**北京邮电大学**

**2023--2024学年第1学期实验报告**

**课程名称： 人工智能原理**

**实验名称： 文本分类**

**实验完成人：**

**姓名：** 陈朴炎 **学号：**  2021211138

**日 期：** 2023 **年** 12**月**

目录

[1 实验内容 3](#_Toc154756424)

[1.1 实验目的 3](#_Toc154756425)

[1.2 实验类型 3](#_Toc154756426)

[1.3 实验要求 3](#_Toc154756427)

[1.4 实验内容 3](#_Toc154756428)

[1.5 实验验收 4](#_Toc154756429)

[2 分工说明 4](#_Toc154756430)

[3 实验环境 5](#_Toc154756431)

[4 实验设计及准备 5](#_Toc154756432)

[4.1 分词 5](#_Toc154756433)

[4.2 TF-IDF 7](#_Toc154756434)

[4.2.1 TF-IDF原理 7](#_Toc154756435)

[4.2.2 TF-IDF 算法步骤 8](#_Toc154756436)

[4.3 LDA 9](#_Toc154756437)

[4.4 SVM 10](#_Toc154756438)

[4.4.1 SVM特点 10](#_Toc154756439)

[4.4.2 SVM基础知识 11](#_Toc154756440)

[5 实验过程 14](#_Toc154756441)

[5.1 文本数据收集 14](#_Toc154756442)

[5.1.1 收集url 14](#_Toc154756443)

[5.1.2 收集文本 18](#_Toc154756444)

[5.2.2 提取特征数据 23](#_Toc154756445)

[5.3 构建TF-IDF矩阵空间 24](#_Toc154756446)

[5.4 LDA 主题模型构建 26](#_Toc154756447)

[5.5 SVM 27](#_Toc154756448)

[6 附录 32](#_Toc154756449)

[6.1 get\_url.py 32](#_Toc154756450)

[6.2get\_news\_treads.py 33](#_Toc154756451)

[6.3 splice\_word.py 35](#_Toc154756452)

[6.4 select\_data.py 37](#_Toc154756453)

[6.5 tfidf.py 38](#_Toc154756454)

[6.6 svm.py 42](#_Toc154756455)

# 文本数据的分类与分析

# 1 实验内容

## 1.1 实验目的

1. 掌握数据预处理的方法，对训练集数据进行预处理； 2. 掌握文本建模的方法，对语料库的文档进行建模； 3. 掌握分类算法的原理，基于有监督的机器学习方法，训练文本分类器； 4. 利用学习的文本分类器，对未知文本进行分类判别； 5. 掌握评价分类器性能的评估方法。

## 1.2 实验类型

数据挖掘算法的设计与编程实现

## 1.3 实验要求

1. 文本类别数：10类； 2. 训练集文档数：>=50000篇；每类平均5000篇。 3. 测试集文档数：>=50000篇；每类平均5000篇。 4. 分组完成实验，组员数量<=3，个人实现可以获得实验加分。

## 1.4 实验内容

利用分类算法实现对文本的数据挖掘，主要包括： 1. 语料库的构建，主要包括利用爬虫收集Web文档等； 2. 语料库的数据预处理，包括文档建模，如去噪，分词，建立数据字典，使用词袋模型或主题模型表达文档等； 注：使用主题模型，如LDA可以获得实验加分； 3. 选择分类算法（朴素贝叶斯/SVM/其他等），训练文本分类器，理解所选的分类算法的建模原理、实现过程和相关参数的含义； 4. 对测试集的文本进行分类 5. 对测试集的分类结果利用正确率和召回率进行分析评价：计算每类正确率、召回率，计算总体正确率和召回率，以及F-score。

## 1.5 实验验收

1. 编写实验报告，实验报告内容必须包括对每个阶段的过程描述，以及实验结果的截图展示。 2. 以线上方式验收实验。

# 2 分工说明

本次实验一个人独立完成。

实验过程大致分为：

1. 文档爬取

2. 类型选择

3. 文档分词、去停用词

4. 构建TF-IDF矩阵空间 / 构建LDA主题模型

5. 使用SVM训练文本分类器并预测评估

共收集一百三十多万个文本数据。筛选、分词后分出十类，每一类的训练集数量为8000篇，每一类的测试集数量介于5000到15000之间。

# 3 实验环境

windows11操作系统

python版本：3.10.7 64-bit

数据集收集：实现爬虫爬取数据

分词工具：jieba 0.42.1

scikit-learn版本：1.3.0

TF-IDF工具：

sklearn.feature\_extraction.text中的TfidfVectorizer

LDA主题模型：

sklearn.decomposition中的LatentDirichletAllocation

预测工具：

sklearn.metrics中的accuracy\_score、classification\_report、confusion\_matrix

VScode版本：1.85.1

# 4 实验设计及准备

## 4.1 分词

分词我通过jieba库来实现。

jieba的算法如下：

1. 基于前缀词典实现高效的词图扫描，生成句子中汉字所有可能成词情况所构成的有向无环图 (DAG) 2. 采用了动态规划查找最大概率路径, 找出基于词频的最大切分组合 3. 对于未登录词，采用了基于汉字成词能力的 HMM 模型，使用了 Viterbi 算法 jieba的主要功能如下：

jieba.cut 方法接受四个输入参数: 需要分词的字符串；cut\_all 参数用来控制是否采用全模式；HMM 参数用来控制是否使用 HMM 模型；use\_paddle 参数用来控制是否使用paddle模式下的分词模式，paddle模式采用延迟加载方式，通过enable\_paddle接口安装paddlepaddle-tiny，并且import相关代码； jieba.cut\_for\_search 方法接受两个参数：需要分词的字符串；是否使用 HMM 模型。该方法适合用于搜索引擎构建倒排索引的分词，粒度比较细待分词的字符串可以是 unicode 或 UTF-8 字符串、GBK 字符串。注意：不建议直接输入 GBK 字符串，可能无法预料地错误解码成 UTF-8jieba.cut 以及 jieba.cut\_for\_search 返回的结构都是一个可迭代的 generator，可以使用 for 循环来获得分词后得到的每一个词语(unicode)，或者用 jieba.lcut 以及 jieba.lcut\_for\_search 直接返回 list jieba.Tokenizer(dictionary=DEFAULT\_DICT) 新建自定义分词器，可用于同时使用不同词典。jieba.dt 为默认分词器，所有全局分词相关函数都是该分词器的映射。 jieba的词性名称如下：



图4-1 jieba词性标号示意图

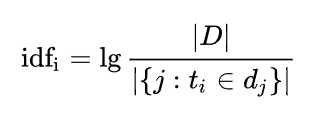
## 4.2 TF-IDF

### 4.2.1 TF-IDF原理

TF-IDF有两层意思：一层是词频“Term Frequency”缩写为TF，另一层是逆文档频率“Inverse Document Frequency”缩写为IDF。

词频指的是某一个给定的词语在该文档中出现的频率。这个数字是对词数的标准化，以防止它偏向长的文档。

逆向档案频率是一个词语普遍重要性的度量。某一特定词语的idf，可以由总文档数目除以包含该词语之档案的数目，再将得到的商取以2为底的对数得到



|D|为语料库中的文档总数；|{j:ti∈dj}|是包含词语ti的文档总数目，如果词语不在语料库中，就会导致分母为零，因此一般情况下会在分母 + 1。

字词的重要性随着它在档案中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降

当有TF和IDF后，将这两个词相乘，就能得到一个词的TF-IDF的值。某个词在文章中的TF-IDF越大，那么一般而言这个词在这篇文章的重要性会越高，所以通过计算文章中各个词的TF-IDF，由大到小排序，排在最前面的几个词，就是该文章的关键词。

可以看到，TF-IDF与一个词在文档中的出现次数成正比，与该词在整个语言中的出现次数成反比。所以，自动提取关键词的算法就很清楚了，就是计算出文档的每个词的TF-IDF值，然后按降序排列，取排在最前面的几个词。

### 4.2.2 TF-IDF 算法步骤

步骤1. 计算词频：

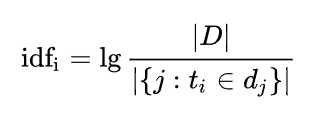
TF = 某个词在文章中的出现次数

考虑到文字有长短之分，为了便于不同文章比较，进行“词频”标准化

TF = 某个词在文章中的出现次数 ➗ 文章总次数

步骤2. 计算逆文档频率

需要一个语料库，来模拟语言的使用环境



步骤3. 计算TF-IDF

TF-IDF = TF × IDF

## 4.3 LDA

LDA主题模型主要用于推测文档的主题分布，可以将文档集中每篇文档的主题以概率分布的形式给出根据主题进行主题聚类或文本分类。 LDA主题模型不关心文档中单词的顺序，通常使用词袋特征（bag-of-word feature）来代表文档。 LDA模型认为主题可以由一个词汇分布来表示，而文章可以由主题分布来表示。比如有两个主题，美食和美妆。LDA说两个主题可以由词汇分布表示，他们分别是： {面包：0.4，火锅：0.5，眉笔：0.03，腮红：0.07} {眉笔：0.4，腮红：0.5，面包：0.03，火锅：0.07} 同样，对于两篇文章，LDA认为文章可以由主题分布这么表示： 《美妆日记》{美妆：0.8，美食：0.1，其他：0.1} 《美食探索》{美食：0.8，美妆：0.1，其他：0.1} 所以想要生成一篇文章，可以先以一定的概率选取上述某个主题，再以一定的概率选取那个主题下的某个单词，不断重复这两步就可以生成最终文章。 在LDA模型中，一篇文档生成的方式如下：

1. 从迪利克雷分布α中取样生成文档i的主题分布θi

2. 从主题的多项式分布θi中取样生成文档i的第j个词的主题zi

3. 从迪利克雷分布β中取样生成主题zi,j对应的词语分布Φzi,j

4. 从词语的多项式分布Φzi,j中采样生成最终词语wi,j

## 4.4 SVM

### 4.4.1 SVM特点

所谓VC维是对函数类的一种度量，可以简单的理解为问题的复杂程度，VC维越高，一个问题就越复杂。

正是因为SVM关注的是VC维，后SVM解决问题的时候，与样本的维数是无关的（甚至样本是上万维的都可以，这使得SVM很适合用来解决文本分类的问题，当然，有这样的能力也因为引入了核函数）

高维模式识别是指样本维数很高，例如文本的向量表示，如果没有经过降维处理，出现几万维的情况很正常，其他算法基本就没有能力应付了，SVM却可以，主要是因为SVM 产生的分类器很简洁，用到的样本信息很少（仅仅用到那些称之为“支持向量”的样本），使得即使样本维数很高，也不会给存储和计算带来大麻烦（相对照而言，kNN算法在分类时就要用到所有样本，样本数巨大，每个样本维数再一高，训练难度非常大）。

### 4.4.2 SVM基础知识

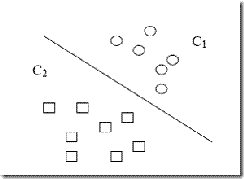


图4-2 线性分类器示意图

C1和C2是要区分的两个类别，在二维平面中它们的样本如上图所示。中间的直线就是一个分类函数，它可以将两类样本完全分开。

一般的，如果一个线性函数能够将样本完全正确的分开，就称这些数据是线性可分的，否则称为非线性可分的。

可以这样想：在一维空间里就是一个点，在二维空间里就是一条直线，三维空间里就是一个平面，可以如此想象下去，如果不关注空间的维数，这种线性函数还有一个统一的名称——超平面（Hyper Plane）

分类间隔：

训练样本Di=(xi,yi)：

xi就是文本向量（维数很高），yi就是分类标记。

在二元的线性分类中，这个表示分类的标记只有两个值，1和-1（用来表示属于还是不属于这个类）。

样本点到某个超平面的间隔： δi=yi(wxi+b) = |wxi+b|=|g(xi)|



图4-3 分类间隔示意图

一个点的集合（就是一组样本）到某个超平面的距离为此集合中离超平面最近的点的距离。

误分次数：

误分次数：代表分类器的误差。误分次数的上界由几何间隔决定！

几何间隔δ越大的解，它的误差上界越小。

因此最大化几何间隔成了我们训练阶段的目标。

误分次数 <= (2R/δ)^2

δ是样本集合到分类面的间隔

R是所有样本向量长度的最长值，代表着样本分布的范围有多广

松弛变量：

并非所有的样本点都有一个松弛变量与其对应。

实际上只有“离群点”才有，所有没离群的点松弛变量都等于0

松弛变量的值实际上标示出了对应的点到底离群有多远，值越大，点就越远。

惩罚因子C决定了你有多重视离群点带来的损失，显然当所有离群点的松弛变量的和一定时，你定的C越大，对目标函数的损失也越大，此时就暗示着你非常不愿意放弃这些离群点，最极端的情况是你把C定为无限大，这样只要稍有一个点离群，目标函数的值马上变成无限大，马上让问题变成无解，这就退化成了硬间隔问题。

惩罚因子C不是一个变量，整个优化问题在解的时候，C是一个你必须事先指定的值；

指定这个值以后，得到一个分类器，然后用测试数据看看结果怎么样，如果不够好，换一个C的值，再解一次优化问题，得到另一个分类器，再看看效果，如此就是一个参数寻优的过程，

这与优化问题本身决不是一回事，优化问题在解的过程中，C一直是定值。 优化问题求解的过程：

就是先试着确定一下w，也就是确定了前面图中的三条直线，这时看看间隔有多大，又有多少点离群，把目标函数的值算一算，再换一组三条直线

分类的直线位置如果移动了，有些原来离群的点会变得不再离群，而有的本来不离群的点会变成离群点），再把目标函数的值算一算，如此往复（迭代），直到最终找到目标函数最小时的w。

惩罚因子C

C所起的作用：表征你有多么重视离群点，C越大越重视，越不想丢掉它们

没有任何规定说必须对所有的松弛变量都使用同一个惩罚因子，我们完全可以给每一个离群点都使用不同的C，这时就意味着你对每个样本的重视程度都不一样

使用惩罚因子可以处理数据偏斜问题：给样本数量少的负类更大的惩罚因子，表示我们重视这部分样本

# 5 实验过程

## 5.1 文本数据收集

### 5.1.1 收集url

本次的实验数据我是在中国新闻网上收集的。

在即时新闻页面上可以查看往日的所有新闻，即使新闻链接为：

<https://www.chinanews.com.cn/scroll-news/news1.html>

页面示意图如下：



图5-1 中国新闻网即时新闻示意图

我们可以发现url为

<https://www.chinanews.com.cn/scroll-news/2023/1214/news.shtml>

其中，<https://www.chinanews.com.cn/> 为中国新闻网的基础域名，/scroll-news/ 表示是滚动新闻页面，2023是年份，1214是月份和日期。

打开网页的Elements，我们查看这个网页的html格式，可以看到，左端的url都保存在<content-left>/<content\_list>下的<url>里面。

因此，可以通过爬取这个前端元素来获取我们所需要的新闻的url。

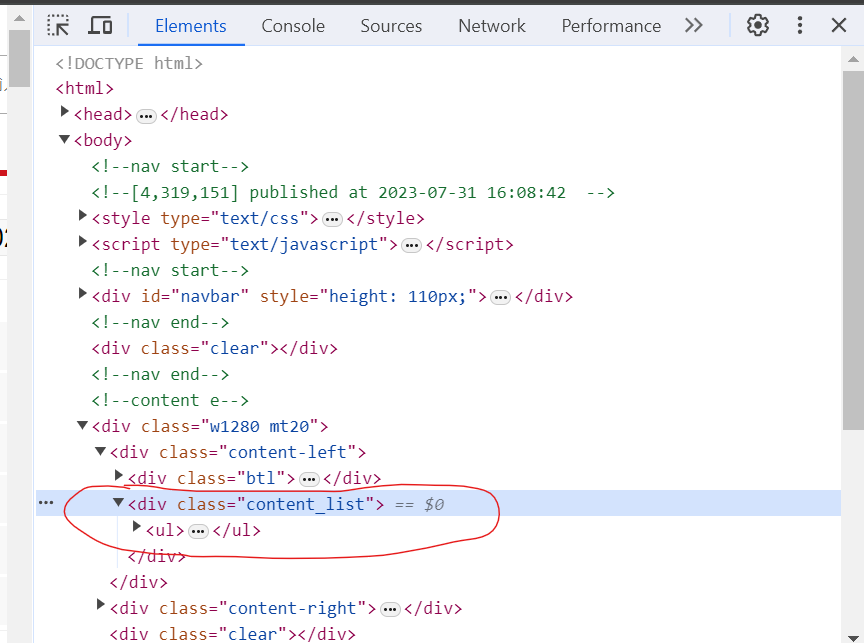


图5-2 网页页面元素示意图

通过以上信息，我们可以使用python的bs4、urllib和request来爬取中国新闻网滚动新闻的url信息，主要操作步骤如下：

1. 设置好基础的url部分，以及年月日列表

2. 通过request 里的get函数来到达某一日滚动新闻页面

3. 通过html parser解析html结构，找到content\_list和url列表

4. 将url爬取下来，放到列表中，写入到文件中

程序如下所示：

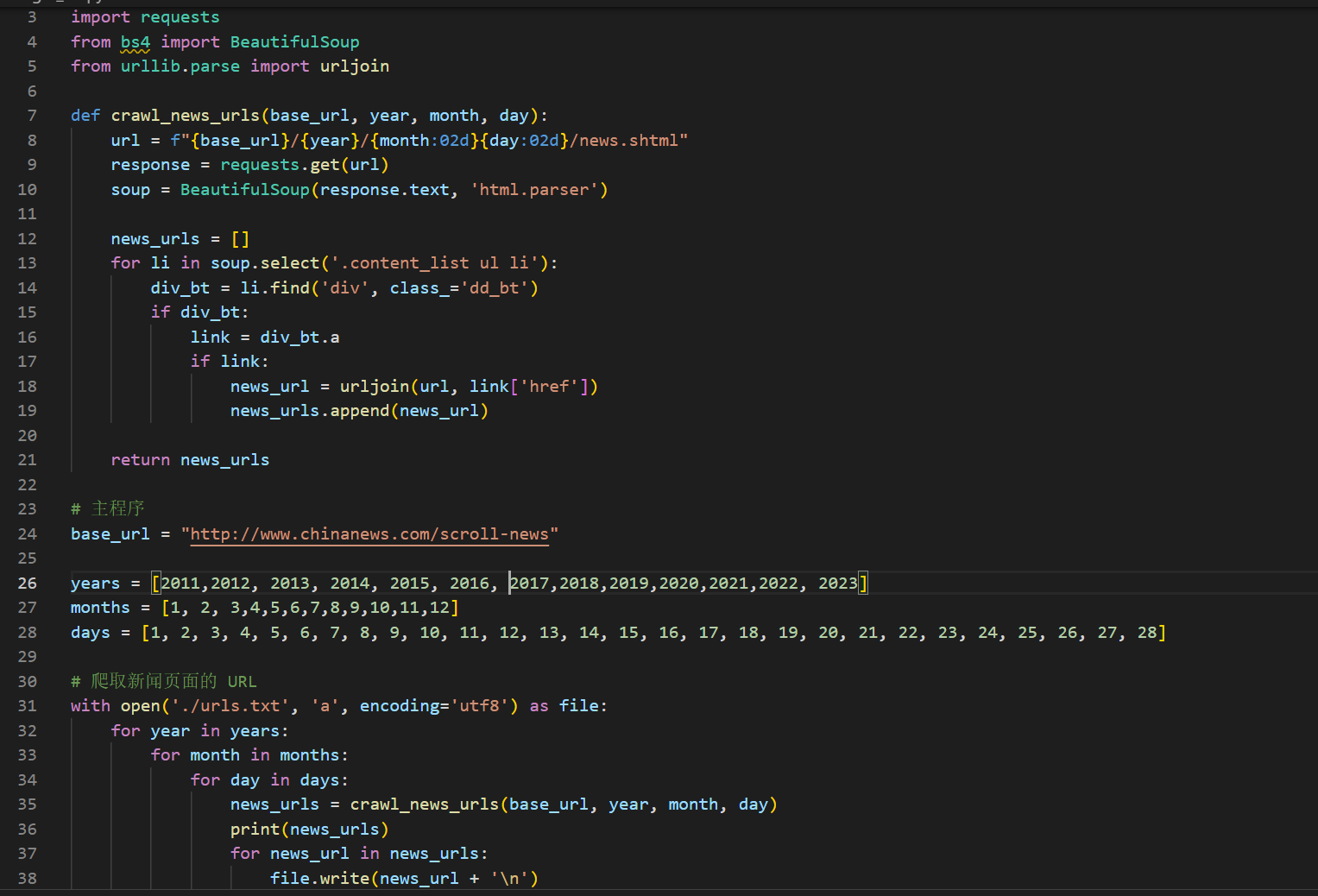


图5-3 收集url程序示意图

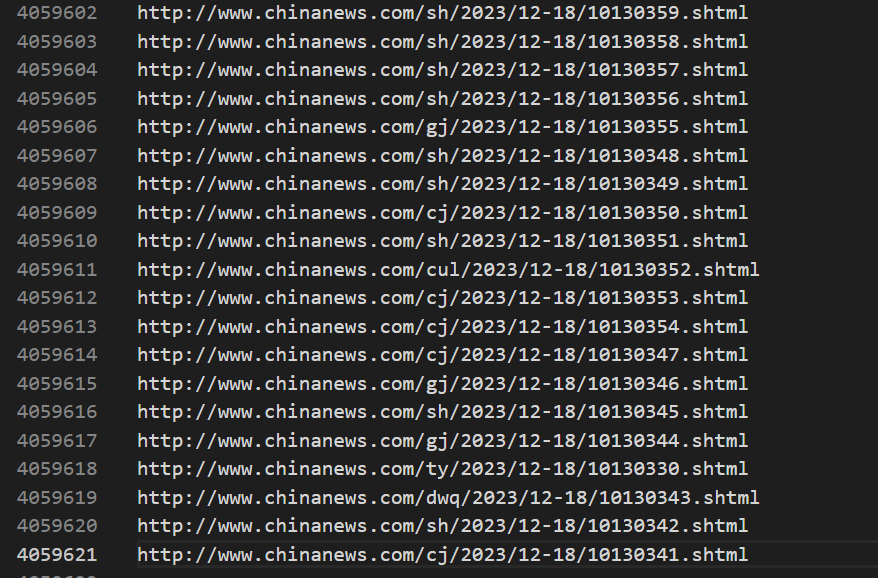


图5-4 url收集结果图

如上图所示，我从2011到2023年收集到了共4059621，四百万余条数据。

### 5.1.2 收集文本

打开收集到的某个url，里面的网页结构如图所示：



图5-5 新闻结构示意图

可以看到，新闻的主体内容是在<body>/<main\_content w1280>/<con\_left>/<content\_maincontent\_content>下的<left\_zw>

因此我们可以通过爬取left\_zw里<p>中的文字信息来获取新闻内容。

具体步骤如下：

1. 打开urls文件，获取urls列表

2. 通过request.get获取网页结构，并通过html parser解析

3. 查找left\_zw类，将<p>中的文本扒下来

4. 保存到文件中

经过本人试验，url中有hd的为无效url，有bbs的也为无效数据。同时要注意，2019年之前的数据是GB2312编码的，2019年之后的文本数据是UTF-8编码的，在爬取中得注意鉴别。

并且只有一个线程跑的话爬取一天都只能爬三万多条文本数据。因此我打算开多个线程。这里我开了400个线程（实际效果并不是单线程的400倍），同时爬取文本数据。

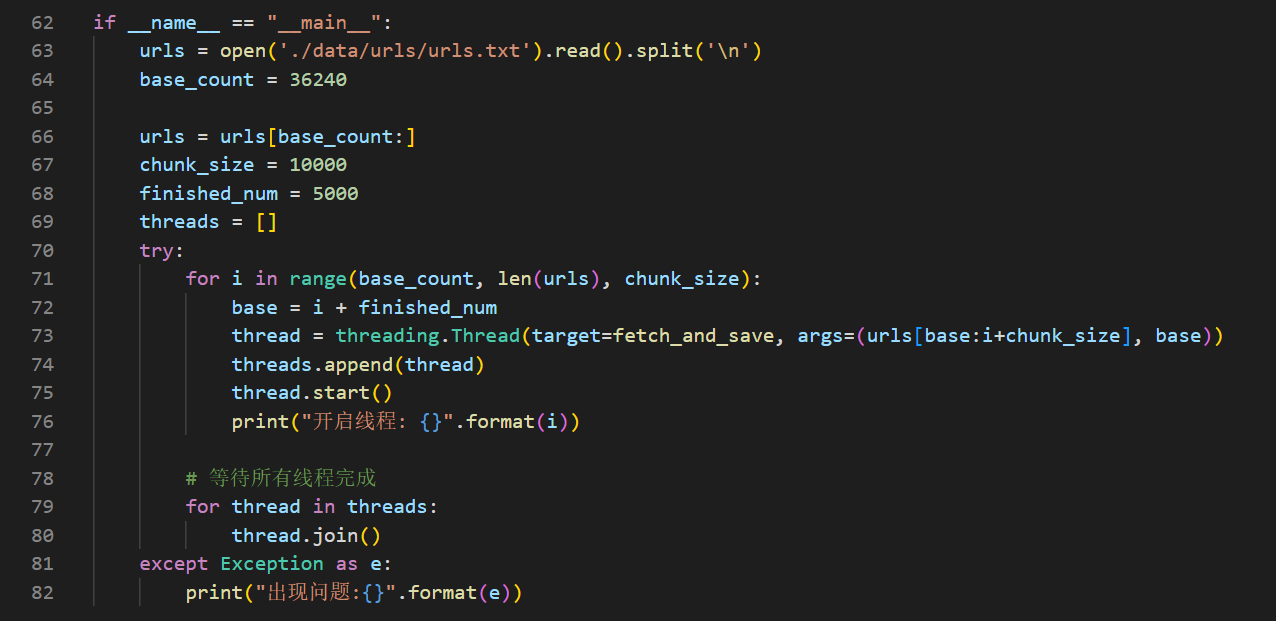


图5-6 多线程爬取文本数据

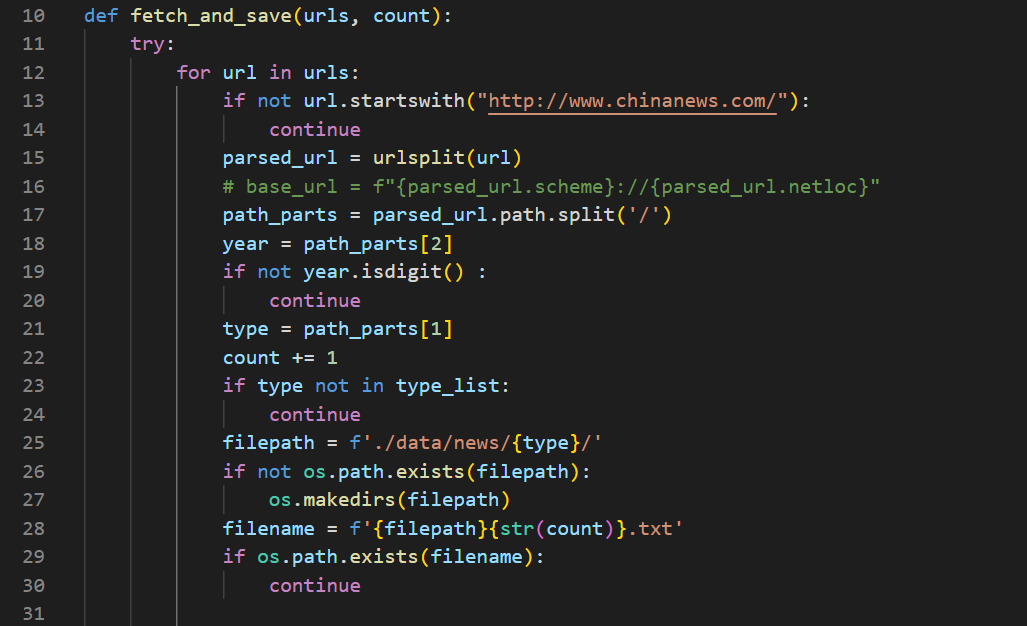


图5-7 爬取文本数据示意图1

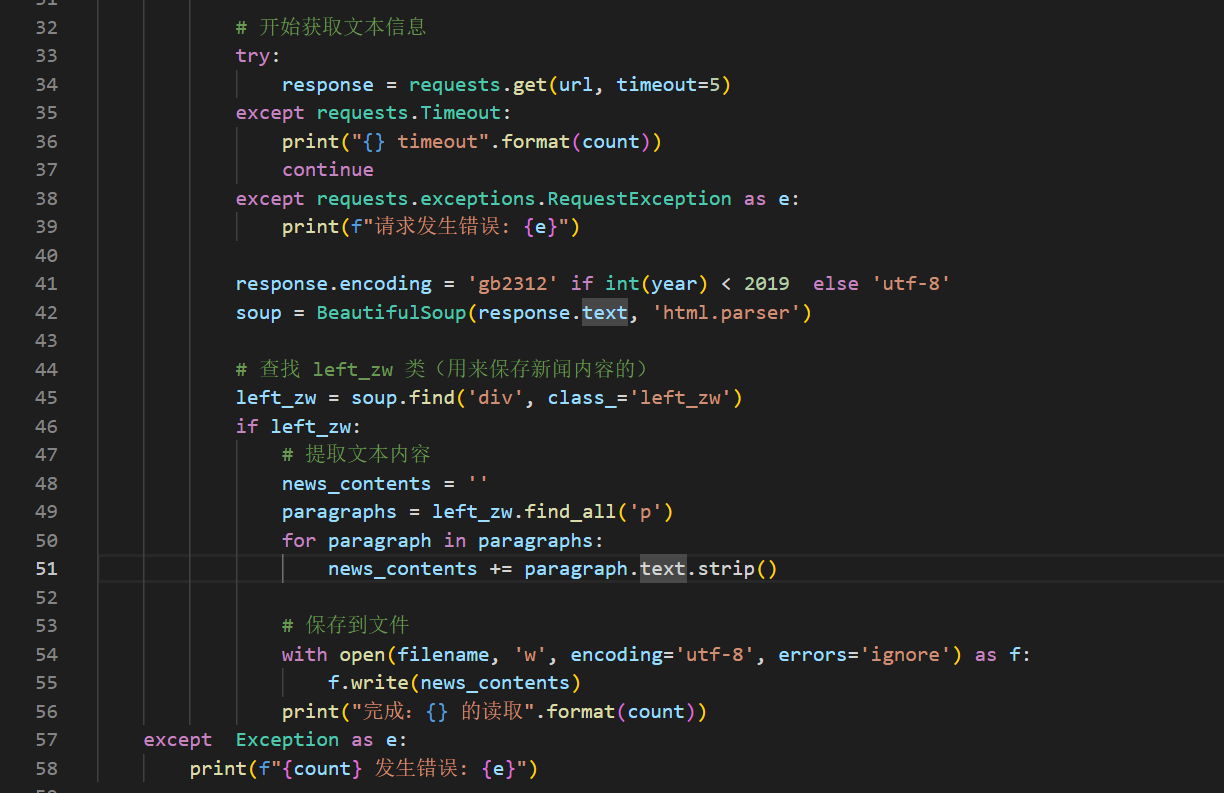


图5-8 爬取文本数据示意图2



图5-9 爬取文本数据结果示意图

在爬取完大一百三十多万条数据之后（此时已经花费两天时间），我统计了各个类型文本数据的分布，文本数量较多的类型和数量对应如下：

|  |
| --- |
| auto 表示汽车，有39615个文本    business 企业 有11292个文本    cj 有121708个文本 财经    cul 有58619个文本，文娱    edu 有19205个文本，代表教育    fortune 25852 也是财经金融    fz 47607 法制    gj 128062 国际    gn  150657 国内    house 26141 房产    hr 27305 华人    mil 28976 军事    ny 22734 能源    sh 203571 社会    stock 34326 证券    tw 36639 台湾    ty 87245 体育    yl 70816 娱乐 |

我选择了比较有代表性的几类，如下

1. 汽车--auto   2. 企业--business   3. 财经--cj     4. 文娱--cul     5. 教育--edu     6. 法制--fz     7. 房产--house   8. 军事--mil     9. 能源--ny     10. 体育--ty

11. 娱乐--yl

其中，由于财经数量太多了，并且其中内容和其他类型的数据重合度较高，因此我将它剔除，剩下最后10类，共561724，平均一类有五万多篇。

在本次实验中，为了能够更准确的划分每一类型的数据，需要用到的词性是名词，且不为人名。

分词的步骤如下：

1. 读取文件到列表中

2. 对于每一个文件，使用cut函数进行分词

3. 给分词后的每个词后加空格

4. 判断是否是名词且不是人名，且不在停用词集合中

5.组合成一个字符串，写入到新的文件中

在分词中，为了提高效率，我给每一类别的数据都开了一个线程，最后还是用了几十分钟才分词完毕，程序示意图如下：

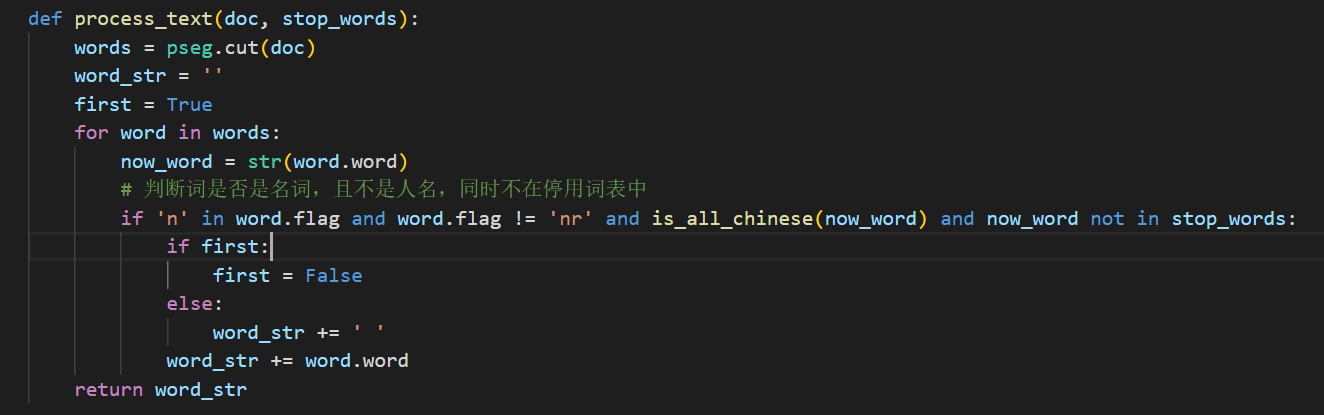


图5-10分词核心函数处理示意图



图5-11 开启多个线程分词处理示意图



图5-12 每一类别的分词处理示意图

### 5.2.2 提取特征数据

由于每篇文章的词数区别很大，在后面提取TF-IDF矩阵的时候会出现很多没必要的行和列，所以我将词数限制在150到500个词。

相关的逻辑简单，直接放程序：



图5-13 提取词数相近的文本

## 5.3 构建TF-IDF矩阵空间

TF-IDF矩阵的向量通过Sklearn中的TfidfVectorizer构建。构造器的参数如下：

max\_features: 这个参数指定了构建的词汇表（vocabulary）中最大特征的数量。如果指定为一个整数，则选择出现频率最高的前 N 个特征；如果是一个浮点数，则表示选择前面的百分之多少的特征。

sublinear\_tf: 这个参数控制 TF（Term Frequency）的缩放。如果设置为 True，则使用 sublinear scaling，即用 1 + log(TF) 替代原始的 TF。

max\_df: 这个参数是一个词频的阈值。如果一个词语在文档中的出现频率超过了设定的阈值，那么它就会被忽略。可以是一个整数（表示词频），也可以是一个浮点数（表示词频占比）。

min\_dif: 这个参数是一个词频的下限阈值。如果一个词语在文档中的出现频率低于设定的阈值，那么它就会被忽略。

在构造矩阵空间之前，我先用Bunch将各个文档给封装起来，包括每个文档的词语列表、文档的类别、文档的名称，这样可以更好的进行训练。同时，我还将训练集和测试集划分开来。训练集为每类8000个文档，剩下的作为测试集。

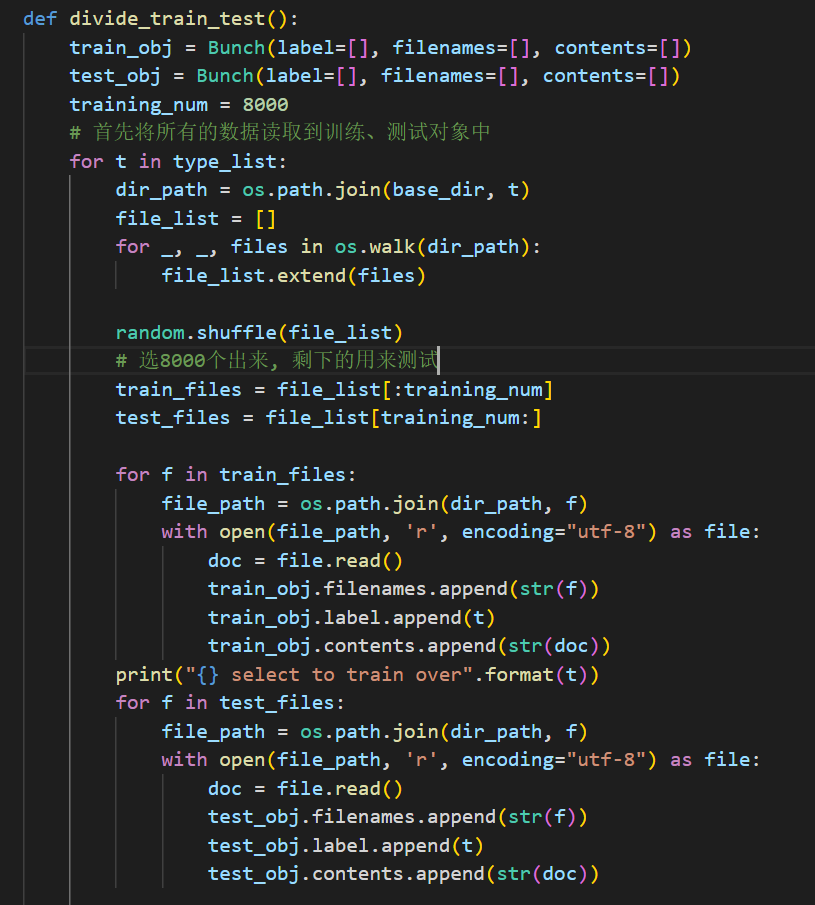


图5-14 封装文档，并划分训练集

在这之后，我将TF-IDF矩阵空间生成出来，只要用到的是TfidfVectorizer中的fit\_transform函数。该函数用于将原始文本数据转换为 TF-IDF矩阵。

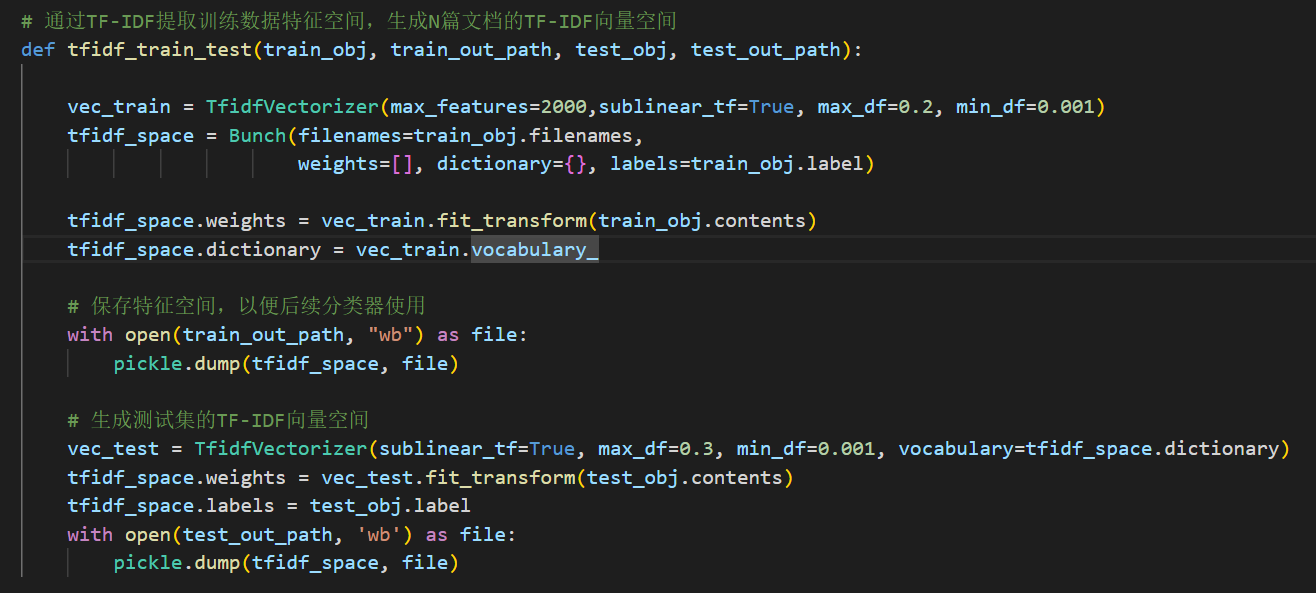


图5-15 生成训练集和测试集的矩阵空间

要注意的是，要将测试集的词典和训练集的词典对应上，才能更好的进行预测，否则之后的预测只是对于测试集上做聚类而已。所以只需要将测试集的矩阵空间中的文档以及标签更改了就行。

## 5.4 LDA 主题模型构建

通过LatentDirichletAllocation来构造lda模型，通过这个类里的fit\_transform来将tfidf矩阵空间中的文档转换成最终词语。程序如下：

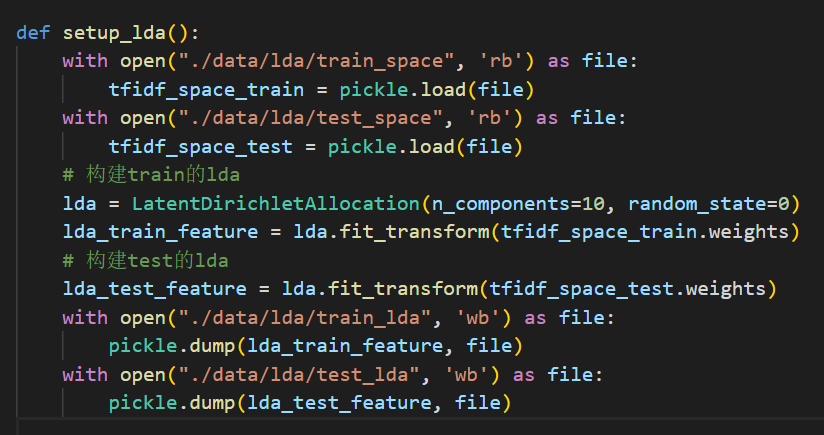


图5-16 构造lda主题模型

其中，LatentDirichletAllocation构造器中，n\_components代表待分类的类别数，按照实验要求是10，random\_state是初始的状态。

fit\_transform过程较长，需要耐心等待。

## 5.5 SVM

首先要将上一步TF-IDF的矩阵空间和LDA主题模型导入。

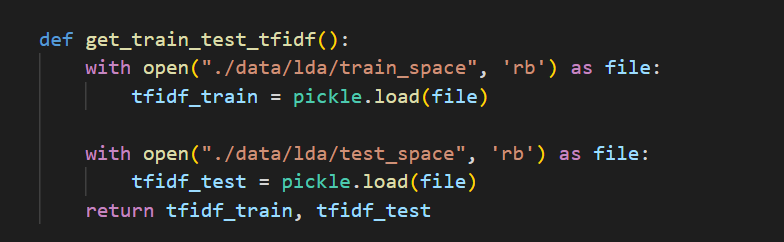


图5-17 读取TF-IDF矩阵空间

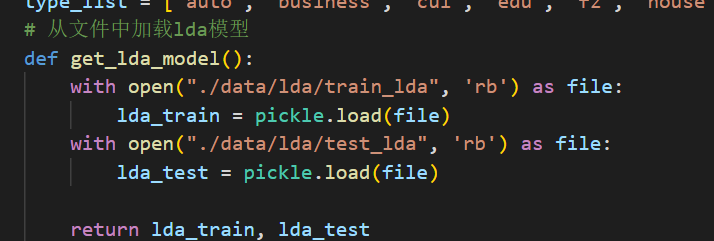


图5-18 读取LDA主题模型

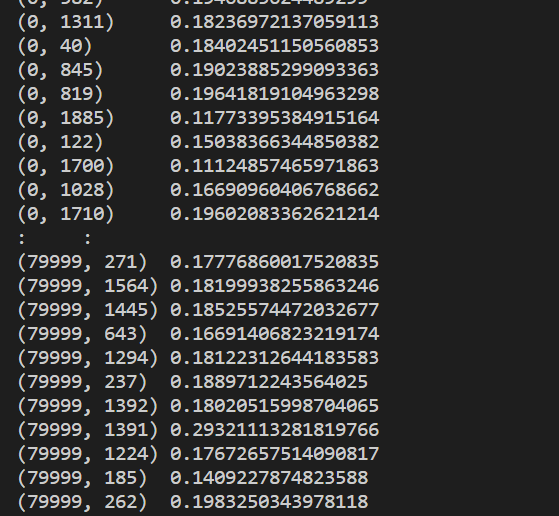


图5-19 训练集权重

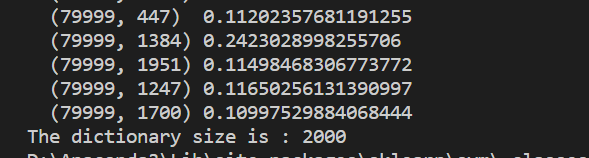


图5-20 字典大小示意图

之后就可以对训练集训练，对测试集预测分析

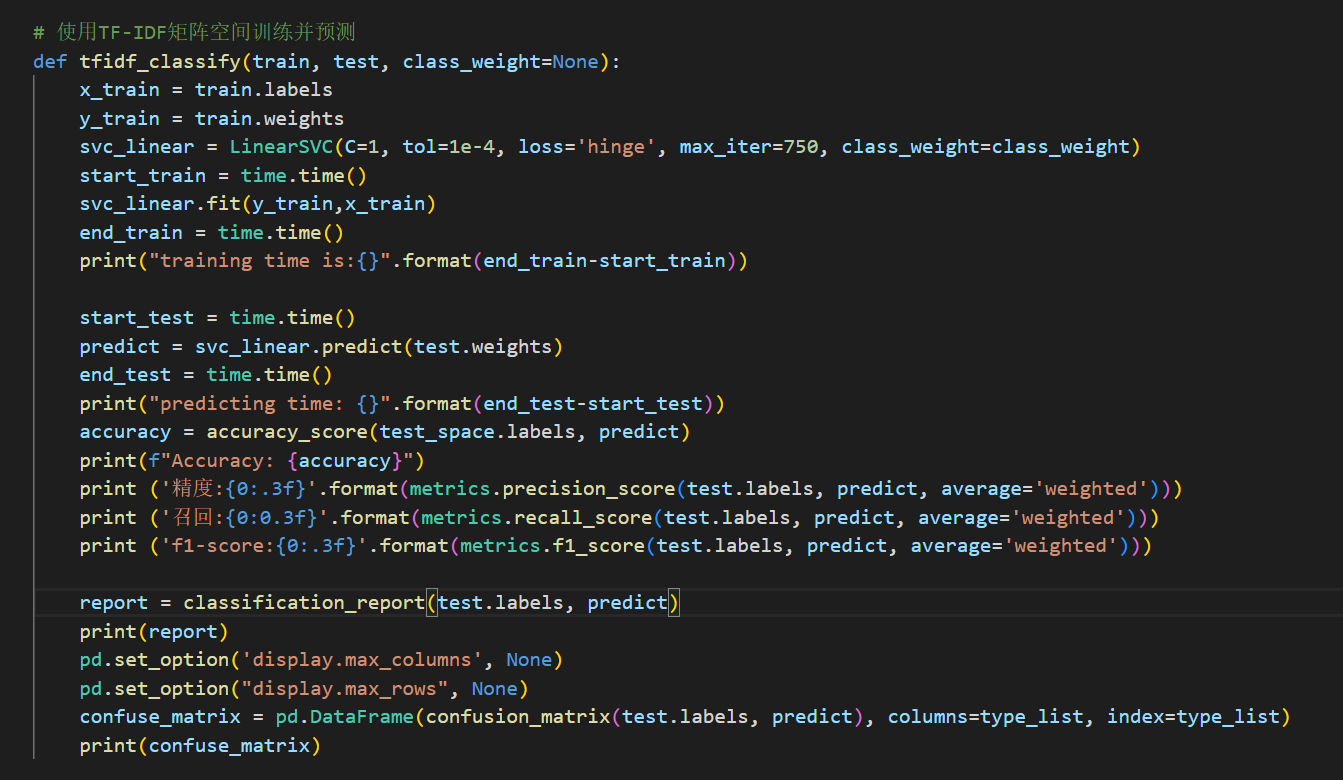


图5-21 使用TF-IDF 矩阵空间训练并预测

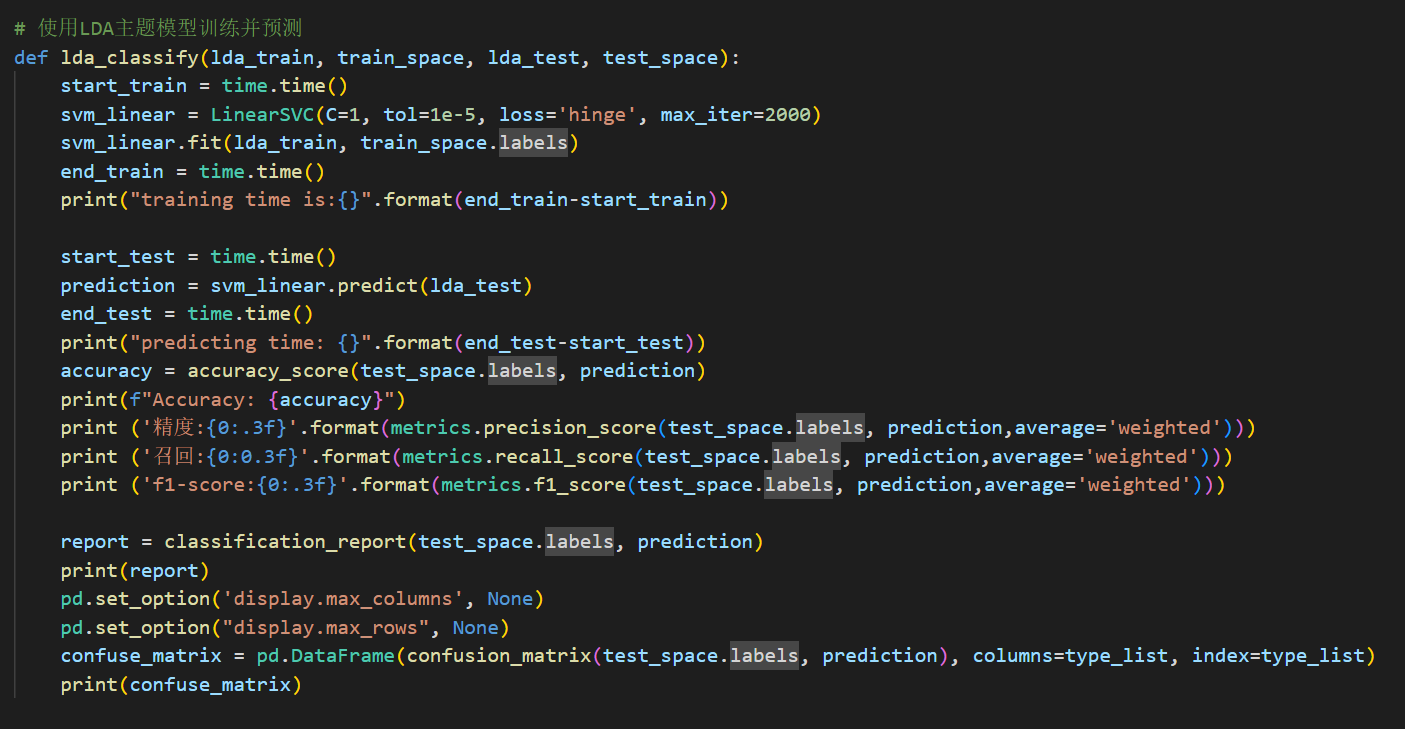


图5-22 使用LDA主题模型训练并预测

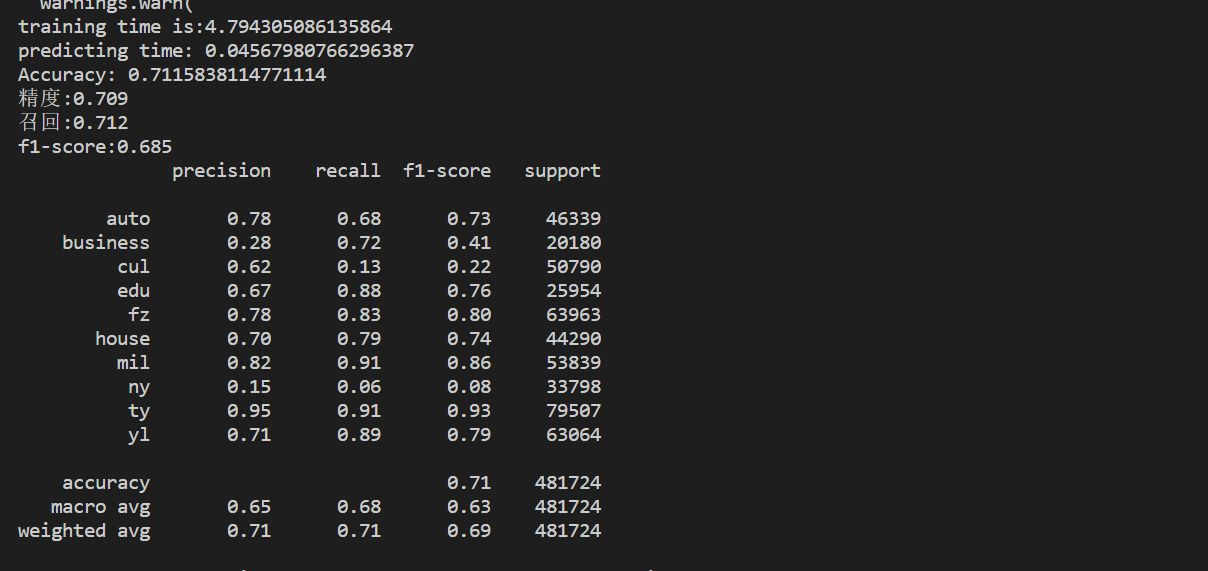


图5-23 使用LDA主题模型预测

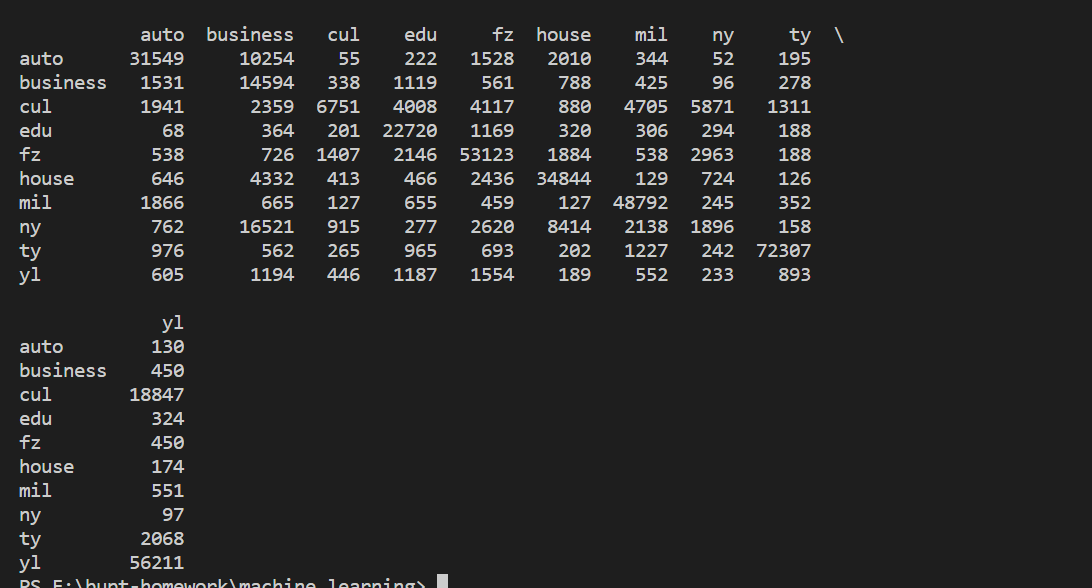


图5-24 使用LDA主题模型的预测混淆矩阵

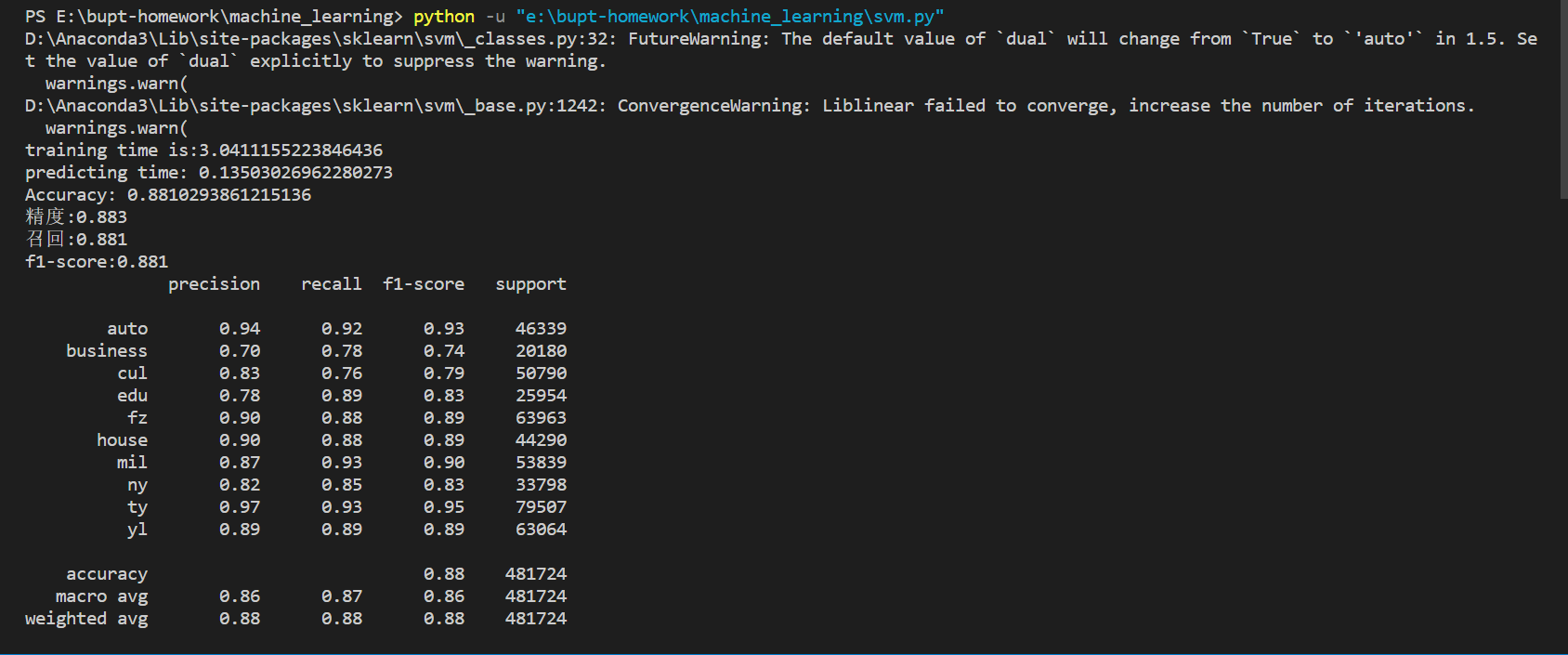


图5-25 使用TF-IDF训练分析

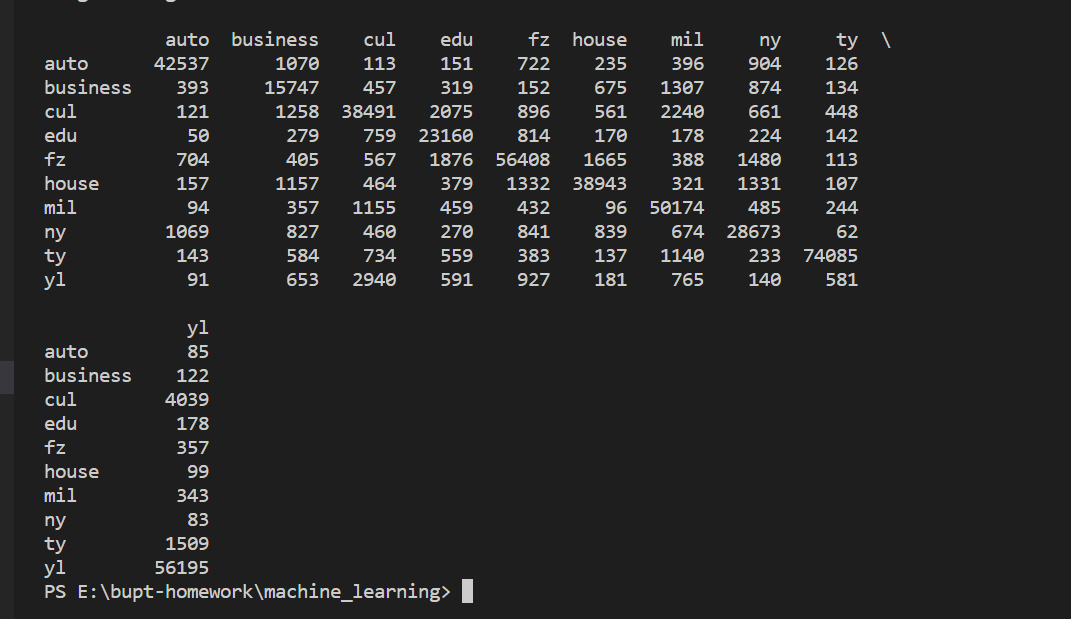


图5-26 使用TF-IDF矩阵空间预测的混淆矩阵

可以看到，单次预测的平均准确度有88.3%。单类最高准确率有97%。错误主要是出在business类上。在class\_weight上，把business的权重调低，如下图所示：

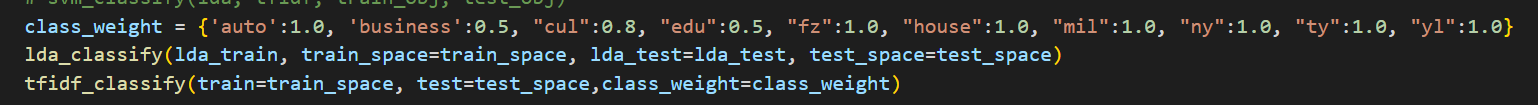


图5-27 参数修改

传入class\_weight参数，再次执行训练+分析，结果如下：

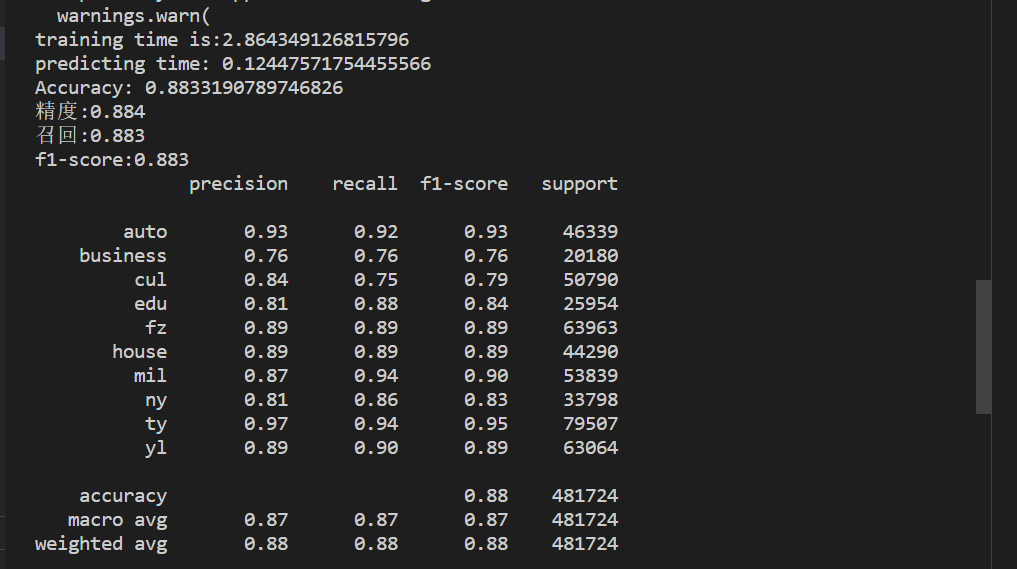


图5-29 调参后的执行结果

可以看到，business类的准确率提升了6%，整体的精确度提升了0.1%。

# 6 附录

## 6.1 get\_url.py

# 这个程序是用来爬取 类似 https://www.chinanews.com/scroll-news/2019/1211/news.shtml 这个网站里的url的

import requests

from bs4 import BeautifulSoup

from urllib.parse import urljoin

def crawl\_news\_urls(base\_url, year, month, day):

    url = f"{base\_url}/{year}/{month:02d}{day:02d}/news.shtml"

    response = requests.get(url)

    soup = BeautifulSoup(response.text, 'html.parser')

    news\_urls = []

    for li in soup.select('.content\_list ul li'):

        div\_bt = li.find('div', class\_='dd\_bt')

        if div\_bt:

            link = div\_bt.a

            if link:

                news\_url = urljoin(url, link['href'])

                news\_urls.append(news\_url)

    return news\_urls

# 主程序

base\_url = "http://www.chinanews.com/scroll-news"

years = [2011,2012, 2013, 2014, 2015, 2016, 2017,2018,2019,2020,2021,2022, 2023]

months = [1, 2, 3,4,5,6,7,8,9,10,11,12]

days = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]

# 爬取新闻页面的 URL

with open('./urls.txt', 'a', encoding='utf8') as file:

    for year in years:

        for month in months:

            for day in days:

                news\_urls = crawl\_news\_urls(base\_url, year, month, day)

                print(news\_urls)

                for news\_url in news\_urls:

                    file.write(news\_url + '\n')

## 6.2get\_news\_treads.py

from urllib.parse import urlsplit

import requests

from bs4 import BeautifulSoup

import os

import threading

type\_list = ["fz", "business", "edu", "mil", "ny", "house"]

def fetch\_and\_save(urls, count):

    try:

        for url in urls:

            if not url.startswith("http://www.chinanews.com/"):

                continue

            parsed\_url = urlsplit(url)

            # base\_url = f"{parsed\_url.scheme}://{parsed\_url.netloc}"

            path\_parts = parsed\_url.path.split('/')

            year = path\_parts[2]

            if not year.isdigit() :

                continue

            type = path\_parts[1]

            count += 1

            if type not in type\_list:

                continue

            filepath = f'./data/news/{type}/'

            if not os.path.exists(filepath):

                os.makedirs(filepath)

            filename = f'{filepath}{str(count)}.txt'

            if os.path.exists(filename):

                continue

            # 开始获取文本信息

            try:

                response = requests.get(url, timeout=5)

            except requests.Timeout:

                print("{} timeout".format(count))

                continue

            except requests.exceptions.RequestException as e:

                print(f"请求发生错误: {e}")

            response.encoding = 'gb2312' if int(year) < 2019  else 'utf-8'

            soup = BeautifulSoup(response.text, 'html.parser')

            # 查找 left\_zw 类（用来保存新闻内容的）

            left\_zw = soup.find('div', class\_='left\_zw')

            if left\_zw:

                # 提取文本内容

                news\_contents = ''

                paragraphs = left\_zw.find\_all('p')

                for paragraph in paragraphs:

                    news\_contents += paragraph.text.strip()

                # 保存到文件

                with open(filename, 'w', encoding='utf-8', errors='ignore') as f:

                    f.write(news\_contents)

                print("完成：{} 的读取".format(count))

    except  Exception as e:

        print(f"{count} 发生错误: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

    urls = open('./data/urls/urls.txt').read().split('\n')

    base\_count = 36240

    urls = urls[base\_count:]

    chunk\_size = 10000

    finished\_num = 5000

    threads = []

    try:

        for i in range(base\_count, len(urls), chunk\_size):

            base = i + finished\_num

            thread = threading.Thread(target=fetch\_and\_save, args=(urls[base:i+chunk\_size], base))

            threads.append(thread)

            thread.start()

            print("开启线程: {}".format(i))

        # 等待所有线程完成

        for thread in threads:

            thread.join()

    except Exception as e:

        print("出现问题:{}".format(e))

### 6.3 splice\_word.py

import os

import jieba.posseg as pseg

import threading

def is\_all\_chinese(strs):

    for \_char in strs:

        if not ('\u4e00' <= \_char <= '\u9fa5'):

            return False

    return True

def process\_text(doc, stop\_words):

    words = pseg.cut(doc)

    word\_str = ''

    first = True

    for word in words:

        now\_word = str(word.word)

        # 判断词是否是名词，且不是人名，同时不在停用词表中

        if 'n' in word.flag and word.flag != 'nr' and is\_all\_chinese(now\_word) and now\_word not in stop\_words:

            if first:

                first = False

            else:

                word\_str += ' '

            word\_str += word.word

    return word\_str

def process\_category(base\_dir, category, stop\_words):

    dir\_path = os.path.join(base\_dir, category)

    count = 0

    out\_dir = "./data/splice/" + category

    os.makedirs(out\_dir, exist\_ok=True)  # 确保文件夹存在，如果不存在则创建

    for \_, \_, files in os.walk(dir\_path):

        for f in files:

            out\_path = os.path.join('./data/splice/' + category, f)

            if os.path.exists(out\_path):

                print(f"文件 {out\_path} 已存在，跳过处理。")

                continue

            file\_path = os.path.join(dir\_path, f)

            with open(file\_path, 'rb') as file\_obj:

                doc = file\_obj.read()

                count += 1

            word\_str = process\_text(doc, stop\_words)

            with open(out\_path, 'w', encoding='utf-8') as out:

                out.write(word\_str)

            if(count % 100 == 0):

                print("{} : {}".format(category, count))

    print(count)

if \_\_name\_\_ == "\_\_main\_\_":

    type\_list = ["auto", "business", "cj", "cul", "edu", "fz", "house", "mil", "ny", "ty", "yl"]

    stop\_word\_path = "./data/stop/stop\_words\_ch.txt"

    # 读取停用词表

    with open(stop\_word\_path, 'r', encoding='gbk') as stop\_word\_file:

        stop\_words = set(stop\_word\_file.read().split('\n'))

    base\_dir = './data/news/'

    threads = []

    try:

        for category in type\_list:

            thread = threading.Thread(target=process\_category, args=[base\_dir, category, stop\_words])

            threads.append(thread)

            thread.start()

        for thread in threads:

            thread.join()

    except Exception as e:

        print("出现问题:{}".format(e))

## 6.4 select\_data.py

import os

type\_list = ["auto", "business", "cul", "edu", "fz", "house", "mil", "ny", "ty", "yl"]

data\_dir = "./data/splice"

def count\_words(text):

    words = text.split()

    return len(words)

def select\_data():

    for t in type\_list:

        dir\_path = os.path.join(data\_dir, t)

        print(t)

        count = 0

        os.makedirs("./data/select/"+t, exist\_ok=True)  # 确保文件夹存在，如果不存在则创建

        for root, \_, files in os.walk(dir\_path):

            for f in files:

                file\_path = os.path.join(root, f)

                with open(file\_path, 'r', encoding='utf-8') as file:

                    content = file.read()

                word\_count = count\_words(content)

                if word\_count >= 150 and word\_count <=500:

                    count += 1

                    out\_path = os.path.join('./data/select/' + t, f)

                    with open(out\_path, 'w', encoding='utf-8') as out:

                        out.write(content)

        print(count)

if \_\_name\_\_ == "\_\_main\_\_":

    select\_data()

## 6.5 tfidf.py

import os

from sklearn.decomposition import LatentDirichletAllocation

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.utils import Bunch

import numpy as np

import pickle

import random

# 文件夹路径

base\_dir = './data/splice/'

type\_list = ["auto", "business", "cul", "edu", "fz", "house", "mil", "ny", "ty", "yl"]

# document\_counts = [54339,28180,58790,33954,71963,52290,61839,41798,87507,71064]

total\_doc = 561890

train\_out\_path = "./data/trainset/train"

test\_out\_path = "./data/testset/test"

def divide\_train\_test():

    train\_obj = Bunch(label=[], filenames=[], contents=[])

    test\_obj = Bunch(label=[], filenames=[], contents=[])

    training\_num = 8000

    # 首先将所有的数据读取到训练、测试对象中

    for t in type\_list:

        dir\_path = os.path.join(base\_dir, t)

        file\_list = []

        for \_, \_, files in os.walk(dir\_path):

            file\_list.extend(files)

        random.shuffle(file\_list)

        # 选8000个出来, 剩下的用来测试

        train\_files = file\_list[:training\_num]

        test\_files = file\_list[training\_num:]

        for f in train\_files:

            file\_path = os.path.join(dir\_path, f)

            with open(file\_path, 'r', encoding="utf-8") as file:

                doc = file.read()

                train\_obj.filenames.append(str(f))

                train\_obj.label.append(t)

                train\_obj.contents.append(str(doc))

        print("{} select to train over".format(t))

        for f in test\_files:

            file\_path = os.path.join(dir\_path, f)

            with open(file\_path, 'r', encoding="utf-8") as file:

                doc = file.read()

                test\_obj.filenames.append(str(f))

                test\_obj.label.append(t)

                test\_obj.contents.append(str(doc))

        print("{} select to test over".format(t))

    # 将二进制数据写入文件

    with open(train\_out\_path, 'wb') as f1:

        pickle.dump(train\_obj, f1)

    with open(test\_out\_path, "wb") as f2:

        pickle.dump(test\_obj, f2)

# 通过TF-IDF提取训练数据特征空间，生成N篇文档的TF-IDF向量空间

def tfidf\_train\_test(train\_obj, train\_out\_path, test\_obj, test\_out\_path):

    vec\_train = TfidfVectorizer(max\_features=2000,sublinear\_tf=True, max\_df=0.2, min\_df=0.001)

    tfidf\_space = Bunch(filenames=train\_obj.filenames,

                        weights=[], dictionary={}, labels=train\_obj.label)

    tfidf\_space.weights = vec\_train.fit\_transform(train\_obj.contents)

    tfidf\_space.dictionary = vec\_train.vocabulary\_

    # 保存特征空间，以便后续分类器使用

    with open(train\_out\_path, "wb") as file:

        pickle.dump(tfidf\_space, file)

    # 生成测试集的TF-IDF向量空间

    vec\_test = TfidfVectorizer(sublinear\_tf=True, max\_df=0.3, min\_df=0.001, vocabulary=tfidf\_space.dictionary)

    tfidf\_space.weights = vec\_test.fit\_transform(test\_obj.contents)

    tfidf\_space.labels = test\_obj.label

    with open(test\_out\_path, 'wb') as file:

        pickle.dump(tfidf\_space, file)

def setup\_lda():

    with open("./data/lda/train\_space", 'rb') as file:

        tfidf\_space\_train = pickle.load(file)

    with open("./data/lda/test\_space", 'rb') as file:

        tfidf\_space\_test = pickle.load(file)

    # 构建train的lda

    lda = LatentDirichletAllocation(n\_components=10, random\_state=0)

    lda\_train\_feature = lda.fit\_transform(tfidf\_space\_train.weights)

    # 构建test的lda

    lda\_test\_feature = lda.fit\_transform(tfidf\_space\_test.weights)

    with open("./data/lda/train\_lda", 'wb') as file:

        pickle.dump(lda\_train\_feature, file)

    with open("./data/lda/test\_lda", 'wb') as file:

        pickle.dump(lda\_test\_feature, file)

# 读取处理好的训练集和测试集

def get\_train\_test():

    with open(train\_out\_path, 'rb') as train:

        train\_obj = pickle.load(train)

    with open(test\_out\_path, 'rb') as test:

        test\_obj = pickle.load(test)

    return train\_obj, test\_obj

if \_\_name\_\_ == "\_\_main\_\_":

    # divide\_train\_test()

    # train, test = get\_train\_test()

    print("get data over")

    # tfidf\_train\_test(train\_obj=train, train\_out\_path="./data/lda/train\_space", test\_obj=test, test\_out\_path="./data/lda/test\_space")

    setup\_lda()

# 失败的加权模型

# 将训练集转换为TF-IDF矩阵, 为矩阵加权，喂给LDA主题模型,并保存

# def lda\_fit(train\_obj):

#     # 计算文档频率（DF）

#     vec\_train = TfidfVectorizer(max\_features=2000, max\_df=0.2, token\_pattern=r'\b\w+\b')

#     df\_matrix = vec\_train.fit\_transform(train\_obj.contents)

#     print(df\_matrix.shape)

#     # doc\_freqs = df\_matrix.transform(train\_obj.contents)

#     weight\_array = np.array(train\_obj.weights)

#     print(weight\_array.shape)

#     # 将 TF-IDF 矩阵按照样本权重进行加权

#     weighted\_tfidf\_matrix = df\_matrix.multiply(1/np.sqrt(weight\_array[:, np.newaxis]))

#     # 构建 LDA 模型并在 TF-IDF 矩阵上训练

#     lda = LatentDirichletAllocation(n\_components=11, random\_state=42)

#     lda.fit(weighted\_tfidf\_matrix)

#     with open("./data/lda/lda\_model", 'wb') as lda\_file:

#         pickle.dump(lda, lda\_file)

#     with open("./data/lda/tfidf", "wb") as tfidf\_file:

#         pickle.dump(vec\_train, tfidf\_file)

# 本来想直接读取，然后加权得到矩阵，但是内存分配不够

# # 存放所有文档的列表

# documents = []

# weights = []

# # 添加每个已分词文件到列表中

# for t in type\_list:

#     dir\_path = os.path.join(base\_dir, t)

#     count = 0

#     for root, dirs, files in os.walk(dir\_path):

#         for f in files:

#             path = os.path.join(root, f)

#             with open(path, 'r', encoding='utf-8') as file:

#                 # 读取文件内容

#                 documents.append(file.read())

#             weights.append(total\_doc / len(files))

#             count += 1

#             if count % 10000 == 0 :

#                 print(count)

#             flag = not flag

# weights\_array = np.array(weights)

# # 将文档列表转换为TF-IDF矩阵

# # max\_df 表示如果一个单词在百分之max\_df的文档中出现，就要省略

# vec\_train = TfidfVectorizer(max\_features=500, max\_df=0.3, token\_pattern=r'\b\w+\b')

# tfidf\_matrix = vec\_train.fit\_transform(documents)

# print(len(weights))

# print(tfidf\_matrix.shape[0])

# weighted\_tfidf\_matrix = tfidf\_matrix.multiply(np.repeat(weights, tfidf\_matrix.shape[0]))

# # 构建LDA模型

# lda = LatentDirichletAllocation(n\_components=11, random\_state=42)

# # 在TF-IDF矩阵上训练LDA模型

# lda.fit(weighted\_tfidf\_matrix)

# # 打印每个主题的前几个关键词

# feature\_names = vec\_train.get\_feature\_names\_out()

# n\_top\_words = 10

# for topic\_idx, topic in enumerate(lda.components\_):

#     top\_words\_idx = topic.argsort()[:-n\_top\_words - 1:-1]

#     top\_words = [feature\_names[i] for i in top\_words\_idx]

#     print(f"Topic #{topic\_idx}: {', '.join(top\_words)}")

## 6.6 svm.py

import os

from sklearn.svm import LinearSVC

from sklearn.decomposition import LatentDirichletAllocation

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.utils import Bunch

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import numpy as np

import pickle

import time

from sklearn import metrics

import pandas as pd

type\_list = ["auto", "business", "cul", "edu", "fz", "house", "mil", "ny", "ty", "yl"]

# 从文件中加载lda模型

def get\_lda\_model():

    with open("./data/lda/train\_lda", 'rb') as file:

        lda\_train = pickle.load(file)

    with open("./data/lda/test\_lda", 'rb') as file:

        lda\_test = pickle.load(file)

    return lda\_train, lda\_test

def get\_train\_test\_tfidf():

    with open("./data/lda/train\_space", 'rb') as file:

        tfidf\_train = pickle.load(file)

    with open("./data/lda/test\_space", 'rb') as file:

        tfidf\_test = pickle.load(file)

    return tfidf\_train, tfidf\_test

# 读取处理好的训练集和测试集

def get\_train\_test():

    with open("./data/trainset/train\_data", 'rb') as train:

        train\_obj = pickle.load(train)

    with open("./data/testset/test\_data", 'rb') as test:

        test\_obj = pickle.load(test)

    return train\_obj, test\_obj

# 使用LDA主题模型训练并预测

def lda\_classify(lda\_train, train\_space, lda\_test, test\_space):

    start\_train = time.time()

    svm\_linear = LinearSVC(C=1, tol=1e-5, loss='hinge', max\_iter=2000)

    svm\_linear.fit(lda\_train, train\_space.labels)

    end\_train = time.time()

    print("training time is:{}".format(end\_train-start\_train))

    start\_test = time.time()

    prediction = svm\_linear.predict(lda\_test)

    end\_test = time.time()

    print("predicting time: {}".format(end\_test-start\_test))

    accuracy = accuracy\_score(test\_space.labels, prediction)

    print(f"Accuracy: {accuracy}")

    print ('精度:{0:.3f}'.format(metrics.precision\_score(test\_space.labels, prediction,average='weighted')))

    print ('召回:{0:0.3f}'.format(metrics.recall\_score(test\_space.labels, prediction,average='weighted')))

    print ('f1-score:{0:.3f}'.format(metrics.f1\_score(test\_space.labels, prediction,average='weighted')))

    report = classification\_report(test\_space.labels, prediction)

    print(report)

    pd.set\_option('display.max\_columns', None)

    pd.set\_option("display.max\_rows", None)

    confuse\_matrix = pd.DataFrame(confusion\_matrix(test\_space.labels, prediction), columns=type\_list, index=type\_list)

    print(confuse\_matrix)

# 使用TF-IDF矩阵空间训练并预测

def tfidf\_classify(train, test, class\_weight=None):

    x\_train = train.labels

    y\_train = train.weights

    svc\_linear = LinearSVC(C=1, tol=1e-4, loss='hinge', max\_iter=750, class\_weight=class\_weight)

    start\_train = time.time()

    svc\_linear.fit(y\_train,x\_train)

    end\_train = time.time()

    print("training time is:{}".format(end\_train-start\_train))

    start\_test = time.time()

    predict = svc\_linear.predict(test.weights)

    end\_test = time.time()

    print("predicting time: {}".format(end\_test-start\_test))

    accuracy = accuracy\_score(test\_space.labels, predict)

    print(f"Accuracy: {accuracy}")

    print ('精度:{0:.3f}'.format(metrics.precision\_score(test.labels, predict, average='weighted')))

    print ('召回:{0:0.3f}'.format(metrics.recall\_score(test.labels, predict, average='weighted')))

    print ('f1-score:{0:.3f}'.format(metrics.f1\_score(test.labels, predict, average='weighted')))

    report = classification\_report(test.labels, predict)

    print(report)

    pd.set\_option('display.max\_columns', None)

    pd.set\_option("display.max\_rows", None)

    confuse\_matrix = pd.DataFrame(confusion\_matrix(test.labels, predict), columns=type\_list, index=type\_list)

    print(confuse\_matrix)

if \_\_name\_\_ == "\_\_main\_\_":

    lda\_train, lda\_test = get\_lda\_model()

    train\_space, test\_space = get\_train\_test\_tfidf()

    # train\_obj, test\_obj = get\_train\_test()

    # svm\_classify(lda, tfidf, train\_obj, test\_obj)

    class\_weight = {'auto':1.0, 'business':1.0, "cul":1.0, "edu":1.0, "fz":1.0, "house":1.0, "mil":1.0, "ny":1.0, "ty":1.0, "yl":1.0}

    lda\_classify(lda\_train, train\_space=train\_space, lda\_test=lda\_test, test\_space=test\_space)

    # tfidf\_classify(train=train\_space, test=test\_space,class\_weight=class\_weight)

# def svm\_classify(lda, tfidf, train, test, loss="hinge", weigh\_param=None):

#     start\_transform = time.time()

#     lda\_train\_feature = lda.transform(train.weights)

#     lda\_test\_feature = lda.transform(test.weights)

#     end\_transform = time.time()

#     print(f"Time for transformation: {end\_transform - start\_transform} seconds")

#     # 构造分类器

#     svm\_linear = LinearSVC(class\_weight=weigh\_param, loss=loss)

#     # 训练SVM

#     start\_training = time.time()

#     # dense\_train = lda\_train\_feature.toarray()

#     svm\_linear.fit(lda\_train\_feature, train.labels)

#     end\_training = time.time()

#     print(f"Time for training: {end\_training - start\_training} seconds")

#     # 预测

#     start\_prediction = time.time()

#     predictions = svm\_linear.predict(lda\_test\_feature)

#     end\_prediction = time.time()

#     print(f"Time for prediction: {end\_prediction - start\_prediction} seconds")

#     # 评估

#     accuracy = accuracy\_score(test.labels, predictions)

#     report = classification\_report(test.labels, predictions)

#     print(f"Accuracy: {accuracy}")

#     print("Classification Report:")

#     print(report)

# def svm\_classify(lda, tfidf, train\_obj, test\_obj):

#     start\_transform = time.time()

#     lda\_train\_feature = lda.transform(tfidf.transform(train\_obj.contents))

#     lda\_test\_feature = lda.transform(tfidf.transform(test\_obj.contents))

#     end\_transform = time.time()

#     print("transform time is : {}".format(end\_transform-start\_transform))

#     # 构建SVM分类器

#     linear\_svm = SVC(kernel='linear')

#     start\_train = time.time()

#     linear\_svm.fit(lda\_train\_feature, train\_obj.labels)

#     end\_train = time.time()

#     print("train time is : {}".format(end\_train-start\_train))

#     # 预测

#     predict\_linear = linear\_svm.predict(lda\_test\_feature)

#     # 评估

#     accuracy\_linear = accuracy\_score(test\_obj.labels, predict\_linear)

#     report\_linear = classification\_report(test\_obj.labels, predict\_linear)

#     print(f"Accuracy: {accuracy\_linear}")

#     print("Classification Report:")

#     print(report\_linear)

#     print("predict time is {}".format(time.time() - end\_train))

    # poly\_svm = SVC(kernel="poly")

    # poly\_svm.fit(lda\_test\_feature, test\_obj.labels)

    # # 预测

    # predict\_poly = poly\_svm.predict(lda\_test\_feature)

    # # 评估

    # accuracy\_poly = accuracy\_score(test\_obj.labels, predict\_poly)

    # report\_poly = classification\_report(test\_obj.labels, predict\_poly)

    # print(f"Accuracy: {accuracy\_poly}")

    # print("Classification Report:")

    # print(report\_poly)