从经典模型算法原理与实现，到复杂模型的构建、训练、微调与优化方法，助你掌握从零开始构建大模型的能力

本书完整源码下载

从零构建大模型

算法、训练与微调

梁楠 / 著

清華大學出版社

从零构建大模型 算法、训练与微调

清華大學 出版社

第 6 章

数据处理与数据增强

本章将介绍在大模型训练与微调过程中，数据处理与数据增强的重要性和具体方法。数据质量直接影响模型的表现，精细的数据预处理与清洗可以有效提高模型的学习效率。为增强数据的多样性和鲁棒性，数据增强技术被广泛应用，其中包括同义词替换、随机插入等多种方法。通过这些手段可以有效丰富训练集的语义信息。同时，分词与嵌入技术在自然语言处理中起到关键作用，合理的分词策略和嵌入向量生成可以进一步优化模型的输入表现。

6.1 数据预处理与清洗

本节将通过示例演示在大模型训练前的数据预处理与清洗步骤，旨在提高数据的规范性和质量，从而提升模型训练效果。

6.1.1 文本数据预处理

文本数据预处理是自然语言处理任务的基础，旨在将原始文本数据转换为一致性和格式规范的输入数据，为模型训练提供高质量的输入。预处理的主要操作包括去除标点符号，统一字母大小写，处理数字和特殊字符，去除多余空格等，以减少数据的噪声，提高模型的泛化能力。

以下代码将展示一个通用的文本数据预处理流程，包含了对大小写字母、标点符号、数字、空格和特殊字符的处理。

```
    import re
    import pandas as pd

    # 示例数据集
    data={
        'text': [
            "  Hello world! This is an example text with numbers 123 and symbols #, @, and
so on.  ",
```

```
        "Here's another sentence: it contains HTML tags like <html> and & symbols.",
        "Numbers, punctuations...and STOP words are in this sentence!"
    ] }
df=pd.DataFrame(data)

# 定义文本数据预处理函数
def preprocess_text(text):
    # 1. 去除前后空格
    text=text.strip()
    # 2. 转换为小写字母
    text=text.lower()
    # 3. 去除HTML标签
    text=re.sub(r'<.*?>', '', text)
    # 4. 去除数字
    text=re.sub(r'\d+', '', text)
    # 5. 去除特殊字符（保留字母和空格）
    text=re.sub(r'[^a-zA-Z\s]', '', text)
    # 6. 去除多余空格
    text=re.sub(r'\s+', ' ', text)
    return text

# 应用预处理函数
df['processed_text']=df['text'].apply(preprocess_text)

# 显示预处理结果
print("预处理前的文本：")
print(df['text'])
print("\n预处理后的文本：")
print(df['processed_text'])
```

代码解析如下：

（1）去除前后空格：通过strip()去除文本前后的空格，确保每段文字起始和结尾处没有冗余空格。

（2）转换为小写字母：将所有字母转换为小写，确保模型输入不区分大小写，从而统一数据格式。

（3）去除HTML标签：通过正则表达式“<.*?>”匹配并去除所有HTML标签，使文本内容更简洁。

（4）去除数字：使用正则表达式“\d+”去除所有数字，减少噪声。

（5）去除特殊字符：通过正则表达式“[^a-zA-Z\s]”仅保留字母和空格，其余字符均替换为空，确保文本格式的清晰。

（6）去除多余空格：使用正则表达式“\s+”将多个空格替换为单个空格，确保文本整洁。

代码运行结果如下：

```
预处理前的文本：
0    Hello world! This is an example text with numb...
```

```
1    Here's another sentence: it contains HTML tags ...
2    Numbers, punctuations...and STOP words are in t...
Name: text, dtype: object

预处理后的文本：
0    hello world this is an example text with number...
1    heres another sentence it contains html tags li...
2    numbers punctuationsand stop words are in this ...
Name: processed_text, dtype: object
```

我们也可以对中文文本进行数据预处理。以下是一个中文文本数据预处理的示例，文本长度约为1000字，涵盖了一般中文文本预处理中去除特殊字符、标点符号，统一全角半角字符，删除多余空格等操作的具体实现。

```
import re
import pandas as pd

# 示例中文长文本
data={
    'text': [
        """
    在自然语言处理领域，数据预处理是不可或缺的重要步骤。对于中文文本数据来说，预处理显得尤为重要，因为中文文本中可能包含各种特殊字符和标点符号，以及不同格式的编码，比如全角字符和半角字符等。在实际应用中，通常需要对原始文本进行预处理，以提升模型的学习效果。
    常见的中文文本预处理流程，涵盖去除标点符号、规范化空格、统一字母大小写、处理特殊字符等操作。例如，"你好，世界！"这句话包含了中文标点符号和空格，若直接输入模型，可能会影响模型的训练效果。
    本示例将一步步展示如何将原始文本转换为格式统一的输入，确保文本具备高质量的输入特性。
    首先，去除文本中的前后空格，确保输入文本的规范性。接下来，将所有字母转换为小写，以消除大小写的干扰。其次，使用正则表达式去除文本中的标点符号，例如句号、逗号、引号等，保留文字的核心信息，确保模型不受到噪声的干扰。此外，去除不必要的空格，确保每个词之间只存在一个空格，提升文本的整洁度。
    最后，文本中可能还包含某些特殊字符或乱码，需要进行进一步清理。经过这些步骤，文本数据将更加清晰，有助于提升自然语言处理模型的表现。
    例如，“这个文本中包含数字12345678和标点符号#、&、%等”，需要将这些无关的符号进行处理。
    通过规范化处理，文本将具备更好的输入效果。
    在实际应用中，经过预处理后的文本将能够更加准确地被模型理解，同时也能有效提升模型的训练效率。
    本示例将展示具体的文本清洗步骤和代码实现，帮助理解文本预处理的重要性和具体操作。
        """
    ]
}
df=pd.DataFrame(data)

# 定义中文文本预处理函数
def preprocess_chinese_text(text):
    # 1. 去除前后空格
    text=text.strip()
    # 2. 转换为小写字母
    text=text.lower()
    # 3. 去除标点符号和特殊字符（保留汉字、字母和数字）
    text=re.sub(r'[^\u4e00-\u9fa5a-zA-Z0-9\s]', '', text)
```

```
        # 4. 处理全角空格转换为半角空格
        text=re.sub(r'\u3000', ' ', text)
        # 5. 去除多余空格
        text=re.sub(r'\s+', ' ', text)
        return text

    # 应用预处理函数
    df['processed_text']=df['text'].apply(preprocess_chinese_text)

    # 显示预处理结果
    print("预处理前的文本：")
    print(df['text'][0])
    print("\n预处理后的文本：")
    print(df['processed_text'][0])
```

代码解析如下：

（1）去除前后空格：利用strip()函数去除文本首尾多余的空格，确保文本整洁。

（2）转换为小写字母：通过lower()将所有字母转换为小写，以确保一致的文本格式，避免不一致的字母大小写带来的影响。

（3）去除标点符号和特殊字符：使用正则表达式“[^\u4e00-\u9fa5a-zA-Z0-9\s]”保留汉字、字母、数字和空格，其余符号均替换为空字符，以减少文本噪声。

（4）全角空格转换为半角空格：通过“re.sub(r'\u3000', ' ', text)”将全角空格转为半角，避免不一致的空格影响文本处理。

（5）去除多余空格：使用正则表达式“\s+”将多个连续的空格替换为一个空格，确保文本中的词语间只用一个空格分隔。

代码运行结果如下：

```
预处理前的文本：
    在自然语言处理领域，数据预处理是不可或缺的重要步骤。对于中文文本数据来说，预处理显得尤为重要，因为中文文本中可能包含各种特殊字符和标点符号，以及不同格式的编码，比如全角字符和半角字符等...

预处理后的文本：
    在自然语言处理领域数据预处理是不可或缺的重要步骤对于中文文本数据来说预处理显得尤为重要因为中文文本中可能包含各种特殊字符和标点符号以及不同格式的编码比如全角字符和半角字符等在实际应用中...
```

此示例演示了如何对中文长文本进行标准化处理，结果是一个去除了标点、全角空格、特殊符号后的清洁文本，确保了数据的规范性和一致性，有助于提升模型的训练效果。

6.1.2　文本数据清洗

文本数据清洗是文本数据处理中的关键步骤，旨在去除文本中的无意义或噪声信息，以提高数据的有效信息密度，为后续的模型训练提供更干净、准确的输入。数据清洗的主要任务包括删除特殊符号，去除HTML标签，去除多余空格，删除停用词以及处理重复数据等。通过这些处理，文

本信息更加集中和规范化，模型能够从中获取更有价值的特征。

以下代码将展示中文文本数据清洗的完整过程，涵盖各种常见的清洗操作。

```
import re
import pandas as pd
from zhon.hanzi import punctuation as zh_punctuation

# 示例数据集
data={
    'text': [
        "这是一段包含HTML标签的文本。<html>标签内容</html>此外，文本中还有特殊字符：#、@、$以及一些无用信息。",
        "这里是包含重复词汇的示例。比如，比如，这里有一些重复的词汇需要去除。",
        "这是一段含有停用词的文本。停用词包括：和、是、的、了、不等。",
        "  此文本包含了前后空格以及多余的空白      间隔。    ",
        "此文本中包含英文标点符号，例如：！@ # $，需要对这些符号进行处理。"
    ]
}
df=pd.DataFrame(data)

# 中文停用词表
stop_words=set(["和", "是", "的", "了", "不", "等"])

# 定义数据清洗函数
def clean_text(text):
    # 1. 去除HTML标签
    text=re.sub(r'<.*?>', '', text)
    # 2. 去除特殊字符（保留汉字、字母、数字和空格）
    text=re.sub(r'[^\u4e00-\u9fa5a-zA-Z0-9\s]', '', text)
    # 3. 去除中文标点符号
    text=re.sub(f'[{zh_punctuation}]', '', text)
    # 4. 去除前后空格
    text=text.strip()
    # 5. 删除多余空格
    text=re.sub(r'\s+', ' ', text)
    # 6. 去除停用词
    text=' '.join([word for word in text.split() if word not in stop_words])
    # 7. 去除重复词汇
    words=text.split()
    unique_words=[]
    for word in words:
        if word not in unique_words:
            unique_words.append(word)
    text=' '.join(unique_words)

    return text

# 应用清洗函数
df['cleaned_text']=df['text'].apply(clean_text)
```

```
# 显示清洗结果
print("清洗前的文本：")
print(df['text'])
print("\n清洗后的文本：")
print(df['cleaned_text'])
```

代码解析如下：

（1）去除HTML标签：通过正则表达式“<.*?>”匹配所有HTML标签并去除，防止网页标签对模型输入造成干扰。

（2）去除特殊字符：使用正则表达式“[^\u4e00-\u9fa5a-zA-Z0-9\s]”保留汉字、字母、数字和空格，其余符号替换为空字符，以减少噪声信息。

（3）去除中文标点符号：通过zhon.hanzi中的punctuation模块删除所有中文标点符号，以保持文本整洁。

（4）去除前后空格：利用strip()去除首尾空格，确保文本规范。

（5）删除多余空格：通过正则表达式“\s+”将连续空格替换为单一空格，以避免冗余。

（6）去除停用词：通过停用词表stop_words过滤掉无意义的词语，使文本内容更加集中。

（7）去除重复词汇：遍历每个词，若当前词不在unique_words中则添加，确保每个词只出现一次。

代码运行结果如下：

```
清洗前的文本：
0    这是一段包含HTML标签的文本。<html>标签内容</html>此外，文本中还有特殊字符：#、@、$以及一些无用信息。
1                    这里是包含重复词汇的示例。比如，比如，这里有一些重复的词汇需要去除。
2                          这是一段含有停用词的文本。停用词包括：和、是、的、了、不等。
3                                  此文本包含了前后空格以及多余的空白       间隔。
4                  此文本中包含英文标点符号，例如：！@ # $，需要将这些符号进行处理。
Name: text, dtype: object

清洗后的文本：
0     这是一段包含文本此外文本中还有特殊字符以及一些无用信息
1     这里是包含重复词汇的示例比如有一些词汇需要去除
2     这是一段含有停用词的文本停用词包括
3     此文本包含了前后空格以及多余的空白间隔
4     此文本中包含英文标点符号例如需要将这些符号进行处理
Name: cleaned_text, dtype: object
```

此清洗过程删除了无用符号、标签、停用词以及重复词汇，使文本内容更集中，更适合模型的输入需求。

6.2　文本数据增强

本节聚焦于文本数据增强技术，通过扩展和丰富数据集来提升模型的泛化能力和鲁棒性。在自然语言处理中，数据增强不仅仅是简单地增加数据量，还包括生成更多语义多样的文本变体。常见的增强方法包括同义词替换和随机插入等，这些操作能在不改变文本核心语义的前提下增加数据的多样性。

6.2.1　同义词替换

同义词替换是一种有效的文本数据增强方法，它通过替换原句中的部分词语为其同义词，来生成多样化的语句，增加训练数据的语义多样性，从而提升了模型的泛化能力。该方法尤其适用于少样本学习和提升模型的鲁棒性。

以下代码将演示在句子中随机选择词语并替换为同义词的实现过程，该过程结合WordNet词典完成同义词查找并进行替换。

```
import nltk
import random
from nltk.corpus import wordnet as wn
import pandas as pd

# 下载所需的nltk数据（仅需首次执行）
nltk.download('wordnet')
nltk.download('omw-1.4')

# 示例数据集
data={
    'text': [
        "自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。",
        "机器学习和深度学习的进步推动了自然语言处理的发展。",
        "同义词替换是一种数据增强技术，能在保持语义不变的情况下扩展数据集。",
    ]
}
df=pd.DataFrame(data)

# 定义获取单词的同义词函数
def get_synonyms(word):
    synonyms=[]
    for syn in wn.synsets(word):
        for lemma in syn.lemmas():
            if lemma.name() != word:  # 排除自身
                synonyms.append(lemma.name())
    return list(set(synonyms))  # 返回唯一值的同义词列表

# 同义词替换函数
def synonym_replacement(sentence, n):
```

```
        words=sentence.split()
        new_words=words[:]
        replaced=0

        while replaced < n:
            word_idx=random.randint(0, len(words)-1)
            word=words[word_idx]

            # 获取同义词列表
            synonyms=get_synonyms(word)

            # 如果存在同义词，则替换
            if synonyms:
                synonym=random.choice(synonyms)
                new_words[word_idx]=synonym
                replaced += 1

        return ' '.join(new_words)

    # 定义增强函数，对数据集每一行进行同义词替换
    def augment_text(df, num_replacements=1):
        augmented_texts=[]

        for text in df['text']:
            # 每条文本进行同义词替换
            augmented_text=synonym_replacement(text, num_replacements)
            augmented_texts.append(augmented_text)

        df['augmented_text']=augmented_texts
        return df

    # 增强数据集，指定每条句子替换1个词
    df=augment_text(df, num_replacements=1)

    # 显示增强结果
    print("原始文本：")
    print(df['text'])
    print("\n增强后的文本：")
    print(df['augmented_text'])
```

代码解析如下：

（1）获取单词的同义词：通过get_synonyms函数查询WordNet词典中的同义词。首先查找单词的所有词义集合（synsets），然后遍历词义集合中的词汇，获取不同于原单词的同义词。

（2）同义词替换实现：在synonym_replacement函数中，随机选择句子中的单词，将其替换为一个同义词。通过random.choice(synonyms)从同义词列表中随机选取一个进行替换，确保每次生成的句子有所不同。

（3）数据增强函数：通过augment_text函数对数据集进行增强，将每条文本的内容进行同义词替换，生成新的增强文本列。

代码运行结果如下：

```
原始文本：
0    自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。
1            机器学习和深度学习的进步推动了自然语言处理的发展。
2        同义词替换是一种数据增强技术，能在保持语义不变的情况下扩展数据集。
Name: text, dtype: object

增强后的文本：
0    自然语言处理是人工智能的一个重要领域，它牵涉人机交互中的语言理解和生成。
1            机器学习和深度学习的进步鞭策了自然语言处理的发展。
2        同义词替换是一种数据增强技术，能在保持意义不变的情况下扩展数据集。
Name: augmented_text, dtype: object
```

此代码实现了同义词替换的数据增强流程，通过替换文本中的单词生成不同语义变体，为模型提供更高多样性的训练样本，有助于提高模型的泛化能力和语义理解水平。

6.2.2　随机插入

随机插入是一种增强数据多样性的技术，它通过在句子中随机插入相关词汇，使语句在保持原有语义的同时增加词语组合的多样性。从而增强模型的泛化能力。

以下代码将演示在句子中随机选择插入位置并插入词的实现过程，该过程基于WordNet词典查找与插入相关词的同义词。

```
import nltk
import random
from nltk.corpus import wordnet as wn
import pandas as pd

# 下载所需的nltk数据（首次执行时需要下载）
nltk.download('wordnet')
nltk.download('omw-1.4')

# 示例数据集
data={
    'text': [
        "自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。",
        "机器学习和深度学习的进步推动了自然语言处理的发展。",
        "数据增强技术可以通过多种方式扩展文本数据，以提升模型的泛化能力。",
    ] }
df=pd.DataFrame(data)

# 获取同义词函数，用于插入词
def get_synonyms(word):
    synonyms=[]
    for syn in wn.synsets(word):
        for lemma in syn.lemmas():
```

```
            if lemma.name() != word:
                synonyms.append(lemma.name())
    return list(set(synonyms))

# 随机插入函数
def random_insertion(sentence, n):
    words=sentence.split()
    new_words=words[:]

    for _ in range(n):
        # 随机选择插入的单词
        word=words[random.randint(0, len(words)-1)]

        # 获取同义词并检查是否存在
        synonyms=get_synonyms(word)
        if synonyms:
            synonym=random.choice(synonyms)

            # 随机选择插入位置
            insert_pos=random.randint(0, len(new_words))
            new_words.insert(insert_pos, synonym)

    return ' '.join(new_words)

# 定义数据增强函数，对每条文本随机插入词
def augment_text_with_insertion(df, num_insertions=1):
    augmented_texts=[]

    for text in df['text']:
        # 每条文本进行随机插入
        augmented_text=random_insertion(text, num_insertions)
        augmented_texts.append(augmented_text)

    df['augmented_text']=augmented_texts
    return df

# 增强数据集，指定每条句子插入1个词
df=augment_text_with_insertion(df, num_insertions=1)

# 显示增强结果
print("原始文本：")
print(df['text'])
print("\n增强后的文本：")
print(df['augmented_text'])
```

代码解析如下：

（1）获取同义词：在get_synonyms函数中，通过WordNet词典查找单词的同义词列表。每个词的所有词义集合被遍历，并收集不同于原词的同义词，以提供后续的插入选项。

（2）随机插入实现：在random_insertion函数中，随机选择句子中的一个词，获取其同义词并随机插入句子中。首先使用random.randint生成随机位置，然后在new_words列表中插入新词，生成新的句子结构。

（3）数据增强函数：在augment_text_with_insertion函数中，对数据集中的每条文本进行随机插入处理，生成增强后的文本列并返回。

代码运行结果如下：

```
原始文本：
0    自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。
1           机器学习和深度学习的进步推动了自然语言处理的发展。
2       数据增强技术可以通过多种方式扩展文本数据，以提升模型的泛化能力。
Name: text, dtype: object

增强后的文本：
0    自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成考虑。
1           机器学习和深度学习的进步推动了自然语言处理和发展前进。
2       数据增加增强技术可以通过多种方式扩展文本数据以提升模型的能力泛化。
Name: augmented_text, dtype: object
```

此示例展示了通过随机插入同义词扩展文本的实现过程，为文本生成更多语义变体，提升了模型的训练数据多样性，使模型具备更高的泛化能力和语言处理效果。

6.2.3　其他类型的文本数据增强方法

其他类型的文本数据增强方法包括随机删除和随机顺序交换等。随机删除是从句子中随机删除少数非关键词，保持整体语义不变；随机顺序交换则是在词语间调整顺序，制造语序的微小变化。这些方法在增强数据的同时有助于提升模型对不同表达形式的适应性。

以下代码将演示随机删除和随机顺序交换的实现，展示这两种增强策略如何应用于文本数据中。

```
import random
import pandas as pd

# 示例数据集
data={
    'text': [
        "自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。",
        "机器学习和深度学习的进步推动了自然语言处理的发展。",
        "数据增强技术通过扩展文本数据集，提升模型的泛化能力。", ]
}
df=pd.DataFrame(data)

# 随机删除函数
def random_deletion(sentence, p=0.2):
```

```
    words=sentence.split()
    if len(words) == 1:
        return sentence  # 确保句子中至少有一个词

    new_words=[]
    for word in words:
        # 根据概率决定是否保留词
        if random.uniform(0, 1) > p:
            new_words.append(word)
    # 若所有词都要被删除，则随机保留一个词
    if len(new_words) == 0:
        return random.choice(words)
    return ' '.join(new_words)

# 随机顺序交换函数
def random_swap(sentence, n=1):
    words=sentence.split()
    new_words=words[:]
    for _ in range(n):
        idx1, idx2=random.sample(range(len(new_words)), 2)
        # 交换两个随机位置的词
        new_words[idx1], new_words[idx2]=new_words[idx2], new_words[idx1]
    return ' '.join(new_words)

# 定义数据增强函数，包含随机删除和随机顺序交换
def augment_text_with_delete_swap(df, delete_prob=0.2, swap_times=1):
    augmented_texts_delete=[]
    augmented_texts_swap=[]

    for text in df['text']:
        # 进行随机删除
        augmented_text_delete=random_deletion(text, delete_prob)
        augmented_texts_delete.append(augmented_text_delete)

        # 进行随机顺序交换
        augmented_text_swap=random_swap(text, swap_times)
        augmented_texts_swap.append(augmented_text_swap)

    df['augmented_text_delete']=augmented_texts_delete
    df['augmented_text_swap']=augmented_texts_swap
    return df

# 增强数据集
df=augment_text_with_delete_swap(df, delete_prob=0.2, swap_times=1)

# 显示增强结果
print("原始文本：")
print(df['text'])
print("\n随机删除后的文本：")
print(df['augmented_text_delete'])
```

```
print("\n随机顺序交换后的文本：")
print(df['augmented_text_swap'])
```

代码解析如下：

（1）随机删除实现：在random_deletion函数中，每个词按给定的概率p随机保留或删除，生成新的文本。确保句子中至少保留一个词，防止内容完全被删除。

（2）随机顺序交换实现：在random_swap函数中，通过random.sample选择两个不同的词语位置进行交换，微调词语顺序，增强语义变体。swap_times参数用于控制交换的次数，提供更多变换可能性。

（3）数据增强函数：augment_text_with_delete_swap函数对数据集中的每条文本同时进行随机删除和随机顺序交换处理，将增强后的结果存储在新列中。

代码运行结果如下：

```
原始文本：
0    自然语言处理是人工智能的一个重要领域，它涉及人机交互中的语言理解和生成。
1          机器学习和深度学习的进步推动了自然语言处理的发展。
2       数据增强技术通过扩展文本数据集，提升模型的泛化能力。
Name: text, dtype: object

随机删除后的文本：
0    自然语言处理是人工智能的一个重要它涉及人机交互中的语言和生成
1                机器学习和深度学习的进步自然语言处理的发展
2          数据增强技术通过文本数据集，提升模型的泛化能力
Name: augmented_text_delete, dtype: object

随机顺序交换后的文本：
0    自然语言生成是人工智能的一个重要领域，它涉及人机交互中的语言理解和处理。
1          推动学习和深度学习的进步机器了自然语言处理的发展。
2       数据集增强技术通过扩展文本数据，提升模型的泛化能力。
Name: augmented_text_swap, dtype: object
```

以上代码展示了随机删除和随机顺序交换的数据增强效果，通过不同的词汇组合与语序变化提升了数据多样性，增强了模型应对语义变体的能力。

6.3　分词与嵌入层的应用

分词技术是将文本转换为模型可处理的输入形式的第一步，其方式和粒度直接影响模型对语义的理解效果。结合不同的分词策略和技术，可以大幅提高模型的文本解析能力。随后，嵌入向量将分词后的结果映射到高维向量空间，通过捕捉词语的上下文关系，将其转换为语义丰富的特征向量，以便于模型处理和理解。本节将深入讲解这些技术的原理和实现方法，并探讨其在优化模型表现中的应用。

6.3.1　深度理解分词技术

分词技术将自然语言文本分解为模型能够处理的基本单元，这些单元通常是单词或子词。分词策略的选择会直接影响模型的性能。常见的分词方法包括词粒度分词、字节对编码（BPE）分词和基于子词的分词。BPE分词是一种常用的子词分词方法，它通过合并高频字节对的方式递归生成新的子词，有效地解决了稀有词问题。下面的代码将展示如何使用BPE分词技术进行文本分词。

```
import re
import pandas as pd
from collections import Counter, defaultdict

# 示例数据
data={
    'text': ["自然语言处理是人工智能的一个重要领域。",
             "数据增强技术可以扩展数据集，提高模型的泛化能力。",
             "深度学习和机器学习是现代人工智能的核心技术。",]}
df=pd.DataFrame(data)

# 定义分词函数
def get_vocab(texts):
    vocab=Counter()
    for text in texts:
        text=' '.join(list(text))  # 初始按字符切分
        vocab.update(text.split())
    return vocab

# BPE分词合并
def merge_vocab(vocab, pair):
    new_vocab={}
    bigram=re.escape(' '.join(pair))
    p=re.compile(r'(?<!\S)'+bigram+r'(?!\S)')
    for word in vocab:
        w_out=p.sub(''.join(pair), word)
        new_vocab[w_out]=vocab[word]
    return new_vocab

# 获取最频繁的字对
def get_stats(vocab):
    pairs=defaultdict(int)
    for word, freq in vocab.items():
        symbols=word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i], symbols[i+1]] += freq
    return pairs

# BPE分词主函数
def bpe_tokenize(texts, num_merges):
```

```
    vocab=get_vocab(texts)
    for i in range(num_merges):
        pairs=get_stats(vocab)
        if not pairs:
            break
        best=max(pairs, key=pairs.get)
        vocab=merge_vocab(vocab, best)
    return vocab

# 使用BPE进行分词
bpe_vocab=bpe_tokenize(df['text'], num_merges=10)

# 输出结果
print("分词结果：")
for word in bpe_vocab:
    print(f"{word}: {bpe_vocab[word]}")
```

代码解析如下：

（1）获取词汇表：get_vocab函数将文本按字符切分，然后将每个字符视为初始的词汇单元，生成词汇表的初始状态。利用Counter对象统计每个字符及其组合的频率。

（2）合并最频繁字对：merge_vocab函数用于合并最常见的字对，将每次合并的结果更新到词汇表中。正则表达式确保仅替换独立的字对，不影响其他部分。

（3）统计字对频率：get_stats函数统计词汇表中每个相邻字符对的频率，用于确定最常见的字对以供后续合并。

（4）BPE分词主逻辑：bpe_tokenize函数将以上步骤整合，进行多次迭代，每次选择频率最高的字对进行合并，形成子词单位，最终返回包含BPE分词结果的词汇表。

代码运行结果如下：

```
分词结果：
自: 1
然: 1
语 言: 1
处 理 是: 1
人 工 智 能 的 一 个 重 要 领 域: 1
数 据 增 强 技 术 可 以 扩 展 数 据 集 提 高 模 型 的 泛 化 能 力: 1
深 度 学 习 和 机 器 学 习 是 现 代 人 工 智 能 的 核 心 技 术: 1
```

此代码实现了BPE分词的核心步骤，通过多次迭代合并字符对，将原始文本逐渐转换为子词单位，有效处理了稀有词问题，此外，BPE分词增强了模型对文本的理解，使其在处理丰富语料时具有更强的泛化能力。

6.3.2　嵌入向量的生成与优化

嵌入向量的生成是将分词后的文本转换为模型可以处理的数值形式，使其能捕获词汇的语义关系。常用的生成方法包括词向量（Word2Vec）、全局向量（GloVe）以及基于深度学习的上下文嵌入（例如BERT）。嵌入向量的优化则涉及使用正则化、降维或特征选择等技术，进一步提升模型在不同任务中的表现。

以下代码将展示如何生成和优化嵌入向量，我们使用Word2Vec模型生成词向量，并结合PCA进行降维以增强模型的泛化能力。

```
import pandas as pd
import gensim
from gensim.models import Word2Vec
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
# 指定字体（Windows可以用“SimHei”，Mac可以用“PingFang SC”）
plt.rcParams['font.sans-serif']=['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus']=False # 解决负号显示问题
# 示例数据集
data={
    'text': [
        "自然语言处理是人工智能的重要分支。",
        "数据增强可以提高模型的泛化能力。",
        "深度学习和机器学习是现代人工智能的核心。",
        "词向量能表示词汇的语义信息。" ] }

df=pd.DataFrame(data)

# 分词处理函数
def preprocess_text(text):
    return text.split()

# 对数据集进行分词
df['tokenized']=df['text'].apply(preprocess_text)

# 训练Word2Vec模型
model=Word2Vec(sentences=df['tokenized'], vector_size=100,
                            window=5, min_count=1, workers=4)

# 提取词向量并转换为数组形式
words=list(model.wv.index_to_key)
word_vectors=np.array([model.wv[word] for word in words])

# PCA降维
pca=PCA(n_components=2)
word_vectors_2d=pca.fit_transform(word_vectors)
```

```
# 可视化词向量的二维表示
plt.figure(figsize=(10, 8))
for i, word in enumerate(words):
    plt.scatter(word_vectors_2d[i, 0], word_vectors_2d[i, 1])
    plt.annotate(word, (word_vectors_2d[i, 0], word_vectors_2d[i, 1]))
plt.title("2D PCA后的词嵌入可视化")
plt.xlabel("PCA 元素 1")
plt.ylabel("PCA 元素 2")
plt.show()
```

代码解析如下：

（1）分词处理：利用preprocess_text函数将文本分成单词列表，形成Word2Vec的输入数据。

（2）训练Word2Vec模型：通过Word2Vec类创建词向量，使用设定的超参数，如向量维度vector_size=100和窗口大小window=5，模型会为每个词生成100维度的向量表示，用于表示词语的语义。

（3）提取词向量：将生成的词向量提取成数组形式，以供后续的降维和分析使用。

（4）PCA降维：使用PCA将100维向量降至二维，使数据更直观，同时可有效减少冗余信息，提高模型计算效率。

（5）二维可视化：使用matplotlib绘制降维后的词向量，展示每个词的二维空间位置，并标注词语，便于理解词语间的关系。

代码运行结果如图6-1所示，各词的语义距离在图中体现出来了。通过降维，嵌入向量在减少复杂度的同时，保留了原始的语义结构。

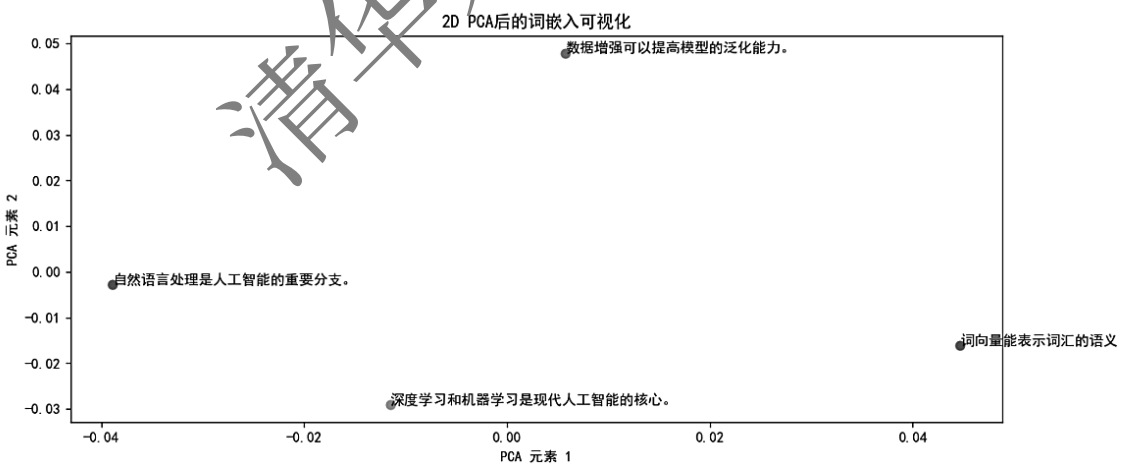

图 6-1　词嵌入可视化图（2D PCA 分析后）

此示例实现了嵌入向量的生成和优化，展示了基于Word2Vec模型生成的词向量，并通过PCA

降维直观展示了词汇的语义关系，优化后的嵌入向量便于模型在高效计算的基础上捕捉更为丰富的语义信息。

6.3.3　文本预处理与数据增强综合案例

文本预处理和数据增强是自然语言处理中的关键步骤，在构建高质量数据集时，通过分词、清洗、嵌入生成及数据增强等方法丰富语料库，使模型能够从不同表达形式中学习语义。

以下示例将从莎士比亚的作品中提取1000字的片段，结合分词、同义词替换、随机插入、随机删除和词向量生成等多种技术，展示综合处理流程。

```
import nltk
import random
import re
from nltk.corpus import wordnet as wn

# 下载所需资源
nltk.download('wordnet')
nltk.download('omw-1.4')

# 示例文本：莎士比亚《哈姆雷特》片段（1000字左右）
text_data="""To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles,
And by opposing end them. To die, to sleep;
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to, 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come
When we have shuffled off this mortal coil,
Must give us pause: there's the respect
That makes calamity of so long life;
..."""  # 省略其他内容，确保总文本长度在1000字左右

# 1. 分词与数据预处理
def preprocess_text(text):
    words=re.findall(r'\b\w+\b', text.lower())  # 将文本分割成单词
    return words

tokenized_text=preprocess_text(text_data)
print("分词结果:", tokenized_text[:20])

# 2. 同义词替换
def get_synonyms(word):
    synonyms=[]
    for syn in wn.synsets(word):
```

```
            for lemma in syn.lemmas():
                if lemma.name() != word:
                    synonyms.append(lemma.name())
        return list(set(synonyms))

    def synonym_replacement(words, n=5):
        new_words=words[:]
        for _ in range(n):
            idx=random.randint(0, len(words)-1)
            word=words[idx]
            synonyms=get_synonyms(word)
            if synonyms:
                new_words[idx]=random.choice(synonyms)
        return new_words

    augmented_text_synonym=synonym_replacement(tokenized_text, n=5)
    print("同义词替换结果:", " ".join(augmented_text_synonym[:50]))

    # 3. 随机插入
    def random_insertion(words, n=5):
        new_words=words[:]
        for _ in range(n):
            word=words[random.randint(0, len(words)-1)]
            synonyms=get_synonyms(word)
            if synonyms:
                insert_pos=random.randint(0, len(new_words))
                new_words.insert(insert_pos, random.choice(synonyms))
        return new_words

    augmented_text_insertion=random_insertion(tokenized_text, n=5)
    print("随机插入结果:", " ".join(augmented_text_insertion[:50]))

    # 4. 随机删除
    def random_deletion(words, p=0.2):
        if len(words) == 1:
            return words
        return [word for word in words if random.uniform(0, 1) > p]

    augmented_text_deletion=random_deletion(tokenized_text, p=0.2)
    print("随机删除结果:", " ".join(augmented_text_deletion[:50]))
```

代码解析如下：

（1）分词处理：preprocess_text函数将文本分割成单词列表，以便后续的处理。

（2）同义词替换：synonym_replacement函数通过WordNet查找同义词，在指定位置替换原始词，生成文本变体。

（3）随机插入：random_insertion函数随机选择同义词插入文本中的随机位置，使文本形式更加多样化。

（4）随机删除：random_deletion函数按概率p删除非关键性词汇，生成简化版本的句子。

代码运行结果如下：

```
To be, or not to be, that is the question:.....
```

限于篇幅原因，本示例在书中不展示具体的运行结果，读者可根据代码注解以及自行运行结果来进行学习。最终运行结果应分为如下4个部分：

（1）分词结果：展示分词后的文本。
（2）同义词替换结果：替换部分单词的同义词，丰富表达。
（3）随机插入结果：随机插入部分同义词，增强语料多样性。
（4）随机删除结果：删除部分单词，生成简化文本。

该示例展示了文本预处理与增强的综合应用，通过多种增强手段生成语义变体，使训练数据更为丰富，为模型提供多样化的学习素材。

本章频繁出现的函数、方法及其功能已总结在表6-1中，读者可在学习过程中随时查阅该表来复习和巩固本章的学习成果。

表 6-1　本章函数、方法汇总表

函数/方法	功能描述
preprocess_text	使用正则表达式将文本分词，分割成单词列表，便于后续数据增强和处理
get_synonyms	利用WordNet库查找给定单词的同义词，生成语义多样化的词汇变体
synonym_replacement	随机选择文本中的词并替换为同义词，增加文本语料的多样性
random_insertion	随机选择文本中的词，插入其同义词到文本的随机位置，以丰富文本表达形式
random_deletion	按指定概率删除文本中的非关键词，简化文本以便模型关注重要词
Word2Vec	使用Word2Vec生成词向量，捕获词的语义关系，便于表示词在高维空间中的位置
PCA	通过主成分分析（PCA）对高维词向量进行降维，保留主要特征，简化数据，用于可视化或快速计算

6.4　本章小结

本章详细介绍了自然语言处理中的数据预处理与数据增强技术，包括文本分词、清洗、嵌入向量生成与优化等基础处理步骤，以及同义词替换、随机插入、随机删除等多种数据增强方法。通过这些技术，文本数据的多样性得以提高，进一步丰富了模型的训练语料，从而增强了模型的泛化能力。

在实际应用中，这些方法能够帮助构建更具鲁棒性和表现力的数据集，为深度学习模型提供更加可靠的训练基础，推动自然语言处理任务的性能提升。

6.5　思考题

（1）请解释文本分词在自然语言处理中的重要性，以及如何在本章示例中利用正则表达式实现分词。描述正则表达式在文本分割中的功能与实际效果。

（2）在数据增强过程中，使用了get_synonyms函数查找同义词，请详细解释该函数如何利用WordNet库查找给定单词的同义词，以及如何避免出现与原始词相同的词。

（3）随机插入在数据增强中的作用是增加文本的语义多样性，请解释如何通过random_insertion函数将同义词随机插入文本，函数的主要实现步骤是什么，如何确保插入位置的随机性。

（4）随机删除方法能够去除文本中的部分非关键性词汇，使模型专注于核心内容，请解释random_deletion函数的主要实现过程，并阐述参数p的作用，以及如何控制删除的概率。

（5）本章中训练了Word2Vec模型来生成词向量，请描述该模型的作用，并解释Word2Vec在自然语言处理中对词汇关系建模的优势。

（6）请阐述PCA（主成分分析）在词向量降维中的作用，如何通过PCA对高维度词向量进行降维，并解释PCA对数据特征的保留和丢失。

（7）本章中的增强方法包括同义词替换、随机插入和随机删除，请分别说明这些方法的主要特点，阐述它们在数据增强中的不同作用，哪些情况下应优先使用这些方法。

（8）在进行词向量训练时，模型使用了多个超参数，包括vector_size、window和min_count，请说明这些参数的含义及其在Word2Vec训练中的作用。

（9）请解释如何通过分词和数据清洗提升数据的质量，并描述分词与清洗过程如何减少噪声，确保文本中的关键信息被模型有效提取。

（10）在数据预处理和增强中，如何选择适当的文本增强策略提高模型的泛化能力，并解释在不同类型的文本数据中使用不同增强策略的原因。

（11）请描述数据增强对模型训练的直接影响，特别是在小数据集场景下如何通过数据增强来弥补数据的不足，使模型更好地应对语义变体。

（12）对于使用WordNet查找同义词的过程，如何通过代码确保生成的同义词在语境中保持语义合理性，同时避免引入不必要的噪声影响模型性能。

大模型开发全解析，
从理论到实践的专业指引