

基于朴素贝叶斯的文本分类模型

{ 姓名：孟川；学号：201814828；班级：2018 级学硕班；导师：陈竹敏 }

摘要：本次实验基于朴素贝叶斯构建了文本分类模型。在 20news-18828 数据集上，先在训练集获得参数（词频、文档频次）并保存，然后在测试集上做预测。最终测试集结果表明，在均使用拉普拉斯平滑的前提下，二项式模型取得了 **85.07%** 的精度，伯努利模型取得了 **78.55%** 的精度。二项式模型的精准度显著高于伯努利模型。

目录

一、训练：获取二项式模型和伯努利模型所需参数	2
1.1 文本预处理（同 KNN）	2
1.1.1 分词（同 KNN）	2
1.1.2 去停止词（同 KNN）	3
1.1.3 抽取词干（同 KNN）	3
1.2 划分训练集与测试集（同 KNN）	4
1.3 构建字典（同 KNN）	5
1.4 得到参数	6
1.4.1 二项式模型参数	6
1.4.2 伯努利模型参数	6
1.5 将参数保存	7
二、测试：使用二项式模型和伯努利模型进行类别预测	9
2.1 读取保存的参数	9
2.2 二项式模型的预测	10
2.2 伯努利模型的预测	10
三、实验	11
3.1 二项式模型	11
3.2 伯努利模型	12
四、结论	12

一、训练：获取二项式模型和伯努利模型所需参数

1.1 文本预处理（同 KNN）

文本预处理会主要使用 NLTK 工具，所以要先进行相应的工具包安装。

```
nltk.download('punkt')
nltk.download('stopwords')
```

图 1 调用命令

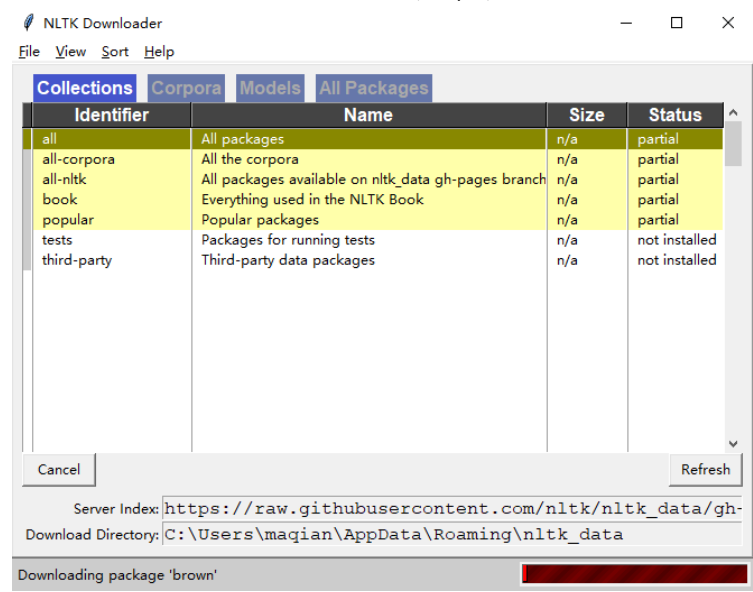


图 2 NLTK 安装

1.1.1 分词（同 KNN）

分词工作主要有以下三个步骤：

- 1. **小写化**：首先用 lower()方法将所有的字母小写化；
- 2. **去符号和数字**：用 string.punctuation 与 string.digits 方法得到所有符号和数字，并用 translate 方法去除，需要注意一点，要将符号或者数字去除的位置加上空格，以防符号连接的单词粘在一起，形成怪异的长单词；
- 3. **用 NLTK 实现分词**：使用 NLTK 工具包的 nltk.word_tokenize 方法实现分词。

```
# 进行分词-----
def get_tokens(text):
    lower = text.lower()
    remove_punctuation_map = {}
    total_string=string.punctuation+string.digits
    space=' '
    remove_punctuation_map = str.maketrans({key:space for key in total_string})
    lower.translate(remove_punctuation_map)
    no_punctuation = lower.translate(remove_punctuation_map)
    tokens = nltk.word_tokenize(no_punctuation)
    return tokens
```

图 3 分词

1.1.2 去停止词（同 KNN）

引入 NLTK 工具包的 nltk.corpus 的 stopwords, 具体方法为 stopwords.words('english'), 对分词结果属于停止词的进行过滤。

```
#过滤停止词-----
def filter_stopwords(tokens):
    more=['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z']
    stoplist=stopwords.words('english')
    stoplist.extend(more)
    filtered_stemmed=[word for word in tokens if not word in stoplist]
    return filtered_stemmed
```

图 4 去停止词

1.1.3 抽取词干（同 KNN）

抽取词干使用 NLTK 工具包中的 nltk.stem.SnowballStemmer('english')方法完成。

```
#抽取词干-----
def get_stem_tokens(tokens):
    stemmed=[]
    s = nltk.stem.SnowballStemmer('english')
    for item in tokens:
        stemmed.append(s.stem(item))
    return stemmed
```

图 5 抽取词干

```
#调用分词、过滤停止词、抽取词干
def use(text):

    tokens=get_tokens(text)
    filter_stopwords_tokens=filter_stopwords(tokens)
    stemmed_tokens=get_stem_tokens(filter_stopwords_tokens)

    return stemmed_tokens
```

图 6 对分词、去停止词与抽取词干进行调用

1.2 划分训练集与测试集（同 KNN）

测试集占整个数据集的 20%，因此从 20 个类的原始文档中分别抽取 20% 作为测试集，剩余 80% 作为训练集，并且每个类抽取之前先用 **shuffle 算法** 进行打乱，保证抽取的随机性。

最终切分的结果是，训练集文档个数为 15056，测试集文档个数为 3772。

```
def split_dataset(rate=0.2):
    trainlist=[]
    testlist=[]
    new_dir='Preprocessed data'
    cate_list=listdir(new_dir)
    for cate in cate_list:
        doc_list=listdir(new_dir+'/'+cate)
        random.shuffle(doc_list)
        j=len(doc_list)*rate
        for i in range(len(doc_list)):

            if i>=0 and i<j:
                testlist.append(cate+'_'+ doc_list[i])

            else:
                trainlist.append(cate+'_'+ doc_list[i])

    datew1=open('index_train or test set'+ '/'+'trainset.txt','w')
    datew2=open('index_train or test set'+ '/'+'testset.txt','w')
    for item in trainlist:
        datew1.write('%s\n' % item)
    for item in testlist:
        datew2.write('%s\n' % item)
    datew1.close()
    datew2.close()

    return len(trainlist)
```

图 7 划分训练集与测试集

1.3 构建字典（同 KNN）

本次实验构建字典的长度为 15749，构建字典的时候应该注意以下几点：

1. 用 dict 来存储字典中的单词；
2. 只用训练集来构建字典，测试集不参与词典的构建；
3. 为了减小字典的长度，减轻计算的负担，先要计算所有词在全部文档出现的总频数，如果该单词的总频数小于 10，则将其剔除，原因是该单词若总频数比较小，说明该单词属于生僻单词，对于文本分类意义不大。

```
#构建字典-----
def build_dict(select='trainset.txt'):
    word_dict={}
    new_word_dict={}
    index_r=open('index_train or test set'+ '/' +select, 'r')
    for item in index_r.readlines():
        new_item=item.strip('\n')
        cate,doc=new_item.split('_')
        load_dir='Preprocessed data'+ '/' +cate+ '/' +doc
        dict_d=open(load_dir, 'r')

        for word in dict_d.readlines():
            new_word=word.strip('\n')
            word_dict[new_word]=word_dict.setdefault(new_word,0)+1
            print(cate+' '+doc+' '+new_word)

        dict_d.close()

    for k,v in word_dict.items():
        if v>=10:
            new_word_dict[k]=v

    index_r.close()

    dict_w=open('dict.txt', 'w')

    for word in new_word_dict:
        dict_w.write('%s\n' %word)
    dict_w.close()

    return new word dict,
```

图 8 构建字典

1.4 得到参数

由于本次实验是要对比二项式模型和伯努利模型的异同, 并且两个模型需要的参数是不一样的, 所以需要计算两套不同的参数。总体的计算方法非常简单, 只需要遍历一遍训练集的所有特征词, 然后同时统计各种所需信息。

```
172
173 # 读取训练集-----
174 index_r=open('index_train or test set'+ '/'+'trainset.txt','r')
175 for item in index_r.readlines():
176     new_item=item.strip('\n')
177     cate,doc=new_item.split('_')
178     total_doc_num=total_doc_num+1
179     cate_doc_num[cate]=cate_doc_num.get(cate,0)+1
180     load_dir='Preprocessed data'+ '/'+'cate'+ '/'+'doc
181     train_read=open(load_dir,'r')
182     cate_per_word_doc_num_more_dict={}
183     for word in train_read.readlines():
184         new_word=word.strip('\n')
185         if new_word not in new_word dict:
186             print(new_word,"舍掉, 不属于字典")
187             continue
188         else:
189             print(new_word,"属于字典, 留下计数")
190             total_word_num=total_word_num+1
191             cate_word_num_dict[cate]=cate_word_num_dict.get(cate,0)+1
192             cate_name=cate+'_'+new_word
193             cate_per_word_num_dict[cate_name]=cate_per_word_num_dict.get(cate_name,0)+1
194             cate_per_word_doc_num_more_dict[cate_name]=cate_per_word_doc_num_more_dict.get(cate_name,0)+1
195     for i in cate_per_word_doc_num_more_dict:
196         cate_per_word_doc_num_dict[i]=1+cate_per_word_doc_num_more_dict.get(i,0)
197     train_read.close()
198     index_r.close()
199
```

图 9 参数获取的核心代码

1.4.1 二项式模型参数

多项式模型需要三个信息, 分别是: 总词频、每个类下的词频、在每一类下某个特征词的词频。

```
# 多项式模型的统计信息-----
total_word_num=0 #总词数
cate_word_num_dict={} #每个类下的总词数
cate_per_word_num_dict={} #在一个类下, 某个词出现的次数
```

图 10 多项式模型参数

1.4.2 伯努利模型参数

伯努利模型需要三个信息, 分别是: 总文档数、每个类下的文档数量、每一类下包含某个特征词的文档数。

```
# 伯努利模型的统计信息-----
total_doc_num=0 #文档的总个数
cate_doc_num={} #每个类的文档个数
cate_per_word_doc_num_dict={} #在一个类下, 某个词出现的文档数
```

图 11 伯努利模型参数

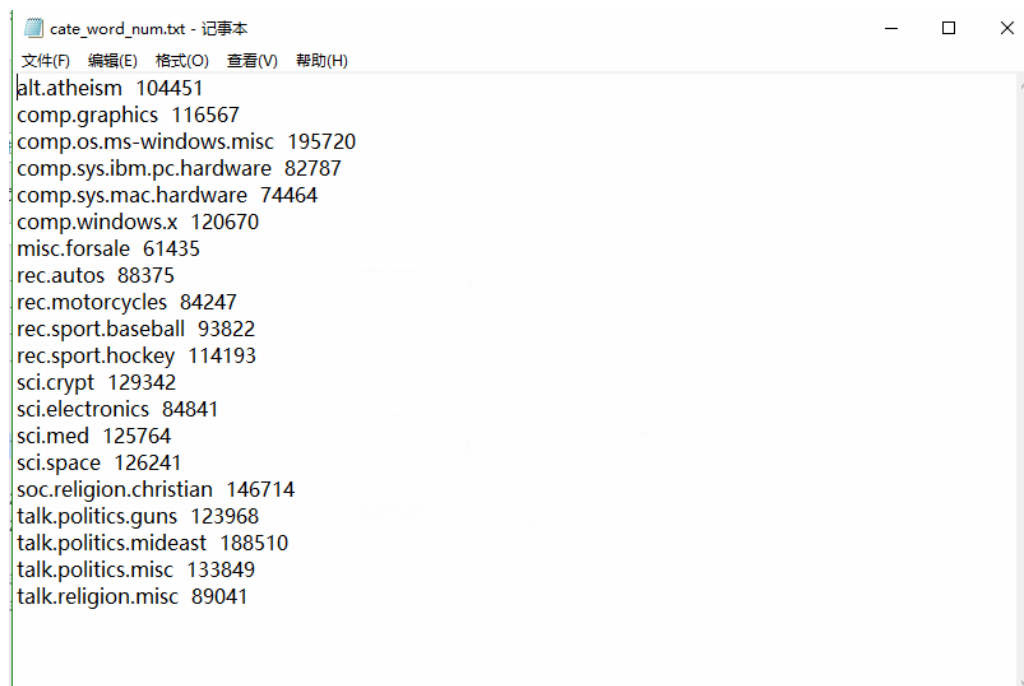
1.5 将参数保存

为了后期朴素贝叶斯方便计算，加快程序运行的速度，我们计算完针对模型的两套参数后，将参数进行保存，后期可直接调用。保存的信息分别是：**总词频、每个类下的词频、在每一类下某个特征词的词频、总文档数、每个类下的文档数量、每一类下包含某个特征词的文档数、字典和字典长度**。以下举出了几个保存的参数实例。



```
cate_doc_num.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
alt.atheism 639
comp.graphics 778
comp.os.ms-windows.misc 788
comp.sys.ibm.pc.hardware 785
comp.sys.mac.hardware 768
comp.windows.x 784
misc.forsale 777
rec.autos 792
rec.motorcycles 795
rec.sport.baseball 795
rec.sport.hockey 799
sci.crypt 792
sci.electronics 784
sci.med 792
sci.space 789
soc.religion.christian 797
talk.politics.guns 728
talk.politics.mideast 752
talk.politics.misc 620
talk.religion.misc 502
```

图 12 每个类下的文档数量



```
cate_word_num.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
alt.atheism 104451
comp.graphics 116567
comp.os.ms-windows.misc 195720
comp.sys.ibm.pc.hardware 82787
comp.sys.mac.hardware 74464
comp.windows.x 120670
misc.forsale 61435
rec.autos 88375
rec.motorcycles 84247
rec.sport.baseball 93822
rec.sport.hockey 114193
sci.crypt 129342
sci.electronics 84841
sci.med 125764
sci.space 126241
soc.religion.christian 146714
talk.politics.guns 123968
talk.politics.mideast 188510
talk.politics.misc 133849
talk.religion.misc 89041
```

图 13 每个类下的词频

per_word_cate_num.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
alt.atheism_frank 120
alt.atheism_uucp 61
alt.atheism_dwyer 63
alt.atheism_subject 756
alt.atheism_genocid 54
alt.atheism_caus 179
alt.atheism_theism 120
alt.atheism_evid 269
alt.atheism_benedikt 91
alt.atheism_articl 580
alt.atheism_qv 2
alt.atheism_fb 2
alt.atheism_horus 33
alt.atheism_ap 34
alt.atheism_mchp 33
alt.atheism_sni 33
alt.atheism_de 104
alt.atheism_write 798
alt.atheism_inform 87
alt.atheism_invari 6
alt.atheism_child 19
alt.atheism_son 18
alt.atheism_daughter 8
```

图 14 每一类下某个特征词的词频

per_word_cate_doc_num.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
alt.atheism_frank 42
alt.atheism_uucp 47
alt.atheism_dwyer 36
alt.atheism_subject 639
alt.atheism_genocid 40
alt.atheism_caus 111
alt.atheism_theism 24
alt.atheism_evid 92
alt.atheism_benedikt 52
alt.atheism_articl 402
alt.atheism_qv 2
alt.atheism_fb 2
alt.atheism_horus 30
alt.atheism_ap 31
alt.atheism_mchp 30
alt.atheism_sni 30
alt.atheism_de 68
alt.atheism_write 514
alt.atheism_inform 47
alt.atheism_invari 4
alt.atheism_child 14
alt.atheism_son 18
alt.atheism_daughter 6
```

图 15 每一类下包含某个特征词的文档数

二、测试：使用二项式模型和伯努利模型进行类别预测

2.1 读取保存的参数

第一步首先把训练阶段保存的参数加载到内存里面。

```
14 def data_load():
15     # 读取训练集的参数-----
16     dict_num=0
17     total_word_num=0 #总词数
18     cate_word_num_dict={} #每个类下的总词数
19     cate_per_word_num_dict={} #在一个类下，某个词出现的次数
20
21     # 伯努利模型的统计信息-----
22     total_doc_num=0 #文档的总个数
23     cate_doc_num={} #每个类的文档个数
24     cate_per_word_doc_num_dict={} #在一个类下，某个词出现的文档数
25
26     open_dict=open('dict.txt','r')
27     new_word_dict=[] #标准字典
28     for dict_word in open_dict.readlines():
29         new_dict_word=dict_word.strip('\n')
30         new_word_dict.append(new_dict_word)
31     open_dict.close()
32
33
34     inf_dict=open('Trained parameters'+ '/'+'dict.txt','r')
35     dict_num=inf_dict.read()
36     inf_dict.close()
37
38     inf_total_word=open('Trained parameters'+ '/'+'total_word_num.txt','r')
39     total_word_num=inf_total_word.read()
40     inf_total_word.close()
41
42     inf_cate_word=open('Trained parameters'+ '/'+'cate_word_num.txt','r')
43     for item in inf_cate_word.readlines():
44         new_item=item.strip('\n')
45         i,j=new_item.split(' ')
46         cate_word_num_dict[i]=j
47     inf_cate_word.close()
48
49     per_word_cate=open('Trained parameters'+ '/'+'per_word_cate_num.txt','r')
50     for item in per_word_cate.readlines():
51         new_item=item.strip('\n')
52         i,j=new_item.split(' ')
53         cate_per_word_num_dict[i]=j
54     per_word_cate.close()
55
56     inf_total_doc=open('Trained parameters'+ '/'+'total_doc_num.txt','r')
57     total_doc_num=inf_total_doc.read()
58     inf_total_doc.close()
59
60
61     inf_cate_doc=open('Trained parameters'+ '/'+'cate_doc_num.txt','r')
62     for item in inf_cate_doc.readlines():
63         new_item=item.strip('\n')
64         i,j=new_item.split(' ')
65         cate_doc_num[i]=j
```

图 16 参数读取

2.2 二项式模型的预测

二项式模型的预测核心代码如下，注意使用拉普拉斯平滑的时候，每个类条件概率的分母加上词表的长度，分子加 1。

```
115 #使用多项式朴素贝叶斯进行运算-----
116 def binomial_Naive_Bayes_Classifier(test_item,dict_num,total_word_num,cate_word_num_dict,cate_per_word_num_dict,cate_doc_num):
117     result_score={}
118     for i in cate_doc_num:
119         result_score[i]=0
120
121     for cate in result_score:
122         result_score[cate]=binomial_term(cate,test_item,dict_num,total_word_num,cate_word_num_dict,cate_per_word_num_dict)
123
124     sortedresult = sorted(result_score.items(), key = lambda x: x[1], reverse=True)
125
126     return sortedresult[0][0]
127
128 def binomial_term(cate,test_item,dict_num,total_word_num,cate_word_num_dict,cate_per_word_num_dict):
129     MLE=0.0
130     for word in test_item:
131         MLE=MLE+binomial_detail(cate,word,cate_per_word_num_dict,dict_num,cate_word_num_dict)
132     prior=float(cate_word_num_dict[cate])/float(total_word_num)
133     bayes=MLE+math.log(prior)
134
135     return bayes
136
137 def binomial_detail(cate,word,cate_per_word_num_dict,dict_num,cate_word_num_dict):
138     dis=cate+'_'+str(word)
139     numerator=1+float(cate_per_word_num_dict.get(dis,0))
140     Denominator=float(dict_num)+float(cate_word_num_dict[cate])
141     term=math.log(numerator/Denominator)
142     return term
```

图 17 二项式模型的预测核心代码

2.2 伯努利模型的预测

伯努利模型的预测核心代码如下，注意使用拉普拉斯平滑的时候，每个类条件概率的分母加 2，分子加 1。

```
144 #实现伯努利模型朴素贝叶斯-----
145 def Bernoulli_Naive_Bayes_Classifier(test_item,total_doc_num ,cate_doc_num,cate_per_word_doc_num_dict,new_word_dict):
146     new_test_item=[] #会重
147     for i in Counter(test_item):
148         new_test_item.append(i)
149     result_score={}
150     for i in cate_doc_num:
151         result_score[i]=0
152
153     for cate in result_score:
154         result_score[cate]=Bernoulli_term(cate,test_item,total_doc_num ,cate_doc_num,cate_per_word_doc_num_dict,new_word_dict)
155
156     sortedresult = sorted(result_score.items(), key = lambda x: x[1], reverse=True)
157
158     return sortedresult[0][0]
159
160 def Bernoulli_term(cate,new_test_item,total_doc_num ,cate_doc_num,cate_per_word_doc_num_dict,new_word_dict):
161     MLE=0.0
162     for word in new_test_item:
163         MLE=MLE+Bernoulli_detail(new_test_item,cate,word,cate_doc_num,cate_per_word_doc_num_dict)
164     prior=float(cate_doc_num[cate])/float(total_doc_num)
165     bayes=MLE+math.log(prior)
166
167     return bayes
168
169 def Bernoulli_detail(new_test_item,cate,word,cate_doc_num,cate_per_word_doc_num_dict):
170     dis=cate+'_'+str(word)
171     numerator=1+float(cate_per_word_doc_num_dict.get(dis,0))
172     Denominator=float(2)+float(cate_doc_num[cate])
173     if word in new_test_item:
174         term=math.log(numerator/Denominator)
175     else:
176         term=math.log(1-(numerator/Denominator))
177
178     return term
179
```

图 18 伯努利模型的预测核心代码

三、实验

以下是二项式模型和伯努利模型的实验结果比较（见图 19）。详细实验细节请见 3.1 和 3.2 部分。

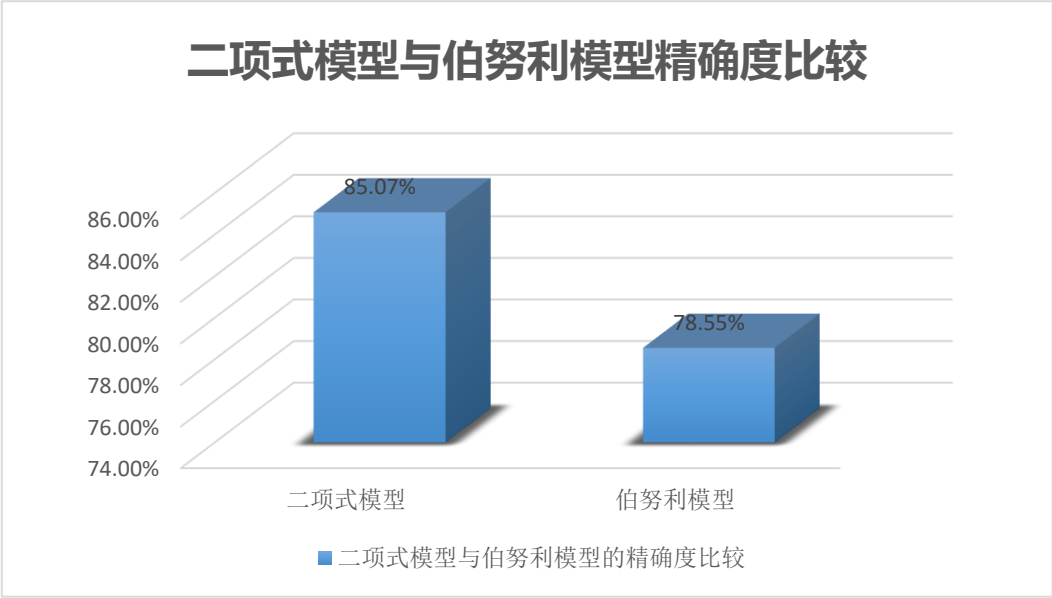


图 19 二项式模型与伯努利模型精确度比较

3.1 二项式模型

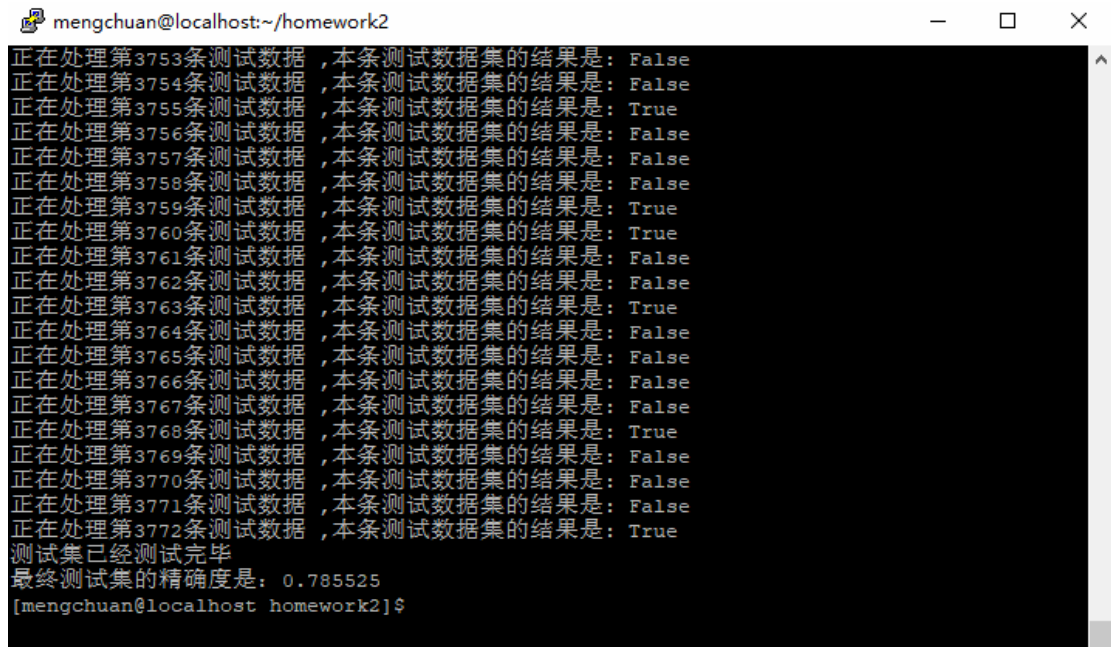
在测试集上运行二项式，最终得到的**精确度为 0.8507**，**分类正确率为 85.07%**。测试集上的结果见图 18。

```
mengchuan@localhost:~/homework2
正在处理第3750条测试数据 ,本条测试数据集的结果是: True
正在处理第3751条测试数据 ,本条测试数据集的结果是: True
正在处理第3752条测试数据 ,本条测试数据集的结果是: False
正在处理第3753条测试数据 ,本条测试数据集的结果是: False
正在处理第3754条测试数据 ,本条测试数据集的结果是: False
正在处理第3755条测试数据 ,本条测试数据集的结果是: True
正在处理第3756条测试数据 ,本条测试数据集的结果是: True
正在处理第3757条测试数据 ,本条测试数据集的结果是: False
正在处理第3758条测试数据 ,本条测试数据集的结果是: False
正在处理第3759条测试数据 ,本条测试数据集的结果是: True
正在处理第3760条测试数据 ,本条测试数据集的结果是: True
正在处理第3761条测试数据 ,本条测试数据集的结果是: False
正在处理第3762条测试数据 ,本条测试数据集的结果是: False
正在处理第3763条测试数据 ,本条测试数据集的结果是: True
正在处理第3764条测试数据 ,本条测试数据集的结果是: False
正在处理第3765条测试数据 ,本条测试数据集的结果是: True
正在处理第3766条测试数据 ,本条测试数据集的结果是: True
正在处理第3767条测试数据 ,本条测试数据集的结果是: True
正在处理第3768条测试数据 ,本条测试数据集的结果是: True
正在处理第3769条测试数据 ,本条测试数据集的结果是: False
正在处理第3770条测试数据 ,本条测试数据集的结果是: False
正在处理第3771条测试数据 ,本条测试数据集的结果是: True
正在处理第3772条测试数据 ,本条测试数据集的结果是: True
测试集已经测试完毕
最终测试集的精确度是: 0.850742
[mengchuan@localhost homework2]$
```

图 20 二项式模型运行结果

3.2 伯努利模型

在测试集上运行伯努利模型，最终得到的**精确度为 0.7855**，**分类正确率为 78.55%**。测试集上的结果见图 18。



```
mengchuan@localhost:~/homework2
正在处理第3753条测试数据 ,本条测试数据集的结果是: False
正在处理第3754条测试数据 ,本条测试数据集的结果是: False
正在处理第3755条测试数据 ,本条测试数据集的结果是: True
正在处理第3756条测试数据 ,本条测试数据集的结果是: False
正在处理第3757条测试数据 ,本条测试数据集的结果是: False
正在处理第3758条测试数据 ,本条测试数据集的结果是: False
正在处理第3759条测试数据 ,本条测试数据集的结果是: True
正在处理第3760条测试数据 ,本条测试数据集的结果是: True
正在处理第3761条测试数据 ,本条测试数据集的结果是: False
正在处理第3762条测试数据 ,本条测试数据集的结果是: False
正在处理第3763条测试数据 ,本条测试数据集的结果是: True
正在处理第3764条测试数据 ,本条测试数据集的结果是: False
正在处理第3765条测试数据 ,本条测试数据集的结果是: False
正在处理第3766条测试数据 ,本条测试数据集的结果是: False
正在处理第3767条测试数据 ,本条测试数据集的结果是: False
正在处理第3768条测试数据 ,本条测试数据集的结果是: True
正在处理第3769条测试数据 ,本条测试数据集的结果是: False
正在处理第3770条测试数据 ,本条测试数据集的结果是: False
正在处理第3771条测试数据 ,本条测试数据集的结果是: False
正在处理第3772条测试数据 ,本条测试数据集的结果是: True
测试集已经测试完毕
最终测试集的精确度是: 0.785525
[mengchuan@localhost homework2]$
```

图 21 伯努利模型运行结果

四、结论

经过本次实验，良好的锻炼了我的代码能力，我有以下心得：

1. K 朴素贝叶斯的运算速度远远快于 KNN，并且代码比 KNN 更加简洁。
2. KNN 的精确对略高于朴素贝叶斯。在该数据集上，KNN 可以实现 87%+ 的精确度，但是在朴素贝叶斯的二项式模型上，只能实现 85%+ 的精确度。
3. **整体而言，朴素贝叶斯的“性价比”要高于 KNN。**