

武汉大学国家网络安全学院

实验报告

课程名称 信 息 检 索

专业年级 网络空间安全

姓 名 范圣悦

学 号 **2018302080152**

协 作 者 无

实验学期 **2020-2021** 学年 第二 学期

课堂时数 **32** 课外时数

填写时间 **2021** 年 **6** 月 **13** 日

实验介绍
<p>【实验名称】： 信息检索</p> <p>【实验目的】： 以命令行形式，输入查询语句完成对文档库的检索与查询，返回文档库中符合要求的前 5 篇文档。</p> <p>【实验环境】： 电脑配置：Intel(R) Core(TM) i5-8300H CPU @2.30GHz 2.30GHz,16G 内存 操作系统：Windows10 家庭中文版[版本 10.0.17763.1217] 开发环境：Pycharm 2020.1 x64 natsort 7.1.1 nltk 3.6.2</p> <p>【参考文献】： [1]词干提取 - Stemming 词形还原 - Lemmatisation,[EB/OL], https://easyai.tech/ai-definition/stemming-lemmatisation, 2018-12-22 [2][分词 - Tokenization] [EB/OL], https://easyai.tech/ai-definition/tokenization,2019-10-05. [3]Lucene 倒排索引原理[EB/OL], https://juejin.cn/post/6947984489960177677, 2020-05-12. [4] What is TF-IDF? [EB/OL], https://monkeylearn.com/blog/what-is-tf-idf , 2019-05-11.</p>
实验内容
<p>【实验方案设计】：</p> <p>1.文档预处理 文档预处理主要分为三部分：词干提取、分词、去停用词。</p> <p><i>(1) 词干提取</i></p>

在词法学和信息检索里，词干提取是去除词缀得到词根的过程（得到单词最一般的写法）。对于一个词的形态词根，词干并不需要完全相同；相关的词映射到同一个词干一般能得到满意的结果，即使该词干不是词的有效根。从 1968 年开始在计算机科学领域出现了词干提取的相应算法。很多搜索引擎在处理词汇时，对同义词采用相同的词干作为查询拓展，该过程叫做归并。词干提取项目一般涉及到词干提取算法或词干提取器。

举例：一个面向英语的词干提取器，例如要识别字符串“cats”、“catlike”和“catty”是基于词根“cat”；“stemmer”、“stemming”和“stemmed”是基于词根“stem”。一根词干提取器可以简化词“fishing”、“fished”、“fish”和“fisher”为同一个词根“fish”。

NLTK 中提供了三种最常用的词干提取器接口，即 Porter stemmer, Lancaster Stemmer 和 Snowball Stemmer。

Porter Stemmer 基于 Porter 词干提取算法。Porter stemmer 并不是要把单词变为规范的那种原来的样子，它只是把很多基于这个单词的变种变为某一种形式。换句话说，它不能保证还原到单词的原本，也就是“created”不一定能还原到“create”，但却可以使“create”和“created”，都得到“creat”。

（2）分词

分词就是将句子、段落、文章这种长文本，分解为以字词为单位的数据结构，方便后续的处理分析工作。

分词的方法大致分为 3 类：

1. 基于词典匹配
2. 基于统计
3. 基于深度学习

给予词典匹配的分词方式

优点：速度快、成本低

缺点：适应性不强，不同领域效果差异大

基本思想是基于词典匹配，将待分词的中文文本根据一定规则切分和调整，然后跟词典中的词语进行匹配，匹配成功则按照词典的词分词，匹配失败通过调整或者重新选择，如此反复循环即可。代表方法有基于正向最大匹配和基于逆向最大匹配及双向匹配法。

基于统计的分词方法

优点：适应性较强

缺点：成本较高，速度较慢

这类目前常用的是算法是 HMM、CRF、SVM、深度学习等算法，比如 stanford、Hanlp 分词工具是基于 CRF 算法。以 CRF 为例，基本思路是

对汉字进行标注训练，不仅考虑了词语出现的频率，还考虑上下文，具备较好的学习能力，因此其对歧义词和未登录词的识别都具有良好的效果。

基于深度学习

优点：准确率高、适应性强

缺点：成本高，速度慢

例如有人尝试使用双向 [LSTM](#)+CRF 实现分词器，其本质上是序列标注，所以有通用性，命名实体识别等都可以使用该模型，据报道其分词器字符准确率可高达 97.5%。

常见的分词器都是使用机器学习算法和词典相结合，一方面能够提高分词准确率，另一方面能够改善领域适应性。

NLTK 提供了分词方法，本实验中使用其提供的正则分词器。

(3) 去停用词

停用词是指在信息检索中，为节省存储空间和提高搜索效率，在处理自然语言数据（或文本）之前或之后会自动过滤掉某些字或词，这些字或词即被称为 **Stop Words**（停用词）。这些停用词都是人工输入、非自动化生成的，生成后的停用词会形成一个停用词表。

对于一个给定的目的，任何一类的词语都可以被选作停用词。通常意义上，停用词大致分为两类：

- 一类是人类语言中包含的功能词，这些功能词极其普遍，与其他词相比，功能词没有什么实际含义，比如'the'、'is'、'at'、'which'、'on'等。但是对于搜索引擎来说，当所要搜索的短语包含功能词，特别是像'The Who'、'The The'或'Take The'等复合名词时，停用词的使用就会导致问题。
- 另一类词包括词汇词，比如'want'等，这些词应用十分广泛，但是对这样的词搜索引擎无法保证能够给出真正相关的搜索结果，难以帮助缩小搜索范围，同时还会降低搜索的效率，所以通常会把这些词从问题中移去，从而提高搜索性能。

因此对于需要处理的文本，在预处理过程中去除其中的停用词，有助于后续处理，NLTK 提供了常见的英文通用词表。

2.索引建立

基于 Lucene 的实现思路，可以对文档库建立**倒排索引**，实现后续的文档检索功能。

(1) 倒排索引基本原理

搜索的核心需求是全文检索，全文检索简单来说就是要在大量文档中找到包含某个单词出现的位置，在传统关系型数据库中，数据检索只能通过 like 来实现。

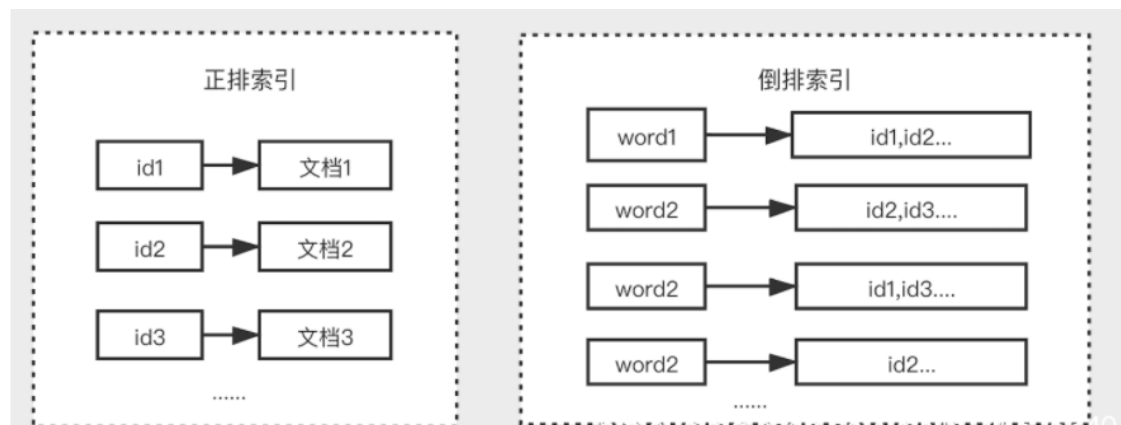
这种实现方式实际会存在很多问题：

- 无法使用数据库索引，需要全表扫描，性能差
- 搜索效果差，只能首尾位模糊匹配，无法实现复杂的搜索需求
- 无法得到文档与搜索条件的相关性

搜索的核心目标实际上是保证搜索的效果和性能，为了高效的实现全文检索，我们可以通过倒排索引来解决。

倒排索引是区别于正排索引的概念：

- 正排索引：是以文档对象的唯一 ID 作为索引，以文档内容作为记录的结构。
- 倒排索引：Inverted index，指的是将文档内容中的单词作为索引，将包含该词的文档 ID 作为记录的结构。



(2) 倒排索引生成过程

假设目前有以下两个文档内容：

苏州街维亚大厦

桔子酒店苏州街店

其处理步骤如下：

1、正排索引给每个文档进行编号，作为其唯一的标识。

文档 id	content
1	苏州街维亚大厦
2	桔子酒店苏州街店

2、生成倒排索引：

- a. 首先要对字段的内容进行分词，分词就是将一段连续的文本按照语义拆分为多个单

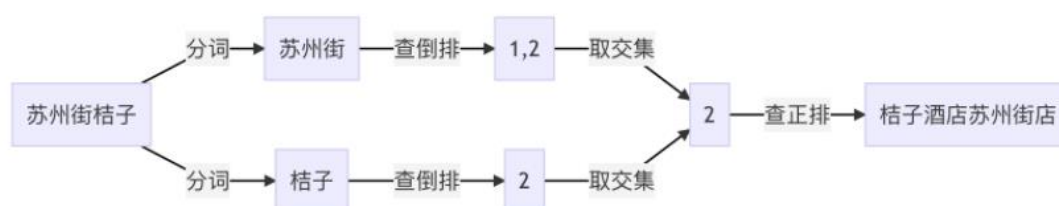
词，这里两个文档包含的关键词有：苏州街、维亚大厦...

- b.然后按照单词来作为索引，对应的文档 id 建立一个链表，就能构成上述的倒排索引结构。

Word	文档 id
苏州街	1,2
维亚大厦	1
维亚	1
桔子	2
酒店	2
大赛	1

有了倒排索引，能快速、灵活地实现各类搜索需求。整个搜索过程中我们不需要做任何文本的模糊匹配。

例如，如果需要在上述两个文档中查询 苏州街桔子，可以通过分词后通过 苏州街 查到 1、2，通过 桔子 查到 2，然后再进行**取交集**等操作得到最终结果。



3.计算文档库中词语的 TF-IDF

TF-IDF 是一种统计方法，用以评估一字词对于一个文件集或一个语料库中的其中一份文件的重要程度。字词的重要性随着它在文件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。

TF-IDF 的主要思想是：如果某个单词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。

(1) TF (Term Frequency) :

词频 (TF) 表示词条 (关键字) 在文本中出现的频率。

这个数字通常会被归一化(一般是词频除以文章总词数)，以防止它偏向长的文

件。

公式: $tf_{ij} = \frac{n_{ij}}{\sum_k n_{kj}}$ 即: $TF_w = \frac{\text{在某一类中词条 } w \text{ 出现的次数}}{\text{该类中所有的词条数目}}$

其中 n_{ij} 是该词在文件中出现的次数，分母则是文件中所有词汇出现的次数总和；

(2) IDF (Inverse Document Frequency) :

逆向文件频率 (IDF) : 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得到。

如果包含词条 t 的文档越少, IDF 越大, 则说明词条具有很好的类别区分能力。

公式: $idf_i = \log \frac{|D|}{|\{j: t_i \in d_j\}|}$

其中, $|D|$ 是语料库中的文件总数。 $|\{j: t_i \in d_j\}|$ 表示包含词语 t_i 的文件数目 (即 $n_{ij} \neq 0$ 的文件数目)。

(3) TF-IDF:

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。

公式: $TF - IDF = TF * IDF$

4. 查询语句与文档之间的相似度

当给定某条查询语句时, 首先将查询语句和文档分别进行预处理 (分词、去停用

词)，对于每一个词，分别查询是否在语句/文档中，最终取交集，返回相关度。

5.代码实验与说明

1.导入工具包

```
import math
import os
import nltk
from nltk.stem import PorterStemmer
from natsort import natsorted
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from collections import defaultdict
from nltk.tokenize import RegexpTokenizer
```

2.文件预处理

```
class Posi_Index_Builder:
    def __init__(self, folderName):
        self.folderName = folderName
        nltk.download('punkt')
        nltk.download('stopwords')
        # 引入英文停用词表
        self.stopWords = set(stopwords.words('english'))
        # 词干提取
        self.stemmer = PorterStemmer()
        # 使用正则分词器
        self.tokenizer = RegexpTokenizer(r'\w+')
        # 语料库
        self.corpus = []
        # 分词、去停用词后的语料库
        self.corpusNormalizedAndTokenized = []
        # 词干索引列表
        self.positionalIndex = defaultdict(list)

        # 读取文件内容
        def readFile(self, fileName):
            notebook_dirctory = os.getcwd()
            file = open(notebook_dirctory + "/" + self.folderName + "/" +
            fileName, 'r', encoding='utf-8')
            dataFromFile = file.read()
            file.close()
            self.corpus.append(dataFromFile)

        # 进行正则分词、去停用词
```



```

def tokenizeAndNormalize(self):
    # 进行正则分词
    for i in range(0, len(self.corpus)):
        self.corpus[i] = self.tokenizer.tokenize(self.corpus[i])
        # 将大写字母转换为小写
        for x in range(len(self.corpus[i])):
            self.corpus[i][x] = self.corpus[i][x].lower()
    # 去除停用词表中的停用词
    for i in self.corpus:
        temp = []
        for j in i:
            if j not in self.stopWords:
                temp.append(j)
        self.corpusNormalizedAndTokenized.append((temp))

3.索引建立
# 建立位置索引
def buildPositionalIndex(self):
    notebook_directory = os.getcwd()
    fileNo = 0
    file_map = {}
    # 进行文件名排序
    files = natsorted(os.listdir(notebook_directory + "\\\" + self.folderName))
    # 读取每个文件中的内容，保存到self.corpus中
    for i in files:
        self.readFile(i)
    # 进行正则分词、去停用词，保存到self.corpusNormalizedAndTokenized中
    self.tokenizeAndNormalize()
    for corpus in self.corpusNormalizedAndTokenized:
        for index, word in enumerate(corpus):
            # 提取当前词的词干
            stemmedTerm = self.stemmer.stem(word)
            # 如果该词干已存在索引列表中
            if stemmedTerm in self.positionalIndex:
                # 频率+1
                self.positionalIndex[stemmedTerm][0] = self.positionalIndex[stemmedTerm][0] + 1

            # 如果当前文件在该词干的文件库中，把当前词加入索引列表
            if fileNo in self.positionalIndex[stemmedTerm][1]:
                self.positionalIndex[stemmedTerm][1][fileNo].append(index)
            # 如果当前文件不在该词干的文件库中，把当前词作为索引
            else:

```

```

        self.positionalIndex[stemmedTerm][1][fileNo] = [index]
dex]

    # 如果该词干不在索引列表中
    else:
        # 初始化词干索引列表
        self.positionalIndex[stemmedTerm] = []
        # 词干频率+1
        self.positionalIndex[stemmedTerm].append(1)
        # 初始化一个空字典
        self.positionalIndex[stemmedTerm].append({})
        # 添加文档编号
        self.positionalIndex[stemmedTerm][1][fileNo] = [index]
    file_map[fileNo] = notebook_dirctory + "/" + self.folderName +
"/" + i
    # 进行下一篇文章的分析
    fileNo += 1

```

4.计算 TF、IDF、TF-IDF

class TFIDF_Builder:

```

    def __init__(self, rawData, data, positionalIndex):
        # 元数据(未处理的语料库)
        self.rawData = rawData
        # 预处理后的语料库
        self.data = data
        # 词频 (term frequency, TF)
        self.tf = []
        # 逆向文件频率 (inverse document frequency, IDF)
        self.idf = []
        # tf-idf 值, 即 TF 与 IDF 的乘积
        self.TFIDF = []
        # 词干索引列表
        self.positionalIndex = positionalIndex

    # 计算每个词在每篇文档中的词频 (TF 值)
    def get_TF(self):
        self.tf = defaultdict(list)
        count = 0
        # 初始化每一个词的 TF 值都为0
        for i in self.rawData:
            for x in i:
                self.tf[x] = [0 for f in range(len(self.rawData))]
        # 统计词频, 保留4位小数
        for i in range(len(self.rawData)):
            for x in self.rawData[i]:
                count = self.rawData[i].count(x)

```

```

        self.tf[x][i] = round(float(count), 4)
    return self.tf

# 计算每一个词的 IDF 值
def get_IDF(self):
    # 初始化列表
    self.idf = defaultdict(list)
    for i in range(len(self.rawData)):
        for x in self.rawData[i]:
            # 把“包含该词语的文件数目”初始化为0
            numberOfDocsWithTerm = 0
            # 若该词包含在该文件中, 则 numnumberOfDocsWithTerm+1
            for f in self.rawData:
                if x in f:
                    numberOfDocsWithTerm += 1
            # 计算 IDF, 用总文件数目除以包含该词语的文件数目, 再取对数,
            # 结果保留 4 位小数
            self.idf[x] = round(math.log10(float(len(self.data) /
            numberOfDocsWithTerm)), 4)

    return self.idf

# 计算每个词在每篇文档中的 TF-IDF 值
def get_TFIDF(self):
    self.TFIDF = defaultdict(float)
    # 初始化每一个词的 TF-IDF 值都为0
    for i in self.rawData:
        for x in i:
            self.TFIDF[x] = [0 for f in range(len(self.rawData))]
    # 计算 TF-IDF 值: 把每个词的 TF 值与 IDF 相乘, 结果保留 4 为小数
    for i in range(len(self.rawData)):
        for x in self.rawData[i]:
            if len(x) <= 1:
                continue
            self.TFIDF[x][i] = round(self.tf[x][i] * self.idf[x],
4)

    return self.TFIDF

```

5. 查询相关度前 5 的文档

计算查询语句与文档之间的相似度

```

def findSimilarity(self, query, doc):
    similarity = 0
    vector_query = []
    vector_doc = []

```

```

# 对查询语句和文档进行分词
temp_query = word_tokenize(query)
temp_doc = word_tokenize(doc)
# 去停用词
sw = set(stopwords.words('english'))
tokenizedAndNormalizedQuery = {word for word in temp_query if word not in sw}
tokenizedAndNormalizedDoc = {word for word in temp_doc if word not in sw}
# vector 包含查询语句和文档中的所有词
vector = tokenizedAndNormalizedDoc.union(tokenizedAndNormalizedQuery)

for word in vector:
    # 如果当前词在文档中
    if word in tokenizedAndNormalizedDoc:
        vector_query.append(1)
    # 如果当前词不在文档中
    else:
        vector_query.append(0)
    # 如果当前词在查询语句中
    if word in tokenizedAndNormalizedQuery:
        vector_doc.append(1)
    # 如果当前词不在查询语句中
    else:
        vector_doc.append(0)
    # 取交集，若某词同时在查询语句和文档中，则为1，否则为0；把所有词取交集的结果相加得到相关度
    for i in range(len(vector)):
        similarity += vector_doc[i] * vector_query[i]
    # 计算最终相似度：
    answer = similarity / float((sum(vector_doc) * sum(vector_query))
    ** 0.5)
    return answer

def findTop5(self, query):
    allSimilarity = []
    for i in range(len(self.rawData)):
        # 对每一篇文档，计算查询语句与当前文档的相似度
        allSimilarity.append((i + 1, self.findSimilarity(query, " ".join(self.rawData[i]))))
    # 按相似度从大到小排序
    allSimilarity.sort(key=lambda x: x[-1], reverse=True)
    # 返回top5 的文档编号及相似度
    return allSimilarity[:5]

```

6.主函数

```
from Posi_Index_Builder import Posi_Index_Builder
from TFIDF_Builder import TFIDF_Builder

if __name__ == '__main__':
    # 文件预处理和建立索引
    Index_Build = Posi_Index_Builder("docs")
    Index_Build.buildPositionalIndex()

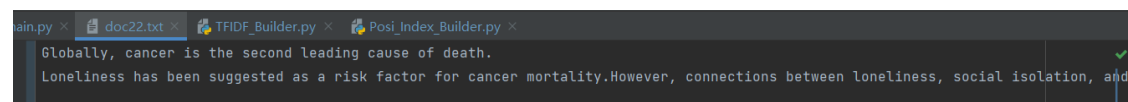
    # 计算语料库中词语的 TF、IDF、TF-IDF
    vectorBuilder = TFIDF_Builder(Index_Build.corpus,
                                   Index_Build.corpusNormalizedAndTokeni
zed,
                                   Index_Build.positionalIndex)

    print('计算每个词在每篇文档中的 TF 值:')
    print(vectorBuilder.get_TF())
    print('计算每一个词的 IDF 值:')
    print(vectorBuilder.get_IDF())
    print('计算每个词在每篇文档中的 TF-IDF 值:')
    print(vectorBuilder.get_TFIDF())

    # 输入查询语句
    query = input('输入查询语句或者 cancel 退出')
    while query.lower() != 'cancel':
        # 把输入的语句与每一篇文档进行比对, 返回最匹配的 5 篇文档
        print(vectorBuilder.findTop5(query))
        # 继续查询
        query = input('输入查询语句或者 cancel 退出')
```

【实验结果分析】：

本实验所用的文档库来自[英语点津](#)中的 50 篇短新闻文本，内容包括体育、奢侈品、新冠疫情、东京奥运会、生活健康等方面内容，文档实例如下：



- 自由查询：

输入 new job，查找与其相关度最高的 5 篇文档，结果如下：

```
输入查询语句或者cancel退出new job
```

```
[(31, 0.30151134457776363), (30, 0.23249527748763857), (40, 0.1414213562373095), (29, 0.1270001270001905), (48, 0.10314212462587934)]
```

```
输入查询语句或者cancel退出
```

显示匹配度最高的前 5 篇文档编号分别是 31、30、40、29、48，而后是他们各自的相关度得分。

- 布尔查询：

输入 heart disease OR The Chinese government，查找与心脏疾病或者中国政府相关度最高的 5 篇文档，结果如下：

```
输入查询语句或者cancel退出heart disease OR The Chinese government
```

```
[(48, 0.11909826683508273), (39, 0.08333333333333333), (47, 0.07580980435789034), (18, 0.07001400420140048), (35, 0.056077215409204434)]
```

```
输入查询语句或者cancel退出
```

显示匹配度最高的前 5 篇文档编号分别是 48、39、47、18、35，而后是他们各自的相关度得分。

评估结果：

准确率(Accuracy)、查准率(Precision)、查全率(Recall)是常见的基本指标。

Accuracy（准确率）

分类正确的样本数 与 样本总数之比。即： $(TP + TN) / (ALL)$ 。

Precision（精确率、查准率）

被正确检索的样本数 与 被检索到样本总数之比。即： $TP / (TP + FP)$ 。

Recall（召回率、查全率）

被正确检索的样本数 与 应当被检索到的样本数之比。即： $TP / (TP + FN)$ 。

更多时候，我们希望能够同时参考 Precision 与 Recall，但又不是像 Accuracy 那样只是泛泛地计算准确率，此时便引入一个新指标 F-Score，用来综合考虑 Precision 与 Recall。

其中 β 用于调整权重，当 $\beta=1$ 时两者权重相同，简称为 F1-Score。若认为 Precision 更重要，则减小 β ，若认为 Recall 更重要，则增大 β 。

编写代码，分别计算上述查询语句得到结果（此处计算 f1 时返回匹配度前 10 的文档）的 F1-score：

- 查询 new job:

```
Accuracy : 0.8400  
Precision : 0.6897  
Recall : 0.7538  
f1-score : 0.7165
```

- 查询 heart disease OR The Chinese government:

```
Accuracy : 0.8500  
Precision : 0.7500  
Recall : 0.8056  
f1-score : 0.7722
```

多次查询并计算 f1，得到最终的 f1-score 平均值为 0.7426:

```
average_f1:0.7426  
  
Process finished with exit code 0
```

【实验总结】：

通过本次实验，我对信息检索有了更深的认识，深刻体会了 TF、IDF 和 TF-IDF 无论对于自然语言处理还是文本分类的有效作用，理解了倒排索引的基本原理，知道了倒排和正排的区别与优劣。通过自己对信息检索代码的实现，提高了自己的自主学习能力和编程能力。


总的来说，通过这门课程的学习，加深了我对信息检索的理解，提高了我的信息检索能力。

评语及评分（指导教师）

【评语】：

该实验报告设计并实现了基于 TF-IDF 实现的文档检索系统，并在自己爬取的数据集上进行了定性评价，关于预处理部分设计良好，基本达到设计目标。关于

实验评价从 precision 等指标上进行评价，但缺少与同类算法的对比与评价算法的设计。

评分: **85** 
日期: **2021-6-26**