
Data mining

作业 1: 基于 KNN 与 VSM 的文本分类模型

作业 2: 基于朴素贝叶斯的文本分类模型

作业 3: 基于 sklearn 的多种聚类算法的实现与评测

姓名: 孟川

学号: 201814828

班级: 2018 级学硕班

导师: 陈竹敏

作业 1：基于 KNN 与 VSM 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 KNN 与 VSM 构建了文本分类模型。在 20news-18828 数据集上，经过 VSM 构建、KNN 构建、N-fold 验证集验证 K 值和测试集测试等实验步骤，最终发现，在 **K 值取 15** 的时候模型表现效果最好，在 **5-fold 验证集**上取得 **87.91%的精确度**，在**测试集**取得 **87.96%**的精确度。

目录

一、vector space model 构建.....	3
1.1 文本预处理	3
1.1.1 分词.....	3
1.1.2 去停止词	4
1.1.3 抽取词干	4
1.2 划分训练集与测试集.....	5
1.3 构建字典.....	6
1.4 计算 TF-IDF 值.....	7
1.4.1 训练集 IDF 值计算	7
1.4.2 训练集 TF 值计算	8
1.4.3 测试集的 IDF 值计算.....	8
1.4.4 测试集的 TF 值计算.....	9
1.5 将 embedding 转化为矩阵	9
二、KNN 的构建	11
2.1 读取训练集和测试集的 embedding 矩阵.....	11
2.2 计算 cosin similarity	11
2.3 排序	12
三、实验	13
3.1 N-fold 验证集实验	13
3.2 测试集实验结果	14
四、结论	15

一、vector space model 构建

1.1 文本预处理

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

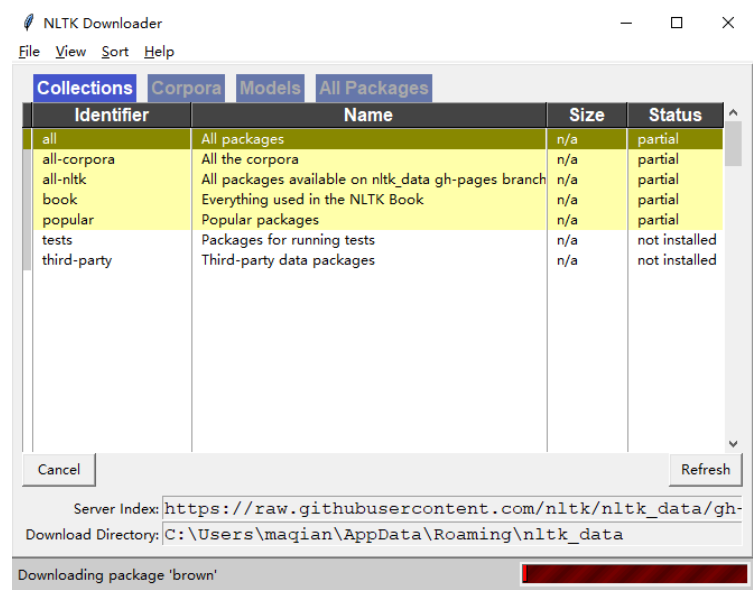


图 2 NLTK 安装

1.1.1 分词

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

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

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

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

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 计算 TF-IDF 值

1.4.1 训练集 IDF 值计算

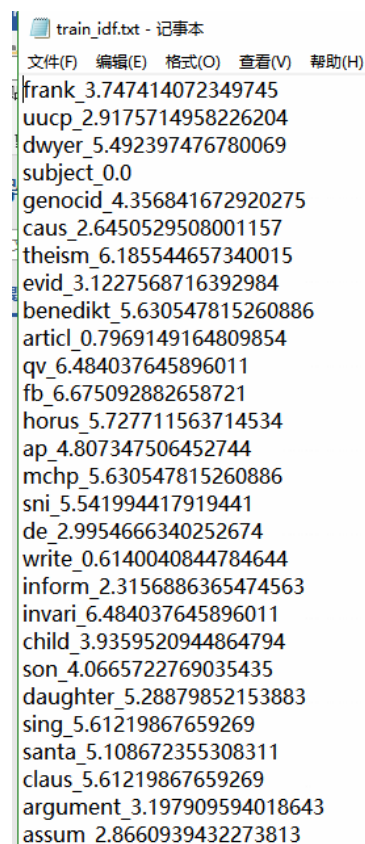
字典中的每个单词针对所有文档的 IDF 值是一致的，但是 TF 值每个文档都不一致，所以 IDF 值只需要算一次，然后存储调用，这样更方便。

```
#计算训练集数据的idf值-----  
def train_idf(word,train_total_list):  
  
    contain_doc_num=sum(1 for doc in train_total_list if word in doc)  
    idf_sore=math.log(len(train_total_list)/contain_doc_num)  
  
    return contain_doc_num,idf_sore
```

图 9 训练集 IDF 计算方法体

```
#先计算训练集的idf值，然后保存起来-----  
  
train_idf_dict={}  
train_word_df={}  
  
for word in new_word_dict:  
    train_word_df[word],train_idf_dict[word]=train_idf(word,train_total_list)
```

图 10 训练集 IDF 计算方法调用



```
train_idf.txt - 记事本  
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)  
frank_3.747414072349745  
uucp_2.9175714958226204  
dwyer_5.492397476780069  
subject_0.0  
genocid_4.356841672920275  
caus_2.6450529508001157  
theism_6.185544657340015  
evid_3.1227568716392984  
benedikt_5.630547815260886  
articl_0.7969149164809854  
qv_6.484037645896011  
fb_6.675092882658721  
horus_5.727711563714534  
ap_4.807347506452744  
mchp_5.630547815260886  
sni_5.541994417919441  
de_2.9954666340252674  
write_0.6140040844784644  
inform_2.3156886365474563  
invari_6.484037645896011  
child_3.9359520944864794  
son_4.0665722769035435  
daughter_5.28879852153883  
sing_5.61219867659269  
santa_5.108672355308311  
claus_5.61219867659269  
argument_3.197909594018643  
assum_2.8660939432273813
```

图 11 计算得到的每个单词的 IDF 值示例

1.4.2 训练集 TF 值计算

在训练集中，字典中的每个单词对于不同文档，TF 值是不同的，所以在遍历词表的基础上，再需要遍历每个文档去计算 TF 值。

算得 TF 之后，再度读取已经计算好的 IDF 值，然后计算得训练集每个文档的 TF-IDF 值，将所有文档的 embedding 存到 LIST 里面保存。

```
# 计算训练集的tf-idf值-----
def train_tf_idf(word,doc,idf_dict):
    if word not in doc:
        tf_sore=0
    else:
        tf_sore=math.log(doc[word])+1
    idf_sore=idf_dict[word]
    tf_idf=tf_sore*idf_sore

    return tf_idf
```

图 12 计算训练集 TF-IDF 方法体

```
# 计算训练集的tf-idf值-----
train_tf_idf_doc=[]
for doc in train_total_list:
    tf_idf_word=[]
    print("读取训练集的文档：")
    print(doc)
    print('\n')

    for word in new_word_dict:
        tf_idf_word.append(train_tf_idf(word,doc,train_idf_dict)) #得到每个词表的值
    train_tf_idf_doc.append(tf_idf_word)#把每个文档的embedding合成一个总的list
```

图 13 计算训练集 TF-IDF 方法调用

1.4.3 测试集的 IDF 值计算

测试集的 IDF 值计算不同于训练集的 IDF 计算，因为测试集每个样例之间是不可见的，每个测试集只能“看得见”所有训练集的文档，所以对于字典中的每个词在每条测试集中 IDF 的计算，每读入 1 条测试集，就要把这条测试集并入原始的训练集中，然后把它们视为一个文档总体，然后再计算 IDF。整个计算过程比较消耗时间。

```
# 计算测试数据的idf值-----
def test_idf(word,train_total_list,train_word_df,test_doc):

    N=len(train_total_list)+1
    if word in test_doc:
        idf_sore=math.log(N/(train_word_df[word]+1))
    else:
        idf_sore=math.log(N/(train_word_df[word]))

    return idf_sore
```

图 14 测试集 IDF 计算方法体

```
# 计算测试集的idf值-----
test_idf_dict={}
for i,test_doc in enumerate(test_total_list):
    test_idf_doc_dict={}
    for word in new_word_dict:
        test_idf_doc_dict[word]=test_idf(word,train_total_list,train_word_df,test_doc)
    test_idf_dict[i]=test_idf_doc_dict
```

图 15 测试集 IDF 计算方法调用

[illegible]

图 19 测试集 embedding

二、KNN 的构建

2.1 读取训练集和测试集的 embedding 矩阵

将 VSM 得到的训练集和测试集的 embedding 读取出来，然后再针对测试集矩阵的每一行（每个样例），去与训练集所有的样例去计算相似度，返回该条测试样例最有可能的类别，然后和真实的 label 相比，看是否对应。

```
def data_load():
    # 读取已经训练好的训练集与测试集的embedding-----
    train=np.loadtxt("embedding/trainset embedding.txt")
    #test=np.loadtxt("embedding/testset embedding.txt")

    train_label_list=[] #训练集的label list
    train_label_value={}
    train_label=open('index_train or test set'+ '/'+'trainset.txt','r')
    for i,item in enumerate(train_label.readlines()):
        new_item=item.strip('\n')
        cate,doc=new_item.split('_')
        train_label_list.append(cate)
        train_label_value[new_item]=train[i]
    train_label.close()

    test_label_list=[] #测试集的label list
    test_label=open('index_train or test set'+ '/'+'testset.txt','r')
    for item in test_label.readlines():
        new_item=item.strip('\n')
        cate,doc=new_item.split('_')
        test_label_list.append(cate)
    test_label.close()
```

图 20 embedding 读取

2.2 计算 cosin similarity

针对每一条 test 样例和 train 样例，分别计算二者的 cosin similarity，并返回结果。

```
def cosin_similarity(test_v, train_v):

    testVect = np.mat(test_v)
    trainVect = np.mat(train_v)
    num = float(testVect * trainVect.T)
    denom = np.linalg.norm(testVect) * np.linalg.norm(trainVect)
    return float(num)/(1.0+float(denom))
```

图 21 cosin similarity 计算

2.3 排序

针对每一条测试样例, 得到与所有的训练样例的 cosin 相似度的值后, 按从大到小排序, 然后取前 K 个, 看哪个类贡献的样例数量最多 (或者看哪个类的累计 cosin 相似度值最大), 然后返回这个类, 这个类就是 KNN 对本测试样例的类别预测结果。

```
#使用KNN进行运算-----
def KNN(train_label_value, test_item, k_num):
    simMap={}
    for cate_doc,value in train_label_value.items():
        similarity = cosin_similarity(test_item,value)

        simMap[cate_doc] = similarity

    sortedSimMap = sorted(simMap.items(), key = lambda x: x[1], reverse=True)

    cateSimMap = {}
    for i in range(k_num):
        cate = sortedSimMap[i][0].split('_')[0]
        cateSimMap[cate] = cateSimMap.get(cate,0) + sortedSimMap[i][1]

    sortedCateSimMap = sorted(cateSimMap.items(),key=lambda x: x[1],reverse=True)

    return sortedCateSimMap[0][0]
```

图 22 排序

三、实验

3.1 N-fold 验证集实验

先通过 N-fold 来确定 KNN 的最佳 K 值, 这里我们采用 **5-fold** 验证法, 验证结果如下:

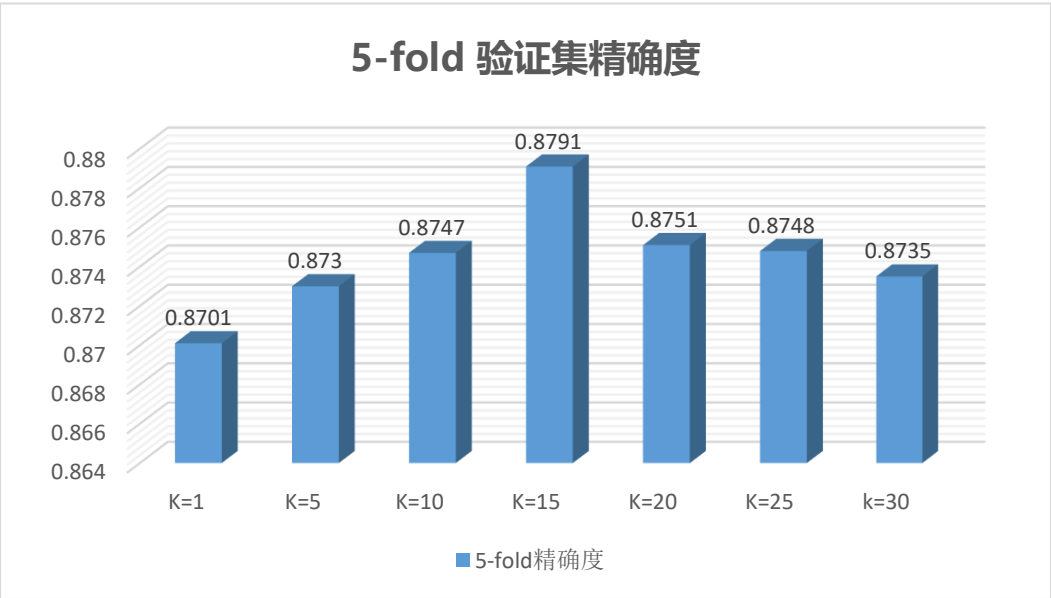


图 23 验证集结果

由上图得知, **K=15** 的时候, 在验证集上采用 **5-fold** 实现了 **0.8791** 的精确度, 因此我们决定 KNN 在测试集上的 K 值为 15。

```
mengchuan@localhost:~/homework1
第2992个验证集的分类结果是: True
正在处理第5-fold的第2993个验证数据集,当前验证集的总数为3002
第2993个验证集的分类结果是: True
正在处理第5-fold的第2994个验证数据集,当前验证集的总数为3002
第2994个验证集的分类结果是: False
正在处理第5-fold的第2995个验证数据集,当前验证集的总数为3002
第2995个验证集的分类结果是: True
正在处理第5-fold的第2996个验证数据集,当前验证集的总数为3002
第2996个验证集的分类结果是: True
正在处理第5-fold的第2997个验证数据集,当前验证集的总数为3002
第2997个验证集的分类结果是: True
正在处理第5-fold的第2998个验证数据集,当前验证集的总数为3002
第2998个验证集的分类结果是: True
正在处理第5-fold的第2999个验证数据集,当前验证集的总数为3002
第2999个验证集的分类结果是: True
正在处理第5-fold的第3000个验证数据集,当前验证集的总数为3002
第3000个验证集的分类结果是: True
正在处理第5-fold的第3001个验证数据集,当前验证集的总数为3002
第3001个验证集的分类结果是: True
正在处理第5-fold的第3002个验证数据集,当前验证集的总数为3002
第3002个验证集的分类结果是: False
第1-fold上的验证集精确度为:0.880053
第2-fold上的验证集精确度为:0.871803
第3-fold上的验证集精确度为:0.888188
第4-fold上的验证集精确度为:0.873796
第5-fold上的验证集精确度为:0.881746
在5-fold验证集上的平均精确度为: 0.8791172881473692
[mengchuan@localhost homework1]$
```

图 24 验证集实现 **0.8791** 的精确度

```

#进行N-fold交叉验证-----
def N_fold_validation(fold_num,train_label_value):
    N_fold_rate=float(1/fold_num)
    N_fold=[]
    cate_total=[]
    cate={}
    #先看看训练集有哪些类
    for cate_doc in train_label_value:
        cate[cate_doc.split('.')[0]]=None
    #把不同类的训练集放到不同的list来存储
    for cate_name in cate:
        cate_list=[]
        for cate_doc in train_label_value:
            if cate_doc.split('.')[0]==cate_name:
                cate_list.append(cate_doc)
        cate_total.append(cate_list)
    #进行5-fold训练
    for i in range(fold_num):
        train_section={}
        test_section={}
        for cate_list in cate_total:
            j=len(cate_list)*N_fold_rate
            for k in range(len(cate_list)):
                if k>=i*j and k<(i+1)*j:
                    test_section[cate_list[k]]=train_label_value[cate_list[k]]
                else:
                    train_section[cate_list[k]]=train_label_value[cate_list[k]]

        count=0
        num=0
        for label,test_item in test_section.items():
            num=num+1
            print("正在处理第%d-fold的第%d个验证数据集,当前验证集的总数为%d"%(i+1,num,len(test_section)))

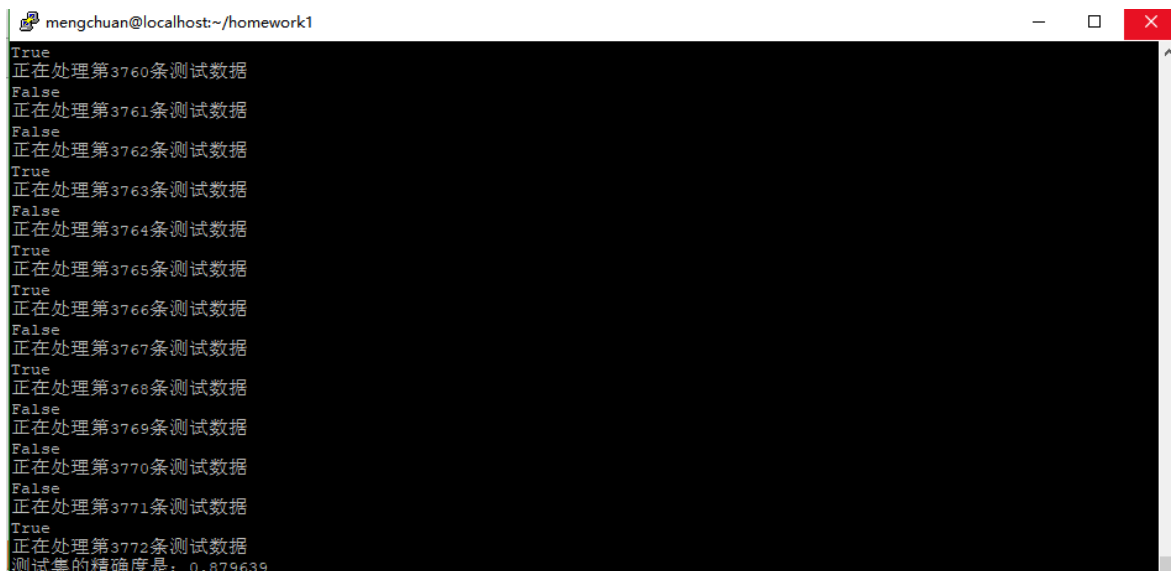
            if label.split('.')[0]==KNN(train_section,test_item ,15):
                count=count+1
            print("第%d个验证集的分类结果是: "% num,label.split('.')[0]==KNN(train_section,test_item ,15))
        accuracy=float(count)/float(len(test_section))
        N_fold.append(accuracy)
    return N_fold

```

图 25 N-fold 验证代码实现

3.2 测试集实验结果

最终我们以 K=15 的 KNN 在测试集上运行，最终得到的精确度为 0.8796，分类正确率为 87.96%。测试集上的结果见图 18。



```

mengchuan@localhost:~/homework1
True
正在处理第3760条测试数据
False
正在处理第3761条测试数据
False
正在处理第3762条测试数据
True
正在处理第3763条测试数据
False
正在处理第3764条测试数据
True
正在处理第3765条测试数据
True
正在处理第3766条测试数据
False
正在处理第3767条测试数据
True
正在处理第3768条测试数据
False
正在处理第3769条测试数据
False
正在处理第3770条测试数据
False
正在处理第3771条测试数据
True
正在处理第3772条测试数据
测试集的精确度是: 0.879639

```

图 26 测试集运行结果

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. KNN 是一项比较有效的方法，实现简单，可以实现较高的精确度，但是实验起来吃内存、耗时间，本次实验使用服务器运行，但是运行一次仍旧需要非常长的时间。
2. 代码的优化非常关键，尽量写代码之前要进行充分的构思，尽量减少循环的使用，读取数据的时候尽量将大文件化成多个小文件，减轻内存的消耗。
3. 要加强代码的封装与规整性，加强方法的调用，减少代码的冗余度，增加代码的可读性。

作业 2：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	17
1.1 文本预处理（同 KNN）	17
1.1.1 分词（同 KNN）	17
1.1.2 去停止词（同 KNN）	18
1.1.3 抽取词干（同 KNN）	18
1.2 划分训练集与测试集（同 KNN）	19
1.3 构建字典（同 KNN）	20
1.4 得到参数	21
1.4.1 二项式模型参数	21
1.4.2 伯努利模型参数	21
1.5 将参数保存	22
二、测试：使用二项式模型和伯努利模型进行类别预测	24
2.1 读取保存的参数	24
2.2 二项式模型的预测	25
2.2 伯努利模型的预测	25
三、实验	26
3.1 二项式模型	26
3.2 伯努利模型	27
四、结论	27

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

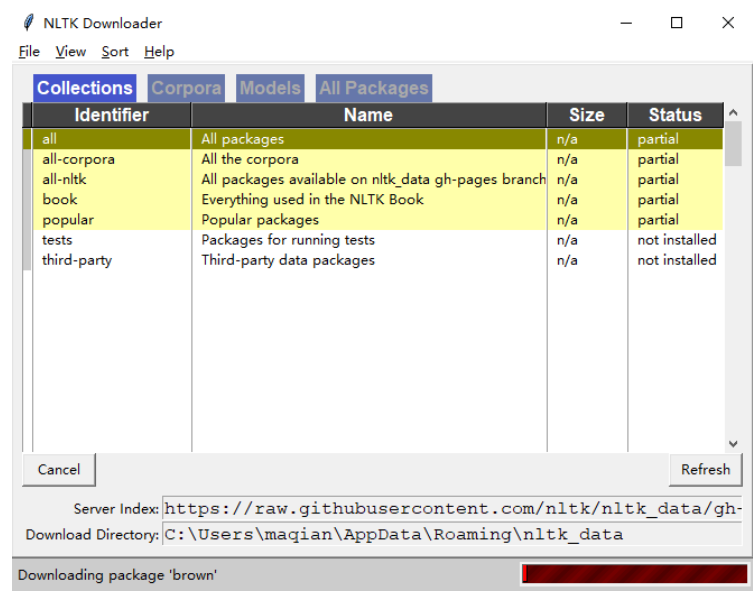


图 2 NLTK 安装

1.1.1 分词（同 KNN）

分词工作主要有以下三个步骤：

- 4. **小写化**：首先用 lower()方法将所有的字母小写化；
- 5. **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；
- 6. **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：

4. 用 dict 来存储字典中的单词；
5. 只用训练集来构建字典，测试集不参与词典的构建；
6. 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

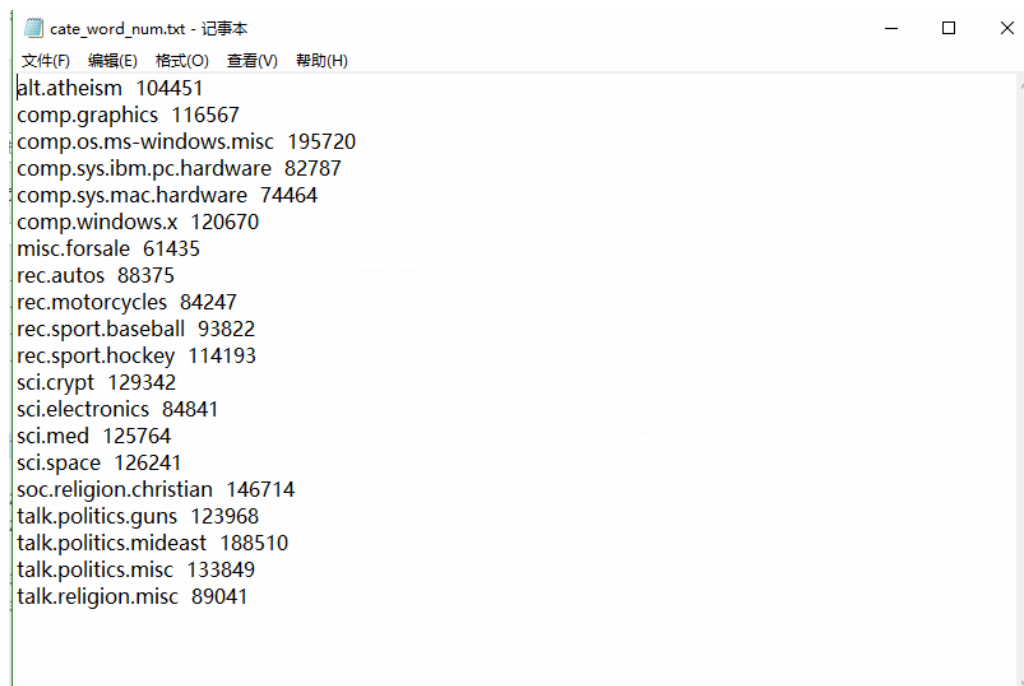
1.5 将参数保存

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

三、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

四、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。

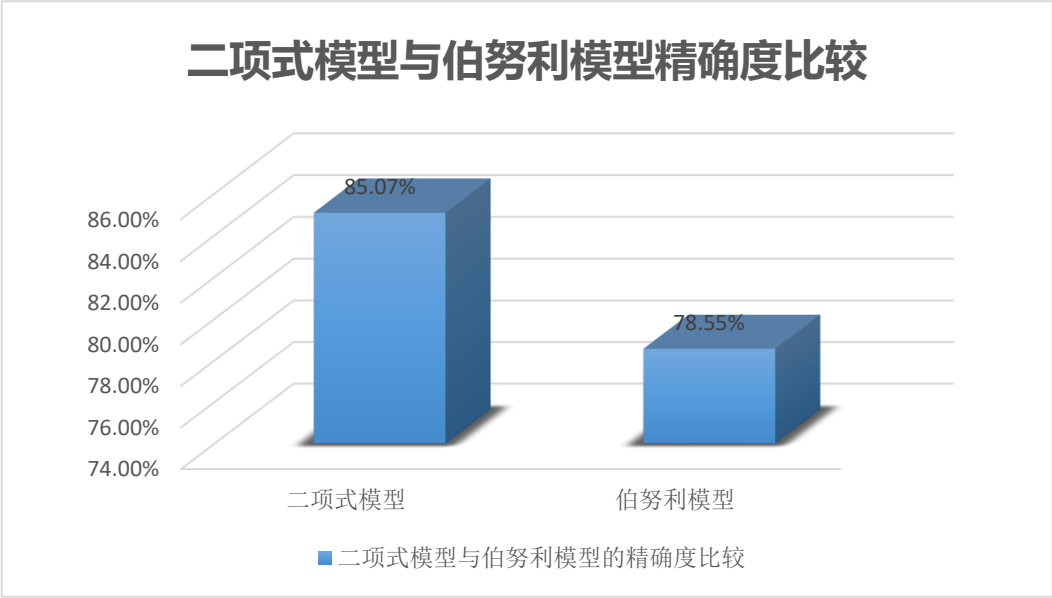


图 19 二项式模型与伯努利模型精确度比较

3.1 二项式模型

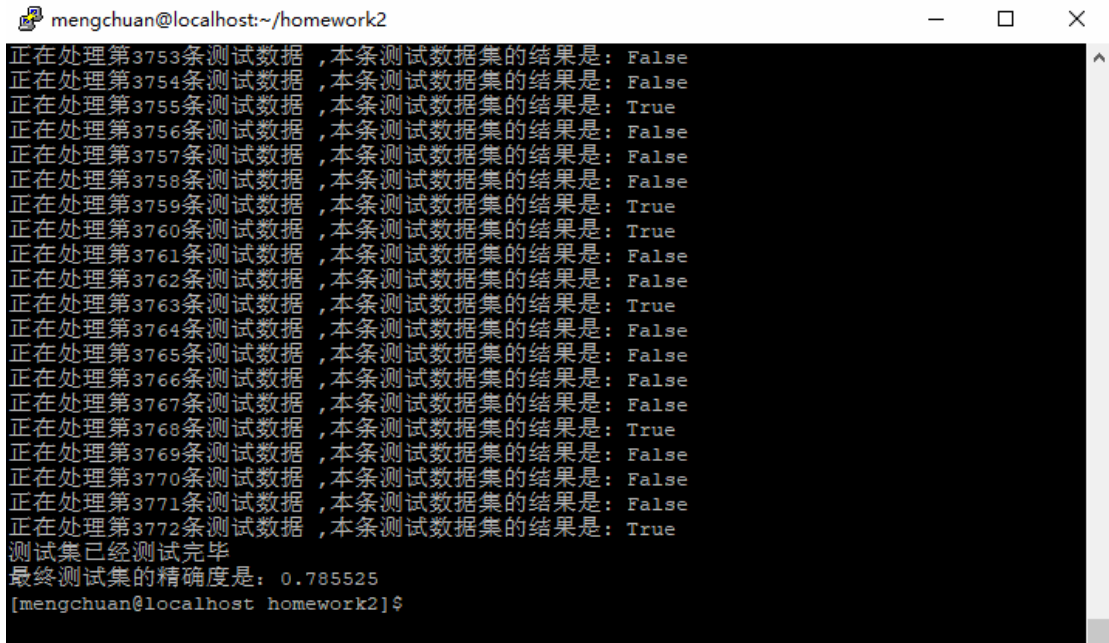
在测试集上运行二项式，最终得到的**精确度为 0.8507**，**分类正确率为 85.07%**。测试集上的结果见图 18。

```
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

6.2 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：

4. 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。
5. 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。
6. **整体而言，朴素贝叶斯的“性价比”要高于 KNN。**

作业 3：基于 sklearn 的多种聚类算法的实现与评测

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

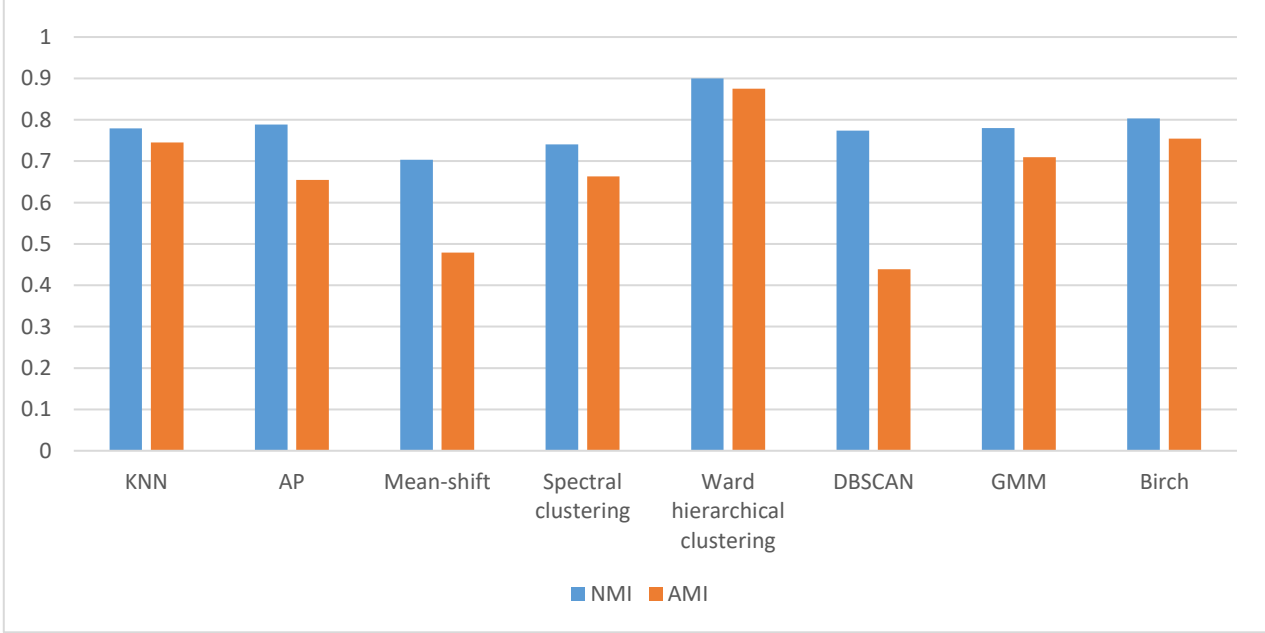
摘要：通过 sklearn 在 Tweets 数据集上应用多种聚类方法，我主要发现了以下结论：

- 1. Ward hierarchical clustering、Birch 与 Gaussian mixture models 三个模型非常适合用于文本内容的聚类，并且三个模型时间效率较高；
- 2. Spectral clustering 模型在时间效率上表现地最好；
- 3. 密度型聚类方法（例如 DBSCAN 与 Mean-shift）往往时间效率比较低，并且在 NMI 与 AMI 两个评测指标上差距较大（AMI 上表现分值更低）。

NMI 与 AMI 评测（所有模型均在最佳参数状态）

	NMI	AMI	运行时间(s)
KNN	0.779365	0.745205	58.606697
Affinity Propagation	0.788888	0.654406	36.910559
Mean-shift	0.703239	0.478830	69.337893
Spectral clustering	0.740351	0.663427	3.623575
Ward hierarchical clustering	0.899808	0.875384	22.521459
DBSCAN	0.773610	0.439140	77.088685
Gaussian mixture models	0.780075	0.709347	8.277973
Birch	0.803051	0.754462	17.408954

NMI与AMI评测（所有模型均在最佳参数状态）



NMI 与 AMI 评测（所有模型均在最佳参数状态）

目录

一、 Tweets 数据集预处理	30
1.1 数据集统计信息	30
1.1.1 基本信息统计	30
1.1.2 cluster label 的频数统计	30
1.2 得到 tf-idf 嵌入	31
1.2.1 调用 sklearn 封装的 tf-idf 方法	31
1.2.2 embedding 与 label 的保存	32
二、调用 sklearn 聚类方法与调参	33
2.1 KNN	33
2.2 Affinity Propagation(AP)	34
2.3 Mean-shift	35
2.4 Spectral clustering	36
2.5 Ward hierarchical clustering	37
2.5 DBSCAN	38
2.5 Gaussian mixtures	39
2.5 Birch	40
三、聚类结果评测	41
3.1 NMI 与 AMI	41
3.2 运行时间评测	42
四、结论	42

一、 Tweets 数据集预处理

预处理部分由两个部分构成：1.操作之前对数据集进行信息统计；2.得到数据集 document 的 tf-idf 表示，并保存。

1.1 数据集统计信息

数据集统计部分主要有两部分构成，分别是基本信息统计与 cluster label 的频率统计。

1.1.1 基本信息统计

以下是数据集的基本信息统计。

表 1 数据集基本信息统计

Document 数量	Cluster label 数量
2472	89

1.1.2 cluster label 的频数统计

我们发现该数据集 cluster label 的频率分布差距比较大，有些 cluster label 频数很大，有的 cluster label 频数很小。详细情况见图 1。

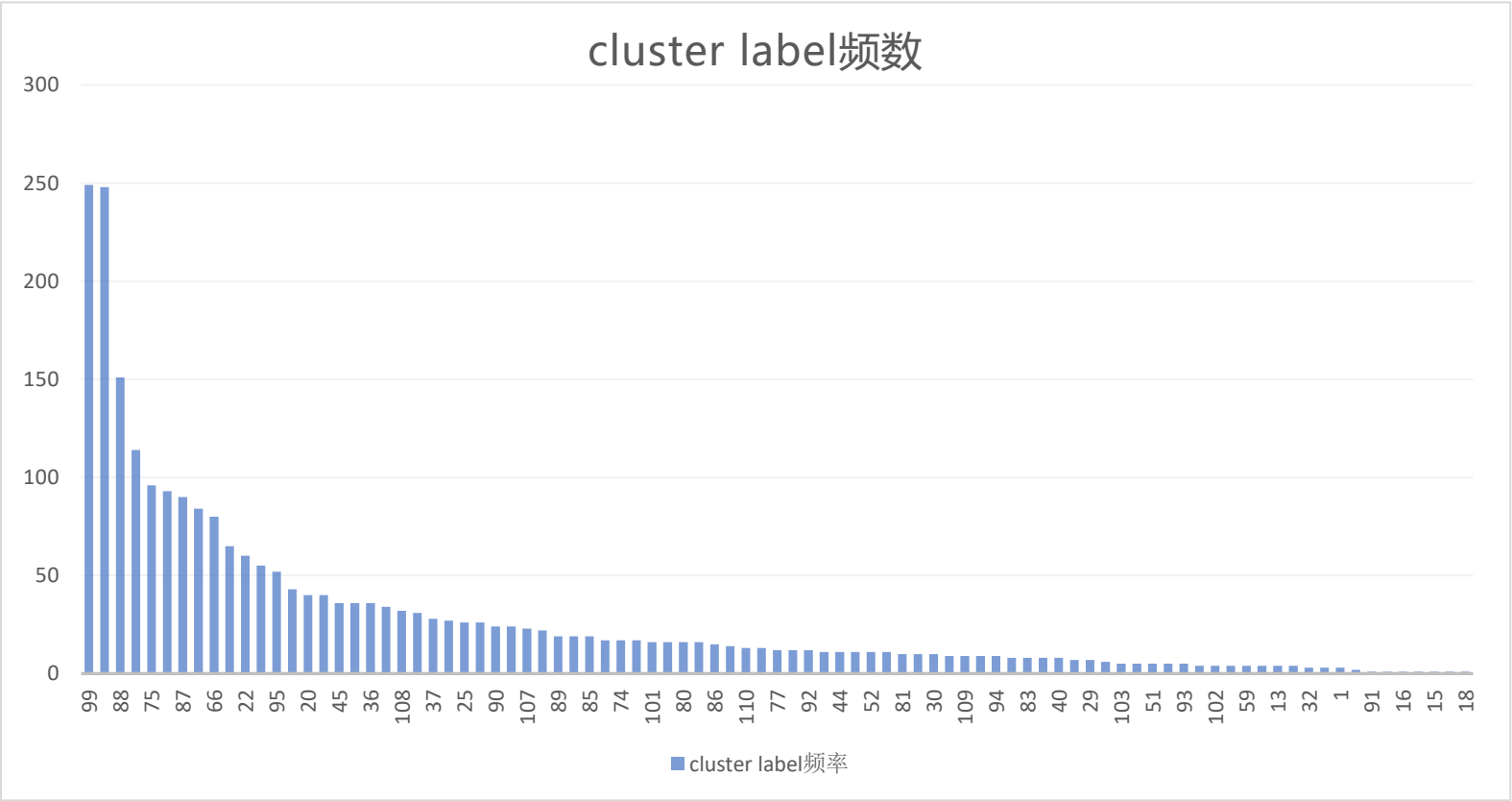


图 1 cluster label 频数统计(cluster label 按频数由大到小的顺序从左至右依次排开)

从图 1 可得，排名比较高的三个 cluster label 99、60、88 别对应的频数是 249、248、151，而排名后 19 个 cluster label 的频数均小于 5，排名后 7 个 cluster label 的频数均为 1。所以我们得知，虽然该数据集的 cluster label 总数为 89，但是很可能频数较小的 cluster label 在聚类中充当了噪音角色。

1.2 得到 tf-idf 嵌入

先得到文档的 tf-idf 嵌入与相应的 cluster label，然后将其保存。

1.2.1 调用 sklearn 封装的 tf-idf 方法

调用 sklearn 的 CountVectorizer、vectorizer.fit_transform、transformer.fit_transform 等方法，构建长度为 5097 的字典，最终得到所有 document 的 tf-idf 表示矩阵，矩阵的维度为 2472×5097 。

```
16
17 #下面使用sklearn调用tf-idf获得每个utterance的embedding
18 vectorizer = CountVectorizer()
19 count = vectorizer.fit_transform(corpus)
20 transformer = TfidfTransformer()
21 tfidf_matrix = transformer.fit_transform(count)
22 np.savetxt("embedding/tf-idf.txt", tfidf_matrix.toarray())
23
```

图 2 使用 sklearn 的 tf-idf 方法

```
'trick', 'trim', 'trip', 'trippy', 'triumph', 'trokar', 'trophy', 'tropical', 'trou', 'trouble', 'troubled',
'trout', 'trouw', 'true', 'truth', 'tsa', 'tube', 'tuesday', 'tuition', 'tuna', 'tune', 'tunesia', 'tunis',
'tunisia', 'tunisian', 'tunkwa', 'tunnel', 'turkish', 'turmoil', 'turn', 'turncoat', 'turquoise', 'tv',
'tweak', 'twee', 'tweet', 'tweeting', 'twelve', 'twenty', 'twilight', 'twist', 'twisted', 'twitition',
'twitter', 'twitterlandd', 'tx', 'tying', 'type', 'typo', 'tyranny', 'tyrant', 'ucla', 'ufo', 'ugc', 'uic',
'uk', 'ulera', 'ultima', 'ultimate', 'ultra', 'umbra', 'unacceptable', 'unanimously', 'unconfirmed',
'undergraduate', 'understand', 'underwood', 'undoubtedly', 'unemployment', 'unexpected', 'unfold',
'unhealthy', 'union', 'unionmajor', 'unique', 'unit', 'unite', 'united', 'university', 'unjust', 'unknown',
'unlocks', 'unreal', 'unrest', 'unsecured', 'unsettled', 'unveils', 'unwitting', 'uoit', 'upcycle', 'update',
'updated', 'upgrade', 'upgraded', 'upi', 'uploaded', 'uprising', 'ur', 'uranium', 'urban', 'urge', 'urged',
'urgently', 'usa', 'usda', 'user', 'usfda', 'usnews', 'usual', 'usually', 'usurps', 'utah', 'va', 'vacation',
'vacuum', 'vader', 'valuable', 'value', 'valve', 'van', 'vancouver', 'variety', 'varying', 'vast', 've',
'vega', 'vegetable', 'vegetarian', 'veneta', 'venezuela', 'venezuelan', 'venti', 'ventura', 'vera',
'veracity', 'verdict', 'verify', 'verizon', 'verry', 'version', 'vessel', 'vi', 'vic', 'vice', 'victim',
'victory', 'vide', 'video', 'videotwitter', 'view', 'vigilant', 'viibryd', 'vinaigrette', 'vincent',
'vinson', 'vintage', 'violates', 'violence', 'violent', 'vip', 'virginia', 'virtual', 'virulent', 'visa',
'visit', 'visited', 'vital', 'vitam', 'vitamin', 'voice', 'voiced', 'void', 'volkswagen', 'volkswagon',
'volt', 'volunteer', 'voov', 'vote', 'votechristina', 'voted', 'voter', 'vow', 'vowed', 'vowing', 'vw',
'wack', 'wacky', 'wade', 'wadlow', 'wager', 'wagering', 'wait', 'waitin', 'waiting', 'wake', 'wal', 'wale',
'wall', 'walleye', 'walmart', 'wannabe', 'wanting', 'war', 'ward', 'warm', 'warmer', 'warming', 'warn',
'warned', 'warner', 'warning', 'wash', 'washing', 'washington', 'wasn', 'waste', 'watch', 'watched',
'watching', 'water', 'wave', 'wavewalk', 'way', 'weapon', 'wear', 'weather', 'web', 'website', 'wedding',
'wedema', 'wednesday', 'wee', 'week', 'weekend', 'weekly', 'weig', 'weigh', 'weight', 'weightcrushers',
'welcome', 'well', 'welle', 'west', 'western', 'weymouth', 'wfnewshinged', 'wfnewsi', 'wft', 'whale',
'whatproblem', 'whip', 'white', 'whitneyport', 'whitm', 'whycollegenow', 'wi', 'widening', 'wife', 'wig',
'wikileaks', 'wilcox', 'wild', 'wildlife', 'wilkinson', 'will', 'william', 'wilmington', 'win', 'wind',
'wine', 'winfrey', 'wing', 'winner', 'winning', 'winnipeg', 'wino', 'wint', 'winter', 'wire', 'wired',
'wisconsin', 'wished', 'wisp', 'wit', 'witness', 'wix', 'wlcenral', 'wo', 'woe', 'woman', 'won', 'wonder',
'wonderful', 'wont', 'woo', 'wood', 'woohoooo', 'woow', 'word', 'work', 'worked', 'worker', 'working',
'workplace', 'workshop', 'world', 'worldchanging', 'worldnews', 'worldnomads', 'worldwide', 'worm', 'worst',
'worth', 'worthy', 'wouldn', 'wound', 'wounded', 'wowsa', 'wrap', 'wrapped', 'wrapup', 'wresting',
'wrestling', 'wright', 'write', 'writer', 'writes', 'writing', 'written', 'wrk', 'wrong', 'wrote', 'xd',
'xerox', 'xlii', 'xlv', 'xmnr', 'xoom', 'xperia', 'xy', 'yahoo', 'yall', 'yap', 'yarrow', 'yasi', 'yb',
'ybf', 'yea', 'yeah', 'yeahboy', 'year', 'yeates', 'yeg', 'yelchin', 'yell', 'yellowfin', 'yemen', 'yemeni',
'ven', 'vesterday', 'vikes', 'vnet', 'vo', 'vonkers', 'vork', 'voughal', 'vouna', 'vounder', 'vourselves',
```

图 3 字典中的词

1.2.2 embedding 与 label 的保存

将 document 的 embedding 矩阵和相应的 label 保存，方便后续使用。

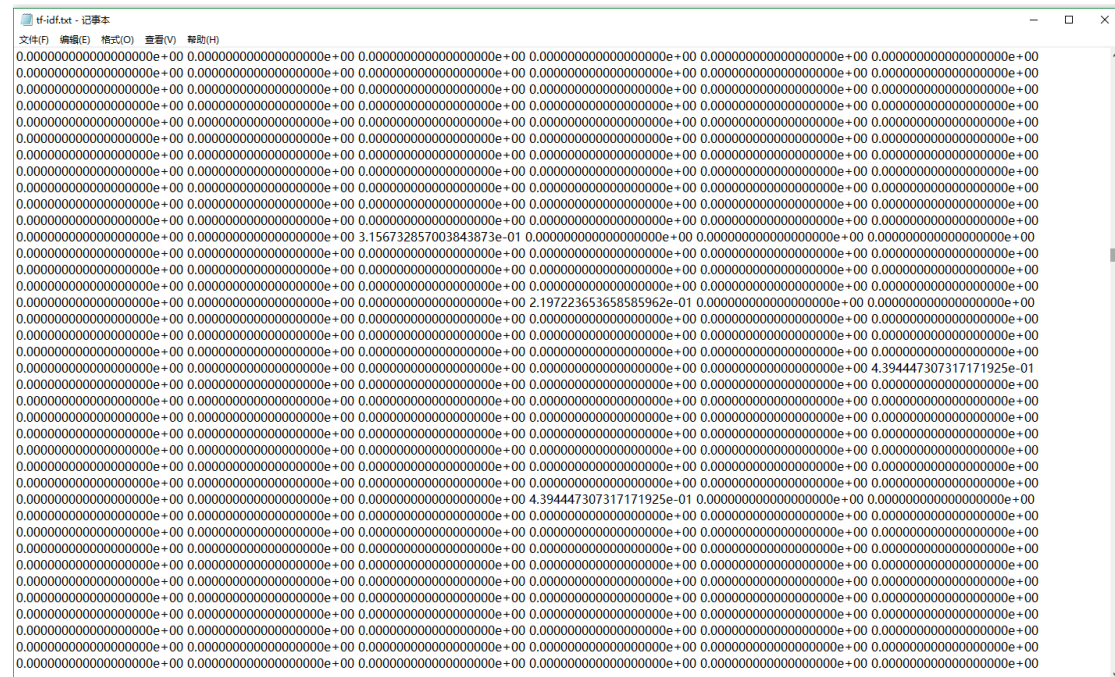


图 4 tf-idf 嵌入矩阵

二、调用 sklearn 聚类方法与调参

因为 AMI 比 NMI 在偶然情况处理更合理，相对更合理一些，**本文模型调参时均用 AMI 作为评测指标。**

2.1 KNN

KNN 的核心代码如下。

```
33 def k_means(data):  
34     t0 = time()  
35     estimator = KMeans(n_clusters=50)#构造聚类器  
36     estimator.fit(data)#聚类  
37     label_pred = estimator.labels_ #获取聚类标签  
38     time_use=time()-t0  
39     return label_pred,time_use  
40
```

图 5 KNN 核心代码

注意 Tweets 数据集的 label 分类个数是 89，所以我们也实现调整 KNN 的聚类个数为 89 (n_clusters=89)，但是发现，当 n_clusters=70 的时候，实验结果最好,AMI 取得最大值 **0.745205**，这可能就是因为 cluster label 排序 70 位之后的 document 数量太少，反而成了噪音干扰了其他正常的聚类。

表 2 KNN 调参表

n_clusters	AMI	运行时间(s)
89	0.679438	73.929518
80	0.709297	65.360532
70	0.745205	58.606697
60	0.727317	54.959479
50	0.650711	46.455092

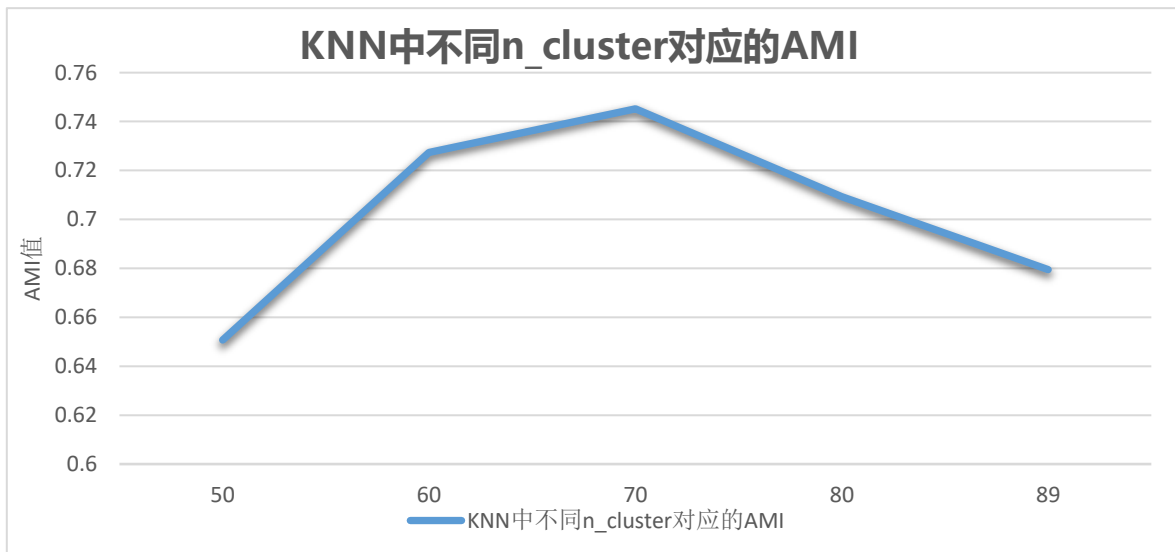


图 6 KNN 中不同 n_cluster 对应的 AMI

2.2 Affinity Propagation(AP)

Affinity Propagation(AP)的核心代码如下。

```
33
36 def Affinity_Propagation(data):
37     af = AffinityPropagation(preference=-5.74).fit(data)
38     cluster_centers_indices = af.cluster_centers_indices_
39     label_pred= af.labels_
40     n_clusters= len(cluster_centers_indices)
41     print(n_clusters)
42     return label_pred
43
```

图 7 Affinity Propagation 核心代码

实验发现在默认情况下，AP 聚类方法非常倾向于得到多个聚类结果（当“preference”=None 时，得到了 320 个类）。而 Tweets 数据集一共才 89 个类，必须调整“preference”来降低每个 point 是聚类中心点的可能性，从而降低聚类所得的类别个数，最终发现当“preference”=-5.74 时，model 的 AMI 指标最高，AMI 达到 **0.654407**，运行时间为 36.910559s。详情见表 3。

表 3 AP 模型的调参表（1）

Preference 值	聚类类别个数	AMI
0	2398	0.010602344528155293
-1	1603	0.1616498679626918
-2	320	0.4896257947296034
-3	218	0.5630028005415774
-4	142	0.6180687349737378
-5	115	0.6287995540653826
-5.5	104	0.6395386600529404
-5.6	117	0.6388484325316311
-5.7	98	0.64935267412474
-5.74	98	0.6544066133519121
-5.8	2472	0.000000

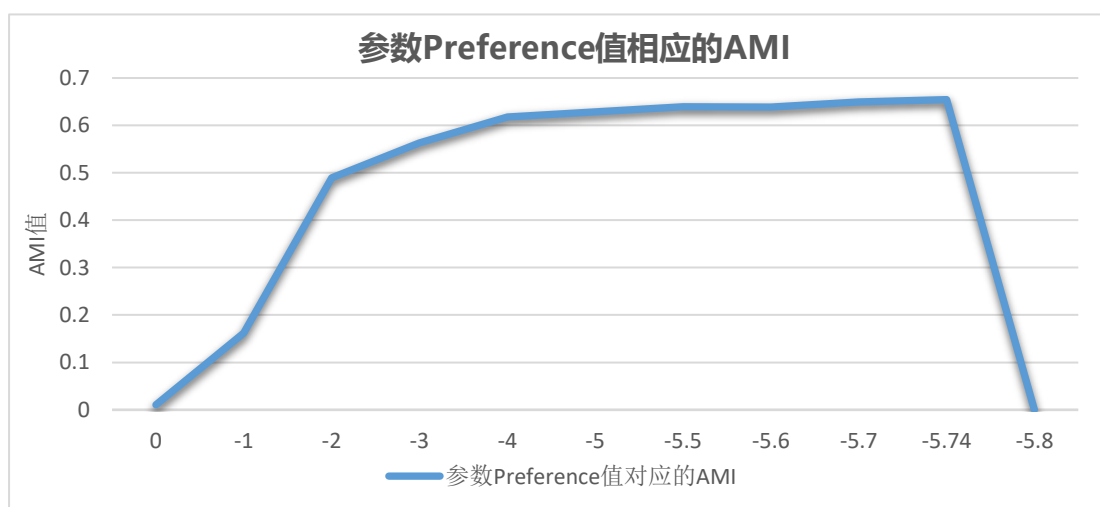


图 8 AP 模型的调参

我们在“preference”=-5.74 的情况下，改变 damping（阻尼系数）的值，观察实验结果发现并没有得到提升。

表 4 AP 模型的调参表（2）

damping 值	聚类类别个数	AMI
0.6	97	0.652058
0.7	97	0.648307
0.8	96	0.647118
0.9	97	0.649740

2.3 Mean-shift

Mean-shift 的预测核心代码如下。

```
51 def Mean_shift(data):
52     t0 = time()
53     bandwidth=0.9
54     ms = MeanShift(bandwidth=bandwidth,bin_seeding=True,cluster_all=True)
55     ms.fit(data)
56     label_pred = ms.labels_
57     labels_unique = np.unique(label_pred)
58     n_clusters = len(labels_unique)
59     time_use=time()-t0
60
61     print("bandwidth参数设置为: ",bandwidth)
62     print("得到的cluster数目为: ",n_clusters)
63     return label_pred,time_use
```

图 9 Mean-shift 核心代码

当 bin_seeding=False 的时候，调整 Bandwidth，得到 AMI 的值见表 5。发现参数“bandwidth”设置为 0.9 的时候，AMI 取得最大值。

表 5 Mean-shift 调参表（1）

bandwidth 值	聚类类别个数	AMI
1	1148	0.12148586730363102
0.9	1634	0.20377716278710623
0.8	1976	0.10020146997456376
0.7	2119	0.0635485205329769
0.6	2208	0.04422927942354482
0.5	2272	0.032440

当 bin_seeding=True 的时候，调整 Bandwidth，得到 AMI 的值见表 6。

表 6 Mean-shift 调参表（2）

bandwidth 值	聚类类别个数	AMI	运行时间(s)
1	299	0.014363	138.083952
0.9	478	0.478830	69.337893
0.8	605	0.377089	36.676130
0.7	696	0.347504	34.564874
0.6	871	0.312399	27.771711
0.5	1257	0.209903	32.459113

整体而言，当 bin_seeding 设置为 true 时，比设置为 False 时提升幅度较大，并且仍旧在“bandwidth”设置为 0.9 的时候 AMI 取得最大值，最大值为 **0.478830**，运行时间为 **69.337893s**。

2.4 Spectral clustering

以下为 Spectral clustering 的核心代码。

```
53
54 def Spectral_clustering(data):
55     sc=SpectralClustering(gamma=0.3,n_clusters=89).fit(data)
56     label_pred =sc.labels_
57
58     return label_pred
59
```

图 10 Spectral clustering 核心代码

当 n_clusters=89 的时候，gamma 值的改变引起 AMI 的改变，详见表 7。我们发现当 gamma 值取 0.5 的时候，AMI 取得最大值 **0.658281**，运行时间为 **4.583276s**。gamma 值取大于 1 的时候，结果普遍较差，在此不列出结果。

表 7 Spectral clustering 调参表（1）

gamma 值	AMI	运行时间(s)
1（默认）	0.555137	4.558172
0.9	0.552800	4.549476
0.8	0.610997	4.330443
0.7	0.638527	4.430617
0.6	0.630709	4.198558
0.5	0.658281	4.583276
0.4	0.633030	4.442534
0.3	0.649792	4.060393
0.2	0.643518	4.314387
0.1	0.646111	4.203846

当 n_clusters=70 的时候，gamma 值的改变引起 AMI 的改变，详见表 8。我们发现当 gamma 值为 0.1 的时候，AMI 取得最大值 **0.663427**，时间为 **3.623575s**，大于在 n_clusters=89 的最优值。

表 8 Spectral clustering 调参表（2）

gamma 值	AMI	运行时间(s)
0.8	0.584451	4.088376
0.7	0.595617	3.740295
0.6	0.600795	3.495676
0.5	0.606832	3.792112
0.4	0.612978	4.347775
0.3	0.641143	3.672531
0.2	0.657407	3.668014
0.1	0.663427	3.623575

2.5 Ward hierarchical clustering

以下为 Ward hierarchical clustering 的核心代码。

```
73 def Ward_hierarchical_clustering(data):
74     t0 = time()
75     whc = AgglomerativeClustering(affinity='euclidean', n_clusters=80, linkage='average').fit(data)
76     label_pred=whc.labels_
77     time_use=time()-t0
78
79     return label_pred,time_use
80
```

图 11 Ward hierarchical clustering 的核心代码

当 **n_clusters=89** 的时候，linkage 值的改变引起 AMI 的改变，详见表 9。我们发现当 linkage 值取 **average** 的时候，AMI 取得最大值 **0.8682575924833005**，运行时间为 **24.077788s**。gamma 值取大于 1 的时候，结果普遍较差，在此不列出结果。

表 9 Ward hierarchical clustering 调参表 (1)

linkage	AMI	运行时间(s)
Ward	0.6948668794233028	22.870451
complete	0.633430556622537	22.734272
average	0.8682575924833005	24.077788

我们固定 linkage 取值 average，测试该方法在选取不同 clusters 时的表现，详见表 10，我们发现当 n_clusters 取 80 的时候，AMI 取得最大值 **0.875384**，运行时间 22.521459，最优值优于 n_clusters=89 的情况。

表 10 Ward hierarchical clustering 调参表 (2)

n_clusters	AMI	运行时间(s)
85	0.873396	23.018158
80	0.875384	22.521459
70	0.834434	22.583766
60	0.817438	22.419841

当 affinity 取除 euclidean 之外的 manhattan 或 cosine 时，最优结果均不如取 euclidean 的情况，在此实验结果略去。

2.5 DBSCAN

DBSCAN 的核心代码如下。

```
81
82 def DBSCAN_(data):
83     t0 = time()
84     db=DBSCAN(eps = 1.1, min_samples = 1).fit(data)
85     label_pred=db.labels_
86     time_use=time()-t0
87     n_clusters = len(set(label_pred))
88     print("聚类得到的cluster数目为: ",n_clusters)
89     return label_pred,time_use
90
```

图 12 DBSCAN 的核心代码

我们在固定参数 eps 为 1 的情况下，调整参数 min_samples 使得 AMI 最大化，详情见表 11。最终我们发现当 eps 为 1，min_samples 为 2 的情况下 AMI 取得最大值 **0.368993**，运行时间 76.109303s。

表 11 DBSCAN 调参表（1）

eps	min_samples	AMI	运行时间(s)
1	1	0.272834	75.518468
1	2	0.368993	76.109303
1	3	0.316936	75.943875
1	4	0.266976	76.912437.

我们在固定参数 min_samples 为 1 的情况下，调整参数 eps 使得 AMI 最大化，详情见表 12。最终我们发现当 eps 为 1.1，min_samples 为 1 的情况下，AMI 取得最大值 **0.439140**，运行时间为 77.088685，此结果要优于参数表 9 所探讨的参数设置。

表 12 DBSCAN 调参表（2）

eps	min_samples	AMI	运行时间(s)
0.5	1	0.032440	9.223348
0.6	1	0.042392	11.669970
0.7	1	0.061659	16.524368
0.8	1	0.088532	25.970736
0.9	1	0.151589	42.258640
1.1	1	0.439140	77.088685
1.2	1	0.209495	75.698330
1.3	1	0.000528	76.858662
1.4	1	0.000000	78.167717
1.5	1	0.000000	75.911795

2.5 Gaussian mixtures

Gaussian mixture models 的核心代码如下。

```
92 def Gaussian_mixtures(data):
93     t0 = time()
94     gmm=mixture.GaussianMixture(n_components=70,covariance_type='spherical').fit(data)
95     label_pred=gmm.predict(data)
96     time_use=time()-t0
97
98     return label_pred,time_use
```

图 13 Gaussian mixture models 核心代码

由于 Gaussian mixture models 的参数 covariance_type 设置为 full、tied 与 diag 时，运行速度非常慢，甚至发生内存溢出的情况，所以本次实验主要考虑 covariance_type 为 spherical 的情况，详情见表 13, AMI 随着 n_components 的变化而变化，最终当 n_components 取值为 70 的时候，AMI 取得最大值 **0.709347**，运行时间 **8.277973s**。

表 13 Gaussian mixtures 调参表 (1)

n_components	AMI	运行时间(s)
6	0.267917	2.418555
7,	0.188810	2.565127
8	0.285478	2.127326
10	0.311632	2.740238
20	0.473211	4.238744
30	0.537531	6.234005
40	0.588370	6.077356
50	0.623506	6.145651
60	:0.695922	7.609859
70	0.709347	8.277973
80	0.671385	9.624450
89	0.680410	8.699822

我们再设置 covariance_type 为 tied，测试模型在 n_components 取值为 70 与 89 时的表现，见表 14 。我们发现当 n_components 取值为 70 的时候，AMI 取得最大值 **0.717415**，运行时间为 **175.478829**。

表 14 Gaussian mixtures 调参表 (2)

n_components	AMI	运行时间(s)
70	0.717415	175.478829
89	0.667417	225.105440

2.5 Birch

Birch 的核心代码如下。

```
99
100 def birch(data):
101     t0 = time()
102     label_pred = Birch(n_clusters = 70,threshold = 0.4, branching_factor =70).fit_predict(data)
103     time_use=time()-t0
104
105     return label_pred,time_use
106
```

图 14 Birch 核心代码

当 n_clusters = 89 时，我们调整模型参数 threshold、branching_factor 似 NM1 最大化，详见表 15 我们发现当 threshold 取 0.4，branching_factor 取 50 的时候，AMI 达到最大值 **0.709309**，运行时间为 18.888673。

表 15 Birch 调参表（1）

threshold	branching_factor	AMI	运行时间(s)
0.4	40	0.708061	18.727634
0.1	50	0.689846	22.979831
0.2	50	0.690894	22.652373
0.3	50	0.698443	19.757403.
0.4	50	0.709309	18.888673
0.5	50	0.703362	15.791411
0.6	50	0.707769	12.468891
0.7	50	0.701796	9.786468
0.8	50	0.687208	7.231609
0.9	50	0.449066	5.142075

当 n_clusters = 70 时，我们调整模型参数 threshold、branching_factor 似 NM1 最大化，详见表 16。我们发现当 threshold 取 0.4，branching_factor 取 70 的时候，AMI 达到最大值 **0.754462**，运行时间为 17.408954，大于在 n_clusters = 89 时的最优值。

表 16 Birch 调参表（2）

threshold	branching_factor	AMI	运行时间(s)
0.1	50	0.722500	21.469722
0.5	50	0.727832	14.994222
0.4	40	0.743386	18.974495
0.4	60	0.745756	17.758003
0.4	70	0.754462	17.408954
0.4	80	0.743896	17.417895
0.4	90	0.737519	17.849248
0.4	100	0.737247	18.029610

三、 聚类结果评测

因为 AMI 比 NMI 在偶然情况处理更合理，相对更合理一些，**本文模型调参时均用 AMI 作为评测指标**，接下来将会评测所有模型在最佳参数状态下的 NMI 与 AMI 指标评测与运行时间的评测。

3.1 NMI 与 AMI

所有模型均在最佳参数状态的 NMI 与 AMI 结果见表 17 与图 15。

表 17 NMI 与 AMI 评测（所有模型均在最佳参数状态）

	NMI	AMI	运行时间(s)
KNN	0.779365	0.745205	58.606697
Affinity Propagation	0.788888	0.654406	36.910559
Mean-shift	0.703239	0.478830	69.337893
Spectral clustering	0.740351	0.663427	3.623575
Ward hierarchical clustering	0.899808	0.875384	22.521459
DBSCAN	0.773610	0.439140	77.088685
Gaussian mixture models	0.780075	0.709347	8.277973
Birch	0.803051	0.754462	17.408954

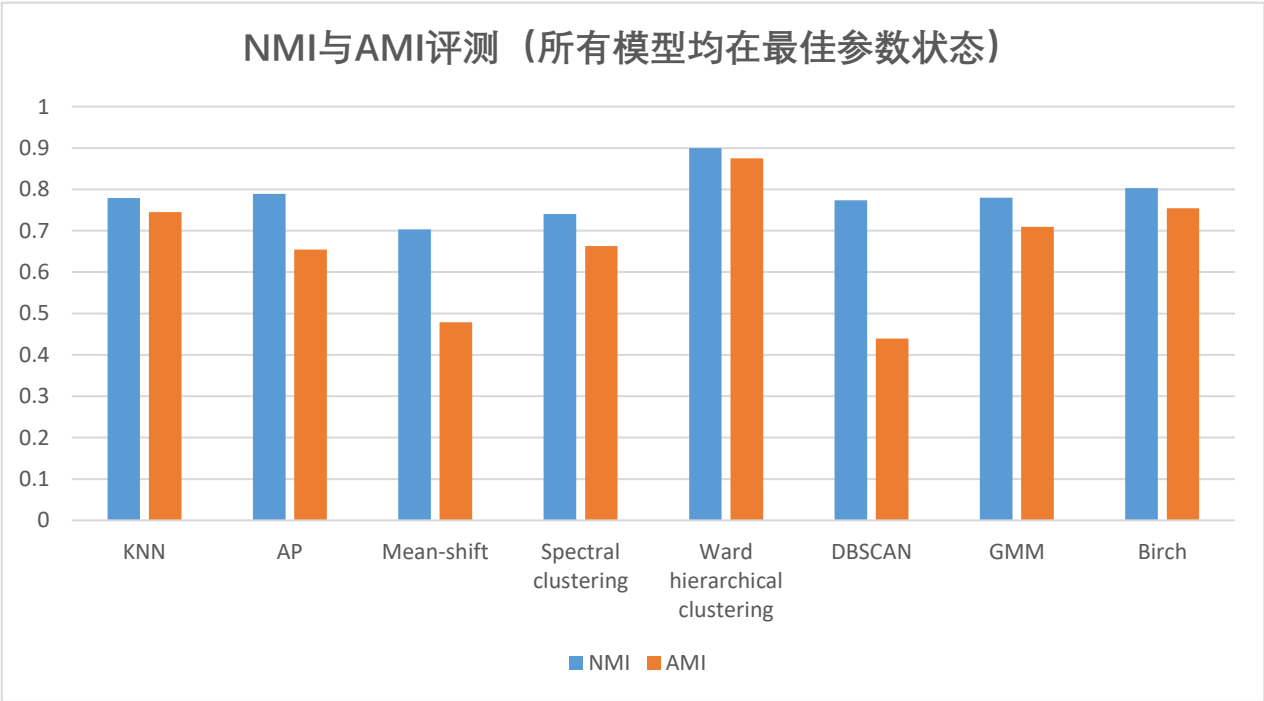


图 15 NMI 与 AMI 评测（所有模型均在最佳参数状态）

由表 17 与图 15 我们发现，**NMI 与 AMI 两个指标基本趋同**（在密度型聚类方法上略有差别），**Ward hierarchical clustering 得到了最高的聚类指标**，其次是 **Birch 与 Gaussian mixture models**，这三个模型表现出对文本内容进行建模的强大能力。

3.2 运行时间评测

模型在最佳参数状态下的时间评测见图 16。

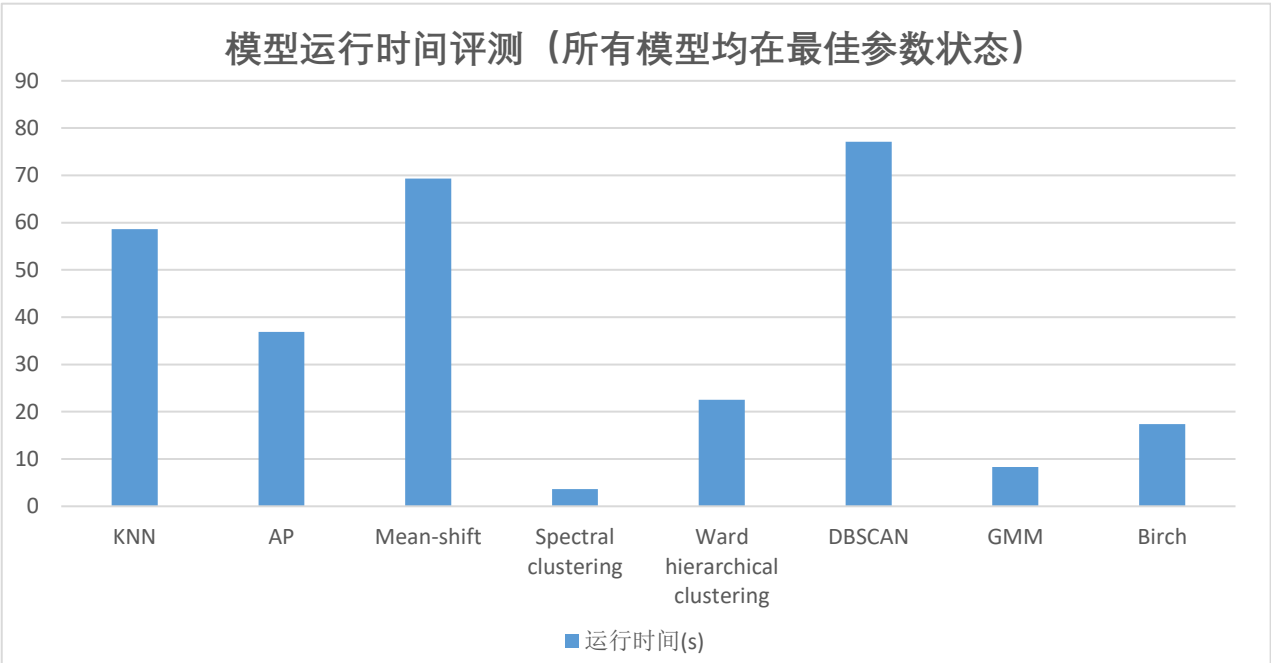


图 16 模型运行时间评测（所有模型均在最佳参数状态）

由图 16 我们可以发现，**Ward hierarchical clustering**、**Birch** 与 **Gaussian mixture models** 三个模型在 **NMI** 与 **AMI** 表现最好的三个模型在**运行时间**上也表现的非常出色，仅次于**运行时间最短**的 **Spectral clustering**。而 DBSCAN、Mean-shift、KNN 三个模型运行时间最长，比较耗费时间，这三个模型在 **NMI** 与 **AMI** 上的表现也比较一般。

四、结论

本次聚类算法的实现与评测增强了我的代码能力与对聚类模型的理解。

1. **Ward hierarchical clustering**、**Birch** 与 **Gaussian mixture models** 三个模型非常适合用于文本内容的聚类，并且三个模型时间效率较高。
2. **Spectral clustering** 模型在时间效率上表现地最好。
3. **密度型聚类方法**（例如 DBSCAN 与 Mean-shift）往往时间效率比较低，并且在 **NMI** 与 **AMI** 两个评测指标上差距较大（AMI 上表现分值偏低）。