《作业2 信息检索系统》  
实验报告

作业2 信息检索系统

1. 概述

本项目针对法律文书数据集（LeCaRd）的8881个文档，通过分词后统计词频，构建TF-IDF倒排索引，并获得词-文档向量矩阵。查询时，将检索分词后构建向量，通过余弦距离算法，计算查询和文档之间的距离来获得相关度。我们的实现在语料上达到了90%左右的平均准确率（前5/10/20个结果）。为了减少词向量，降低内存占用，我们针对词-文档矩阵使用SVD（奇异值分解）进行降维，将词向量减少到1500个，同时仍然仍获得很好的匹配效果。我们进一步调整K的值，增大K到2500的时候，准确率甚至略好于没有降维之前的结果。

* 1. 数据集介绍

LeCaRd(https://github.com/myx666/LeCaRD)是一个中文法律案例检索数据集，包含了10700个案例。由于要验证准确率和召回率，我们使用的是LeCaRd的原始语料库（https://drive.google.com/file/d/1vQdX1MegFVtmoh0XCd4mav5PBkep7q0h/view?usp=sharing）。这个语料库包含了全部 的8万多条数据，并且有一个关键词-文档的列表。通过这个列表，后面可以校验我们的准确率。

每个文档的结构如下：

|  |  |  |
| --- | --- | --- |
| 字段 | 说明 | 样例（局部内容） |
| ajId | 案件ID | 1136ac34-cc88-431a-b469-810060c5b7f2 |
| ajName | 案件名称 | 侯甫楼、蔺春荣等票据诈骗罪一案 |
| ajjbqk | 案件基本事实 | 原判认定，被告人侯甫楼为山西前利水泥有限公司（属自然人投资或控股的有限责任公司.... |
| cpfxgc | 法院意见 | 本院认为，上诉人侯甫楼、蔺春荣、白晋生以非法占有为目的，使用伪造的银行承兑汇票骗取他人50万... |
| pjjg | 案件判决书 | 一、维持山西省临县人民法院（2013）临刑初字第210号刑事判决第一项 |
| qw | 全文 | 山西省吕梁市中级人民法院 刑事判决书 （2014）吕刑终字第184号 抗诉机关山西省临县人民..... |
| writId | 本文档的唯一ID | 0195fd9270022211aba7d58b3af392d5a35 |
| writName | 文档名称 | 侯甫楼、蔺春荣等票据诈骗罪二审刑事判决书 |
| path | 物理路径 | 1136ac34-cc88-431a-b469-810060c5b7f2/0195fd9270022211aba7d58b3af392d5a35.json |



Figure 数据概览

可以注意到，这里全文的内容和基本事实，判决书等几个字段重叠了。经过检查，我们发现全文内容会比较杂乱一些，多了“XXX人民法院“、判决书、判决年月日等信息。但是其他字段基本上都是可以在全文中找到的。因此我们在构建倒排索引的时候，只使用qw字段。

另外，原数据有80000多条字段，起初我们使用全部文档进行构建，但是构建到一半时因为内存溢出而失败。因此我们改换思路，从有分类的文档中挑选出8881条进行构建。

除了案件数据外，我们还使用了数据集提供的一个分类表作为评估的参考，以及数据集提供的停用词表和罪名术语表来提高分词的效果。分类表将在评估一小节具体叙述，停用词表和罪名表的截图如下：

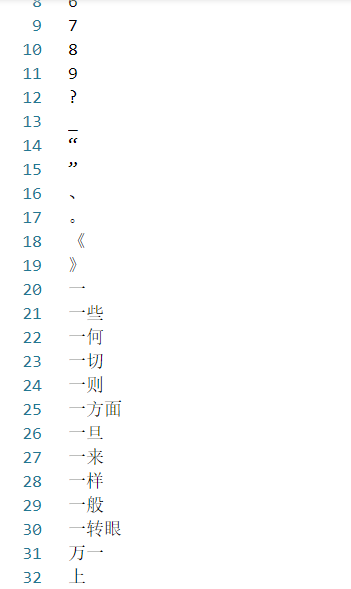
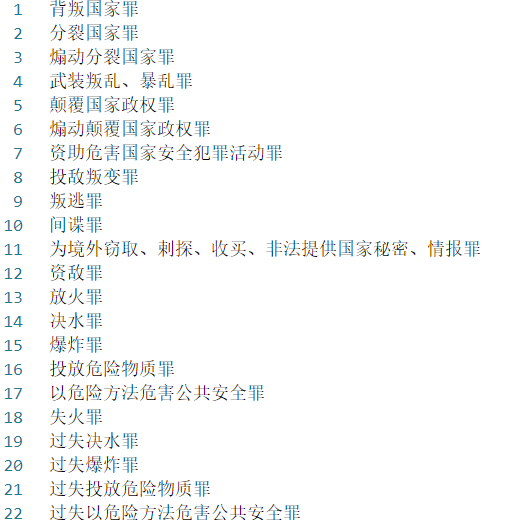
 

Figure 停用词表和罪名表

1. 系统结构和算法
   1. 倒排索引的构建

构建倒排索引首先要统计词频(TF)。我们将采用如下算法对数据进行处理：

For 每一篇文档

l : 列表 = 对文档进行分词，获得一个列表

d : 字典= 规约列表，转化为词：词频的字典形式

保存文档编号和词频

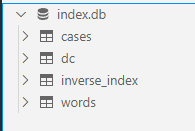
在我们实际的工作中，我们使用了多进程来提高速率。因为每一篇文档都是相对独立的，因此可以每一篇文档保存一个结果，然后把所有结果汇总起来，就可以得到一个词对文档的列表。我们一开始使用的是8个进程，但是python每次启动进程都需要很长的时间，8个进程有时候反而特别慢。最后我们只使用了4个进程。

得到TF矩阵（确切的说，词-文档向量）后，我们进一步计算IDF（逆文档频率）。IDF的公式如下：

在上面的过程中，我们已经计算出了每个词包含的文档，统计一个词文档列表的长度，再套用上面的公式就可以计算得到IDF了。最后，将TF和IDF相乘，这样就算出了每个词-文档的TF-IDF值。

我们将这个矩阵保存下来，并将上面过程中的文档数据，词数据，倒排索引全部保存下来，使用sql持久化到硬盘上。

最后我们得到的数据库如下：





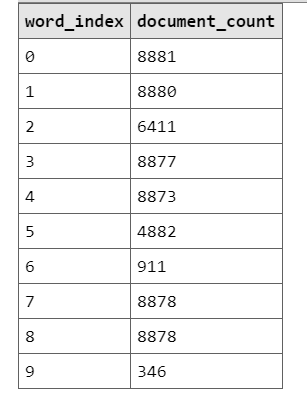
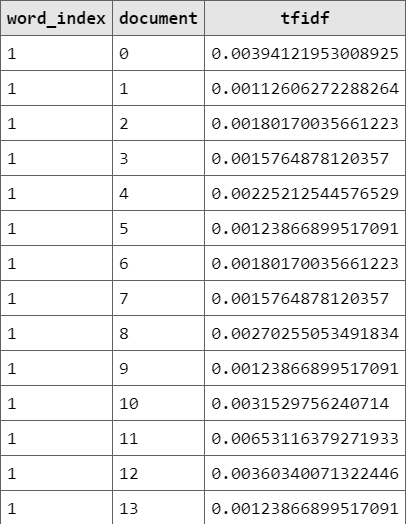


Figure 数据库内容

上述各表的记录数量如下：

|  |  |
| --- | --- |
| 表名 | 记录数 |
| cases（文档） | 8881 |
| dc（词出现文档数量统计） | 162450 |
| words（词） | 162450 |
| inverse\_index（倒排索引） | 8008865 |

可以看到，整个矩阵的填充因子仅为0.005（千分之五），是一个比较稀疏的矩阵。

数据库总占用空间为465MB。

以上具体的代码见build.py

* 1. 查询

查询前，我们首先将词表和文档频数表读入内存，方便后续操作。

查询时，我们收到一个字符串。首先将字符串做分词处理，然后统计词频，得到一个词频向量。同时，词的文档逆频率也可以很快地从之前的数据中得到，进而可以算出这个查询的TF-IDF查询向量。

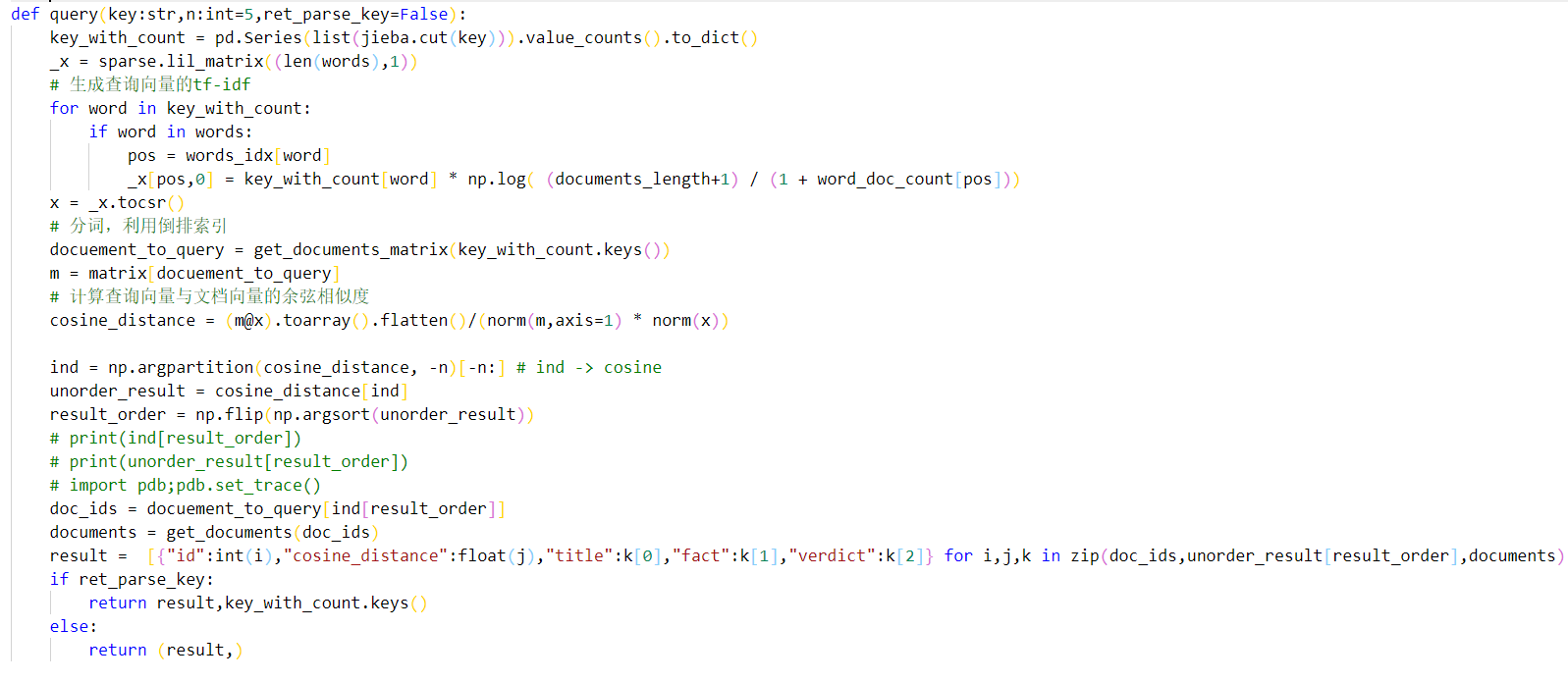
在实践中，因为查询一般不会特别长，而我们的词库往往特别大，因此词向量就会特别稀疏。这里可以考虑使用稀疏矩阵来压缩使用的空间。一个非稀疏的向量大概需要100多M的内存，而其对应的稀疏矩阵可能只需要几百KB的空间。

然后，我们通过倒排索引，获取词向量相关的词-文档子矩阵。这里同样用的是稀疏矩阵。上面词向量的优化对于较大内存的计算机可能没有必要，但是对于词-文档稀疏矩阵还是很有必要的。

记查询向量为，矩阵为，那么使用下面的式子计算余弦值：

其中表示对M转置矩阵按行求范数（即每篇文档的范数）。

然后通过排序找到余弦值最大的向量，根据向量的id来查找对应的文档，并返回给用户。



* 1. 界面和展示接口

我们使用Fastapi构建了一个简单的web api，然后通过Vue设计了一个简单的web界面来展示我们的结果。

其中，后端的Api接口如下：

/query

|  |  |  |
| --- | --- | --- |
| 参数 | 是否必需 | 说明 |
| key | 是 | 查询的字符串 |
| n | 否 | 返回文档的条数。 |

返回的数据包括分词的结果和查询到的文档。分词结果用于高亮关键词。

前端部分，我们设计了一个简单的页面。其中，搜索分词的结果是从服务端返回的，不能编辑。

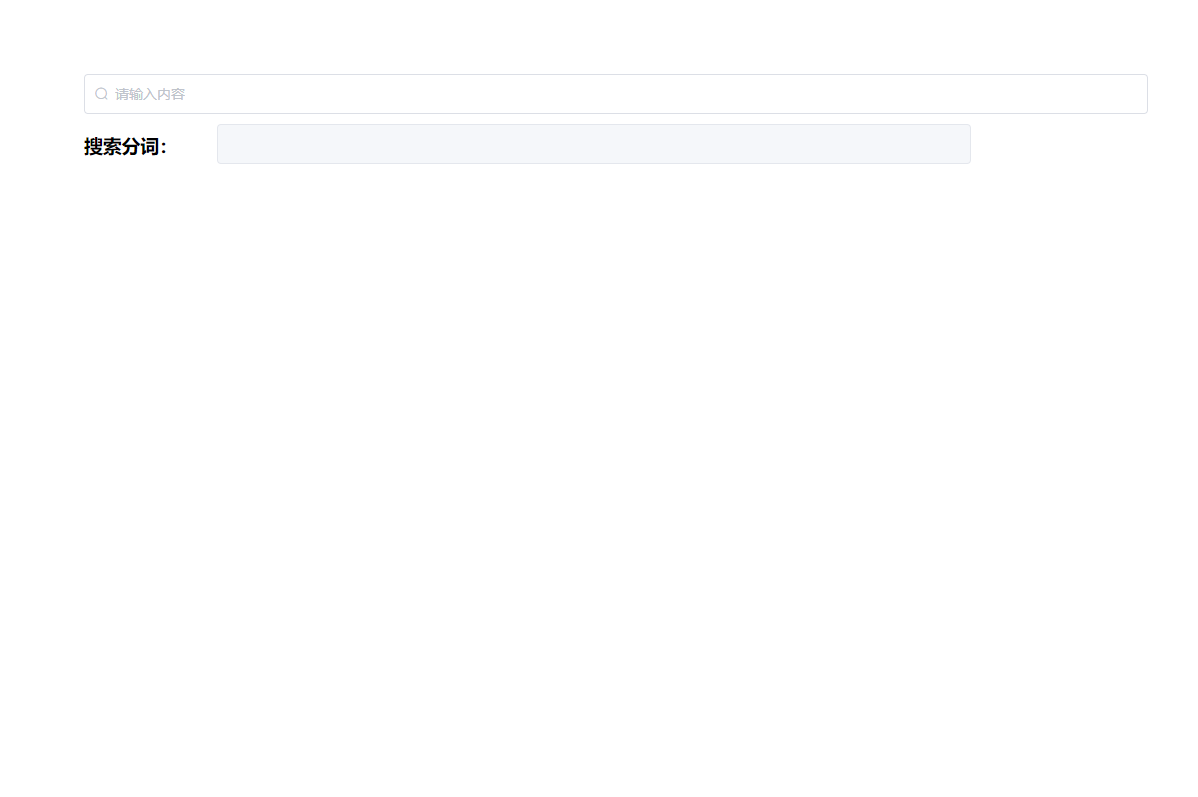


Figure 界面（默认状态）

当输入关键词后，前端通过HTTP向后端发起请求。取到数据后，检索数据，把关键词做高亮处理。



Figure 界面（搜索”盗窃罪“）

* 1. 减少词的尝试：不分词

上面的算法总体来说运行得比较好，但是生成的矩阵过大。需要想办法减小矩阵的大小。

我们一开始的想法比较直接，就是尝试不分词，直接统计每个字的词频，然后去生成倒排索引。因为汉字的总体数量是有效的，常用的汉字就只有5000-6000个，而加上不常用汉字，符号，最多也就20000-30000个词。相比于上面的16万个词，效果还是比较明显的。

于是我们用这种方案，构建了倒排索引。索引数据库的记录数如下：

|  |  |  |
| --- | --- | --- |
|  | 分词 | 不分词 |
| 表名 | 记录数 | 记录数 |
| cases（文档） | 8881 | 8881 |
| dc（词出现文档数量统计） | 162450 | 5915 |
| words（词） | 162450 | 5915 |
| inverse\_index（倒排索引） | 8008865 | 4879691 |

可以看到，词表仅有5915个词，而倒排索引数量的总量虽然减少了，但比例提高了。

整体下来，数据库大小降为338MB，算的上有所提升。

但是，进行实测检索结果的时候，其准确率却不尽如人意。前5条，10条的准确率都可以和分词持平。但是范围扩大到50条，100条的时候，准确率下降到只有50以下了。这可能主要是因为不分词带来的噪音太大了。不分词的倒排索引不可取。

* 1. 降维优化(潜在语义分析，LSA)

于是我们有了另一种想法：去把TF-IDF矩阵降维。常见的降维算法就是PCA和SVD。我们在收集资料的过程中，发现这种思路实际上很早就有人提出来了。给词频矩阵降维是潜在语义分析(LSA，Latent semantic analysis)的一种思路。具体来说，LSA做了一个比较经典的假设：出现在相同语段里面的词语具有相似的语义。因此，如果我们把出现在相同上下文的词语合并在一起，既能提高搜索的泛化程度，也能降低整个矩阵的大小，起到了一石二鸟的效果。

而进行LSA的数据工具就是截断奇异值分解（Truncated SVD）。奇异值分解能够把矩阵分解成

其中σ是特征值，也可以看做是一个加权。将特征值从大到小依次排列，选取前k个特征值重新计算矩阵，这样就可以把矩阵从m\*n降阶到k\*n了。

对于本次实验，我们需要降阶的主要是词汇的维度。经过实验，设置k为1500是一个比较合理的值。K=1500的时候，保留了88.83%的方差，训练时间差不多为3分钟。

我们使用python库 sklearn根据当前矩阵训练出一个Truncated SVD的模型，然后将矩阵输入进去，就可以得到一个被压缩的矩阵了。具体训练的代码如下：

from sklearn.decomposition import TruncatedSVD

from scipy import sparse

import numpy as np

svd = TruncatedSVD(n\_components=1500, n\_iter=7, random\_state=42)

svd.n\_jobs=-1

matrix = sparse.load\_npz("./document\_matrix.npz")

matrix = matrix.T

svd.fit(matrix)

import pickle

pickle.dump(svd,open("./lsa.pkl","wb"))

matrix = svd.transform(matrix)

np.save("./compress\_matrix",matrix)

查询时的向量也需要进行SVD处理后才能与新的压缩矩阵进行相似度距离的计算。

x = svd.transform(x.T).T

压缩以后，矩阵变为(8881,1500)，显著减少词表的大小，方便了在内存中的加载。当然，由于构建查询向量时仍然需要用到之前的词表，因此数据库大小和之前一样。

此外，在后面的演示中可以看到，压缩以后显著提高了余弦相似度。这可能与词表变小有关系。

具体评估的结果见第三节 结果评估

1. 结果评估

我们分别对普通版本和压缩版本进行测试。

测试的基准是数据集中对罪名的分类。其文件截图如下：



可以看到，主要是由一个罪名，加上一个列表组成的。列表里面主要是文件的路径，这里我将文件路径转为文档的id，同时只选取了在8881篇文档里的路径。测试数据中，罪名文档最多的是有760篇，最少的只有一篇。为了方便统计，我选取了其中列表长度大于100的文档，总共有65项罪名是满足这个条件的，测试，其数据密度图如下：

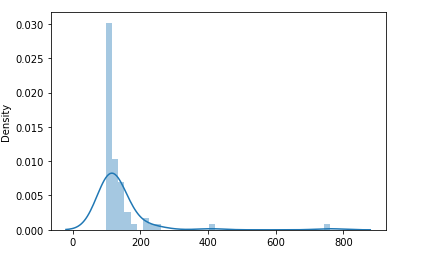


Figure 测试数据每种罪名的个数和密度

主要统计的指标是平均准确率和平均召回率。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 普通 | | SVD压缩（LSA） | |
| 前n个结果\指标 | 平均准确率 | 平均召回率 | 平均准确率 | 平均召回率 |
| 5 | 0.89846154 | 0.0323474 | 0.92 | 0.03309719 |
| 10 | 0.90615385 | 0.06545172 | 0.90307692 | 0.06524397 |
| 20 | 0.88230769 | 0.12877512 | 0.88538462 | 0.1291662 |
| 30 | 0.86512821 | 0.22332875 | 0.86358974 | 0.1914942 |
| 40 | 0.85076923 | 0.28745645 | 0.85115385 | 0.25515969 |
| 50 | 0.84123077 | 0.31999064 | 0.83692308 | 0.31887456 |
| 100 | 0.78092308 | 0.66474594 | 0.76369231 | 0.65975545 |

下面是准确率的比较：

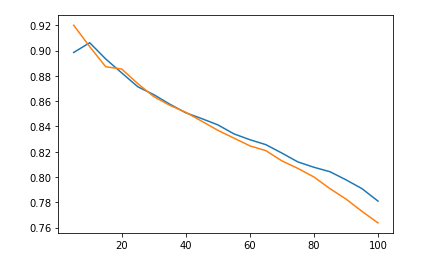


Figure 准确率图（黄色为SVD压缩，蓝色为普通）

可以看到，在搜索结果比较少的情况下，压缩后的效果要略好于普通的效果；搜索结果数变多的时候，压缩和不压缩的结果不相上下。但是，在搜索结果大于50之后，压缩后的效果明显差于不压缩的结果。

分析原因，一方面可能是我压缩的维度调整得过小；另一方面，潜在语义分析适合文本数量比较多，且覆盖范围也比较大的语料库。而我的语料库的文字量大约100万这个数量级的，只能算比较小的语料库；另外，这个语料库专门针对法律文书，用词比较单一，因此可能效果比较差。

但是，做降维压缩的目的本身就只是为了减少词的数量，降低矩阵占用的内存，从这个角度看，这个压缩做的还是比较成功的。

我们进一步调整了K的大小，调整为2500，覆盖了95%的方差。这一次结果明显比之前好了很多。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 普通 | | SVD压缩（K=2500） | |
| 前n个结果\指标 | 平均准确率 | 平均召回率 | 平均准确率 | 平均召回率 |
| 5 | 0.89846154 | 0.0323474 | 0.93538462 | 0.03363204 |
| 10 | 0.90615385 | 0.06545172 | 0.89948718 | 0.06565937 |
| 20 | 0.88230769 | 0.12877512 | 0.88430769 | 0.13023986 |
| 30 | 0.86512821 | 0.22332875 | 0.87120879 | 0.19369472 |
| 40 | 0.85076923 | 0.28745645 | 0.86230769 | 0.25764192 |
| 50 | 0.84123077 | 0.31999064 | 0.852 | 0.32276489 |
| 100 | 0.78092308 | 0.66474594 | 0.77369231 | 0.66266965 |

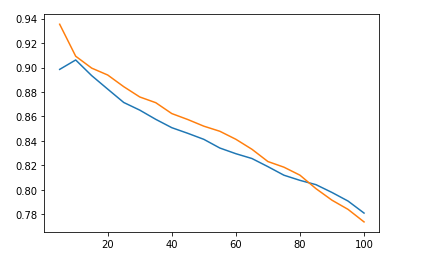
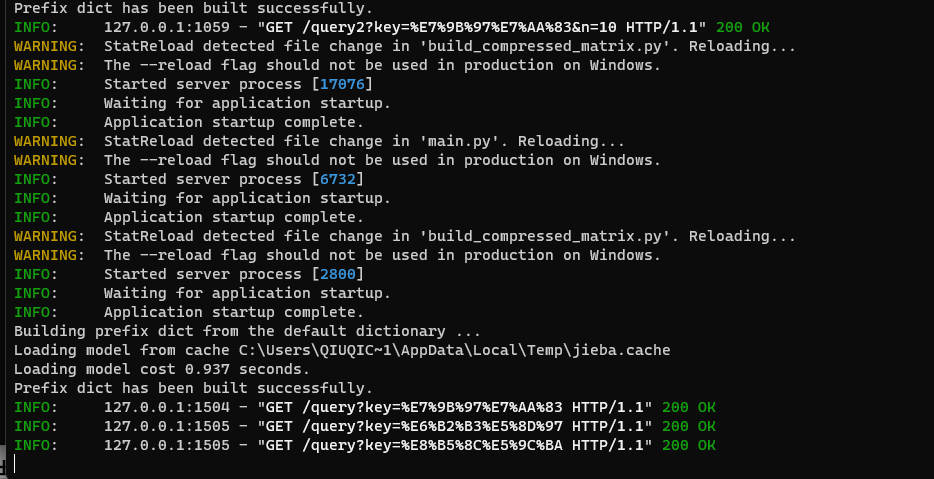


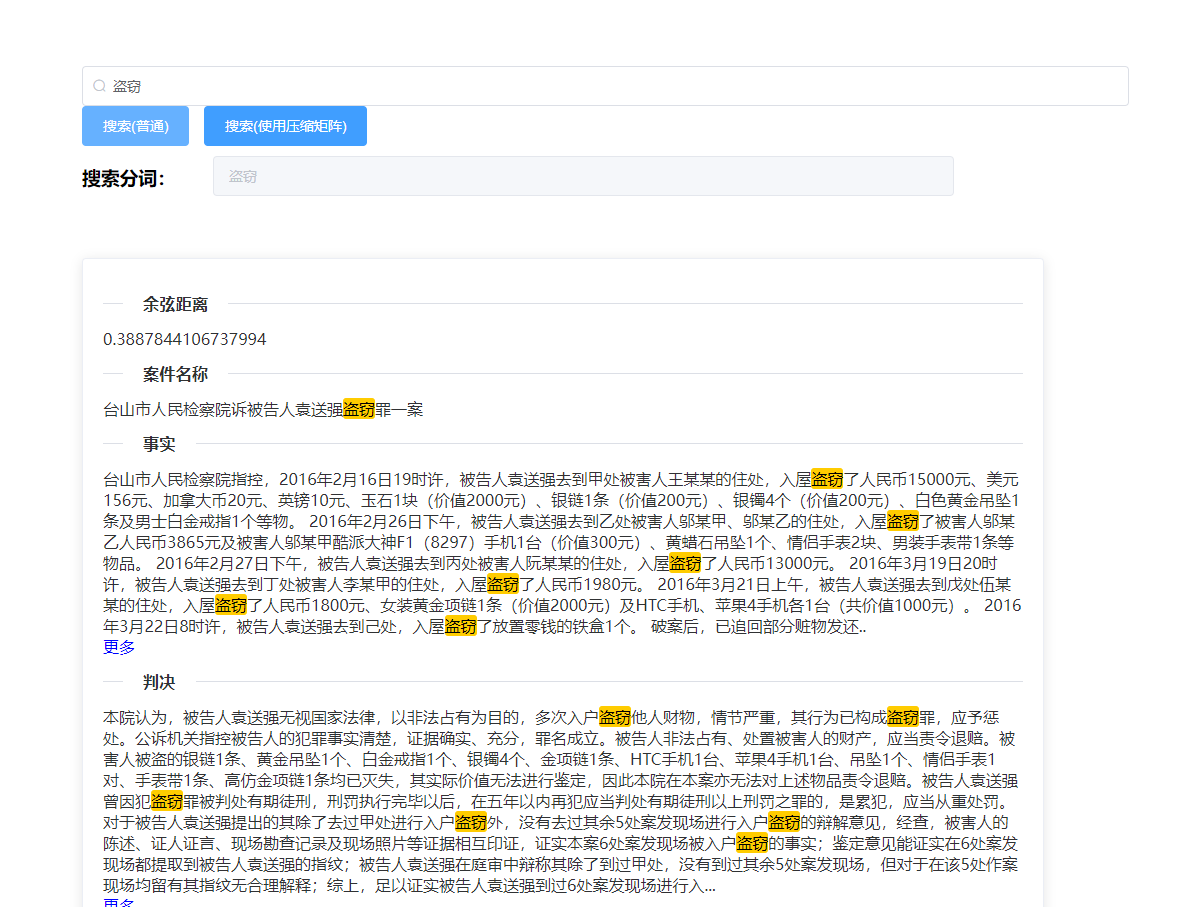
Figure 准确率图（黄色为SVD压缩K=2500，蓝色为普通）

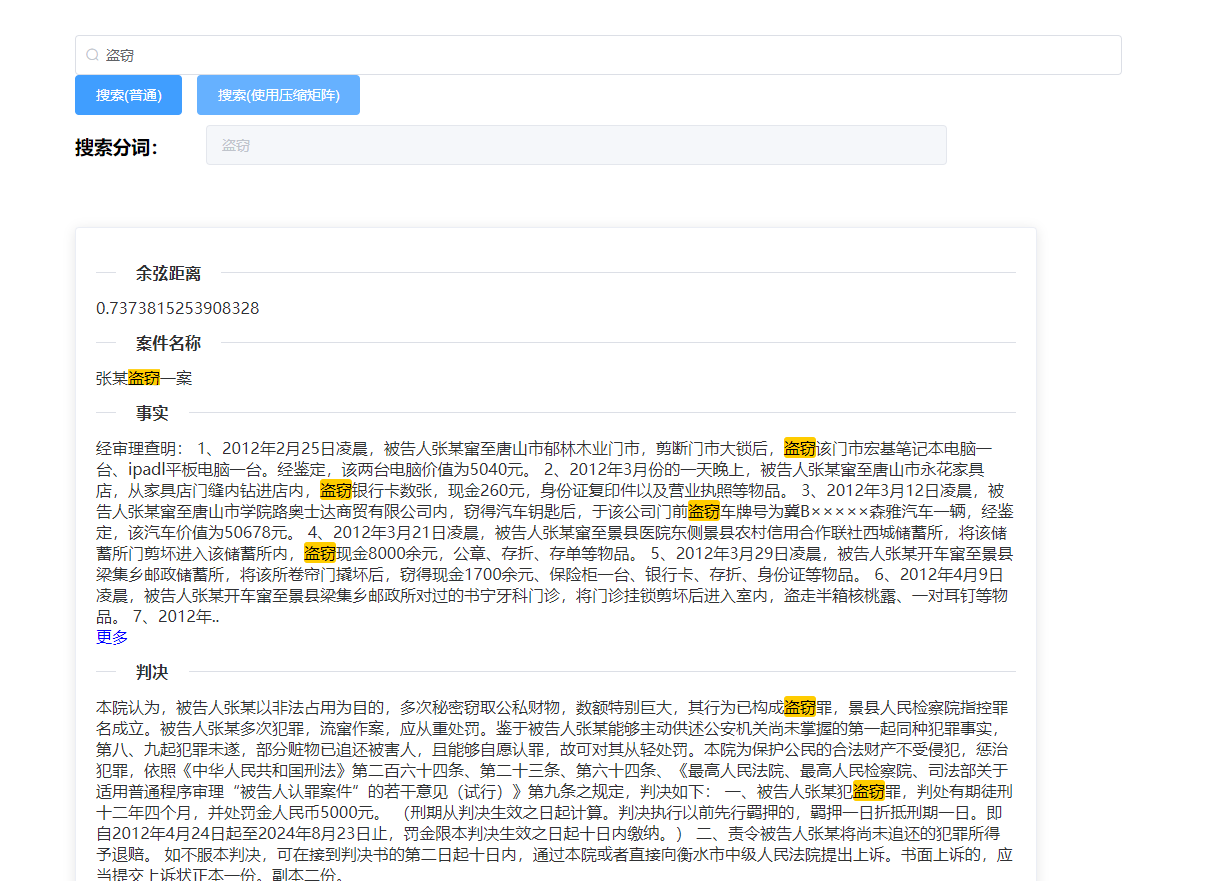
1. 演示

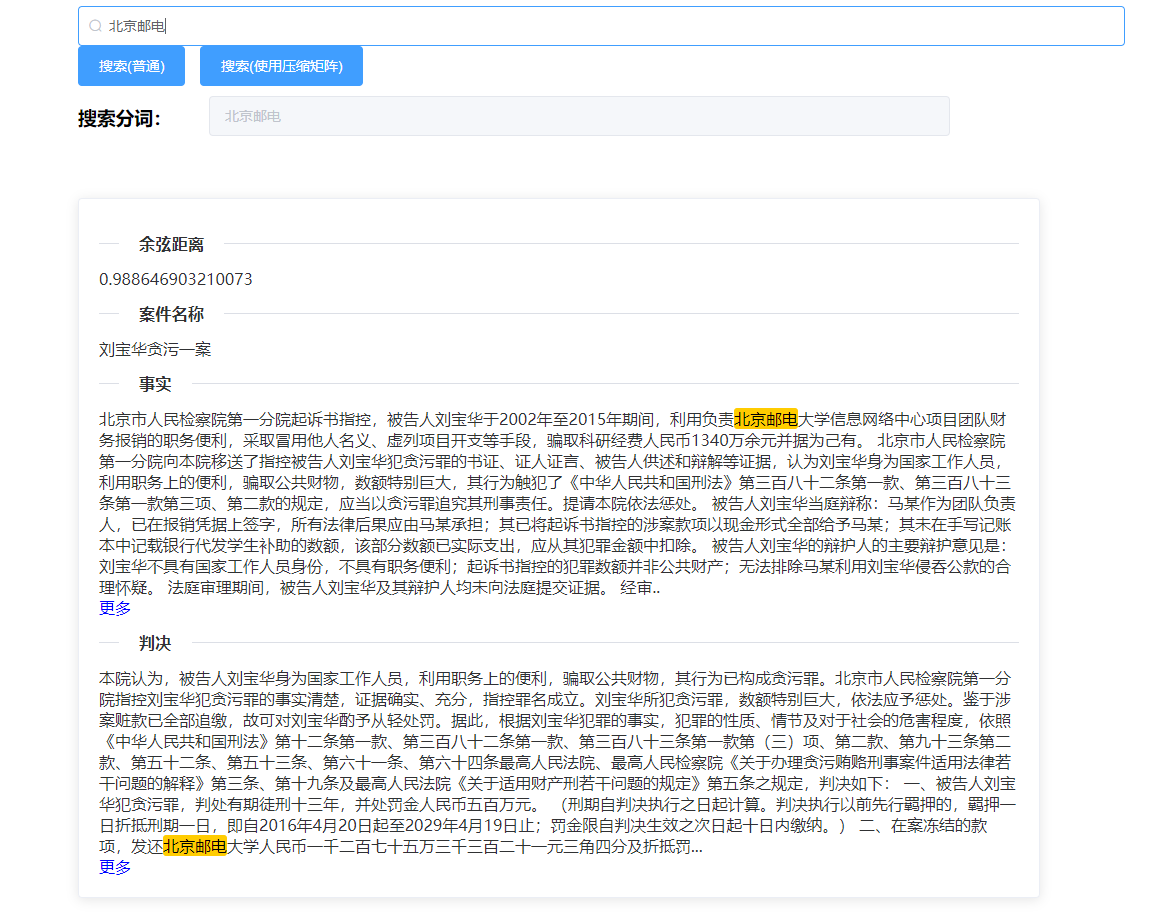
演示部分主要参见课堂上的演示。这里只截取几个典型的运行时图片。

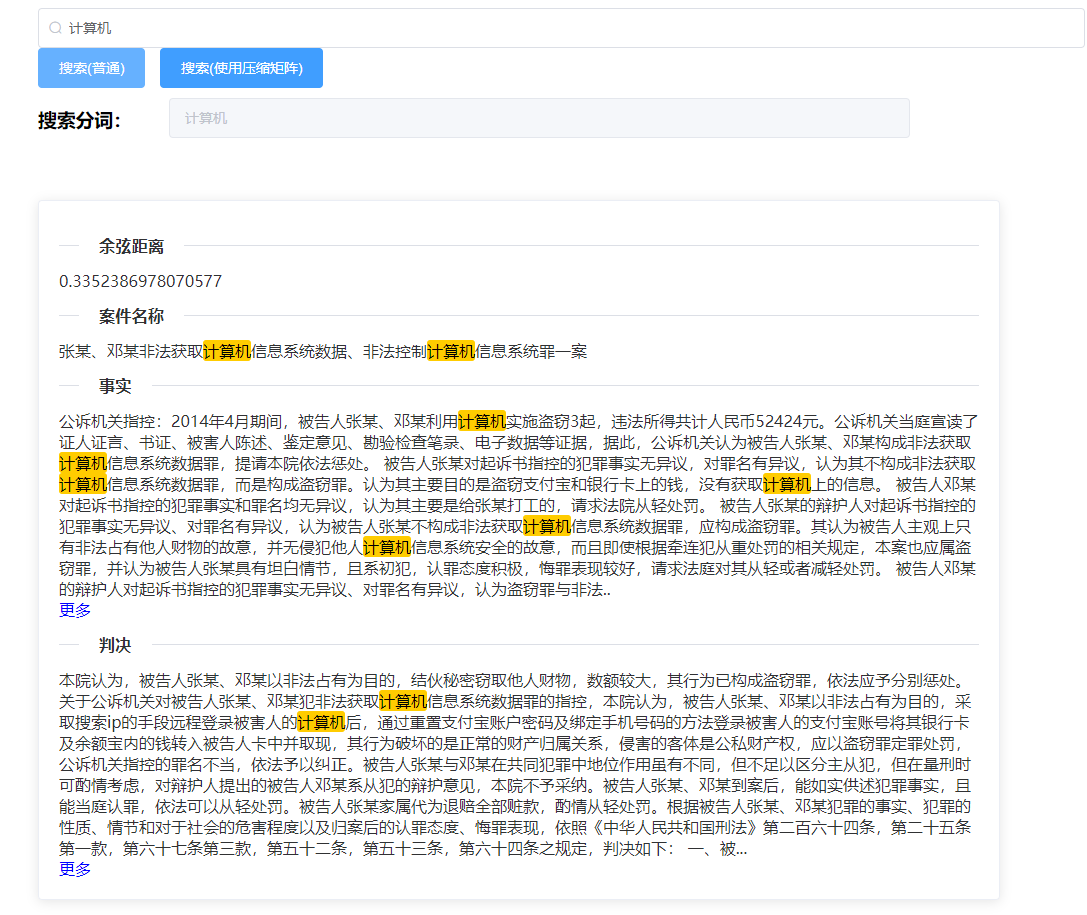
下面是后端的控制台

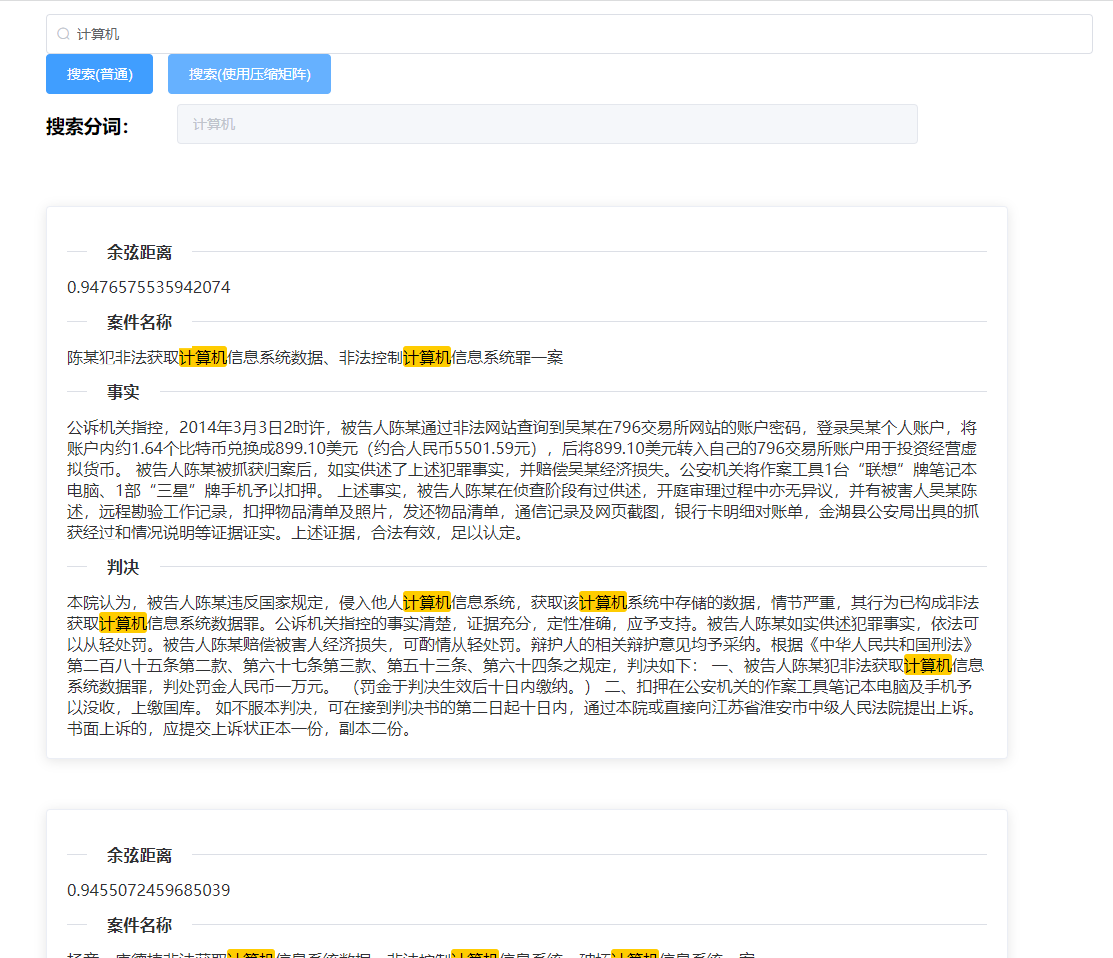












1. 总结

在这个实验中，我们进一步学习和实践了倒排索引，词向量，余弦距离等知识，设计了一个法律文书的检索系统。我们在此基础上进一步探究降维矩阵，并取得了较好的成果。