贝叶斯分析-多项式朴素贝叶斯分类

姓名：熊荣康

学号：SA21229005

摘要

对于文本分类

本文利用xxx方法分析了xxx数据，发现了xxx结论

> key words: 多项式朴素贝叶斯,文本分类

1. 数据和问题描述

(本节描述数据的来源,特点,以及研究的问题)

对于文本分类，通常有14中分类算法。8种传统算法：kl临近、决策树、多层感知器、朴素贝叶斯（包括伯努利贝叶斯、高斯贝叶斯和多项式贝叶斯）、逻辑回归和支持向量机；4种集成学习算法：随机森林、AdaBoost、lightGBM和xgBoost；2种深度学习算法：前馈神经网络和LSTM。

多项式朴素贝叶斯通常用于文本分类，其特征都是指待分类文本的单词出现次数或者频次。这里用 20 个网络新闻组语料库（20 Newsgroups corpus，约 20 000 篇新闻）的单词出现次数作为特征，使用多项式朴素贝叶斯对这些新闻组进行分类。

2. 模型与数据分析

2.1 贝叶斯分类

朴素贝叶斯模型是一组非常简单快速的分类算法，通常适用于维度非常高的数据集。因为运行速度快，而且可调参数少，因此非常适合为分类问题提供快速粗糙的基本方案。朴素贝叶斯分类器建立在贝叶斯分类方法的基础上，其数学基础是贝叶斯定理（Bayes’stheorem）——一个描述统计量条件概率关系的公式。在贝叶斯分类中，我们希望确定一个具有某些特征的样本属于某类标签的概率，通常记为$P(L|features)$。贝叶斯定理告诉我们，可以直接用下面的公式计算这个概率

$$\frac{P(L_1|features)}{P(L_1|features)} = \frac{P(features|L_1)P(L_1)}{P(features|L_2)P(L_2)} \tag{1}$$

现在需要一种模型，帮我们计算每个标签的 $P(features|Li)$。这种模型被称为生成模型，因为它可以训练出生成输入数据的假设随机过程（或称为概率分布）。为每种标签设置生成模型是贝叶斯分类器训练过程的主要部分。虽然这个训练步骤通常很难做，但是我们可以通过对模型进行随机分布的假设，来简化训练工作，之所以称为“朴素”或“朴素贝叶斯”，是因为如果对每种标签的生成模型进行非常简单的假设，就能找到每种类型生成模型的近似解，然后就可以使用贝叶斯分类。不同类型的朴素贝叶斯分类器是由对数据的不同假设决定的。首先导入需要用的程序库：

多项式朴素贝叶斯通常用于文本分类，其特征都是指待分类文本的单词出现次数或者频 次。这里用 20 个网络新闻组语料库（20 Newsgroups corpus，约 20 000 篇新闻）的单词出现次数作为特征，演示如何用多项式朴素贝叶斯对这些新闻组进行分类。

2.2 高斯朴素贝叶斯

分类器假设每个标签的数据都服从简单的高斯分布，假设有如图(1)所示的数据，代码见附录A.1

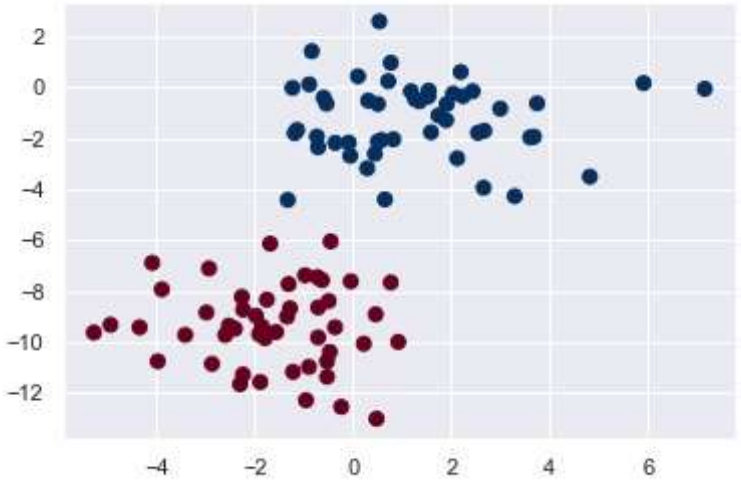

图1 高斯朴素贝叶斯分类数据

一种快速创建简易模型的方法是假设数据服从高斯分布，且变量无协方差（线性无关）。只要找出每个标签的所有样本点均值和标准差，再定义一个高斯分布，就可以拟合模型。
通过每种类型的生成模型，可以计算出任意数据点的似然估计$P(features|L_1)$，然后根据贝叶斯定理计算出后验概率比值，从而确定每个数据点可能性最大的标签。

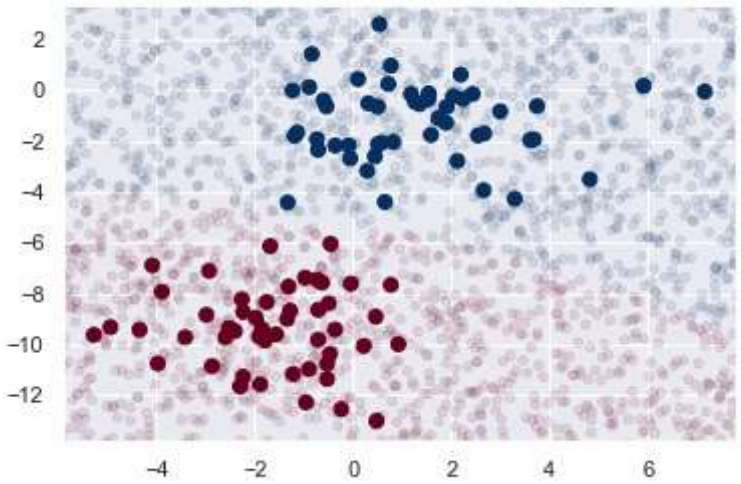

图2 高斯朴素贝叶斯分类可视化图

可以得到两个标签的后验概率，如果需要评估分类器的不确定性，那么这类贝叶斯方法非常有用。

$P(L_1\|features)$	$P(L_2\|features)$
0.	1.
0.	1.
0.	1.
1.	0.
0.	1.
0.	1.
0.08	0.92
0.	1.
0.	1.
1.	0.
1.	0.
1.	0.
0.89	0.11

由于分类的最终效果只能依赖于一开始的模型假设，因此高斯朴素贝叶斯经常得不到非常好的结果。但是，在许多场景中，尤其是特征较多的时候，这种假设并不妨碍高斯朴素贝叶斯成为一种有用的方法。

2.3 多项式朴素贝叶斯

多项式朴素贝叶斯（multinomial naive Bayes）它假设特征是由一个简单多项式分布生成的。多项分布可以描述各种类型样本出现次数的概率，因此多项式朴素贝叶斯非常适合用于描述出现次数或者出现次数比例的特征。这里模型数据的分布不再是高斯分布，而是用多项式分布代替。

多项式朴素贝叶斯通常用于文本分类，其特征都是指待分类文本的单词出现次数或者频次。这里用 20 个网络新闻组语料库（20 Newsgroups corpus，约 20 000 篇新闻）的单词出现次数作为特征，使用多项式朴素贝叶斯对这些新闻组进行分类。

下载数据代码在附录B.1。选择20类新闻，下载训练集和测试集（附录B.2），然后选择其中一篇新闻进行查看（结果见附录B.3）

介绍使用什么贝叶斯方法,为什么用该方法. 如果数据分析存在某些困难, 介绍你如何使用该方法解决了这些困难.

2.5 数据分析结果

为了让这些数据能用于机器学习，需要将每个字符串的内容转换成数值向量。可以创建一个管道，将$TF–IDF$向量化方法（详情参见附录D）与多项式朴素贝叶斯分类器组合在一起。代码见附录C.1

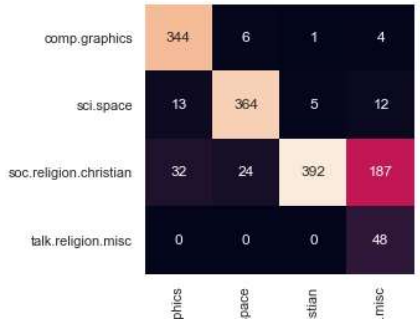

图3(A) 多项式朴素贝叶斯文本分类混淆矩阵,4类新闻

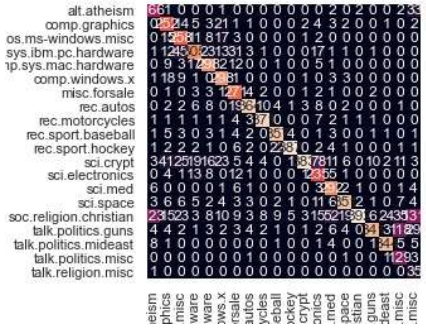

图3(B) 多项式朴素贝叶斯文本分类混淆矩阵,20类新闻

现在我们有一个可以对任何字符串进行分类的工具了，只要用管道的 `predict()` 方法就可以预测（代码见附录C.2）。下面的函数可以快速返回字符串的预测结果：

字符串	预测结果
'sending a payload to the ISS'	'sci.space'
'discussing islam vs atheism'	'soc.religion.christian`
'determining the screen resolution'	'comp.graphics`

介绍使用该方法分析所研究数据的结果, 不要直接粘贴程序输出, 重新组织分析结果,使用表格或图表说明.

3. 结论

由于朴素贝叶斯分类器对数据有严格的假设，因此它的训练效果通常比复杂模型的差。其优点主要体现在以下四个方面

- 训练和预测的速度非常快
- 直接使用概率预测
- 通常很容易解释
- 可调参数（如果有的话）非常少

这些优点使得朴素贝叶斯分类器通常很适合作为分类的初始解。如果分类效果满足要求，那么万事大吉，你获得了一个非常快速且容易解释的分类器。但如果分类效果不够好，那么你可以尝试更复杂的分类模型，与朴素贝叶斯分类器的分类效果进行对比，看看复杂模型的分类效果究竟如何。

朴素贝叶斯分类器非常适合用于以下应用场景。

- 假设分布函数与数据匹配（实际中很少见）
- 各种类型的区分度很高，模型复杂度不重要
- 非常高维度的数据，模型复杂度不重要

在新维度会增加样本数据信息量的假设条件下，高维数据的簇中心点比低维数据的簇中心点更分散。因此，随着数据维度不断增加，像朴素贝叶斯这样的简单分类器的分类效果会和复杂分类器一样，甚至更好——只要你有足够的数据，简单的模型也可以非常强大。

附录

A.1

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

from sklearn.datasets import make_blobs
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu');
plt.savefig('././images/fig1.jpg')
```

A.2

```
# 使用Scikit-Learn 的 sklearn.naive_bayes.GaussianNB 评估器
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y)

# 生成一些新数据来预测标签
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2)
ynew = model.predict(Xnew)

# 将这些新数据画出来，看看决策边界的位置
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='RdBu')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='RdBu', alpha=0.1)
plt.axis(lim);

# 用predict_proba方法计算样本属于某个标签的概率
yprob = model.predict_proba(Xnew)
yprob[-20:].round(2)
```

B.1

```
from sklearn.datasets import fetch_20newsgroups
data = fetch_20newsgroups()
data.target_names
```

输出结果:

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
```

```
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

B.2

```
# categories = ['talk.religion.misc', 'soc.religion.christian', 'sci.space',
# 'comp.graphics']
categories = data.target_names
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

B.3

```
From: guykuo@carson.u.washington.edu (Guy Kuo)
Subject: SI Clock Poll - Final Call
Summary: Final call for SI clock reports
Keywords: SI,acceleration,clock,upgrade
Article-I.D.: shelley.1qvfo9INNc3s
Organization: University of Washington
Lines: 11
NNTP-Posting-Host: carson.u.washington.edu

A fair number of brave souls who upgraded their SI clock oscillator have
shared their experiences for this poll. Please send a brief message
detailing
your experiences with the procedure. Top speed attained, CPU rated speed,
add on cards and adapters, heat sinks, hour of usage per day, floppy disk
functionality with 800 and 1.4 m floppies are especially requested.

I will be summarizing in the next two days, so please add to the network
knowledge base if you have done the clock upgrade and haven't answered this
poll. Thanks.

Guy Kuo <guykuo@u.washington.edu>
```

C.1

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
model = make_pipeline(TfidfVectorizer(), MultinomialNB())

# 通过这个管道，就可以将模型应用到训练数据上，预测出每个测试数据的标签
model.fit(train.data, train.target)
labels = model.predict(test.data)

```

```
# 这样就得到每个测试数据的预测标签，可以进一步评估评估器的性能了
from sklearn.metrics import confusion_matrix
mat = confusion_matrix(test.target, labels)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
xticklabels=train.target_names, yticklabels=train.target_names)
plt.xlabel('true label')
plt.ylabel('predicted label');
```

C.2

```
def predict_category(s, train=train, model=model):
    pred = model.predict([s])
    return train.target_names[pred[0]]
```

D.文本特征

分类特征

一种常见的非数值数据类型是分类数据，例如，浏览房屋数据的时候，除了看到“房价”（price）和“面积”（rooms）之类的数值特征，还会有“地点”（neighborhood）信息，数据可能像这样：

```
In[1]: data = [
{'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
{'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
{'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
{'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
]
```

可以把分类特征用映射关系编码成整数：

```
In[2]: {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3}
```

但是，在 Scikit-Learn 中这么做并不是一个好办法：这个程序包的所有模块都有一个基本假设，那就是数值特征可以反映代数量（algebraic quantities）。因此，这样映射编码可能会让人觉得存在`Queen Anne < Fremont < Wallingford`，甚至还有`Wallingford - Queen Anne = Fremont`，这显然是没有意义的。

面对这种情况，常用的解决方法是独热编码。它可以有效增加额外的列，让 0 和 1 出现在对应的列分别表示每个分类值有或无。当你的数据是像上面那样的字典列表时，用 ScikitLearn 的 DictVectorizer 类就可以实现：

```
In[3]: from sklearn.feature_extraction import DictVectorizer
vec = DictVectorizer(sparse=False, dtype=int)
vec.fit_transform(data)
Out[3]: array([[ 0, 1, 0, 850000, 4],
[ 1, 0, 0, 700000, 3],
[ 0, 0, 1, 650000, 3],
[ 1, 0, 0, 600000, 2]], dtype=int64)
```

`neighborhood` 字段转换成三列来表示三个地点标签，每一行中用 1 所在的列对应一个地点。当这些分类特征编码之后，你就可以和之前一样拟合 Scikit-Learn 模型了。如果要看每一列的含义，可以用下面的代码查看特征名称：

```
In[4]: vec.get_feature_names()
Out[4]: ['neighborhood=Fremont',
'neighborhood=Queen Anne',
'neighborhood=Wallingford',
'price',
'rooms']
```

这种方法也有一个显著的缺陷：如果你的分类特征有许多枚举值，那么数据集的维度就会急剧增加。然而，由于被编码的数据中有许多 0，因此用稀疏矩阵表示会非常高效：

```
In[5]: vec = DictVectorizer(sparse=True, dtype=int)
vec.fit_transform(data)
Out[5]: <4x5 sparse matrix of type '<class 'numpy.int64'>'
with 12 stored elements in Compressed Sparse Row format>
```

在拟合和评估模型时， Scikit-Learn 的许多（并非所有）评估器都支持稀疏矩阵输入。`sklearn.preprocessing.OneHotEncoder` 和 `sklearn.feature_extraction.FeatureHasher` 是 `Scikit-Learn` 另外两个为分类特征编码的工具。

文本特征

一种常见的特征工程需求是将文本转换成一组数值，绝大多数社交媒体数据的自动化采集，都是依靠将文本编码成数字的技术手段。数据采集最简单的编码方法之一就是单词统计：给你几个文本，让你统计每个词出现的次数，然后放到表格中。例如下面三个短语：

```
In[6]: sample = ['problem of evil',
'evil queen',
'horizon problem']
```

面对单词统计的数据向量化问题时，可以创建一个列来表示单词“problem”、单词“evil”和单词“horizon”等。虽然手动做也可以，但是用 Scikit-Learn 的 CountVectorizer 更是可以轻松实现：

```
In[7]: from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
X = vec.fit_transform(sample)
X
Out[7]: <3x5 sparse matrix of type '<class 'numpy.int64'>'
with 7 stored elements in Compressed Sparse Row format>
```

结果是一个稀疏矩阵，里面记录了每个短语中每个单词的出现次数。如果用带列标签的`DataFrame`来表示这个稀疏矩阵就更方便了：

```
In[8]: import pandas as pd
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
Out[8]: evil horizon of problem queen
0 1 0 1 1 0
1 1 0 0 0 1
2 0 1 0 1 0
```

不过这种统计方法也有一些问题：原始的单词统计会让一些常用词聚集太高的权重，在分类算法中这样并不合理。解决这个问题的方法就是通过`TF-IDF`（term frequency–inversedocument frequency，词频逆文档频率，IDF的大小与一个词的常见程度成反比），通过单词在文档中出现的频率来衡量其权重。计算这些特征的语法和之前的示例类似：

```
In[9]: from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
X = vec.fit_transform(sample)
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
Out[9]: evil horizon of problem queen
0 0.517856 0.000000 0.680919 0.517856 0.000000
1 0.605349 0.000000 0.000000 0.000000 0.795961
2 0.000000 0.795961 0.000000 0.605349 0.000000
```

参考文献