



# 文本相似度

## 智能信息网络实验报告

班级：110

姓名：耿翊中

学号：2021213382

2024 年 3 月

## 目录

智能信息网络实验报告 .....	1
1. 实验要求 .....	3
2. 实验原理 .....	4
2.1 向量语义学 .....	4
2.2 稀疏向量表示 .....	4
2.3 稠密向量表示 .....	4
3. 实验设置 .....	5
3.1 预处理数据 .....	5
3.1.1 数据集选择 .....	5
3.1.2 读取数据 .....	5
3.2 稀疏向量方法 .....	5
3.2.1 提取关键词 .....	5
3.2.2 计算相似度方法 .....	6
3.3 稠密向量方法 .....	12
3.3.1 语料处理 .....	12
3.3.2 训练模型 .....	13
3.3.3 相似度计算 .....	14
3.4 性能比较 .....	15
4. 实验结果分析 .....	16
4.1 实验结果 .....	16
4.2 结果分析 .....	17
5. 扩展：大模型方法 .....	18
5.1 与之前方法的异同 .....	18
5.2 Bert 模型代码原理 .....	18
5.3 Bert 模型效果 .....	21
6. 总结与思考 .....	错误！未定义书签。

## 1. 实验要求

文本相似度

文本相似度旨在识别两段文本在语义上是否相似。文本相似度在自然语言处理领域是一个重要研究方向，同时在信息检索、新闻推荐、智能客服等领域都发挥重要作用，具有很高的商业价值。

## 2. 实验原理

使用向量语义学来进行文本相似度的比较。使用向量语义学进行文本相似度研究的实验原理基于将文本（可以是词、短语、句子或文档）表示为向量的思想。通过这种表示，文本的语义信息被编码在向量空间中，使得我们可以利用数学方法来量化和比较文本之间的相似度。

### 2.1 向量语义学

向量语义学的核心思想是将词表示为由词的分布导出的多维语义空间中的一个点。这些用于表示词的向量称为嵌入(embeddings)，这些嵌入可以是稠密的，如 word2vec 模型生成的，也可以是稀疏的，如基于 tf-idf 或 PPMI 的向量。

主要通过以下几个步骤来实现：

#### 1. 文本表示：

将文本转换为向量的过程是向量语义学的核心。这可以通过多种方法实现，包括但不限于词袋模型（Bag of Words, BoW）、TF-IDF、Word2Vec、GloVe 或 BERT 等。这些方法有的生成稀疏向量（如 BoW、TF-IDF），有的生成稠密向量（如 Word2Vec、GloVe）。

#### 2. 向量空间模型：

文本向量化后，每个文本（或词）在多维向量空间中占据一个点。相似的文本在这个空间中的距离较近，而不相似的文本距离较远。这种空间模型允许我们使用几何方法来评估文本之间的相似度。

#### 3. 相似度度量：

在将文本转换为向量后，可以通过计算向量之间的距离或相似度来衡量文本之间的相似性。常用的度量包括余弦相似度、欧氏距离、曼哈顿距离等。其中，余弦相似度是最常用的方法之一，因为它衡量的是向量间的角度差异，能够有效地反映语义相似度，且不受向量长度的影响。

### 2.2 稀疏向量表示

使用稀疏向量（如基于 TF-IDF 或 PPMI 的向量）：

优点：稀疏向量往往更容易理解和解释，因为它们直接与词频相关联。这些向量也往往需要更少的存储空间。

缺点：稀疏向量可能无法充分捕捉词之间的复杂语义关系，尤其是在高维空间中，许多方向可能并不重要。

### 2.3 稠密向量表示

使用 word2vec 的稠密向量：

优点：word2vec 生成的嵌入可以捕获丰富的语义信息和词之间的关系，例如同义词或上下文相似性。稠密向量通常能提供更精确的相似度评估。

缺点：这些嵌入需要大量的数据来训练，并且可能需要更多的计算资源和存储空间。此外，它们的结果可能不如稀疏向量那样直观易解。

## 3. 实验设置

### 3.1 预处理数据

#### 3.1.1 数据集选择

从 <https://tianchi.aliyun.com/dataset/106411?t=1710225696845> 获得了 AFQMC: 蚂蚁金融语义相似度数据集。

数据集分为 train.json, dev.json, test.json 三个, 本次实验只选用 dev.json 来测试不同模型之间的性能差异即可。

AFQMC: 蚂蚁金融语义相似度数据集

天池小喵喵 2021-07-15 3419 108 CC-BY-SA-NC 4.0

新建Notebook

内容 Notebook 评论

描述

AFQMC (Ant Financial Question Matching Corpus) : 蚂蚁金融语义相似度数据集, 该数据集由蚂蚁金服提供。

数据列表

数据名称	上传日期	大小	下载
dev.json	2021-07-15	538.52KB	<a href="#">↓</a>
test.json	2021-07-15	470.57KB	<a href="#">↓</a>
train.json	2021-07-15	4.19MB	<a href="#">↓</a>

#### 3.1.2 读取数据

从本地读取数据并存储在列表中供后续使用

```
1. texts = [] # 存储句子对和标签
2. with open('data/dev.json', 'r', encoding='utf-8') as file:
3.     for line in file:
4.         json_line = json.loads(line)
5.         # 直接从 json_line 中提取 sentence1, sentence2 和 label, 并转换为元组格式
6.         texts.append((json_line['sentence1'], json_line['sentence2'], int(json_line['label'])))
```

### 3.2 稀疏向量方法

#### 3.2.1 提取关键词

稀疏向量方法需要对每一段文本先进行分词和关键词提取的文本预处理, 然后使用这些关键词来计算文本之间的相似度。

```
1. def extract_keyword(content): # 提取关键词
2.     # 切割
3.     seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
4.     # 提取关键词
```

```
5.         keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=False)
6.         return keywords
```

### 3.2.2 计算相似度方法

#### 1. 余弦相似度 (CosineSimilarity)

创建一个词的“one-hot”编码，这是一种稀疏表示，其中每个维度代表一个词是否出现。  
计算两组词向量之间的余弦相似度。

```
1.     class CosineSimilarity(object):
2.
3.
4.         def __init__(self, content_x1, content_y2):
5.             self.s1 = content_x1
6.             self.s2 = content_y2
7.
8.         @staticmethod
9.         def extract_keyword(content): # 提取关键词
10.             # 切割
11.             seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
12.             # 提取关键词
13.             keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=False)
14.             return keywords
15.
16.         @staticmethod
17.         def one_hot(word_dict, keywords): # oneHot 编码
18.             # cut_code = [word_dict[word] for word in keywords]
19.             cut_code = [0] * len(word_dict)
20.             for word in keywords:
21.                 cut_code[word_dict[word]] += 1
22.             return cut_code
23.
24.         def main(self):
25.             # 去除停用词
26.             jieba.analyse.set_stop_words('data/stopwords.txt')
27.
28.             # 提取关键词
29.             keywords1 = self.extract_keyword(self.s1)
30.             keywords2 = self.extract_keyword(self.s2)
31.             # 词的并集
32.             union = set(keywords1).union(set(keywords2))
33.             # 编码
34.             word_dict = {}
35.             i = 0
```

```

36.         for word in union:
37.             word_dict[word] = i
38.             i += 1
39.         if len(word_dict) == 0:
40.             return 0.0
41.         # oneHot 编码
42.         s1_cut_code = self.one_hot(word_dict, keywords1)
43.         s2_cut_code = self.one_hot(word_dict, keywords2)
44.         # 余弦相似度计算
45.         sample = [s1_cut_code, s2_cut_code]
46.         # 除零处理
47.         try:
48.             sim = cosine_similarity(sample)
49.             return sim[1][0]
50.         except Exception as e:
51.             print(e)
52.         return 0.0

```

## 2. Jaccard 相似度 (JaccardSimilarity)

计算两组关键词集合的交集和并集。

使用交集和并集的大小来计算 Jaccard 相似度。

计算 Jaccard 相似度:

首先计算两组关键词集合的交集大小，即两段文本共同的关键词数量。

计算两组关键词集合的并集大小，即两段文本中所有不重复的关键词数量。

Jaccard 相似度就是交集大小除以并集大小的比例。这个比例表示两段文本在关键词层面的重合度，相似度越高，说明文本越相似。

```

1.     class JaccardSimilarity(object):
2.
3.
4.         def __init__(self, content_x1, content_y2):
5.             self.s1 = content_x1
6.             self.s2 = content_y2
7.
8.         @staticmethod
9.         def extract_keyword(content): # 提取关键词
10.            # 切割
11.            seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
12.            # 提取关键词
13.            keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=False)
14.            return keywords
15.
16.         def main(self):

```

```

17.         # 去除停用词
18.         jieba.analyse.set_stop_words('data/stopwords.txt')
19.
20.         # 分词与关键词提取
21.         keywords_x = self.extract_keyword(self.s1)
22.         keywords_y = self.extract_keyword(self.s2)
23.
24.         # jaccard 相似度计算
25.         intersection = len(list(set(keywords_x).intersection(set(keywords_y))))
26.         union = len(list(set(keywords_x).union(set(keywords_y))))
27.         # 除零处理
28.         sim = float(intersection) / union if union != 0 else 0
29.         return sim

```

### 3. 编辑距离 (LevenshteinSimilarity)

使用 Levenshtein 距离（一种基于字符的距离测度）来衡量两组关键词串联后字符串之间的相似度。

```

1.     class LevenshteinSimilarity(object):
2.
3.
4.         def __init__(self, content_x1, content_y2):
5.             self.s1 = content_x1
6.             self.s2 = content_y2
7.
8.         @staticmethod
9.         def extract_keyword(content): # 提取关键词
10.            # 切割
11.            seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
12.            # 提取关键词
13.            keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=False)
14.            return keywords
15.
16.         def main(self):
17.             # 去除停用词
18.             jieba.analyse.set_stop_words('data/stopwords.txt')
19.
20.             # 提取关键词
21.             keywords1 = ', '.join(self.extract_keyword(self.s1))
22.             keywords2 = ', '.join(self.extract_keyword(self.s2))
23.
24.             # ratio 计算 2 个字符串的相似度，它是基于最小编辑距离
25.             distances = Levenshtein.ratio(keywords1, keywords2)
26.             return distances

```



#### 4. MinHash 相似度 (MinHashSimilarity)

对关键词使用 MinHash 算法来估计它们集合的 Jaccard 相似度，这对于处理大型数据集特别有用。

MinHash 计算:

为每段文本创建一个 MinHash 对象。MinHash 对象对应于该文本的一系列哈希值，这些哈希值是通过将文本中的每个元素（在这里是关键词）应用一系列哈希函数得到的。

对于文本中的每个关键词，通过编码转换为字节串后，使用 `update` 方法更新对应 MinHash 对象。这个过程实际上是在计算每个关键词的哈希值，并保留当前遇到的最小哈希值。

使用 `jaccard` 方法计算两个 MinHash 对象之间的 Jaccard 相似度。这个方法实际上是比较两个 MinHash 对象的哈希值，根据它们相匹配的哈希值的比例来估算原始数据集的 Jaccard 相似度。

```
1. class MinHashSimilarity(object):
2.
3.     def __init__(self, content_x1, content_y2):
4.         self.s1 = content_x1
5.         self.s2 = content_y2
6.
7.     @staticmethod
8.     def extract_keyword(content): # 提取关键词
9.         # 切割
10.        seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
11.        # 提取关键词
12.        keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=False)
13.        return keywords
14.
15.    def main(self):
16.        # 去除停用词
17.        jieba.analyse.set_stop_words('data/stopwords.txt')
18.
19.        # MinHash 计算
20.        m1, m2 = MinHash(), MinHash()
21.        # 提取关键词
22.        s1 = self.extract_keyword(self.s1)
23.        s2 = self.extract_keyword(self.s2)
24.
25.        for data in s1:
26.            m1.update(data.encode('utf8'))
27.        for data in s2:
28.            m2.update(data.encode('utf8'))
29.
```

30. `return m1.jaccard(m2)`

## 5. SimHash 相似度 (SimHashSimilarity)

生成关键词的 SimHash 值，这是另一种稀疏表示，但它减少了维度并尝试保持相似文档的哈希值相近。

比较两个 SimHash 值之间的汉明距离来估计相似度。

生成 SimHash 值：

对每个关键词，将其转换为一个 64 位的二进制字符串（使用 `get_bin_str` 方法）。

根据关键词的权重调整每一位的值：如果某一位是 1，则加上该关键词的权重；如果是 0，则减权重。

将所有关键词的贡献累加起来，然后对每一位进行汇总：如果最终的累加值大于 0，则该位的 SimHash 值为 1；否则为 0。

计算两个 SimHash 值的汉明距离：

汉明距离指的是两个等长字符串之间对应位置的不同字符的个数。在这里，它用于衡量两个 SimHash 值的差异。

计算两个文本的 SimHash 值之间的汉明距离，然后通过汉明距离来估算文本的相似度。具体来说，相似度可以通过  $1 - (\text{汉明距离} / \text{SimHash 值的长度})$  来计算。

```
1. class SimHashSimilarity(object):
2.
3.     def __init__(self, content_x1, content_y2):
4.         self.s1 = content_x1
5.         self.s2 = content_y2
6.
7.     @staticmethod
8.     def get_bin_str(source): # 字符串转二进制
9.         if source == "":
10.            return 0
11.        else:
12.            t = ord(source[0]) << 7
13.            m = 1000003
14.            mask = 2 ** 128 - 1
15.            for c in source:
16.                t = ((t * m) ^ ord(c)) & mask
17.            t ^= len(source)
18.            if t == -1:
19.                t = -2
20.            t = bin(t).replace('0b', '').zfill(64)[-64:]
21.            return str(t)
22.
23.     @staticmethod
```

```

24.     def extract_keyword(content): # 提取关键词
25.         # 切割
26.         seg = [i for i in jieba.cut(content, cut_all=True) if i != '']
27.         # 提取关键词
28.         keywords = jieba.analyse.extract_tags("|".join(seg), topK=4, withWeight=True)
29.         return keywords
30.
31.     def run(self, keywords):
32.         ret = []
33.         for keyword, weight in keywords:
34.             bin_str = self.get_bin_str(keyword)
35.             key_list = []
36.             for c in bin_str:
37.                 weight = math.ceil(weight)
38.                 if c == "1":
39.                     key_list.append(int(weight))
40.                 else:
41.                     key_list.append(-int(weight))
42.             ret.append(key_list)
43.         # 对列表进行"降维"
44.         rows = len(ret)
45.         cols = len(ret[0])
46.         result = []
47.         for i in range(cols):
48.             tmp = 0
49.             for j in range(rows):
50.                 tmp += int(ret[j][i])
51.             if tmp > 0:
52.                 tmp = "1"
53.             elif tmp <= 0:
54.                 tmp = "0"
55.             result.append(tmp)
56.         return "".join(result)
57.
58.     def main(self):
59.         # 去除停用词
60.         jieba.analyse.set_stop_words('data/stopwords.txt')
61.
62.         # 提取关键词
63.         s1 = self.extract_keyword(self.s1)
64.         s2 = self.extract_keyword(self.s2)
65.
66.         sim_hash1 = self.run(s1)
67.         sim_hash2 = self.run(s2)

```

```

68.         # print(f'相似哈希指纹 1: {sim_hash1}\n 相似哈希指纹 2: {sim_hash2}')
69.         length = 0
70.         for index, char in enumerate(sim_hash1):
71.             if char == sim_hash2[index]:
72.                 continue
73.             else:
74.                 length += 1
75.         return length

```

### 3.3 稠密向量方法

使用 word2vec 方法，转换为稠密向量，生成的嵌入可以捕获丰富的语义信息和词之间的关系

#### 3.3.1 语料处理

这部分代码负责准备和预处理用于训练 Word2Vec 模型的语料。它主要执行以下操作：

**繁体转简体：**使用 langconv 库将文本从繁体中文转换为简体中文，以减少语言的多样性和复杂性，使模型更容易训练。

**分词：**使用 jieba 库对文本进行分词。中文是一种无空格分隔的语言，因此需要通过分词来识别单独的词。

**保存预处理后的文本：**将分词后的文本保存为一行一句的格式，每个词之间用空格分隔，为 Word2Vec 模型训练做准备。

这个过程通过读取维基百科的压缩 XML 备份，将每篇文章转换成一行人分词后的简体中文文本。

```

1.     # 中文语料预处理，繁体字转化为简体字
2.     from gensim.corpora import WikiCorpus
3.     from langconv import *
4.     import jieba
5.
6.
7.     def my_function():
8.         space = ' '
9.         i = 0
10.        l = []
11.        zhwiki_name = 'E:\selfcode(Externel)\zhwiki\zhwiki-latest-pages-articles.xml.bz2'
12.        f = open('reduce_zhiwiki.txt', 'w', encoding='utf-8')
13.        # xml 文件中读出训练语料
14.        wiki = WikiCorpus(zhwiki_name, dictionary={})
15.        for text in wiki.get_texts():
16.            for temp_sentence in text:
17.                # 繁体字转换为简体
18.                temp_sentence = Converter('zh-hans').convert(temp_sentence)
19.                # 分词

```

```

20.         seg_list = list(jieba.cut(temp_sentence))
21.         for temp_term in seg_list:
22.             l.append(temp_term)
23.         f.write(space.join(l) + '\n')
24.         l = []
25.         i = i + 1
26.
27.         if (i % 200 == 0):
28.             print('Saved ' + str(i) + ' articles')
29.         f.close()
30.
31.
32. if __name__ == '__main__':
33.     my_function()

```

### 3.3.2 训练模型

这部分代码负责使用预处理后的语料训练 Word2Vec 模型。具体步骤如下：

读取预处理文本：打开之前生成的分词文件。

模型设置和训练：使用 Gensim 库的 Word2Vec 类来训练模型。参数包括：

**sg**：定义模型的训练算法。0 表示使用 CBOW 算法，1 表示使用 Skip-gram 算法。

**window**：窗口大小，决定了词与其上下文词的最大距离。

**min\_count**：忽略出现次数少于此值的所有词。

**workers**：训练模型的线程数。

保存模型：训练完成后，模型被保存以供后续使用。

```

1.  # Word2vec 词向量模型训练
2.  # -*- coding: utf-8 -*-
3.  from gensim.models import Word2Vec
4.  from gensim.models.word2vec import LineSentence
5.  import logging
6.
7.  logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
8.
9.
10. def my_train_function():
11.     wiki_news = open('reduce_zhiwiki.txt', 'r', encoding='utf-8')
12.     model = Word2Vec(LineSentence(wiki_news), sg=0, window=5, min_count=5, workers=9)
13.     model.save('vectors.bin')
14.
15.
16. if __name__ == '__main__':
17.     my_train_function()

```

### 3.3.3 相似度计算

这部分代码使用训练好的 **Word2Vec** 模型来计算两个句子的相似度。具体步骤如下：

加载词向量模型：初始化时，从指定路径加载预训练的 **Word2Vec** 模型。

句子向量化：通过对句子中每个词的向量取平均来得到句子的向量表示。

计算相似度：使用余弦相似度来衡量两个句子向量之间的相似度。

返回相似度结果：计算出的相似度结果可以用于比较两个句子的相似程度。

```
1. import numpy as np
2. from gensim.models import KeyedVectors
3. import jieba
4. from sklearn.metrics.pairwise import cosine_similarity
5.
6.
7. class word2vec:
8.     def __init__(self, s1, s2, model_path='./word2vec/vectors.bin'):
9.         # 初始化时加载词向量模型
10.         self.model = KeyedVectors.load(model_path)
11.         self.s1 = s1
12.         self.s2 = s2
13.
14.     def wordavg(self, words):
15.         # 对句子中的每个词的词向量简单做平均作为句子的向量表示
16.         return np.mean([self.model.wv[word] for word in words if word in self.model.wv], axis=0)
17.
18.     def main(self):
19.         # 计算两个句子的相似度
20.         s1_list = jieba.lcut(self.s1, cut_all=True) # jieba 分词
21.         s2_list = jieba.lcut(self.s2, cut_all=True)
22.         s1_avg = self.wordavg(s1_list) # s1 的向量表示
23.         s2_avg = self.wordavg(s2_list) # s2 的向量表示
24.         try:
25.             result = cosine_similarity(s1_avg.reshape(1, -1), s2_avg.reshape(1, -1))
26.         except:
27.             return -1
28.         # print(result[0][0])
29.         return result[0][0]
30.
31. # 示例使用
32. # model_path = './word2vec/vectors.bin' # 设置模型路径
33. # sentence_similarity = SentenceSimilarity(model_path)
34. # s1 = "沙宣洗发水"
35. # s2 = "沙宣控油洗发水"
36. # print(sentence_similarity.word2vec_similarity(s1, s2)) # 输出两个句子的相似度
```

### 3.4 性能比较

设计用于评估不同文本相似度模型（包括稠密向量和稀疏向量方法）的性能。

由于之前稀疏向量的五种方法和稠密向量方法都被封装成了相同形式的类，所以可以直接统一调用，使用数据集 `dev.json` 来评估模型的效果。

可以比较不同文本相似度计算方法的性能，包括基于稠密向量的方法（如 `Word2Vec`）和基于稀疏向量的方法（如 `CosineSimilarity`、`JaccardSimilarity`、`LevenshteinSimilarity`、`MinHashSimilarity` 和 `SimHashSimilarity`）。这有助于确定哪种方法最适合您的特定应用或数据集。

#### 1. 数据准备：

代码首先从一个名为 `dev.json` 的文件中加载文本数据。这个文件应该包含多个 JSON 格式的记录，每条记录包含一对句子（`sentence1` 和 `sentence2`）及其相似度标签（`label`，通常是 0 或 1，表示句子是否相似）。

#### 2. 模型测试函数（`test_model`）：

`model_class`: 传入的相似度计算模型类，如 `CosineSimilarity` 或 `word2vec`。

`threshold`: 相似度阈值，用来判断两个句子是否相似（相似度高于此阈值的句子对被视为相似）。

函数遍历所有句子对，对每对句子使用指定的相似度模型计算相似度。

根据计算出的相似度和阈值判断句子对是否相似，并将此预测结果（0 或 1）存储在 `predictions` 列表中。

实际标签存储在 `actuals` 列表中。

使用 `accuracy_score` 从 `sklearn.metrics` 计算模型的准确率，即预测正确的句子对比例。

3. 模型评估：代码定义了一个 `model_classes` 列表，包含所有将要评估的模型类。然后，它遍历这个列表，对每个模型类调用 `test_model` 函数，并打印出每个模型的准确率。

4. 异常处理：在计算相似度时，如果发生任何异常（例如，当句子中的单词不在词向量模型的词汇表中时），该句子对会被跳过，程序继续处理下一对句子。

```
1. import numpy as np
2. from gensim.models import KeyedVectors
3. import jieba
4. from sklearn.metrics.pairwise import cosine_similarity
5.
6.
7. class word2vec:
8.     def __init__(self, s1, s2, model_path='./word2vec/vectors.bin'):
9.         # 初始化时加载词向量模型
10.         self.model = KeyedVectors.load(model_path)
11.         self.s1 = s1
12.         self.s2 = s2
13.
14.     def wordavg(self, words):
15.         # 对句子中的每个词的词向量简单做平均作为句子的向量表示
16.         return np.mean([self.model.wv[word] for word in words if word in self.model.wv], axis=0)
17.
18.     def main(self):
```

```

19.         # 计算两个句子的相似度
20.         s1_list = jieba.lcut(self.s1, cut_all=True) # jieba 分词
21.         s2_list = jieba.lcut(self.s2, cut_all=True)
22.         s1_avg = self.wordavg(s1_list) # s1 的向量表示
23.         s2_avg = self.wordavg(s2_list) # s2 的向量表示
24.         try:
25.             result = cosine_similarity(s1_avg.reshape(1, -1), s2_avg.reshape(1, -1))
26.         except:
27.             return -1
28.         # print(result[0][0])
29.         return result[0][0]
30.
31. # 示例使用
32. # model_path = './word2vec/vectors.bin' # 设置模型路径
33. # sentence_similarity = SentenceSimilarity(model_path)
34. # s1 = "沙宣洗发水"
35. # s2 = "沙宣控油洗发水"
36. # print(sentence_similarity.word2vec_similarity(s1, s2)) # 输出两个句子的相似度

```

## 4. 实验结果分析

### 4.1 实验结果

使用 dev.json 作测试后，五个稀疏向量方法的准确率分别为：

CosineSimilarity Accuracy: 0.5544485634847081

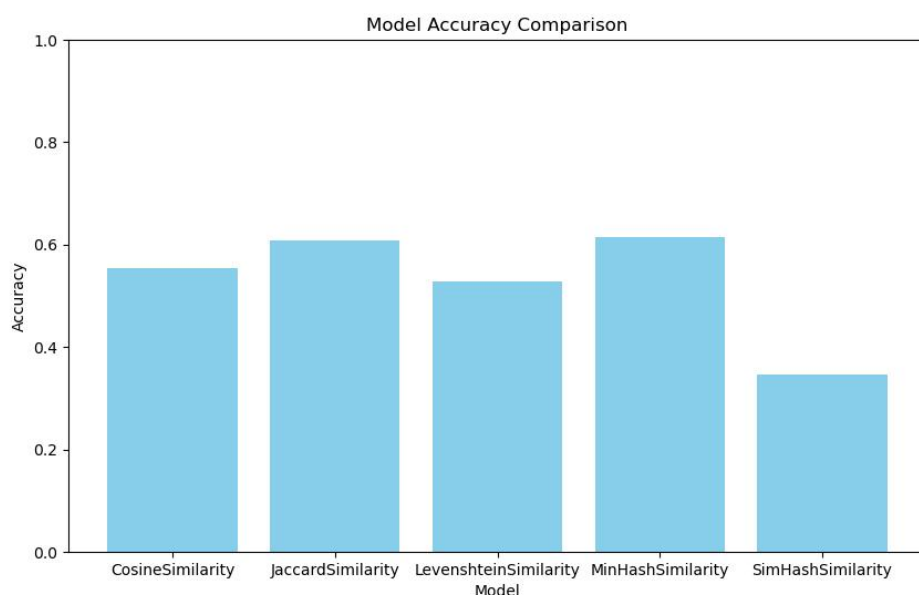
JaccardSimilarity Accuracy: 0.6091288229842446

LevenshteinSimilarity Accuracy: 0.5280352177942539

MinHashSimilarity Accuracy: 0.6146895273401297

SimHashSimilarity Accuracy: 0.3465909090909091





稠密向量方法 word2vec 的准确率为：0.4

## 4.2 结果分析

余弦相似度 (CosineSimilarity): 准确率为 55.44%，表明在比较文档的特征向量时，有一定的效用，但不是特别高。这可能是因为余弦相似度侧重于度量角度而非长度差异，所以如果两个文本的用词不够丰富或关键词选择不恰当，可能会影响准确率。

Jaccard 相似度 (JaccardSimilarity): 准确率为 60.91%，比余弦相似度高，这表明在比较文档时，基于集合的方法（如关键词的交集和并集）可能比基于向量的方法更为有效。Jaccard 相似度对于稀疏数据集表现通常更好。

编辑距离 (LevenshteinSimilarity): 准确率为 52.80%，是五种方法中的第三高。这表明通过计算两个字符串之间的编辑距离来衡量相似度的方法在该数据集上效果一般。这可能是因为编辑距离更适合比较短文本或单词级别的相似度，而不是长文本。

MinHash 相似度 (MinHashSimilarity): 准确率为 61.47%，是这些方法中最高的。这表明 MinHash 是一个在处理大规模数据集时效果较好的方法，特别是在需要估计大量文档对的 Jaccard 相似度时。

SimHash 相似度 (SimHashSimilarity): 准确率仅为 34.66%，是所有方法中最低的。这可能表明 SimHash 在这个特定的数据集或场景下表现不佳，可能是因为 SimHash 对文本的微小变化非常敏感，导致相似度评分较低，特别是在处理具有高度异质性的文本时。

总体而言，MinHash 和 Jaccard 相似度在这个例子中表现最好，表明它们在处理特定类型的数据集时可能更有效。然而，选择最合适的相似度计算方法应该基于具体的应用场景、数据类型和处理需求。此外，每种方法的表现也强烈依赖于预处理步骤（如分词、停用词去除、权重分配等）的质量和适应性。

使用预训练的稠密向量效率过低，且正确率一般，不适宜这个任务。其主要原因是因为所使用的 gensim 是一个纯 CPU 驱动的方法，因此在接下来的方法中使用的 bert 模型用 GPU 加速大大提升效率。

## 5. 扩展：大模型方法

在当前时代，大模型技术正在成为潮流。

这里选择了大模型 **BERT**：使用了大规模预训练的语言模型，能够捕捉更深层次的语言特征和上下文信息。这是一种基于 **Transformer** 的模型，具有更高的复杂性和表示能力。

### 5.1 与之前方法的异同

模型复杂性：

稀疏向量和稠密向量：这些方法通常使用较简单的数学模型（如余弦相似度或 Jaccard 相似度）来计算文本之间的相似度。它们依赖于文本的表面特征（如词频）或预训练的词向量。

**BERT**：使用了大规模预训练的语言模型，能够捕捉更深层次的语言特征和上下文信息。这是一种基于 **Transformer** 的模型，具有更高的复杂性和表示能力。

处理过程：

稀疏向量和稠密向量：通常涉及文本预处理、分词、向量化等步骤。对于稀疏向量，还需要选择特征词；对于稠密向量，需要通过词嵌入将词转换为向量。

**BERT**：使用预训练的 **tokenizer** 将文本转换为 **token ids**，然后将这些 **ids** 输入到 **BERT** 模型中。**BERT** 能够处理更长的上下文，并为每个 **token** 生成复杂的嵌入。

训练与推断：

稀疏向量和稠密向量：这些方法不需要额外的训练（除非你从头开始训练词嵌入模型），通常只需要基于预先计算的向量进行相似度计算。

**BERT**：虽然 **BERT** 模型是预训练的，但在这里它被进一步微调（即训练）来执行特定任务——判断两个句子是否相似。这涉及到更多的计算和数据，但可以提供更准确的结果。

应用场景：

稀疏向量和稠密向量：这些方法适合快速、近似的相似度计算，特别是在资源受限的环境中。

**BERT**：适合需要高精度和理解深层语义关系的应用场景。由于它能够理解复杂的上下文和语言结构，因此在理解上下文或句子之间的复杂关系方面更有效。

### 5.2 Bert 模型代码原理

#### 1. 数据集导入

定义了一个名为 **MyDataset** 的类，这个类是为了适配 **PyTorch** 的数据加载到深度学习模型中。这个类继承自 **torch.utils.data.Dataset**，是自定义数据集在 **PyTorch** 中的标准做法。**MyDataset** 类专门处理句子对，例如在自然语言处理任务中使用。

```
1. class MyDataset(Data.Dataset):
2.     def __init__(self, texts, tokenizer, max_length=128):
3.         self.texts = texts
4.         self.tokenizer = tokenizer
5.         self.max_length = max_length
6.
7.     def __len__(self):
8.         return len(self.texts)
9.
```

```

10.     def __getitem__(self, index):
11.         # 获取索引对应的句子对和标签
12.         sentence1, sentence2, label = self.texts[index]
13.
14.         # 使用 tokenizer 处理句子对
15.         # 这里我们处理两个句子，所以要传入两个句子作为参数
16.         encoded_pair = self.tokenizer.encode_plus(
17.             sentence1, sentence2,
18.             max_length=self.max_length,
19.             padding='max_length',
20.             truncation=True,
21.             return_tensors='pt'
22.         )
23.
24.         token_ids = encoded_pair['input_ids'].squeeze(0) # Tensor of token ids
25.         attn_masks = encoded_pair['attention_mask'].squeeze(0) # Binary tensor indicating padded
        values
26.         token_type_ids = encoded_pair['token_type_ids'].squeeze(
27.             0) # Binary tensor indicating sentence1 and sentence2 tokens
28.
29.         return token_ids, attn_masks, token_type_ids, torch.tensor(label)

```

## 2. Bert 模型架构

BertForSentencePairClassification 类定义了一个基于 BERT 的句子对分类模型，它能够处理成对的句子并预测它们之间的关系，如是否具有相同的意图、是否相互矛盾等。这种类型的模型广泛应用于自然语言处理中的各种任务，如情感分析、问答系统和语义相似度评估。

```

1.     class BertForSentencePairClassification(nn.Module):
2.         def __init__(self, bert_model, hidden_size=768, num_labels=2):
3.             super(BertForSentencePairClassification, self).__init__()
4.             self.num_labels = num_labels
5.             self.bert = AutoModel.from_pretrained(bert_model)
6.             self.classifier = nn.Linear(hidden_size, num_labels)
7.
8.         def forward(self, input_ids, attention_mask=None, token_type_ids=None):
9.             # 通过 BERT 模型获取句子表示
10.            outputs = self.bert(input_ids, attention_mask=attention_mask, token_type_ids=token_type_id
            s)
11.            pooled_output = outputs.pooler_output
12.
13.            # 将表示传递给分类器来获取最终的相似度分数
14.            logits = self.classifier(pooled_output)
15.
16.            return logits

```

### 3. 训练模型

定义了一个名为 `BertForSentencePairClassification` 的类，它是一个基于预训练的 BERT 模型构建的用于句子对分类任务的神经网络。其主要作用是接受一对句子，通过 BERT 模型提取特征，然后使用一个线性层进行分类。

特别的，由于模型参数量较大，需要在 GPU 环境下进行训练，本次实验的 GPU 环境为 V100，需要 30G 以上的显存。

```
1. def train_model(model, train_loader, val_loader, optimizer, num_epochs=3):
2.     for epoch in range(num_epochs):
3.         model.train()
4.         total_loss = 0
5.         num = 0
6.         for batch in train_loader:
7.             batch = tuple(b.to(device) for b in batch)
8.             input_ids, attention_mask, token_type_ids, labels = batch
9.
10.            # 清除之前的梯度
11.            optimizer.zero_grad()
12.
13.            # 前向传播
14.            outputs = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
15.            loss = nn.CrossEntropyLoss()(outputs, labels)
16.
17.            # 反向传播和优化
18.            loss.backward()
19.            optimizer.step()
20.
21.            total_loss += loss.item()
22.            num += 1
23.            if num % 5 == 0:
24.                print(f'has processed {num} batches')
25.
26.            torch.save(model.state_dict(), 'model.pth')
27.
28.            avg_loss = total_loss / len(train_loader)
29.            print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.5f}")
30.
31.            # 在每个 epoch 后评估模型
32.            evaluate(model, val_loader)
```

### 4. 评估模型

利用之前训练好的模型进行评估。

```
1. def evaluate(model, loader):
2.     model.eval()
```

```

3.         total = 0
4.         correct = 0
5.         with torch.no_grad():
6.             for batch in loader:
7.                 batch = tuple(b.to(device) for b in batch)
8.                 input_ids, attention_mask, token_type_ids, labels = batch
9.
10.                outputs = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
11.                predictions = torch.argmax(outputs, dim=1)
12.
13.                total += labels.size(0)
14.                correct += (predictions == labels).sum().item()
15.
16.            accuracy = 100 * correct / total
17.            print(f'Accuracy: {accuracy}%')

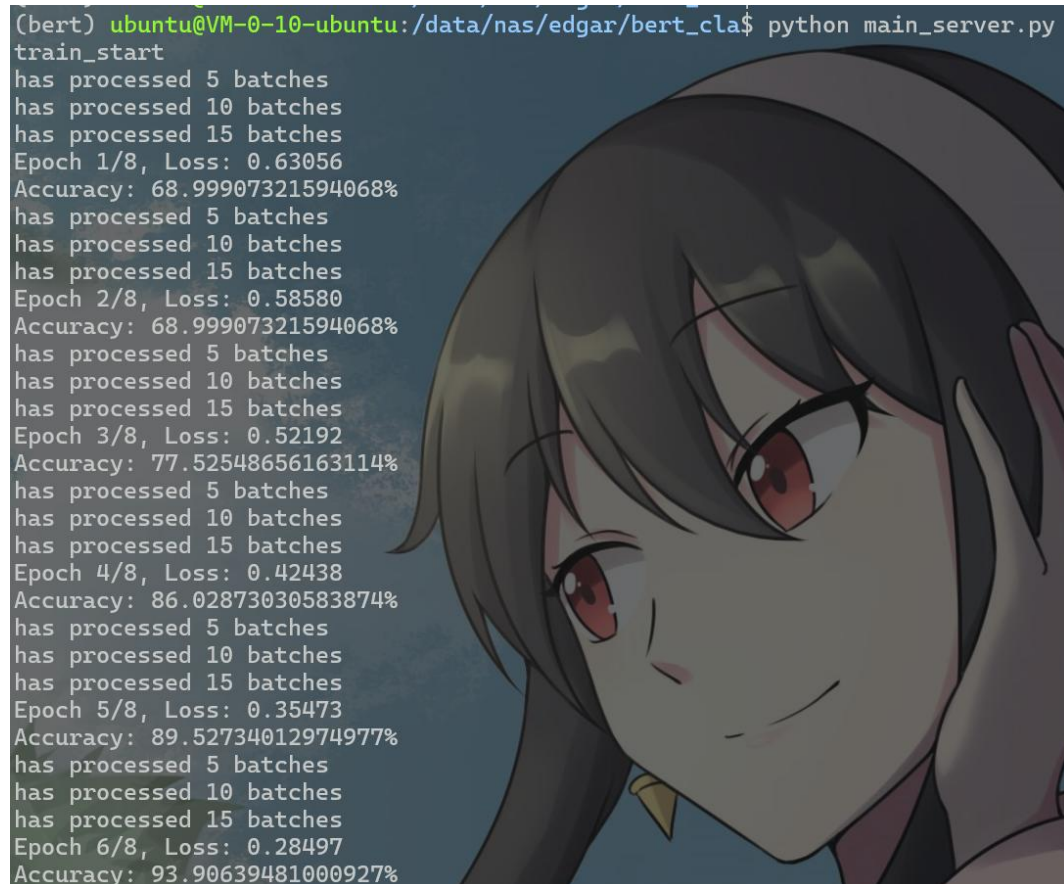
```

## 5. 3Bert 模型效果

使用了 train.json 中 34334 条数据进行训练，使用 dev.json 进行测试。

模型损失随着训练收敛，正确率也要比之前简单方法有显著提高。

训练过程：



```

(bert) ubuntu@VM-0-10-ubuntu: /data/nas/edgar/bert_cla$ python main_server.py
train_start
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 1/8, Loss: 0.63056
Accuracy: 68.99907321594068%
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 2/8, Loss: 0.58580
Accuracy: 68.99907321594068%
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 3/8, Loss: 0.52192
Accuracy: 77.52548656163114%
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 4/8, Loss: 0.42438
Accuracy: 86.02873030583874%
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 5/8, Loss: 0.35473
Accuracy: 89.52734012974977%
has processed 5 batches
has processed 10 batches
has processed 15 batches
Epoch 6/8, Loss: 0.28497
Accuracy: 93.90639481000927%

```

利用训练好的模型直接进行测试：

**Accuracy: 93.91%**