**实验报告**

实验名称：2.文本预处理-分词

实验目的：使用最大正向匹配算法和最大反向匹配算法，将中文文本尽可能多的分割为词典中的单词，并通过对（1）分词长度（2）分词总数（3）不在字典中的词数（4）在字典中的单字数量，四个指标进行优选，最终得出分词结果。

实验原理：

*#最大正向匹配算法与实现原理：*

*# （0）从文章开头寻找重要分隔符，分割短句*

*# （1）在当前短句内，取最大长度，并将所取词放入字典匹配*

*# （2）若匹配成功，则导出该词；若失败，则缩短长度*

*# （3）若词长为1，则该词不再字典中*

*# （4）跳转到下一词继续匹配，直到匹配完整个短句*

*# （5）匹配完成后，寻找下一个重要分隔符，对下一个短句进行匹配，直到文章结束*

实验过程：

import math

import re

data = []

1. 文件读入

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_1.文件读入\_\_\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

with open('/Users/jason/Documents/GitHub/Information\_Search/20210316\_ChineseSegmentation/Test\_Data/data\_partial\_utf8.txt', 'r+') as dataFile:

for line in dataFile.readlines():

if line != None:

data.append(line.strip('\n'))

1. 提取汉字和文本中的重要分隔符。

通过重要分隔符（此处包括“，”、“。”、“,”、“？”、“！”、“、”、“；”），把文段分割成短句，通过移动指针，在短句内实现最大匹配。（重要分隔符天然通过语义将文段分割，是提高分词准确率的基础）

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_2.提取汉字和重要分隔符\_\_\*\*\*\*\*\*\*\*\*\*\*\*\**

dataString = ''.join(data)

charactor = re.compile('[\u4e00-\u9fa5\，\。\,\?\!\、\;]')

dataChineseOnly = charactor.findall(dataString)

1. 检索字典，找到最大长度的词的长度，将该长度作为所取词的初始长度。

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_3.检索字典词语最大长度\_\_\*\*\*\*\*\*\*\*\*\*\*\*\*\**

Dictionary = {}

MAXLENGTH = 0

with open('/Users/jason/Documents/GitHub/Information\_Search/20210316\_ChineseSegmentation/Test\_Data/dict\_utf8.txt', 'r+') as dictFile:

for line in dictFile.readlines(): *#按行读入*

if line != None:

*#构造词库字典*

Dictionary[line.strip('\n')] = Dictionary.get((line.strip('\n')), 1)

if len(line) > MAXLENGTH: *#找到最大长度，并记为最大匹配长度*

MAXLENGTH = len(line)

1. 开始正向最大匹配

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_4.正向最大匹配\_\_\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*#通过重要分隔符，把文段分割成短句，通过移动指针，在短句内实现最大匹配。（重要分隔符天然通过语义将文段分割，是提高分词准确率的基础）*

breakingPoint = -1 *#指向重要分隔符的位置*

paragraphLength = len(dataChineseOnly)

headPointer = 0 *#指向片段始端*

tailPointer = 0 *#指向片段末端*

Pointer = 0 *#寻找重要分隔符的位置*

forwordNotFound = {} *#未匹配单字（未在字典中查询到）*

forwordNotFoundTotal = 0 *#未匹配单字计数器*

segmentationBuffer\_0 = [] *#存放分割完成的字符串*

*#（0）找到间断点（重要分隔符）*

while breakingPoint < paragraphLength - 1: *#间断点在范围内*

headPointer = breakingPoint + 1 *#头指针指向上一个重要分隔符的下一位*

while Pointer < paragraphLength: *#在文章长度范围内，寻找下一个重要分隔符*

if (dataChineseOnly[Pointer] == '，') or (dataChineseOnly[Pointer] == '。') or (dataChineseOnly[Pointer] == ',') or (dataChineseOnly[Pointer] == '?') or (dataChineseOnly[Pointer] == '!') or (dataChineseOnly[Pointer] == ';') or (dataChineseOnly[Pointer] == '、'):

breakingPoint = Pointer

Pointer += 1

break

Pointer += 1

*#\*\*\*\*\*\*\*\*\*\*\*匹配单个词语（每次）\*\*\*\*\*\*\*\*\*\*\*\*\**

while headPointer < breakingPoint: *#保证该词在当前短句内*

tailPointer = headPointer + MAXLENGTH - 1 *#尾指针初始化指向最大长度处*

if tailPointer >= breakingPoint: *#越界判断*

tailPointer = breakingPoint - 1

flag = False *#标记该词是否可在字典中检索到*

while tailPointer >= headPointer :

segmentation = ''.join(dataChineseOnly[headPointer:tailPointer+1])

if segmentation in Dictionary : *#检索*

segmentationBuffer\_0.append(segmentation)

flag = True *#检索成功*

Break

else :

tailPointer -= 1 *#查找失败，减短长度继续查找*

if flag == False : *#检索失败*

segmentation = dataChineseOnly[headPointer]

segmentationBuffer\_0.append(segmentation)

forwordNotFound[segmentation] = forwordNotFound.get(segmentation) *#将单字加入未知列表*

forwordNotFoundTotal += 1 *#检索失败计数器*

tailPointer += 1 *#此时的尾指针在头指针之前一位，通过加一实现复位*

headPointer = tailPointer + 1 *#头指针指向下一个词的开始位置*

*#如此循环，直到该短句内所有词语都匹配完成*

segmentationBuffer\_0.append(dataChineseOnly[breakingPoint])

1. 正向匹配结果文件写入

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

fileName = dataFile.name

dot = re.compile('\.')

fileName = re.sub(dot, '\_Forword.', fileName)

result = '/'.join(segmentationBuffer\_0)

resultFile = open(fileName, 'w')

resultFile.write(result)

1. 反向最大匹配

原理：

*#反向最大匹配算法与实现原理：*

*# （0）从文章结尾寻找重要分隔符，分割短句*

*# （1）在当前短句内，取最大长度，并将所取词放入字典匹配*

*# （2）若匹配成功，则导出该词；若失败，则缩短长度*

*# （3）若词长为1，则该词不再字典中*

*# （4）跳转到上一词继续匹配，直到匹配完整个短句*

*# （5）匹配完成后，寻找上一个重要分隔符，对上一个短句进行匹配，直到文章结束*

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_5.反向最大匹配\_\_\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

breakingPoint = paragraphLength - 1 *#指向重要分隔符的位置*

paragraphLength = len(dataChineseOnly)

headPointer = breakingPoint - 1 *#指向片段始端*

tailPointer = breakingPoint - 1 *#指向片段末端*

Pointer = paragraphLength - 2 *#寻找重要分隔符的位置*

reverseNotFound = {} *#未匹配单字（未在字典中查询到）*

reverseNotFoundTotal = 0 *#未匹配单字计数器*

segmentationBuffer\_1 = [] *#存放分割完成的字符串*

*#（0）找到间断点（重要分隔符）*

while breakingPoint > 0 : *#间断点在范围内*

tailPointer = breakingPoint - 1 *#头指针指向上一个重要分隔符的上一位*

while Pointer > 0: *#在文章长度范围内，寻找上一个重要分隔符*

if (dataChineseOnly[Pointer] == '，') or (dataChineseOnly[Pointer] == '。') or (dataChineseOnly[Pointer] == ',') or (dataChineseOnly[Pointer] == '?') or (dataChineseOnly[Pointer] == '!') or (dataChineseOnly[Pointer] == ';') or (dataChineseOnly[Pointer] == '、'):

breakingPoint = Pointer

segmentationBuffer\_1.append(dataChineseOnly[breakingPoint])

Pointer -= 1

break

Pointer -= 1

if Pointer == 0:

breakingPoint = -1

*#\*\*\*\*\*\*\*\*\*\*\*匹配单个词语（每次）\*\*\*\*\*\*\*\*\*\*\*\*\**

while tailPointer > breakingPoint: *#保证该词在当前短句内*

headPointer = tailPointer - MAXLENGTH + 1 *#尾指针初始化指向最大长度处*

if headPointer <= breakingPoint: *#越界判断*

headPointer = breakingPoint + 1

flag = False *#标记该词是否可在字典中检索到*

while headPointer <= tailPointer :

segmentation = ''.join(dataChineseOnly[headPointer:tailPointer+1])

if segmentation in Dictionary : *#检索*

segmentationBuffer\_1.append(segmentation)

flag = True *#检索成功*

Break

else :

headPointer += 1 *#查找失败，减短长度继续查找*

if flag == False : *#检索失败*

segmentation = dataChineseOnly[tailPointer]

segmentationBuffer\_1.append(segmentation)

reverseNotFound[segmentation] = reverseNotFound.get(segmentation) *#将单字加入未知列表*

reverseNotFoundTotal += 1 *#检索失败计数器*

headPointer -= 1 *#此时的尾指针在头指针之前一位，通过加一实现复位*

tailPointer = headPointer - 1 *#头指针指向下一个词的开始位置*

*#如此循环，直到该短句内所有词语都匹配完成*

1. 反向最大匹配结果文件写入：

*##\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

fileName = dataFile.name

dot = re.compile('\.')

fileName = re.sub(dot, '\_Reverse.', fileName)

segmentationBuffer\_1 = segmentationBuffer\_1[::-1]

result = '/'.join(segmentationBuffer\_1)

resultFile = open(fileName, 'w')

resultFile.write(result)

1. 数据比对

*#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\_\_6.数据比对\_\_\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

print('forwordNotFoundTotal: {*:,*}\nreverseNotFoundTotal: {*:,*}\nforwordFoundTotal: {*:,*}\nreverseFoundTotal: {*:,*}\nforwordAverageLength: {*:.6f*}\nreverseAverageLength: {*:.6f*}\n'.format(forwordNotFoundTotal,reverseNotFoundTotal,forwordFoundTotal,reverseFoundTotal,forwordParaLength/forwordFoundTotal,reversePrarLength/reverseFoundTotal))

实验结果：

I）总的来说，正向匹配和反向匹配都跑出了结果，对比一下数据：

正向匹配不在字典中的单字forwordNotFoundTotal: 3,299

反向匹配不在字典中的单字reverseNotFoundTotal: 3,273

正向匹配查找成功的总词数forwordFoundTotal: 6,140

反向匹配查找成功的总词数reverseFoundTotal: 6,168

正向匹配平均词长forwordAverageLength: 2.187948

反向匹配平均词长reverseAverageLength: 2.182231

综合来看，反向匹配略微优于正向匹配，性能差异为样本量的0.75%。正向匹配在平均词长方面表现优于反向匹配；反向匹配在成功总次数上优于正向匹配。

II）存在问题：

1. 虽然正向匹配和反向匹配性能方面有些许差异，但通过实验发现，决定匹配成功率最根本的条件是给定字典的全面性。给定字典中收录的词语越多，匹配成功率和准确性就越高，反之则低，而且通过优化词典所带来的匹配性能的提高，是单方面优化算法所不能达到的。有如下举例：/股/神、/首/次、/轿/跑；等常用词不在字典中，会导致匹配效率和准确性大大降低。
2. 正向匹配在一些条件下会出现匹配错位的情况：例如“元人民币”，正向匹配为：元人/民/币/；反向匹配为：/元/人民币/。“深兰科技”，正向匹配为：/深/兰科/技 ；反向匹配为：深/兰/科技。正向错位，而反向匹配成功。
3. 对于专有名词（如姓名，公司名，书名等），期待的结果是将专有名词逐字分割，而不划归为字典中的词语，实验发现反向匹配更能达到预期。
4. 对于名词前的单字形容词，动词前的单字副词，表示时态的单字词，匹配时会以单字划分，但是这些单字往往对修饰词的作用很大，对词语的理解，和动词的时态起关键作用，不应该被独立分割。如：“老股东”，/老/股东/；“下滑了”，/下滑/了/；“被称为”，/被/称为/。

实验者：赵恩正

学号：2027406081

日期：March 20, 2021