## 背景

文本挖掘是指从大量文本数据中抽取实现未知的、可理解的、最终可用的知识的过程，同时运用这些知识更好地组织信息以便将来参考。即从非结构化的文本中寻找知识的过程。

目前文本挖掘主要有7个主要领域：

* 搜索和信息检索IR
* 文本聚类：使用聚类方法对词汇、片段、段落或文件进行分组和归类
* 文本分类：对片段、段落或文件进行分组和归类，在使用数据挖掘分类方法的基础上，经过训练地标记实例模型
* Web挖掘：在互联网上进行数据和文本挖掘，病特别关注网络的规模和相互联系
* 信息抽取IE：从非结构文本中识别和抽取有关的事实和关系；从非结构化或者半结构化的文本中抽取结构化的抽取出结构化数据的过程
* 自然语言处理NLP：从语法、语义的角度发现语言本质的结构和所表达的意义

## 文本分类系统（python 3.5）

中文语言的文本分类技术和流程，主要包括下面几个步骤：  
1. 预处理：去除文本噪声信息，例如HTML标签，文本格式转换，检测句子边界

2. 中文分词：使用中文分词器为文本分词，并去除停用词

3. 构建词向量空间：统计文本词频，生成文本的词向量空间

4. 权重策略——TF-IDF：使用TF-IDF发现特征词，并抽取为反映文档主题的特征

5. 分类词：使用算法训练分类器

6. 评价分类结果

***1. 预处理***

a. 选择处理文本的范围

b. 建立分类文本语料库

* 训练集语料

已经分好类的文本资源

* 测试集语料

待分类的文本语料，可以使训练集的一部分，也可以是外部来源的文本语料

c. 文本格式转化：使用Python的lxml库去除html标签

d. 检测句子边界：标记句子的结束

***2. 中文分词***

分词是将连续的字序列按照一定的规范重新组合成词序列的过程，中文分词即将一个汉字序列（句子）切分成一个个独立的单词，中文分词很复杂，从某种程度上并不完全是一个算法问题，最终概率论解决了这个问题，算法是基于概率图模型的条件随机场（CRF）

分词是自然语言处理中最基本，最底层的模块，分词精度对后续应用模块的影响很大，文本或句子的结构化表示是语言处理中最核心的任务，目前文本的结构化表示分为四大类：词向量空间、主体模型、依存句法的树表示、RDF的图表示。

下面给出中文词的示例代码：

*# -\*- coding: utf-8 -\*-***import** os  
**import** jieba  
  
  
**def** savefile(savepath, content):  
 fp = open(savepath, **"w"**,encoding=**'gb2312'**, errors=**'ignore'**)  
 fp.write(content)  
 fp.close()  
  
  
**def** readfile(path):  
 fp = open(path, **"r"**, encoding=**'gb2312'**, errors=**'ignore'**)  
 content = fp.read()  
 fp.close()  
 **return** content  
  
*# corpus\_path = "train\_small/" # 未分词分类预料库路径  
# seg\_path = "train\_seg/" # 分词后分类语料库路径*corpus\_path = **"test\_small/"** *# 未分词分类预料库路径*seg\_path = **"test\_seg/"** *# 分词后分类语料库路径*catelist = os.listdir(corpus\_path) *# 获取改目录下所有子目录***for** mydir **in** catelist:  
 class\_path = corpus\_path + mydir + **"/"** *# 拼出分类子目录的路径* seg\_dir = seg\_path + mydir + **"/"** *# 拼出分词后预料分类目录* **if not** os.path.exists(seg\_dir): *# 是否存在，不存在则创建* os.makedirs(seg\_dir)  
 file\_list = os.listdir(class\_path)  
 **for** file\_path **in** file\_list:  
 fullname = class\_path + file\_path  
 content = readfile(fullname).strip() *# 读取文件内容* content = content.replace(**"\r\n"**, **""**).strip() *# 删除换行和多余的空格* content\_seg = jieba.cut(content)  
 savefile(seg\_dir + file\_path, **" "**.join(content\_seg))  
  
print(**"分词结束"**)

为了后续生成词向量空间模型的方便，这些分词后的文本信息还要转换成文本向量信息并对象化，利用了Scikit-Learn库的Bunch数据结构，具体代码如下：

**import** os  
  
**import** pickle  
**from** sklearn.datasets.base **import** Bunch  
*#Bunch 类提供了一种key，value的对象形式  
#target\_name 所有分类集的名称列表  
#label 每个文件的分类标签列表  
#filenames 文件路径  
#contents 分词后文件词向量形式***def** readfile(path):  
 fp = open(path, **"r"**, encoding=**'gb2312'**, errors=**'ignore'**)  
 content = fp.read()  
 fp.close()  
 **return** content  
  
bunch=Bunch(target\_name=[],label=[],filenames=[],contents=[])  
  
*# wordbag\_path="train\_word\_bag/train\_set.dat"  
# seg\_path="train\_seg/"*wordbag\_path=**"test\_word\_bag/test\_set.dat"**seg\_path=**"test\_seg/"**catelist=os.listdir(seg\_path)  
bunch.target\_name.extend(catelist)*#将类别信息保存到Bunch对象***for** mydir **in** catelist:  
 class\_path=seg\_path+mydir+**"/"** file\_list=os.listdir(class\_path)  
 **for** file\_path **in** file\_list:  
 fullname=class\_path+file\_path  
 bunch.label.append(mydir)*#保存当前文件的分类标签* bunch.filenames.append(fullname)*#保存当前文件的文件路径* bunch.contents.append(readfile(fullname).strip())*#保存文件词向量  
  
#Bunch对象持久化*file\_obj=open(wordbag\_path,**"wb"**)  
pickle.dump(bunch,file\_obj)  
file\_obj.close()  
  
print(**"构建文本对象结束"**)

***3. 向量空间模型***

由于文本在储存为向量空间时维度较高，为节省储存空间和提高搜索效率，在文本分类之前会自动过滤掉某些字词，这些字或词被称为停用词，停用此表可以到点这里下载。

4. 权重策略：TF-IDF方法

在一份给定的文件里，词频 (term frequency, TF) 指的是某一个给定的词语在该文件中出现的次数。这个数字通常会被归一化（分子一般小于分母 区别于IDF），以防止它偏向长的文件。（同一个词语在长文件里可能会比短文件有更高的词频，而不管该词语重要与否。）

逆向文件频率 (inverse document frequency, IDF)是一个词语普遍重要性的度量。某一特定词语的IDF，可以由总文件数目除以包含该词语之文件的数目，再将得到的商取对数得到。

某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF。因此，TF-IDF倾向于过滤掉常见的词语，保留重要的词语。

TFIDF的主要思想是：如果某个词或短语在一篇文章中出现的频率TF高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。TFIDF实际上是：TF \* IDF，TF词频(Term Frequency)，IDF反文档频率(Inverse Document Frequency)。TF表示词条在文档d中出现的频率（另一说：TF词频(Term Frequency)指的是某一个给定的词语在该文件中出现的次数）。IDF的主要思想是：如果包含词条t的文档越少，也就是n越小，IDF越大，则说明词条t具有很好的类别区分能力。如果某一类文档C中包含词条t的文档数为m，而其它类包含t的文档总数为k，显然所有包含t的文档数n=m+k，当m大的时候，n也大，按照IDF公式得到的IDF的值会小，就说明该词条t类别区分能力不强。（另一说：IDF反文档频率(Inverse Document Frequency)是指果包含词条的文档越少，IDF越大，则说明词条具有很好的类别区分能力。）但是实际上，如果一个词条在一个类的文档中频繁出现，则说明该词条能够很好代表这个类的文本的特征，这样的词条应该给它们赋予较高的权重，并选来作为该类文本的特征词以区别与其它类文档。这就是IDF的不足之处.

在一份给定的文件里，词频（term frequency，TF）指的是某一个给定的词语在该文件中出现的频率。这个数字是对词数(term count)的归一化，以防止它偏向长的文件。（同一个词语在长文件里可能会比短文件有更高的词数，而不管该词语重要与否。）对于在某一特定文件里的词语 来说，它的重要性可表示为：

分子是该词在文件中出现的次数，分母是在文件中所有字词的出现次数之和

逆向文件频率（IDF）是一个词语普遍重要性的度量，某一特定词语的IDF，可以由总文件数目除以包含该词语的文件的数目，再将得到的商取对数：

|D|是语料库中的文件总数，j是包含词语的文件数目，如果该词语不在语料库中，就会导致分母为零，因此一般情况下分母还要额外再加上1

之后计算词频和逆向文件频率的乘积，某一特定文件内的高词语频率，以及该词语在整个文件集合中的低文件频率，可以产生出高权重的TF-IDF，因此TF-IDF倾向于过滤掉常见的词语，保留重要的词语。代码如下：

**import** os  
**from** sklearn.datasets.base **import** Bunch  
**import** pickle*#持久化类***from** sklearn **import** feature\_extraction  
**from** sklearn.feature\_extraction.text **import** TfidfTransformer*#TF-IDF向量转换类***from** sklearn.feature\_extraction.text **import** TfidfVectorizer*#TF-IDF向量生成类***def** readbunchobj(path):  
 file\_obj=open(path,**"rb"**)  
 bunch=pickle.load(file\_obj)  
 file\_obj.close()  
 **return** bunch  
**def** writebunchobj(path,bunchobj):  
 file\_obj=open(path,**"wb"**)  
 pickle.dump(bunchobj,file\_obj)  
 file\_obj.close()  
  
**def** readfile(path):  
 fp = open(path, **"r"**, encoding=**'gb2312'**, errors=**'ignore'**)  
 content = fp.read()  
 fp.close()  
 **return** content  
  
  
path=**"train\_word\_bag/train\_set.dat"**bunch=readbunchobj(path)  
  
*#停用词*stopword\_path=**"train\_word\_bag/hlt\_stop\_words.txt"**stpwrdlst=readfile(stopword\_path).splitlines()  
*#构建TF-IDF词向量空间对象*tfidfspace=Bunch(target\_name=bunch.target\_name,label=bunch.label,filenames=bunch.filenames,tdm=[],vocabulary={})  
*#使用TfidVectorizer初始化向量空间模型*vectorizer=TfidfVectorizer(stop\_words=stpwrdlst,sublinear\_tf=**True**,max\_df=0.5)  
transfoemer=TfidfTransformer()*#该类会统计每个词语的TF-IDF权值  
  
#文本转为词频矩阵，单独保存字典文件*tfidfspace.tdm=vectorizer.fit\_transform(bunch.contents)  
tfidfspace.vocabulary=vectorizer.vocabulary\_  
  
*#创建词袋的持久化*space\_path=**"train\_word\_bag/tfidfspace.dat"**writebunchobj(space\_path,tfidfspace)

***5.使用朴素贝叶斯分类模块***

常用的文本分类方法有kNN最近邻法，朴素贝叶斯算法和支持向量机算法，一般而言：

kNN算法原来最简单，分类精度尚可，但是速度最快支

朴素贝叶斯算法对于短文本分类的效果最好，精度很高

支持向量机算法的优势是支持线性不可分的情况，精度上取中

上文代码中进行操作的都是训练集的数据，下面是测试集（抽取字训练集），训练步骤和训练集相同，首先是分词，之后生成词向量文件，直至生成词向量模型，不同的是，在训练词向量模型时需要加载训练集词袋，将测试集产生的词向量映射到训练集词袋的词典中，生成向量空间模型，代码如下：

**import** os  
**from** sklearn.datasets.base **import** Bunch  
**import** pickle*#持久化类***from** sklearn **import** feature\_extraction  
**from** sklearn.feature\_extraction.text **import** TfidfTransformer*#TF-IDF向量转换类***from** sklearn.feature\_extraction.text **import** TfidfVectorizer*#TF-IDF向量生成类***from** TF\_IDF **import** space\_path  
  
  
**def** readbunchobj(path):  
 file\_obj=open(path,**"rb"**)  
 bunch=pickle.load(file\_obj)  
 file\_obj.close()  
 **return** bunch  
**def** writebunchobj(path,bunchobj):  
 file\_obj=open(path,**"wb"**)  
 pickle.dump(bunchobj,file\_obj)  
 file\_obj.close()  
**def** readfile(path):  
 fp = open(path, **"r"**, encoding=**'gb2312'**, errors=**'ignore'**)  
 content = fp.read()  
 fp.close()  
 **return** content  
  
*#导入分词后的词向量bunch对象*path=**"test\_word\_bag/test\_set.dat"**bunch=readbunchobj(path)  
  
*#停用词*stopword\_path=**"train\_word\_bag/hlt\_stop\_words.txt"**stpwrdlst=readfile(stopword\_path).splitlines()  
  
*#构建测试集TF-IDF向量空间*testspace=Bunch(target\_name=bunch.target\_name,label=bunch.label,filenames=bunch.filenames,tdm=[],vocabulary={})  
  
*#导入训练集的词袋*trainbunch=readbunchobj(**"train\_word\_bag/tfidfspace.dat"**)  
  
*#使用TfidfVectorizer初始化向量空间*vectorizer=TfidfVectorizer(stop\_words=stpwrdlst,sublinear\_tf=**True**,max\_df=0.5,vocabulary=trainbunch.vocabulary)  
transformer=TfidfTransformer();  
testspace.tdm=vectorizer.fit\_transform(bunch.contents)  
testspace.vocabulary=trainbunch.vocabulary  
  
*#创建词袋的持久化*space\_path=**"test\_word\_bag/testspace.dat"**writebunchobj(space\_path,testspace)

下面执行多项式贝叶斯算法进行测试文本分类并返回精度，代码如下：

**import** pickle  
**from** sklearn.naive\_bayes **import** MultinomialNB *# 导入多项式贝叶斯算法包*  
  
**def** readbunchobj(path):  
 file\_obj = open(path, **"rb"**)  
 bunch = pickle.load(file\_obj)  
 file\_obj.close()  
 **return** bunch  
  
  
*# 导入训练集向量空间*trainpath = **"train\_word\_bag/tfidfspace.dat"**train\_set = readbunchobj(trainpath)  
*# d导入测试集向量空间*testpath = **"test\_word\_bag/testspace.dat"**test\_set = readbunchobj(testpath)  
*# 应用贝叶斯算法  
# alpha:0.001 alpha 越小，迭代次数越多，精度越高*clf = MultinomialNB(alpha=0.001).fit(train\_set.tdm, train\_set.label)  
  
*# 预测分类结果*predicted = clf.predict(test\_set.tdm)  
total = len(predicted);rate = 0  
**for** flabel, file\_name, expct\_cate **in** zip(test\_set.label, test\_set.filenames, predicted):  
 **if** flabel != expct\_cate:  
 rate += 1  
 print(file\_name, **": 实际类别："**, flabel, **"-->预测分类："**, expct\_cate)  
*# 精度*print(**"error\_rate:"**, float(rate) \* 100 / float(total), **"%"**)

***6.分类结果评估***

机器学习领域的算法评估有三个基本指标：

* 召回率（recall rate,查全率）：是检索出的相关文档数与文档库中所有相关文档的比率，衡量的是检索系统的查全率

召回率=系统检索到的相关文件/系统所有相关的文件综述

* 准确率（Precision，精度）：是检索出的相关文档数于检索出的文档总数的比率，衡量的是检索系统的查准率

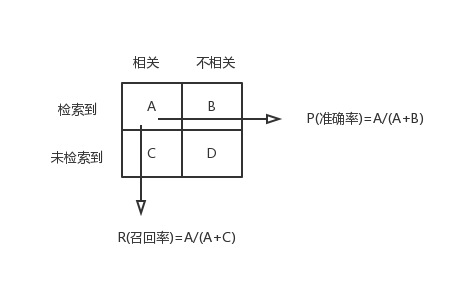
准确率=系统检索到的相关文件/系统所有的检索到的文件数

准确率和召回率是相互影响的，理想情况下是二者都高，但是一般情况下准确率高，召回率就低；召回率高，准确率就低

* F-Score（）：计算公式为：

当=1时就是最常见的-Measure

三者关系如下：



具体评估代码如下：

**import** numpy **as** np  
**from** sklearn **import** metrics  
*#评估***def** metrics\_result(actual,predict):  
 print(**"精度：{0:.3f}"**.format(metrics.precision\_score(actual,predict)))  
 print(**"召回：{0:0.3f}"**.format(metrics.recall\_score(actual,predict)))  
 print(**"f1-score:{0:.3f}"**.format(metrics.f1\_score(actual,predict)))  
metrics\_result(test\_set.label,predicted)