4.中文分词&新词发现

1. 正向最大匹配

a. 步骤

1. 收集一个词表

2. 对于一个待分词的字符串，从前向后寻找最长的，在此表中出现的词，在词边界做切分

3. 从切分处重复步骤2，直到字符串末尾

- 词表：

北京	生前
北京大学	前来
大学	报到
大学生	

句子：

北京大学生前来报到

北京大学 / 生前 / 来 / 报到

i. 实现方式一

- 1.找出词表中最大词长度
- 2.从字符串开头开始选取最大词长度的窗口，检查窗口内的词是否在词表中
- 3.如果在词表中，在词边界处进行切分，之后移动到词边界处，重复步骤2
- 4.如果不在词表中，窗口右边界回退一个字符，之后检查窗口词是否在词表中

ii. 实现方式2

- 1.从前向后进行查找
- 2.如果窗口内的词是一个词前缀则继续扩大窗口
- 3.如果窗口内的词不是一个词前缀，则记录已发现的词，并将窗口移动到词边界

2. 反向最大匹配

与正向最大匹配类似，但从后向前进行匹配

3. 双向最大匹配

同时进行正向和反向最大匹配，并比较二者结果

4. 分词的评价标准

- *如何比较?*
- 1.单字词
- 词表中可以有单字，从分词的角度，我们也会把它称为一个词
- 2.非字典词
- 未在词表中出现过的词，一般都会被分成单字
- 3.词总量
- 不同切分方法得到的词数可能不同

- 计算哪种切分方式总词频最高
- 词频事先根据分词后语料统计出来

5. 基于词表的分词缺点

1. 对词表极为依赖，如果没有词表，则无法进行；如果词表中缺少需要的词，结果也不会正确
2. 切分过程中不会关注整个句子表达的意思，只会将句子看成一个个片段
3. 如果文本中出现一定的错别字，会造成一连串影响
4. 对于人名等的无法枚举实体词无法有效的处理

6. 基于机器学习的分词

- 对于每一个字，我们想知道它是不是一个词的边界

- 上 | 海 | 自 | 来 | 水 | 来 | 自 | 海 | 上
- 0 1 0 0 1 0 1 1 1
- 蓝色表示不是词边界，红色表示是词边界

我们只需要得到一个与字符串等长的0，1序列

7. 其他

- 自定义Dataset最关键是重写__len__和__getitem__两个函数
- RNN中已经带有tanh激活函数，并不需要再定义激活函数

8. 新词发现

如YYDS、南方小土豆、泼天的富贵

- 假设没有词表，如何从文本中发现新词？
- 随着时间推移，新词会不断出现，固有词表会过时
- 补充词表有利于下游任务
- 词相当于一种固定搭配
- 词的内部应该是稳固的
- 内部凝固度 $\frac{1}{n}\log\frac{p(W)}{p(c_1)\cdots p(c_n)}$
- 词的外部应该是多变的
- 左右熵 $H(U)=E[-\log p_i]=-\sum_{i=1}^{n}p_i\log p_i$

1. $p(W)$ 表示该词的出现概率，$p(c_1)$ 表示词中的字出现的概率，**如“魑魅魍魉”，作为单个字是十分少见的（概率低），但作为整个词概率则很高**，因此内部凝固度高
2. $H(U)$ 表示左右熵，描述的是词左右的混乱程度，p_i 表示该词左边出现的字的概率（因为我们假设不进行分词，**如“但是”，放在广泛的场景都可以使用，左右搭配可以很混乱**，因此也是外部是多变的。

作为新词发现时，将上面两个指标做加权，这是为数不多的**无监督**任务

新词发现 Python

```python
import math
from collections import defaultdict

class NewWordDetect:
    def __init__(self, corpus_path):
        self.max_word_length = 5
        # 记录词频
        self.word_count = defaultdict(int)
        # 记录词左邻词频，嵌套的dict
        #  key              value
        #  word             dict
        #           key             value
        #         leftWord          count
        self.left_neighbor = defaultdict(dict)
        # 记录词右邻词频，同上
        self.right_neighbor = defaultdict(dict)
        self.load_corpus(corpus_path)
        self.calc_pmi()
        self.calc_entropy()
        self.calc_word_values()


    #加载语料数据，并进行统计
    def load_corpus(self, path):
        with open(path, encoding="utf8") as f:
            for line in f:
                sentence = line.strip()
                for word_length in range(1, self.max_word_length):
                    self.ngram_count(sentence, word_length)
        return

    #按照窗口长度取词,并记录左邻右邻
    def ngram_count(self, sentence, word_length):
        for i in range(len(sentence) - word_length + 1):
            word = sentence[i:i + word_length]
            self.word_count[word] += 1
            # 如果大于1，说明有左邻
            if i - 1 >= 0:
                char = sentence[i - 1]
                self.left_neighbor[word][char] = self.left_neighbor[word].get(char, 0) + 1
            # 如果小于length，说明有右邻
            if i + word_length < len(sentence):
                char = sentence[i +word_length]
```

```
                self.right_neighbor[word][char] = self.right_neighbor[word].get(char, 0) + 1
        return

    #计算熵
    def calc_entropy_by_word_count_dict(self, word_count_dict):
        total = sum(word_count_dict.values())
        entropy = sum([-(c / total) * math.log((c / total), 10) for c in word_count_dict.values()])
        return entropy

    #计算左右熵
    def calc_entropy(self):
        self.word_left_entropy = {}
        self.word_right_entropy = {}
        for word, count_dict in self.left_neighbor.items():
            self.word_left_entropy[word] = self.calc_entropy_by_word_count_dict(count_dict)
        for word, count_dict in self.right_neighbor.items():
            self.word_right_entropy[word] = self.calc_entropy_by_word_count_dict(count_dict)


    #统计每种词长下的词总数
    def calc_total_count_by_length(self):
        self.word_count_by_length = defaultdict(int)
        # word表示词，count表示该词出现的个数
        for word, count in self.word_count.items():
            self.word_count_by_length[len(word)] += count
        return

    #计算互信息(pointwise mutual information)
    def calc_pmi(self):
        self.calc_total_count_by_length()
        self.pmi = {}
        # 词频是在对应词长度下单独计算的
        for word, count in self.word_count.items():
            p_word = count / self.word_count_by_length[len(word)]
            p_chars = 1
            # 统计词中的字频乘积
            for char in word:
                p_chars *= self.word_count[char] / self.word_count_by_length[1]
            self.pmi[word] = math.log(p_word / p_chars, 10) / len(word)
        return

    def calc_word_values(self):
```

```
        self.word_values = {}
        for word in self.pmi:
            if len(word) < 2 or ", " in word:
                continue
            pmi = self.pmi.get(word, 1e-3)
            le = self.word_left_entropy.get(word, 1e-3)
            re = self.word_right_entropy.get(word, 1e-3)
            # 这里直接乘积并不好，因为左右混论程度会相互抵消，应该取最小值，才能消除左右两端的相互影响
            self.word_values[word] = pmi * min(le, re)

if __name__ == "__main__":
    nwd = NewWordDetect("sample_corpus.txt")
    # print(nwd.word_count)
    # print(nwd.left_neighbor)
    # print(nwd.right_neighbor)
    # print(nwd.pmi)
    # print(nwd.word_left_entropy)
    # print(nwd.word_right_entropy)
    # 对每个词的评分进行排序，并分别输出长度为2，3，4的前10个词
    value_sort = sorted([(word, count) for word, count in nwd.word_values.items()], key=lambda x:x[1], reverse=True)
    print([x for x, c in value_sort if len(x) == 2][:10])
    print([x for x, c in value_sort if len(x) == 3][:10])
```

corpus：语料库

9. 重要词评价

- 假如一个词在某类文本（假设为A类）中出现次数很多，而在其他类别文本（非A类）出现很少，那么这个词是A类文本的重要词（高权重词）。

恒星、黑洞 ——→ 天文

- 反之，如果一个词在出现在很多领域，则其对于任意类别的重要性都很差。

	政治？	？
中国	地理？	你好？
	经济？	？
	足球？	？

NLP中经典统计值：$TF \cdot IDF$

a. TF 词频

$TF =$ 某个词在某类别中出现的次数/该类别词总数

b. IDF 逆文档频率

$$IDF = \log\left(\frac{\text{语料库的文档总数}}{\text{包含该词的文档数}+1}\right)$$

+1是为了防止分母为0。

$TF \cdot IDF$ 高代表该领域重要程度高！

Python

```python
import jieba
import math
import os
import json
from collections import defaultdict

"""
tfidf的计算和使用
"""


# 统计tf和idf值
def build_tf_idf_dict(corpus):
    tf_dict = defaultdict(dict)
    #   key         value
    # 文档序号        dict
    #           key        value
    #            词       词出现的次数
    idf_dict = defaultdict(set)
    #   key         value
    #    词          set : 文档序号，最终用于计算每个词在多少篇文档中出现过
    for text_index, text_words in enumerate(corpus):
        for word in text_words:
            if word not in tf_dict[text_index]:
                tf_dict[text_index][word] = 0
            tf_dict[text_index][word] += 1
            idf_dict[word].add(text_index)
    # 计算每个词在多少个文档中出现过
    idf_dict = dict([(key, len(value)) for key, value in idf_dict.items
()])
    return tf_dict, idf_dict


# 根据tf值和idf值计算tfidf
def calculate_tf_idf(tf_dict, idf_dict):
    tf_idf_dict = defaultdict(dict)
    #   key         value
    #  文档序号       dict
    #            key    value
    #            词      TF·IDF值
    for text_index, word_tf_count_dict in tf_dict.items():
        for word, tf_count in word_tf_count_dict.items():
            # 每个词所在文档的词频
            tf = tf_count / sum(word_tf_count_dict.values())
```

```
            # tf-idf = tf * log(D/(idf + 1))
            tf_idf_dict[text_index][word] = tf * math.log(len(tf_dict) / (idf_dict[word] + 1))
    return tf_idf_dict


# 输入语料 list of string
def calculate_tfidf(corpus):
    # 先进行分词，自动输出分词结果，并用","作为间隔
    corpus = [jieba.lcut(text) for text in corpus]
    tf_dict, idf_dict = build_tf_idf_dict(corpus)
    tf_idf_dict = calculate_tf_idf(tf_dict, idf_dict)
    return tf_idf_dict


# 根据tfidf字典，显示每个领域topK的关键词
def tf_idf_topk(tfidf_dict, paths=[], top=10, print_word=True):
    topk_dict = {}
    for text_index, text_tfidf_dict in tfidf_dict.items():
        # dict.items
        # 这里的key相当于java中Arrays.sort的比较器，这里是根据第二个维度的元素(词的TF·IDF值)进行比较
        word_list = sorted(text_tfidf_dict.items(), key=lambda x: x[1], reverse=True)
        topk_dict[text_index] = word_list[:top]
        if print_word:
            print(text_index, paths[text_index])
            for i in range(top):
                print(word_list[i])
            print("----------")
    return topk_dict


def main():
    dir_path = r"category_corpus/"
    corpus = []
    paths = []
    for path in os.listdir(dir_path):
        path = os.path.join(dir_path, path)
        if path.endswith("txt"):
            corpus.append(open(path, encoding="utf8").read())
            paths.append(os.path.basename(path))
    tf_idf_dict = calculate_tfidf(corpus)
    tf_idf_topk(tf_idf_dict, paths)


if __name__ == "__main__":
```

```
88      main()
```

c. 特点

- 1.tf-idf的计算非常依赖分词结果，如果分词出错，统计值的意义会大打折扣
- 2.每个词，对于每篇文档，有不同的tf-idf值，所以不能脱离数据讨论tfidf
- 3.假如只有一篇文本，不能计算tf-idf
- 4.类别数据均衡很重要
- 5.容易受各种特殊符号影响，最好做一些预处理

d. 应用场景

- **搜索引擎**
  - 1.对于已有的所有网页（文本），计算每个网页中，词的TFIDF值
  - 2.对于一个输入query进行分词
  - 3.对于文档D，计算query中的词在文档D中的TFIDF值总和，作为query和文档的相关性得分

搜索引擎 Python

```python
import jieba
import math
import os
import json
from collections import defaultdict
from calculate_tfidf import calculate_tfidf, tf_idf_topk

"""
基于tfidf实现简单搜索引擎
"""

jieba.initialize()


# 加载文档数据（可以想象成网页数据），计算每个网页的tfidf字典
def load_data(file_path):
    corpus = []
    with open(file_path, encoding="utf8") as f:
        documents = json.loads(f.read())
        for document in documents:
            # 将标题和内容进行拼接，这里相当于标题和内容看成一样的权重
            corpus.append(document["title"] + "\n" + document["content"])
        tf_idf_dict = calculate_tfidf(corpus)
    return tf_idf_dict, corpus
    #   key         value
    #  文档序号       dict
    #            key    value
    #            词      TF·IDF值


def search_engine(query, tf_idf_dict, corpus, top=3):
    # 先对要搜索的内容进行分词
    query_words = jieba.lcut(query)
    res = []
    # 遍历所有文档，统计得分
    for doc_id, tf_idf in tf_idf_dict.items():
        score = 0
        # 对要搜索的字符串中的每个词在某个文档内TF·IDF值的加和
        for word in query_words:
            score += tf_idf.get(word, 0)
        # 该字符串在每个文档的TF·IDF值
        res.append([doc_id, score])
    # 对每个文档的TF·IDF值进行排序
    res = sorted(res, reverse=True, key=lambda x: x[1])
```

```
        for i in range(top):
            doc_id = res[i][0]
            print(corpus[doc_id])
            print("--------------")
        return res


if __name__ == "__main__":
    # 包含标题，内容两个key
    path = "news.json"
    tf_idf_dict, corpus = load_data(path)
    while True:
        query = input("请输入您要搜索的内容:")
        search_engine(query, tf_idf_dict, corpus)

```

- **文本摘要**

  - 1.通过计算TFIDF值得到每个文本的关键词。
  - 2.将包含关键词多的句子，认为是关键句。
  - 3.挑选若干关键句，作为文本的摘要。

摘要生成 Python

```python
import jieba
import math
import os
import random
import re
import json
from collections import defaultdict
from calculate_tfidf import calculate_tfidf, tf_idf_topk

"""
基于tfidf实现简单文本摘要
"""

jieba.initialize()


# 加载文档数据（可以想象成网页数据），计算每个网页的tfidf字典
def load_data(file_path):
    corpus = []
    with open(file_path, encoding="utf8") as f:
        documents = json.loads(f.read())
        for document in documents:
            assert "\n" not in document["title"]
            assert "\n" not in document["content"]
            corpus.append(document["title"] + "\n" + document["content"])
        tf_idf_dict = calculate_tfidf(corpus)
    return tf_idf_dict, corpus


# 计算每一篇文章的摘要
# 输入该文章的tf_idf词典，和文章内容
# top为人为定义的选取的句子数量
# 过滤掉一些正文太短的文章，因为正文太短在做摘要意义不大
def generate_document_abstract(document_tf_idf, document, top=3):
    sentences = re.split("? |! |。", document)
    # 过滤掉正文在五句以内的文章
    if len(sentences) <= 5:
        return None
    result = []
    for index, sentence in enumerate(sentences):
        sentence_score = 0
        words = jieba.lcut(sentence)
        for word in words:
            # 统计word在该文档内的TF·IDF值
```

```
            sentence_score += document_tf_idf.get(word, 0)
        # 做一个归一化，否则越长的句子的得分会越高
        sentence_score /= (len(words) + 1)
        result.append([sentence_score, index])
    result = sorted(result, key=lambda x: x[0], reverse=True)
    # 权重最高的可能依次是第10，第6，第3句，将他们调整为出现顺序比较合理，即3,6,10
    # 选得分最高的top个句子，根据index再排一次序
    important_sentence_indexs = sorted([x[1] for x in result[:top]])
    return "。".join([sentences[index] for index in important_sentence_indexs])


# 生成所有文章的摘要
def generate_abstract(tf_idf_dict, corpus):
    res = []
    for index, document_tf_idf in tf_idf_dict.items():
        title, content = corpus[index].split("\n")
        abstract = generate_document_abstract(document_tf_idf, content)
        if abstract is None:
            continue
        corpus[index] += "\n" + abstract
        res.append({"标题": title, "正文": content, "摘要": abstract})
    return res


if __name__ == "__main__":
    path = "news.json"
    tf_idf_dict, corpus = load_data(path)
    res = generate_abstract(tf_idf_dict, corpus)
    writer = open("abstract.json", "w", encoding="utf8")
    # ensure_ascii 设置为 False，以便生成更易读的 JSON 字符串，不用强行转换为非ascii字符
    writer.write(json.dumps(res, ensure_ascii=False, indent=2))
    writer.close()

```

- **文本相似度**

- 对所有文本计算tfidf后，从每个文本选取tfidf较高的前n个词，得到一个词的集合S。
- 对于每篇文本D，计算S中的每个词的词频，将其作为文本的向量。
- 通过计算向量夹角余弦值，得到向量相似度，作为文本的相似度

- 向量夹角余弦值计算：

$$\cos\theta = \frac{\sum_{i=1}^{n}(A_i \times B_i)}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}} = \frac{A \cdot B}{|A| \times |B|}$$

- 代码演示*

```python
#coding:utf8
import jieba
import math
import os
import json
from collections import defaultdict
from calculate_tfidf import calculate_tfidf, tf_idf_topk

"""
基于tfidf实现文本相似度计算
"""

jieba.initialize()

#加载文档数据（可以想象成网页数据），计算每个网页的tfidf字典
#之后统计每篇文档重要在前10的词，统计出重要词词表
#重要词词表用于后续文本向量化
def load_data(file_path):
    corpus = []
    with open(file_path, encoding="utf8") as f:
        documents = json.loads(f.read())
        for document in documents:
            corpus.append(document["title"] + "\n" + document["content"])
    tf_idf_dict = calculate_tfidf(corpus)
    # 每个文取前5个词构成词表
    topk_words = tf_idf_topk(tf_idf_dict, top=5, print_word=False)
    vocab = set()
    for words in topk_words.values():
        for word, _ in words:
            vocab.add(word)
    print("词表大小：", len(vocab))
    return tf_idf_dict, list(vocab), corpus


#passage是文本字符串
#vocab是词列表
#向量化的方式：计算每个重要词在文档中的出现频率
def doc_to_vec(passage, vocab):
    vector = [0] * len(vocab)
    passage_words = jieba.lcut(passage)
    for index, word in enumerate(vocab):
        vector[index] = passage_words.count(word) / len(passage_words)
    return vector

```

```python
#先计算所有文档的向量
def calculate_corpus_vectors(corpus, vocab):
    corpus_vectors = [doc_to_vec(c, vocab) for c in corpus]
    return corpus_vectors

#计算向量余弦相似度
def cosine_similarity(vector1, vector2):
    x_dot_y = sum([x*y for x, y in zip(vector1, vector2)])
    sqrt_x = math.sqrt(sum([x ** 2 for x in vector1]))
    sqrt_y = math.sqrt(sum([x ** 2 for x in vector2]))
    if sqrt_y == 0 or sqrt_y == 0:
        return 0
    # 分母加上很小的数，防止出现除0
    return x_dot_y / (sqrt_x * sqrt_y + 1e-7)


#输入一篇文本，寻找最相似文本
def search_most_similar_document(passage, corpus_vectors, vocab):
    input_vec = doc_to_vec(passage, vocab)
    result = []
    for index, vector in enumerate(corpus_vectors):
        score = cosine_similarity(input_vec, vector)
        result.append([index, score])
    result = sorted(result, reverse=True, key=lambda x:x[1])
    return result[:4]


if __name__ == "__main__":
    path = "news.json"
    tf_idf_dict, vocab, corpus = load_data(path)
    corpus_vectors = calculate_corpus_vectors(corpus, vocab)
    passage = "魔兽争霸"
    for corpus_index, score in search_most_similar_document(passage, corpus_vectors, vocab):
        print("相似文章:\n", corpus[corpus_index].strip())
        print("得分：", score)
        print("--------------")

```

10. TF·IDF优势

- 1.可解释性好
- 可以清晰地看到关键词
- 即使预测结果出错，也很容易找到原因
- 2.计算速度快
- 分词本身占耗时最多，其余为简单统计计算
- 3.对标注数据依赖小
- 可以使用无标注语料完成一部分工作
- 4.可以与很多算法组合使用
- 可以看做是词权重

11. TF·IDF劣势

- 1.受分词效果影响大
- 2.词与词之间没有语义相似度
- 3.没有语序信息（词袋模型）
- 4.能力范围有限，无法完成复杂任务，如机器翻译和实体挖掘等
- 5.样本不均衡会对结果有很大影响
- 6.类内样本间分布不被考虑

2. 如“您好”和“你好”在TF·IDF中是两个完全不同的词
3. 没有序列关系，像Pooling一样
4. 在标注数据少的情况下，TF·IDF比较常用

12. 编程练习：

给定词表，给出所有可能的分词结果

- 感觉能用回溯

13. 代码分析

a. forward_segmentation_method1

前向分割第一种方式，通过最大词长度，回退搜索

b. forward_segmentation_method2

前向分割第二种方式，记录词是否是其他词的前缀或真词