基于SVD和SGNS方法构建汉语词向量

1. 研究背景

在自然语言处理领域，词向量是一种表示词语语义信息的重要方式。词向量将每个词语表示成一个向量，在向量空间中相似的词语被映射到相近的位置，从而可以用向量之间的距离或夹角来衡量词语之间的相似度或关联度。

在过去几年中，随着深度学习技术的不断发展和应用，基于神经网络的词向量生成方法逐渐成为主流。其中，SGNS（Skip-Gram with Negative Sampling）是一种常用的词向量生成方法，它通过在大规模语料上训练Skip-Gram模型，采用负采样技术进行训练，

最终提取Skip-Gram模型中学习到的词向量作为最终的词向量。SGNS方法在多个自然语言处理任务中都取得了较好的效果，成为了当前最常用的词向量生成方法之一。

除了SGNS方法，SVD（奇异值分解）也是一种常用的词向量生成方法。SVD方法通过对词-文档矩阵进行奇异值分解，得到词向量矩阵。虽然SVD方法在一些任务上表现良好，但其在大规模语料上的计算复杂度较高，难以处理超大规模的语料库。

因此，为了确定哪种方法更适合汉语词向量的生成，本实验对基于SVD和SGNS两种方法构建汉语词向量进行了评测比较。通过比较两种方法生成的词向量在评测任务上的表现，可以为汉语词向量的生成提供一定的指导和参考。

2. 实验目的

本实验旨在比较基于SVD和SGNS两种方法构建汉语词向量的效果，并对其进行评测比较，以确定哪种方法更适合汉语词向量的生成。

3. 实验步骤

3.1 数据准备

我们采用的训练语料是`training.txt`，其中包含近90000个汉语单词、汉字及各种标点符号。检测语料是`pku_sim_test.txt`，其中包含500双意义相近的汉语单词，通过计算词向量之间的距离（如余弦相似度），可以评估词向量的相似性，检验词向量的训练成果。

3.2 SVD方法

预处理

`delPunctuation`函数用于移除输入文件中的标点符号，并将处理后的文本写入新文件。`string.punctuation`的预定义字符串匹配英文标点符号，`zhon.hanzi.punctuation`的预定义字符串匹配中文标点符号，将所有匹配到的标点符号去除

如果希望统计总共有多少个非零奇异值，全部奇异值之和等信息，我们必须要控制共现矩阵的大小，否则svd计算会花费大量时间。所以，在计算全部奇异值时，我们不得不舍弃一部分低频词。

```
    count = []
    count.extend(Counter(words).most_common(vocab_size))
```

```
    vocab = [v[0] for v in count]
    vocab = set(vocab)
```

这段代码首先初始化了一个空列表`count`，然后使用Python的`collections.Counter`类对`words`列表中的词语进行计数，并找出出现频率最高的`vocab_size`个词语及其频率，将这些高频词语及其频率作为一个元组列表添加到`count`中。接着，得到所有词汇的集合，采用集合的形式便于后续使用。

读取语料库

`read_corpus`函数读取处理过的文本文件，并返回一个包含所有句子的列表。

构建共现矩阵

`build_cooccurrence_matrix`函数根据给定的窗口大小计算词语间的共现次数，生成一个以词语对为键，共现次数为值的字典。为了避免真正的共现矩阵占用太大的内存，我们创建了一个稀疏矩阵（这里使用的是 COO 格式），并在转换为 CSR 格式后返回这个矩阵以及词汇表（vocab）和词语到索引的映射（word_to_index）。

```
cooccurrence_matrix = defaultdict(int)

# 计算每个词语的频率和共现频率
for i, word in enumerate(words):
    if word in vocab:
        for j in range(i - window_size, i + window_size + 1):
            if j >= 0 and j < len(words) and word != words[j]
and words[j] in vocab:
                cooccurrence_matrix[word, words[j]] += 1
......
# 初始化一个COO格式的稀疏矩阵，其行数和列数等于词汇表的大小
cmatrix = coo_matrix((vocab_size, vocab_size))

# 填充稀疏矩阵的值、行索引和列索引
rows = []
cols = []
data = []
```

```
for (i, j), value in cooccurrence_matrix.items():
    row_index = word_to_index[i]
    col_index = word_to_index[j]
    rows.append(float(row_index))
    cols.append(float(col_index))
    data.append(float(value))

# 将行索引、列索引和数据列表赋值给COO格式的稀疏矩阵
cmatrix.data = np.array(data)
cmatrix.row = np.array(rows)
cmatrix.col = np.array(cols)

# 转换为CSR格式，这样可以进行更多的稀疏矩阵操作
csr_matrix = cmatrix.tocsr()
```

SVD分解与降维

`svd_embedding` 函数接收构造好的共现矩阵，并使用 `sparse.linalg.svds` 和 `numpy.linalg.svd` 对其进行 SVD 分解。得到的 U、σ 被用来构建词嵌入矩阵，最后对词嵌入向量进行归一化处理，使其长度为1。

`numpy.linalg.svd` 和 `scipy.sparse.linalg.svds` 都是用来计算奇异值分解（Singular Value Decomposition，简称SVD）的函数，但它们之间有以下几个关键区别：

1. **适用对象**：
   - `numpy.linalg.svd` 主要设计用于稠密矩阵（Dense Matrix），即矩阵中的大部分元素都不为零的矩阵。对于完全填充的大型矩阵，虽然也能够处理，但如果矩阵规模很大，可能会遇到内存限制等问题。
   - `scipy.sparse.linalg.svds` 是专门为稀疏矩阵（Sparse Matrix）设计的，稀疏矩阵中非零元素相对较少。在处理大规模且具有高度稀疏性的矩阵时，它更为高效，因为它只处理非零元素，极大地减少了计算和存储需求。
2. **截断奇异值**：
   - `numpy.linalg.svd` 返回完整的奇异值分解结果，包括所有的左奇异向量（U）、奇异值（σ）和右奇异向量（V^H）。

- `scipy.sparse.linalg.svds` 允许用户指定计算并返回前k个最大的奇异值以及相应的奇异向量。这对于大数据集和降维应用场景非常有用，比如主成分分析（PCA）中的低秩近似，只需要关注最重要的几个奇异值即可。

在本次实验中，我在构建词向量的过程中采用了`scipy.sparse.linalg.svds` ，用它指定截断奇异值；在计算全部奇异值的过程中，我采用了`numpy.linalg.svd`

```
# 应用SVD分解并降维
def svd_embedding(csr_matrix, n_components=5):
    # 使用SVD进行降维
    U, sigma, Vt = svds(csr_matrix, k=n_components)
    # 重新组合U和Vt来形成词向量
    embeddings = np.dot(U, np.diag(sigma))

    scalar_array = np.linalg.norm(embeddings, axis=1)
    # Normalize embeddings to unit length
    embeddings = (embeddings.T / scalar_array).T

    return embeddings, sigma
```

此代码片段定义了一个名为 `svd_embedding` 的函数，该函数利用SVD对输入的稀疏矩阵进行降维处理。首先进行SVD分解得到左奇异向量 `U`、奇异值 `sigma` 以及右奇异向量转置 `Vt`。接着通过组合计算生成降维后的词向量，并对其进行单位长度归一化处理。最后返回归一化后的词向量和奇异值

```
def get_sigma(csr_matrix, choose=10):
    sigma = np.linalg.svd(csr_matrix.toarray(), compute_uv=Fa
lse)

    vocab_size = len(sigma)
    # 指定要保存的文件路径
    file_path = f"sigma/array_data_vocab_size_{vocab_size}.tx
t"

    # 使用 numpy.savetxt 函数将数组写入文本文件
    np.savetxt(file_path, sigma)
```

```
    zero_count = 0
    for i in sigma:
        if i == 0.0:
            zero_count += 1

    sum_sigma = sum(sigma)
    sum_choose = sum(sigma[:choose])
    return sigma, zero_count, sum_sigma, sum_choose
```

这段代码定义了一个名为 `get_sigma` 的函数，其功能是对输入的 CSR 稀疏矩阵执行奇异值分解（SVD），提取并处理奇异值部分。首先，通过 `np.linalg.svd` 函数获取矩阵的奇异值数组 `sigma`，注意仅计算奇异值而不计算奇异向量。然后，根据矩阵的奇异值数量（即词汇表大小）创建一个文件路径，并将奇异值数组以文本形式保存到该文件中。接下来，统计奇异值中为零的个数，并计算所有奇异值之和以及前 `choose` 个奇异值之和（`choose` 参数默认为10）。最后，函数返回奇异值数组、零值计数、总奇异值之和以及选定数量奇异值之和。

3.3 SGNS方法

读取语料库文件并预处理

使用`read_corpus`函数读取语料库文件，并进行预处理，将文本转换为单词序列，并为每个单词分配一个唯一的索引。

下采样 初始化采样表

使用`subsampling`函数对训练数据进行下采样处理，以降低高频词的影响，提高模型训练效果。

```
def subsampling(data, count):
    '''下采样 降低在训练模型时对高频词的过度关注'''
    count = [ele[1] for ele in count]
    frequency = np.array(count) / sum(count)

    # P 存储每个词调整以后的概率
```

```
    P = dict()
    for idx, x in enumerate(frequency):
        y = (math.sqrt(x / 0.001) + 1) * 0.001 / x
        P[idx] = y

    # 进行下采样
    subsampled_data = list()
    for word in data:
        if random.random() < P[word]:
            subsampled_data.append(word)
    return subsampled_data
```

该函数`subsampling`旨在对输入的文本数据进行下采样处理，通过对高频词汇设置较低的保留概率，以减少它们在训练词嵌入模型时的权重，避免过拟合。函数首先计算语料库中各词汇的频率分布，并依据公式转换为每个词汇的下采样概率。接着，遍历原始数据，以随机方式依据概率决定词汇是否保留。最终返回已下采样的词汇列表。

使用`init_sample_table`函数初始化一个采样表，用于负采样时的采样。

```
def init_sample_table(count):
    '''初始化采样表'''
    count = [ele[1] for ele in count]
    pow_frequency = np.array(count)**0.75
    power = sum(pow_frequency)
    ratio = pow_frequency / power
    # 采样表的大小
    table_size = 1e8
    # 每个词的采样次数
    count = np.round(ratio * table_size)
    sample_table = []
    for idx, x in enumerate(count):
        sample_table += [idx] * int(x)
    return np.array(sample_table)
```

该函数`init_sample_table`用于初始化一个采样表，以支持Skip-Gram模型在训练时对词汇进行采样。首先，它计算了输入词汇频率列表中每个词的0.75次方，并求和得到总功率。接着，根据总功率计算每个词在采样表中的相对比例。设定采样表大小为1亿，并根据每个词的比例确定其在采样表中的采样次数，四舍五入到整数。最后，将每个词的索引按照其采样次数复制并添加到`sample_table`列表中，最后将列表转换为NumPy数组并返回。这个采样表在训练时用于从高频词汇中随机抽取合适的负样本。

生成训练批次

使用`generate_batch`函数生成训练批次，包括正样本和负样本，以供模型训练使用。

```
data_index = 0
def generate_batch(train_data, sample_table, neg_sample_num,
window_size, batch_size):
    ......

    span = 2 * window_size + 1
    # 上下文词汇索引
    context = np.ndarray(shape=(batch_size,2 * window_size),
dtype=np.int64)
    # 当前词汇索引
    labels = np.ndarray(shape=(batch_size), dtype=np.int64)
    ......

    buffer = data[data_index:data_index + span]
    pos_u = []
    pos_v = []

    for i in range(batch_size):
        data_index += 1
        context[i,:] = buffer[:window_size]+buffer[window_siz
e+1:]
        labels[i] = buffer[window_size]
        ......

        for j in range(span-1):
```

```
            pos_u.append(labels[i])
            pos_v.append(context[i,j])
    neg_v = np.random.choice(sample_table, size=(batch_size*2
*window_size, neg_sample_num))
    return np.array(pos_u), np.array(pos_v), neg_v
```

该函数`generate_batch`用于生成词嵌入训练批次数据，它在训练数据上滑动窗口，收集每个窗口中心词及其上下文词作为正样本，并从采样表中随机抽取负样本。当窗口滑动到数据边界时会循环回到数据开头。最终返回中心词、正向上下文词和负向上下文词对应的索引数组。

构建Word2Vec模型

定义了`Word2Vec`类，该类继承自`nn.Module`，包括目标词和上下文词的嵌入层，以及前向传播函数。

```
class Word2Vec(nn.Module):
    def __init__(self, vocab_size, vector_size):
        super(Word2Vec, self).__init__()
        self.vocab_size = vocab_size
        self.vector_size = vector_size

        # Embeddings for target and context words
        self.target_embeddings = nn.Embedding(vocab_size, vec
tor_size)
        self.context_embeddings = nn.Embedding(vocab_size, ve
ctor_size)
        self.init_emb()

    def init_emb(self):
        ......

    def forward(self, u_pos, v_pos, v_neg, batch_size):
                ......
```

训练模型并保存

使用`train`函数对构建的Word2Vec模型进行训练，通过优化器更新模型参数，迭代多个epoch。

使用`save_embedding`函数保存训练得到的词向量到文件中，以便后续使用。

3.4 词向量评测

计算余弦相似度

Cosine_Similarity_test 函数读取测试文件，该文件中每行包含一对需要计算相似度的词语。函数遍历每一行，找到对应词语在词嵌入矩阵中的向量，并计算它们之间的余弦相似度。最后将相似度结果写入到 'result/svd.txt' 文件中。

```
def Cosine_Similarity_test(testpath, vocab, word_to_index, em
beddings):
    # 读取文件并解析每一行
        ......
    # 遍历每一行
    for line in lines:
        # 分割每行中的两个子词
        words = line.strip().split()

        # 检查两个子词是否都在word_embeddings中
        if len(words) == 2 and words[0] in vocab and words[1]
in vocab:
            # 获取两个子词的词向量
            vec1 = embeddings[word_to_index[words[0]]]
            vec2 = embeddings[word_to_index[words[1]]]

            # 计算余弦相似度
            dot_product = np.dot(vec1, vec2)
            norm_vec1 = np.linalg.norm(vec1)
            norm_vec2 = np.linalg.norm(vec2)
            sim_svd = dot_product

            # 将余弦相似度添加到列表中
            similarity_list.append(sim_svd)
```

```
            else:
                # 如果任一词向量不存在，设置相似度为0
                similarity_list.append(0.0)
            ......
```

这段代码定义了一个名为`Cosine_Similarity_test`的函数，其功能是从给定文件路径`testpath`中读取文本行，并计算每行中两个词语之间的余弦相似度。

函数首先读取测试文件并逐行处理。对于文件中的每一行，将行内容分割成两个子词，并检查这两个子词是否都在预定义的词汇表`vocab`中。如果满足条件，则通过`word_to_index`映射获取它们在词向量矩阵`embeddings`中的向量表示`vec1`和`vec2`。

接下来，函数计算这两个词向量之间的余弦相似度。首先计算两个向量的点积`dot_product`，然后分别计算两个向量各自的L2范数`norm_vec1`和`norm_vec2`，最后得出余弦相似度`sim_svd`并将其添加到`similarity_list`列表中。

如果一行中的任何一个词不在词汇表中，则将该对词语的相似度设置为0，并将其添加到相似度列表中。

```
Words: 包袱 段子, Cosine Similarity: 0.28840260967782905
Words: 由此 通过, Cosine Similarity: 0.9704541760832621
```

4. 模型参数和执行细节

SVD方法

由于内存限制，我们去掉部分低频词，控制词表大小为20000，使用`numpy.linalg.svd`方法进行svd分解。

- 总共有多少个非零奇异值：没有统计到0奇异值，但是有很多非常小但不为0的奇异值，小于0.01的奇异值有24个，这些奇异值可能由于计算机浮点数精度的限制而被误判为非零值
- 选取了多少个奇异值：选取了前10个奇异值
- 选取的奇异值之和：**561109.1086706324**
- 全部奇异值之和：**211065.9964873199**
- 二者的比例：**0.3761585638618003**

SGNS方法

- 所用初始词向量来源：这个模型中的词向量是在`Word2Vec`类初始化时随机初始化得到的
- 词向量维数：100
- 训练算法的学习率：0.2
- 训练批次大小：32
- 训练轮数：10

5. 结论与讨论

- 从实验结果可以看出，SGNS方法相对于SVD方法在词向量生成任务中表现更好，具有更高的准确率和相关系数。
- SGNS方法通过在大规模语料上进行训练，能够更好地捕捉词语之间的语义关系，因此在词向量生成任务中取得了更好的效果。
- 但需要注意的是，SGNS方法在训练过程中需要更多的计算资源和时间，因此在实际应用中需要计算成本和性能要求。

6. 参考文献

1. [Mikolov2013ICLRworkshop]Tomas Mikolov, Greg Corrado, Kai Chen, Jeffrey Dean, Efficient
   Estimation of Word Representation in Vector Space, ICLR2013 workshop.
2. [Mikolov2013NIPS]Tomas Mikolov, Ilya Sutskever, Kai Chen. Distributed Representations of
   Words and Phrases and their Compositionality, Advances in Neural Information Processing Systems.2013.