Lecture 2 超基础复习笔记 - 正则表达式与文档相似度

零、先理解问题（为什么需要这些？）

现实场景

场景1：查找抄袭

> 你是老师，收到100篇作业
> 怎么快速找出哪些作业内容相似（可能抄袭）？
> → 需要计算文档相似度！

场景2：新闻推荐

> 用户刚看了一篇关于"足球"的新闻
> 怎么找出其他类似的足球新闻推荐给他？
> → 需要比较新闻内容的相似度！

场景3：提取文本中的单词

> 原文：Hello, World! How are you?
> 只想要单词：Hello World How are you
> → 需要用正则表达式提取单词！

一、正则表达式（超级简单版）

什么是正则表达式?

正则表达式 = 文本匹配的模式（规则）

想象你在玩"找不同"游戏，需要一个规则来描述你要找什么。

基本符号（必须记住！）

符号	意思	例子	匹配结果
`[A-Z]`	任意大写字母	`[A-Z]`	匹配：A, B, C, ..., Z
`[a-z]`	任意小写字母	`[a-z]`	匹配：a, b, c, ..., z
`[A-Za-z]`	任意字母	`[A-Za-z]`	匹配：a, A, b, B, ...
`+`	**至少1次**（Kleene plus）	`[a-z]+`	匹配：a, ab, abc, hello
`*`	**0次或多次**（Kleene star）	`[a-z]*`	匹配：(空), a, ab, abc
`[0-9]`	任意数字	`[0-9]+`	匹配：1, 23, 456

图解理解

例子1：`[A-Za-z]+` 的含义

```
[A-Za-z]  →  任意一个字母（大写或小写）
       +  →  至少出现1次，可以多次

匹配的例子：
✅ "Hello"   (5个字母)
✅ "a"       (1个字母)
✅ "ABC"     (3个大写字母)
❌ ""        (0个字母，不满足"至少1次")
❌ "123"     (不是字母)
```

例子2：`[a-z]*` vs `[a-z]+`

```
文本："Hello123World"

[a-z]+  →  只匹配：ello，orld  （至少1个小写字母）
[a-z]*  →  匹配：H(空)e(空)l(空)l(空)o(空)1(空)2(空)3(空)W(空)o(空)r(空)l(空)d(空)
           （可以是0个，所以到处都匹配）
```

结论：一般用 `+` 更有用！

二、Python 中使用正则表达式

基本步骤（三步走）

```
import re  # 第1步：导入re模块

# 第2步：编译正则表达式（创建匹配规则）
wordRE = re.compile('[A-Za-z]+')

# 第3步：用规则去匹配文本
text = "Hello, World! 123"
words = wordRE.findall(text)
print(words)  # 输出：['Hello', 'World']
```

详细解释每一步

第1步：`import re`

- `re` 是 Python 内置的正则表达式模块
- 就像 `import math` 一样，导入后才能用

第2步：`re.compile('[A-Za-z]+')`

```
wordRE = re.compile('[A-Za-z]+')
```

分解理解： - `re.compile()` - 把字符串变成"匹配规则" - `'[A-Za-z]+'` - 规则内容："1个或多个字母" - `wordRE` - 变量名，存储这个规则（像一个"筛子"）

为什么要compile? - 把规则"编译"一次，后面重复用很多次 - 更快！如果不编译，每次匹配都要重新解析规则

第3步：`wordRE.findall(text)`

```
text = "Hello, World! 123"
words = wordRE.findall(text)
# words = ['Hello', 'World']
```

工作过程（图解）：

```
原文：H e l l o ,  W o r l d !  1 2 3

规则：[A-Za-z]+ （找连续的字母）

扫描：
H → 字母✅，继续
e → 字母✅，继续
l → 字母✅，继续
l → 字母✅，继续
o → 字母✅，继续
, → 不是字母❌，停止，输出"Hello"

  → 空格，跳过
W → 字母✅，开始新的匹配
o → 字母✅，继续
r → 字母✅，继续
l → 字母✅，继续
d → 字母✅，继续
! → 不是字母❌，停止，输出"World"

（后面的123都不是字母，跳过）

最终结果：['Hello', 'World']
```

三、Jaccard 相似度（核心算法！）

什么是相似度?

相似度 = 两个东西有多像

比如： - 双胞胎相似度很高（90%） - 陌生人相似度很低（10%）

Jaccard 系数原理

公式（数学版）:

```
J(A, B) = |A ∩ B| / |A ∪ B|
```

公式（人话版）:

```
相似度 = 共同拥有的东西数量 / 总共拥有的东西数量
```

图解理解（超重要！）

例子：比较两个人的爱好

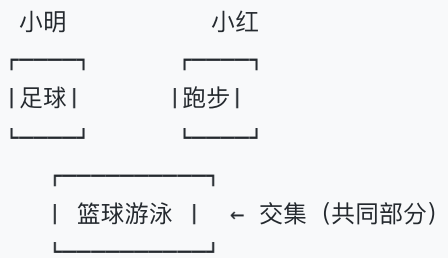
小明的爱好: `{足球, 篮球, 游泳}` **小红的爱好:** `{篮球, 游泳, 跑步}`

```
第1步：找交集（共同爱好）
A ∩ B = {篮球, 游泳}
共同爱好数量 = 2

第2步：找并集（所有爱好）
A ∪ B = {足球, 篮球, 游泳, 跑步}
所有爱好数量 = 4

第3步：计算相似度
J = 2 / 4 = 0.5   (50%相似)
```

Venn图:



文档相似度的计算

简单例子（二进制权重）

文档1： "我 爱 编程" **文档2：** "我 爱 音乐"

```
文档1的词集合：{我，爱，编程}
文档2的词集合：{我，爱，音乐}

交集：{我，爱}              → 2个词
并集：{我，爱，编程，音乐} → 4个词

相似度 = 2 / 4 = 0.5
```

复杂例子（考虑词频）

文档1： "我 爱 爱 爱 编程"（"爱"出现3次） **文档2：** "我 爱 音乐"

词频统计：

```
文档1：{我:1，爱:3，编程:1}
文档2：{我:1，爱:1，音乐:1}
```

计算过程（带权重）：

```
所有词：{我，爱，编程，音乐}

对每个词计算：
- "我"：doc1有1次，doc2有1次
  → 交集贡献：min(1, 1) = 1
  → 并集贡献：max(1, 1) = 1

- "爱"：doc1有3次，doc2有1次
  → 交集贡献：min(3, 1) = 1
  → 并集贡献：max(3, 1) = 3

- "编程"：doc1有1次，doc2有0次
  → 交集贡献：min(1, 0) = 0
  → 并集贡献：max(1, 0) = 1

- "音乐"：doc1有0次，doc2有1次
  → 交集贡献：min(0, 1) = 0
  → 并集贡献：max(0, 1) = 1

总交集 = 1 + 1 + 0 + 0 = 2
总并集 = 1 + 3 + 1 + 1 = 6

相似度 = 2 / 6 = 0.333
```

四、代码详细解释（逐行讲解）

整体结构

```
# 第1部分：处理命令行参数
# 第2部分：读取停用词
# 第3部分：统计每个文档的词频
# 第4部分：计算所有文档对的相似度
# 第5部分：排序并输出结果
```

第1部分：命令行参数（了解即可）

```
import sys, re, getopt

opts, args = getopt.getopt(sys.argv[1:], 'hs:pbI:')
opts = dict(opts)
```

什么是命令行参数?

```
# 在终端运行程序时的额外选项
python program.py -s stoplist.txt file1.txt file2.txt
                  ↑               ↑
                  选项            文件名
```

参数说明： - `-s stoplist.txt` - 指定停用词文件 - `-p` - 使用词干提取 - `-b` - 使用二进制权重（不考虑词频） - `-I pattern` - 用通配符指定文件（如 `*.txt`）

代码解释:

```
opts, args = getopt.getopt(sys.argv[1:], 'hs:pbI:')
```

- `sys.argv` - 所有命令行参数的列表
- `sys.argv[0]` - 程序名
- `sys.argv[1:]` - 从第2个开始的参数
- `'hs:pbI:'` - 定义哪些选项可用
- `h` - 不需要值的选项
- `s:` - 需要值的选项（冒号表示后面跟参数）

第2部分：读取停用词

```
stops = set()
if '-s' in opts:
    with open(opts['-s'], 'r') as stop_fs:
        for line in stop_fs:
            stops.add(line.strip())
```

什么是停用词?

停用词 = 太常见、没有实际意义的词

例如： - 英文：the, a, an, is, are, of, to, in - 中文：的, 了, 吗, 呢, 吧

为什么要去掉停用词?

```
文档1: "the cat is on the mat"
文档2: "the dog is on the mat"

如果不去掉"the, is, on":
共同词: {the, is, on, mat} = 4个
看起来很相似!

去掉停用词后:
文档1: {cat, mat}
文档2: {dog, mat}
共同词: {mat} = 1个
更能看出差异!
```

代码逐行解释:

```
stops = set()  # 创建空集合（用set因为查找快）
```

```
if '-s' in opts:  # 如果用户指定了停用词文件
```

```
with open(opts['-s'], 'r') as stop_fs:
# opts['-s'] 是停用词文件的路径
# 'r' 表示只读模式
# stop_fs 是文件对象
```

```
for line in stop_fs:  # 读取每一行
    stops.add(line.strip())  # 去掉换行符，加入集合
```

`line.strip()` 的作用：

```
line = "the\n"  # 文件中每行结尾有换行符
line.strip()    # "the"  (去掉了\n)
```

第3部分：统计词频（核心！）

```
def count_words(filename, stops):
    wordRE = re.compile('[A-Za-z]+')
    counts = {}
    with open(filename, 'r') as infile:
        for line in infile:
            for word in wordRE.findall(line.lower()):
                if word not in stops:
                    if '-p' in opts:
                        word = stem_word(word)
                    if word in counts:
                        counts[word] += 1
                    else:
                        counts[word] = 1
    return counts
```

整体流程图

```
读取文件
  ↓
逐行处理
  ↓
提取单词（正则表达式）
  ↓
转小写
  ↓
检查是否是停用词
  ↓
（可选）词干提取
  ↓
统计词频
  ↓
返回字典
```

逐行详解

1. 创建正则表达式

```
wordRE = re.compile('[A-Za-z]+')
```

- 匹配：1个或多个字母
- 作用：提取单词，忽略标点符号

2. 创建空字典

```
counts = {}
```

- 用来存储：`{单词: 出现次数}`
- 例如：`{'hello': 3, 'world': 2}`

3. 打开文件

```
with open(filename, 'r') as infile:
```

4. 逐行读取

```
for line in infile:
```

- 假设文件内容：`Hello, World! Python is great.`
- 第1次循环：`line = "Hello, World!\n"`
- 第2次循环：`line = "Python is great.\n"`

5. 提取单词

```
for word in wordRE.findall(line.lower()):
```

拆解这一行（重要！）：

```
# 步骤1：转小写
line.lower()
# "Hello, World!" → "hello, world!"

# 步骤2：用正则表达式提取单词
wordRE.findall("hello, world!")
# 返回列表：['hello', 'world']

# 步骤3：遍历每个单词
for word in ['hello', 'world']:
```

6. 检查停用词

```
if word not in stops:
```

- 如果 `word` 不在停用词列表，继续处理
- 否则跳过（不统计这个词）

7. 词干提取（可选）

```
if '-p' in opts:
    word = stem_word(word)
```

什么是词干提取？ - 把单词变成"词根"形式 - 例如： - running → run - played → play - better → better（不规则，可能不变）

为什么要词干提取？

```
文档1: I am running
文档2: I like to run

不提取词干:
文档1: {running}
文档2: {run}
共同词: 0个（不相似）

提取词干后:
文档1: {run}
文档2: {run}
共同词: 1个（相似！）
```

8. 统计词频

```
if word in counts:
    counts[word] += 1  # 已存在，次数+1
else:
    counts[word] = 1   # 第一次出现，设为1
```

执行过程示例：

```
# 假设处理句子："hello world hello"

# 第1个词："hello"
counts = {}
word = "hello"
word in counts?  → False
counts["hello"] = 1
# counts = {"hello": 1}

# 第2个词："world"
word = "world"
word in counts?  → False
counts["world"] = 1
# counts = {"hello": 1, "world": 1}

# 第3个词："hello"
word = "hello"
word in counts?  → True!
counts["hello"] += 1  # 1变成2
# counts = {"hello": 2, "world": 1}
```

第4部分：计算Jaccard相似度

```
def jaccard(doc1, doc2):
    wds1 = set(doc1)
    wds2 = set(doc2)
    if '-b' in opts:
        over  = len(wds1 & wds2)
        under = len(wds1 | wds2)
    else:
        over = under = 0
        for wd in (wds1 | wds2):
            if wd in doc1 and wd in doc2:
                over += min(doc1[wd], doc2[wd])
            wmax = 0
            if wd in doc1:
                wmax = doc1[wd]
            if wd in doc2:
                wmax = max(doc2[wd], wmax)
            under += wmax
    if under > 0:
        return over / under
    else:
        return 0.0
```

输入和输出

输入：

```
doc1 = {'hello': 2, 'world': 1, 'python': 1}
doc2 = {'hello': 1, 'world': 1, 'java': 1}
```

输出：

```
0.5  # 相似度50%
```

二进制模式（`-b` 选项）

忽略词频，只看有没有：

```
wds1 = set(doc1)  # {'hello', 'world', 'python'}
wds2 = set(doc2)  # {'hello', 'world', 'java'}
```

集合运算（Python语法）：

```
wds1 & wds2  # 交集（&符号）
# {'hello', 'world'}

wds1 | wds2  # 并集（|符号）
# {'hello', 'world', 'python', 'java'}
```

计算相似度：

```
over  = len(wds1 & wds2)  # 交集大小 = 2
under = len(wds1 | wds2)  # 并集大小 = 4

return over / under  # 2/4 = 0.5
```

带权重模式（默认）

考虑词频：

```
wds1 | wds2  # 所有不同的词
# {'hello', 'world', 'python', 'java'}

over = under = 0
```

遍历每个词：

词1："hello"

```
wd = "hello"

# 两个文档都有这个词吗?
if wd in doc1 and wd in doc2:  # True
    over += min(doc1[wd], doc2[wd])
    # over += min(2, 1) = 1

# 计算最大值
wmax = 0
if wd in doc1:
    wmax = doc1[wd]  # wmax = 2
if wd in doc2:
    wmax = max(doc2[wd], wmax)  # max(1, 2) = 2
under += wmax  # under += 2

# 现在: over=1, under=2
```

词2："world"

```
wd = "world"

# 都有
over += min(1, 1) = 1
# over = 1 + 1 = 2

wmax = max(1, 1) = 1
under += 1
# under = 2 + 1 = 3
```

词3："python"

```
wd = "python"

# doc1有，doc2没有
if wd in doc1 and wd in doc2:  # False
    # 不执行

wmax = 0
if wd in doc1:
    wmax = 1
if wd in doc2:  # False
    # 不执行
under += 1
# under = 3 + 1 = 4
```

词4："java"

```
wd = "java"

# doc1没有，doc2有
# 交集贡献 = 0

wmax = doc2["java"] = 1
under += 1
# under = 4 + 1 = 5
```

最终结果：

```
over = 2
under = 5
return 2 / 5 = 0.4
```

第5部分：计算所有文档对

```
results = {}
for i in range(len(docs)-1):
    for j in range(i+1, len(docs)):
        pair_name = '%s <> %s' % (filenames[i], filenames[j])
        results[pair_name] = jaccard(docs[i], docs[j])
```

为什么两层循环?

假设有3个文档：A, B, C

需要比较的对： - A vs B - A vs C - B vs C

不需要比较： - A vs A（自己跟自己） - B vs A（跟 A vs B 重复）

循环解释

```
docs = [doc_A, doc_B, doc_C]  # 3个文档
len(docs) = 3

# 外层循环：i = 0, 1
for i in range(len(docs)-1):  # range(2) → 0, 1
    # 内层循环：j从i+1开始
    for j in range(i+1, len(docs)):
```

执行过程：

```
i=0, j=1：比较 doc_A vs doc_B
i=0, j=2：比较 doc_A vs doc_C
i=1, j=2：比较 doc_B vs doc_C
```

图解：

```
     A   B   C
A    -   ✓   ✓
B        -   ✓
C            -

只计算上三角（不包括对角线）
```

第6部分：排序输出

```
top_N = 20
pairs = sorted(results, key=lambda v:results[v], reverse=True)
if top_N > 0:
    pairs = pairs[:top_N]

for pair in pairs:
    print('[%d] %s = %.3f' % (c, pair, results[pair]))
```

`sorted()` 函数详解

```
results = {
    'A <> B': 0.3,
    'A <> C': 0.7,
    'B <> C': 0.5
}

pairs = sorted(results, key=lambda v:results[v], reverse=True)
```

参数解释： - `results` - 要排序的字典 - `key=lambda v:results[v]` - 排序依据（按值排序） - `reverse=True` - 降序（从大到小）

`lambda` 是什么？

```
# lambda是匿名函数（简写函数）

# 普通写法：
def get_value(v):
    return results[v]
pairs = sorted(results, key=get_value)

# lambda简写：
pairs = sorted(results, key=lambda v:results[v])
```

排序过程：

```
原始:
A <> B: 0.3
A <> C: 0.7
B <> C: 0.5

排序后（降序）:
A <> C: 0.7  ← 最相似
B <> C: 0.5
A <> B: 0.3
```

取前N个

```
pairs = pairs[:top_N]  # 列表切片，取前20个
```

五、完整执行示例（手工模拟）

输入数据

文件1 (news01.txt):

```
Python is great. Python is easy.
```

文件2 (news02.txt):

```
Java is great. Java is powerful.
```

停用词 (stoplist.txt):

```
is
```

执行过程

步骤1：统计词频

处理文件1：

```
原文：Python is great. Python is easy.

1. 正则提取单词并转小写：
   ['python', 'is', 'great', 'python', 'is', 'easy']

2. 去掉停用词"is"：
   ['python', 'great', 'python', 'easy']

3. 统计词频：
   doc1 = {
       'python': 2,
       'great': 1,
       'easy': 1
   }
```

处理文件2：

```
原文：Java is great. Java is powerful.

统计结果：
doc2 = {
    'java': 2,
    'great': 1,
    'powerful': 1
}
```

步骤2：计算相似度

二进制模式 (`-b`)：

```
wds1 = {python, great, easy}
wds2 = {java, great, powerful}

交集 = {great}              → 1个
并集 = {python, great, easy, java, powerful} → 5个

相似度 = 1 / 5 = 0.2
```

带权重模式：

```
所有词：{python, great, easy, java, powerful}

"python": min(2,0)=0, max(2,0)=2  → over+=0, under+=2
"great":  min(1,1)=1, max(1,1)=1  → over+=1, under+=1
"easy":   min(1,0)=0, max(1,0)=1  → over+=0, under+=1
"java":   min(0,2)=0, max(0,2)=2  → over+=0, under+=2
"powerful": min(0,1)=0, max(0,1)=1 → over+=0, under+=1

over = 1
under = 7

相似度 = 1 / 7 ≈ 0.143
```

六、两种实现的对比

使用正则表达式 vs 不使用

方法1：用正则表达式

```
wordRE = re.compile('[A-Za-z]+')
for word in wordRE.findall(line.lower()):
    # 处理word
```

优点： - 代码简洁（1行） - 功能强大（可以匹配复杂模式）

提取结果：

```
"Hello, World! 123" → ['hello', 'world']
```

方法2：不用正则表达式

```
def remove_punctuation(text):
    return ''.join(char for char in text if char not in string.punctuation)

words = remove_punctuation(line).lower().split()
```

步骤：

```
1. 去标点："Hello, World!" → "Hello World"
2. 转小写："hello world"
3. 分割：['hello', 'world']
```

对比： | 特性 | 正则表达式 | 不用正则 | |------|-----------|---------| | 代码量 | 少 | 多 | | 速度 | 快 | 慢 | | 灵活性 | 高 | 低 | | 学习难度 | 中 | 简单 |

七、纸笔考试重点

必须掌握

1. ✅ **正则表达式基本符号**
2. `[A-Z]`, `[a-z]`, `[0-9]`
3. `+` (至少1次), `*` (0次或多次)
4. `[A-Za-z]+` 的含义
5. ✅ **Jaccard相似度计算**
6. 公式：交集 / 并集
7. 二进制模式：只看有没有
8. 带权重模式：考虑词频

9. 手工计算示例
10. ✅ **词频统计**
11. 用字典存储：`{单词: 次数}`
12. `if word in counts` 的判断
13. `counts[word] += 1` 的更新
14. ✅ **停用词的作用**
15. 为什么要去掉
16. 怎么去掉

可能的考题

题型1：正则表达式匹配

题目：用正则表达式 `[a-z]+` 匹配 "Hello123world"，结果是什么？

▶ **答案**

题型2：Jaccard计算

题目：

```
文档A: {apple: 2, banana: 1}
文档B: {apple: 1, orange: 1}

计算Jaccard相似度（带权重模式）
```

▶ **答案**

题型3：词频统计

题目：给定代码片段，说明执行后 `counts` 的值

```
counts = {}
words = ['hello', 'world', 'hello']

for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1
```

▶ 答案

八、记忆口诀

正则表达式

方括号范围定，
加号至少一，
星号可为零，
找词用加号。

Jaccard相似度

交集除并集，
相同比总数，
二进制看有无，
权重看多少。

词频统计

> 字典存词频，
> 存在就加一，
> 不存就设一，
> 循环把词分。

九、常见错误

❌ 错误1：混淆 `*` 和 `+`

```
# 错误理解
'[a-z]*' 匹配至少1个小写字母 ❌

# 正确理解
'[a-z]*' 匹配0个或多个小写字母 ✅
'[a-z]+' 匹配至少1个小写字母 ✅
```

❌ 错误2：忘记转小写

```
# 错误
wordRE.findall(line)
# "Hello" 和 "hello" 被认为是不同的词

# 正确
wordRE.findall(line.lower())
# 都变成 "hello"
```

❌ 错误3：Jaccard计算错误

```
# 错误：用加法
over = len(wds1) + len(wds2)  ❌

# 正确：用交集
over = len(wds1 & wds2)  ✅
```

十、实战练习

练习1：正则匹配

给定文本："Python3.9 is Great!"

用 `[A-Za-z]+` 匹配，结果是?

▶ **答案**

练习2：相似度计算

```
文档1：I love Python
文档2：I like Python

停用词：I

计算相似度（二进制模式）
```

▶ **答案**

祝你考试顺利！ 🎉

记住： - 正则表达式是工具，用来提取文本 - Jaccard是算法，用来计算相似度 - 理解原理比记代码更重要！