undergoes acoustic tests'}, 'id': 'http://www.nasa.gov/image-feature/surround-sound-orion-service-module-for-artemis-1-
undergoes-acoustic-tests', 'guidislink': False, 'published': 'Thu, 30 May 2019 09:51 EDT', 'published_parsed':
time.struct_time(tm_year=2019, tm_mon=5, tm_mday=30, tm_hour=13, tm_min=51, tm_sec=0, tm_wday=3, tm_yday=150, tm_isdst=0),
'source': {'href': 'http://www.nasa.gov/rss/dyn/image_of_the_day.rss', 'title': 'NASA Image of the Day'}, {'title': 'How to
Travel at (Nearly) the Speed of Light', 'title_detail': {'type': 'text/plain', 'language': 'en', 'base': 'http://
www.nasa.gov/', 'value': 'How to Travel at (Nearly) the Speed of Light'}, 'links': [{'rel': 'alternate', 'type': 'text/html',
'href': 'http://www.nasa.gov/image-feature/how-to-travel-at-nearly-the-speed-of-light'}, {'length': '161098', 'type': 'image/
jpeg', 'href': 'http://www.nasa.gov/sites/default/files/thumbnails/image/g2014-018_mms_narrated_orbitv4_ipod_lg.594_print.jpg',
'rel': 'enclosure'}], 'link': 'http://www.nasa.gov/image-feature/how-to-travel-at-nearly-the-speed-of-light', 'summary': 'Learn
about the three ways to travel at (nearly) the speed of light.', 'summary_detail': {'type': 'text/html', 'language': 'en',
'base': 'http://www.nasa.gov/', 'value': 'Learn about the three ways to travel at (nearly) the speed of light.', 'id':
'http://www.nasa.gov/image-feature/how-to-travel-at-nearly-the-speed-of-light', 'guidislink': False, 'published': 'Wed, 29 May
2019 11:19 EDT', 'published_parsed': time.struct_time(tm_year=2019, tm_mon=5, tm_mday=29, tm_hour=15, tm_min=19, tm_sec=0,
tm_wday=2, tm_yday=149, tm_isdst=0), 'source': {'href': 'http://www.nasa.gov/rss/dyn/image_of_the_day.rss', 'title': 'NASA
Image of the Day'}, {'title': 'Jezero Crater, Mars 2020's Landing Site', 'title_detail': {'type': 'text/plain', 'language':
'en', 'base': 'http://www.nasa.gov/', 'value': 'Jezero Crater, Mars 2020's Landing Site'}, 'links': [{'rel': 'alternate',
'type': 'text/html', 'href': 'http://www.nasa.gov/image-feature/jezero-crater-mars-2020s-landing-site'}, {'length': '550154',
'type': 'image/jpeg', 'href': 'http://www.nasa.gov/sites/default/files/thumbnails/image/pia23239.jpg', 'rel': 'enclosure'}],
'link': 'http://www.nasa.gov/image-feature/jezero-crater-mars-2020s-landing-site', 'summary': 'This false color image shows
part of an unnamed crater in Mars' Arabia Terra.', 'summary_detail': {'type': 'text/html', 'language': 'en', 'base': 'http://
www.nasa.gov/', 'value': 'This false color image shows part of an unnamed crater in Mars' Arabia Terra.', 'id': 'http://
www.nasa.gov/image-feature/jezero-crater-mars-2020s-landing-site', 'guidislink': False, 'published': 'Tue, 28 May 2019 13:16
EDT', 'published_parsed': time.struct_time(tm_year=2019, tm_mon=5, tm_mday=28, tm_hour=17, tm_min=16, tm_sec=0, tm_wday=1,
tm_yday=148, tm_isdst=0), 'source': {'href': 'http://www.nasa.gov/rss/dyn/image_of_the_day.rss', 'title': 'NASA Image of the
Day'}}}]
60
```

In [4]:

## 2 文档词袋模型

我们将每个词的出现与否作为一个特征，这可以被描述为 **词集模型(set-of-words model)**。如果一个词在文档中出现不止一次，这可能意味着包含该词是否出现在文档中所不能表达的某种信息，这种方法被称为 **词袋模型(bag-of-words model)**。在词袋中，每个单词可以出现多次，而在词集中，每个词只能出现一次。为适应词袋模型，需要对函数 `setOfWords2Vec()` 稍加修改，修改后的函数为 `bagOfWords2Vec()`。

如下给出了基于词袋模型的朴素贝叶斯代码。它与函数 `setOfWords2Vec()` 几乎完全相同，唯一不同的是每当遇到一个单词时，它会增加词向量中的对应值，而不只是将对应的数值设为 1。

```
"""
```

函数说明：根据 `coclaList` 里的词汇表，将输入的 `inputSet` 向量化，向量的每个元素为 1 或者 0

Parameters:

`vocabList` - `createVocabList` 函数返回的词汇表

`inputSet` - 要输入的切分词条列表

Returns:

`returnVec` - 向量化后的文档

```
"""
```

```
def bagOfWords2VecMN(vocabList, inputSet):
```

```
    # 初始化 returnVec 列表，列表长度为 vocabList 长度，元素为 0
```

```
    returnVec = [0] * len(vocabList)
```

```
    # 遍历 inputSet 所有单词，如果出现了词汇表中的单词，则将输出的文档向量中的对应值设为 1
```

```
    for word in inputSet:
```

```
        if word in vocabList:
```

```
            returnVec[vocabList.index(word)] += 1
```

```
return returnVec
```

### 3 对文档中的词进行排序

```
"""
```

函数说明：遍历词汇表中的每个词并统计它在文本中出现的次数，然后根据出现次数从高到低对词典进行排序，

最后返回排序最高的5个单词

Parameters:

**vocabList** - 词汇表

**fullText** - 传入的文本

Returns:

**sortedFreq[:5]** - 返回出现次数最高的5个单词

```
"""
```

```
def calcMostFreq(vocabList, fullText):
    # 初始化一个频率字典
    freqDict = {}
    # 遍历词汇表中的每一个词
    for token in vocabList:
        # 统计每个词在fullText文本中出现的次数，并传入到字典中
        freqDict[token] = fullText.count(token)
    # 根据每个词出现的次数从高到底对字典进行排序
    # sortedFreq返回的是键值对列表
    sortedFreq = sorted(freqDict.items(), key=operator.itemgetter(1),
reverse=True)
    return sortedFreq[:5]
```

### 4 对贝叶斯垃圾邮件分类器进行自动化处理 (localWords函数)

函数localWords()与程序清单4-5中的spamTest()函数几乎相同，区别在于这里访问的是RSS源 而不是文件。然后调用函数calcMostFreq()来获得排序最高的30个单词并随后将它们移除。函数的剩余部分与spamTest()基本类似，不同的是最后一行要返回下面要用到的值。

```
"""
```

函数说明：对贝叶斯垃圾邮件分类器进行自动化处理，与spam的区别就是文件来源，以及去除出现最高的30个词，其他一样

Parameters:

**feed1** - RSS来源1

**feed0** - Rss来源0

Returns:

对测试集中的每封邮件进行分类，若邮件分类错误，则错误数加 1，最后返回总的错误百分比

```
"""
```

```
def localWords(feed1, feed0):
    docList = []
    classList=[]
    fullText = []
    #返回两者的最小长度
    minLen = min(len(feed1['entries']),len(feed0['entries']))
    for i in range(minLen):
```

```

wordList = textParse(feed1['entries'][i]['summary'])
docList.append(wordList)
fullText.extend(wordList)
classList.append(1)

wordList = textParse(feed0['entries'][i]['summary'])
docList.append(wordList)
fullText.extend(wordList)
classList.append(0)

vocabList = createVocabList(docList)
top30Words = calcMostFreq(vocabList, fullText)
#去掉出现次数最高的那些词
for pariW in top30Words:
    if pariW[0] in vocabList:
        vocabList.remove(pariW[0])

trainingSet = list(range(2*minLen))
testSet = []
#随机取20个作为测试，剩下的作为训练
for i in range(8):
    randIndex = int(random.uniform(0, len(trainingSet)))
    testSet.append(trainingSet[randIndex])
    del (trainingSet[randIndex])

trainMat = []
trainClasses = []
for docIndex in trainingSet:
    trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
    trainClasses.append(classList[docIndex])
p0V, p1V, pSpam = trainNB0(np.array(trainMat), np.array(trainClasses))
errorCount = 0
for docIndex in testSet:
    wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
    if classifyNB(np.array(wordVector), p0V, p1V, pSpam) !=
classList[docIndex]:
        errorCount += 1
print('错误率: %.2f%%' % (float(errorCount) / len(testSet) * 100))
return vocabList, p0V, p1V

```

## 5 RSS源的选择

因为书上的两个数据源都是时间较早的, 现在都已经失效了, 返回不了数据, 因此对fee1和fee0两个RSS源进行修改, 具体如下:

```

feed1= feedparser.parse('http://www.nasa.gov/rss/dyn/image_of_the_day.rss')
feed0 =
feedparser.parse('http://www.cppblog.com/kevinlynx/category/6337.html/rss')

```

## 6 汇总代码

把用的函数汇总一起, 就可以得到

```
import numpy as np
import operator
import feedparser
import re
import random
```

"""

函数说明：根据coclaList里的词汇表，将输入的inputSet向量化,向量的每个元素为1或者0

Parameters:

vocabList - createVocabList函数返回的词汇表

inputSet - 要输入的切分词条列表

Returns:

returnVec - 向量化后的文档

"""

```
def bagOfWords2VecMN(vocabList, inputSet):
```

```
# 初始化returnVec列表，列表长度为vocabList长度，元素为0
```

```
returnVec = [0] * len(vocabList)
```

```
# 遍历inputSet所有单词,如果出现了词汇表中的单词，则将输出的文档向量中的对应值设为1
```

```
for word in inputSet:
```

```
    if word in vocabList:
```

```
        returnVec[vocabList.index(word)] += 1
```

```
return returnVec
```

"""

函数说明：将切分的样本词条整理成不重复的词条

Parameters:

dataSet - 样本数据集(即之前切分的样本词条)

Returns:

vocabSet - 不重复的词条

"""

```
def createVocabList(dataSet):
```

```
# 初始化一个空的集合(集合是不重复的)
```

```
vocaSet = set([])
```

```
# 遍历dataSet每个元素
```

```
for document in dataSet:
```

```
    # 取并集
```

```
    vocaSet = vocaSet | set(document)
```

```
# 返回不重复的列表
```

```
return list(vocaSet)
```

"""

函数说明：接收一个大字符串并将其解析为字符串列表

Parameters:

bigString - 一个大字符串

Returns:

lowerString - 去掉少于2个字符的字符串，并将所有字符串转换为小写，返回字符串列表

"""

```
def textParse(bigString):
```

```
# '\w*' 表示除单词,数字意外的任意字符串
```

#两种模式,一种是包含的正则表达式的字符串创建模式对象,一种是直接使用  
#下面的也可以这样写

```
"""
regEx = re.compile('\w*')
listOfTokens = regEx.split(a)
"""

listOfTokens = re.split('\w+',bigString) #listOfTokens本身也是列表
#列表生成式
lowerString = [tok.lower() for tok in listOfTokens if len(tok) > 2]
return lowerString
```

"""

函数说明: 朴素贝叶斯分类器训练函数

Parameters:

trainMatrix - 训练文档矩阵, 即函数setOfWords2Vec返回的returnVec构成的矩阵  
trainCategory - 训练类别(标签)向量, 即函数loadDataSet返回的classVec

Returns:

p0Vect - 非侮辱类的条件概率数组  
p1Vect - 侮辱类的条件概率数组  
pAbusive - 文档属于侮辱类的概率

"""

```
def trainNB0(trainMatrix, trainCategory):
    # 计算文档的数目, 即trainMatrix列表的长度(和trainCategory数目是对应的,相同的)
    # postingList每一行是一个文档, 对应trainMatrix中的一个元素, 也即是一个returnVec
    numTrainDocs = len(trainMatrix)
    # 单词个数(trainMatrix[0]=trainMatrix[1]=....)
    numWords = len(trainMatrix[0])
    # 侮辱性文件的出现概率, 即trainCategory中所有的1的个数,
    # 代表的就是多少个侮辱性文件, 与文件的总数相除就得到了侮辱性文件的出现概率
    pAbusive = sum(trainCategory) / float(numTrainDocs)

    # 初始化词条出现次数, 初始为0, 改进为1
    p0Num = np.ones(numWords)
    p1Num = np.ones(numWords)
    # 分母初始化为0啊, 改进为2
    p0Denom = 2
    p1Denom = 2
    for i in range(numTrainDocs):
        # 是否是侮辱性文件
        if trainCategory[i] == 1:
            # 如果是侮辱性文件, 对侮辱性文件的向量进行加和
            # 注意array是可以直接和对应长度的列表相加的.
            p1Num += trainMatrix[i] # [0,1,1,...] + [0,1,1,...]->[0,2,2,...]
            # 对向量中的所有元素进行求和, 也就是计算所有侮辱性文件中出现的单词总数
            p1Denom += sum(trainMatrix[i])
        else:
            p0Num += trainMatrix[i]
            p0Denom += sum(trainMatrix[i])

    # 类别1, 即侮辱性文档的[P(F1|C1), P(F2|C1), P(F3|C1), P(F4|C1), P(F5|C1)...]列表
    # 即在1类别下, 每个单词出现的概率, 改进为对数
    p1Vect = np.log(p1Num / p1Denom) # 如# [1,2,3,5]/90->[1/90,...]
    # 类别0, 即正常文档的[P(F1|C0), P(F2|C0), P(F3|C0), P(F4|C0), P(F5|C0)...]列表
    # 即在0类别下, 每个单词出现的概率, 改进为对数
```



```
p0Vect = np.log(p0Num / p0Denom)
return p0Vect, p1Vect, pAbusive
```

"""

函数说明：朴素贝叶斯分类函数

Parameters:

vec2Classify - 待分类数组,如[1,0,1,...],注意是np.array类型的

p0Vec - trainNB返回的p0Vect, 也就是类别为0,即正常类文档的条件概率数组,  
即

[log(P(X1|C0)),log(P(X2|C0)),log(P(X3|C0)),log(P(X4|C0)),log(P(X5|C0))....]数组

p1Vec - trainNB返回的p1Vect,也即是类别为1,即侮辱类文档的条件概率数组,  
即

[log(P(X1|C1)),log(P(X2|C1)),log(P(X3|C1)),log(P(X4|C1)),log(P(X5|C1))....]数组

pClass1 - 文档属于侮辱类的概率,即trainNB中的pAbusive

Returns:

0 - 文档属于正常类,非侮辱类

1 - 文档属于侮辱类

"""

```
def classifyNB(vec2Classify, p0Vec, p1Vec, pClass1):
```

```
# 因为p1Vect已经取了对数,所以要计算p(x_{i}|c1)相乘,也就是计算求和,最后再加上
log(pClass1)即可
```

```
# 同时, 因为分母p(w)都是一致的, 所以只需要计算并比较分子大小即可
```

```
# 同时, 根据训练得到的p1Vec,即条件概率数组, 乘以待分类向量,即可得到待分类的条件概率
```

```
p1 = sum(vec2Classify * p1Vec) + np.log(pClass1)
```

```
p0 = sum(vec2Classify * p0Vec) + np.log(1 - pClass1)
```

```
if p1 > p0:
```

```
    return 1
```

```
else:
```

```
    return 0
```

"""

函数说明：遍历词汇表中的每个词并统计它在文本中出现的次数，然后根据出现次数从高到低对词典进行排序，

最后返回排序最高的30个单词

Parameters:

vocabList - 词汇表

fullText - 传入的文本

Returns:

sortedFreq[:30] - 返回出现次数最高的30个单词

"""

```
def calcMostFreq(vocabList, fullText):
```

```
#初始化一个频率字典
```

```
freqDict = {}
```

```
#遍历词汇表中的每一个词
```

```
for token in vocabList:
```

```
    #统计每个词在fullText文本中出现的次数, 并传入到字典中
```

```
    freqDict[token] = fullText.count(token)
```

```
#根据每个词出现的次数从高到底对字典进行排序
```

```
#sortedFreq返回的是键值对列表
```

```
sortedFreq = sorted(freqDict.items(),
```

```
key=operator.itemgetter(1),reverse=True)
```

```
return sortedFreq[:5]
```



"""

函数说明：对贝叶斯垃圾邮件分类器进行自动化处理，与spam的区别就是文件来源，以及去除出现最高的30个词，其他一样

Parameters:

feed1 - RSS来源1

feed0 - Rss来源0

Returns:

对测试集中的每封邮件进行分类，若邮件分类错误，则错误数加 1，最后返回总的错误百分比

"""

```
def localWords(feed1, feed0):
    docList = []
    classList = []
    fullText = []
    #返回两者的最小长度
    minLen = min(len(feed1['entries']), len(feed0['entries']))
    for i in range(minLen):
        wordList = textParse(feed1['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(1)

        wordList = textParse(feed0['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(0)

    vocabList = createVocabList(docList)
    top30Words = calcMostFreq(vocabList, fullText)
    #去掉出现次数最高的那些词
    for pariW in top30Words:
        if pariW[0] in vocabList:
            vocabList.remove(pariW[0])

    trainingSet = list(range(2*minLen))
    testSet = []
    #随机取20个作为测试，剩下的作为训练
    for i in range(8):
        randIndex = int(random.uniform(0, len(trainingSet)))
        testSet.append(trainingSet[randIndex])
        del (trainingSet[randIndex])

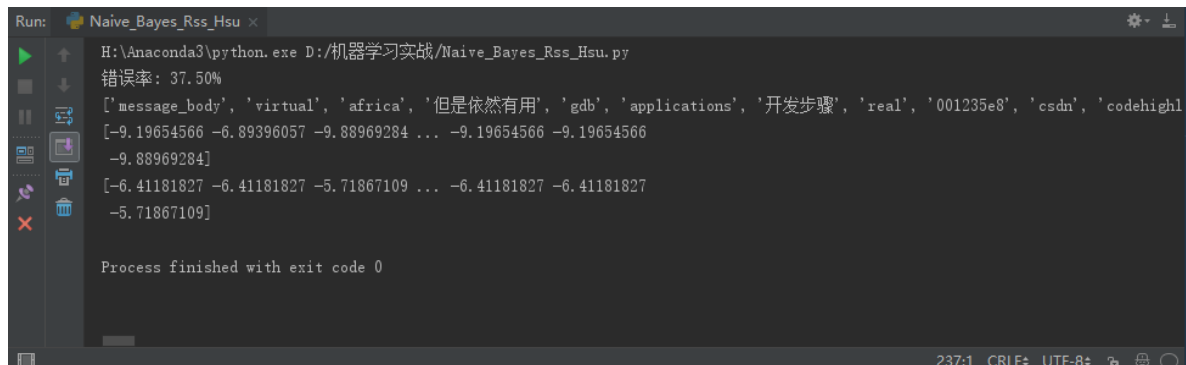
    trainMat = []
    trainClasses = []
    for docIndex in trainingSet:
        trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
        trainClasses.append(classList[docIndex])
    p0V, p1V, pSpam = trainNB0(np.array(trainMat), np.array(trainClasses))
    errorCount = 0
    for docIndex in testSet:
        wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
        if classifyNB(np.array(wordVector), p0V, p1V, pSpam) !=
classList[docIndex]:
            errorCount += 1
    print('错误率: %.2f%%' % (float(errorCount) / len(testSet) * 100))
    return vocabList, p0V, p1V
```

```

if __name__ == '__main__':
    feed1= feedparser.parse('http://www.nasa.gov/rss/dyn/image_of_the_day.rss')
    feed0 =
feedparser.parse('http://www.cppblog.com/kevinlynx/category/6337.html/rss')
    vocabList, p0V, p1V = localWords(feed1, feed0)
    print(vocabList)
    print(p0V)
    print(p1V)

```

运行的结果如下:



```

Run: Naive_Bayes_Rss_Hsu x
H:\Anaconda3\python.exe D:/机器学习实战/Naive_Bayes_Rss_Hsu.py
错误率: 37.50%
['message_body', 'virtual', 'africa', '但是依然有用', 'gdb', 'applications', '开发步骤', 'real', '001235e8', 'csdn', 'codehigh1
[-9.19654566 -6.89396057 -9.88969284 ... -9.19654566 -9.19654566
-9.88969284]
[-6.41181827 -6.41181827 -5.71867109 ... -6.41181827 -6.41181827
-5.71867109]

Process finished with exit code 0
237:1 CRLF UTF-8

```

然后, 来看看用sklearn中贝叶斯模块如何进行快速的分类.

## 三 利用sklearn中贝叶斯模块进行分类

我们知道sklearn中每个对应的模块都有fit方法, fit方法就是拟合或者说训练的过程. 同时还有predict方法, 就是预测或者叫分类. 因此, 上面的代码里, 我们就不需要trainNB0函数, 也不需要classifyNB函数, 同时对localWords就行修改, 最后的代码如下:

```

import numpy as np
import operator
import feedparser
import re
import random
from sklearn.naive_bayes import MultinomialNB

"""
函数说明: 根据vocabList里的词汇表, 将输入的inputSet向量化, 向量的每个元素为1或者0

Parameters:
    vocabList - createVocabList函数返回的词汇表
    inputSet - 要输入的切分词条列表

Returns:
    returnVec - 向量化后的文档
"""

def bagOfWords2VecMN(vocabList, inputSet):

```

```

# 初始化returnVec列表, 列表长度为vocabList长度, 元素为0
returnVec = [0] * len(vocabList)
# 遍历inputSet所有单词, 如果出现了词汇表中的单词, 则将输出的文档向量中的对应值设为1
for word in inputSet:
    if word in vocabList:
        returnVec[vocabList.index(word)] += 1
return returnVec

```

"""

函数说明: 将切分的样本词条整理成不重复的词条

Parameters:

dataSet - 样本数据集(即之前切分的样本词条)

Returns:

vocabSet - 不重复的词条

"""

```

def createVocabList(dataSet):
    # 初始化一个空的集合(集合是不重复的)
    vocaSet = set([])
    # 遍历dataSet每个元素
    for document in dataSet:
        # 取并集
        vocaSet = vocaSet | set(document)
    # 返回不重复的列表
    return list(vocaSet)

```

"""

函数说明: 接收一个大字符串并将其解析为字符串列表

Parameters:

bigString - 一个大字符串

Returns:

lowerString - 去掉少于2个字符的字符串, 并将所有字符串转换为小写, 返回字符串列表

"""

```

def textParse(bigString):
    # '\w*'表示除单词, 数字意外的任意字符串
    # 两种模式, 一种是包含的正则表达式的字符串创建模式对象, 一种是直接使用
    # 下面的也可以这样写

    """
    regex = re.compile('\w+')
    listOfTokens = regex.split(a)
    """
    listOfTokens = re.split('\w+', bigString) # listOfTokens本身也是列表
    # 列表生成式
    lowerString = [tok.lower() for tok in listOfTokens if len(tok) > 2]
    return lowerString

```

```
"""
```

函数说明：遍历词汇表中的每个词并统计它在文本中出现的次数，然后根据出现次数从高到低对词典进行排序，

最后返回排序最高的5个单词

Parameters:

`vocabList` - 词汇表

`fullText` - 传入的文本

Returns:

`sortedFreq[:5]` - 返回出现次数最高的5个单词

```
"""
```

```
def calcMostFreq(vocabList, fullText):
    # 初始化一个频率字典
    freqDict = {}
    # 遍历词汇表中的每一个词
    for token in vocabList:
        # 统计每个词在fullText文本中出现的次数，并传入到字典中
        freqDict[token] = fullText.count(token)
    # 根据每个词出现的次数从高到底对字典进行排序
    # sortedFreq返回的是键值对列表
    sortedFreq = sorted(freqDict.items(), key=operator.itemgetter(1),
reverse=True)
    return sortedFreq[:5]
```

```
"""
```

函数说明：对贝叶斯垃圾邮件分类器进行自动化处理，与spam的区别就是文件来源，以及去除出现最高的30个词，其他一样

Parameters:

`feed1` - RSS来源1

`feed0` - RSS来源0

Returns:

对测试集中的每封邮件进行分类，若邮件分类错误，则错误数加 1，最后返回总的错误百分比

```
"""
```

```
def localWords(feed1, feed0):
    docList = []
    classList = []
    fullText = []
    # 返回两者的最小长度
    minLen = min(len(feed1['entries']), len(feed0['entries']))
    for i in range(minLen):
        wordList = textParse(feed1['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(1)

        wordList = textParse(feed0['entries'][i]['summary'])
        docList.append(wordList)
        fullText.extend(wordList)
        classList.append(0)

    vocabList = createVocabList(docList)
```

```

top30words = calcMostFreq(vocabList, fullText)
# 去掉出现次数最高的那些词
for pariW in top30words:
    if pariW[0] in vocabList:
        vocabList.remove(pariW[0])

trainingSet = list(range(2 * minLen))
testSet = []
# 随机取20个作为测试, 剩下的作为训练
for i in range(8):
    randIndex = int(random.uniform(0, len(trainingSet)))
    testSet.append(trainingSet[randIndex])
    del (trainingSet[randIndex])

trainMat = []
trainClasses = []
for docIndex in trainingSet:
    trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
    trainClasses.append(classList[docIndex])
classifier = MultinomialNB()
classifier.fit(np.array(trainMat), np.array(trainClasses))
#p0V, p1V, pSpam = trainNB0(np.array(trainMat), np.array(trainClasses))
errorCount = 0
for docIndex in testSet:
    wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
    if classifier.predict(np.array(wordVector).reshape(1,-1)) !=
classList[docIndex]:
        errorCount += 1
print('错误率: %.2f%%' % (float(errorCount) / len(testSet) * 100))
errors = float(errorCount) / len(testSet)
return errors

if __name__ == '__main__':
    feed1 = feedparser.parse('http://www.nasa.gov/rss/dyn/image_of_the_day.rss')
    feed0 =
feedparser.parse('http://www.cppblog.com/kevinlynx/category/6337.html/rss')
localWords(feed1, feed0)

```

其实修改的核心部分就是:

```

for docIndex in trainingSet:
    trainMat.append(bagOfWords2VecMN(vocabList, docList[docIndex]))
    trainClasses.append(classList[docIndex])
classifier = MultinomialNB()
classifier.fit(np.array(trainMat), np.array(trainClasses))
#p0V, p1V, pSpam = trainNB0(np.array(trainMat), np.array(trainClasses))
errorCount = 0
for docIndex in testSet:
    wordVector = bagOfWords2VecMN(vocabList, docList[docIndex])
    if classifier.predict(np.array(wordVector).reshape(1,-1)) != classList[docIndex]:
        errorCount += 1
print('错误率: %.2f%%' % (float(errorCount) / len(testSet) * 100))

```

最后看下运行结果

```
In [4]: runfile('C:/Users/HeatonHsu/.spyder-py3/temp.py', wdir='C:/Users/HeatonHsu/.spyder-py3')  
错误率: 0.00%
```

```
In [5]:
```

没错误, 也不知道我改的对不对, 提高这么多.

朴素贝叶斯的内容就这么多, 下一节开始看看经典的logistic回归.