一、背景知识

1.Neo4j图数据库

Neo4j 是一个图形数据库，就像传统的关系数据库中的 Oracel 和 MySQL一样，用来持久化数据。Neo4j 是最近几年发展起来的新技术，属于 NoSQL 数据库中的一种。

知识图谱是一种基于图的数据结构，由节点和边组成。其中节点即实体，由一个全局唯一的 ID 标示，关系（也称属性）用于连接两个节点。通俗地讲，知识图谱就是把所有不同种类的信息连接在一起而得到一个关系网络，提供了从“关系”的角度去分析问题的能力。

而 Neo4j 作为一种经过特别优化的图形数据库，有以下优势：

- 数据存储：不像传统数据库整条记录来存储数据，Neo4j 以图的结构存储，可以存储图的节点、属性和边。属性、节点都是分开存储的，属性与节点的关系构成边，这将大大有助于提高数据库的性能。
- 数据读写：在 Neo4j 中，存储节点时使用了 Index-free Adjacency 技术，即每个节点都有指向其邻居节点的指针，可以让我们在时间复杂度为 O(1) 的情况下找到邻居节点。另外，按照官方的说法，在 Neo4j 中边是最重要的，是 First-class Entities，所以单独存储，更有利于在图遍历时提高速度，也可以很方便地以任何方向进行遍历。
- 资源丰富：Neo4j 作为较早的一批图形数据库之一，其文档和各种技术博客较多。
- 同类对比：Flockdb 安装过程中依赖太多，安装复杂；Orientdb，Arangodb 与 Neo4j 做对比，从易用性来说都差不多，但是从稳定性来说，neo4j 是最好的。

综合上述以及因素，Neo4j 是做知识图谱比较简单、灵活、易用的图形数据库。

2.知识图谱技术

构建一个知识图谱的完整流程主要分为以下几个步骤

2.1 知识抽取

> 知识抽取的目的是从非结构化的文本中提取出结构化的三元组数据

知识抽取主要分为：

- 实体抽取：实体抽取也称为命名实体识别，主要有基于规则和词典的方法、基于机器学习的模型预测方法（前面做过的实体抽取项目）
- 关系抽取：关系抽取主要用于判断实体之间的关联关系，主要分为
  - 限定领域关系抽取：即根据预设好的关系集合，判断给定实体是否满足某一个关系（类似分类任务，F(实体1,实体2,文本)->关系），对这种有监督的分类任务的训练方式进行细分
    - pipeline方式：先对文本中的实体进行抽取，接着再将抽取出的实体和文本作为输入进行关系抽取；
    - 联合训练方式：直接将文本中的实体和关系同时抽取出来；
  - 开放领域关系抽取：主要是基于序列标注对模型进行训练，使模型具备自动识别并提取关系的能力
- 属性抽取：实体的属性可以看作是实体与属性值之间的一种关系，因此可以将大部分属性抽取的任务转换为关系抽取的任务；

一个三元组的txt文档形式如下（知识抽取这部分的目标就是构建一个txt文档，该文档中的文本形式如图所示）

```
Jay（周杰伦制作专辑）    唱片公司        BMG/Alfa Music
Jay（周杰伦制作专辑）    发行时间        2000-11-7
Jay（周杰伦制作专辑）    获得奖项        台湾第12届金曲奖颁奖典礼最佳流行音乐演唱专辑奖
Jay（周杰伦制作专辑）    曲目数量        10
Jay（周杰伦制作专辑）    外文名称        Jay
Jay（周杰伦制作专辑）    音乐风格        流行，RAP
Jay（周杰伦制作专辑）    制作人  周杰伦
Jay（周杰伦制作专辑）    中文名称        周杰伦同名专辑
Jay（周杰伦制作专辑）    专辑歌手        周杰伦
Jay（周杰伦制作专辑）    专辑语言        国语
jay周杰伦       软件大小        1.09M
jay周杰伦       软件名称        Jay周杰伦
jay周杰伦       软件平台        Android
TheOne（周杰伦演唱会）   I S R C 9787884772247
TheOne（周杰伦演唱会）   歌手    周杰伦
TheOne（周杰伦演唱会）   介  质  CD
TheOne（周杰伦演唱会）   中文名  TheOne
TheOne演唱会    专辑歌手        周杰伦
阿爸（周杰伦、洪荣宏演唱歌曲）  MV导演  周杰伦
阿爸（周杰伦、洪荣宏演唱歌曲）  编曲    洪敬尧
阿爸（周杰伦、洪荣宏演唱歌曲）  发行时间        2010年
阿爸（周杰伦、洪荣宏演唱歌曲）  歌曲时长        04:29
```

- 图中这些三元组不能直接用于Neo4j数据库，还需要将其分为“实体-关系-实体”和“实体-属性-属性值”，因为这两种类型的三元组在建立Neo4j的时候使用的语句是不一样的；
- 直接从网上扒下来的三元组很可能面临上述需要三元组类型拆分的问题，而事实上若我们手动构建三元组，在构建的时候就对三元组的类型进行区分，则无需额外进行拆分；

2.2 知识融合

> 知识融合的目的是清洗三元组以提升数据质量

由于知识图谱中的知识来源广泛，存在知识质量良莠不齐、来自不同数据源的知识重复、知识间的关联不够明确等问题，所以必须要进行知识的融合。

知识融合主要分为：

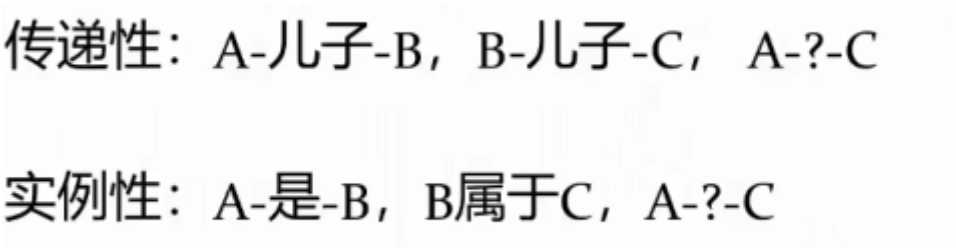
- 实体对齐：将不同来源的知识认定为真实世界中的同一实体（如“万里长城”和“长城”是同一个实体），主要做法是根据不同实体包含的属性之间的相似度来判断是否是同一实体；
- 实体消歧：将同一名称但指代不同事物的实体区分（比如“苹果”既可能是科技公司，也可能是一种水果），主要方式是结合其上下文相关信息来判断，或者根据其特定的属性来判断（“创始人”这个属性几乎就只能指代“苹果”公司）
- 属性对齐：不同数据源对实体的属性记录可能使用不同的词语（比如对“出生日期”、“生日”这些属性实际指代的是同一种属性），一般方式是使用文本相似度来计算，或者借助属性值是否相等、相似来判断；

2.3 知识推理

> 知识推理的目的是挖掘或补全数据

在已有的知识库（三元组、图谱）基础上进一步挖掘隐含的知识，从而丰富、扩展知识库。

一般方式是使用“规则+句法”，如下



当然也可以基于模型，要么是给出两个实体来推断其关系，要么是给出一个实体一个属性判断另一个实体，或者是直接给出一个三元组判断该三元组是否成立

2.4 知识表示

> 知识表示的目的是将数据进行向量化表示，方便后续应用

将知识图谱中的实体，关系，属性等转化为向量，进而利用向量间的计算关系，反映实体间的关联性。

3.知识抽取详解

3.1 开放知识抽取

> 知识图谱的构建核心，实际上就是对三元组的构建，当我们拥有处理完成的三元组后，将其放入Neo4j图数据库就可以形成基本的可视化的知识图谱。因此本节我们详细介绍如何从非结构化的文本中提取出(实体，关系,实体)三元组。

我们的原始数据形式（即未标注的暴雨洪涝灾情文本）如下，知识抽取的任务就是从正文文本中抽取出（实体，关系,实体）形式的三元组。

```
“8·12”暴雨洪涝致甘肃8市州受灾 死亡43人
2010年08月19日 18:42
中国新闻网
    中新网兰州8月19日电 （记者 刘薛梅）
甘肃省政府新闻办19日下午召开新闻发布会称，8月11日至12日，甘肃陇南、天水、定西、甘南、兰州、庆阳、临夏、平凉等8个市州的42个县市区、271个乡镇、2241个村遭遇特大暴雨洪涝灾害，致159.38万人受灾，因灾死亡43人、失踪25人、受伤339人。目前已紧急转移安置15.9万人。    甘肃省民政厅副厅长徐亚荣说，据初步统计，“8.12”特大暴雨洪涝灾害造成8个市州的农作物受灾面积7.71万公顷，其中成灾5.99万公顷、绝收1.15万公顷，毁坏耕地0.66万公顷。倒塌房屋1.93万间、损坏房屋5.43万间。直接经济损失42.46亿元。  目前，各地灾区正在进一步核实灾情。完
```

一般的能够很好的建模及其应用的知识抽取都是限定知识抽取，原因也很简单，对于任何一个抽取任务(实体识别ner，实体关系抽取re以及事件抽取ee)，其问题的确定性越高则其优化目标越可能被明确。

在限定领域下的实体识别ner，实体关系抽取任务re，以及事件抽取任务ee，都是在预定义schema的范畴下进行的，先定义好实体类型，实体关系/属性类型，以及事件要素，然后采用基于规则、基于模型的方法来进行训练预测。然而这个过程不够灵活，并且定义规范的schema并不是一件容易的事情。因此越来越多的场景会需要我们针对给定的随意文本，抽取出其中的知识元组，也就是开放知识抽取。

开放抽取最大的价值在于海量、起量快，在没有约束的情况下，可以快速生成大量有意义的知识，但没有约束也就成为这种方法在后期处理上较难问题的根源（如果按照限定schema进行抽取，则可以直接使用抽取得到的结果）。

开放知识抽取领域有以OpenIE为代表的多个系统，从技术发展的脉络来看，主要包括基于规则(无监督)的和基于模型(有监督)的几种方法。其中基于规则的方法可细分为基于词性模板、基于依存句法模式；基于模型的可细分为基于序列标注、基于seq2seq生成、基于span的分类。

总的来说，开放抽取的流程，可以理解为给定一个句子，从原文中抽取符合要求的spo的成分，然后再根据要求，对so进行实体标签分类，对p进行关系标准化或者聚类，从而完成规范管理，这与限定域抽取的顺序是有一定颠倒的（如果不进行后续的标准化工作，其效果将大打折扣）

本项目中我们使用的是无监督基于规则的开放知识抽取，因为对于有监督而言问题的关键还是标注数据的获取规模较小（我们手里并没有很好的气象文本标注数据）。

无监督基于规则的开放信息抽取，优点在于不需要标注数据，只需要利用语法或者句法规则识别出特定的成分，筛选出高质量的三元组。无监督基于规则的代表系统有textrunner和srandfordioe，感兴趣可自行Google查看（因不同的系统其详细处理方式不同，故不再详细展开）。

3.2 基于深度学习

> 这是一开始我尝试使用的方式，但后来经过尝试后发现效果并不好，因此并没有采用该方法。

基于深度学习的三元组提取是指，假设我们拥有一个已经训练好（或微调好）的领域实体-关系抽取模型，它能够完成的任务应该是输入一句话（如“据报道称，新冠肺炎患者经常会发热、咳嗽，少部分患者会胸闷、乏力，其病因包括：1.自身免疫系统缺陷\n2.人传人”），输出如下形式的内容（即三元组(实体,关系,客体)）

```
[
 {
  "text": "据报道称，新冠肺炎患者经常会发热、咳嗽，少部分患者会胸闷、乏力，其病因包括：1.自身免疫系统缺陷\n2.人传人",
  "triples": [
   [
    "新冠肺炎",
    "主治",
    "肺炎"
   ],
   [
    "新冠肺炎",
    "主治",
    "发热"
   ],
   [
    "新冠肺炎",
    "主治",
    "咳嗽"
   ],
   [
    "新冠肺炎",
    "主治",
    "胸闷"
   ],
   [
    "新冠肺炎",
    "主治",
    "乏力"
   ],
   [
    "新冠肺炎",
    "注意事项",
    "自身免疫系统缺陷"
   ],
   [
    "新冠肺炎",
    "注意事项",
    "人传人"
   ]
  ]
```

```
 }
]
```

一般我们都使用的是BilSTM-CRF或Bert预训练模型来进行实体识别与抽取任务以及实体关系的联合抽取，无论使用哪种模型，都需要训练数据。要求的训练数据形式要求也是三元组形式，大致如下

```
[
 {
  "text": "13的三体综合征的临床表现是13-三体综合征主要特征为严重智能低下、特殊面容、手足及生殖器畸形，并可伴有严重的致死性畸形",
  "spo_list": [
   [
    "13-三体综合征",
    "临床表现",
    "特殊面容"
   ],
   [
    "13-三体综合征",
    "临床表现",
    "严重的致死性畸形"
   ]
  ]
 }
]
```

简单来说，基于深度学习的三元组抽取就是基于带标注的语料库利用机器学习或深度学习的相关模型实现实体（如时间、地点、经济损失等）和关系（发生于、发生在等）的抽取。

这需要进行手动标注和划分得到训练数据（针对专业领域的知识图谱的三元组除了手动处理数据外没有更好的方法，因为网上根本就没有现成的训练数据）。我们拥有的是部分带实体标注（注意并非关系标注）的数据，主要分为位置实体、承载体实体、人口面积实体、时间实体四类实体，大致形式如下（位置实体），但是这种类型的文本对于当前我们要进行的任务来说没有什么意义。

```
25日，记者从省民政厅获悉，受今年第10号台风“麦德姆”外围环流影响，我省部分地区出现暴雨到大暴雨和局地强对流天气过程，致使部分地区遭受洪涝灾害。
截至7月25日16时，我省 河源/LOC 、 揭阳/LOC 、 云浮/LOC 等3个市、4个县(市、区)、16个镇(街)不同程度受灾，全省受灾人口1.83万人；因雷击死亡1人( 郁南县/LOC 都城镇/LOC ); 紧急转移安置0.21万；倒塌房屋88间，113间房屋不同程度损坏；农作物受灾面积2746公顷，其中绝收69公顷；直接经济损失约0.23亿元。
灾情发生后，灾区民政部门迅速派出工作组深入灾区一线开展灾害救助工作，及时收集、整理、上报灾情和灾害救助工作情况，妥善安置受灾群众，做好因灾死亡人员家属抚慰和善后工作，及时发放救灾款物，保障了受灾群众基本生活。(记者/李强)
```

前面说手动标注和划分训练数据不现实，那么是否可以使用现成的模型进行训练数据的提取呢？这里尝试使用斯坦福大学的OpenIe进行（实体，关系，实体）三元组的提取，其OpenIe服务器的启动命令（如果是中文的话需要额外增加配置命令并下载中文包）

```
for /l %i in (9000,1,9100) do @(echo %i & netstat -aon | findstr /R /C:"%i.*LISTENING" >nul || (java -mx4g -cp "*" edu.stanford.nlp.pipeline.StanfordCoreNLPServer -port %i -timeout 15000 -threads 5 -maxCharLength 100000 & goto :eof))
```

经过实测这个工具的效果不是很好（可以说相当差劲，这也证实不进行微调直接应用在专业领域的模型表现明显差强人意）。经过实测，使用其他的一些比如中文的处理工具hanlp、tlp等直接提取三元组的效果也不好，要么就是请求次数有限制，要么是结果太差完全不能使用。

3.3 基于规则

因此我们需要切换处理思路。现在的目标是要构建专业领域的知识图谱，基于深度学习需要先手动提取训练数据（三元组），对模型进行微调后得到可用于专业领域三元组提取的模型。但是因为根本没有用于训练的三元组数据，手动提取的方式也几乎不可能（一方面是数据量要求太大，人工提取完全不可能，另一方面对于这种专业领域的三元组提取也存在知识的界限），因此使用深度学习模型进行三元组的提取的方案被否决。

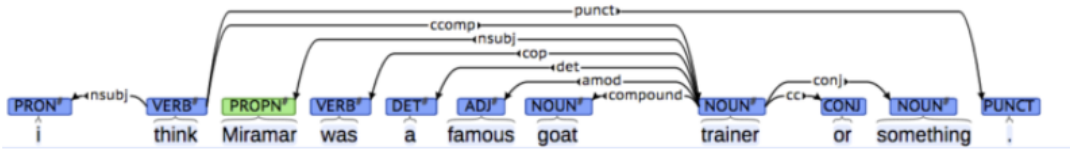
除了基于机器学习的方式，还有一种最基本的方法就是基于规则。注意此处所说的基于规则并不是传统的基于正则表达式或自定义的规则模式，因为对于我们手里的气象文本来说无论是使用正则表达式还是自定义的规则模式的效果都很差。这里使用的方式是先对句子进行依存句法分析（推荐借助现成工具实现），得到句子成分之间的关系之后，我们可以提取其中的如“主谓宾”或其他依赖关系作为（实体,关系,实体）三元组的成分。

3.3.1 依存句法分析

句法分析的基本任务是确定句子的句法结构或句子中词汇之间的依赖关系，句法分析一般不是语言分析任务的最终目标，但是它往往是实现最终目标的重要环节。句法分析主要分为句法结构分析和依存关系分析（也称依存句法分析）。

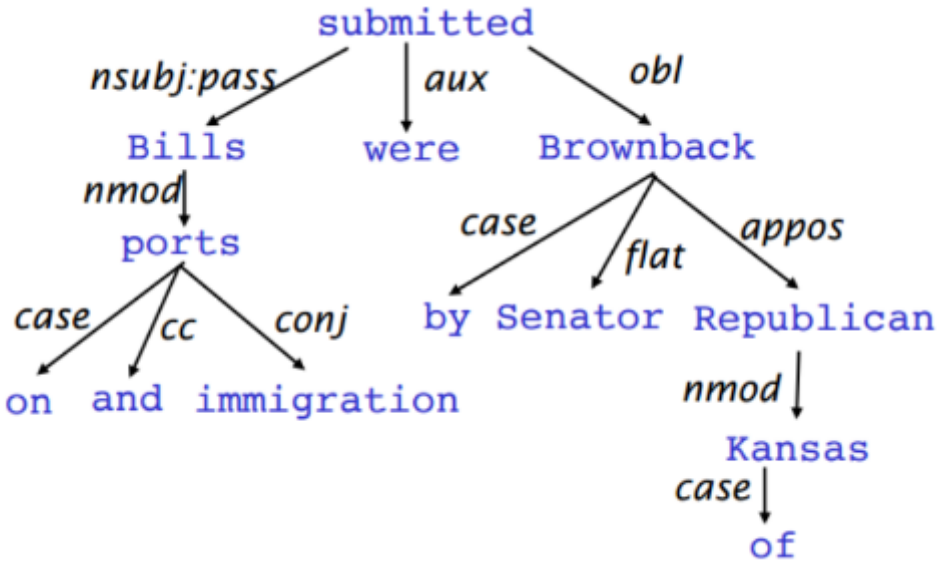
相比于 CFG(上下文无关文法)，依存关系语法更关注与词语间关系、高度词汇化，能更好的应用于问答系统与关系抽取等场景。另外，这种语法对就这种词语的顺序要求相对比较低，所以在处理一些语法复杂、次序排列更灵活的语言时，依存关系语法比 CFG 更有优势。

依存文法分析考虑的是句子中单词与单词之间的依存关系，而这种依存关系其实就是语法，例如主谓、动宾、形容词修饰名词等。因为构建语法树的时候选择根节点是随机的，所以选择不同的根节点可能得到不同的树结构，依存句法分析的任务就是在所有可能的树结构中选择出最合适的结构，一般依存句法分析的将原来的句子使用箭头标注出关系



（图中增加了一个根节点“ROOT”，使得completed也有依赖对象）

另一种Dependency Structure的表现形式是树状结构



总的来说，依存句法分析旨在通过分析句子中词语之间的依存关系来确定句子的句法结构，其中依存句法分析标注关系集合（此处展示的是DDParser对应的集合，注意不同工具的集合可能不同）如下图所示

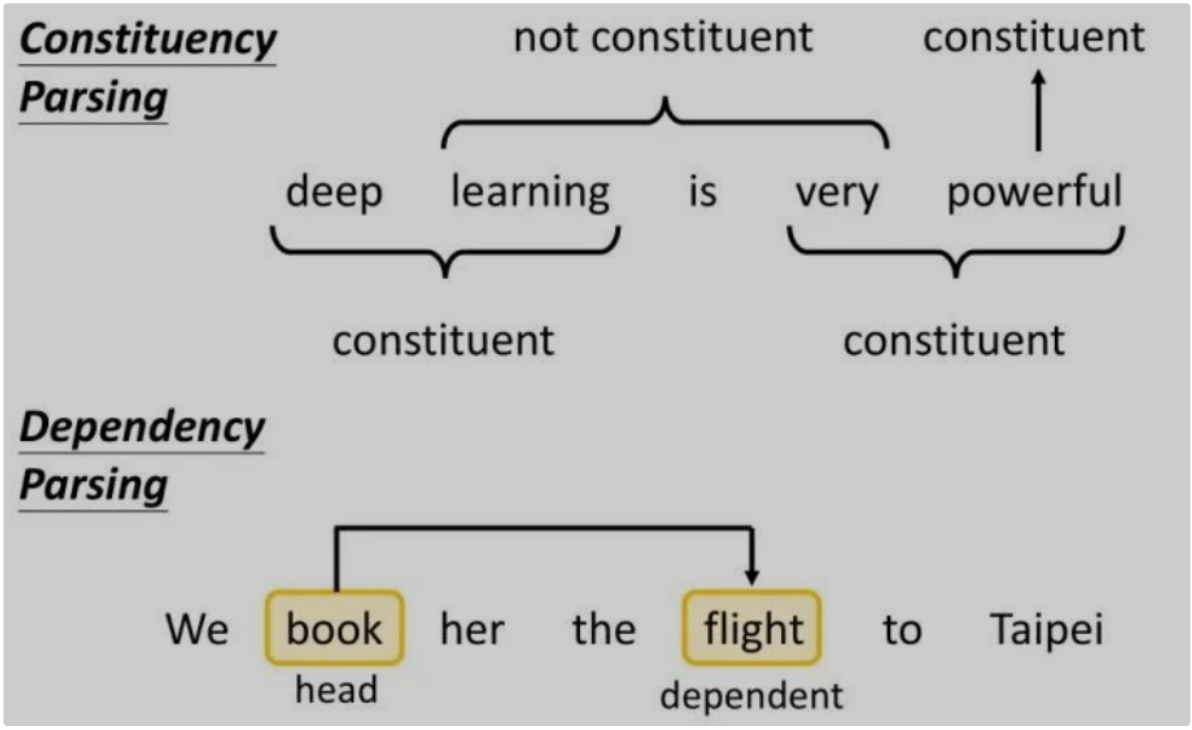
Label	关系类型	说明	示例
SBV	主谓关系	主语与谓词间的关系	他送了一本书(他<--送)
VOB	动宾关系	宾语与谓词间的关系	他送了一本书(送-->书)
POB	介宾关系	介词与宾语间的关系	我把书卖了（把-->书）
ADV	状中关系	状语与中心词间的关系	我昨天买书了（昨天<--买）
CMP	动补关系	补语与中心词间的关系	我都吃完了（吃-->完）
ATT	定中关系	定语与中心词间的关系	他送了一本书(一本<--书)
F	方位关系	方位词与中心词的关系	在公园里玩耍(公园-->里)
COO	并列关系	同类型词语间关系	叔叔阿姨(叔叔-->阿姨)
DBL	兼语结构	主谓短语做宾语的结构	他请我吃饭(请-->我，请-->吃饭)
DOB	双宾语结构	谓语后出现两个宾语	他送我一本书(送-->我，送-->书)
VV	连谓结构	同主语的多个谓词间关系	他外出吃饭(外出-->吃饭)
IC	子句结构	两个结构独立或关联的单句	你好，书店怎么走？(你好<--走)
MT	虚词成分	虚词与中心词间的关系	他送了一本书(送-->了)
HED	核心关系	指整个句子的核心	

在依存句法分析中，考虑某些最有可能形成（实体,关系,实体）三元组的组成成分：

- 名词和名词短语（实体）：名词和名词词组通常代表句子中的实体。它们可以是专有名词（人名、地名、组织），也可以是通用名词（物体、概念等）；
- 动词和动词短语（关系）：动词和动词词组通常表示实体之间的关系或动作。它们描述了实体正在做什么，或者它们是如何相互关联的；
- 至于介词和介词短语、形容词和形容词短语、副词和副词短语等作为实体或关系的修饰，可以不考虑以此降低文本分析的复杂度；

> Q：成分句法分析和依存句法分析的区别是什么？

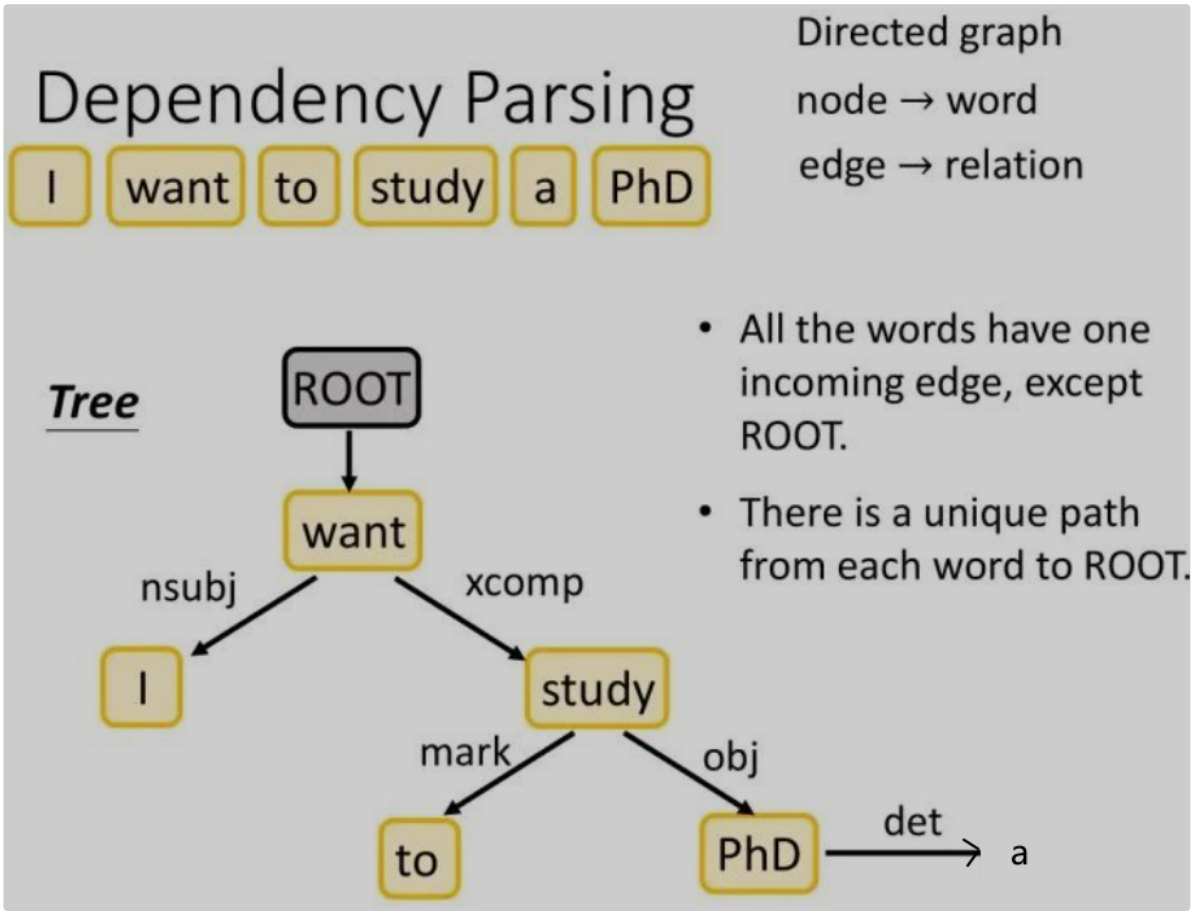
A：成分句法分析相当于考虑广义上的嵌套关系命名实体识别，依存句法分析相当于文本结构化任务中与命名实体识别配合的实体关系抽取任务。



成分句法分析考虑的是，某两个相邻词汇是否能够接在一起构成成分，而依存句法分析考虑的是两个词汇之间的关系。

依存分析不考虑两个词汇是否一定相邻：

- book与flight并不直接相邻，但是flight是book的宾语，book是flight的主语，使用有向箭头来表示这种依存关系，其中箭头的起点称为head，终点称为dependent。



更具体的说，依存分析就是将一个句子变成一个有向图：

- 图中的每个节点是一个词汇，每条边是一种关系；
- 除了根节点（在实际的句子中该根节点不代表任何词汇），每个节点词汇只有一个入度的边，但是每个词汇都可以指向多个其他词汇；

- 依存分析得到的有向图是一个树状结构，其中的每个词汇都有唯一的一条路径回溯到根节点；

依存分析树的构建非常简单，只需要丢给第一个二分类器两个词汇和上下文，该二分类器会首先判断两个词汇之间是否存在依存关系，若存在则第二个多分类器会输出其关系的类别；

对于一个有N个word的句子，整个依存分析模型的输入是ROOT加上N个词汇组成的句子，分析的过程就是取出$(N+1)^2$个单词对，逐个丢入分类器进行关系分类；

依存分析有很多方法来完成，这里因为篇幅的原因故不再赘述，感兴趣可自行Google。

3.3.2 DDParser工具

我们选择的依存句法分析工具是百度的DDParser，其Github项目地址为：https://github.com/baidu/DDParser。选择该工具的原因一方面是配置环境较简单；另一方面百度作为最大的中文搜索引擎，其对中文语料的处理会更加合理。配置好环境之后，使用DDParser来进行测试

```
import pandas as pd
# 依存句法分析
from ddparser import DDParser
ddp = DDParser(use_pos=True)
str = '清华大学研究核能的教授有哪些'
result = ddp.parse(str)
# 格式化输出
col = ('FROM','POSTAG','HEAD','DEPREL',)
row = []
for res in result:
    for i in range(len(res['word'])):
        data = {"FROM":res['word'][i], "POSTAG":res['postag'][i], "HEAD":res['head']
[i], "DEPREL":res['deprel'][i]}
        row.append(data)

df = pd.DataFrame(row,columns=col)
df.fillna('_',inplace = True) # 缺省值
df.index = df.index + 1  # 把索引号加1视为ID，一定不能直接另索引号为ID，0是虚拟ROOT节点不能占用
print(df)
```

得到的格式化输出如下（可以看到效果还不错，第一列是ID列，ID为0可以认为是虚拟的ROOT根节点）

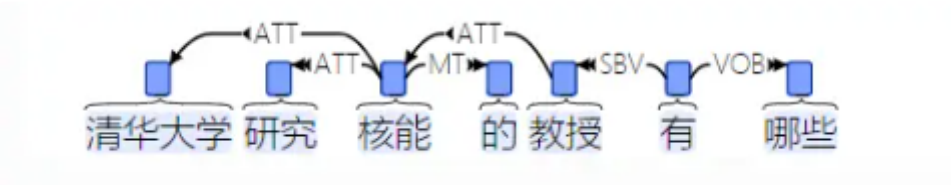
	FROM	POSTAG	HEAD	DEPREL
1	清华大学	ORG	3	ATT
2	研究	v	3	ATT
3	核能	n	5	ATT
4	的	u	3	MT
5	教授	n	6	SBV
6	有	v	0	HED
7	哪些	r	6	VOB

- DEPREL列
    - FORM：当前词语或标点
    - POSTAG：当前词语或标点的词性（POSTAG是细粒度词性标注，使用的是百度的LAC，参考链接：https://github.com/baidu/lac，对应的词性和专有类别标签如下表）

- HEAD：当前词语或标点的中心词id
- DEPREL：当前词语或标点与中心词的依存关系

标签	含义	标签	含义	标签	含义	标签	含义
n	普通名词	f	方位名词	s	处所名词	nw	作品名
nz	其他专名	v	普通动词	vd	动副词	vn	名动词
a	形容词	ad	副形词	an	名形词	d	副词
m	数量词	q	量词	r	代词	p	介词
c	连词	u	助词	xc	其他虚词	w	标点符号
PER	人名	LOC	地名	ORG	机构名	TIME	时间

其对应的可视化图形如下（因为长文本输出的CoNLL非常难直接看出关系，所以可视化非常有必要，在线可视化工具：http://spyysalo.github.io/conllu.js/，注意需要在输入框中输入完整的CoNLL格式文本）



通过给出的依存句法分析标注关系集合可以进行对应：

- “有”是整句话的核心；
- “教授”和“有”之间存在主谓关系，“教授”是主语，“有”是谓语；
- “有”与“哪些”之间存在动宾关系；
- “清华大学”、“研究”与“核能”之间存在定中关系，可以认为是“清华大学核能”或“研究核能”；
- “核能”与“教授”之间存在定中关系；
- “核能”的虚词成分是“的”；

从上述分析结果可以提取出如下三元组：

- （清华大学，研究，核能）
- （教授，研究，核能）
- （教授，有，哪些）

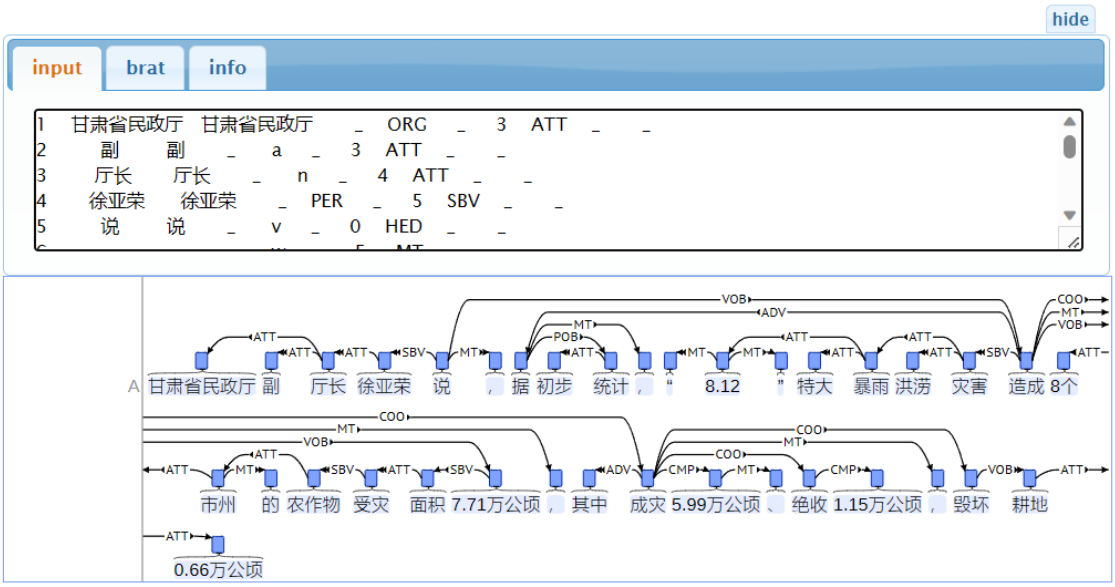
现在回到我们要分析的气象文本来，这里拿气象文本中的一个句子做分析

“甘肃省民政厅副厅长徐亚荣说，据初步统计，“8.12”特大暴雨洪涝灾害造成8个市州的农作物受灾面积7.71万公顷，其中成灾5.99万公顷、绝收1.15万公顷，毁坏耕地0.66万公顷”

这是一个非常长的句子，直接使用LLM(大语言模型)提取三元组得到的结果并不稳定（毕竟是生成式模型）。我们先手动提取三元组，然后观察三元组中的实体-关系-实体之间有什么关系。介于研究领域的特殊性，规定实体仅限于“地点”、“承载体”、“人口面积”和“时间”；规定关系包含但不仅限于“发生于”

- （暴雨洪涝，造成，农作物受灾）
- （农作物，受灾面积，7.71万公顷）
- （市州，成灾，5.99万公顷）
- （市州，绝收，1.15万公顷）
- （市州，毁坏耕地，0.66万公顷）

当然上述三元组是根据我们主观意识来进行抽取的，似乎并不存在什么明显的规律。我们来看依存句法分析的结果



可以看到因为这个句子非常的长，这就导致生成的依存图很复杂。实际上在进行依赖分析以提取三元组之前，删除如形容词和副词等修饰词是一种常见的预处理方法，通过删除这些修饰语，可以更专注于句子中的核心实体和关系（当然删除修饰语可能会导致一些信息的丢失，需要根据具体的任务进行微调）。

二、程序设计

1.需求分析

搜索与推荐是从海量信息中提取感兴趣内容的两种技术手段：

- 相对于主动检索的方式，推荐是一种被动接收的过程，省略了用户主动检索带来的各种问题；
- 但是推荐系统中的稀疏性问题依然严重降低了推荐系统的准确度；

知识图谱因为具有强大的知识互联能力，将知识图谱应用在推荐算法中无疑可以极大的提升推荐系统的性能。

知识图谱一般根据覆盖范围和应用场景大致分为通用知识图谱和领域知识图谱：

- 通用知识图谱依赖常识知识，具有涵盖范围广、实体与实体间关系数量大的特点；
- 领域知识图谱虽然在规模上比通用知识图谱小，但在内容上专业性更强；

本次项目要求基于已有的语料库（亦或使用爬虫爬取）构建气象灾情知识图谱。在已有的数据集基础上，对数据集进行处理，通过知识抽取、知识融合等技术将三元组信息存入Neo4j图数据库中，完成知识图谱的构建，并进行可视化展示。

2.总体设计

2.1 数据预处理

整个项目使用的语料数据是老师给的未标注的暴雨洪涝灾情文本，文本的基本形式如下

“8·12”暴雨洪涝致甘肃8市州受灾 死亡43人
2010年08月19日 18:42
中国新闻网
　　中新网兰州8月19日电（记者 刘薛梅）甘肃省政府新闻办19日下午召开新闻发布会称，8月11日至12日，甘肃陇南、天水、定西、甘南、兰州、庆阳、临夏、平凉等8个市州的42个县市区、271个乡镇、2241个村遭遇特大暴雨洪涝灾害，致159.38万人受灾，因灾死亡43人、失踪25人、受伤339人。目前已紧急转移安置15.9万人。　　甘肃省民政厅副厅长徐亚荣说，据初步统计，“8.12”特大暴雨洪涝灾害造成8个市州的农作物受灾面积7.71万公顷，其中成灾5.99万公顷、绝收1.15万公顷，毁坏耕地0.66万公顷。倒塌房屋1.93万间、损坏房屋5.43万间。直接经济损失42.46亿元。　目前，各地灾区正在进一步核实灾情。完

文本的第二行和第三行对于提取暴雨洪涝灾情三元组没有意义，因此需要删除。因为标题和正文内容的处理方式略有不同，因此分开介绍。

处理正文内容首先要提取出正文内容，简单认为所有语料的正文内容都是从第四行开始，利用python的 readlines 方法和 if not in 判断语句可以删除前面几行。因为正文内容一般都比较长，因此还需要借助句号对内容进行分割，将得到的句子存储在目标文件中，每个文件各占一行。如下是预处理正文内容的代码

```
def text_ext():
    target_file = open('middle_data.txt', 'w', encoding='utf-8') # 中间文件
    datapath = './original_data'
    dirs = os.listdir(datapath)
    for dir in dirs:
        fname = datapath + '\\' + dir
        # print(fname)
        with open(fname, 'r', encoding='utf-8') as src_file:
            lines = src_file.readlines()
            dirtyid = [0,1,2] # 删除1,2, 3行
            str_new = ''
            for x in range(len(lines)):
                if x not in dirtyid:
                    str_new = str_new + lines[x].rstrip('\n') # 删除换行符
            # print(str_new)
            # str_new是一个非常长的文本，接下来需要删除其中的空格，然后按照句号将其分为句子
            no_spaces = re.sub(r"\s+", "", str_new)
            # print(no_spaces)
            sentences = no_spaces.split('。') # 以句号为分隔符，将文本分为句子
            for sentence in sentences:
                target_file.write(sentence + '\n') # 将句子写入到目标文件中，每个句子占一
行
    target_file.close()
```

正文内容经预处理过后得到的文本形式如下

```
记者2日从国家防汛抗旱总指挥部办公室获悉，今年以来全国因洪涝灾害死亡377人、失踪94人
据国家防办最新统计，今年以来的洪涝灾害共造成28省(区、市)5458万人受灾，因灾死亡377人、失踪94人，农作物受灾4581千公顷，倒塌房屋16万间
与2000年以来同期相比，洪涝灾害受灾人口偏少五成，死亡人口偏少七成，农作物受灾面积偏少五成，倒塌房屋偏少八成多
与去年同期相比，受灾人口、死亡人口、直接经济损失偏少四成，农作物受灾面积均偏少五成，倒塌房屋偏少六成
刚刚过去的8月，全国共发生2次较强降水过程，降水量与常年基本持平但分布不均，有21省(区、市)590万人遭受洪涝灾害，其中贵州、重庆、福建、浙江受灾较重
(记者林晖)
中新网6月13日电据民政部网站消息，民政部国家减灾办发布2016年5月份全国自然灾害基本情况，6－10日和19－21日南方降水过程影响范围广、灾害程度深
据统计，洪涝和地质灾害共造成全国17个省(自治区、直辖市)和新疆生产建设兵团629.4万人次受灾，因灾死亡失踪101人，紧急转移安置18.9万人次；倒塌房屋1万间，损坏房屋9.6万间；直接经济损失90.3亿元，南方12省(自治区、直辖市)各项灾情指标占全国9成以上，福建、湖南和广东等地灾情较重
```

对标题的预处理相对简单，直接提取文本首行内容即可，使用的方法和第一次作业“新闻文本分类”使用的方法相同。将提取到的新闻标题写入目标文件中

```python
def title_ext():
    target_file=open('./data/title_data.txt','w',encoding='utf-8')
                    #汇总结果文件
    #提取data文件夹中所有文件的第一行
    datapath = './original_data'
    dirs = os.listdir(datapath)            #dirs得到所有txt文件名
    for dir in dirs:
        fname = datapath+'\\'+dir
        with open(fname,'r',encoding='utf-8')as src_file:
            lines =src_file.readlines()
            first_line = lines[0] # 取第一行
            print(first_line)
            target_file.write(first_line)
    target_file.close()
```

提取到的新闻标题内容如下

```
2014年以来全国因洪涝灾害死亡377人 失踪94人
5月份全国自然灾害以洪涝、地质和风雹灾害为主
6月28日以来南方暴雨洪涝灾害致150万余人受灾
7月1日以来暴雨洪涝风雹灾害致70人死亡或失踪
7月以来16省份遇暴雨洪涝灾害 200余万人受灾
7月以来洪涝灾害造成24省份2027万人次受灾
7月份以来南方洪涝灾害已导致5793.1万人受灾
“8·12”暴雨洪涝致甘肃8市州受灾 死亡43人
“麦德姆”致广东部分地区洪灾 受灾人口1.83万
三峡库首宜昌夷陵区今夏已连遭6次特大洪灾
专家：湖南临湘市特大山洪泥石流灾害系自然灾害
东北4省份局部发生洪涝灾害 致11.9万人受灾1人死亡
东北三省洪灾已致85人死亡 习近平李克强做出指示
东北华北风雹洪涝灾情持续 已致内蒙辽宁3人死亡
东北华南遭受大面积洪涝灾害 辽宁因灾死亡54人
```

2.2 三元组提取

前面简单的介绍了百度的DDParser工具对句法进行分析，下面我们将利用对句子的分析结果从中提取出三元组。

尽管前面简单的使用句号对长文本进行了分句，但实际上某些划分后的文本长度仍然较长，原因是文本的句子划分并不只是靠句号，其他诸如问号、换行符等都可以作为划分句子的标准，因此需要进一步的进行划分，这里使用正则表达式来实现

```
def split_sents(self, content):
        return [sentence for sentence in re.split(r'[? ?! !。 ; ;: :\n\r]', content) if
sentence]
```

该方法接受一个名为content的参数，该参数是一个字符串，表示待处理的文本内容。该方法的目的是将字符串进行拆分，将其作为列表返回。

- re.split()方法：按照能够匹配的子串将字符串分割后返回列表 -- 此模式表示一个字符集，该字符集包括各种标点符号和换行字符，这些标点符号和字符通常用于中文和英文的句子结尾，当函数遇到这些字符时进行拆分
- 通过列表推导式，迭代从上一步获得的列表中的每个元素句子，将列表中的空字符串去除，确保最后的结果只包含非空语句
- 最后返回一个语句列表，其中包含了所有的句子

  总的来说，该方法根据常见的标点符号和换行符将给定的文本内容拆分为单独的句子，删除了空字符串，最后返回结果列表。

下面的build_parse_child_dict()方法执行句法分析，主要输入参数如下：

- words:单词列表
- postags:词性标签
- rel_id:依存关系id
- relation:依存关系

该方法处理所提供的单词、词性和依赖性信息，以生成两个列表：child_dict_list和format_parse_list，前者以字典的形式表示每个单词的子依赖性，后者为每个单词提供格式化信息。

```
def build_parse_child_dict(self, words, postags, rel_id, relation):
        child_dict_list = [] # 保存句法依存儿子节点的字典
        format_parse_list = [] # 保存格式化后的依存关系
        for index in range(len(words)):
            child_dict = dict() # 空字典child_dict
            for arc_index in range(len(rel_id)): # 使用arc_index变量迭代rel_id列表的索引
                if rel_id[arc_index] == index+1:   # 检查arc_index处的rel_id值是否等于
index+1，如果是，则表示依赖关系对应于索引处的当前单词。然后，代码相应地更新child_dict
                    if rel_id[arc_index] in child_dict: # 如果当前的rel_id值已经作为关键
字存在于child_dict中，则arc_index会附加到相应的索引列表中
                        child_dict[relation[arc_index]].append(arc_index)
                    else: # 否则，将创建一个新键，并将其值初始化为包含arc_index的空列表
                        child_dict[relation[arc_index]] = []
                        child_dict[relation[arc_index]].append(arc_index)
            child_dict_list.append(child_dict) # 在处理了当前单词的所有依赖关系后，
child_dict被附加到child_dict_list
        heads = ['Root' if id == 0 else words[id - 1] for id in rel_id]  # 将每个依赖关
系id与其对应的单词匹配，若id为0则为根节点
        for i in range(len(words)):
            # ['ATT', '李克强', 0, 'nh', '总理', 1, 'n']
            a = [relation[i], words[i], i, postags[i], heads[i], rel_id[i]-1,
postags[rel_id[i]-1]] # 为每个单词创建解析信息的格式化表示
```

```
            format_parse_list.append(a)
        # child_dict_list包含每个单词的依赖关系，format_parse_list包含每个单词的格式化解析
信息
        return child_dict_list, format_parse_list
```

我们举一个例子来理解，假设函数的输入如下

```
words = ['I', 'ate', 'an', 'apple']
postags = ['PRON', 'VERB', 'DET', 'NOUN']
rel_id = [2, 0, 4, 2]
relation = ['nsubj', 'root', 'det', 'obj']
```

函数首先会初始化两个空列表，child_dict_list和format_parse_list。

接着在第一个循环中使用索引变量对单词列表的索引进行迭代，在每个循环中都会为每个单词创建一个新的空字典child_dict。

第二个循环使用arc_index变量对rel_id列表的索引进行迭代。它检查arc_index处的rel_id值是否等于index+1（因为rel_id索引从1开始）。如果它们匹配，则意味着依赖关系对应于索引处的当前单词。然后进行判断，如果rel_id值已经作为child_dict中的键存在，则arc_index将被附加到相应的索引列表中。否则，将创建一个新键，并将其值初始化为包含arc_index的空列表。此步骤为每个单词建立子依赖关系。在我们的例子中，child_dict_list将会是如下形式

```
[
    {},
    {'nsubj': [0]},
    {'det': [2]},
    {'obj': [3]}
]
```

- 索引0处的字典为空，因为单词“I”没有依赖项。
- 索引1处的字典（对应于“ate”）有一个依赖项，关系“nsubj”指向索引0处的单词（“I”）。
- 索引2处的字典（对应于“an”）有一个依赖项，关系“det”指向索引2处（“an”）的单词。
- 索引3处的字典（对应于“apple”）有一个依赖项，关系“obj”指向索引3处（“apple”）的单词。

接下来构建一个heads列表，该列表将每个依赖关系id与其对应的单词进行匹配，例子中的heads将是[“ate”、“Root”、“apple”、“ate”]。

接着进入下一个for循环，该循环会对每个单词创建一个格式化的表示，这将结合各种信息，我们的例子中得到的format_parse_list将是

```
[
    ['nsubj', 'I', 0, 'PRON', 'ate', 1, 'VERB'],
    ['root', 'ate', 1, 'VERB', 'Root', 0, 'PRON'],
    ['det', 'an', 2, 'DET', 'apple', 3, 'NOUN'],
    ['obj', 'apple', 3, 'NOUN', 'ate', 1, 'VERB']
]
```

每个子列表表示一个单词的格式化信息。比如第一个子列表对应具有“nsubj”关系的单词“I”，它的索引为0，词性标签为“PRON”，它的head word是“ate”，head word的词性为“VERB”。

build_parse_child_dict()方法只是一个辅助方法，主要用于辅助构造解析树，而build_parse_child_dict()方法的参数需要使用DDParser对句子进行依存分析得到

```
def parser_main(self, sentence):
        res = self.parser.parse(sentence, )[0] # 调用DDParser的parse方法，对句子进行依存句法分析
        # 分别将解析的结果提取出来
        words = res["word"]
        postags = res["postag"]
        rel_id = res["head"]
        relation = res["deprel"]
        # 根据提取的信息构建解析树结构，并返回两个值：child_dict_list 和 format_parse_list
        child_dict_list, format_parse_list = self.build_parse_child_dict(words, postags, rel_id, relation) # 调用build_parse_child_dict方法，为句子中的每个词语维护一个保存句法依存儿子节点的字典
        return words, postags, child_dict_list, format_parse_list # 该方法返回解析后的词语、词性标注、以子字典列表形式表示的解析树（child_dict_list）以及格式化的解析列表（format_parse_list）
```

merge_ATT方法，该方法对解析树中的依存关系进行了合并操作。它识别出连续的 'ATT' 或 'ADV' 依存关系，并将相应的词合并为一个单词。合并后的词被添加到 ATTs 列表中，并且每个合并序列中最后一个词的索引被添加到 retain_nodes 集合中。该方法最终返回输入的 words 列表的修改版本、原始的 postags 列表、修改后的 format_parse_list 列表以及 retain_nodes 集合。

```
def merge_ATT(self, words, postags, format_parse_list):
        words_ = words # 浅拷贝
        retain_nodes = set()
        ATTs = []
        ATT = []
        format_parse_list_ = []
        for parse in format_parse_list: # 遍历format_parse_list集合，每个parse都是一个包含特定依存关系信息的列表
            dep = parse[0]
            if dep in ['ATT', 'ADV']: # 如果依存关系是ATT或ADV，则将其添加到ATT列表中
                ATT += [parse[2], parse[5]]
            else: # 如果依存类型不是 'ATT' 或 'ADV'，代码会检查 ATT 列表是否为空
                if ATT: # 如果ATT列表不为空，则将其合并为一个字符串，并将其添加到ATTs列表中
                    body = ''.join([words[i] for i in sorted(set(ATT))])
                    ATTs.append(body) # 将 ATT 列表中索引对应的词连接起来形成一个单词（body），并将其添加到 ATTs 列表中
                    retain_nodes.add(sorted(set(ATT))[-1]) # 将 ATT 列表中最后一个索引添加到retain_nodes集合中
                    words_[sorted(set(ATT))[-1]] = body # 将 ATT 列表中最后一个索引对应的词替换为body
                else:
                    retain_nodes.add(parse[2]) # 如果 ATT 列表为空，意味着序列中没有 'ATT' 或 'ADV' 依存关系，此时只需将当前 parse 的索引添加到 retain_nodes 集合中即可
                ATT = []
        for indx, parse in enumerate(format_parse_list):
            if indx in retain_nodes: # 检查当前索引（indx）是否存在于 retain_nodes 集合中
                parse_ = [parse[0], words_[indx], indx, postags[indx], words_[parse[5]], parse[5], postags[parse[5]]] # 如果存在，则将该索引对应的词替换为 words_[indx]
                format_parse_list_.append(parse_) # 将该parse_添加到format_parse_list_中
        return words_, postags, format_parse_list_, retain_nodes
```

到这里可能我们会有一个疑问，为什么需要合并如“ATT”(定语修饰语)和“ADV”(状语修饰语)呢？实际上是为了将多个词汇合并为一个单词，从而更加有效的捕捉它们之间的关系。这里我们可以举个例子，原始句子为“The big red apple on the table is delicious.”，则“big”、“red”和“on the table”作为ATT修饰“apple”，解析树将为每个修饰语建立单独的依存关系，很明显这是一个碎片化的表示，如果将这三个成分合并为一个成分，即“big red on the table”，作为“apple”的ATT修饰，在保留基本信息的同时简化了结构，提高了解析树的可读性。

接下来的extract方法根据词性标注（postags）、依存解析（child_dict_list）和其他语言信息从给定的句子中提取语义三元组（主语-动词-宾语关系）。该方法旨在通过利用词性标注和依存解析信息从句子中提取主谓宾和主谓补关系，提取的三元组提供了关于句子的语义结构和关系的见解。

```
def extract(self, words, postags, child_dict_list, arcs, retain_nodes):
        svos = [] # 用于保存抽取出的三元组
        for index in range(len(postags)): # 遍历句子中的每个词
            if index not in retain_nodes: # 如果该词不在retain_nodes集合中，则跳过该词 -
- 只考虑对句子中的特定节点进行提取
                continue
            # 检查给定索引处的当前词语是否具有非空的词性标注（postags[index]）。如果不为空，
则根据句法结构提取语义三元组
            if postags[index]:
                child_dict = child_dict_list[index] # 获取当前词语的依存解析信息
                # 查找主谓宾（SVO）三元组，通过检查是否存在主语（SBV）和宾语（VOB）作为子节
点来确定
                if 'SBV' in child_dict and 'VOB' in child_dict:
                    r = words[index] # 提取谓语(动词)
                    e1 = words[child_dict['SBV'][0]] # 提取主语
                    e2 = words[child_dict['VOB'][0]] # 提取宾语
                    if e1.replace(' ', '') and e2.replace(' ', ''):
                        svos.append([e1, r, e2]) # 提取的主语（e1）、动词（r）和宾语
（e2）以语义三元组[e1, r, e2]的形式添加到svos列表中，分别表示主语、动词和宾语

                # 通过查找主语（SBV）和补语（CMP）关系来检查主谓补（SVC）三元组是否存在
                if 'SBV' in child_dict and 'CMP' in child_dict:
                    e1 = words[child_dict['SBV'][0]] # 提取主语
                    cmp_index = child_dict['CMP'][0] # 提取补语
                    r = words[index] + words[cmp_index] # 将谓词和补语连接起来形成动词
(谓词)
                    if 'POB' in child_dict_list[cmp_index]: # 检查补语的子节点是否存在介
宾关系（POB）
                        e2 = words[child_dict_list[cmp_index]['POB'][0]] # 如果存在介宾
关系，则提取介宾关系对应的宾语
                        if e1.replace(' ', '') and e2.replace(' ', ''):
                            svos.append([e1, r, e2]) # 提取的主语（e1）、谓词（r）和宾语
（e2）以语义三元组[e1, r, e2]的形式添加到svos列表中
        return svos
```

extract方法侧重使用特定的句法模式（主语-动词-宾语和主语-动词-补语）来提取三元组，我们这里介绍一种同样是基于词性标注和依存解析等语言信息提取语义三元组的方法ruler2。ruler2方法可以看作是extract方法的拓展，通过考虑其他模式（如带有形容词修饰的主语-动词宾语）来拓展提取定语后置和介词短语的主动补语（即介宾关系的主谓动补关系），它结合了额外的模式来捕捉句子中更加广泛的语义关系。ruler2的大部分逻辑和extract是相同的，代码注释如下

```
def ruler2(self, words, postags, child_dict_list, arcs):
        svos = [] # 用于保存抽取出的三元组
        for index in range(len(postags)): # 遍历句子中的每个词
```

```
            tmp = 1
            if tmp == 1: # 这个条件总是为真，因此这个条件是多余的...
                if postags[index]: # 如果语义角色标注（postags[index]）不为空，则使用语义角色标注的结果进行三元组抽取
                    child_dict = child_dict_list[index]
                    if 'SBV' in child_dict and 'VOB' in child_dict: # 查找主谓宾（SVO）三元组，通过检查是否存在主语（SBV）和宾语（VOB）作为子节点来确定
                        r = words[index] # 提取谓语(动词)
                        e1 = self.complete_e(words, postags, child_dict_list, child_dict['SBV'][0]) # 提取主语
                        e2 = self.complete_e(words, postags, child_dict_list, child_dict['VOB'][0]) # 提取宾语
                        if e1.replace(' ', '') and e2.replace(' ', ''):
                            svos.append([e1, r, e2]) # 提取的主语（e1）、动词（r）和宾语（e2）以语义三元组[e1, r, e2]的形式添加到svos列表中，分别表示主语、动词和宾语

                    relation = arcs[index][0]
                    head = arcs[index][2]
                    if relation == 'ATT': # 检查当前词语的依存关系（relation）是否为“ATT”（定语后置），并且检查该词语的头部词语索引（head）
                        if 'VOB' in child_dict: # 进一步检查是否存在宾语（VOB）作为子节点
                            # 若存在，则使用complete_e方法来获取完整的实体词语，并构建谓词（r）
                            e1 = self.complete_e(words, postags, child_dict_list, head - 1)
                            r = words[index]
                            e2 = self.complete_e(words, postags, child_dict_list, child_dict['VOB'][0])
                            # 构建一个临时字符串（temp_string），由谓词和宾语拼接而成。然后检查主语是否以该临时字符串开头，如果是，则从主语中移除该临时字符串的部分
                            temp_string = r + e2
                            if temp_string == e1[:len(temp_string)]:
                                e1 = e1[len(temp_string):]
                            # 检查临时字符串是否不在主语中，如果是，则将主语、谓词和宾语以语义三元组的形式添加到svos列表中
                            if temp_string not in e1:
                                if e1.replace(' ', '') and e2.replace(' ', ''):
                                    svos.append([e1, r, e2])


                    if 'SBV' in child_dict and 'CMP' in child_dict: # 通过查找主语（SBV）和补语（CMP）关系来检查主谓补（SVC）三元组是否存在
                        e1 = self.complete_e(words, postags, child_dict_list, child_dict['SBV'][0]) # 提取主语
                        cmp_index = child_dict['CMP'][0] # 提取补语
                        r = words[index] + words[cmp_index] # 将谓词和补语连接起来形成动词(谓词)
                        if 'POB' in child_dict_list[cmp_index]: # 检查补语的子节点是否存在介宾关系（POB）
                            e2 = self.complete_e(words, postags, child_dict_list, child_dict_list[cmp_index]['POB'][0]) # 如果存在介宾关系，则提取介宾关系对应的宾语
                            if e1.replace(' ', '') and e2.replace(' ', ''):
                                svos.append([e1, r, e2]) # 提取的主语（e1）、谓词（r）和宾语（e2）以语义三元组[e1, r, e2]的形式添加到svos列表中
```

```
        return svos
```

在ruler2方法中，使用到的一个重要的方法是complete_e，该方法根据句子中的词语信息、词性标注和依存解析结果，生成一个完整的实体词语，代码的注释如下

```
def complete_e(self, words, postags, child_dict_list, word_index):
        child_dict = child_dict_list[word_index] # 获取词语索引对应的子节点字典
        prefix = '' # 存储前缀
        # 如果 child_dict 中存在 'ATT'（定语）关系，则通过循环遍历所有定语关系的子节点，使用递归调用 complete_e 方法获取每个子节点的完整实体词语，并将其添加到 prefix 中
        if 'ATT' in child_dict:
            for i in range(len(child_dict['ATT'])):
                prefix += self.complete_e(words, postags, child_dict_list, child_dict['ATT'][i])
        postfix = '' # 存储后缀
        # 如果 postags[word_index] 为 'v'（动词）
        if postags[word_index] == 'v':
            if 'VOB' in child_dict: # 如果 child_dict 中存在 'VOB'（动宾关系），则通过递归调用 complete_e 方法获取宾语的完整实体词语，并将其添加到 postfix 中
                postfix += self.complete_e(words, postags, child_dict_list, child_dict['VOB'][0])
            if 'SBV' in child_dict: # 如果 child_dict 中存在 'SBV'（主谓关系），则通过递归调用 complete_e 方法获取主语的完整实体词语，并将其与 prefix 进行拼接
                prefix = self.complete_e(words, postags, child_dict_list, child_dict['SBV'][0]) + prefix
        # 返回拼接后的字符串，由 prefix、words[word_index] 和 postfix 组成，表示完整的实体词语
        return prefix + words[word_index] + postfix
```

这里简单举一个例子来理解complete_e方法的工作原理，假设有这样一个句子"John eats an apple."，输入complete_e的参数分别为：

- words: ["John", "eats", "an", "apple"]
- postags: ["NNP", "VBZ", "DT", "NN"] (["专有名词", "动词原形", "限定词", "名词"])
- child_dict_list: [{}, {}, {}, {}] (在此示例中为空)
- word_index: 1 ("eats"的索引)

从单词"eats"开始调用complete_e方法，由于"eats"没有"ATT"（定语）关系，因此prefix的循环不执行。接着由于"eats"是一个动词，因此需要检查child_dict中是否存在"VOB"（动宾关系）关系。在这个例子中"eats"不存在"VOB"关系，因此postfix为空字符串。同样的，"eats"不存在"SVB"(主谓)关系，因此prefix也是空字符串。最后，该方法返回prefix、words[word_index]和postfix的拼接。在这个例子中，它返回"eats"。这表明在调用complete_e方法处理句子"John eats an apple."中的单词"eats"时，因为"eats"没有额外的修饰词或依赖关系，因此简单的返回"eats"作为完整实体。

以上就是基于依存分析提取文本中的三元组主要使用的函数，当实例化SVOParser类后，调用triples_main方法对正文文本和标题文本的每一行进行三元组提取，将得到的三元组进行合并（代码比较简单这里就不再展示，详情参见代码文件），最终形成如下形式的原始三元组文件

```
['记者', '获悉', '全国死亡']
['洪涝灾害', '造成', '28省区5458万人受灾']
['全国', '发生', '降水']
['万人', '遭受', '洪涝灾害']
['民政部国家减灾办', '发布', '基本情况']
['洪涝灾害', '造成', '17个省自治区629.4万人次受灾']
['灾情指标', '占', '全国9成以上']
['损房屋数量', '均为', '次低值']
['南方地区', '出现', '降雨过程']
['部分省份', '遭受', '袭击']
['68个县', '遭受', '等洪涝灾害']
['洪涝灾害', '造成', '150.1万人受灾']
['四川', '有', '1人死亡']
['灾情信息', '称', '造成70人20个省份死亡']
['部分地区', '出现', '降雨过程']
['局地', '伴有', '对流天气']
['314个县', '遭受', '等洪涝灾害']
['7月1日以来暴雨', '造成', '754.5万人受灾']
['中国', '出现', '降雨过程']
['中国出现降雨过程', '导致', '201.4万人受灾']
['等东部地', '出现', '降雨过程']
['局地', '伴有', '对流天气']
['河北', '遭受', '等洪涝灾害']
['3人', '造成', '死亡']
['等东部地', '有', '强过程']
```

2.3 三元组清理

基本的三元组已经提取完成，但是可以看到存在很多相似的三元组以及一些无用的三元组（比如实体是标点符号）。因此这部分我们主要进行的就是对三元组进行清洗，得到质量较高、可用于构建知识图谱的三元组。

经过对原始三元组的查看，基本确定要清洗的三元组类型如下：

- 两个完全相同的三元组 ['东北局部地区', '遭受', '洪涝灾害'] 和 ['东北局部地区', '遭受', '洪涝灾害'] -- 去重
- 头实体或尾实体是单个字符(包括标点符号)的三元组，如['已', '致', '85人死亡']、[',', '致', '85人死亡']、['18人', '死', '4'] -- 在中文中要表示一个实体几乎是不可能只用一个字的
- 头实体或尾实体以“等”开头（这是我们文本的特殊性导致的一个问题），这种三元组不用删除，只需要删除头实体的“等”即可，如['等湘省份', '遭受', '洪涝灾害'] -- 规范化
- 头实体或尾实体是以数字开头的（同样是文本特殊性导致），如['1.6亿', '投入', '5万人']，这种三元组不具备语义 -- 语义角度，灾情文本中两个数字实体之间几乎不会有什么有用的关系（头实体和尾实体都是数字开头），亦或头实体或尾实体以数字开头，往往并不是我们需要的“唯一”实体
- 头实体或尾实体以“你”、“我”、“他”人称代词开头，不符合作为唯一实体的要求
- 头实体等于尾实体，自己与自己存在关系，从常理上讲行不通
- 头实体或尾实体意义接近，可以认为是两个相同的三元组如 ['两部门', '启动', '四级应急响应'] 和 ['两部门', '启动', 'Ⅳ级应急响应'] -- 可以用jaccard距离或余弦相似度来判断

2.3.1 重复三元组

首先删除重复的三元组，这里借助了集合元素唯一的特性，很方便的得到了去重之后的三元组

```
triplets = []
with open('./data/triples_data.txt', 'r') as file:
    triplets = [line.strip() for line in file]
unique_triplets = set()
for triplet in triplets:
    unique_triplets.add(triplet)
unique_triplets = list(unique_triplets)
with open('./data/unique_triples_data.txt', 'w') as file:
    file.write('\n'.join(unique_triplets))
```

2.3.2 非法三元组

接下来删除其他不符合条件的三元组（这里不包括实体相似的三元组，这将放在之后处理），这里使用的逻辑也非常简单，直接利用if语句进行判断即可。

需要注意的是因为我们的三元组尽管看起来像列表，实际上是以字符串的形式在文本中存储的，因此借助ast.literal_eval()函数将字符串转换为列表，进而可以访问三元组中的实体或者关系。

```
with open('./data/unique_triples_data.txt', 'r') as
file_in,open('./data/clear_triples_data.txt', 'w') as file_out:
    for line in file_in:
        triplet_str = line.strip()
        # print(triplet_str)
        triplet_list = ast.literal_eval(triplet_str)
        head_entity = triplet_list[0]
        relationship = triplet_list[1]
        tail_entity = triplet_list[2]
        if not (len(head_entity) ==1 or len(tail_entity) ==1 or
head_entity[0].isdigit() or tail_entity[0].isdigit() or head_entity == tail_entity or
\
                    head_entity[0]=="我" or tail_entity[0]=="我" or
head_entity[0]=="你" or tail_entity[0]=="你" or head_entity[0]=="他" or
tail_entity[0]=="他"):
            # print(triplet_list)
            if head_entity[0] == "等":
                head_entity = head_entity.replace("等","")
            if tail_entity[0] == "等":
                tail_entity = tail_entity.replace("等","")
            # print(head_entity,relationship,tail_entity)

 file_out.write(str(head_entity)+'\t'+str(relationship)+'\t'+str(tail_entity)+'\n')
```

2.3.3 实体&关系限定

额外的，还需要限定关系的类型数量以及实体的类型数量。一开始想要基于语料库中已有的实体标签和三元组中存在的实体的交集作为实体，这种效果不会很好，原因是语料中的实体如“人”、“房屋”、“农作物”，这还仅仅只是承载体，如果要算上人口面积（诸如“死亡失踪101人/ADP”，“转移安置18.9万人次/ATP”）、时间标签（如“7月份/DS”，“7月21日/DO”）等，因为本身的标注实体的效果就不好，如果再和我们使用依存分析得到的结果进行一个“负负得正”的操作，最后得到的效果会很差。同时，也不能为了减少实体就刻意的去直接删除某些实

体，这是不负责任的表现，比如“江西省”和“江西减灾委”就是两个实体，不能将其作为同一个实体。对于关系也是如此，如果仅仅只想限制关系的数量仅限于“发生在”、“有”这种，不仅不现实，而且会极大的减少三元组的数量。不能从集合的角度来处理实体和关系的话，可以从词嵌入即更高层次的技术来处理。

因为实体和关系现在都可以看作是单独的中文词语，因此处理词语特别是同义词、近义词，比如“出现”、“出现在”、“出现于”这三种关系实际上都是同一种关系，将其融合以减少关系的数量 -- 可以使用一种被称为同义词规范化的技术，选择使用最短的同义词或使用频率最频繁的同义词甚至随机的同义词作为规范形式。

注意因为这里需要使用到词嵌入，与后面直接使用TF-IDF向量计算余弦相似度做区分。后面的处理过程没有使用词嵌入是因为这两个步骤处理的视角不同。此处需要考虑为词嵌入是因为考虑到语义方面的不同（这需要深层次的对语言的理解，词嵌入是一个不错的选择）；而删除相似三元组是仅仅只需要将汉字看作是普通的集合中的元素，只需要比对集合中元素的个数即可。

现在的任务是寻找并清理中文近义词、同义词，要么使用现成的库（可能在专业邻域的效果不是特别好），要么自己训练word2vec（利用专业邻域文本），此处我们选择借助gensim库和手中已有的语料先训练一个专业领域的word2vec模型。具体代码文件参考word2vec.ipynb。主要步骤分为：

1. 创建语料文件：将text_data.txt和title_data.txt合并成一个文件，命名为data.txt；
2. 格式转换：将txt文件转化为csv文件便于之后的语料预处理；
3. 语料预处理：去除无用字符、分词、过滤通用词；
4. 词向量训练：借助gensim.models.word2vec中的LineSentence训练属于自己的词向量，训练完毕后将词向量和模型保存；

训练好的词向量模型可以用于词语的相似性计算，比如我们想要计算“暴雨”和“洪涝”这两个词之间的相似性，可以使用以下代码

```
# 计算相似度
model = Word2Vec.load('./model/corpus.model')
word_vectors = model.wv
similarity_score = word_vectors.similarity('暴雨', '洪涝') # 注意：这里的词语必须在语料中出现过，否则会报错
print(similarity_score)
```

返回的similarity_score是0.8775766，可以看到效果还是很好的。需要注意进行比较的两个词必须在语料中出现过，否则函数会返回错误。

这里我们不直接使用这个训练好的词向量进行相似性的计算，我们将二进制的模型打包成gz格式的文件，替换掉synonyms库（这是github上的一个中文近义词开源项目）中原本的通用词向量文件。使用synonyms库的好处在于可以处理没有出现在语料库中的词语。

```
>> Synonyms on loading stopwords [d:\Download_software\My_miniconda\envs\pytorch_py3.7\lib\site-packages\synonyms\data\stopwords.txt] ...
>> Synonyms on loading vectors [d:\Download_software\My_miniconda\envs\pytorch_py3.7\lib\site-packages\synonyms\data\words.vector.gz] ...
```

接下来使用synonyms库提供的compare方法同样可以计算两个词语之间的相似性

```
word1 = "暴雨"
word2 = "洪涝"
r = synonyms.compare(word1, word2, seg=False)
print(r)
```

接下来定义替换相似词语的函数，其基本的思想非常简单，就是设置一个相似性门限和unique_words列表。输入的word单词与unique_words列表中的每一个单词进行相似性的计算，如果计算出的相似度similarity大于max_similarity，则记录该similarity以及对应的unique_word。当比较完unique_words列表中的所有单词后，如果最终的max_similarity大于设置的门限，则表示输入的word在unique_words列表中有相似的词语，该词语就是max_similarity对应的unique_word；如果最终的max_similarity小于门限，则表示输入的word在unique_words列表中没有相似的词语，因此该word作为unique_words的新元素被加入。

```
def replace_similar_words(word,unique_words,threshold=0.7):
    if unique_words == []:
```

```
            unique_words.append(word)
            return word
        else:
            max_similarity = 0
            unique_str = ''
            for unique_word in unique_words:
                similarity = synonyms.compare(word, unique_word, seg=False)
                if similarity > max_similarity:
                    max_similarity = similarity # 保存最大相似度
                    unique_str = unique_word # 保存最大相似度对应的词
            if max_similarity >= threshold:
                return unique_str # 返回最大相似度对应的词
            else:
                unique_words.append(word)
                return word
```

有了替换相似性单词的函数，接下来只需要分别遍历txt文件，提取第一列、第二列、第三列的词语，分别进行replace_similar_words操作，然后再将每一列对应的每一行的词语拼接即可。执行完毕后得到synonyms_triples_data.txt文件。

2.3.4 相似三元组

最后是删除相似三元组的操作，这里既可以使用经典的余弦相似度，也可以使用易于理解的jaccard算法。经过测试，余弦相似度的效果更好，因此这里选择使用余弦相似度进行相似文本的清理。

```
triplets = []
with open('./data/synonyms_triples_data.txt', 'r') as file:
    triplets = [line.strip() for line in file]
unique_and_similar_triplets = []
# 将每个三元组与现有的唯一三元组进行比较
for triplet in tqdm(triplets):
    is_duplicate = False
    for unique_triplet in unique_and_similar_triplets:
        # 将三元组转换为TF-IDF向量
        vectorizer = TfidfVectorizer()
        tfidf_matrix = vectorizer.fit_transform([triplet, unique_triplet])
        # 计算余弦相似度
        similarity = cosine_similarity(tfidf_matrix[0], tfidf_matrix[1])[0][0]
        # 相似性阈值（根据需要进行调整）
        threshold = 0.8
        if similarity >= threshold:
            is_duplicate = True
            break
    # 跳过相似的三元组
    if is_duplicate:
        continue
    # 添加唯一的三元组
    unique_and_similar_triplets.append(triplet)
# 写入文件
with open('./data/final_triplets_data.txt', 'w') as file:
    file.write('\n'.join(unique_and_similar_triplets))
print('finished...')
```

因为上述代码的基本思想是将每个三元组于现有的唯一的三元组进行比较，因此该部分代码在三元组数量较大的时候（实测当三元组数量为3000时需要花费1.5h）将耗费大量时间运行。因此当需要处理的三元组数量较大时可以事先进行切分，比如1000个三元组一组执行上述代码。

经过清洗的三元组如下，可以看到相较于清洗之前的三元组来说，清洗之后的三元组质量更高。

```
天水地	出现 强降雨过程
损失 是	惨重
受灾范围	集中在	福建省
受灾地区	搜救 失踪人员
四川省雅安市汉源县万工乡境内后背山	发生 山体滑坡
此次暴雨	造成 严重损失
江西省防总升级	发布 防汛
副总理汪洋	强调 吉林省摆
强降雨突破极值	引发 山洪泥石流
局部地区	出现 洪涝灾害
湖南 拉响 防汛级应急响应
云南云龙县	遭受 洪涝灾害
河北河南两地	遭受 洪涝风雹灾害
公共服务司司长陈振林	介绍 暴雨情况
相关部门	组织 开展抗灾
徐银海	冒	大雨
强降雨	致	严峻
受灾市县	调拨 救灾物资
多方力量	参与 抢险
洪涝灾害	致	级救灾应急响应启动
```

2.4 三元组转化&知识图谱

拥有了用于构建知识图谱的三元组之后，接下来构建知识图谱的流程就相对简单，主要流程如下：

1.识别唯一实体：因为我们的三元组已经是主谓宾的结构，因此可以直接从三元组的主语和宾语位置提取所有唯一的实体，这些实体将成为知识图中的节点

2.创建节点：在知识图谱中为每个识别出的唯一实体创建一个节点，为每个节点分配一个唯一的标识符或标签

3.创建关系：将三元组中的谓词映射到知识图谱中的关系上 -- 对于每个三元组，根据谓词在相应的主语和宾语节点之间创建一个关系

（注意，知识图谱与其他数据结构不同的地方就在于两个实体之间是可以存在多种不同的关系的，而传统的图或关系数据库模型中，实体之间的关系通常局限于一种类型）

这里我们使用的数据库是图数据库neo4j，如果要手动一个个导入三元组是不现实的，因此我们借助neo4j提供的admin-import工具导入。但是admin-import工具只能导入csv格式的文件，因此在导入之前还需要对三元组格式进行转化，将txt文本格式的三元组转化为csv格式。以下是转化代码

```
# 读取三元组文件
h_r_t_name = [":START_ID", "role", ":END_ID"]
h_r_t = pd.read_table("./data/clear_triples_data.txt", decimal="\t", names=h_r_t_name)
print(h_r_t.info())
print(h_r_t.head())

# 去除重复实体
entity = set()
entity_h = h_r_t[':START_ID'].tolist()
entity_t = h_r_t[':END_ID'].tolist()
```

```
for i in entity_h:
    entity.add(i)
for i in entity_t:
    entity.add(i)
print(entity)
# 保存节点文件
csvf_entity = open("./data/entity.csv", "w", newline='', encoding='utf-8')
w_entity = csv.writer(csvf_entity)
# 实体ID，要求唯一，名称，LABEL标签，可自己不同设定对应的标签
w_entity.writerow(("entity:ID", "name", ":LABEL"))
entity = list(entity)
entity_dict = {}
for i in range(len(entity)):
    w_entity.writerow(("e" + str(i), entity[i], "my_entity"))
    entity_dict[entity[i]] = "e"+str(i)
csvf_entity.close()
# 生成关系文件，起始实体ID，终点实体ID，要求与实体文件中ID对应，:TYPE即为关系
h_r_t[':START_ID'] = h_r_t[':START_ID'].map(entity_dict)
h_r_t[':END_ID'] = h_r_t[':END_ID'].map(entity_dict)
h_r_t[":TYPE"] = h_r_t['role']
h_r_t.pop('role')
h_r_t.to_csv("./data/roles.csv", index=False)
```

上述程序运行后将会得到entity.csv文件和roles.csv文件，这两个csv文件实际上就是neo4j的三元组数据的另一种组成形式，通过组合这两个文件来组成初始化图数据库的数据。

entity.csv是实体文件，这个文件记录了实体的名称以及编号和标签数据。需要注意文件的表头需要固定

- entity:ID：实体id编号，自己生成的不重复数据编号；
- name：实体名称；
- :LABEL：标签，若存在多个标签则使用英文分号';'分割；

	A	B	C
1	Column1	Column2	Column3
2	entity:ID	name	:LABEL
3	e0	编织袋物资	my_entity
4	e1	百色市田林县利周乡平布村尾央屯	my_entity
5	e2	贵州省气象台	my_entity
6	e3	强暴雨	my_entity
7	e4	建成预警系统	my_entity
8	e5	四川省减灾委	my_entity
9	e6	拨款程序	my_entity
10	e7	救灾工作	my_entity
11	e8	具体数字	my_entity
12	e9	悲痛	my_entity
13	e10	侦查	my_entity
14	e11	救助紧急生活	my_entity
15	e12	阶段性胜利	my_entity
16	e13	医疗组	my_entity
17	e14	灾区一线	my_entity
18	e15	街道乡镇受灾	my_entity
19	e16	新疆已有88个县市相继发生多种灾害	my_entity
20	e17	应急管理部	my_entity

roles.csv文件是关系描述文件，其必须包含的表头如下

- :START_ID：对应于entity.csv中的entity:ID；
- :TYPE：关系文字；
- :END_ID：对应于entity.csv中的entity:ID；

	Column1	Column2	Column3
2	:START_ID	:END_ID	:TYPE
3	e3871	e3607	正在
4	e4444	e1915	利用
5	e2979	e1710	组织
6	e2039	e4648	遭
7	e1771	e1117	面对
8	e1543	e3713	挪到
9	e992	e4819	获悉
10	e2368	e1130	为
11	e1830	e2915	加强
12	e1496	e1857	造成
13	e449	e3795	416
14	e1951	e3174	受到
15	e2374	e3513	启动
16	e444	e4845	为
17	e49	e1306	获悉
18	e2875	e5149	提高
19	e758	e4648	遭
20	e2733	e2503	抵达

拥有上述两个csv文件后，在批量导入之前先将data目录下的databases和transactions中有关neo4j和system的文件夹全部删除（否则会报错数据库已存在，注意不要误删别的文件！），还需要注意的是桌面版的neo4j不能批量导入（至少我没找到办法），因此推荐使用社区版的neo4j。neo4j批量导入csv文件命令如下（该命令意思是将csv文件的内容全部导入到neo4j数据库中），需要在neo4j-admin文件所在的位置（在我的电脑是'D:\Download_software\My_Neo4j_zh\neo4j-chs-community-5.7.0-windows\bin'）下在终端使用如下命令（注意将yourpath替换为自己的路径）

```
neo4j-admin database import full --nodes=yourpath/entity.csv --
relationships=yourpath/roles.csv neo4j
```

我的电脑中的nodes和relationships中的路径均为'D:/My_code/Pycharm/大三下文档/KGTriples/data'，因此完整的命令为

```
neo4j-admin database import full --nodes=D:/My_code/Pycharm/大三下文
档/KGTriples/data/entity.csv --relationships=D:/My_code/Pycharm/大三下文
档/KGTriples/data/roles.csv neo4j
```

出现如下输出表示导入成功

```
IMPORT DONE in 2s 94ms.
Imported:
  2386 nodes
  4554 relationships
  4772 properties
Peak memory usage: 1.032GiB
```

导入完毕后，在相同目录下使用命令启动neo4j服务

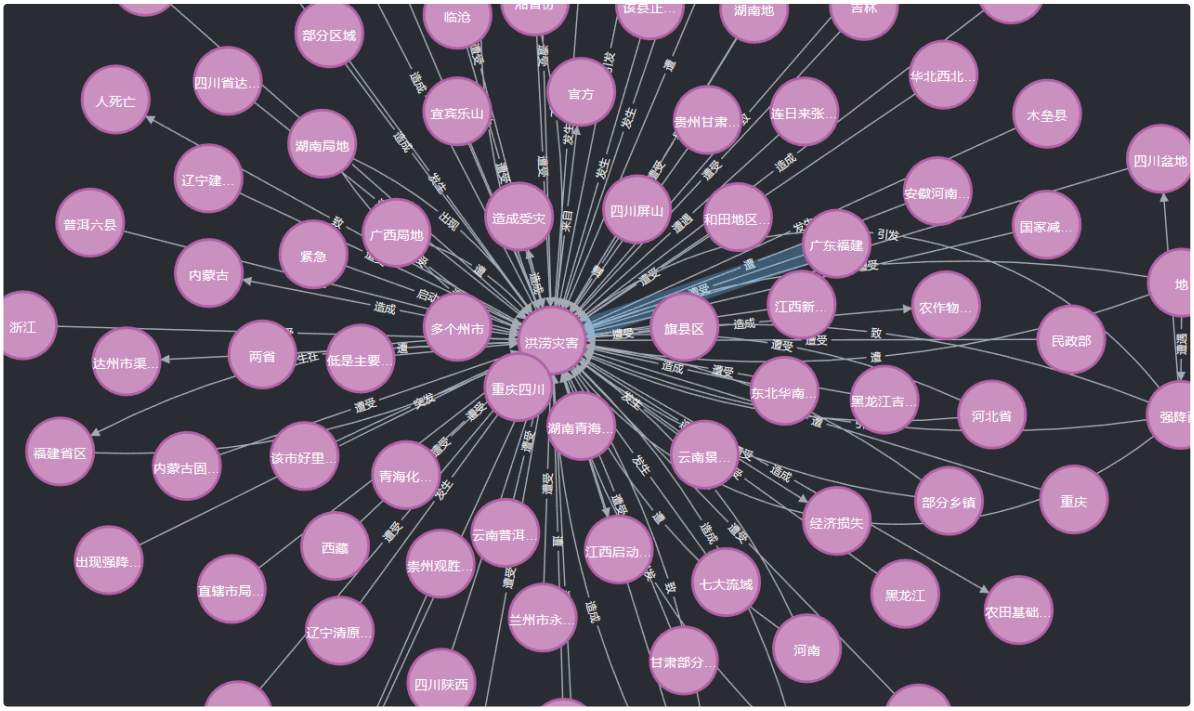
```
neo4j console
```

出现如下形式的输出表示服务启动成功，此时在浏览器网址栏输入http://localhost:7474/即可进入neo4j内置的可视化界面（首次登录应该需要账号和密码，默认都是neo4j，修改密码后请记住密码，忘记密码可能需要删除整个数据库）

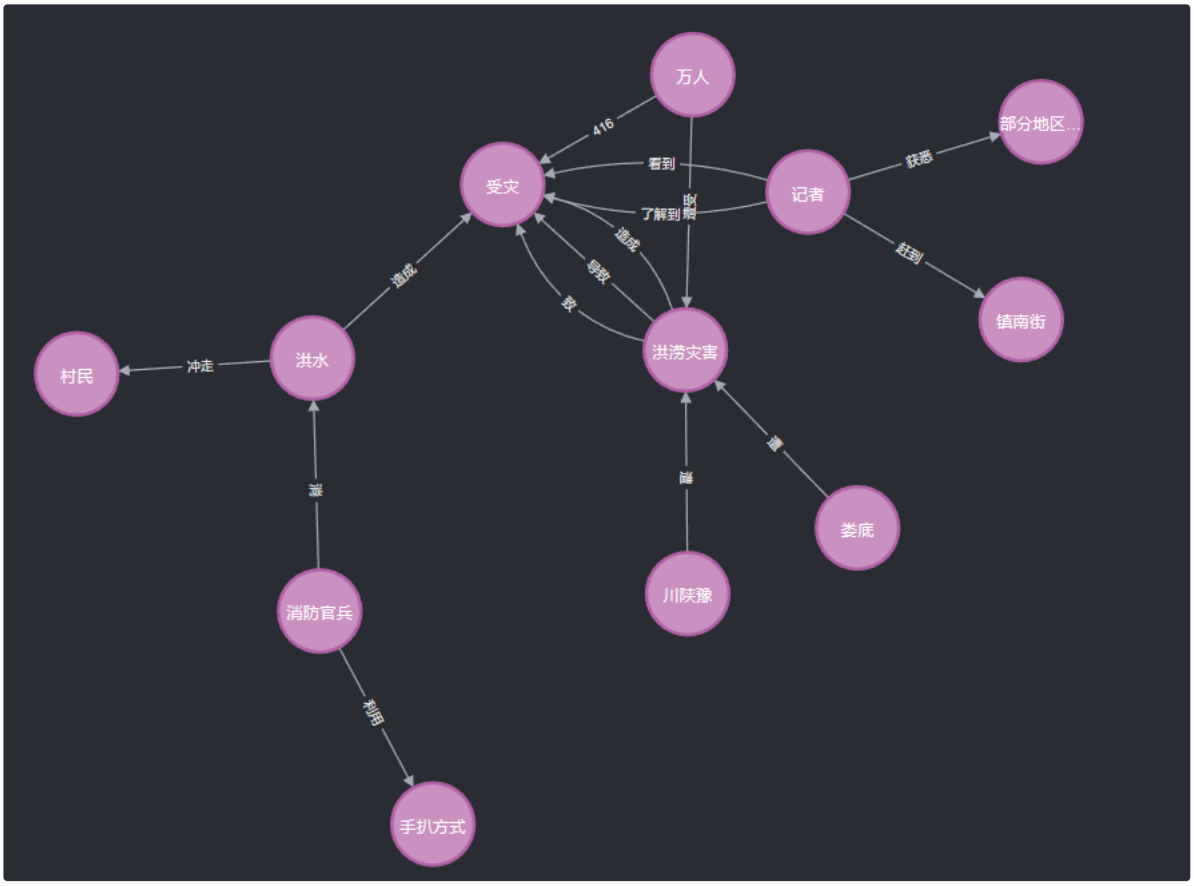
```
2023-05-18 14:26:41.344+0000 INFO  Starting...
2023-05-18 14:26:41.987+0000 INFO  This instance is ServerId{27c846df} (27c846df-8d4d-40da-b925-ba292f6600d8)
2023-05-18 14:26:42.929+0000 INFO  ======== Neo4j 5.7.0 ========
2023-05-18 14:26:44.876+0000 INFO  Bolt enabled on localhost:7687.
2023-05-18 14:26:45.584+0000 INFO  Remote interface available at http://localhost:7474/
2023-05-18 14:26:45.584+0000 INFO  id: FB3C9766D1388788B56539855BB04130786FDAB898F7F18C3235426CE51A221E
2023-05-18 14:26:45.584+0000 INFO  name: system
2023-05-18 14:26:45.584+0000 INFO  creationDate: 2023-04-23T13:01:50.015Z
2023-05-18 14:26:45.584+0000 INFO  Started.
```

Neo4j Browser提供了一个专用的图形可视化界面，使用户可以以图形形式查看节点、关系和属性。它支持各种功能，如缩放、平移、扩展和折叠节点，以及直接在界面中运行Cypher查询。Neo4j Browser中的可视化是基于存储在Neo4j中的图形数据自动生成的，使其成为探索和分析知识图的方便工具，而不需要额外的转换。相较于并不是专为知识图可视化进行设计的ECharts和Django，Neo4j Browser在使用Neo4j作为底层数据库时是知识图可视化的一个更精简的选项。

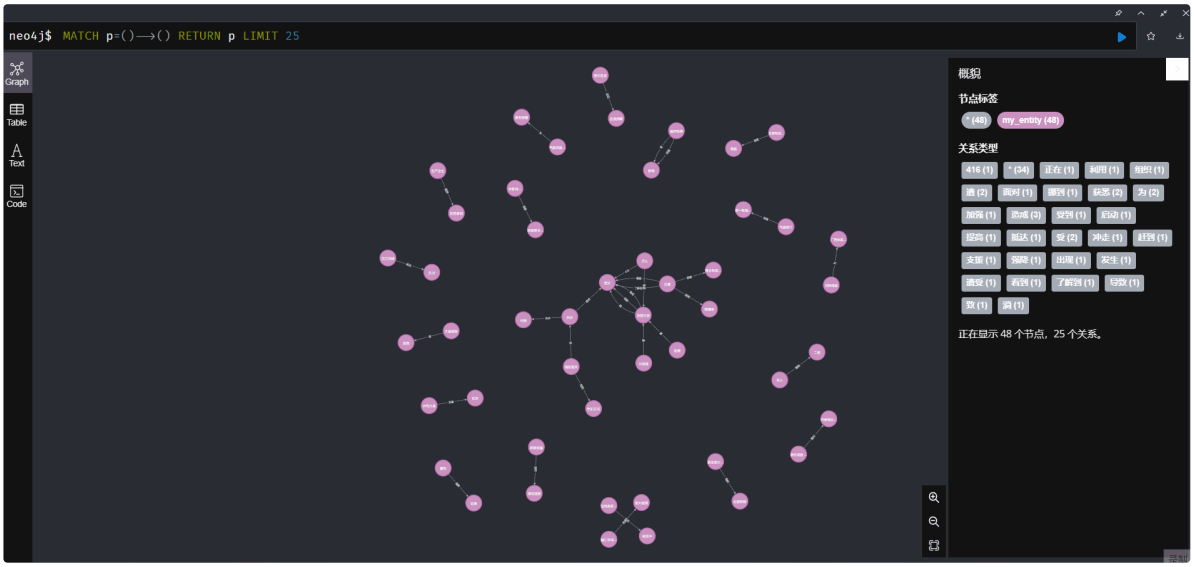
这里我们展示部分以“洪涝灾害”为中心的知识图（整个图太大了展示不完）



可以看到以洪涝灾害为中心，与其他的实体之间以不同的关系进行联系。除了上述以某个实体作为中心进行展示以外，也可以以关系网的形式进行展示



通过关系网，可以进行诸如“娄底遭洪涝灾害，导致受灾”这样简单的知识推理过程。最后我们使用一个gif来简单展示一下neo4j图数据库的魅力



三、实验总结

本次作业的基本要求是构建面向暴雨洪涝灾情的知识图谱。一开始拿到老师给的语料的时候我是不知道应该如何处理的，经过大致的了解后，我确定了基本的工作流程，最重要也是最基础的一个步骤就是获取到三元组。然而获取三元组的过程并不容易，一开始我花费大量时间，尝试使用机器学习模型进行三元组的自动提取。但是使用通用的机器学习模型在中文语料上的效果非常差劲，因此考虑是否可以针对专业领域对模型进行微调，进而使用微调过后的模型进行三元组的自动提取。然而当我兴致勃勃的将预训练模型下载并准备微调的时候才发现我压根就没有能够用于训练的三元组，这非常类似于“鸡生蛋，蛋生鸡”的问题。因此在一段时间内我陷入了迷茫，当我思考是否真的只能使用手动提取这种最原始的方式的时候，我的脑海里突然迸发出自然语言处理领域研究方法的发展过程：基于规则->基于统计->基于机器学习，既然基于机器学习不行，那么是否可以使用类似基于正则表达式匹配的方式？后来经过实验发现正则表达式的表达能力相对较弱，在后续查询资料的过程中我发现了将依存分

析和三元组提取结合的方法，这无疑是为我确定了崭新的方向。因此在这之后我依次使用了现有的多种自然语言处理工具如LTP、DDParser、OpenIE等，并最终确定使用百度的DDParser工具。对句子进行依存分析后，不能简单的提取“主谓宾”作为三元组，还需要考虑其他因素，通过阅读大佬的论文和方法使我明白了基于依存分析提取三元组并不比训练模型轻松，当然最后凭借不懈的努力我成功的完成了三元组提取、三元组清洗以及知识图谱构建等全部流程。这也启发我在之后的学习过程中遇到困难应当多方位思考，从不同的角度寻找解决问题的方法。

四、参考链接

- 知识图谱_哔哩哔哩_bilibili；
- 【完全自学知识图谱】半天我居然就学会了知识抽取实战及三大Neo4j数据库、医药问答系统、电影推荐系统基于知识图谱构建实战（人工智能AI/深度学习实战）_哔哩哔哩_bilibili；
- Neo4j构建一个简单知识图谱 - 农夫三拳有點疼 - 博客园 (cnblogs.com)；
- Candysad/neo4j: neo4j notes in Chinese (github.com)；
- Neo4j桌面版使用说明 - 知乎 (zhihu.com)（社区版无法批量导入csv数据）；
- 篇一：Neo4j图形数据库的安装与环境配置（普通版与Desktop版） - CodeAntenna；
- (2条消息) Windows 10 64位系统下Neo4j安装教程（2021.1.13）_在windows环境中安装neo4j_jing_zhong的博客-CSDN博客；
- 博客 - 微云数聚 (we-yun.com)；
- Neo4j 5.x 简体中文版指南 (we-yun.com)；
- Neo4j 5.x 简体中文版指南 (we-yun.com)；
- neo4j启动失败的解决方案 - 一杯明月 - 博客园 (cnblogs.com)；
- chatopera/Synonyms；
- 词向量训练_1；
- 词向量训练_2；
- 替换synonyms的词向量；
- 自然语言处理_中级 - Tintoki_blog (gintoki-jpg.github.io)；
- 图文并茂带你了解依存句法分析 (qq.com)；
- 开放知识图谱构建必读：封闭域VS开放知识抽取与4大类开放抽取常用方法概述 (qq.com)；
- 基于依存分析的开放式中文实体关系抽取方法 - 豆丁网 (docin.com)；
- (5条消息) 百度DDParser的依存分析_Dawn_www的博客-CSDN博客；
- HIT-SCIR/ltp: Language Technology Platform (github.com)；