

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.....	2
1.1 文本预处理	2
1.1.1 分词.....	2
1.1.2 去停止词	3
1.1.3 抽取词干	3
1.2 划分训练集与测试集	4
1.3 构建字典	5
1.4 计算 TF-IDF 值	6
1.4.1 训练集 IDF 值计算	6
1.4.2 训练集 TF 值计算	7
1.4.3 测试集的 IDF 值计算.....	7
1.4.4 测试集的 TF 值计算.....	8
1.5 将 embedding 转化为矩阵	8
二、KNN 的构建	10
2.1 读取训练集和测试集的 embedding 矩阵.....	10
2.2 计算 cosin similarity	10
2.3 排序	11
三、实验	12
3.1 N-fold 验证集实验	12
3.2 测试集实验结果	13
四、结论	14

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

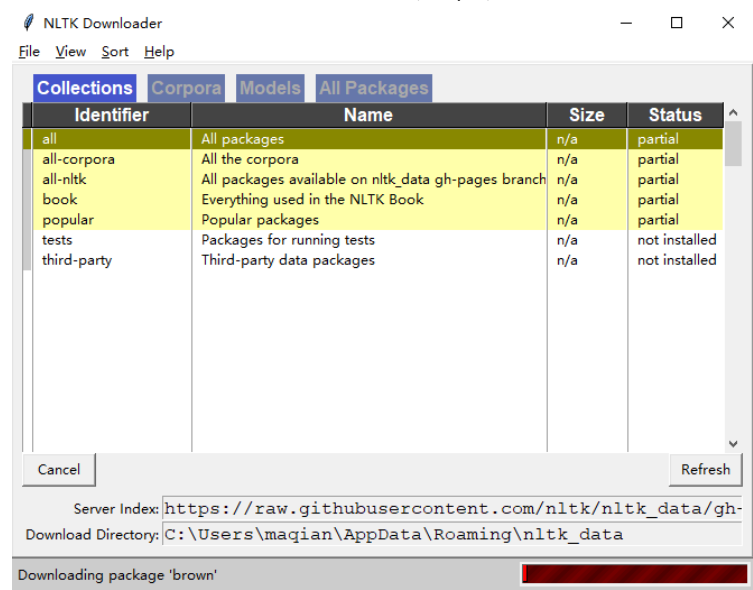


图 2 NLTK 安装

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

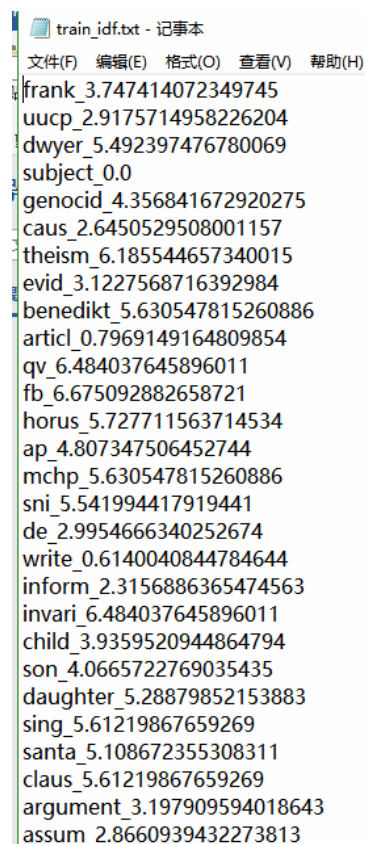
字典中的每个单词针对所有文档的 IDF 值是一致的，但是 TF 值每个文档都不一致，所以 IDF 值只需要算一次，然后存储调用，这样更方便。

```
#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

1.4.4 测试集的 TF 值计算

字典中的每个词在测试集的 TF 计算与在训练集中的计算类似，得到 TF 值后，再与 IDF 相乘得到测试集的 TF-IDF 值。测试集的 embedding 单独存放在一个 list 中。

```
#计算测试集上的tf-idf值
def test_tf_idf(word, test_doc, idf_dict):
    if word not in test_doc:
        tf_sore=0
    else:
        tf_sore=math.log(test_doc[word])+1
    idf_sore=idf_dict[word]
    tf_idf=tf_sore*idf_sore

    return tf_idf
```

图 16 测试集 TF-IDF 计算方法体

```
#计算测试集的tf-idf总值
test_tf_idf_doc=[]
for i,test_doc in enumerate(test_total_list):
    tf_idf_word=[]
    print("读取测试集的文档：")
    print(test_doc)
    print('\n')
    for word in new_word_dict:
        tf_idf_word.append(test_tf_idf(word,test_doc,test_idf_dict[i]))
    test_tf_idf_doc.append(tf_idf_word)
```

图 17 测试集 TF-IDF 计算方法调用

1.5 将 embedding 转化为矩阵

为了后期 KNN 方便计算，在这里我使用 **numpy** 方法，将测试集与训练集的 embedding 转化为矩阵。有以下几个注意事项：

1. 训练集矩阵维度为 15056×15749 ，测试集的矩阵维度为 3772×15749 ，矩阵的每一行对应一个文档，每一列对应一个字典中的特征词。
2. 为了方便调用，将训练集和测试集的矩阵保存，保存之后 VSM 部分结束。

[illegible]

图 18 训练集 embedding

[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:

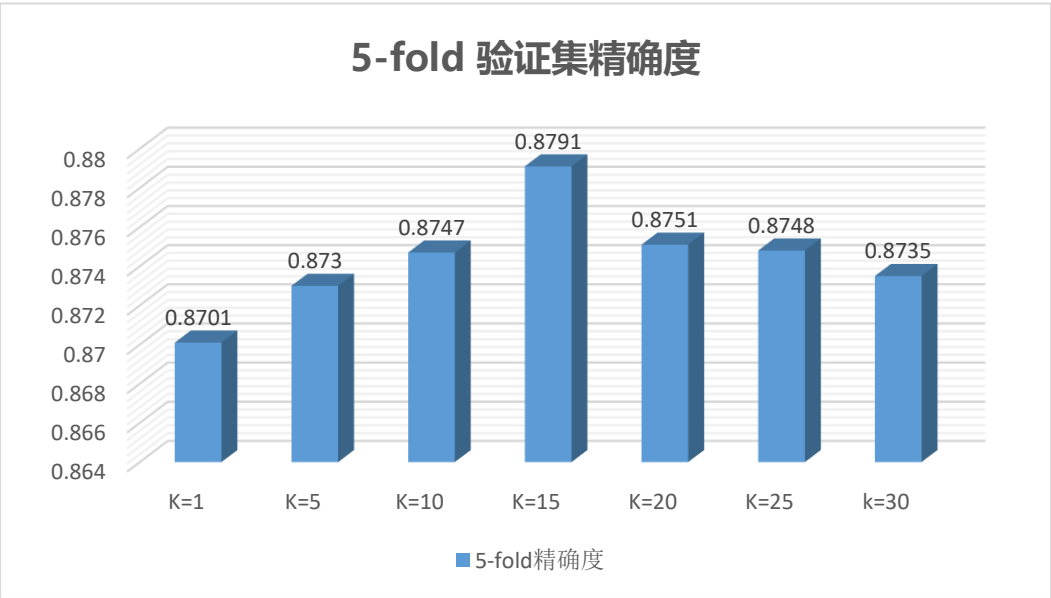


图 23 验证集结果

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。