



基于 SVD 分解以及基于 SGNS 两种方法构建汉语词向量

自然语言处理课程实验报告

班级：110

姓名：耿翊中

学号：2021213382

2024 年 4 月

目录

基于 SVD 分解以及基于 SGNS	1
两种方法构建汉语词向量	1
自然语言处理课程实验报告	1
1. 实验要求	3
2. SVD 分解方法	4
2.1 数据预处理	4
2.3 应用 svd 分解得到词向量	6
2.4 计算余弦相似度	7
2.5 类结构化	9
2.6 模型参数调整与计算	9
2.7 实验总结	10
3. SGNS 方法	11
3.1 数据预处理（构建正、负样本对）	11
3.2 实现 SGNS 模型	14
3.3 训练 SGNS 模型	15
3.4 训练和测试过程	18
3.5 参数调整	21

1. 实验要求

作业内容：

一、概述：分别基于 SVD 分解以及基于 SGNS 两种方法构建汉语词向量并进行评测。

二、具体说明：

1、语料：training.txt。

2、基于 SVD 分解的方法：获取高维 distributional 表示时 $K=5$ ，SVD 降维后的维数自定，获得子词向量 vec_sta 。之后基于该向量计算 pku_sim_test.txt 中同一行中两个子词的余弦相似度 sim_svd 。当 pku_sim_test.txt 中某一个词没有获得向量时(该词未出现在该语料中)，令其所在行的两个词之间的 $sim_svd=0$ 。

3、基于 SGNS 的方法：SGNS 方法中窗口 $K=2$ ，子词向量维数自定，获得向量 vec_sgns 。之后基于该子词向量计算 pku_sim_test.txt 中同一行中两个词的余弦相似度 sim_sgns 。当 pku_sim_test.txt 中某一个词没有获得向量时(该词未出现在该语料中)，令其所在行的两个词之间的 $sim_sgns=0$ 。

4、两种方法的结果输出要求(因为是机器判定，请一定按如下格式输出)：

4.1 保持 pku_sim_test.txt 编码(utf-8)不变，保持原文行序不变

4.2 每行在行末加一个 tab 符之后写入该行两个词的 sim_sv ，再加一个 tab 符之后写入该行两个词的 sim_sgns 。

4.3 输出文件命名方式：学号。

5、所有输出文本均采用 Unicode(UTF-8)编码

6、算法采用 Python (3.0 以上版本) 实现

2. SVD 分解方法

2.1 数据预处理

从 training.txt 中读取句子数据，并转化为列表；统计得到词表。为后续的文本分析（如构建共现矩阵和应用 SVD 分解）创建基础。

由于在统计奇异值具体信息时，不能使用稀疏矩阵来进行分解，而完整的共现矩阵太大，所以需要设定低频词过滤，得到过滤后较小的词表。

```
1. def preprocess_file(self):
2.     """从文件中读取文本并预处理：统计词频"""
3.     print("开始预处理文件...")
4.     word_frequency = {} # 新增：用于统计词频的字典
5.     processed_sentences = []
6.     with open(self.train_path, 'r', encoding='utf-8') as f:
7.         for line in f:
8.             words = line.strip().split()
9.             filtered_words = []
10.            for word in words:
11.                if word.strip() != '':
12.                    filtered_words.append(word)
13.                    if word in word_frequency:
14.                        word_frequency[word] += 1
15.                    else:
16.                        word_frequency[word] = 1
17.            processed_sentences.append(" ".join(filtered_words))
18.        # 移除低频词
19.        min_frequency = 2 # 设定最低频率阈值
20.        filtered_vocab = {word for word, freq in word_frequency.items() if freq >= min_frequency}
21.    return processed_sentences, filtered_vocab
```

具体过程解释：

初始化词频字典：word_frequency = {} 创建了一个空字典，用于存储每个词及其对应的出现频次。

初始化处理后的句子列表：processed_sentences = [] 创建了一个空列表，用于存储处理后的句子。

读取训练文件：使用 with open(self.train_path, 'r', encoding='utf-8') 来打开指定路径的训练文件，这里使用了 'utf-8' 编码以支持多种语言文本。

逐行处理文本：

分词：words = line.strip().split() 将每一行文本首先去除首尾空白（strip()），然后按空格分割成单词列表。

过滤空字符串：通过 if word.strip() != '' 检查每个词是否为空字符串（仅包含空白的字符串），确保只处理有实际内容的词。

统计词频：如果词非空，将其添加到 filtered_words 列表中，并在 word_frequency 字典中更新该词的频次。如果该词已存在于字典中，频次加一；否则，将该词及其频次设为 1。

保存处理后的句子：使用 `" ".join(filtered_words)` 将过滤后的词重新组合成句子，然后添加到 `processed_sentences` 列表中。

移除低频词：

设定最低频率阈值 `min_frequency = 20`，意味着只有出现频次达到或超过这个阈值的词才会被保留。

`filtered_vocab = {word for word, freq in word_frequency.items() if freq >= min_frequency}` 通过列表推导式创建一个新的集合，包含所有出现频次达到或超过 20 次的词。

返回结果：方法返回两个值：`processed_sentences` 和 `filtered_vocab`。前者是一个经过预处理（去除空字符串并统计词频）的句子列表，后者是一个集合，包含了所有满足频次要求的词。

2.2 构建词共现矩阵

共现矩阵是自然语言处理（NLP）中的一种重要数据结构，用于表示词汇项在特定上下文中同时出现的频率。其中的行和列代表语料库中的词汇，而每个元素的值表示对应行和列的词汇项在预定的窗口大小内共同出现的次数。共现矩阵捕获了词汇之间的关系，这些关系将用于后续的相似度分析中。

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

```
1. def build_cooccurrence_matrix(self, sentences, filtered_vocab):
2.     """基于过滤后的词汇表构建共现矩阵"""
3.     print("构建共现矩阵...")
4.     vocab_index = {word: i for i, word in enumerate(filtered_vocab)}
5.     cooccurrence_matrix = np.zeros((len(filtered_vocab), len(filtered_vocab)), dtype=np.float32)
6.     for sentence in sentences:
7.         words = sentence.split()
8.         for i, word in enumerate(words):
9.             if word in vocab_index: # 确保单词在过滤后的词汇表中
10.                 target_word_index = vocab_index[word]
11.                 start = max(0, i - self.window_size)
12.                 end = min(len(words), i + self.window_size + 1)
13.                 for j in range(start, end):
14.                     if i != j and words[j] in vocab_index: # 同样确保上下文词在词汇表中
15.                         context_word_index = vocab_index[words[j]]
16.                         cooccurrence_matrix[target_word_index, context_word_index] += 1
```

```

17.     print("共现矩阵构建完成。")
18.     return cooccurrence_matrix, vocab_index

```

具体过程解释：

初始化词汇索引：通过 `{word: i for i, word in enumerate(filtered_vocab)}` 创建一个字典，为每个过滤后的词汇分配一个唯一的索引。这为构建矩阵提供了一个映射，将词汇映射到矩阵的行和列上。

创建空的共现矩阵：使用 `np.zeros((len(filtered_vocab), len(filtered_vocab)), dtype=np.float32)` 初始化一个方阵，其维度等于过滤后的词汇表的大小。这个矩阵用于记录词汇之间的共现频次。

填充共现矩阵：

遍历每个句子中的每个词。

使用一个窗口(由 `self.window_size` 定义)来确定每个目标词周围考虑的上下文范围。

对于窗口内的每个上下文词，如果它也在过滤后的词汇表中，则在共现矩阵的相应位置（即目标词和上下文词的索引处）增加 1，表示它们在给定窗口大小内共同出现。

完成提示与返回：在构建完成后打印提示，并返回构建好的共现矩阵及词汇索引。

2.3 应用 svd 分解得到词向量

1. 为什么要进行 svd 分解

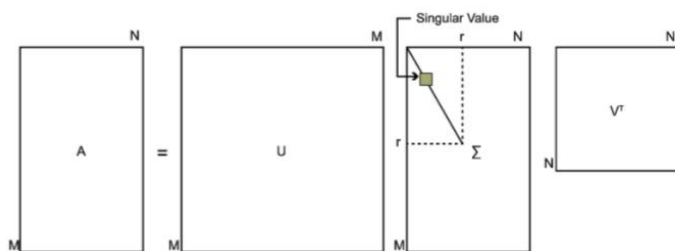
共现矩阵通过统计词汇之间的共现频次，能够捕捉到词汇之间的语义关系。通常，语义上相关或功能上相似的词在文本中更频繁地共同出现。但共现矩阵通常是高维的，并且稀疏。通过应用降维技术（如 SVD），可以从中提取出更为紧凑、高效的词向量表示，这些表示捕获了词汇的主要语义特征。

2. svd 分解的基本原理

SVD也是对矩阵进行分解，但是和特征分解不同，SVD并不要求要分解的矩阵为方阵。假设我们的矩阵A是一个 $m \times n$ 的矩阵，那么我们定义矩阵A的SVD为：

$$A = U \Sigma V^T$$

其中U是一个 $m \times m$ 的矩阵， Σ 是一个 $m \times n$ 的矩阵，除了主对角线上的元素以外全为0，主对角线上的每个元素都称为奇异值，V是一个 $n \times n$ 的矩阵。U和V都是酉矩阵，即满足 $U^T U = I, V^T V = I$ 。下图可以很形象的看出上面SVD的定义：



对于奇异值，它跟我们特征分解中的特征值类似，在奇异值矩阵中也是按照从大到小排列，而且奇异值的减少特别的快，在很多情况下，前 10% 甚至 1% 的奇异值的和就占了全部的奇异值之和的 99% 以上的比例。也就是说，我们也可以用最大的 k 个的奇异值和对应的左右奇异向量来近似描述矩阵。

3. svd 分解的具体代码实现

在计算较大的矩阵时，可以使用 `scipy` 库中的 `svds` 方法，对稀疏矩阵进行分解

```

1.     # 使用 lil_matrix 初始化稀疏共现矩阵，并指定为浮点数据类型

```

```

2. cooccurrence_matrix = lil_matrix((len(vocab), len(vocab)), dtype=np.float32)
3.
4. def apply_svd(self, cooccurrence_matrix):
5.     k = self.k
6.     # 对稀疏共现矩阵应用 SVD 分解
7.     U, Sigma, VT = svds(cooccurrence_matrix, k=k)
8.     U_k = U
9.
10.    return U_k

```

而在分析完整的奇异值性质的时候,需要进行完整的奇异值分解,可以使用 `scipy` 库中的 `svd` 方法

```

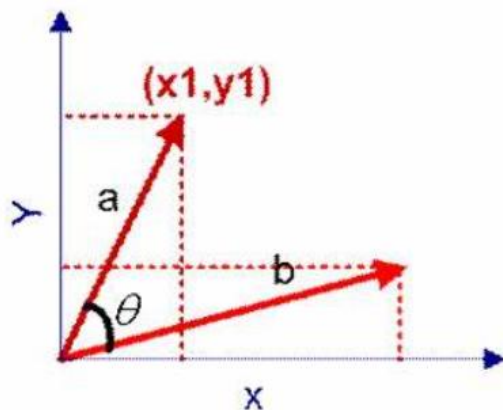
1. cooccurrence_matrix = np.zeros((len(filtered_vocab), len(filtered_vocab)), dtype=np.float32)
2.
3. def apply_svd(self, cooccurrence_matrix):
4.     print("应用 SVD 分解...")
5.     k = self.k
6.     # 将稀疏矩阵转换为稠密矩阵, 因为 scipy.linalg.svd 不接受稀疏矩阵
7.     dense_matrix = cooccurrence_matrix
8.     # 应用完整的 SVD 分解
9.     U_full, Sigma_full, VT_full = svd(dense_matrix, full_matrices=False, lapack_driver='gesvd')
10.
11.    # 从完整的奇异值中获取前 k 个奇异值和向量
12.    U_k = U_full[:, :k]
13.    Sigma_k = Sigma_full[:k]
14.
15.    return U_k

```

最终分解后得到每一个向量 k 个维度的词向量, 这里的 k 就是新词向量的维度

2.4 计算余弦相似度

余弦相似度是衡量两个非零向量在方向上的相似度或者差异度的一种度量方法, 不受两个向量的长度影响。在自然语言处理(NLP)和信息检索中, 余弦相似度经常被用来计算两个文档或者两个词向量之间的相似度。



计算过程

计算点积：首先，计算两个向量的点积，即相应元素的乘积之和。

计算向量的范数：接着，分别计算这两个向量的欧几里得范数，即每个向量的元素平方之和的平方根。

计算余弦值：将两个向量的点积除以它们范数的乘积，得到的值即为余弦相似度。

$$\begin{aligned} \cos(\theta) &= \frac{\sum_{i=1}^n (x_i \times y_i)}{\sqrt{\sum_{i=1}^n (x_i)^2} \times \sqrt{\sum_{i=1}^n (y_i)^2}} \\ &= \frac{a \bullet b}{||a|| \times ||b||} \end{aligned}$$

公式(4)

知乎 @summe

```
1. def cosine_similarity(self, vec1, vec2):
2.     """计算两个向量之间的余弦相似度"""
3.     dot_product = np.dot(vec1, vec2)
4.     norm_vec1 = np.linalg.norm(vec1)
5.     norm_vec2 = np.linalg.norm(vec2)
6.     similarity = dot_product / (norm_vec1 * norm_vec2)
7.     return similarity
8.
9. def calculate_similarity_pairs(self, reduced_matrix, vocab_index):
10.    """计算 pku_sim_test.txt 中每一行两个词的余弦相似度"""
11.    similarities = []
12.    with open(self.test_path, 'r', encoding='utf-8') as f: # 确保使用正确的编码
13.        for line in f:
14.            word1, word2 = line.strip().split()
15.            # 检查词汇是否存在
16.            if word1 in vocab_index and word2 in vocab_index:
17.                vec1 = reduced_matrix[vocab_index[word1]]
18.                vec2 = reduced_matrix[vocab_index[word2]]
19.                similarity = self.cosine_similarity(vec1, vec2)
20.            else:
21.                similarity = 0 # 如果任一词汇不存在，相似度为 0
22.                similarities.append((word1, word2, similarity))
23.    return similarities
```

cosine_similarity 函数实现了上述计算余弦相似度的过程。它接收两个向量 vec1 和 vec2 作为输入，首先计算它们的点积，然后计算各自的范数，最后用点积除以范数的乘积

得到余弦相似度。

`calculate_similarity_pairs` 函数遍历一个文件中的所有词对，对于文件中的每一行，提取两个词 `word1` 和 `word2`。如果这两个词都在词汇索引 `vocab_index` 中，它会用它们的向量（从 `reduced_matrix` 中获取）来计算余弦相似度。如果任一词不存在于词汇索引中，它们的相似度被设置为 0。最后，所有的相似度值被收集到一个列表中并返回。

2.5 类结构化

在分步按照流程完成各个步骤后，构建类 `Svd_dec` 方法。这个过程包括文本的预处理、构建共现矩阵、应用 SVD 分解以及计算相似度。

类结构与参数

`__init__`: 初始化函数接收五个参数：停用词文件路径 `stopwords_path`、训练集路径 `train_path`、测试集路径 `test_path`、窗口大小 `window_size` 和保留的奇异值数 `k`。这些参数为处理文本和计算相似度提供了必要的配置。

`preprocess_file`: 从训练集中读取文本，移除停用词，统计词频，并过滤掉低频词。这一步确保了后续处理的文本是干净且有意义的。

`build_cooccurrence_matrix`: 基于预处理后的词汇构建共现矩阵，这一矩阵记录了词汇之间的共现信息，为 SVD 分解提供了基础数据。

`apply_svd`: 对共现矩阵应用 SVD 分解，以降低词向量的维度，并保留最重要的 `k` 个维度。这一步是整个过程的核心，它使用了 `scipy.linalg.svd` 来执行分解。

`cosine_similarity`: 计算两个向量之间的余弦相似度，用于评估词向量之间的相似性。

`calculate_similarity_pairs`: 从测试集中读取词对，并使用之前计算得到的降维词向量和 `cosine_similarity` 函数来计算它们之间的相似度。

```
class Svd_dec:

    新 *
    def __init__(self, stopwords_path, train_path, test_path, window_size=5, k=5):...

    1 个用法 新 *
    def preprocess_file(self):...

    1 个用法 新 *
    def build_cooccurrence_matrix(self, sentences):...

    1 个用法 新 *
    def apply_svd(self, cooccurrence_matrix):...

    1 个用法 新 *
    def cosine_similarity(self, vec1, vec2):...

    1 个用法 新 *
    def calculate_similarity_pairs(self, reduced_matrix, vocab_index):...
```

2.6 模型参数调整与计算

实验要求中 `window_size=5`，故只需要调整降维后的维数即可

为了评估不同维数下的效果，我采取了计算输出结果的平均相似度和人工抽样观察法，来评估输出的相似度结果是否合理。

并在 `svd` 分解后计算总共有多少个非零奇异值，选取了多少个奇异值，选取的奇异值之

和、全部奇异值之和以及二者的比例。来观察是否实践结果和理论一样，前几个奇异值之和占所有奇异值之和的绝大部分。去除低频词后选取奇异值为 15582 个

降维后维数	平均相似度	选取奇异值数	选取奇异值之和	全部奇异值之和	二者比例
5	0.36536904340889303	5	288400.25	744184.9375	0.3875384032726288
10	0.25919638577848675	10	328639.875	744185.0	0.4416104555130005
20	0.19800177237275057	20	363620.1875	744185.4375	0.4886150360107422
100	0.10417586326168385	100	439746.125	744185.0625	0.5909096598625183

2.7 实验总结

1. 维度对相似度的影响

随着降维后的维数增加，平均相似度逐渐降低。这可能是因为增加维数使得词向量包含了更多的信息，但同时也可能引入了一些噪声或者不那么重要的信息，使得部分不相似的词对的相似度被放大，从而降低了整体的平均相似度。

2. 奇异值数和奇异值之和的分析

选取的奇异值数与相似度：随着保留的奇异值数量增加，可以看到，选取的奇异值之和占全部奇异值之和的比例逐渐增大。这意味着随着维度的增加，我们保留了更多的信息。

奇异值的累积占比：实验数据显示，即使是最小的降维数（5 个奇异值），选取的奇异值之和已经占到了全部奇异值之和的大约 38.75%，而 100 个奇异值时，这一比例增加到了约 59.09%。这一结果符合 SVD 的理论基础，即数据的主要信息往往被前几个奇异值所捕捉。随着保留的奇异值数量增加，累积的信息量也在增加，但增速会随着奇异值数量的增加而递减，这是因为奇异值的大小一般会按照从大到小的顺序排列，越往后的奇异值代表的信息量越少。

3. 所得结论

维度选择：在选择降维后的维数时，需要在保留足够信息（通过奇异值之和的比例来衡量）和维护高平均相似度之间找到平衡。虽然更高的维数可能包含更多信息，但也可能降低相似度的准确性。根据结果，较低的维数（如 5 或 10）能够保持相对较高的平均相似度，这可能是因为在这个维度下，模型能够捕捉到最关键的语义信息，而忽略了噪声。

奇异值之和比例：理论上，选取的奇异值之和应当占全部奇异值之和的绝大部分，以确保信息的大部分被保留。实践中，可以根据这个比例来判断是否需要增加保留的奇异值数量。你的实验数据显示，即便是最低的降维设置（5 维），也已经保留了相当一部分的信息。随着维度的增加，虽然可以保留更多信息，但平均相似度的下降也提示了维度增加带来的潜在问题。

性能与准确度权衡：在实际应用中，选择适当的维数不仅需要考虑到信息的保留量，还需要考虑到计算成本。较高的维度虽然能够提供更多信息，但也意味着更高的计算成本和可能的性能下降。当维度上升后，计算所需要的时间也同时显著增加。

3. SGNS 方法

3.1 数据预处理（构建正、负样本对）

```
class Data_prepare:

    def __init__(self, stopwords_path, train_path, window_size, num_negative_samp
        self.stopwords_path = stopwords_path
        self.train_path = train_path
        self.window_size = window_size
        self.processed_sentences = None
        self.vocab = None
        self.num_negative_samples = num_negative_samples
```

1. 数据预处理要获得什么

Skip-gram 模型的核心思想是通过当前词来预测其上下文中的词。具体来说，模型尝试预测在给定的一个词的情况下，它周围的词是什么。例如，给定一个句子“The quick brown fox jumps over the lazy dog”，如果选择“fox”作为当前词，根据预定的窗口大小（假设窗口大小为 2），模型的任务是预测“quick”、“brown”、“jumps”、“over”这些上下文词。

所以首先我们需要获取正样本对，即每一个词对应的相邻词对。

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	...

input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine
a	not
a	make
a	machine
a	in
machine	make
machine	a
machine	in
machine	the
in	a
in	machine
in	the
in	likeness

为了训练模型，我们还需要进行负采样，即随机生成并不相邻的词对，作为负样本对。

Pick randomly from vocabulary (random sampling)

input word	output word	target
not	thou	1
not	aaron	0
not	taco	0
not	shalt	1
not	make	1

Word	Count	Probability
aardvark		
aarhus		
aaron		
taco		
thou		
zyzzyva		

2. 代码实现

初始化

`__init__`方法是类的构造函数，用于初始化对象。在这里，它接收停用词文件路径、训练数据文件路径、窗口大小、负样本数量等参数，并将它们保存为对象的属性。

```

1. def __init__(self, stopwords_path, train_path, window_size, num_negative_samples):
2.     self.stopwords_path = stopwords_path
3.     self.train_path = train_path
4.     self.window_size = window_size
5.     self.processed_sentences = None
6.     self.vocab = None
7.     self.num_negative_samples = num_negative_samples

```

分词和去除停用词

`preprocess_file`方法读取训练数据文件和停用词文件。对于训练数据中的每一行，它首先按空格进行分词，然后去除停用词和空字符串，最后将处理过的句子保存在一个列表中。

```

1. # 分词
2. def preprocess_file(self):
3.     """从文件中读取文本并预处理：去除停用词"""
4.     with open(self.stopwords_path, 'r', encoding='utf-8') as f:
5.         stopwords = set([line.strip() for line in f])
6.         processed_sentences = []
7.         with open(self.train_path, 'r', encoding='utf-8') as f:
8.             for line in f:
9.                 words = line.strip().split()
10.                filtered_words = [word for word in words if word not in stopwords and word.strip()
11.                                   != '']
12.                processed_sentences.append(" ".join(filtered_words))
13.            self.processed_sentences = processed_sentences
14.            return processed_sentences

```

构建词汇表

`build_vocab` 方法遍历预处理后的句子，并为句子中的每个唯一词汇分配一个唯一的索引（即词汇表中的位置）。这个词汇表是一个字典，键是词汇本身，值是词汇的索引。

```
1. # 构建词汇表
2. def build_vocab(self):
3.     vocab = {}
4.     for sentence in self.processed_sentences:
5.         for word in sentence.split():
6.             if word not in vocab:
7.                 vocab[word] = len(vocab)
8.     self.vocab = vocab
9.     return vocab
```

生成训练数据

`generate_training_data` 方法首先生成正样本对。对于预处理过的每个句子，它遍历句子中的每个词，然后根据窗口大小确定该词的上下文词，将中心词与上下文词的索引作为正样本对添加到列表中。

接着,为每个正样本对生成负样本。负样本是随机选择的,不属于中心词当前窗口的词汇索引。通过控制负样本的数量,可以调整模型训练的难度和方向。

```

1.     def generate_training_data(self):
2.         positive_pairs = []
3.         negative_samples = []
4.
5.         # 生成所有可能的正样本对
6.         for sentence in self.processed_sentences:
7.             words = sentence.split()
8.             for i, center_word in enumerate(words):
9.                 center_word_index = self.vocab[center_word]
10.                for j in range(max(0, i - self.window_size), min(i + self.window_size + 1, len(words))):
11.                    if i != j:
12.                        context_word = words[j]
13.                        context_word_index = self.vocab[context_word]
14.                        positive_pairs.append((center_word_index, context_word_index))
15.
16.         # 生成负样本
17.         vocab_indices = list(self.vocab.values())
18.         for _ in range(len(positive_pairs)):
19.             negatives = []
20.             while len(negatives) < self.num_negative_samples:
21.                 negative = random.choice(vocab_indices)
22.                 if negative not in negatives: # 确保负样本是唯一的
23.                     negatives.append(negative)
24.             negative_samples.append(negatives)
25.

```

26. `return positive_pairs, negative_samples`

保存预处理数据

`save_preprocessed_data` 方法将正样本对、负样本和词汇表保存到指定的文件路径。这里使用 `pickle` 进行序列化，便于之后的加载和使用。

整合预处理流程

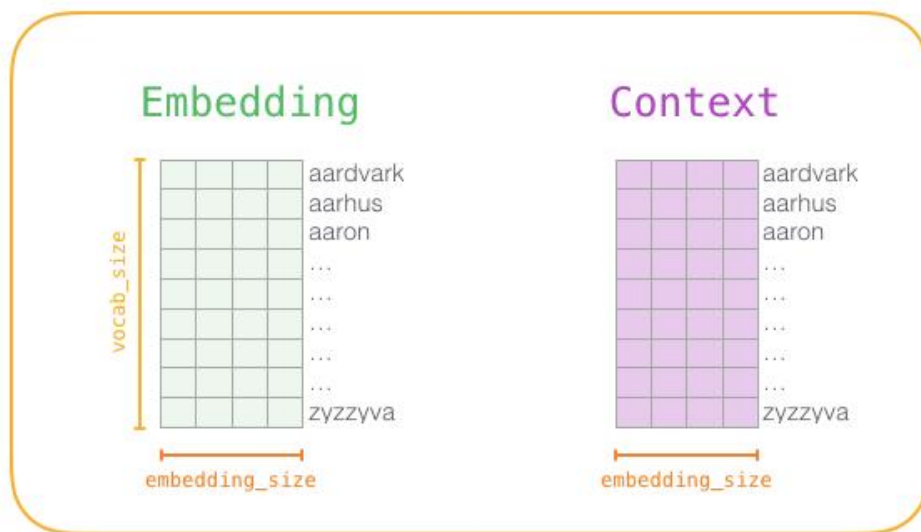
`pre_data` 方法整合了以上的预处理流程，从读取和预处理文件开始，到构建词汇表，生成训练数据（包括正负样本对），最后保存预处理数据。此外，它还打印出词汇量、正样本对数量以及每个正样本对对应的负样本数量，以供检查。

```
1. # 保存预处理数据
2. def save_preprocessed_data(self, positive_pairs, negative_samples, vocab, filepath):
3.     with open(filepath, 'wb') as f:
4.         pickle.dump({
5.             'positive_pairs': positive_pairs,
6.             'negative_samples': negative_samples,
7.             'vocab': vocab
8.         }, f)
9.     print(f"Preprocessed data saved to {filepath}")
10.
11. def pre_data(self):
12.     self.preprocess_file()
13.     self.build_vocab()
14.     positive_pairs, negative_samples = self.generate_training_data()
15.
16. # 输出样本数量进行检查
17. print(f"词汇储量为: {len(self.vocab)}")
18. print(f"总共有{len(positive_pairs)}个正样本对")
19. print(f"每个正样本对对应{len(negative_samples[0])}个负样本")
20.
21. self.save_preprocessed_data(positive_pairs, negative_samples, self.vocab, 'preprocessed_data.pkl')
22. return positive_pairs, negative_samples, len(self.vocab), self.vocab
```

3.2 实现 SGNS 模型

在已经了解 `skip-gram` 和负例采样两个概念之后，就要将它们实际运用到 `word2vec` 的训练过程中来。

首先，我们创建两个矩阵——中心词 `Cencer(Embedding)` 矩阵和上下文词 `Context` 矩阵。这两个矩阵在我们的词汇表中嵌入了每个单词（所以 `vocab_size` 是他们的维度之一）。第二个维度是我们希望每次嵌入的长度（`embedding_size`）。



代码实现：

```
1. class SGNSModel(nn.Module):
2.     def __init__(self, vocab_size, embedding_dim):
3.         super(SGNSModel, self).__init__()
4.         # 中心词嵌入层
5.         self.center_embeddings = nn.Embedding(vocab_size, embedding_dim)
6.         # 上下文词嵌入层
7.         self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)
8.         # 词嵌入维度
9.         self.embedding_dim = embedding_dim
10.
11.         # 初始化权重
12.         self.init_weights()
```

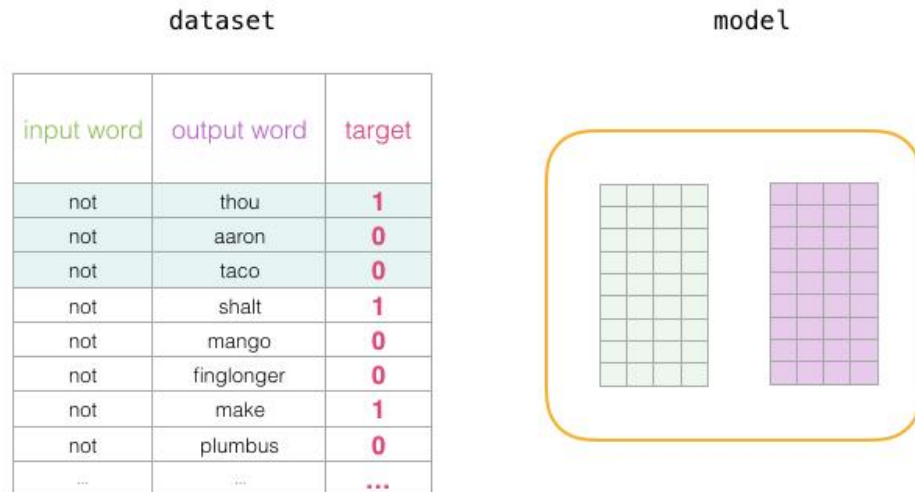
权重初始化：在训练过程开始时，使用随机值初始化这些矩阵

```
1. def init_weights(self):
2.     # Xavier 初始化的一种常见选择是使用均匀分布
3.     initrange = 0.5 / self.embedding_dim
4.     self.center_embeddings.weight.data.uniform_(-initrange, initrange)
5.     self.context_embeddings.weight.data.uniform_(-initrange, initrange)
```

3.3 训练 SGNS 模型

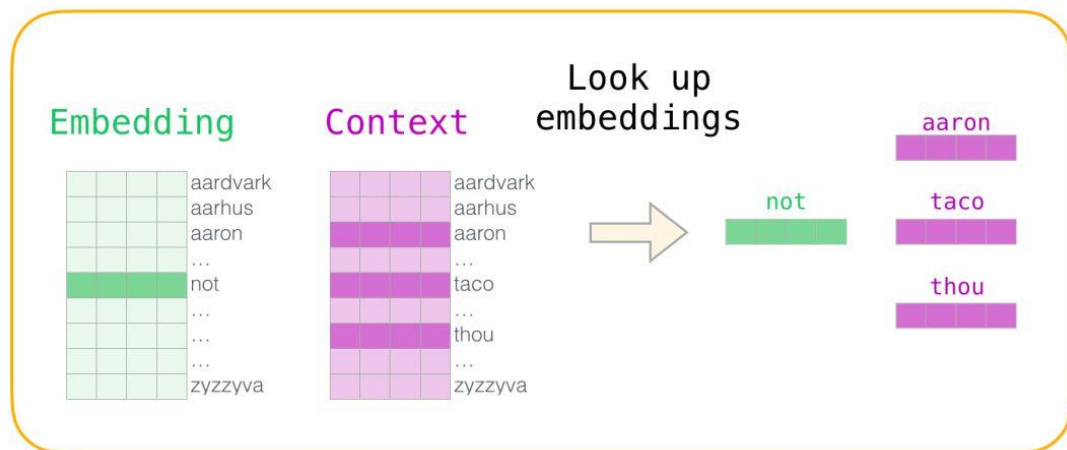
在每个训练步骤中，我们采取一个相邻的例子及其相关的非相邻例子。

例如：



上图中有四个单词：输入单词 **not** 和输出/上下文单词: **thou**（实际邻居词），**aaron** 和 **taco**（负面例子）。

我们继续查找它们的嵌入——对于输入词，我们查看 **Embedding** 矩阵。对于上下文单词，我们查看 **Context** 矩阵（即使两个矩阵都在我们的词汇表中嵌入了每个单词）。









```

1. def forward(self, center_word_indices, context_word_indices, negative_word_indices):
2.     # 查找中心词和上下文词的嵌入
3.     center_embeds = self.center_embeddings(center_word_indices)
4.     context_embeds = self.context_embeddings(context_word_indices)

```


然后，我们计算输入嵌入与每个上下文嵌入的点积。在每种情况下，结果都将是表示输入和上下文嵌入的相似性的数字。并使用 **Sigmoid** 函数，来将结果都转换为正值，且处于 0 到 1 之间。

input word	output word	target	input • output	sigmoid()
not 	thou 	1	0.2	0.55
not 	aaron 	0	-1.11	0.25
not 	taco 	0	0.74	0.68







```

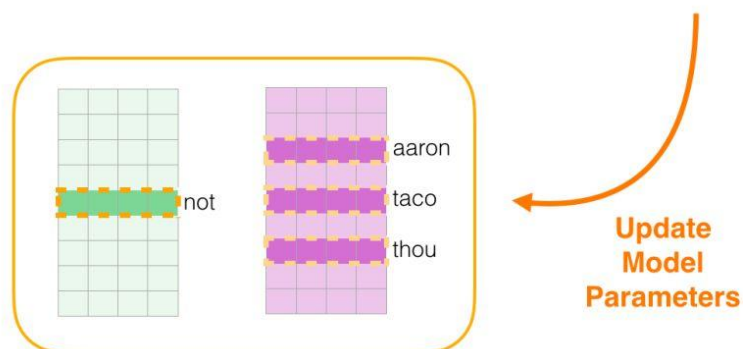
1. # 计算正样本对的得分
2.     positive_scores = torch.sum(center_embeds * context_embeds, dim=1)
3.     # 使用 logsigmoid 处理正样本得分
4.     log_target = F.logsigmoid(positive_scores)
5.
6.     # 查找负样本的嵌入并计算得分
7.     negative_embeds = self.context_embeddings(negative_word_indices)
8.     # 负样本得分的计算稍有不同，需要额外的维度操作
9.     negative_scores = torch.bmm(negative_embeds, center_embeds.unsqueeze(2)).squeeze(2)
10.    # 使用 logsigmoid 处理负样本得分
11.    sum_log_sampled = F.logsigmoid(-1 * negative_scores)

```

而后定义损失，来进行训练。既然未经训练的模型已做出预测，而且我们确实拥有真实目标标签来作对比，那么让我们计算模型预测中的误差吧。为此我们只需从目标标签中减去 sigmoid 分数。error = target - sigmoid_scores

通过不断优化这个过程，得到最终的词向量。

input word	output word	target	input • output	sigmoid()	Error
not 	thou 	1	0.2	0.55	0.45
not 	aaron 	0	-1.11	0.25	-0.25
not 	taco 	0	0.74	0.68	-0.68



```

1. # 损失是正样本和负样本得分的 logsigmoid 之和的负数
2.     loss = -1 * (log_target.sum() + sum_log_sampled.sum()) / (len(positive_scores) + len(negative_scores))

```

3.

4. `return loss`

3.4 训练和测试过程

训练过程:

模型准备: 在每次训练开始前, 通过 `model.train()` 设置模型为训练模式。

损失记录: 用列表记录每个 `epoch` 的平均损失值, 以及每 200 个 `batch` 的损失。

迭代训练: 对于每个 `epoch`, 迭代训练数据集中的所有样本。

数据加载: 通过 `DataLoader` 从 `data_loader` 中加载 `batch` 数据 (中心词、上下文词、负样本词), 并移动到指定的设备 (GPU 或 CPU)。

梯度清零: 使用 `optimizer.zero_grad()` 清除旧的梯度信息。

前向传播: 计算模型的损失。

反向传播: 通过 `loss.backward()` 计算梯度。

优化器更新: 使用 `optimizer.step()` 更新模型参数。

损失记录和打印: 每 200 个 `batch` 打印当前的损失和时间信息, 每 500 个 `batch` 保存模型和词向量, 以便进行测试。

损失可视化: 每个 `epoch` 结束后, 绘制并保存损失变化图。

模型保存: 保存训练完成的模型参数和词向量。

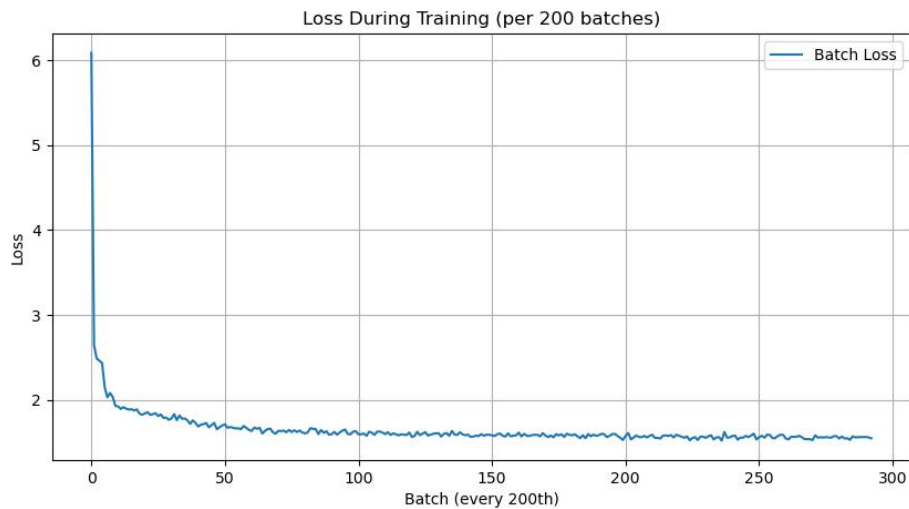
```
1. def train_process(model, data_loader, optimizer, device, epochs=5):
2.     model.train()
3.     loss_values = [] # 用于存储每个 epoch 的平均损失
4.     for epoch in range(epochs):
5.         batch_loss_values = [] # 用于存储每 200 个 batch 的损失
6.         start_time = time.time() # 记录每个 epoch 开始的时间
7.         total_loss = 0
8.         for batch_idx, (center, context, negatives) in enumerate(data_loader):
9.             center = center.to(device)
10.            context = context.to(device)
11.            negatives = negatives.to(device)
12.
13.            optimizer.zero_grad()
14.            loss = model(center, context, negatives)
15.            loss.backward()
16.            optimizer.step()
17.
18.            total_loss += loss.item()
19.
20.            # 特定批次打印时间和损失信息
21.            if (batch_idx + 1) % 5 == 0: # 假设每 200 个 batch 打印一次
22.                elapsed = time.time() - start_time
23.                print(
24.                    f"Epoch {epoch + 1}, Batch {batch_idx + 1}, Current Loss: {loss.item()}, Time
                    elapsed: {elapsed:.2f} seconds")
```

```

25.         batch_loss_values.append(loss.item()) # 将当前损失加入列表
26.         if (batch_idx + 1) % 500 == 0: # 假设每 500 个 batch 打印一次
27.             # 保存模型和词向量
28.             # 保存模型参数
29.             torch.save(model.state_dict(), 'sgns_model.pth')
30.             # 保存词向量
31.             word_vectors = model.center_embeddings.weight.data
32.             torch.save(word_vectors, 'word_vectors.pth')
33.
34.             test(test_path, embedding_dim)
35.
36.             # 绘制每个 epoch 的损失变化
37.             plt.figure(figsize=(10, 5))
38.             plt.plot(batch_loss_values, label='Batch Loss')
39.             plt.xlabel('Batch (every 200th)')
40.             plt.ylabel('Loss')
41.             plt.title('Loss During Training (per 200 batches)')
42.             plt.legend()
43.             plt.grid(True)
44.             plt.savefig(f'../result/batch_loss_epoch{epoch + 1}.png') # 保存损失图像到本地
45.             plt.show() # 显示图像
46.
47.             average_loss = total_loss / len(data_loader)
48.             print(f"Epoch {epoch + 1}, Loss: {average_loss}")
49.             loss_values.append(average_loss)
50.
51.             # 保存模型和词向量
52.             # 保存模型参数
53.             torch.save(model.state_dict(), 'sgns_model.pth')
54.             # 保存词向量
55.             word_vectors = model.center_embeddings.weight.data
56.             torch.save(word_vectors, 'word_vectors.pth')
57.
58.             test(test_path, embedding_dim)
59.
60.             # 绘制损失图
61.             plt.figure(figsize=(10, 5))
62.             plt.plot(loss_values, label='Training Loss')
63.             plt.xlabel('Epoch')
64.             plt.ylabel('Loss')
65.             plt.title('Loss During Training')
66.             plt.legend()
67.             plt.grid(True)
68.             plt.savefig('training_loss.png') # 保存损失图像到本地

```

69. `plt.show()` # 显示图像
训练过程中损失变化图像:



测试过程:

加载词向量和词汇表: 加载训练过程中保存的词向量和词汇表。

模型实例化和加载参数: 根据词汇表大小和嵌入维度初始化 **SGNS** 模型, 并加载训练好的模型参数。

计算相似度: 对测试集中的词对计算余弦相似度, 并打印结果。如果词对中的词不在词汇表中, 则相似度为 0。

结果整合和保存: 读取原始相似度文件, 添加 **SGNS** 模型计算的相似度数据, 并保存到结果文件。

```
1. def test(test_path, embedding_dim):
2.     # 加载词向量
3.     word_vectors = torch.load('word_vectors.pth')
4.     # 加载词汇表
5.     with open('vocab.pkl', 'rb') as f:
6.         vocab = pickle.load(f)
7.         len_vocab = len(vocab)
8.
9.     # 实例化模型
10.    vocab_size = len_vocab # 假定的词汇表大小
11.    model = SGNSModel(vocab_size, embedding_dim)
12.    model.load_state_dict(torch.load('sgns_model.pth'))
13.
14.    # 进行预测
15.    similarities_sgns, average_similarity = predict_similarity(test_path, vocab, word_vectors)
16.
17.    # 读取原始文件
18.    with open('../result/similarities.txt', 'r', encoding='utf-8') as file:
19.        lines = file.readlines()
20.    # 修改每一行, 追加 SGNS 相似度数据
```

```

21.     modified_lines = []
22.     for line, (_, _, sim_sgns) in zip(lines, similarities_sgns):
23.         modified_line = line.strip() + f"\t{sim_sgns}\n" # 追加 SGNS 数据
24.         modified_lines.append(modified_line)
25.
26.     # 将修改后的内容写回到新文件或覆盖原文件
27.     with open('../result/similarity.txt', 'w', encoding='utf-8') as file:
28.         file.writelines(modified_lines)

```

3.5 参数调整

基于 SGNS 方法的 word2vec 训练过程中两个关键超参数是窗口大小和负样本数量，其中窗口大小已经规定为 2。

1. 窗口大小

定义：窗口大小定义了目标词周围上下文的范围。例如，窗口大小为 2 意味着模型将考虑目标词前后各两个词作为上下文。

影响：较小的窗口尺寸使模型更加关注于学习词汇的局部语义特征（如同义词），而较大的窗口尺寸帮助模型捕捉更广泛的上下文信息，可能捕捉到词的更广泛用法或语法特征。窗口大小的选择取决于特定的任务需求。故这个任务中选取的窗口大小为 2。

2. 负样本数量

定义：在负采样中，每个正样本会对随机选取一定数量的负样本（即不是目标词上下文的词）进行训练。

影响：负样本数量的增加通常会提高训练的难度，但也可以提高模型的鉴别能力，从而更好地区分上下文中的词和随机词。然而，过多的负样本可能会导致训练时间显著增加而没有相应的性能提升，甚至可能损害模型的性能。

在实验过程中，我发现较低的负样本数量会导致所得到的相似度结果普遍偏高，故最终选择负样本数量为 20。

3. 词向量维度

定义：词向量维度是词嵌入的大小，即每个词被表示为多少维的向量。

影响：较高的维度可以提供更多的参数来捕捉复杂的语义关系，但也可能导致过拟合，特别是在数据量较少的情况下。较低的维度可能无法充分表达词之间的细微差别。

在刚开始实验的时候，只将词向量维度设定为 100，导致所得到的结果普遍偏高，平均相似度达到 0.9 以上，最终选择了词向量维度为 500

4. 学习率

定义：学习率决定了在每次参数更新时对损失梯度的应用程度。

影响：过高的学习率可能导致训练不稳定，甚至使得模型无法收敛。而过低的学习率会导致训练过程缓慢，需要更多的时间来收敛。适当的学习率可以平衡训练速度和模型性能。

实践后最终将学习率设定为 0.1

5. 训练批次大小

定义：训练轮数是整个训练集用于训练模型的次数。

影响：训练轮数越多，模型就有更多的机会学习数据。然而，过多的训练轮数可能会导致过拟合，尤其是当模型在训练集上表现良好，但在未见过的数据上表现不佳时。适当的轮数应该通过早停（early stopping）等技术来确定，以在训练时间和模型性能之间找到最佳平衡点。

模型在训练中损失很快就达到稳定状态，最终批次定位 1

附录：

自己在这个实验过程中进行了多种尝试

实验中自己创建的相关 github：

[piedpiperG/nlp_course \(github.com\)](https://github.com/piedpiperG/nlp_course)

[piedpiperG/Text-similarity-research \(github.com\)](https://github.com/piedpiperG/Text-similarity-research)