第 3 章 知识抽取技术

本章学习目标

（1）了解知识抽取的概念及其在信息处理中的作用。

（2）掌握如何从自然语言文本中抽取实体、关系、事件等多个知识图谱要素。

（3）掌握基于不同原理的知识抽取方法，如基于规则、机器学习和深度学习的方法。

本章围绕知识抽取技术展开，详细介绍了从自然语言文本中识别、抽取知识的关键方法。针对知识抽取的基本概念和阶段划分进行了说明，帮助读者理解知识抽取在信息处理及知识图谱构建中的基础作用。本章深入探讨了三种主要的知识图谱要素抽取方法：实体抽取、关系抽取与事件抽取。每种方法均涵盖基于规则、机器学习和深度学习等不同原理的实现方式，展示了各自的优势与局限性。在实体抽取、关系抽取和事件抽取的部分，不仅阐述了核心技术的应用背景，还分析了最新的研究进展和面临的挑战。通过本章内容，读者将对知识抽取的核心技术及其在实际应用中的可能性和潜力形成全面而深入的认识。

3.1 知识抽取的概念及阶段划分

知识抽取的概念最早于 20 世纪 70 年代后期出现在自然语言处理领域，指自动化地从文本中发现和抽取相关信息，并将多个文本碎片中的信息进行合并，将非结构化数据转换为结构化数据。随着知识图谱的发展，知识抽取在 21 世纪初期开始兴起，尤其是在 2010 年后迅速发展。知识图谱通过构建实体和关系的网络，能够更好地组织和利用知识，推动了知识抽取技术的广泛应用和研究。例如，谷歌在 2012 年推出的知识图谱显著提升了搜索引擎的智能化水平，进一步促进了知识抽取技术的发展和普及。知识抽取不仅被用于搜索引擎，还广泛应用于信息检索、推荐系统、问答系统等领域，成为现代信息处理和人工智能的重要组成部分，图 3.1 介绍了知识抽取的不同阶段。



图 3.1 知识抽取的不同阶段

▶ 3.1.1　数据预处理阶段

数据预处理是知识抽取过程中至关重要的阶段，主要包括数据收集和数据清洗两个步骤。其目标是确保数据的质量和可用性，为后续的特征提取、分析和模型训练奠定坚实的基础。数据收集是数据预处理的起点，旨在从各种来源获取相关的结构化或非结构化数据。这些来源可以包括网页、数据库、文档等。收集数据时，需要明确数据的范围和来源，选择合适的收集策略，并确保所收集的数据具有足够的质量和可用性。高质量的数据收集为后续步骤的顺利进行奠定了基础。

数据清洗是数据预处理的关键步骤，目的是去除重复、无关或错误的数据，确保数据集的高质量和高可用性。去重处理可以消除数据集中的重复条目，避免在后续分析中引入偏差，保证结果的准确性。同时，清除无关或错误的数据可以提高数据集的相关性和准确性，减少对模型训练和分析的干扰。数据清洗还包括对数据进行格式化处理，如统一日期时间格式、单位和标记规范等，以确保数据的一致性和便于处理。通过有效的数据预处理，可以为特征提取、分析和模型训练提供可靠的数据基础，确保后续工作的顺利进行和结果的准确性。

▶ 3.1.2　实体抽取阶段

实体抽取是从文本中识别和提取具有特定意义的词或短语的过程，而命名实体识别(Named Entity Recognition，NER)是实体抽取的核心部分。NER 专注于识别特定类别的实体，如人名、地名和机构名，是实现实体抽取的基础步骤。在实体识别后，实体扩展则进一步丰富这些已识别实体的信息，通过各种方法抽取出与这些实体相关或相似的其他实体，从而为后续的信息检索和分析提供更全面的支持。

1. 命名实体识别

命名实体识别是自然语言处理中的一个重要任务，旨在识别文本中特定类型的实体，如人名、地名、机构名等。这一过程通常可以通过基于规则的方法或基于机器学习的方法来实现。基于规则的方法依赖预先定义的规则或模式，如使用正则表达式来识别实体；而基于机器学习的方法则利用已标注的训练数据，训练模型自动识别实体。无论采用何种方法，命名实体识别的目标都是准确地识别出文本中的特定类型实体，为后续的信息提取和分析提供可靠的基础。

2. 实体扩展

实体扩展是在已识别的实体基础上，利用特定的规则和算法，进一步抽取出与这些实体相关或相似的其他实体。这一过程可以通过多种方法实现，包括基于知识图谱的方法、基于语义相似度的方法及基于上下文的方法等。在基于知识图谱的方法中，可以利用已有的知识图谱结构，通过实体之间的关系来扩展相关实体。在基于语义相似度的方法中，可以利用词嵌入等技术，计算实体之间的语义相似度，进而扩展相关实体。基于上下文的方法依赖文本中实体的上下文信息，通过分析实体周围的文本内容来扩展相关实体。这些方法的应用可以帮助丰富实体的语义信息，为后续的信息检索、问答系统等任务提供更全面和准确的支持。

▶ 3.1.3　关系抽取阶段

在自然语言处理(Natural Language Processing，NLP)的广阔领域中，知识抽取占据了一个至关重要的位置。它致力于从非结构化的文本数据中提取出有价值的结构化信息。而关系

抽取作为知识抽取的一个重要子任务,专注于识别和分类文本中实体之间的语义联系。

在关系抽取中,关系模式发现是一个探索性的过程,通过深入剖析文本内容,试图揭示隐藏在字里行间的实体间的潜在关系。这些关系可能表现为一种隐含的逻辑联系,也可能以明确的语义形式存在。通过细致的分析,可以从文本中提炼出多种多样的关系模式,如因果关系、对比关系、并列关系等。如果要从文本中有效地提取实体关系,则需要捕捉与这些关系密切相关的特征,如句法路径、词向量和上下文信息等。通过机器学习或深度学习模型将实体之间的关系分类为具体的关系类型,如购买关系、亲属关系或雇佣关系。通过这些步骤,关系抽取能够自动化地从文本中提取结构化的实体关系信息。

例如,在一个电商评论中,“我购买了××品牌的手机,质量非常好”这句话里,“我”和“××品牌的手机”之间就存在一个明确的购买关系。通过关系抽取技术,可以自动地将这种关系识别并提取出来。

▶ 3.1.4 事件抽取阶段

事件抽取(event extraction)是从文本中识别和提取特定类型事件及其相关信息的过程。该阶段的目标是自动识别文本中的事件,并提取与每个事件相关的关键要素,如事件发生的时间、地点、参与者、事件类型、原因和结果等。通过事件抽取,可以将非结构化的文本数据转换为结构化的事件信息,为后续的数据分析和应用提供支持。

在事件抽取过程中,首先定义需要识别的事件类型,如自然灾害、金融交易、犯罪事件等;然后通过分析文本内容,识别出与这些事件类型相关的句子或段落;接着从这些句子或段落中提取出关键要素,包括事件发生的时间(如具体的日期或时间段)、地点(如城市或国家名)、参与者(如个人、组织或团体)及其他相关信息(如事件的原因、过程和结果)。

▶ 3.1.5 后处理与评估阶段

知识抽取的后处理与评估阶段是确保知识抽取质量的关键环节,包括结果校验和效果评估。

1. 结果校验

结果校验是知识抽取过程中的关键环节,旨在对抽取出的知识进行校验和修正,以确保其准确性和完整性。这一过程涉及人工或自动化的检查方法,包括对抽取结果的逐项审核、与已有知识库的对比以及利用领域专家的反馈等。通过结果校验,可以发现并纠正抽取过程中可能存在的错误、遗漏或歧义,从而提高知识抽取的质量和可信度。结果校验是知识抽取流程中不可或缺的一环,对于保证最终抽取结果的准确性和可靠性具有至关重要的作用。

2. 效果评估

效果评估是对知识抽取过程中所得结果的质量进行定量和定性评价的过程。通过一系列评估指标,如准确率、召回率、F1 值等,对知识抽取的效果进行全面评估,以指导后续的优化和改进。准确率衡量了抽取出的知识中正确的比例,召回率衡量了抽取出的知识中真实的比例,而 F1 值综合考虑了准确率和召回率,是一个综合性评价指标。需要注意的是,这些阶段并不是完全独立的,它们之间可能存在交叉和重叠。在实际应用中,需要根据具体的数据和任务需求,对阶段进行灵活调整和优化。

3.2　实体抽取技术

▶ 3.2.1　实体抽取的概念

实体抽取又称为实体识别，是知识图谱构建的一个基础性工作，也是进一步实现关系抽取、事件抽取等更加复杂知识结构抽取的一个前提条件。实体抽取任务的主要目标是从文本中识别出代表实体的边界并进一步判别其类型。

实体抽取技术的基本原理是通过对文本进行语法和语义分析，识别并提取其中包含的特定实体，包括识别命名实体（如人名、地名、组织机构名等）及其他领域特定的实体（如日期、时间、货币等）。常用的实体抽取方法包括基于规则的方法、基于统计的方法和基于深度学习的方法。

▶ 3.2.2　规则式实体抽取

基于规则的实体抽取技术是实体抽取中最传统的方法之一。它通过预先定义的规则来匹配和提取实体，这些规则可以基于正则表达式、词典、语法模式等方式进行定义。这种技术具有较高的灵活性，能够根据特定需求进行定制和调整。同时，基于规则的实体抽取技术在许多领域都有广泛的应用，包括文本分类、新闻自动化处理、社交媒体分析等。基于规则的实体抽取技术的工作流程包括以下几个步骤。

（1）规则制定：根据特定领域的语法和语义规则，制定用于识别实体的抽取规则。

（2）文本解析：对待抽取的文本进行解析，识别其中的词性、句法结构等信息。

（3）规则匹配：预先制定的抽取规则被应用于经过解析的文本中，以识别和提取符合规则的实体片段。这一过程不仅是简单地将规则应用到文本上，还涉及模式识别、匹配验证、实体标注、冲突处理和输出格式化等具体操作。文本解析后生成的结构化信息（如词性标注和句法结构）为匹配提供基础。规则匹配引擎逐词扫描文本，应用正则表达式、词典匹配或语法模式来识别特定的实体片段。例如，正则表达式可用于匹配日期格式，词典可用于匹配地名或人名，而语法模式则用于匹配特定的句法结构。在匹配到可能的实体片段后，还需进行验证，确保其符合预定义规则，如确认日期格式的有效性。一旦匹配成功，文本片段会被标注为特定类型的实体，并记录其位置信息和相关属性。

（4）实体输出：在可能出现的规则冲突中，通过优先级或上下文分析策略解决冲突，最终将匹配的实体片段提取并输出为结构化信息（如 JSON 或 XML 格式）。

正则应用举例：电话号码信息的抽取。

```
1.    regex = "1[34578]\d{9}"
2.    text = "电话号码信息是 1xxxxxxxxxx"
3.    phone_number = re.findall(regex, text)
4.    print(phone_n umber)
5.    #抽取结果:['1xxxxxxxxxx']
```

▶ 3.2.3　基于机器学习的实体抽取

基于机器学习的实体关系抽取方法通常首先将实体关系抽取问题转换为二分类或多分类问题。然后，利用机器学习算法对已有的大量样本进行特征编码、建模和训练，形成能够自动

预测未知文本实体关系模式的机器学习工具模型。不同机器学习方法在内在上的区别主要体现在对文本进行特征编码、建模和训练方法的差异,以及对文本中词汇或句子间依赖关系的理解与长短捕获的程度和方式上的不同;而在外在上的区别主要表现为对样本文本进行序列、语义标注的格式不同,最终导致抽取的准确率和召回率也有所不同。

机器学习模型应用于实体抽取最重要的思想是获取实体的特征信息,本节以经典的隐马尔可夫模型 HMM、条件随机场 CRF 以及支持向量机(Support Vector Machine,SVM)进行介绍。其中,HMM 和 CRF 有着相似的思想,它们都和随机场有关。随机场是由若干位置组成的整体,当给每个位置按照某种分布随机赋予一个值后,其全体称为随机场。马尔可夫随机场是随机场的特例,它假设随机场中某一个位置的赋值仅与和它相邻的位置的赋值有关,和与其不相邻的位置的赋值无关。CRF 是马尔可夫随机场的特例,它假设马尔可夫随机场中只有 X 和 Y 两种变量,X 一般是给定的,而 Y 一般是在给定 X 条件下的输出。


视频讲解

1. HMM 模型

1) 基本概念

隐马尔可夫模型(Hidden Markov Model,HMM)是一种常见的统计模型,用于描述时间序列数据。由于命名实体识别(NER)问题可以看作一种序列标注问题,因此 HMM 也被广泛应用于 NER 任务中。HMM 的核心假设包括齐次马尔可夫假设和观测独立性假设。齐次马尔可夫假设意味着模型在任意时刻的状态转移概率只依赖前一时刻的状态,而观测独立性假设则指观测序列只依赖当前时刻的马尔可夫链状态,与其他观测或状态无关。基于这两个假设,HMM 可以有效地处理序列数据的预测和分类问题。在解决 NER 问题时,HMM 将每个词的标注(即实体类型)视为隐藏状态,而每个词的观测序列则是根据隐藏状态生成的。通过训练,HMM 可以学习到词与标注之间的概率分布,从而在给定观测序列的情况下预测最可能的标注序列。在使用 HMM 进行 NER 时,通常采用 BIOES 标注方法对每个词进行标注。BIOES 代表 Begin,Inside,Outside,End,Single,分别表示一个实体开始、实体内部、不属于任何实体、实体结束和单个单词实体。通过标注每个词的 BIOES 标签,可以识别出文本中的不同实体。

在利用 HMM 进行 NER 实现的过程中,可以采用传统的基于规则的方法或基于深度学习的方法。基于规则的方法通过手动定义规则来识别实体,而基于深度学习的方法则利用神经网络自动学习词与标注之间的关系。目前,基于深度学习的方法在 NER 任务中取得了更好的效果,但 HMM 仍然是一种简单有效的解决 NER 问题的方法。在实际应用中,HMM 可以与其他机器学习算法结合使用,以提高命名实体识别的准确率。

2) 基本原理

HMM 是统计模型,用来描述一个隐含未知参数的马尔可夫过程,确定隐含的参数,然后利用参数来做进一步的分析。一般 HMM 包含三个参数:λ、状态序列 I、观测序列 O。其中 λ 表示为

$$\lambda = (\boldsymbol{A}, \boldsymbol{B}, \Pi) \tag{3.1}$$

$$\boldsymbol{A} = [a_{ij}] \tag{3.2}$$

$$\boldsymbol{B} = [b_j(k)] \tag{3.3}$$

其中,$\boldsymbol{A}$ 为转移概率矩阵;a_{ij} 表示从状态 i 转移到状态 j 的概率;$\boldsymbol{B}$ 为发射概率矩阵;$b_j(k)$ 表示在状态 j 的条件下生成观测 v_k 的概率;Π 为初始状态概率。HMM 针对 λ、状态序列 I、观测序列 O 这三个参数可以应用于三个基本问题:概率计算问题、学习问题以及解码问题。

如图 3.2 所示，HMM 的输入 x 称为观测值，对应 NER 任务就是输入的文本；输出 y 称为状态值，对应 NER 任务就是标签值。通过当前输入 x_i 和上一时刻 y_{i-1} 状态预测当前状态值的概率，以此类推，得到概率最高的一条状态序列。

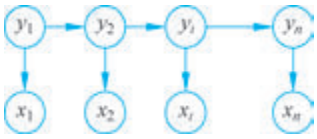


图 3.2　HMM

3）代码示例

以下是使用 HMM 对观察序列进行解码，并输出最可能的状态序列。在实际应用中，需要根据具体的数据和任务进行模型的训练和调优。

```
import numpy as np
from hmmlearn import hmm
from sklearn.preprocessing import LabelEncoder

#示例数据：训练数据
sentences = [
    (["Steve", "Jobs", "is", "the", "CEO", "of", "Apple"], ["PERSON", "PERSON", "O", "O",
    "O", "O", "ORG"]),
    (["Elon", "Musk", "founded", "SpaceX"], ["PERSON", "PERSON", "O", "ORG"])
]
#词汇表和标签集合
words = set(word for sentence, _ in sentences for word in sentence)
tags = set(tag for _, tag_seq in sentences for tag in tag_seq)

#创建词汇表和标签的编码器
word_encoder = LabelEncoder()
tag_encoder = LabelEncoder()
word_encoder.fit(list(words))
tag_encoder.fit(list(tags))
#将词和标签转换为索引
X_train = [word_encoder.transform(sentence) for sentence, _ in sentences]
y_train = [tag_encoder.transform(tag_seq) for _, tag_seq in sentences]
#将训练数据转换为适合 HMM 输入的格式
X_train_flattened = np.concatenate([x.reshape(-1, 1) for x in X_train])
y_train_flattened = np.concatenate([y.reshape(-1) for y in y_train])

#HMM 参数
n_states = len(tag_encoder.classes_)
n_observations = len(word_encoder.classes_)
#创建 HMM
model = hmm.MultinomialHMM(n_components=n_states, n_iter=100)
#训练模型
model.fit(X_train_flattened)
#测试数据
test_sentence = ["Steve", "founded", "Apple"]
#转换测试数据
X_test = word_encoder.transform(test_sentence).reshape(-1, 1)

#预测
logprob, predicted_tags = model.decode(X_test, algorithm="viterbi")
#将预测的标签转换回实际的标签
predicted_tags = tag_encoder.inverse_transform(predicted_tags)
#显示结果
print("Sentence:", test_sentence)
print("Predicted Tags:", predicted_tags)
```

该示例演示了如何使用 HMM 对观察序列进行解码,并输出最可能的状态序列。模型的输入包括训练数据和测试数据,其中训练数据由若干句子及其对应的标签组成,每个句子是一个词序列,而标签是与每个词对应的实体类别(例如,PERSON、ORG、O 等)。

```
sentences = [(["Steve", "Jobs", "is", "the", "CEO", "of", "Apple"], ["PERSON","PERSON", "O",
"O", "O", "O", "ORG"]), ([" Elon", " Musk", " founded", " SpaceX"], [" PERSON", " PERSON",
"O", "ORG"])]
```

测试数据是一个或多个句子,下面希望对这些句子中的每个词进行实体类别的预测。测试数据的输入格式与训练数据类似,但不包含标签。

```
test_sentence = ["Steve", "founded", "Apple"]
```

```
Sentence: ['Steve', 'founded', 'Apple']
Predicted Tags: ['PERSON' 'O' 'ORG']
```

图 3.3 HMM 示例代码的输出结果

模型的输出如图 3.3 所示,预测结果显示:Steve 被标记为 PERSON(人名),founded 被标记为 O(非实体)。

Apple 被标记为 ORG(组织名),这个结果表明模型识别出了 Steve 作为人名(PERSON)和 Apple 作为组织(ORG),而 founded 被标记为非实体(O),这与实际标签基本一致。

在实际应用中,需要根据具体的数据和任务进行模型的训练和调优。

视频讲解

2. CRF 模型

1) 基本概述

条件随机场(Conditional Random Field,CRF)是一种用于序列标注和分段的机器学习模型。在命名实体识别任务中,CRF 被广泛用于序列标注问题,因为它能够有效地处理标记之间的依赖关系,同时避免了对输入数据的长度限制。

2) 模型原理

CRF 与 HMM 类似,但 HMM 是在输出独立性假设的条件下成立的,当前输出仅依赖当前标签的假设,而 CRF 是将所有特征进行全局归一化可以求得全局的最优值,也就是在输出时依赖整个输入序列。线性链条件随机场 CRF 的图结构如图 3.4 所示。

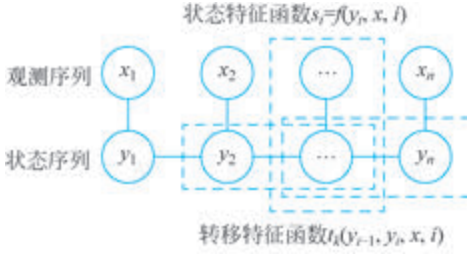


图 3.4 线性链条件随机场 CRF 的图结构

假设 X 与 Y 分别为输入序列与输出序列,则条件概率 $P(Y|X)$ 在 X 为 x_i 的条件下,Y 取值为 y_i 的概率分布可以表示为

$$p(y \mid x,\lambda)=\frac{1}{Z(x)}\exp\left[\sum_{j_i}\lambda_j t_j(y_{i-1},y_i,x,i)+\sum_{k_i}u_k s_k(y_i,x,i)\right] \tag{3.4}$$

其中,$Z(x)$为概率归一化系数:

$$Z(x)=\sum_{y}\exp\left[\sum_{j_i}\lambda_j t_j(y_{i-1},y_i,x,i)+\sum_{k_i}u_k s_k(y_i,x,i)\right] \tag{3.5}$$

其中,i 表示序列位置索引,t_j 是转移特征函数,s_k 是状态特征函数,λ,u 分别是对应特征函数的权值。k_i 和 j_i 分别表示特征函数的索引。

3）训练方法

CRF（条件随机场）的训练过程旨在找到一组参数，使得在给定输入序列的条件下模型能够生成最准确的输出标签序列。具体来说，CRF 模型的训练依赖最大化对数似然函数，即在所有训练样本上计算标签序列的条件概率，并通过优化算法调整参数以提高模型预测的准确性。以下是 CRF 的主要训练步骤。

（1）定义对数似然函数：在 CRF 中，对于一个输入序列 X 和标签序列 Y，目标是最大化条件概率 $P(Y \mid X)$。CRF 使用对数似然函数来衡量预测序列与真实序列之间的匹配程度。似然函数的形式如下：

$$\log P(Y \mid X) = \sum_i \sum_j \lambda_j f_j(Y, X, i) - \log Z(X) \tag{3.6}$$

其中，$f_j(Y, X, i)$是特征函数，λ_j 是特征函数对应的权重，$Z(X)$是归一化因子，确保所有可能标签序列的概率总和为 1。

（2）计算归一化因子 $Z(X)$：归一化因子 $Z(X)$是所有可能的标签序列的条件概率之和。计算 $Z(X)$通常会用到动态规划，因为可能标签序列的数量随着序列长度呈指数增长。

（3）优化对数似然函数：一旦定义了对数似然函数，就可以通过梯度下降类的优化算法来求解参数。以下为通常使用的优化方法。

梯度下降：直接计算目标函数的梯度并更新参数。

拟牛顿方法（如 L-BFGS）：这是训练 CRF 的常用算法，它通过近似计算 Hessian 矩阵来加速收敛。

随机梯度下降（SGD）：在大型数据集上，SGD 可以帮助 CRF 更快地训练。

（4）正则化：为防止过拟合，可以在损失函数中加入正则化项（如 L2 正则化），限制模型复杂度。

通过上述步骤，CRF 模型在训练集上可以找到一组最佳参数，从而在未知数据上实现对序列的准确标注。

4）应用实例

下面通过一个简单的例子来演示如何使用条件随机场进行地址识别。假设有一段文本："张三家住在上海市浦东新区。"需要识别出"张三"和"上海市浦东新区"这两个实体。首先，需要对文本进行预处理，提取出与实体相关的特征，例如词性、位置信息等。然后，将这些特征作为输入，使用 CRF 模型进行训练。在训练过程中，模型会学习如何根据上下文信息预测每个位置上的地址类型。一旦模型训练完成，可以使用它来对新的文本进行地址识别。对于给定的句子，首先提取出相关的特征，然后使用训练好的 CRF 模型进行推断，得出每个位置上的地址类型标签序列。最后，将连续的标签序列转换为地址边界和类型。

在实际应用中，为了提高命名实体识别的准确率，还可以结合其他技术和方法，例如深度学习模型、规则等。通过集成多种方法的优势，可以构建更加健壮和高效的地址识别系统。

3. 支持向量机 SVM

1）基本概念

支持向量机（SVM）是一种常用的监督学习模型，广泛应用于分类和回归问题。SVM 的核心思想是找到一个能够最大化类间间隔的超平面，从而在特征空间中将不同类别的数据点分开。在 NER 任务中，SVM 可用于识别和分类文本中的实体。SVM 模型通过将输入数据

映射到高维空间,并在高维空间中寻找最优的分离超平面,实现对数据的分类。在处理非线性数据时,SVM 利用核函数(Kernel)将数据映射到更高维的空间,从而实现非线性分类。

在 NER 任务中,SVM 将每个词作为一个输入特征,并对其进行分类,判断其是否属于某个命名实体。通过训练,SVM 可以学习到词与其标签之间的关系,从而在给定新文本时准确识别其中的命名实体。在使用 SVM 进行 NER 时,通常采用 BIO 或 BIOES 标注方法对每个词进行标注。通过标注每个词的标签,可以识别出文本中的不同实体。

SVM 在 NER 中的应用优势在于其良好的泛化能力和处理高维数据的能力,同时也能通过核函数有效处理非线性分类问题。

2) 模型原理

SVM 的基本原理是通过寻找一个最优超平面,将不同类别的数据点分开,并最大化类间的间隔。对于线性可分的数据,SVM 通过如下优化问题找到最优超平面:

$$\min_{w,b} \frac{1}{2} \| \boldsymbol{w} \|^2 \tag{3.7}$$

$$\text{subject to } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + \boldsymbol{b}) \geqslant 1 \forall i \tag{3.8}$$

其中,$\boldsymbol{w}$ 是超平面的法向量;$\boldsymbol{b}$ 是偏置项;y_i 是数据点 $\boldsymbol{x}_i$ 的标签(+1 或−1)。

对于非线性可分的数据,SVM 引入核函数 $K(x,x')$将数据映射到高维空间,使得在新空间中数据可以线性分开。常用的核函数如下。

线性核:$K(\boldsymbol{x},\boldsymbol{x}')=\boldsymbol{x},\boldsymbol{x}'$

多项式核:$K(\boldsymbol{x},\boldsymbol{x}')=(\boldsymbol{x}\cdot\boldsymbol{x}'+1)^d$

高斯核:$K(\boldsymbol{x},\boldsymbol{x}')=\exp(-\gamma \| \boldsymbol{x}-\boldsymbol{x}' \|^2)$

Sigmoid 核:$K(\boldsymbol{x},\boldsymbol{x}')=\tanh(\alpha\boldsymbol{x}\cdot\boldsymbol{x}'+c)$

在处理 NER 任务时,SVM 的输入是词特征,输出是相应的标签。通过当前词的特征和模型参数预测当前词的标签,以此类推,得到最优的标签序列。

3) 训练方法

SVM 的训练过程如下。

(1) 选择核函数:SVM 模型的性能在很大程度上依赖核函数的选择。常用的核函数如下。

① 线性核:适用于线性可分的情况。

② 径向基函数核(RBF Kernel):适用于非线性情况,能够映射到高维空间。

③ 多项式核:适用于数据的多项式关系。

(2) 设置超参数:调整 SVM 模型的超参数,例如 c(惩罚参数)和 γ(RBF 核函数的参数)。c 控制分类器的复杂度,γ 影响决策边界的曲线度。可以使用网格搜索(grid search)和交叉验证(cross-validation)来优化这些超参数。

(3) 训练模型:使用训练数据中的特征向量和对应的标签来训练 SVM 模型。训练过程中,SVM 模型通过优化目标函数,寻找最佳的决策边界,以最大化类别间的间隔。

4) 应用实例

下面通过一个简单的例子来演示如何使用支持向量机进行实体识别。假设有一段文本:“张三家住在上海市浦东新区。”现需要识别出“张三”和“上海市浦东新区”这两个实体。首先,需要对文本进行预处理,提取出与实体相关的特征,例如词性、位置信息等。然后,将这些特征

作为输入，使用 SVM 模型进行训练。在训练过程中，模型会学习如何根据上下文信息预测每个位置上的实体类型。一旦模型训练完成，即可以使用其来对新的文本进行实体识别。对于给定的句子，首先提取出相关的特征，然后使用训练好的 SVM 模型进行推断，得出每个位置上的实体类型标签序列。最后，将连续的标签序列转换为实体边界和类型。

5）代码示例

以下是一个简单的 Python 代码示例，演示如何使用 SVM 进行 NER 任务，采用 BIO 标注体系，识别文本中的人名和地名。

```
import numpy as np
from sklearn import svm
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.pipeline import make_pipeline
from sklearn.feature_extraction.text import TfidfTransformer

#示例数据
sentences = [
    "张三家住在上海市浦东新区。",
    "李四在北京工作。",
    "王五搬到了深圳市南山区。",
    "赵六在广州上学。",
    "孙七在南京旅游。",
    "周八在武汉出差。",
    "郑九去了杭州。",
    "钱十在成都生活。",
    "吴十一在重庆。",
    "郑十二在天津读书。"
]
labels = [
    ["B-PER", "O", "O", "B-LOC", "I-LOC", "I-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "O", "O", "B-LOC", "I-LOC", "I-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"],
    ["B-PER", "O", "B-LOC", "O"]
]

#提取特征
def extract_features(sentence):
    return [char for char in sentence]
#准备数据
X = [extract_features(sentence) for sentence in sentences]
#保证每个字符有对应的标签
y = []
for sentence, label in zip(sentences, labels):
    char_labels = []
    label_idx = 0
```

```
        for char in sentence:
            if char in ["。"]:  #跳过标点符号
                char_labels.append("O")
            else:
                if label_idx < len(label):
                    char_labels.append(label[label_idx])
                else:
                    char_labels.append("O")
                label_idx += 1
        y.append(char_labels)
    #展平数据
    flat_X = [item for sublist in X for item in sublist]
    flat_y = [item for sublist in y for item in sublist]

    #检查特征和标签的长度是否一致
    assert len(flat_X) == len(flat_y), "The number of features and labels must be equal"
    #标签编码
    le = LabelEncoder()
    y_encoded = le.fit_transform(flat_y)
    #特征向量化
    vectorizer = CountVectorizer(tokenizer = lambda txt: txt,
    lowercase = False, token_pattern = None)
    X_vectorized = vectorizer.fit_transform(flat_X)

    #划分训练集和测试集
    X_train, X_test, y_train, y_test = train_test_split(X_vectorized, y_encoded, test_size =
    0.3, random_state = 42)
    #创建 SVM 分类器
    clf = svm.SVC(kernel = 'linear', class_weight = 'balanced')

    #训练模型
    clf.fit(X_train, y_train)
    #预测
    y_pred = clf.predict(X_test)
    #评估模型
    print(classification_report(y_test, y_pred, target_names = le.classes_, zero_division = 0))

    #测试新数据
    new_sentence = "王五搬到了深圳市南山区。"
    new_features = extract_features(new_sentence)
    new_vectorized = vectorizer.transform(new_features)
    new_pred = clf.predict(new_vectorized)
    new_labels = le.inverse_transform(new_pred)
    print("Predicted labels for new sentence:", new_labels)
```

该模型使用支持向量机(SVM)分类器,根据 BIO 标注体系对句子中的字符进行命名实体识别(NER),识别出人名和地名。每个字符作为独立特征,通过模型训练后,预测测试集和新句子的字符标签,并生成分类报告来评估模型性能。

输入的句子为["张三家住在上海市浦东新区。","李四在北京工作。","王五搬到了深圳市南山区。","赵六在广州上学。","孙七在南京旅游。","周八在武汉出差。","郑九去了杭州。","钱十在成都生活。","吴十一在重庆。","郑十二在天津读书。"]。

输入的标签为[["B-PER","O","O","B-LOC","I-LOC","I-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","O","O","B-LOC","I-LOC","I-LOC","O"],["B-PER",

"O","B-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","B-LOC","O"],["B-PER","O","B-LOC","O"]]。

模型的输出包含以下几点。

准确率(Accuracy)：模型的总体准确率为0.78，这表示模型在所有测试样本中有78% 的预测是正确的。

B-LOC 标签：精确率为0.50，召回率为1.00，F1 分数为0.67。表示所有标记为 B-LOC 的预测中有50%是正确的，所有真实为 B-LOC 的标签中有100%被正确识别。

B-PER 标签：精确率为1.00，召回率为0.25，F1 分数为0.40。表示所有标记为 B-PER 的预测中有100% 是正确的，但只有25%的真实 B-PER 标签被正确识别。

I-LOC 标签：精确率为0.00，召回率为0.00，F1 分数为0.00。这表明模型没有成功识别任何 I-LOC 标签。

O 标签：精确率为0.79，召回率为1.00，F1 分数为0.88。表示模型在预测 O 标签时表现良好，几乎所有真实的 O 标签都被正确识别。

宏平均(Macro Avg)：宏平均精确率为0.57，宏平均召回率为0.56，宏平均 F1 分数为0.49。宏平均是对每个类别的指标进行简单平均，忽略类别的不平衡性。

加权平均(Weighted Avg)：加权平均精确率为0.72，加权平均召回率为0.78，加权平均 F1 分数为0.71。加权平均考虑了每个类别在测试集中出现的频率，更能反映模型整体性能。

如图3.5所示，新句子的预测标签为['B-PER', 'O', 'O', 'O', 'B-LOC', 'O', 'I-LOC', 'O', 'O', 'O', 'O', 'O']。新句子“王五搬到了深圳市南山区。”的预测结果显示，模型正确地识别了“王五”为人名(B-PER)和“深圳市南山区”的一部分(B-LOC 和 I-LOC)。

	precision	recall	f1-score	support
B-LOC	0.50	1.00	0.67	1
B-PER	1.00	0.25	0.40	4
I-LOC	0.00	0.00	0.00	3
O	0.79	1.00	0.88	19
accuracy			0.78	27
macro avg	0.57	0.56	0.49	27
weighted avg	0.72	0.78	0.71	27

```
Predicted labels for new sentence: ['B-PER' 'O' 'O' 'O' 'B-LOC' 'O' 'I-LOC' 'O' 'O' 'O' 'O' 'O']
```

图 3.5　支持向量机示例代码运行结果

支持向量机是一种强大的分类工具，特别适用于复杂、高维数据的分类问题。通过核函数扩展，SVM 可以有效处理非线性分类问题。在命名实体识别任务中，SVM 可以通过学习词与其标签之间的关系，实现对文本中命名实体的准确识别。尽管在处理大规模数据时可能存在性能瓶颈，但其理论基础和实际应用效果仍使其在机器学习领域占有重要地位。

▶ 3.2.4　基于深度学习的实体抽取

随着深度学习方法在自然语言处理领域的广泛应用，深度神经网络也被成功应用于命名实体识别问题，并取得了很好的效果。与传统的统计模型相比，基于深度学习的方法直接以文本中词的向量为输入，通过神经网络实现端到端的命名实体识别，不再依赖人工定义的特征。一般不同的神经网络结构在命名实体识别过程中扮演编码器的角色，它们基于初始输入以及

词的上下文信息,得到每个词的新向量表示,最后再通过 CRF 模型输出对每个词的标注结果。本节将以卷积神经网络(CNN)、循环神经网络(RNN)、双向编码器表示转换器(BERT)、双向长短期记忆网络-条件随机场(BiLSTM-CRF)这几类经典模型为例,介绍如何使用深度学习进行关系抽取,并提供一些实际操作建议。


视频讲解

1. CNN 模型

1) CNN 基本概念

卷积神经网络(Convolutional Neural Network, CNN)是一种专门用于处理网格结构数据(如图像和视频)的深度学习模型。CNN 通过局部感受野、权重共享和下采样等机制,能够有效地提取和处理输入数据中的特征。与传统的全连接神经网络相比,CNN 在处理高维数据(如图像)时具有显著的优势。

(1) 卷积层(Convolutional Layer):卷积层是 CNN 的核心组件之一,它通过滤波器(卷积核)对输入数据进行滤波处理。这些滤波器可以学习到不同特征,例如边缘、纹理等。

(2) 池化层(Pooling Layer):池化层用于减少特征图的尺寸,同时保留重要的特征信息。常用的池化操作包括最大池化和平均池化。

(3) 激活函数(Activation Function):在卷积层之后,通常会应用一个非线性激活函数,如 ReLU(Rectified Linear Unit),用于引入非线性特征。

(4) 全连接层(Fully Connected Layer):在经过一系列卷积和池化操作后,通常会将特征图展平并连接到全连接层,用于进行最终的分类或回归操作。

2) 公式解析

(1) 卷积层:卷积层的作用是通过卷积操作提取输入数据中的局部特征,应用一系列可学习的滤波器(或称为卷积核)在输入数据上滑动,并进行点积运算,生成特征图。其中卷积核是一个小矩阵,通常大小为 3×3、5×5 等,用于在输入图像上滑动并提取特征。假设输入图像尺寸为 $X\times Y$,卷积核为 K,步长为 S,填充为 P,则输出特征图的大小 $X'\times Y'$ 计算公式为

$$X'=\frac{X-K+2\cdot P}{S}+1 \tag{3.9}$$

$$Y'=\frac{Y-K+2\cdot P}{S}+1 \tag{3.10}$$

(2) 激活函数:激活函数通常用于引入非线性。常用的激活函数包括 ReLU、Sigmoid 和 tanh。ReLU 的公式为

$$\mathrm{ReLU}(z)=\max(0,z) \tag{3.11}$$

其中,z 是卷积层的输出。

(3) 池化层:池化层的作用是对卷积层提取的特征进行降维,减小特征图的尺寸,同时保留重要特征。常用的池化方法包括最大池化和平均池化。最大池化的公式为

$$y(i,j)=\max_{(m,n)\in \mathrm{pool}} x(i+m,j+n) \tag{3.12}$$

其中,$x(i,j)$是输入特征图;$y(i,j)$是输出池化特征图;pool 是池化窗口的范围。

(4) 全连接层:全连接层用于将池化层的输出特征图展平(flatten),并通过一系列全连接层进行分类或回归。全连接层的输出计算公式为

$$\boldsymbol{y}=\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b} \tag{3.13}$$

其中,$\boldsymbol{x}$ 是展平后的输入特征向量;$\boldsymbol{W}$ 是权重矩阵;$\boldsymbol{b}$ 是偏置向量;$\boldsymbol{y}$ 是输出向量。

3）训练过程

在模型训练阶段，必须利用标记好的数据来训练 CNN 模型。具体做法是，将标记好的关系数据分为训练集、验证集和测试集。然后，利用训练集来训练 CNN 模型，利用验证集来调整模型的超参数（如学习率、卷积核大小等），最后使用测试集来评估模型的性能。

在训练过程中，需要定义一个损失函数，用来衡量模型预测结果与真实结果之间的差异。常用的损失函数有交叉熵损失函数、均方误差损失函数等。通过最小化损失函数，可以不断更新模型的参数，从而优化模型的性能。

交叉熵损失函数和均方误差损失函数公式分别为 $L(y,\hat{y})=-\sum_{i=1}^{n} y_i \log \hat{y}_i$，$L(y,\hat{y})=\sum_{i=1}^{n}(y_i-\hat{y}_i)^2/n$，其中 y 是真实的标签值（正类值为 1，负类值为 0），$\hat{y}$ 是预测的概率值，n 表示样本数量，损失函数表征真实样本标签和预测概率之间的差值。

4）代码示例

下面是一个简单的基于卷积神经网络的实体抽取的示例代码，用于从文本中提取人名。

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# 定义 CNN 模型
def create_cnn_model(input_length, vocab_size, embedding_dim, num_classes):
    model = models.Sequential([
        layers.Embedding(input_dim = vocab_size, output_dim = embedding_dim, input_length = input_length),
        layers.Conv1D(128, 5, activation = 'relu'),
        layers.GlobalMaxPooling1D(),
        layers.Dense(num_classes, activation = 'softmax')
    ])
    return model

# 示例数据（文本序列）
texts = ['Zhang San lives in Pudong New District, Shanghai.', 'Li Si works in Beijing.']
# 构建词汇表和将文本转换为序列
tokenizer = tf.keras.preprocessing.text.Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
# 计算输入序列长度和词汇表大小
input_length = len(sequences[0])
vocab_size = len(tokenizer.word_index) + 1
# 填充序列长度
sequences_padded = tf.keras.preprocessing.sequence.pad_sequences(sequences, padding = 'post')
# 样本标签（示例中只包含人名）
labels = np.array([1, 0])  # 将标签转换为 Numpy 数组

# 创建 CNN 模型实例
model = create_cnn_model(input_length, vocab_size, embedding_dim = 50, num_classes = 2)
# 编译模型
model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',metrics = ['accuracy'])
# 训练模型
model.fit(sequences_padded, labels, epochs = 10, batch_size = 1)
```

```
35.
36.  # 使用模型进行预测
37.  predictions = model.predict(sequences_padded)
38.  print(predictions)
```

这个示例代码创建了一个简单的卷积神经网络模型,用于识别文本中的人名。输入数据是两个示例句子,通过使用 Conv1D 卷积层 GlobalMaxPooling1D 池化层,模型学习了文本数据的特征。训练过程中,模型的损失逐步降低,准确率逐渐提高,如图 3.6 所示。

```
Epoch 1/10
2/2 [==============================] - 0s 0s/step - loss: 0.6905 - accuracy: 0.5000
Epoch 2/10
2/2 [==============================] - 0s 0s/step - loss: 0.6121 - accuracy: 1.0000
Epoch 3/10
2/2 [==============================] - 0s 0s/step - loss: 0.5590 - accuracy: 1.0000
Epoch 4/10
2/2 [==============================] - 0s 0s/step - loss: 0.5075 - accuracy: 1.0000
Epoch 5/10
2/2 [==============================] - 0s 0s/step - loss: 0.4635 - accuracy: 1.0000
Epoch 6/10
2/2 [==============================] - 0s 0s/step - loss: 0.4212 - accuracy: 1.0000
Epoch 7/10
2/2 [==============================] - 0s 0s/step - loss: 0.3808 - accuracy: 1.0000
Epoch 8/10
2/2 [==============================] - 0s 0s/step - loss: 0.3434 - accuracy: 1.0000
Epoch 9/10
2/2 [==============================] - 0s 0s/step - loss: 0.3048 - accuracy: 1.0000
Epoch 10/10
2/2 [==============================] - 0s 0s/step - loss: 0.2710 - accuracy: 1.0000
[[0.18840235 0.8115976 ]
 [0.7549606  0.2450394 ]]
```

图 3.6　卷积神经网络示例代码运行结果

模型的输入包括文本数据和标签数据,其中文本数据为 texts = ['Zhang San lives in Pudong New District, Shanghai.', 'Li Si works in Beijing.']。标签数据为 labels = [1, 0](这里 1 表示“人名”,0 表示“非人名”)。

模型的输出包括训练的 10 个周期(epoch),每个周期结束时显示了损失(loss)和准确率(accuracy)。最终,模型在训练集上的损失为 0.2710,准确率为 1.0000。

[[0.18840235 0.8115976], [0.7549606 0.2450394]]是模型对每个文本序列的预测结果,表示每个类别的概率。对于第一条文本 "Zhang San lives in Pudong New District, Shanghai.",模型预测的类别概率为 [0.188, 0.812],表明它更倾向于预测为“人名”类别。对于第二条文本 "Li Si works in Beijing.",模型预测的类别概率为 [0.755, 0.245],表明它更倾向于预测为“非人名”类别。


视频讲解

2. RNN 模型

1) RNN 基本概念

循环神经网络(Recurrent Neural Network, RNN)是一种专门用于处理序列数据的神经网络架构,能够通过其循环连接结构在不同的时间步之间传递信息,从而捕捉和记住序列中的上下文信息。RNN 在每个时间步上不仅处理当前输入,还参考之前的隐藏状态,使其擅长处理和预测时间依赖关系,这在自然语言处理、语音识别和时间序列预测等任务中尤为有效。

(1) 循环层(Recurrent Layer):RNN 的核心组件之一是循环层,它通过循环连接来处理序列数据,使模型能够记住输入序列中的上下文信息。循环层能够处理时间序列数据或序列数据,例如文本、语音信号等。

(2) 隐藏状态(Hidden State)：隐藏状态是 RNN 在每个时间步更新和传递的信息，反映了到当前时间步为止序列的记忆。隐藏状态在每个时间步通过前一个隐藏状态和当前输入进行更新。

(3) 激活函数(Activation Function)：在每个时间步中，RNN 使用激活函数(如 tanh 或 ReLU)来引入非线性特征，使模型能够更好地捕捉复杂的模式和关系。

(4) 输出层(Output Layer)：在经过所有时间步的处理后，RNN 通常会将隐藏状态或其变换结果传递到输出层，用于最终的分类、回归或其他预测任务。

2) 公式解析

(1) 循环层：循环层的作用是通过循环连接处理序列数据。循环操作的公式如下：

$$\boldsymbol{h}_t = \sigma(\boldsymbol{W}_h \cdot \boldsymbol{h}_{t-1} + \boldsymbol{W}_x \cdot \boldsymbol{x}_t + \boldsymbol{b}_h) \tag{3.14}$$

其中，$\boldsymbol{h}_t$ 表示当前时间步的隐藏状态。$\boldsymbol{h}_{t-1}$ 表示前一个时间步的隐藏状态；$\boldsymbol{x}_t$ 表示当前时间步的输入数据；$\boldsymbol{W}_h$ 和 $\boldsymbol{W}_x$ 是权重矩阵，$\boldsymbol{b}_h$ 是偏置向量；σ 是激活函数(如 tanh 或 ReLU)。

(2) 激活函数：激活函数用于引入非线性。在 RNN 中，常用的激活函数有 Sigmoid 和 tanh。这些函数能够处理序列数据的特性，通过其平滑的曲线帮助网络捕捉时间序列中的依赖关系。tanh 的公式为

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3.15}$$

其中，z 是循环层的线性变换输出。

相比之下，在 CNN 中，ReLU 激活函数是最常用的选择。ReLU 函数的计算简单且能够有效缓解梯度消失问题，使得 CNN 能够更好地处理图像数据。

(3) 输出层：输出层的作用是将隐藏状态映射到输出标签空间。输出层的计算公式为

$$\boldsymbol{y}_t = \mathrm{softmax}(\boldsymbol{W}_y \cdot \boldsymbol{h}_t + \boldsymbol{b}_y) \tag{3.16}$$

其中，$\boldsymbol{y}_t$ 是当前时间步的输出；$\boldsymbol{W}_y$ 是输出层的权重矩阵；$\boldsymbol{b}_y$ 是偏置向量。

3) 训练过程

在模型训练阶段，需要利用标记好的数据来训练 RNN 模型。具体做法是，将标记好的关系数据分为训练集、验证集和测试集。然后，利用训练集来训练 RNN 模型，利用验证集来调整模型的超参数(如学习率、隐藏状态维度等)，最后使用测试集评估模型的性能。

在训练过程中，需要定义一个损失函数，用来衡量模型预测结果与真实结果之间的差异。常用的损失函数有交叉熵损失函数、均方误差损失函数等。通过最小化损失函数，可以不断更新模型的参数，从而优化模型的性能。

4) 代码示例

下面是一个简单的基于 RNN 的实体抽取的示例代码，用于从文本中提取人名。

```
import torch
import torch.nn as nn
import torch.optim as optim

#定义 RNN 模型
class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
```

```
            self.hidden_size = hidden_size
            self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
            self.fc = nn.Linear(hidden_size, output_size)
        def forward(self, x):
            h0 = torch.zeros(1, x.size(0), self.hidden_size)
            out, _ = self.rnn(x, h0)
            out = self.fc(out[:, -1, :])
            return out

#参数设置
input_size = 50                                          #输入特征维度
hidden_size = 128                                        #隐藏层维度
output_size = 10                                         #输出类别数
learning_rate = 0.001
num_epochs = 10
#创建模型、损失函数和优化器
model = RNNModel(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

#示例数据(文本序列)
texts = ['Zhang San lives in Pudong New District, Shanghai.', 'Li Si works in Beijing.']
#创建词汇表
vocab = set(word for text in texts for word in text.split())
vocab = {word: idx for idx, word in enumerate(vocab)}
vocab_size = len(vocab)
#随机初始化词向量
embeddings = torch.randn(vocab_size, input_size)
#将文本转换为索引序列
def text_to_tensor(text, vocab, embeddings, max_len):
    tokens = text.split()
    tensor = torch.zeros(max_len, embeddings.size(1))
    for i, token in enumerate(tokens):
        if i >= max_len:
            break
        tensor[i] = embeddings[vocab[token]]
    return tensor
#定义输入序列的最大长度
max_len = 10

#训练模型(示例代码,具体训练数据需替换)
for epoch in range(num_epochs):
    for text in texts:
        inputs = text_to_tensor(text, vocab, embeddings, max_len).unsqueeze(0) #添加 batch
                                                                               #维度
        labels = torch.randint(0, output_size, (1,))      #示例标签
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

该模型定义了一个 RNN 来处理输入的文本序列,将文本字符向量化并传递给 RNN 层,从而预测文本中的人名标签。训练过程中,模型通过前馈传播生成输出并计算损失,利用反向传播更新参数。模型最终输出预测的标签,用于识别输入文本中的人名。

模型输入为文本数据['Zhang San lives in Pudong New District, Shanghai.', 'Li Si works in Beijing.']，每个单词的词嵌入是随机初始化的。文本数据转换为词嵌入向量序列，然后输入 RNN 模型。

```
Epoch [1/10], Loss: 2.4353
Epoch [2/10], Loss: 2.3259
Epoch [3/10], Loss: 2.2451
Epoch [4/10], Loss: 2.3004
Epoch [5/10], Loss: 2.2643
Epoch [6/10], Loss: 2.5340
Epoch [7/10], Loss: 2.2102
Epoch [8/10], Loss: 2.4889
Epoch [9/10], Loss: 2.4526
Epoch [10/10], Loss: 2.2933
```

图 3.7　循环神经网络示例代码运行结果

模型的输出是经过全连接层后的结果，输入损失函数进行计算，如图 3.7 所示。损失函数使用的是交叉熵损失(Cross Entropy Loss)，用于分类任务。在训练过程中，损失值表示模型在当前 Epoch 中对训练数据的预测误差。损失值越小，表明模型的预测越接近真实标签。损失值并没有持续下降，而是出现了一些波动。模型的训练过程的不稳定，可能是由于训练数据过少或随机标签导致的。



3. BERT 模型

1) 基本概念

双向编码器表示转换器(Bidirectional Encoder Representations from Transformers，BERT)是一种基于 Transformer 架构的预训练语言模型，通过预训练大规模无标记文本数据，学习到通用的语言表示，然后在特定任务上进行微调。BERT 模型的创新之处在于它能够在无监督的情况下，利用双向的 Transformer 架构，如图 3.8 所示，捕捉句子中的上下文信息，从而提高对语言的理解能力。

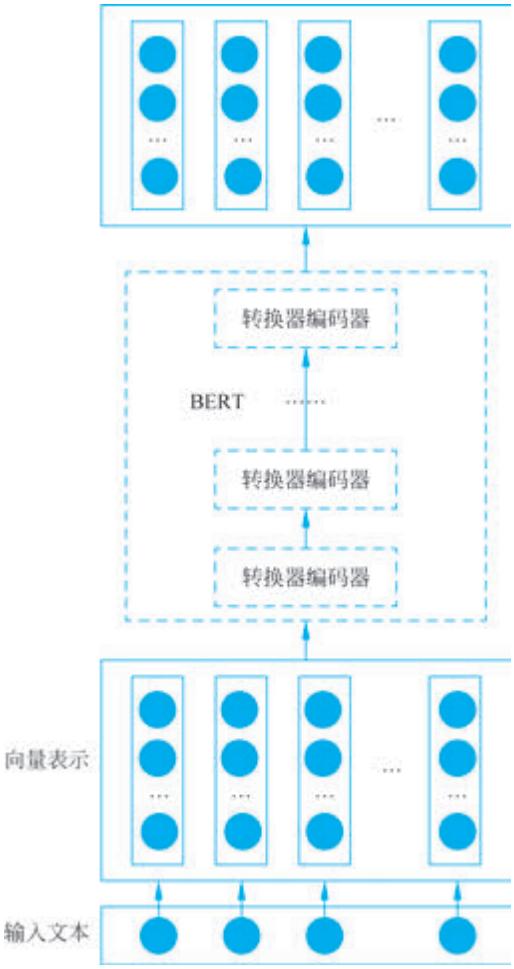


图 3.8　BERT 模型结构

2) 公式解析

(1) 在 MLM(Masked Language Model,掩码语言模型)任务中,BERT 模型通过随机掩码输入文本中的一部分词,然后预测掩码位置的词是什么。MLM 任务的损失函数的计算公式如下:

$$L_{\text{MLM}} = -\sum_{i=1}^{N} \log P(w_i' \mid w_1, \cdots, w_{i-1}, w_{i+1}, \cdots, w_N) \tag{3.17}$$

其中,N 表示输入文本中词的总数;w_i'表示掩码位置的真实词;$P(w_i' \mid w_1, \cdots, w_{i-1}, w_{i+1}, \cdots, w_N)$表示模型预测掩码位置的词的概率。

(2) 在 NSP(Next Sentence Prediction,下一句预测)任务中,BERT 模型通过判断两个句子是否连续出现在文本中,来学习句子级别的语义关系。NSP 任务的损失函数的计算公式如下:

$$L_{\text{NSP}} = -\sum_{i=1}^{M} \log p(\text{IsNext}(s_i)) - \sum_{i=1}^{M} \log p(\text{NotNext}(s_i)) \tag{3.18}$$

其中,M 表示输入样本的数量。$\text{IsNext}(s_i)$表示句子 s_i 是连续的。$\text{NotNext}(s_i)$表示句子 s_i 不是连续的。

3) 训练过程

BERT 模型的训练过程分为预训练和微调两个阶段。

(1) 预训练阶段:在预训练阶段,需要使用大规模无标记文本数据。BERT 模型通过掩码语言模型(MLM)和下一句预测(NSP)等任务,学习到丰富的语言表示,以此预训练 BERT 模型的参数。

(2) 微调阶段:在微调阶段,使用少量标记数据,针对特定任务微调 BERT 模型的参数,例如文本分类、命名实体识别等,通过少量标记数据调整模型参数以适应特定任务。

4) 代码示例

以下代码是一个用 BERT 模型进行文本分类微调的示例:

```
import torch
from transformers import BertTokenizer, BertForSequenceClassification
from torch.utils.data import DataLoader, TensorDataset

#加载预训练的 BERT 模型和分词器
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)

#示例文本数据
texts = ['Zhang San lives in Pudong New District, Shanghai.', 'Li Si works in Beijing.']
#分词和编码
inputs = tokenizer(texts, return_tensors='pt', padding=True, truncation=True)
#微调模型
model.train()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)
labels = torch.tensor([1, 0])  #示例标签

#创建数据集
dataset = TensorDataset(inputs['input_ids'], inputs['attention_mask'], labels)
dataloader = DataLoader(dataset, batch_size=1)

```

```
    # 训练模型
for epoch in range(3):
    total_loss = 0
    for input_ids, attention_mask, labels in dataloader:
        optimizer.zero_grad()
        outputs = model(input_ids = input_ids, attention_mask = attention_mask, labels =
        labels)
        loss = outputs.loss
        total_loss += loss.item()
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch + 1}, Average Loss: {total_loss / len(dataloader)}")
```

该模型利用预训练的 BERT 模型，将输入文本进行分词编码后，使用 BERT 的特征提取能力，结合分类头(classification head)进行二分类任务。模型通过前向传播计算损失，并在每个训练周期内利用反向传播进行参数优化，逐步微调 BERT 模型以适应具体的文本分类任务。

模型的输入包括文本数据和标签，文本数据为"Zhang San lives in Pudong New District, Shanghai.""Li Si works in Beijing."标签包括 1(表示正向标签，如正面情感)，0(表示负向标签，如负面情感)。

模型的输入为每个 Epoch 的平均损失(Loss)，损失值越小，模型在训练集上的表现越好，如图 3.9 所示。

```
Epoch 1, Average Loss: 0.5424288064241409
Epoch 2, Average Loss: 0.6062267422676086
Epoch 3, Average Loss: 0.545927107334137
```

图 3.9　BERT 模型示例代码运行结果

Epoch 1：损失较低，表明模型在第一个 epoch 中开始学习，但仍有优化空间。

Epoch 2：损失增加，这可能是由于模型的学习过程中的不稳定性或过拟合等问题。

Epoch 3：损失再次下降，说明模型在后续的训练过程中开始更好地拟合数据。

4. BiLSTM-CRF 模型


视频讲解

1) 基本概念

双向长短期记忆网络-条件随机场(BiLSTM-CRF)是一种结合了双向长短期记忆网络(BiLSTM)和条件随机场(CRF)的序列标注模型，广泛应用于命名实体识别(NER)等自然语言处理任务中。BiLSTM-CRF 模型通过 BiLSTM 捕捉序列中的上下文信息，并使用 CRF 层进行最优标签序列的解码，能够有效提高序列标注的准确性。

(1) 双向 LSTM 层(BiLSTM Layer)：双向 LSTM 层是 BiLSTM-CRF 模型的核心组件之一，通过在输入序列的前后两个方向分别进行 LSTM 计算，能够捕捉到每个位置上更全面的上下文信息。BiLSTM 层对每个时间步的输入进行前向和后向的处理，然后将这两个方向的输出拼接起来作为最终的表示。

(2) 条件随机场层(CRF Layer)：CRF 层用于在序列标注任务中对标签序列进行全局解码。CRF 层通过建模标签之间的转移关系，能够在考虑上下文信息的基础上对每个时间步的标签进行联合预测，确保输出的标签序列是全局最优的。

(3) 隐藏状态(Hidden State)：在 BiLSTM 层中，每个时间步都会维护一个前向和后向的隐藏状态，这些隐藏状态共同捕捉了输入序列的上下文信息。CRF 层使用 BiLSTM 层的隐藏状态作为输入，进行标签序列的解码。

2) 公式解析

(1) 在介绍双向 LSTM 之前,先解释一下单向 LSTM 的基本公式。LSTM 单元的更新包括以下步骤。

遗忘门(Forget Gate):

$$f_t = \sigma(\boldsymbol{W}_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \tag{3.19}$$

其中,σ 是 Sigmoid 激活函数,$\boldsymbol{W}_f$ 是遗忘门的权重矩阵,$\boldsymbol{b}_f$ 是遗忘门的偏置项,$\boldsymbol{h}_{t-1}$ 是前一个时间步的隐藏状态,$\boldsymbol{x}_t$ 是当前时间步的输入。

输入门(Input Gate):

$$i_t = \sigma(\boldsymbol{W}_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \tag{3.20}$$

其中,$\boldsymbol{W}_i$ 是输入门的权重矩阵,$\boldsymbol{b}_i$ 是输入门的偏置项。

候选记忆单元(Cell Candidate):

$$C_t = \tanh(\boldsymbol{W}_C \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_C) \tag{3.21}$$

其中,$\boldsymbol{W}_C$ 是候选记忆单元的权重矩阵,$\boldsymbol{b}_C$ 是候选记忆单元的偏置项。

记忆单元(Cell State):

$$C_t = f_t \odot C_{t-1} + i_t \odot C_t \tag{3.22}$$

其中,$\odot$ 表示逐元素相乘。

输出门(Output Gate):

$$o_t = \sigma(\boldsymbol{W}_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \tag{3.23}$$

其中,$\boldsymbol{W}_o$ 是输出门的权重矩阵,$\boldsymbol{b}_o$ 是输出门的偏置项。

隐藏状态(Hidden State):

$$\boldsymbol{h}_t = o_t \odot \tanh(C_t) \tag{3.24}$$

(2) 双向 LSTM 层(BiLSTM Layer)。

双向 LSTM 层的前向和后向计算公式如下:

前向 LSTM:

$$\overrightarrow{\boldsymbol{h}_t} = \overrightarrow{\mathrm{LSTM}}(\boldsymbol{x}_t, \overrightarrow{\boldsymbol{h}_{t-1}}) \tag{3.25}$$

后向 LSTM:

$$\overleftarrow{\boldsymbol{h}_t} = \overleftarrow{\mathrm{LSTM}}(\boldsymbol{x}_t, \overleftarrow{\boldsymbol{h}_{t+1}}) \tag{3.26}$$

其中,$\overrightarrow{\boldsymbol{h}_t}$ 和$\overleftarrow{\boldsymbol{h}_t}$ 分别表示前向和后向的隐藏状态,$\boldsymbol{x}_t$ 表示当前时间步的输入。

双向 LSTM 的最终输出为

$$\boldsymbol{h}_t = [\overrightarrow{\boldsymbol{h}_t}; \overleftarrow{\boldsymbol{h}_t}] \tag{3.27}$$

其中,$[\cdot;\cdot]$ 表示向量拼接操作。

(3) 条件随机场层(CRF Layer)。

CRF 层的目标是找到使得标签序列得分最高的路径。得分函数定义为

$$s(X, y) = \sum_{t=1}^{T} (P(y_t \mid h_t) + A(y_{t-1}, y_t)) \tag{3.28}$$

其中,X 表示输入序列,y 表示标签序列,$P(y_t \mid h_t)$表示在时间步 t 上的标签得分,$A(y_{t-1}, y_t)$表示标签之间的转移得分。

3) 训练过程

在模型训练阶段,需要利用标记好的数据来训练 BiLSTM-CRF 模型。具体做法是,将标记好的关系数据分为训练集、验证集和测试集。然后,利用训练集来训练 BiLSTM-CRF 模型,

利用验证集来调整模型的超参数(如学习率、隐藏状态维度等),最后使用测试集评估模型的性能。

在训练过程中,需要定义一个损失函数,用来衡量模型预测结果与真实结果之间的差异。CRF 层通常使用对数似然损失函数,通过最小化损失函数,不断更新模型的参数,从而优化模型的性能。

4) 代码示例

下面是一个简单的基于 BiLSTM-CRF 的实体抽取示例代码,用于从文本中提取人名和地名。

```
import torch
import torch.nn as nn
import torch.optim as optim
from transformers import BertTokenizer, BertModel

class BiLSTM(nn.Module):
    def __init__(self, bert_model, hidden_dim, tagset_size):
        super(BiLSTM, self).__init__()
        self.bert = BertModel.from_pretrained(bert_model)
        self.lstm = nn.LSTM(self.bert.config.hidden_size, hidden_dim
// 2, num_layers=1, bidirectional=True, batch_first=True)
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.loss_fn = nn.CrossEntropyLoss(ignore_index=-1)  # Ignore  padding tokens
                                                             # in the loss calculation
    def forward(self, sentences):
        with torch.no_grad():
            embeddings = self.bert(sentences)[0]
        lstm_out, _ = self.lstm(embeddings)
        emissions = self.hidden2tag(lstm_out)
        return emissions
    def loss(self, sentences, tags):
        emissions = self(sentences)
        # Reshape emissions and tags for CrossEntropyLoss
        emissions = emissions.view(-1, emissions.size(-1))
        tags = tags.view(-1)
        return self.loss_fn(emissions, tags)
    def predict(self, sentences):
        emissions = self(sentences)
        _, predicted_tags = torch.max(emissions, dim=-1)
        return predicted_tags

# 示例数据
sentences = ["张三家住在上海市浦东新区。", "李四在北京工作。"]
labels = [["B-PER", "O", "O", "B-LOC", "I-LOC", "I-LOC", "O"], ["B-PER", "O", "B-
LOC", "O"]]
tokenizer = BertTokenizer.from_pretrained('bert-base-chinese')
tag2idx = {"O": 0, "B-PER": 1, "I-PER": 2, "B-LOC": 3, "I-LOC": 4}
idx2tag = {v: k for k, v in tag2idx.items()}
def encode_labels(labels, tag2idx):
    return [[tag2idx[tag] for tag in label] for label in labels]
def prepare_data(sentences, labels, tokenizer, tag2idx):
    tokenized = [tokenizer.encode(sentence, add_special_tokens=True)
 for sentence in sentences]
```

```
    max_len = max([len(seq) for seq in tokenized])
    padded_sentences = [seq + [tokenizer.pad_token_id] * (max_len - len(seq)) for seq
    in tokenized]
    encoded_labels = encode_labels(labels, tag2idx)
    padded_labels = [label + [-1] * (max_len - len(label)) for label in encoded_
    labels]
    return torch.tensor(padded_sentences), torch.tensor(padded_labels)
X, y = prepare_data(sentences, labels, tokenizer, tag2idx)

#创建模型
model = BiLSTM('bert-base-chinese', hidden_dim=128,
tagset_size=len(tag2idx))
optimizer = optim.Adam(model.parameters(), lr=0.001)

#训练模型
model.train()
for epoch in range(100):
    optimizer.zero_grad()
    loss = model.loss(X, y)
    loss.backward()
    optimizer.step()
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')

#测试模型
model.eval()
with torch.no_grad():
    predictions = model.predict(X)
    for i, sentence in enumerate(sentences):
        tokens = tokenizer.tokenize(sentence)
        predicted_tags = [idx2tag[p.item()] for p in predictions[i] if p.item() != -1]
        print(f'Sentence: {sentence}')
        print(f'Tokens: {tokens}')
        print(f'Predicted tags: {predicted_tags}')
```

模型使用预训练的 BERT 模型生成字符嵌入(embeddings),然后通过双向 LSTM 处理嵌入,最后将输出传递给全连接层,生成实体类别标签的概率分布。训练过程中,模型计算损失并通过反向传播进行优化。在推理阶段,模型对每个句子的字符进行标注,预测出人名和地名的标签。

模型的输入包括文本数据和实际标签,文本数据为"张三家住在上海市浦东新区。""李四在北京工作。"实际标签为["B-PER", "O", "O", "B-LOC", "I-LOC", "I-LOC", "O", "O", "I-LOC", "O", "O", "I-LOC", "I-LOC", "O"],["B-PER", "O", "B-LOC", "O"]。

模型的输出包括训练过程中的损失值以及对句子 1 和句子 2 的预测。训练过程中,损失值从 0.6537 下降到 0.0068,说明模型在训练数据上的拟合情况非常好。损失值逐渐减小表示模型在优化目标上逐步改进,训练的效果也在持续提升。

对句子 1(张三家住在上海市浦东新区。)的预测 Tokens 为['张', '三', '家', '住', '在', '上', '海', '市', '浦', '东', '新', '区', '。'],Predicted tags 为['B-PER', 'O', 'O', 'B-LOC', 'I-LOC', 'I-LOC', 'O', 'O', 'I-LOC', 'O', 'O', 'I-LOC', 'I-LOC', 'B-PER', 'B-PER']。

对句子 2(李四在北京工作。)的预测 Tokens 为['李', '四', '在', '北', '京', '工', '作', '。'],Predicted tags 为['B-PER', 'O', 'B-LOC', 'O', 'O', 'O', 'O', 'O', 'B-PER', 'B-PER', 'O', 'O',

'O', 'O', 'O']。

模型预测的标签部分是正确的，但也有一些不准确的地方，如图 3.10 所示。特别是，“张三家”的标签是 ['B-PER', 'O', 'O']，这与实际的 ['B-PER', 'B-PER', 'O'] 不完全匹配。而“上海市浦东新区”的标签预测大部分是正确的，虽然有些 I-LOC 标签在句子的末尾出现了。需要对更多的测试数据进行评估，以确认模型的泛化能力。

```
Epoch [10/100], Loss: 0.6537
Epoch [20/100], Loss: 0.2393
Epoch [30/100], Loss: 0.0922
Epoch [40/100], Loss: 0.0410
Epoch [50/100], Loss: 0.0229
Epoch [60/100], Loss: 0.0185
Epoch [70/100], Loss: 0.0127
Epoch [80/100], Loss: 0.0105
Epoch [90/100], Loss: 0.0079
Epoch [100/100], Loss: 0.0068
Sentence: 张三家住在上海市浦东新区。
Tokens: ['张', '三', '家', '住', '在', '上', '海', '市', '浦', '东', '新', '区', '。']
Predicted tags: ['B-PER', 'O', 'O', 'B-LOC', 'I-LOC', 'I-LOC', 'O', 'O', 'I-LOC', 'O', 'O', 'I-LOC', 'I-LOC', 'B-PER', 'B-PER']
Sentence: 李四在北京工作。
Tokens: ['李', '四', '在', '北', '京', '工', '作', '。']
Predicted tags: ['B-PER', 'O', 'B-LOC', 'O', 'O', 'O', 'O', 'O', 'B-PER', 'B-PER', 'O', 'O', 'O', 'O', 'O']
```

图 3.10　BiLSTM-CRF 模型示例代码运行结果

在使用深度学习进行关系抽取时，需要注意以下几点。

(1) 数据预处理：数据预处理是关系抽取的前置步骤之一，在进行关系抽取之前对文本数据进行的一系列处理操作。这些操作包括分词、去停用词、词向量表示等。分词是将文本拆分成单词或词组的过程，以便模型理解和处理。去停用词是指去除文本中常见但无实际意义的词语，如“的”“是”等，以减少噪声对模型的影响。词向量表示则是将文本中的词语映射到向量空间，以便模型能够更好地理解词语之间的语义关系。这些预处理步骤有助于提高模型的性能，使其更有效地进行关系抽取任务。

(2) 模型选择：模型选择是在进行深度学习模型设计时的关键步骤，需要根据具体任务的特点来选择合适的模型架构。例如，在文本分类任务中，可以选择卷积神经网络(CNN)或循环神经网络(RNN)等模型。CNN 适用于捕捉局部特征和模式，对于识别文本中的局部结构非常有效；而 RNN 则适用于处理时序数据，能够捕捉文本中的上下文信息。除了 CNN 和 RNN 外，还有诸如长短期记忆网络(LSTM)、门控循环单元(GRU)等模型也可以考虑。在选择模型时，需要综合考虑模型的性能、训练效率、可解释性等因素，并根据任务需求进行权衡和选择。

(3) 超参数调整：超参数调整是在训练过程中的重要步骤，它涉及调整模型的超参数，如学习率、批大小、卷积核大小等。这些超参数的选择直接影响着模型的性能和收敛速度。学习率控制着模型参数更新的步长，合适的学习率能够加快模型收敛速度，提高训练效率；批大小影响着梯度更新的频率和计算效率，选择合适的批大小可以平衡训练速度和模型性能；而卷积核大小等超参数则影响着模型对特征的提取能力。在进行超参数调整时，通常会采用网格搜索、随机搜索、贝叶斯优化等方法，通过在一定范围内搜索最优的超参数组合，以达到提高模型性能的目的。

(4) 模型评估：模型评估是在训练完成后，利用测试集来评估模型性能的过程。常用的评估指标包括准确率、召回率、F1 值等。准确率衡量了模型预测正确样本的比例，召回率衡量了模型能够找出所有真实正例的能力，而 F1 值综合考虑了准确率和召回率，是一个综合性的评价指标。除了这些常见的评估指标外，还可以根据具体任务的特点设计其他自定义的评估

指标。通过模型评估,可以了解模型在真实数据上的表现情况,从而判断模型的优劣,并指导后续的优化和改进。

3.2.5 实体抽取面临的挑战与机遇

目前,命名实体识别在英文文本中取得了显著进展,特别是基于深度学习的技术,如循环神经网络(RNN)、长短期记忆网络(LSTM)和 Transformer 等模型,在此任务中表现出了强大的能力。此外,预训练语言模型 BERT 等在命名实体识别任务中也取得了显著成果。

然而实体识别仍然面临着标签分布不平衡、实体嵌套等问题,这些限制了其在现实应用中的发展。对于中文实体识别,还存在一些特有的挑战,比如中文缺乏自然分词、词汇变化多样、简化表达等现象。实体识别是语义理解和构建知识图谱的关键环节,也是进一步抽取三元组和关系分类的前提。

3.3 关系抽取技术

3.3.1 关系抽取的概念

关系抽取(relation extraction)是信息抽取领域的一个基本任务,于 1988 年在 MUC 大会上首次提出。作为从文本中获取知识图谱三元组的主要技术手段,关系抽取旨在识别并提取文本中两个或多个实体之间的目标关系。这一任务对于知识图谱的补全具有重要意义。

相较于实体提取,关系抽取在现实世界中要复杂得多。自然句子的形式多种多样,关系的提取难度也相应增加。为了准确地识别和分类实体之间的关系,关系抽取不仅需要识别出句子中的实体,还需要深入理解这些实体之间的上下文关系。具体而言,关系抽取涉及对句子的语法结构、语义信息以及上下文线索的分析。通过这些分析,关系抽取能够有效地从非结构化文本中提取出有价值的关系信息,从而为各种 NLP 应用提供支持。


视频讲解

3.3.2 依存句法的概念及关系抽取的应用

依存句法分析是一种自然语言处理技术,用于分析句子的结构。通过分析词语之间的依存关系,可以更好地理解句子的语义信息。在实体关系抽取领域,依存句法分析发挥着重要的作用,主要体现在以下几方面。

1. 确认实体间的关系

通过分析句子中实体之间的依存关系,分析者可以确定它们之间的关系。例如,在句子“猫抓老鼠”中,“猫”和“老鼠”之间存在动作关系,表明“猫”在抓“老鼠”。

2. 识别复杂的语义关系

依存句法分析有助于识别复杂的语义关系,如递进、转折、条件等。这些关系对于理解实体之间的联系至关重要。例如,“如果你学习努力,你就会成功”中的条件关系表明学习努力与成功之间的因果联系。

3. 提取潜在的实体关系

通过分析大量文本数据,可以发现其中潜在的实体关系。这些关系可能不太明显,但通过依存句法分析可以揭示它们。例如,在社交媒体评论中,通过分析用户之间的提及和关系,可以了解他们之间的互动和联系。

以下是使用 Python 和 spacy 库进行依存句法分析并提取实体关系的核心代码的示例。

```
import spacy
#加载 spacy 模型
nlp = spacy.load("en_core_web_sm")
#示例句子
sentence = "The cat caught the mouse."

#进行依存句法分析
doc = nlp(sentence)
#打印依存句法分析结果
for token in doc:
    print(f"Token: {token.text}, Head: {token.head.text}, Dep: {token.dep_}")

#提取实体关系
relations = []
for token in doc:
    #找到名词(实体)之间的依存关系
    if token.dep_ in ("nsubj", "dobj"):
        subject = [w for w in token.head.lefts if w.dep_ == "nsubj"]
        if subject:
            subject = subject[0]
            relations.append((subject.text, token.head.text, token.text))
#打印提取的实体关系
print("Extracted relations:")
for relation in relations:
    print(relation)
```

该代码使用 spaCy 进行依存句法分析，通过解析句子中的词汇关系，识别主语-动词-宾语的依存结构。模型首先加载预训练的语言模型对句子进行分词和依存关系标注，然后根据依存标签提取符合条件的实体关系三元组。最终输出提取的实体关系，展示出文本中的基本句法结构。

输入句子为"The cat caught the mouse." 经过 spaCy 的依存句法分析，分析了每个词的依存关系。

输出中提取了两个关系，('cat', 'caught', 'cat')和('cat', 'caught', 'mouse')，其中('cat', 'caught', 'cat')是无效的，而另一个关系正确地表示了主语 cat 和宾语 mouse 之间的动作 caught，符合句子的实际语义，如图 3.11 所示。

```
Token: The, Head: cat, Dep: det
Token: cat, Head: caught, Dep: nsubj
Token: caught, Head: caught, Dep: ROOT
Token: the, Head: mouse, Dep: det
Token: mouse, Head: caught, Dep: dobj
Token: ., Head: caught, Dep: punct
Extracted relations:
('cat', 'caught', 'cat')
('cat', 'caught', 'mouse')
```

图 3.11　依存句法示例代码运行结果

▶ 3.3.3　基于机器学习的关系抽取

基于机器学习的关系抽取是一种利用机器学习方法从文本中提取实体之间的语义关系的技术。该方法首先将关系抽取问题转换为一个分类任务，然后利用机器学习算法对已标记的大量数据进行特征提取、模型训练和预测，从而自动识别文本中实体之间的关系。这种方法的关键在于特征的设计和模型的构建。特征通常包括词语、句法结构、上下文信息等，可以通过词向量、句法分析、依存句法树等方法进行提取。而模型则可以采用支持向量机(SVM)、逻辑回归、神经网络等机器学习算法来实现。

根据是否需要预先对数据进行标注，基于机器学习的关系抽取方法可以分为监督学习、半监督学习和无监督学习。

1. 监督学习

为了使用监督学习进行关系抽取,需要构建一个包含标注实体对和关系的训练数据集。这些数据集通常由人工标注,以确保标注的准确性和一致性。利用标注好的训练数据集对选定的监督学习模型进行训练。在训练过程中,模型会学习如何根据输入的文本特征预测实体之间的关系。通过使用测试数据集对训练好的模型进行评估,可以了解模型在关系抽取任务上的性能表现。特征向量法与核函数法是关系抽取任务常用的监督学习方法。


视频讲解

1) 特征向量法

特征向量法是一种在关系抽取任务中常用的方法,它利用特征向量来表示从包含特定实体对的句子中提取出的语义特征。这些特征向量包括了诸如词向量、词性标签、句法分析信息等。随后,利用支持向量机、最大熵、条件随机场等模型进行关系抽取。此类模型通过学习特征向量与实体间关系之间的映射关系,从而完成关系抽取任务。特征向量法能够有效地捕获句子中的语义信息,具备一定的灵活性和泛化能力,适用于各种类型的关系抽取任务。

下面是一个简单的示例代码,演示如何使用词袋模型(Bag of Words)和支持向量机进行关系抽取。

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import SVC
from sklearn.pipeline import Pipeline

#示例数据
sentences = ['John likes apples.', 'Alice hates oranges.', 'Mary loves bananas.']
labels = ['positive', 'negative', 'positive']  #正负样本标签
#特征提取器(词袋模型)
vectorizer = CountVectorizer()
#支持向量机分类器
classifier = SVC(kernel='linear')
#构建管道
model = Pipeline([('vectorizer', vectorizer), ('classifier', classifier)])

#训练模型
model.fit(sentences, labels)

#使用模型进行预测
new_sentences = ['Peter enjoys grapes.', 'Emma dislikes pineapples.']
predictions = model.predict(new_sentences)
print(predictions)
```

这个示例代码使用了词袋模型作为特征提取器,并将提取的特征输入支持向量机分类器中进行训练。最后,使用训练好的模型对新的句子进行关系抽取预测。

输入的三个句子分别为'John likes apples.' 'Alice hates oranges.' 'Mary loves bananas.'。句子的情感标签分别为'positive' 'negative' 'positive '。新数据为'Peter enjoys grapes.' 'Emma dislikes pineapples.'。

输出对新句子预测的情感标签均为 'positive'。

模型将 'Peter enjoys grapes.' 和 'Emma dislikes pineapples.' 都预测为 'positive'。这可能是因为这两个句子中的某些特征与训练数据中的正面样本相似,或是由于训练样本非常有限(只有 3 个句子),模型在训练过程中可能存在偏差。模型可能没有足够的信息来准确区分各种情感,因此它可能在新数据上的表现并不优秀。

2）核函数法


视频讲解

核函数法的基本分类和原理在3.2节已进行过相关介绍。核函数法是一种常用于关系抽取任务的方法，它通过核函数计算两个实体之间的相似度，并利用这种相似度来训练分类模型。核函数可以将输入的数据映射到更高维的特征空间，从而使得线性模型能够更好地拟合非线性关系。在关系抽取任务中，通常会使用核函数计算实体对之间的相似度，例如，常用的核函数包括高斯核函数、多项式核函数等。然后，利用这些相似度信息，可以训练分类模型来判断实体对之间是否存在特定的关系。核函数法能够有效地捕捉实体之间的语义关系，具有一定的鲁棒性和泛化能力，因此在关系抽取任务中得到了广泛应用。

下面是一个简单的示例代码，演示如何使用高斯核函数和支持向量机进行关系抽取。

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import SVC
from sklearn.metrics.pairwise import rbf_kernel

#示例数据
entity_pairs = [('John', 'apples'), ('Alice', 'oranges'), ('Mary', 'bananas')]
labels = ['positive', 'negative', 'positive']  #正负样本标签
#将实体对转换为字符串
entity_pairs_str = [' '.join(pair) for pair in entity_pairs]
#使用CountVectorizer将字符串转换为数值表示
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(entity_pairs_str)
#计算实体对之间的相似度矩阵
similarity_matrix = rbf_kernel(X)

#支持向量机分类器
classifier = SVC(kernel='precomputed')

#训练模型
model = classifier.fit(similarity_matrix, labels)

#使用模型进行预测
new_entity_pairs = [('Peter', 'grapes'), ('Emma', 'pineapples')]
new_entity_pairs_str = [' '.join(pair) for pair in new_entity_pairs]
new_X = vectorizer.transform(new_entity_pairs_str)
new_similarity_matrix = rbf_kernel(new_X, X)
predictions = model.predict(new_similarity_matrix)
print(predictions)
```

这个示例代码使用了高斯核函数计算实体对之间的相似度矩阵，并将相似度矩阵作为输入训练支持向量机分类器。然后，使用训练好的模型对新的实体对进行关系抽取预测。

输入的实体对为[('John', 'apples'), ('Alice', 'oranges'), ('Mary', 'bananas')]，标签为['positive', 'negative', 'positive']，新实体对为[('Peter', 'grapes'), ('Emma', 'pineapples')]。

输出的结果为['positive', 'positive']。

由于'Peter grapes'和'Emma pineapples'与'John apples'和'Mary bananas'在词向量空间中具有相似性，因此SVM分类器预测它们为positive。模型在训练数据上已经学会了区分正负样本标签，且在新数据上也保持了一致的分类效果。

2. 半监督学习


视频讲解

半监督学习利用标记数据（少量）和未标记数据（大量）来训练模型，目的是提高模型的泛

化能力和预测的准确性。相比于纯监督学习,半监督学习在标签数据稀缺的情况下能更有效地利用无标签数据。在关系抽取领域主要有两种方法:远程监督学习和 Bootstrapping 方法。

远程监督方法通过将知识库与非结构化文本对齐,自动构建大量训练数据,减少对人工标注数据的依赖,增强模型的跨领域适应能力。其基本假设是,如果两个实体在知识图谱中存在某种关系,那么包含这两个实体的句子就可能表达这种关系。

Bootstrapping 方法则是利用少量初始实例作为种子集合,然后根据这些实例学习关系抽取的模板,并不断迭代,从文本中抽取更多的关系实例。这种方法的优点在于关系抽取的构建成本低,适合大规模任务,并且具备发现新关系的能力。然而,它也面临着一些挑战,例如对初始种子的敏感性、语义漂移问题以及结果准确率较低等。

半监督学习通过结合少量标注数据和大量未标注数据,可以在数据标注成本较高的情况下提升模型性能。Bootstrapping 是一种半监督学习方法,通常用于关系抽取任务。它利用少量的标注种子实例,通过迭代扩展来逐步提高模型的精度。

以下是一个完整的使用 Bootstrapping 方法进行关系抽取的 Python 示例代码。此代码通过从种子实例开始,逐步扩展关系集合,从文本语料库中提取“创始人-公司”关系。

```
import re
#初始种子实例 (Founder, Company)
seed_relations = [
    ("Steve Jobs", "Apple"),
    ("Bill Gates", "Microsoft")
]
#示例文本语料库
corpus = [
    "Steve Jobs founded Apple in 1976.",
    "Bill Gates founded Microsoft.",
    "Elon Musk started SpaceX.",
    "Mark Zuckerberg created Facebook in his dorm room.",
    "Larry Page and Sergey Brin founded Google.",
    "Jeff Bezos founded Amazon in 1994.",
    "Steve Wozniak co-founded Apple with Steve Jobs."
]

#函数:查找符合关系模式的句子
def find_relation_patterns(corpus, seed_relations):
    patterns = []
        #简单的模式识别,寻找类似 "X founded Y" 的结构
    pattern = re.compile(r"(\b[A-Z][a-z]+\s+[A-Z][a-z]+\b)\s+founded\s+(\b[A-Z][a-z]+\b)")
        for sentence in corpus:
        match = pattern.search(sentence)
        if match:
            founder, company = match.groups()
            #检查是否是新的关系,如果是,添加到种子实例中
            if (founder, company) not in seed_relations:
                patterns.append((founder, company))
        return patterns

#使用种子实例进行关系扩展
new_relations = find_relation_patterns(corpus, seed_relations)
#将找到的新关系加入种子实例中
```

```
    seed_relations.extend(new_relations)

    #输出结果
    print("Updated relations:")
    for relation in seed_relations:
        print(f"Founder: {relation[0]}, Company: {relation[1]}")
        print(classification_report(y_test, y_pred, labels=[0, 1, 2]))
```

输入的初始种子实例（seed_relations）为初始的“创始人-公司”关系对，用于引导 Bootstrapping 过程。seed_relations = [("Steve Jobs", "Apple"),("Bill Gates", "Microsoft")]文本语料库（corpus）包含若干句子的列表，这些句子中存在要提取的“创始人-公司”关系。

```
Updated relations:
Founder: Steve Jobs, Company: Apple
Founder: Bill Gates, Company: Microsoft
Founder: Sergey Brin, Company: Google
Founder: Jeff Bezos, Company: Amazon
```

图 3.12　Bootstrapping 模型示例代码运行结果

代码成功地通过 Bootstrapping 方法，从初始种子实例和文本语料库中提取了新的关系对，如图 3.12 所示，提取出的新关系对包括 Sergey Brin -> Google 和 Jeff Bezos -> Amazon。


视频讲解

3. 无监督学习

监督学习和半监督学习都需要提前确定关系的类型，事实上，在大规模语料库中，人们往往无法预测所有类型的实体关系，因此，无监督学习特别适用于标记数据稀缺或获取成本高昂的场景。无监督学习是一种不依赖标注数据进行训练的机器学习方法，而基于无监督学习的关系抽取则是指在没有标注数据的情况下，从文本中自动提取实体之间的关系的技术。

在这种方法中，通常使用一些聚类、聚合或图分析等技术来从文本数据中发现实体之间的关联性，进而推断它们之间的关系，包括从文本中提取出共现模式、词语共现统计信息、语义相似度等特征，并使用这些特征来推断实体之间的关系。无监督方法通常需要大规模的语料库作为支持。利用语料库的冗余度，挖掘可能的关系模式集，发现未知领域中的新关系。然而，无监督学习方法在关联名称的准确描述和低频关联的召回率方面仍存在挑战。

下面是一个完整的代码示例，演示如何使用 K-Means 聚类算法进行无监督关系抽取。

```
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans
import numpy as np

#示例句子
sentences = [
    "<Person> is the CEO of <Organization>",
    "<Person> was born in <Location>",
    "<Person> works at <Organization>",
    "<Person> lives in <Location>",
    "<Person> is the president of <Organization>"
]

#步骤 1:文本预处理和特征提取
#使用 TF-IDF 将句子向量化
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(sentences)

#步骤 2:应用 K-Means 聚类
```

```
# 假设我们希望发现 2 种主要关系类型
num_clusters = 2
kmeans = KMeans(n_clusters = num_clusters, random_state = 42)
kmeans.fit(X)
# 获取每个句子的聚类标签
labels = kmeans.labels_

# 步骤 3:分析聚类结果
# 输出每个句子对应的聚类标签
for sentence, label in zip(sentences, labels):
    print(f"Sentence: '{sentence}' is in cluster: {label}")

# 步骤 4:显示聚类中心的关键词
# 找出每个聚类的主要特征词
order_centroids = kmeans.cluster_centers_.argsort()[:, ::-1]
terms = vectorizer.get_feature_names_out()
```

这段代码示例展示了如何使用 K-Means 聚类算法进行无监督关系抽取。首先,它对示例句子进行文本预处理和特征提取,利用 TF-IDF 向量化句子。接着,使用 K-Means 聚类算法将句子分为两类,并为每个句子分配聚类标签。最后,输出每个句子的聚类结果,并分析每个聚类的主要特征词,以便更好地理解所提取的关系。

输入的文本数据为 sentences = ["< Person > is the CEO of < Organization >", "< Person > was born in < Location >", "< Person > works at < Organization >", "< Person > lives in < Location >", "< Person > is the president of < Organization >"]。

输出如图 3.13 所示,聚类 0 包含了句子 < Person > is the CEO of < Organization >, < Person > works at < Organization >和 < Person > is the president of < Organization >。这些句子描述了某人与组织之间的职业关系。

```
Sentence: '<Person> is the CEO of <Organization>' is in cluster: 1
Sentence: '<Person> was born in <Location>' is in cluster: 0
Sentence: '<Person> works at <Organization>' is in cluster: 1
Sentence: '<Person> lives in <Location>' is in cluster: 0
Sentence: '<Person> is the president of <Organization>' is in cluster: 1
```

图 3.13　K-Means 聚类算法示例代码运行结果

聚类 1 包含了句子 < Person > was born in < Location > 和 < Person > lives in < Location >。这些句子描述了某人与地点之间的出生或居住关系。

▶ 3.3.4　基于深度学习的关系抽取

随着深度学习方法在自然语言处理领域的广泛应用,深度神经网络也被成功应用于关系抽取任务,并取得了显著的进展。与传统的统计模型相比,基于深度学习的方法直接以文本中词的向量为输入,通过神经网络实现端到端的关系抽取,不再依赖人工定义的特征。当前,主要用于关系抽取的深度学习模型包括 CNN 卷积神经网络、RNN 循环神经网络和引入注意力机制的神经网络。这些神经网络结构通常作为编码器,基于输入的词向量及其上下文信息,生成文本中实体对之间关系的表示。最终,网络通过分类器输出实体对的关系标签,从而实现关系抽取,基于深度学习的实体关系抽取框架如图 3.14 所示。

(1) 获取有标签数据:获取有标签数据是有监督方法和远程监督方法的关键步骤。有监督方法通过人工标记的方式获取有标签数据集,即由人工标记者对文本数据进行手动标注,确

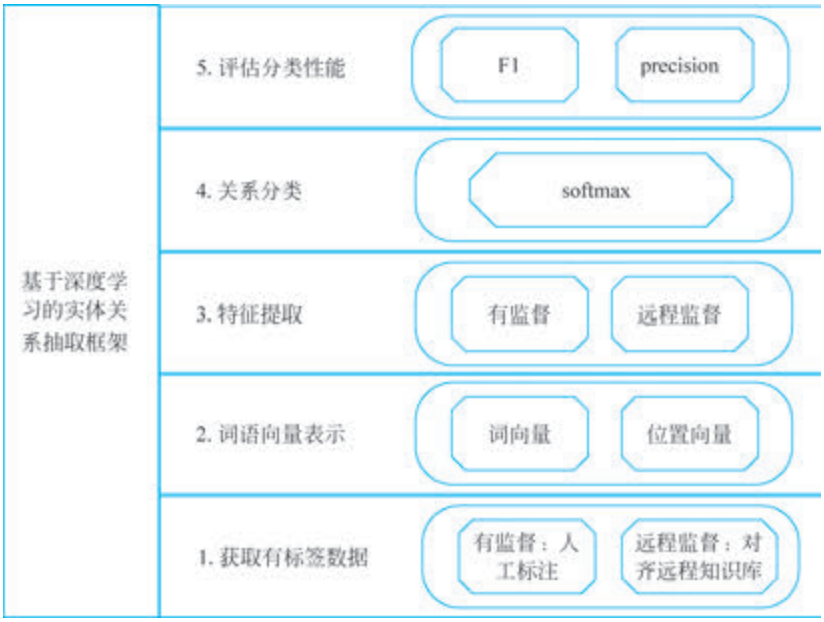


图 3.14　基于深度学习的实体关系抽取框架

定文本中的实体和关系，并为其打上相应的标签。这种方法通常需要大量的人力和时间成本，但标注的数据质量相对较高，适用于训练高质量的模型。

而远程监督方法则通过自动对齐远程知识库和文本数据，从而获取有标签数据集。远程监督方法利用现有的知识库，例如维基百科或 Freebase 等，自动将知识库中的实体和关系对齐到文本数据中，并将其作为标签数据集。尽管远程监督方法可以快速获得大量的标签数据，但其存在着误差传播和不一致性等问题，因此需要结合其他方法进行数据清洗和验证，以提高数据的质量和可信度。

（2）构建词向量表示：将有标签句子分词，将每个词语编码成计算机可以接收的词向量，并求出每个词语与句子中实体对的相对位置，作为这个词语的位置向量，将词向量与位置向量组合作为这个词语的最终向量表示。

（3）进行特征提取：进行特征提取的过程包括将句子中每个词语的向量表示输入神经网络中，然后利用神经网络模型提取句子特征，最后训练一个特征提取器。在这个过程中，每个词语的向量表示可以通过预训练的词嵌入模型（如 Word2Vec、GloVe 等）获得。神经网络模型通常采用卷积神经网络（CNN）、循环神经网络（RNN）、长短期记忆网络（LSTM）或注意力机制等模型来提取句子特征。这些模型能够有效地捕捉句子中的语义信息和结构特征，并将其转换为高维的句子特征表示。最后，利用这些特征训练一个特征提取器，以用于后续的关系抽取任务。这种方法能够充分利用神经网络模型的表征学习能力，提取句子的丰富语义特征，从而提高关系抽取的性能和泛化能力。

（4）关系分类：关系分类是关系抽取任务中的一个关键步骤，它通过将特征提取出的向量放入非线性层进行分类，最终提取出实体对之间的关系。在这个过程中，首先根据预先定义好的关系种类，将特征提取出的向量作为输入，传入一个或多个非线性层，如全连接层、多层感知器等，以进行关系分类。这些非线性层能够对输入的特征进行复杂的变换和组合，从而学习到特定关系的表示。最后，利用分类器输出的结果，即实体对之间的关系类型，作为最终的关系分类结果。这种方法能够有效地利用神经网络的表征学习能力，提高关系抽取的准确性和

泛化能力,适用于各种类型的关系分类任务。

(5) 评估分类性能:评估分类性能是关系抽取任务中的最后一步,它旨在对关系分类结果进行客观、全面的评估。通常,可以使用各种评估指标来衡量分类性能,其中包括但不限于准确率、召回率、F1 值等。


视频讲解

1. 基于 CNN 的关系抽取

CNN 的基本概念在 3.2 节已进行过相关介绍,基于卷积神经网络(CNN)的关系抽取是一种利用 CNN 模型从文本中提取实体之间关系的方法。CNN 作为一种深度学习模型,主要用于处理图像类的网格化数据,也可应用于处理序列数据,例如文本,通过将文本表示为二维矩阵并利用卷积操作来提取局部特征。

在基于 CNN 的关系抽取中,首先将文本中的实体和上下文转换为词向量表示,并组合成一个二维矩阵作为 CNN 的输入。然后,利用多个卷积核对这个矩阵进行卷积操作,以捕获不同层次的特征。卷积操作可以识别文本中的局部模式和特征,有助于提取实体之间的关系。接着,通过池化操作对卷积后的特征进行降维和提取最重要的信息。最后,通过全连接层或其他神经网络结构进行关系分类。

CNN 的基本结构包括两层:一为特征提取层,每个神经元的输入与前一层的局部接收域相连,并提取该局部的特征;二是特征映射层,网络的每个计算层由多个特征映射组成,每个特征映射是一个平面,平面上所有神经元的权值相等,减少了网络中自由参数的个数。由于同一特征映射面上的神经元权值相同,所以 CNN 网络可以并行学习。结合句子级与词汇级特征的关系抽取框架如图 3.15 所示。

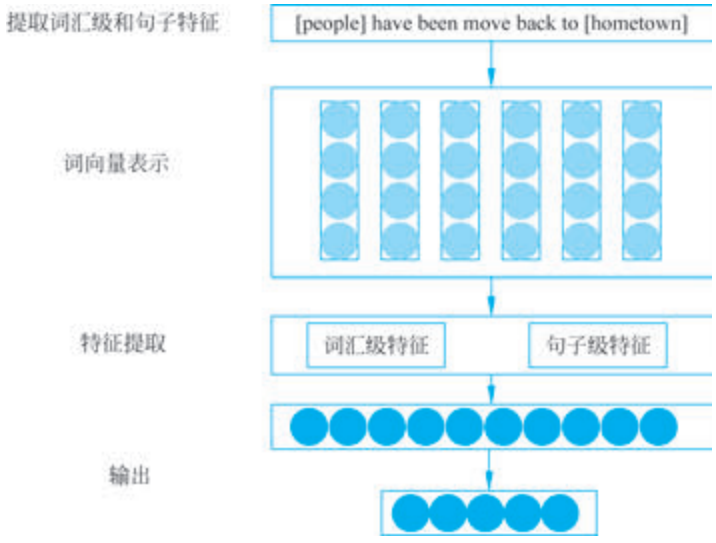


图 3.15　基于 CNN 的关系抽取框架

下面是一个简单的基于卷积神经网络(CNN)的关系抽取模型的示例代码。这个模型将句子作为输入,通过卷积层提取特征,并最终分类以识别句子中的关系。

```
1.    import torch
2.    import torch.nn as nn
3.    import torch.nn.functional as F
4.    import torch.optim as optim
5.    from torch.utils.data import DataLoader, Dataset
```

```
import numpy as np

#定义一个简单的关系抽取数据集
class RelationExtractionDataset(Dataset):
    def __init__(self, sentences, labels, word_to_idx):
        self.sentences = sentences
        self.labels = labels
        self.word_to_idx = word_to_idx
    def __len__(self):
        return len(self.sentences)
    def __getitem__(self, idx):
        sentence = self.sentences[idx]
        label = self.labels[idx]
        #将句子转换为索引
        indices = [self.word_to_idx[word] for word in sentence]
        return torch.tensor(indices), torch.tensor(label)

#定义 CNN 模型
class CNNRelationExtractor(nn.Module):
     def __init__(self, vocab_size, embedding_dim, num_filters, filter_sizes, num_
     classes):
        super(CNNRelationExtractor, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.convs = nn.ModuleList([ nn.Conv2d(1, num_filters, (fs, embedding_dim)) for
        fs in filter_sizes ])
        self.fc = nn.Linear(num_filters * len(filter_sizes), num_classes)
    def forward(self, x):
        x = self.embedding(x)  #[batch_size, seq_len, embedding_dim]
        x = x.unsqueeze(1)     #增加通道维度 [batch_size, 1, seq_len, embedding_dim]
        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs]  #卷积并应用 ReLU
        x = [F.max_pool1d(feature_map, feature_map.size(2)).squeeze(2) for feature_map
        in x]                  #池化
        x = torch.cat(x, 1)    #拼接所有的特征映射
        x = self.fc(x)         #全连接层
        return F.log_softmax(x, dim=1)

#示例数据
sentences =[
    ["the", "cat", "sits", "on", "the", "mat"],
    ["the", "dog", "barked", "at", "the", "man"],
    ["the", "bird", "flew", "over", "the", "river"]
]
labels = [0, 1, 0]             #示例标签
word_to_idx = {word: i for i, word in enumerate(set(sum(sentences, [])))}  #词汇表
#超参数
vocab_size = len(word_to_idx)
embedding_dim = 50
num_filters = 10
filter_sizes = [2, 3, 4]
num_classes = 2
learning_rate = 0.001
batch_size = 2
num_epochs = 10
#数据集和数据加载器
dataset = RelationExtractionDataset(sentences, labels, word_to_idx)
```

```
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
#初始化模型、损失函数和优化器
model = CNNRelationExtractor(vocab_size, embedding_dim, num_filters, filter_sizes, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

#训练模型
for epoch in range(num_epochs):
    for sentences, labels in data_loader:
        optimizer.zero_grad()
        output = model(sentences)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}")

#测试模型(使用同一数据集作为示例)
model.eval()
with torch.no_grad():
    for sentences, labels in data_loader:
        output = model(sentences)
        predictions = output.argmax(dim=1)
        print(f"Predictions: {predictions.numpy()}, Ground Truth: {labels.numpy()}")
```

这段代码示例展示了如何使用卷积神经网络(CNN)进行关系抽取。首先,它定义了一个关系抽取数据集类,用于处理输入句子及其对应标签,并将句子转换为词汇索引。接着,定义了一个 CNN 模型,该模型通过嵌入层和多个卷积层提取特征,最后通过全连接层进行分类。代码包含超参数的设置、数据集的创建、模型的初始化,以及训练和测试模型的过程。在训练过程中,模型通过反向传播优化损失函数以提高分类准确性。最后,通过在同一数据集上进行测试,模型输出了预测结果与真实标签的比较。

模型的输入为 sentences = [["the", "cat", "sits", "on", "the", "mat"], ["the", "dog", "barked", "at", "the", "man"], ["the", "bird", "flew", "over", "the", "river"]]。

```
Epoch 1/10, Loss: 0.8077706098556519
Epoch 2/10, Loss: 0.5394904613494873
Epoch 3/10, Loss: 0.3211021423339844
Epoch 4/10, Loss: 0.22638840973377228
Epoch 5/10, Loss: 0.32969215512275696
Epoch 6/10, Loss: 0.13810425996780396
Epoch 7/10, Loss: 0.10006216913461685
Epoch 8/10, Loss: 0.08936918526887894
Epoch 9/10, Loss: 0.18008601665496826
Epoch 10/10, Loss: 0.056846778839826584
Predictions: [0 0], Ground Truth: [0 0]
Predictions: [1], Ground Truth: [1]
```

图 3.16　CNN 示例代码运行结果

每个 Epoch 结束后,模型会输出当前 Epoch 的损失值(Loss)。从结果中可以看到,损失值逐渐减小,表示模型的性能随着训练的进行而逐步提高,如图 3.16 所示。

Predictions 是模型对测试数据的预测结果。每个数字表示模型预测的类别标签。在这个例子中,模型在第一个批次预测了两个样本的标签 [0, 0],在第二个批次预测了一个样本的标签 [1]。

Ground Truth 是测试数据的真实标签。每个数字表示该样本的实际类别标签。在这个例子中,第一个批次的真实标签为 [0, 0],第二个批次的真实标签为 [1]。在测试数据上,模型正确地预测了每个样本的标签,这表明模型在这个简单的数据集上表现良好。


视频讲解

2. 基于 RNN 的关系抽取

RNN 的基本概念在 3.2 节已进行过相关介绍,基于 RNN 模型进行关系抽取的方法通常涉及将输入文本转换为向量表示。具体来说,RNN 模型会为文本中的每个词分配一个向量,

这些向量捕捉了词的语义信息。在关系抽取任务中，模型通过遍历文本的结构（例如，通过时间步或上下文序列）来捕捉词之间的关系。每个词的向量在传递过程中会被更新，RNN 通过这些更新捕捉相邻词或短语间的关系。最终，模型根据这些向量表示来判断词对之间的关系。

RNN 代码在 CNN 的基础上修改即可：

```
#定义 RNN 模型
class RNNRelationExtraction(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_classes):
        super(RNNRelationExtraction, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.LSTM(embedding_dim, hidden_dim, batch_first = True)
        self.fc = nn.Linear(hidden_dim, num_classes)
        def forward(self, x):
        x = self.embedding(x)
        x, _ = self.rnn(x)
        x = self.fc(x[:, -1, :])  #取最后一个时间步的输出
        return x

#训练模型
model = RNNRelationExtraction(vocab_size, embedding_dim, hidden_dim, num_classes)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
criterion = nn.CrossEntropyLoss()
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

模型的输入为 sentences = [["the", "cat", "sits", "on", "the", "mat"], ["the", "dog", "barked", "at", "the", "man"], ["the", "bird", "flew", "over", "the", "river"]]。

模型输出的每一行代表一个训练周期（epoch）结束时的损失值。从 Epoch 1 到 Epoch 10，损失值逐渐降低，表明模型在训练过程中逐步学会了如何更好地进行预测，表现出训练过程中的学习效果，如图 3.17 所示。

```
Epoch 1/10, Loss: 0.8279173970222473
Epoch 2/10, Loss: 0.6722185015678406
Epoch 3/10, Loss: 0.5498077869415283
Epoch 4/10, Loss: 0.4522491991519928
Epoch 5/10, Loss: 0.37143924832344055
Epoch 6/10, Loss: 0.30225637555122375
Epoch 7/10, Loss: 0.24255891144275665
Epoch 8/10, Loss: 0.19190025329589844
Epoch 9/10, Loss: 0.1501738429069519
Epoch 10/10, Loss: 0.1168246641755104
Predictions: [0 1 1], Ground Truth: [0 1 1]
```

图 3.17　RNN 示例代码运行结果

Predictions：[0 1 1]表示模型对测试数据的预测结果。Ground Truth：[0 1 1]是对应的真实标签。预测结果和真实标签完全一致，表示模型在当前测试集上的预测非常准确。

3. 基于 LSTM 的关系抽取

由于梯度消失、梯度爆炸等问题，传统的 RNN 在实际中很难处理长期依赖，后面时间的节点对于前面时间的节点感知力下降。而 LSTM 网络通过 3 个门控操作及细胞状态解决了这些问题，能够从语料中学习到长期依赖关系。

基于长短期记忆网络（LSTM）的关系抽取是利用 LSTM 模型来识别文本中实体之间的关系的方法。LSTM 是一种循环神经网络（RNN）的变体，能够有效地处理序列数据并解决梯度消失或梯度爆炸等问题。

在关系抽取任务中，基于 LSTM 的方法通常将文本中的每个单词或字符表示为向量，并

将其作为输入序列提供给 LSTM 模型。LSTM 模型会逐步处理输入序列,并利用其内部的门控机制来捕获序列中的长期依赖关系。通过对整个文本序列进行建模,LSTM 能够学习到单词之间的语义信息和上下文关系,从而推断出实体之间的关系。

LSTM 通过一系列门控机制来控制信息的流动,具体包括以下公式。

(1) 输入门:

$$i_t = \sigma(\boldsymbol{W}_i \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \tag{3.29}$$

(2) 遗忘门:

$$f_t = \sigma(\boldsymbol{W}_f \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \tag{3.30}$$

(3) 输出门:

$$o_t = \sigma(\boldsymbol{W}_o \cdot [\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \tag{3.31}$$

Yan 等在 2015 年提出了基于 LSTM 的融合句法依存分析树的最短路径以及词向量特征、词性特征、WordNet 特征、句法类型特征来进行关系抽取,该论文的模型如图 3.18 所示。

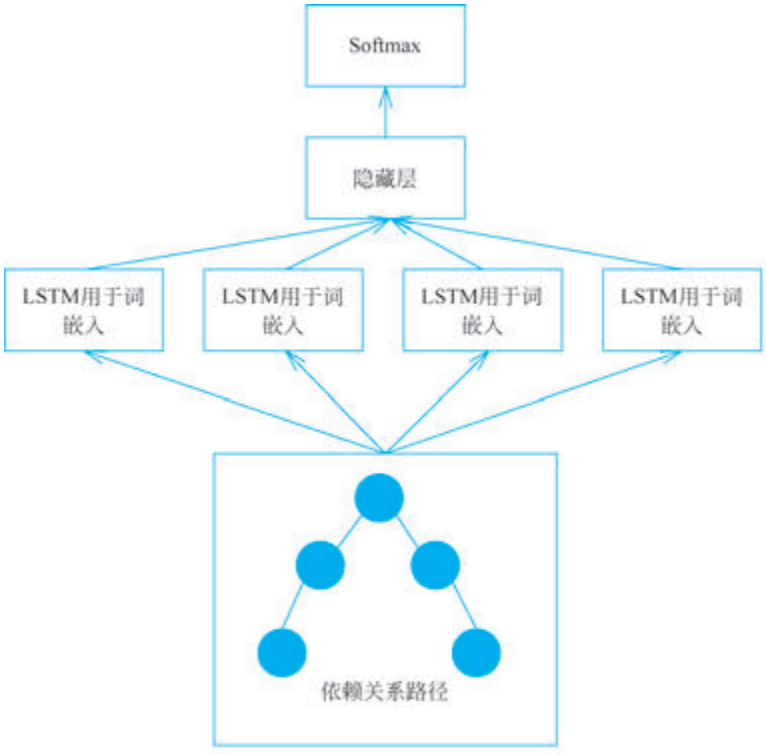


图 3.18　基于 LSTM 及最短依存路径的关系抽取方法

首先,如图 3.18 下半部分所示,利用斯坦福解析器将句子解析为依赖树,并提取最短依赖路径(SDP)作为网络的输入,沿着 SDP 使用 4 种不同类型的信息(称为通道),包括单词、词性标签、语法关系和 WordNet 上位词;在每个通道中,词语被映射成向量,捕获输入的基本含义,两个递归神经网络分别沿着 SDP 的左右子路径获取信息,网络中的 LSTM 单元用于有效信息的传播;之后,如图 3.18 上半部分所示,最大池化层从每个路径中的 LSTM 节点收集信息,来自不同通道的池化层连接在一起,然后输入隐藏层;最后,使用 Softmax 输出层用于关系分类。

▶ 3.3.5　关系抽取面临的挑战和机遇

关系抽取技术迎来了许多机遇，技术创新推动了其识别准确性的提升，语义角色标注和注意力机制等方法能够更有效地捕捉实体间的关系。在大数据和迁移学习的支持下，关系抽取技术在跨领域和跨语言的应用中展现出更强的适应性。同时，关系抽取的应用范围不断扩展，在知识图谱构建和情感分析等领域发挥着重要作用，显示出巨大的价值与潜力。

与此同时，关系抽取技术也面临诸多挑战，尤其在语义理解的复杂性、数据稀缺以及计算资源需求方面。自然语言的多样性和歧义性要求模型能够准确把握上下文并推理隐含关系，而标注数据的不足则限制了模型的泛化能力，尤其在特定领域的任务上尤为突出。此外，高性能计算资源的成本较高，许多研究者难以获得充足的资源来支持关系抽取任务。

3.4　事件抽取技术

▶ 3.4.1　事件抽取的概念

事件抽取是指将包含事件信息的非结构化文本转换为结构化形式的过程。作为信息抽取领域的重要分支，事件抽取致力于从文本中识别和提取具体的事件及其相关要素，如事件触发词、参与者、时间、地点和其他背景信息。近年来，事件抽取技术在多个领域得到了广泛应用，包括情报分析、智能问答、信息检索和推荐系统等。这项技术不仅能够提升信息处理的效率和准确性，还能为各类应用提供高质量的基础数据支持，促进更深入的决策分析和知识推理。


视频讲解

▶ 3.4.2　管道式事件抽取

管道式事件抽取是一种常见的文本信息处理方法，旨在从文本中抽取出事件信息。它通常包括多个阶段，每个阶段都执行特定的任务，最终将文本中的事件信息提取出来。

事件抽取是管道式事件抽取的最终目标，其目标是从文本中抽取出描述事件的关键信息，包括事件的主体、时间、地点、动作等。事件抽取通常需要对文本进行语义分析和推理，结合实体识别和关系抽取的结果进行综合分析。

BERT 模型是谷歌于 2018 年 10 月底发布的，旨在解决 word2vec 等模型存在的不足。BERT 的基本概念在 3.2 节已进行过相关介绍，相比之前的预训练模型（如 word2vec、ELMo 等），BERT 采用了模型迁移的方法，将预训练模型与下游任务模型相结合。其核心目标是将下游 NLP 任务的活动逐渐迁移到预训练生成的词向量上。作为一种深度双向 Transformer 模型，BERT 通过在大规模无标签文本上进行预训练，学习到了丰富的语义表示。在 BERT 事件抽取中，首先需要对输入文本进行分词，并将分词结果转换为对应的词向量。然后，将这些词向量输入 BERT 模型中进行编码。BERT 模型通过多层 Transformer 编码器对输入序列进行处理，从而得到每个词的上下文相关表示。

BERT 的出现标志着自然语言处理领域的重要进步，因为它能够更好地理解语言的上下文和语义关系。在预训练阶段，BERT 通过执行预训练任务来学习语言的深层表示，其中包括掩码语言模型（MLM）和下一句预测（NSP）等任务。在微调阶段，BERT 可以根据具体的 NLP 任务添加任务特定的输出层进行微调，以适应不同的应用场景，如情感分析、问答、命名实体识别等。

BERT 模型的结构由多层 Transformer 编码器组成，每一层都包含自注意力机制和前馈

神经网络,使得模型能够捕捉不同级别的语言信息。在性能表现方面,BERT 在多个 NLP 任务上取得了当时最先进的结果,推动了自然语言处理技术的发展,并激发了对 BERT 结构改进和在不同语言及领域中应用的后续研究工作。

下面是一个简单的管道式事件抽取的实战代码。假设一段文本,下面将使用 spaCy 库来进行实体识别,然后手动标注一些关系,最后抽取出事件信息。可以根据自己的需求和具体场景进一步扩展和优化这段代码。下面是示例代码:

```
import spacy

#加载 Spacy 的英文模型
nlp = spacy.load("en_core_web_sm")
#定义待处理的文本
text = "Elon Musk is the CEO of Tesla, Inc., a company based in California. He was born on June 28, 1971."

#使用 Spacy 进行实体识别
doc = nlp(text)
#打印实体识别结果
print("Entities:")
for ent in doc.ents:
    print(ent.text, ent.label_)
#手动标注一些关系
relations = [("Elon Musk", "CEO", "Tesla, Inc."), ("Tesla, Inc.", "based in", "California")]
#打印关系抽取结果
print("\nRelations:")
for relation in relations:
    print(relation)
#手动抽取事件信息
events = {"Person": "Elon Musk", "Action": "born", "Date": "June 28, 1971"}
#打印事件抽取结果
print("\nEvents:")
for key, value in events.items():
    print(f"{key}: {value}")
```

模型的输入为 text = "Elon Musk is the CEO of Tesla, Inc., a company based in California. He was born on June 28, 1971."。

模型的输出为 Entities:Elon Musk PERSON,Tesla, Inc. ORG,California GPE,June 28, 1971 DATE。Relations:('Elon Musk', 'CEO', 'Tesla, Inc.'),('Tesla, Inc.', 'based in', 'California')。Events:Person: Elon Musk, Action: born,Date: June 28, 1971。

```
Entities:
Elon Musk PERSON
Tesla, Inc. ORG
California GPE
June 28, 1971 DATE

Relations:
('Elon Musk', 'CEO', 'Tesla, Inc.')
('Tesla, Inc.', 'based in', 'California']

Events:
Person: Elon Musk
Action: born
Date: June 28, 1971
```

图 3.19 管道式事件抽取示例代码运行结果

实体识别效果良好,模型准确地识别了文本中的人名、组织、地点和日期。标记类型符合预期,显示了 Spacy 在命名实体识别任务中的有效性,如图 3.19 所示。

手动标注的关系展示了公司职位和公司位置的联系。结果准确地反映了文本中的关系。然而,关系的抽取是手动完成的,实际应用中可能需要自动化工具来提高效率和准确性。

事件信息正确地识别了人物、事件动作和日期。事

件抽取显示了文本中的关键时间点和人物，但这种手动方法可能不适合大规模数据处理，实际应用中通常需要自动化的方法来提升处理能力。


视频讲解

▶ 3.4.3　联合式事件抽取

联合式事件抽取综合考虑了事件触发词识别/分类和事件元素识别/分类的方法。不再将事件抽取任务分解为两个独立的阶段，而是将事件触发词和事件元素的识别作为一个整体来处理，并同时考虑它们之间的关系。

具体而言，联合式事件抽取模型通过一个统一的模型来联合建模事件触发词和事件元素的识别任务。该模型同时考虑文本中的事件触发词及其相关的事件元素，从而在一个统一的框架下完成事件抽取任务。这种方法能够更好地捕捉实体与关系之间的语义信息，提高信息抽取的准确性。下面是关于联合式事件抽取的一些相关知识。

(1) 模型架构：联合式事件抽取通常采用深度学习模型，如双向长短期记忆网络(BiLSTM)或转换器模型(如 BERT)。这些模型可以同时处理文本中的实体和关系，并在端到端的框架中进行联合训练。

(2) 数据标注：联合式事件抽取需要标注文本中的实体和它们之间的关系。标注数据通常采用 BIO(即 Begin-Inside-Outside)标记法，其中 B 表示实体的起始位置，I 表示实体的内部位置，O 表示实体之外的位置。关系标注则可以使用实体对的方式进行标注，即标注出实体对之间的关系类型。

(3) 损失函数：联合式事件抽取通常采用多任务学习的方式进行训练，因此需要设计合适的损失函数来同时考虑实体抽取和关系抽取任务。常用的损失函数包括交叉熵损失函数和序列标注损失函数。

(4) 评估指标：评估联合式事件抽取模型的性能通常使用实体级别和关系级别的评估指标，如准确率、召回率和 F1 值等。同时，也可以采用端到端的评估指标来评估整个事件抽取系统的性能。

基于序列标注的联合模型 BiLSTM-CRF 通过双向长短期记忆网络(BiLSTM)捕捉文本的前向和后向依赖关系，理解事件触发词和事件元素的上下文信息，随后利用条件随机场对 BiLSTM 的输出进行序列标注，从而联合识别事件触发词和事件元素。这种方法能够有效捕捉上下文依赖和标签依赖，提高标注的准确性，适合处理序列标注任务。

联合式事件抽取的优势在于能够充分利用事件触发词和事件元素之间的语义信息和依赖关系，从而提高了事件抽取的准确性和效率。相较于管道式模型，联合式模型避免了级联错误，并且更好地捕捉了事件触发词与事件元素之间的关联性，因此在事件抽取任务中具有更好的性能和效果。

接下来以简单的联合式事件抽取为例，使用 BiLSTM-CRF 模型来进行实体抽取和关系抽取，然后从文本中抽取出事件信息。读者可以根据自己的需求和具体场景进一步扩展和优化这段代码。

```
1.    import numpy as np
2.    import torch
3.    import torch.nn as nn
4.    import torch.optim as optim
5.    from sklearn.preprocessing import LabelEncoder
```

```
from torch.utils.data import Dataset, DataLoader
import torch.nn.functional as F

#假设有一些文本数据和对应的实体及关系标签
texts = ['Elon Musk is the CEO of Tesla, Inc., a company based in California.','Bill Gates founded Microsoft Corporation in 1975.']
entity_labels = [[('Elon Musk', 'Person'), ('Tesla, Inc.', 'Organization'), ('California', 'Location')], [('Bill Gates', 'Person'), ('Microsoft Corporation', 'Organization'), ('1975', 'Date')]]
relation_labels = [[('Elon Musk', 'CEO', 'Tesla, Inc.'), ('Tesla, Inc.', 'based in', 'California')], [('Bill Gates', 'founder', 'Microsoft Corporation')]]

#构建词汇表
vocab = set()
for text in texts:
    words = text.lower().split()
    vocab.update(words)
word_index = {word: index + 1 for index, word in enumerate(vocab)}
word_index['<PAD>'] = 0
#将文本转换为索引序列
sequences = [[word_index[word] for word in text.lower().split()] for text in texts]
max_length = max(len(seq) for seq in sequences)
sequences_padded = torch.tensor([seq +[0] * (max_length - len(seq)) for seq in sequences])

#构建实体和关系标签的编码字典
entity_label_encoder = LabelEncoder()
relation_label_encoder = LabelEncoder()
flat_entity_labels = [label for sublist in entity_labels for label in sublist]
entity_label_encoder.fit([label[1] for label in flat_entity_labels] + ['<PAD>'])
flat_relation_labels = [label for sublist in relation_labels for label in sublist]
relation_label_encoder.fit([label[1] for label in flat_relation_labels] + ['<PAD>'])
#将实体和关系标签转换为索引序列
def encode_labels(labels, label_encoder, max_length):
    encoded_labels = []
    for label_set in labels:
        label_indices = [label_encoder.transform([label[1]])[0] for label in label_set]
        encoded_labels.append(label_indices + [0] * (max_length - len(label_indices)))
    return torch.tensor(encoded_labels)
entity_labels_encoded = encode_labels(entity_labels, entity_label_encoder, max_length)
relation_labels_encoded = encode_labels(relation_labels, relation_label_encoder, max_length)

#创建数据集和数据加载器
class EventDataset(Dataset):
    def __init__(self, sequences, entity_labels, relation_labels):
        self.sequences = sequences
        self.entity_labels = entity_labels
        self.relation_labels = relation_labels
    def __len__(self):
        return len(self.sequences)
    def __getitem__(self, idx):
        return self.sequences[idx], self.entity_labels[idx], self.relation_labels[idx]
dataset = EventDataset(sequences_padded, entity_labels_encoded, relation_labels_encoded)
train_loader = DataLoader(dataset, batch_size=1, shuffle=True)

```

```
#定义 BiLSTM 模型
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, entity_tagset_size,
    relation_tagset_size):
        super(BiLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.bilstm = nn.LSTM(embedding_dim, hidden_dim, bidirectional = True, batch_
        first = True)
        self.hidden2entity = nn.Linear(hidden_dim * 2, entity_tagset_size)
        self.hidden2relation = nn.Linear(hidden_dim * 2, relation_tagset_size)
    def forward(self, x):
        embeds = self.embedding(x)
        lstm_out, _ = self.bilstm(embeds)
        entity_scores = self.hidden2entity(lstm_out)
        relation_scores = self.hidden2relation(lstm_out)
        return entity_scores, relation_scores

#初始化模型
embedding_dim = 128
hidden_dim = 64
entity_tagset_size = len(entity_label_encoder.classes_)
relation_tagset_size = len(relation_label_encoder.classes_)
vocab_size = len(word_index)
model = BiLSTM(vocab_size, embedding_dim, hidden_dim,
entity_tagset_size, relation_tagset_size)
optimizer = optim.Adam(model.parameters(), lr = 0.001)
criterion = nn.CrossEntropyLoss()

#训练模型
n_epochs = 10
for epoch in range(n_epochs):
    model.train()
    total_loss = 0
    for sequences, entity_labels, relation_labels in train_loader:
        model.zero_grad()
        entity_scores, relation_scores = model(sequences)
        entity_scores = entity_scores.view(-1, entity_tagset_size)
        relation_scores = relation_scores.view(-1, relation_tagset_size)
        entity_labels = entity_labels.view(-1)
        relation_labels = relation_labels.view(-1)
        entity_loss = criterion(entity_scores, entity_labels)
        relation_loss = criterion(relation_scores, relation_labels)
        loss = entity_loss + relation_loss
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f'Epoch {epoch + 1}, Loss: {total_loss/len(train_loader)}') epochs = 10, batch_
    size = 1, verbose = 1)
```

该模型首先准备文本数据及其对应的实体和关系标签，然后构建词汇表并将文本转换为索引序列。接着，定义一个 BiLSTM 模型，包括嵌入层和 LSTM 层，以处理输入序列并输出实体和关系得分。在训练阶段，通过计算损失并优化模型参数，模型逐步学习从文本中提取事件信息。每个训练周期结束时，打印出平均损失以监控训练进度。

模型的输入包括文本数据、实体标签和关系标签，文本数据为 texts = ['Elon Musk is

the CEO of Tesla, Inc., a company based in California.','Bill Gates founded Microsoft Corporation in 1975.']。

实体标签为 entity_labels = [[('Elon Musk', 'Person'), ('Tesla, Inc.', 'Organization'), ('California', 'Location')], [('Bill Gates', 'Person'), ('Microsoft Corporation', 'Organization'), ('1975', 'Date')]]。

```
Epoch 1, Loss: 3.0208131074905396
Epoch 2, Loss: 2.7946279048919678
Epoch 3, Loss: 2.5545220375061035
Epoch 4, Loss: 2.3118282556533813
Epoch 5, Loss: 2.103016436100006
Epoch 6, Loss: 1.908055067062378
Epoch 7, Loss: 1.6878281235694885
Epoch 8, Loss: 1.4853694438934326
Epoch 9, Loss: 1.2907716631889343
Epoch 10, Loss: 1.1000929474830627
```

图 3.20 管道式事件抽取示例代码运行结果

关系标签为 relation_labels = [[('Elon Musk', 'CEO', 'Tesla, Inc.'), ('Tesla, Inc.', 'based in', 'California')], [('Bill Gates', 'founder', 'Microsoft Corporation')]]。

模型输出了训练过程中的损失值,每个训练 epoch 的损失值显示了模型在逐渐学习和改进。损失值从 3.02 下降到 1.10,显示了模型在训练过程中的改进和学习能力,如图 3.20 所示。


视频讲解

▶ 3.4.4 篇章级事件抽取

篇章级事件抽取是指在文本的篇章级别上,从一段或多段连续的文本中抽取出事件信息的过程。与句子级或段落级的事件抽取不同,篇章级事件抽取需要考虑更广泛的语境和跨文本的信息,以理解和抽取事件的全貌和关联关系。这一任务需要处理长距离依赖和复杂的上下文关系,以捕捉文本中跨句甚至跨段落的事件关联。文档级事件抽取包括两部分:旨在从文档中发现事件提及的关键事件检测和用于填充事件论元的论元补足。

篇章级事件抽取是从一篇篇章级别的文本中提取出事件信息的过程。在这个过程中,需要识别出文本中的实体和它们之间的关系,并将它们组织成事件的形式。

1. 关键事件检测

关键事件检测的目标是从文档中发现和识别事件提及。这一过程需要综合考虑整篇文档的语境信息,识别出哪些段落或句子中包含了重要的事件信息。这通常利用高级的自然语言处理技术,如深度学习和图神经网络,来处理文本的长距离依赖关系和复杂的上下文信息。

示例:在 2011 年的一次重大地震中,位于日本的城市东京受到了严重影响。地震发生后,许多建筑物倒塌,造成了大量人员伤亡。救援人员迅速展开了紧急救援行动,许多国家也向日本提供了援助。

在上述文本中,关键事件检测需要识别“地震”“救援行动”和“国际援助”作为关键事件。

2. 论元补足

论元补足旨在识别和填充事件的各种论元(如时间、地点、参与者等),从而构建完整的事件结构。在篇章级事件抽取任务中,论元补足需要处理跨句、跨段的依赖关系,以捕捉事件及其论元之间的复杂关联。

论元补足的目的是在识别出关键事件后,进一步填充这些事件的论元(如时间、地点、参与者等)。这一过程需要精细地解析文本,识别出与事件相关的实体及其关系,并将它们准确地组织成事件的论元结构,如:

事件:地震
时间:2011 年
地点:日本东京
影响:建筑物倒塌,大量人员伤亡

事件：救援行动
参与者：救援人员
行动：紧急救援

事件：国际援助
参与者：许多国家
行动：提供援助

下面是一个简单的篇章级事件抽取的示例代码，其中使用 BiLSTM-CRF 模型进行实体识别和关系抽取：

```
import numpy as np
import tensorflow as tf
from keras.models import Model
from keras.layers import Input, LSTM, Embedding, Dense,
TimeDistributed, Bidirectional

#假设有一些篇章级别的文本数据
texts =[
    'Elon Musk is the CEO of Tesla, Inc., a company based in California.',
    'Bill Gates founded Microsoft Corporation in 1975.'
]

#构建词汇表
vocab = set()
for text in texts:
    words = text.lower().split()
    vocab.update(words)
word_index = {word: index + 1 for index, word in enumerate(vocab)}
word_index['<PAD>'] = 0
#将文本转换为索引序列
sequences = [[word_index[word] for word in text.lower().split()] for text in texts]
#填充序列
max_length = max(len(seq) for seq in sequences)
sequences_padded =
tf.keras.preprocessing.sequence.pad_sequences(sequences, maxlen=max_length)
#构建实体标签
entity_labels =[
    [('Elon Musk', 'Person'), ('Tesla, Inc.', 'Organization'), ('California', 'Location')],
    [('Bill Gates', 'Person'), ('Microsoft Corporation', 'Organization'), ('1975', 'Date')]
]
#构建关系标签
relation_labels =[
    [('Elon Musk', 'CEO', 'Tesla, Inc.'), ('Tesla, Inc.', 'based in', 'California')], [('Bill
    Gates', 'founder', 'Microsoft Corporation')]
]
#构建实体和关系标签的编码字典
entity_labels_index = {'<PAD>': 0, 'Person': 1, 'Organization': 2, 'Location': 3, 'Date': 4}
relation_labels_index = {'<PAD>': 0, 'CEO': 1, 'founder': 2, 'based in': 3}
#将实体标签转换为数字索引，并填充
def pad_entity_labels(labels, max_length, label_index):
```

```
    padded_labels = []
    for label_sequence in labels:
        sequence = [label_index.get(label, 0) for _, label in label_sequence]
                                                    #只考虑实体标签
        padded_labels.append(sequence + [0] * (max_length - len(sequence)))
                                                    #填充到 max_length
    return np.array(padded_labels)
entity_labels_padded = pad_entity_labels(entity_labels, max_length, entity_labels_index)
#将关系标签转换为数字索引,并填充
def pad_relation_labels(labels, max_length, label_index):
    padded_labels = []
    for label_sequence in labels:
        #这里只考虑关系标签,忽略实体部分
        sequence = [label_index.get(relation, 0) for _, relation, _ in label_sequence]
        padded_labels.append(sequence + [0] * (max_length - len(sequence)))
                                                    #填充到 max_length
    return np.array(padded_labels)
relation_labels_padded = pad_relation_labels(relation_labels, max_length, relation_
labels_index)

#定义模型
input_layer = Input(shape=(max_length,))
embedding_layer = Embedding(len(word_index), 128, input_length=max_length)(input_
layer)
bilstm_layer = Bidirectional(LSTM(64, return_sequences=True))(embedding_layer)
entity_output_layer = TimeDistributed(Dense(len(entity_labels_index), activation='softmax'))
(bilstm_layer)
relation_output_layer = TimeDistributed(Dense(len(relation_labels_index), activation='softmax'))
(bilstm_layer)

#实体抽取模型
entity_model = Model(inputs=input_layer, outputs=entity_output_layer)
entity_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=
['accuracy'])

#训练实体抽取模型
entity_model.fit(sequences_padded, entity_labels_padded, epochs=10, batch_size=1,
verbose=1)

#关系抽取模型
relation_model = Model(inputs=input_layer, outputs=relation_output_layer)
relation_model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=
['accuracy'])

#训练关系抽取模型
relation_model.fit(sequences_padded, relation_labels_padded, epochs=10, batch_size=1,
verbose=1)
```

在这个篇章级事件抽取的示例代码中,首先通过对文本数据构建词汇表并将文本转换为索引序列;然后使用填充方法处理不同长度的序列;接着定义了实体和关系标签的编码字典,并将标签转换为数字索引,确保它们的长度一致;最后构建了一个 BiLSTM 模型,其中包含嵌入层、双向 LSTM 层和时间分布层,用于输出实体和关系标签。模型分别编译和训练,以识别文本中的实体和关系,并输出训练结果。

sequences_padded 是模型的输入数据，是对原始文本进行词汇索引转换和填充后的结果。每个文本被表示为一个固定长度的序列，其中包含了词汇表中每个词的索引。(batch_size，max_length)是输入的形状，其中 batch_size 是每个训练批次中的样本数量，max_length 是填充后的序列长度。

对关系抽取模型 Epoch 1/10 到 Epoch 10/10：训练损失（Loss)第一个周期的损失值是 1.2531，逐渐减少到 0.3560。这表明关系抽取模型在训练过程中也逐渐改进了对关系标签的预测。准确率（Accuracy)从 0.5000 开始，逐渐提高到 0.8846。准确率的提高表明关系抽取模型在训练过程中对关系标签的预测逐渐变得更准确，如图 3.21 所示。

```
Epoch 1/10
2/2 [==============================] - 2s 6ms/step - loss: 1.2531 - accuracy: 0.5000
Epoch 2/10
2/2 [==============================] - 0s 5ms/step - loss: 1.0540 - accuracy: 0.8846
Epoch 3/10
2/2 [==============================] - 0s 7ms/step - loss: 0.8699 - accuracy: 0.8846
Epoch 4/10
2/2 [==============================] - 0s 6ms/step - loss: 0.7056 - accuracy: 0.8846
Epoch 5/10
2/2 [==============================] - 0s 6ms/step - loss: 0.5714 - accuracy: 0.8846
Epoch 6/10
2/2 [==============================] - 0s 6ms/step - loss: 0.4756 - accuracy: 0.8846
Epoch 7/10
2/2 [==============================] - 0s 6ms/step - loss: 0.4185 - accuracy: 0.8846
Epoch 8/10
2/2 [==============================] - 0s 6ms/step - loss: 0.3888 - accuracy: 0.8846
Epoch 9/10
2/2 [==============================] - 0s 5ms/step - loss: 0.3726 - accuracy: 0.8846
Epoch 10/10
2/2 [==============================] - 0s 6ms/step - loss: 0.3560 - accuracy: 0.8846
```

图 3.21　BiLSTM-CRF 模型示例代码运行结果

▶ 3.4.5　事件抽取面临的挑战和机遇

事件抽取作为信息抽取中的重要任务之一，能够检测句子提到的某些事件，也可以对事件类型进行分类与识别事件论元。它为知识图谱、推荐系统、信息检索等任务提供基础的数据支持。同时，事件抽取在语义分析、篇章理解、自动问答等领域也具有重要意义。然而，人工智能的发展对事件抽取的准确性提出了更高的要求，深度学习虽已成为事件抽取的关键技术，但在领域自适应性和召回率方面仍有很大的提升空间。

事件抽取发展至今，在研究内容上逐渐由单一领域向多领域、跨领域发展，事件类型的定义方式表示为由人工预先定义转变为事件类型自动发现、挖掘；在研究方法上，深度学习的方法在事件抽取任务的性能上表现良好，基于深度学习的框架日益成为主流，在此基础上结合远程监督、强化学习、少样本学习和零资源学习的思想等可以为事件抽取性能的提升提供新思路。特别是中文事件抽取的任务复杂，将深度学习的方法应用在其中是今后需要努力的一个方向。

3.5　知识抽取实践

▶ 3.5.1　实践环境

本项目利用 PaddleNLP 的 UIE 模型，实践应用于实体抽取任务，具体处理打车数据和快

递单数据。通过这两个不同领域的数据集,项目展示了如何使用 UIE 模型进行命名实体识别(NER),实现对文本中关键实体(如地址、姓名、时间、费用等)的自动识别和标注,从而验证了 UIE 模型在实际场景中的有效性和应用价值。实践环境如下。

(1) 硬件配置。

计算机:Legion Y7000 2020。

CPU:Intel Core i5-10200H。

内存:16GB。

GPU:NVIDIA GeForce GTX 1650。

(2) 软件配置。

操作系统:Windows 10 (64 位)。

Python 版本:3.7.4。

深度学习框架:PaddlePaddle 2.3.2。

自然语言处理库:PaddleNLP 2.4.0。

可视化工具:VisualDL 2.2.0。

环境配置确保了模型训练和评估的高效性,同时支持可视化结果的展示。

▶ 3.5.2 数据集

数据集被划分为两部分:训练集和测试集。训练集用来训练、构建模型,测试集用来评估预测结果。

(1) 快递单数据集(waybill.jsonl):包含一系列快递单信息,每条记录包括一个唯一的 ID、文本内容及其标注的实体信息。

实体信息如下。

- 姓名:收件人姓名。
- 省份:收件人所在省份。
- 城市:收件人所在城市。
- 县区:收件人所在县区。
- 详细地址:收件人具体地址。
- 电话:收件人联系电话。

示例:

```
{
  "id": 57,
  "text": "昌胜远黑龙江省哈尔滨市南岗区宽桥街 *** 186 - **** - 1296",
  "relations": [],
  "entities": [
    {"id": 111, "start_offset": 0, "end_offset": 3, "label": "姓名"},
    {"id": 112, "start_offset": 3, "end_offset": 7, "label": "省份"},
    {"id": 113, "start_offset": 7, "end_offset": 11, "label": "城市"},
    {"id": 114, "start_offset": 11, "end_offset": 14, "label": "县区"},
    {"id": 115, "start_offset": 14, "end_offset": 20, "label": "详细地址"},
    {"id": 116, "start_offset": 20, "end_offset": 31, "label": "电话"}
  ]
}
```

(2) 打车数据集(doccano_ext.jsonl):记录了用户打车的相关信息,包括时间、出发地、目

的地和费用等，每条记录也包含一个唯一的 ID、文本内容及其标注的实体信息。

实体信息如下。

- 时间：打车的时间。
- 出发地：出发地点。
- 目的地：目的地。
- 费用：打车费用。

示例：

```
{
  "id": 1,
  "text": "昨天晚上十点加班打车回家 58 元",
  "relations": [],
  "entities": [
    {"id": 0, "start_offset": 0, "end_offset": 6, "label": "时间"},
    {"id": 1, "start_offset": 11, "end_offset": 12, "label": "目的地"},
    {"id": 2, "start_offset": 12, "end_offset": 14, "label": "费用"}
  ]}
```

(3) 测试集：采用以下 5 个文本数据作为测试集。

{"城市内交通费 7 月 5 日金额 180 元广州至佛山", "从百度大厦到龙泽苑东区打车费 45 元", "上海虹桥高铁到杭州时间是 9 月 24 日费用是 73 元", "上周末坐动车从北京到上海花费 598 元", "昨天北京飞上海花费 600 元"}。

▶ 3.5.3　操作步骤及运行结果

1. 数据集加载

在本次实践中，使用了两个数据集：快递单数据和打车数据。这些数据集包含了各种实体，如出发地、目的地、时间等。

```
1.    # ! wget https://paddlenlp.bj.bcebos.com/model_zoo/uie/waybill.jsonl
2.    # ! mv waybill.jsonl ./data/
3.    # ! mv doccano_ext.jsonl ./data/
4.    # ! mv dev_test.jsonl ./data/
```

为实现数据集的高效加载和配置管理，实践需要设置一些关键参数，这些参数在不同任务类型(例如抽取和分类)中具有重要作用。以下是主要参数的简要说明。

doccano_file：从 doccano 导出的数据标注文件。

save_dir：训练数据的保存目录，默认存储在 data 目录下。

negative_ratio：最大负例比例，该参数只对抽取类型任务有效，适当构造负例可提升模型效果。负例数量和实际的标签数量有关，最大负例数量 = negative_ratio * 标签数量。该参数只对训练集有效，默认为 5。为了保证评估指标的准确性，验证集和测试集默认构造全负例。

splits：划分数据集时训练集、验证集所占的比例。默认为[0.8, 0.1, 0.1]表示按照 8∶1∶1 的比例将数据划分为训练集、验证集和测试集。

task_type：选择任务类型，可选有抽取和分类两种类型的任务。

options：指定分类任务的类别标签，该参数只对分类类型任务有效。默认为["正向", "负向"]。

prompt_prefix：声明分类任务的 prompt 前缀信息，该参数只对分类类型任务有效。默认为"情感倾向"。

is_shuffle：是否对数据集进行随机打散，默认为 True。

Seed：随机种子，默认为 1000。

Separator：实体类别/评价维度与分类标签的分隔符，该参数只对实体/评价维度级分类任务有效，默认为“＃＃”。

下面这段名为 doccano.py 的 Python 脚本，通过命令行参数设置了多项配置，用于处理 Doccano 导出的标注数据。

```
!python doccano.py \
    --doccano_file ./data/doccano_ext.jsonl \
    --task_type 'ext' \
    --save_dir ./data \
    --splits 0.8 0.1 0.1 \
    --negative_ratio 5
```

输出部分展示如图 3.22 所示。

输出部分展示:

{"content": "上海到北京机票1320元", "result_list": [{"text": "上海", "start": 0, "end": 2}], "prompt": "出发地"}
{"content": "上海到北京机票1320元", "result_list": [{"text": "北京", "start": 3, "end": 5}], "prompt": "目的地"}
{"content": "上海到北京机票1320元", "result_list": [{"text": "1320", "start": 7, "end": 11}], "prompt": "费用"}
{"content": "上海虹桥到杭州东站高铁g7555共73元时间是10月14日", "result_list": [{"text": "上海虹桥", "start": 0, "end": 4}], "prompt": "出发地"}
{"content": "上海虹桥到杭州东站高铁g7555共73元时间是10月14日", "result_list": [{"text": "杭州东站", "start": 5, "end": 9}], "prompt": "目的地"}
{"content": "上海虹桥到杭州东站高铁g7555共73元时间是10月14日", "result_list": [{"text": "73", "start": 17, "end": 19}], "prompt": "费用"}
{"content": "上海虹桥到杭州东站高铁g7555共73元时间是10月14日", "result_list": [{"text": "10月14日", "start": 23, "end": 29}], "prompt": "时间"}
{"content": "昨天晚上十点加班打车回家58元", "result_list": [{"text": "昨天晚上十点", "start": 0, "end": 6}], "prompt": "时间"}
{"content": "昨天晚上十点加班打车回家58元", "result_list": [{"text": "家", "start": 11, "end": 12}], "prompt": "目的地"}
{"content": "昨天晚上十点加班打车回家58元", "result_list": [{"text": "58", "start": 12, "end": 14}], "prompt": "费用"}
{"content": "2月20号从南山到光明二十元", "result_list": [{"text": "2月20号", "start": 0, "end": 5}], "prompt": "时间"}

图 3.22　训练集输出部分展示

2. 模型训练

使用 PaddleNLP 的 UIE 模型，进行实体抽取模型的训练。训练前需要对数据进行预处理，包括文本清洗和标注。然后，初始化模型，设置训练参数，开始训练过程。

```
!python finetune.py \
    --train_path "./data/train.txt" \
    --dev_path "./data/dev.txt" \
    --save_dir "./checkpoint" \
    --learning_rate 1e-5 \
    --batch_size 8 \
    --max_seq_len 512 \
    --num_epochs 100 \
    --model "uie-base" \
    --seed 1000 \
    --logging_steps 10 \
    --valid_steps 50 \
    --device "gpu"
```

读者在使用时，推荐使用 GPU 环境，否则可能会出现内存溢出的错误。CPU 环境下，建议将 model 修改为 uie-tiny，并适当调整 batch_size。可通过将--num_epochs 设置为一个大一些的数值，进行多轮次训练，以提高准确率。可配置参数说明如下。

train_path：训练集文件路径。

dev_path：验证集文件路径。

save_dir：模型存储路径，默认为./checkpoint。

learning_rate：学习率，默认为 1e-5。

batch_size：批处理大小。结合显存情况进行调整，若出现显存不足时，请适当调低这个参数，默认为 16。

max_seq_len：文本最大切分长度，输入超过最大长度时会对输入文本进行自动切分，默认为 512。

num_epochs：训练轮次，默认为 100。

model 选择模型，程序会基于选择的模型进行模型微调，可选值有 uie-base 和 uie-tiny，默认为 uie-base。

seed：随机种子，默认为 1000。

logging_steps：日志打印的间隔 steps 数，默认为 10。

valid_steps：evaluate 的间隔 steps 数，默认为 100。

device：选用什么设备进行训练，可选 cpu 或 gpu。

部分训练效果展示(见图 3.23)：具体输出已折叠。

```
[2022-07-12 15:09:47,643] [    INFO] - global step 250, epoch: 13, loss: 0.00045, speed: 3.90 step/s
[2022-07-12 15:09:47,910] [    INFO] - Evaluation precision: 1.00000, recall: 1.00000, F1: 1.00000
[2022-07-12 15:09:50,399] [    INFO] - global step 260, epoch: 13, loss: 0.00043, speed: 4.02 step/s
[2022-07-12 15:09:52,966] [    INFO] - global step 270, epoch: 14, loss: 0.00042, speed: 3.90 step/s
[2022-07-12 15:09:55,464] [    INFO] - global step 280, epoch: 14, loss: 0.00040, speed: 4.00 step/s
[2022-07-12 15:09:58,028] [    INFO] - global step 290, epoch: 15, loss: 0.00039, speed: 3.90 step/s
[2022-07-12 15:10:00,516] [    INFO] - global step 300, epoch: 15, loss: 0.00038, speed: 4.02 step/s
[2022-07-12 15:10:00,781] [    INFO] - Evaluation precision: 1.00000, recall: 1.00000, F1: 1.00000
[2022-07-12 15:10:03,348] [    INFO] - global step 310, epoch: 16, loss: 0.00036, speed: 3.90 step/s
[2022-07-12 15:10:05,836] [    INFO] - global step 320, epoch: 16, loss: 0.00035, speed: 4.02 step/s
[2022-07-12 15:10:08,393] [    INFO] - global step 330, epoch: 17, loss: 0.00034, speed: 3.91 step/s
[2022-07-12 15:10:10,888] [    INFO] - global step 340, epoch: 17, loss: 0.00033, speed: 4.01 step/s
```

图 3.23　训练效果展示

3. 模型评估

模型训练完成后，使用验证集对模型进行评估。评估指标主要包括准确率和召回率，这些指标帮助读者了解模型的性能以及在实际应用中的表现。

```
!python evaluate.py \
    --model_path ./checkpoint/model_best \
    --test_path ./data/test.txt \
    --batch_size 16 \
    --max_seq_len 512
```

下面对代码中的参数进行逐一说明。

model_path：进行评估的模型文件夹路径，路径下需包含模型权重文件 model_state.pdparams 及配置文件 model_config.json。

test_path：进行评估的测试集文件。

batch_size：批处理大小，请结合机器情况进行调整，默认为 16。

max_seq_len：文本最大切分长度，输入超过最大长度时会对输入文本进行自动切分，默认为 512。

```
!python evaluate.py \
    --model_path ./checkpoint/model_best \
    --test_path ./data/test.txt \
    --debug
```

下面对代码中的参数进行逐一说明。

model：选择所使用的模型，可选的模型包括 uie-base、uie-medium、uie-mini、uie-micro 和 uie-nano，默认为 uie-base。

debug：是否开启 debug 模式对每个正例类别分别进行评估，该模式仅用于模型调试，默认关闭。

4. 结果预测

使用训练好的模型对新数据进行预测。下面将通过示例文本展示模型对不同实体的识别能力，并输出实体。

```
from pprint import pprint
import json
from paddlenlp import Taskflow
def openreadtxt(file_name):
    data = []
    file = open(file_name,'r',encoding = 'UTF - 8')   #打开文件
    file_data = file.readlines()                      #读取所有行
    for row in file_data:
        data.append(row)                              #将每行数据插入 data 中
    return data
data_input = openreadtxt('./input/nlp.txt')
schema = ['出发地', '目的地','时间']
few_ie = Taskflow('information_extraction', schema = schema, batch_size = 1,task_path =
'./checkpoint/model_best')
results = few_ie(data_input)
with open("./output/test.txt", "w + ",encoding = 'UTF - 8') as f:
#a :写入文件,若文件不存在则会先创建再写入,但不会覆盖原文件,而是追加在文件末尾
    for result in results:
        line = json.dumps(result, ensure_ascii = False)   #对中文默认使用的 ascii 编码.
                                                          #想输出真正的中文需要指定
                                                          #ensure_ascii = False
        f.write(line + "\n")
print("数据结果已导出")
```

输入文件展示如图 3.24 所示。

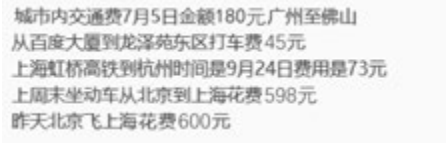


图 3.24　输入文件展示

输出文件展示如图 3.25 所示。

{"出发地": [{"text": "广州", "start": 15, "end": 17, "probability": 0.9073772252165782}], "目的地": [{"text": "佛山", "start": 18, "end": 20, "probability": 0.9927365183877761}], "时间": [{"text": "7月5日", "start": 6, "end": 10, "probability": 0.9978010396512218}]}
{"出发地": [{"text": "百度大厦", "start": 1, "end": 5, "probability": 0.9688251474094072}], "目的地": [{"text": "龙泽苑东区", "start": 6, "end": 11, "probability": 0.9877913072493669}]}
{"目的地": [{"text": "杭州", "start": 7, "end": 9, "probability": 0.9929172180094881}], "时间": [{"text": "9月24日", "start": 12, "end": 17, "probability": 0.9953342057701597}]}
{"出发地": [{"text": "北京", "start": 7, "end": 9, "probability": 0.9730483667174711}], "目的地": [{"text": "上海", "start": 10, "end": 12, "probability": 0.9884861303093971}], "时间": [{"text": "上周末", "start": 0, "end": 3, "probability": 0.9977407699595275}]}
{"出发地": [{"text": "北京", "start": 2, "end": 4, "probability": 0.9741889533535561}], "目的地": [{"text": "上海", "start": 5, "end": 7, "probability": 0.9928200521486445}], "时间": [{"text": "昨天", "start": 0, "end": 2, "probability": 0.9731559534465504}]}

图 3.25　输出文件展示

5. 结果说明

在实践中，使用了 PaddleNLP 的 UIE 模型对“出发地”“目的地”以及“时间”这三个实体进行了抽取任务，并成功从输入文本中提取了相关信息。输出结果中展示了每条记录中识别的实体信息，包括文本片段、起始位置、结束位置，以及模型对该实体识别的置信度概率。模型对大部分数据都能够准确地进行信息抽取，并提供较高的置信度。

下面以输入文本“城市内交通费 7 月 5 日金额 114 广州至佛山”为例进行详细介绍。

出发地：模型识别出“广州”，在原文中位于第 15～17 个字符位置。模型对这一识别的置信度为 0.91，表明模型对其判断有较高的自信。

目的地：识别出的目的地是“佛山”，位于第 18～20 个字符，置信度为 0.99，进一步说明模型对该识别的准确性非常确信。

时间：时间信息被识别为“7 月 5 日”，位于第 6～10 个字符，置信度为 0.99。这是模型认为高度准确的信息，说明模型对这种常见的时间格式有很强的识别能力。

通过实践结果可以看出，UIE 模型对这类自然语言表达中的关键信息具备出色的识别能力，尤其是对地名和日期等标准化信息的提取，其置信度基本维持在 0.9 以上。这种高置信度的输出，说明该模型在中文环境下的实体抽取任务中可以实现可靠的精确识别。

3.6　本章小结

本章深入探讨了知识抽取的主要阶段及其核心内容。知识抽取作为信息处理和人工智能领域的重要技术，旨在从海量数据中抽取有价值的实体、关系和事件，为后续的进一步探讨知识融合技术，讨论如何将来自不同来源、不同格式的知识进行整合，形成统一且一致的知识体系。

本章详细阐述了实体抽取阶段中的命名实体识别和实体扩展技术，从文本中抽取出具有特定意义的实体，为后续的关系抽取和事件抽取提供了必要的实体信息；聚焦关系抽取阶段。通过模式匹配、基于规则的方法和基于机器学习的方法，成功地从文本中抽取出了实体之间的语义关系，进一步丰富了知识的表示。还介绍了事件抽取阶段，通过识别事件触发词、抽取事件论元以及进行事件结构化，能够将文本中的事件信息以结构化的形式表示出来，为后续的决策分析和推理提供了有力的支持。

习题 3

在线测试

一、 选择题

1. 知识抽取的主要目的是(　　)。

A. 从文本中识别特定格式的数据　　B. 从非结构化数据中提取结构化信息
C. 对数据进行简单的清洗和整理　　D. 提高文本的可读性

2. 在知识抽取中,命名实体识别通常用于识别(　　)类型的实体。
A. 人名、地名、机构名　　B. 动词、形容词、名词
C. 句子、段落、篇章　　D. 数字、符号、字母

3. (　　)不是知识抽取的常用方法。
A. 基于规则的方法　　B. 基于模板的方法
C. 基于随机猜测的方法　　D. 基于机器学习的方法

4. 在关系抽取中,通常使用(　　)技术来识别实体之间的关系。
A. 情感分析　　B. 文本分类　　C. 语义角色标注　　D. 文本生成

5. 知识抽取中的事件抽取主要关注(　　)。
A. 文本中的时间信息　　B. 文本中的事件及其相关论元
C. 文本中的情感倾向　　D. 文本中的主题分类

6. 知识抽取的效果评估通常使用(　　)指标。
A. 准确率　　B. 召回率　　C. F1 值　　D. 文本长度

二、 简答题

1. 简述知识抽取的基本定义及其在信息处理中的作用。
2. 列举几种常见的数据预处理操作,并解释它们在知识抽取中的重要性。
3. 什么是命名实体识别? 其在知识抽取中有什么应用?
4. 简述基于规则的关系抽取方法的基本思路及其优缺点。
5. 在知识抽取中,为什么需要进行后处理与评估? 列举几种常见的评估指标。
6. 简述事件抽取的主要任务及其在实现过程中的关键步骤。