2020-2021 学年
知识表示课程设计报告

任课教师：吴天星

院　系 人工智能学院

专　业 计算机科学与技术（人工智能）

组　别 第 16 组

小组成员名单

学号	姓名
09118119	黄一凡
09118113	曹思辰
09118131	唐云龙
09118120	徐浩卿
09118102	张妍
09118104	谈笑

目录

1 整体概况

1.1 设计目标

在本次知识表示课程设计中，我们基于中文维基百科，通过数据解析、类别推断、本体构建、事实抽取、数据清洗、可视化、知识补全等步骤，建立了以电影人物为主题的领域知识图谱。

1.2 构建框架

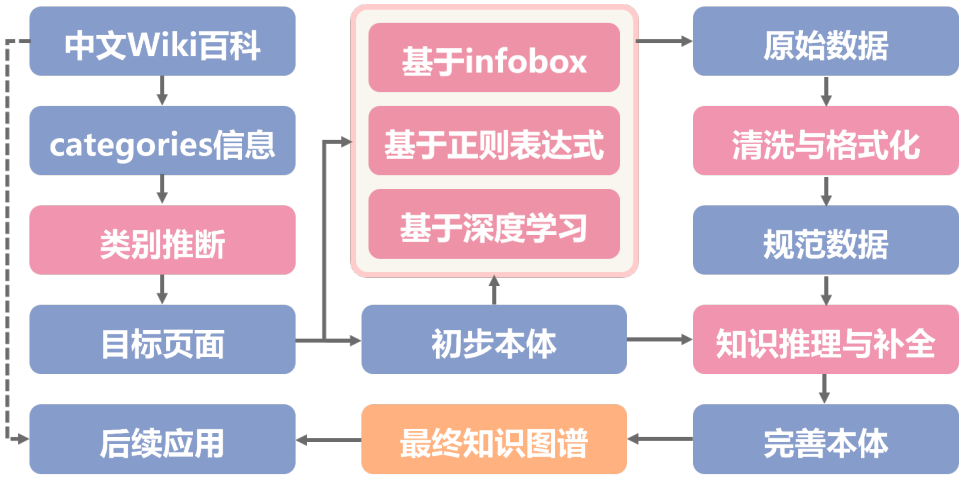


图 1: 整体框架

1.3 运行环境

环境	版本
Python	3.8.3
neo4j-community	4.1.8
apoc	4.1.0.6
neosemantics	4.1.0.0
py2neo	2021.0.1
tqdm	4.50.2
jiagu	0.2.3
regex	2020.11.13
geocoder	1.0.6
mwparserfromhell	0.6.2

表 1: 环境配置

1.4 最终结果

最终我们的知识图谱拥有 28813 个实例，727454 条三元组。由此可见，我们构造的知识图谱规模较大，完备性较好，便于后续相关应用的开发。

2 数据解析

我们主要使用中文维基百科的 dump 文件获取原始数据。由于整个数据库文件大小过大，解析时比较费时，并且我们只需要其中的一小部分页面数据，因此我们决定将其分解，每个文件保存一个页面的信息，同时将需要的页面选出。该文件是以 XML 语法构造的，可以使用一些 XML 解析工具将整个中文维基数据库分解为各页面文件。通过分析每个页面的内容，我们可以进一步筛选出构建知识图谱需要的页面文件和信息。

2.1 设计流程

首先，我们使用将中文维基的原始 dump 文件分解为各页面，之后提取每个页面的 Category 信息，找出所需的电影相关页面并保存在相应的目录中，最后使用 mwparserfromhell 匹配并提取页面的 Infobox 以供后续的事实抽取。

2.1.1 文章提取

中文维基百科中含有部分繁体汉字，为了统一和防止歧义，我们先使用 Python 的 OpenCC 模块将所有繁体中文汉字转换为简体中文。根据观察我们发现，原始 dump 文件中，每个页面数据是由 page 标签划分的，一个 page 标签即对应一个页面，因此我们计划按 page 标签将原始 dump 文件划分为各个单页面文件。由于原始 dump 文件大小比较大，使用 DOM 解析器运行较慢，我们改使用 SAX 解析器进行响应式解析，即：解析器逐字读取文件，触发器检测所读取的内容并判断单个页面的开始与结束，并保存页面。在实际实现中，我们构建了一个继承自 XMLGeneratetor 的 XMLBreaker 类用于分割 XML 文件，并实现了其 startElement、endElement 方法，用于检测 page 标签。代码中的 break_after 参数指定了读取多少数量的 page 标签即保存为一个文件，我们需要将其分割为单个页面文件，因此调用时设为了 1，即每个 page 标签保存为一个文件。保存页面文件由对象 out 管理，此处我们实现了一个简单的类 CycleFile，具有写入、新建文件等功能，提供给 XMLBreaker 使用。部分代码如下：

```
from xml.sax import parse
from xml.sax.saxutils import XMLGenerator

class CycleFile(object):
    ...

class XMLBreaker(XMLGenerator):
    def __init__(self, break_into=None, break_after=1000, out=None, *args, **kwargs):
        ...

    def startElement(self, name, attrs):
```

```
        ...

    def endElement(self, name):
        ...

filename, break_into, break_after = sys.argv[1:]
parse(filename, XMLBreaker(break_into, int(break_after), out=CycleFile(filename)))
```

2.1.2 信息提取

对于分解出并已经筛选过的每一个页面，在后续处理中需要获取页面的内容与其他信息。一般来说，一个页面含有 title、id、revision（内含 contributor、text 等）等标签，如图 2 所示。我们使用的 dump 文件为最新修订版本，因此每个页面数据中只含有一个 revision 标签，其中也只对应一个 text 标签。因此，我们可以将该 text 标签视作该页面的正文部分内容。

图 2: 页面内容示例

使用 Python 自带的 DOM 解析器，可以将 XML 文件解析为树状结构。我们从中选出页面中的 title、id、text 三个标签，将内容文本保存到字典中以获得单个页面的解析。代码如下所示：

```
from xml.dom.minidom import parse
import xml.dom.minidom

def page_extract(page_path):
    extracted_data={}
    DOMTree = xml.dom.minidom.parse(page_path)
    collection = DOMTree.documentElement

    page = collection.getElementsByTagName("page")[0]
    title = page.getElementsByTagName('title')[0].childNodes[0].data
    entity_id = page.getElementsByTagName('id')[0].childNodes[0].data
    text = page.getElementsByTagName('text')[0].childNodes[0].data

```

```
    extracted_data['title']=title
    extracted_data['id']=entity_id
    extracted_data['text']=text

    return extracted_data
```

获取页面的字典式解析后，我们发现页面中的 Infobox 存在于 text 字段的文本中。维基百科原始文本中的 Infobox 是按照维基模版 (template) 语法构建的。每种模版拥有一个名称，页面显示时按照名称来排版内容。根据人工统计，我们发现电影相关的页面 Infobox 的名称大部分为 “Infobox” 或者 “艺人”。mwparserfromhell 模块可以解析维基百科的原始文本数据，并提取出正文中含有的所有模版。我们通过正则表达式匹配名称为 Infobox 或艺人的模版，获得 Infobox 信息。

```
import mwparserfromhell

wiki = mwparserfromhell.parse(text)
matches = wiki.filter_templates()
pattern = r'(Infobox|艺人)'
infobox_matches = [match for match in matches if re.match(pattern, str(match.name)) is not None]
```

生成的模版对象已经由 mwparserfromhell 包提取出了所含有的键值对，我们将其中所有值非空的属性全部保存到字典中返回。至此，我们已经封装了对原始页面文件进行初步信息解析的过程：输入原始页面 XML 文件，输出该页面的 title、text、id 组成的字典以及该页面可能存在的 Infobox 属性字典。

2.1.3 页面筛选

获取了中文维基百科中的所有页面之后，我们需要将电影相关的页面选择出来，这样可以大幅降低数据量，加快我们的后续处理工作。在设计筛选方法时，我们发现，构建知识图谱需要使用到的页面都是关于一个实体的页面。而对于每个实体，我们都需要判断其类别以确定它在我们知识图谱中所处的位置。判断页面是否与电影相关，则相当于判断该页面内介绍的实体是否属于我们预定义的这些类别。因此，通过推断该实体的类别，即可决定是否将该页面数据留下，同时还可以根据实体的类别将页面放到不同的目录中进行区分。维基页面的类别主要通过设置的 Category 字段提供，这类字段在单个页面中可以具有多个，类似 “标签”，我们可以基于标签信息推断出其类别。基于上述思想，对于每个页面，我们首先将页面文本中的所有 “Category” 字段提取出来。定义一个正则表达式串，我们可以方便地获取这类信息：

```
import re

category=[]
pattern=r'(?<=Category:).*?(?=(]]))'
carry=re.finditer(pattern, text)
for item in carry:
    category.append(item.group())
```

之后，我们可以使用第 3 节中的类别推断方法获取该页面的类别（演员、导演、编剧）。若该页面不属于任何一个类别，则说明该页面是无关页面，可以删去；若该页面属于其中之一，则将其放入对应的目录中。

2.2 已封装函数

经过上述流程，我们对所要处理的数据格式及内容有了初步了解，并设计了算法进行初步的信息提取。由于后续步骤也需要用到此处获得的信息，我们将上述流程封装进数个函数，以便利后续调用。

```
def page_extract(page_path):
    '''
    输入：
        page_path - 字符串，需解析的XML页面文件路径（内含单个页面）。

    输出：
        extracted_data - 字典，含有'title'、'text'、'id'三个字段，分别对应标题、正文文本和ID号。
    '''
    ...

def infobox_extract(text):
    '''
    输入：
        text - 字符串，需提取Infobox的页面正文文本（由page_extract提供）。

    输出：
        properties - 列表，元素为多个字典，每个字典对应一个匹配的Infobox，字典内为Infobox的属性。
    '''
    ...
```

3 类别推断

经过数据解析与提取后，我们需要对每一个实例进行类别推断，其中最复杂同时也最关键的是对人物进行类别推断。通过先验知识，我们首先将人物大致分为四类：电影演员、电影导演、电影编剧以及普通人。接着，我们对数据解析后得到的结果进行观察，发现类别信息大多存在于 Infobox 与 category 信息中，于是对这两方面分别进行了探索与思考。

3.1 基于 Infobox 信息的类别推断

Infobox 中本身就含有类别信息，如“艺人”、“学校”等，但是直接提取该信息作为类别信息存在着许多不足：

1. 在部分页面中缺少 Infobox 信息，从而导致其类别信息的缺失；
2. Infobox 中的类别较为笼统，如“男艺人”，“艺人”等，难以从其中确定准确身份。

综合考虑以上缺点，我们决定使用更为全面、细致的 category 信息来进行类别推断。

3.2 基于 category 信息的类别判断

如图3所示，在中文维基中，对于每一个页面都存在着相应的 category 信息，这其中也蕴含着我们需要的类别信息。如从“香港电影女演员”、“一级演员”等信息中，可以推断该页面的类别为

电影演员。

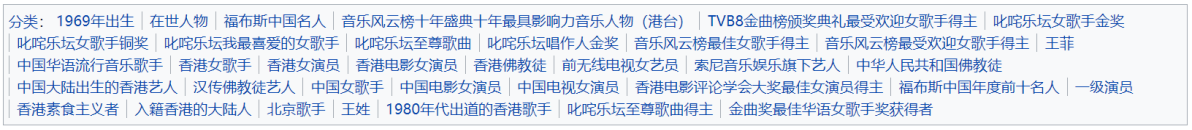


图 3: 中文维基 category 信息

然而，category 信息中还存在着许多噪声，因此，为了准确提取有效信息，过滤噪声，我们定义了如下的类别提取规则：

> 对于页面 p 的 categories 集合 $C_p = \{c_1, c_2, \ldots, c_n\}$ 与类别 i 的规则 $r_i \in R$，若存在这样一个 $c_s \in C_p$：在 c_s 中包含 r_i 中所有的关键词，且不存在这样一个 $c_t \in C_p$：在 c_t 中包含 r_i 中任意一个违禁词，那么我们认为页面 p 属于类别 i。

据此，我们建立的具体推断规则如表2所示。

表 2: 推断规则 R

类别	关键词	违禁词
电影演员	电影; 演员	协会; 处女作; 奖
电影导演	电影; 导演	协会; 处女作; 奖; 导演电影
电影编剧	电影; 编剧	协会; 处女作; 奖; 编剧电影

在制定推断规则时，我们充分考虑了 category 信息中可能出现的各种情况，建立了全面推断规则，在准确识别类别的基础上，也尽可能避免了噪声的干扰。具体实现代码如下：

```
# 利用categories信息对某个页面进行type inference
def type(page_path, key_words):
    page = page_extract(page_path)
    classes = key_words.keys()
    result = set()

    if page != -1:
        category = re_category(page['text'])
        ...
            for item in category:
                item_signal=True
                for y in ykey:
                    if y not in item:
                        item_signal=False
                        break
                for n in nkey:
                    if n in item:
                        item_signal=False
                        break
                signal=item_signal
        ...
    return list(result)
```

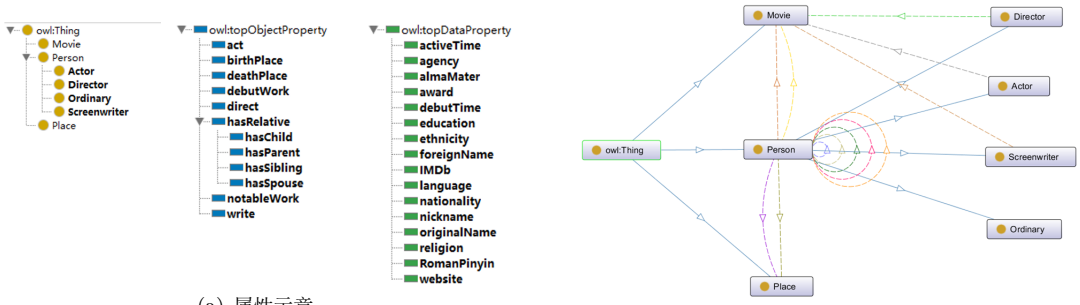
4 本体构建

4.1 初步本体构建

在进行初步本体构建时，我们采用了自底向上的构建方法。流程如下：

1. 抽取某一类（演员、导演、编剧）目标页面所有 Infobox 中的属性；
2. 对 infobox 和消歧后的各属性分别计数，记为 count{Infobox} 和 count$\{p_k\}$；
3. 对于某个属性 i，若 $\frac{\text{count}\{p_i\}}{\text{count}\{\text{Infobox}\}} > \frac{1}{3}$，我们便认为属性 i 是该类的属性。

我们首先对 Infobox 中提取出的属性进行消歧以及数量统计，据此分析出初步本体。其次，使用 protégé 对本体进行构造建模，效果如图4所示。



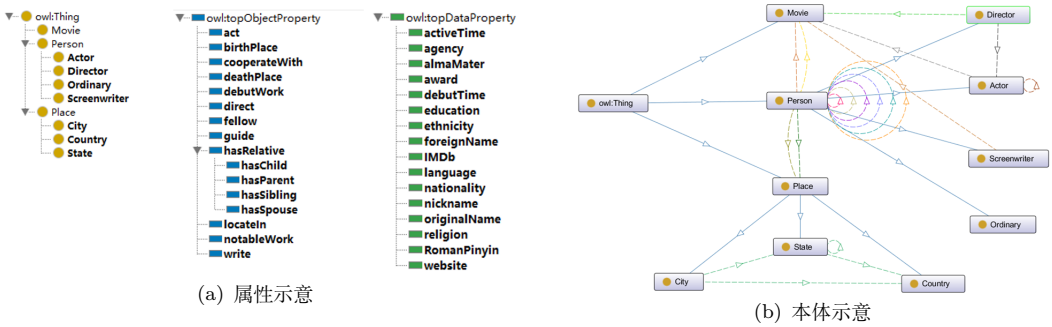
(a) 属性示意　　(b) 本体示意

图 4: 初步本体

在初步构建的本体中，共有 7 个概念，28 个属性（其中 Object Property×12，Data Property×16）。

4.2 完善本体构建

经过初步的本体构建后，我们对电影相关人物有了基础的框架与建模，经过数据清洗以及知识补全后，我们对其有了进一步的认识，由此对本体构建进一步完善，如图5所示。



(a) 属性示意　　(b) 本体示意

图 5: 完善本体

本体经过进一步的完善后，共有 10 个概念，32 个属性（其中 Object Property×16，Data Property×16）。由此可见，经过完善后，地理位置信息拥有了更为完善的层级结构，人物之间的关系也更为丰富。

5 事实抽取

事实信息的来源错综复杂，在事实抽取部分，我们主要采取三种对策来实现对于电影人物相关的知识抽取：

- **基于 infobox 的事实抽取：**根据部分页面中存在的 infobox 直接进行事实抽取。
- **基于正则表达式的事实抽取：**使用正则表达式对网页中的 text 数据进行事实抽取。
- **基于深度学习的事实抽取：**使用工具 jiagu 对网页中的 text 数据进行三元组关系预测。

5.1 基于 inforbox 的事实抽取

在多数电影人物的维基百科中，通常有 infobox 来对人物基本信息进行介绍，infobox 的常见储存信息有人物姓名、照片、出生日期等。infobox 的具体抽取代码在2.1.2中已经有过具体的介绍，在此不再赘述。如图6所示，我们截取了“王菲”中文维基百科中的部分 infobox，对其基本的个人信息进行了罗列。通过基于 infobox 的事实抽取，我们能够获得该页面可能存在的属性字典与对应事实。

图 6: 基于 infobox 的事实抽取

鉴于基于 infobox 的事实抽取是对 infobox 内现有信息的简单罗列，对 infobox 的依赖度较高。而实际中有接近 $\frac{1}{10}$ 的维基百科缺少 infobox，并且 infobox 所含有的信息也未必能满足我们的要求，这就导致了基于 infobox 的事实抽取的应用相对局限。为获取更为完善的信息，我们需要采取其他方式。

5.2 基于正则表达式的事实抽取

为了不仅仅局限于 infobox 内所给出的简单信息，我们可以对完整的网页 text 进行事实抽取。通过使用正则表达式，我们便能从非结构化的文本中提取大量的结构化信息，从而避免对大规模文本的繁琐手工操作。

表 3: 基于正则表达式的事实抽取

序号	属性	正则表达式
1	name	(?<= 叫)[\u4e00-\u9fa5]+ˆ.*?(?= （.*）)
2	originalName	(?<= 原名 | 本名 | 开始叫)[\u4e00-\u9fa5]+ (?<= 原名 | 本名 | 开始叫)[\u4e00-\u9fa5]+.*?(?=[, 。])
3	foreignName	(?<= 外文名 | 英文名 | 英语：)[a-zA-Z]+
4	nickname	(?<= 绰号 | 又叫 | 也叫 | 绰号叫 | 绰号是)[\u4e00-\u9fa5]+
5	rommanPinyin	(?<= 罗马拼音是 | 罗马拼音为 | 罗马拼音 | 罗马拼音：)[\u4e00-\u9fa5]+
6	Job	(?<= 职业是 | 职业为)[\u4e00-\u9fa5]+.*?
7	notableWork	(?<= 代表作《| 代表作品《| 知名作品《)[\u4E00-\u9FA5A-Za-z0-9_]+(?=》) (?<= 《)[\u4E00-\u9FA5A-Za-z0-9_]+?(?=》.* 代表作 |》.* 代表作品 |》.* 知名作品)
8	debutWork	(?<= 出道作《| 出道作品《)[\u4E00-\u9FA5A-Za-z0-9_]+(?=》) (?<= 《)[\u4E00-\u9FA5A-Za-z0-9_]+?(?=》.* 出道作 |》.* 出道作品)
9	activeYear	(?<= 于 | 在)[0-9]+?(?= 年.* 活跃 | 年.* 活耀)
10	debutDate	(?<= 于 | 在)[0-9]+?(?= 年.* 出道)
11	birthPlace	(?<=(出生于 | 诞生于 | 出生在)).*?(?=(, |。))
12	deathPlace	(?<= 在 | 于)[\u4e00-\u9fa5]+?(?= 逝世)
13	award	(?<= 获得)[\u4e00-\u9fa5]+? 奖 (?<= 获)[得ˆ][\u4e00-\u9fa5]+? 奖
14	language	(?<=(说 | 讲))[\u4e00-\u9fa5]+?(文 | 语)
15	website	(?<= 网站 \s?)[\w\-\.]+
16	motherSchool	(?<= 就读于)[\u4e00-\u9fa5]+?(学校?) (?<=(于 | 在))[\u4e00-\u9fa5]+?(学校?)(?= 就读)
17	education	[\u4e00-\u9fa5]+?(?= 学历)
18	religion	(?<= 信仰)[\u4e00-\u9fa5]+?
19	nationality	(?<=(出生于 | 诞生于 | 出生在)).*?(?=(, |。)) (?<=(在)).*?(?=(出生 | 诞生))
20	hasRelative	(?<=(姐 | 弟 | 哥 | 妹)).*?(?=(是 | 合)) (?<=(是)).*?(?= 的 (姐 | 弟 | 哥 | 妹))
21	hasSpouse	(?<=(跟 | 与)).*?(?=(结婚 | 相恋))
22	hasParent	(?<=(父亲 | 母亲)).*?(?=(是)) (?<=(父亲 | 母亲))[a-zA-Z]*(?=)
23	hasChild	(?<=(女儿 | 儿子 | 孩子 | 小孩)).*?(?=(出生 | 诞生 | 降临 | 是))

在本次实验中，我们总共选取了 28 个特征。对于其中的每一个特征，我们通过观察本体中各属性的特点，根据特点撰写了相应的正则表达式，来获取其中的结构化信息。忽略无法撰写正则表

达式的特征，我们在表3中罗列了“name”、“originalName”、“foreignName”等 23 个属性对应的正则表达式。尽管正则表达式能够对网页中的 text 信息进行足量的抽取，但是单单从表3即可观察到，人工撰写的正则表达式考虑的情况非常有限，无法较好应对灵活多变文本中的句式结构。所以当面对大规模文本数据时，抽取出的数据较为杂乱，且经常会抽出一整个句子。这样的结果通常会为后期的数据清洗带来一定量的麻烦。

5.3 基于深度学习的事实抽取

我们可以利用基于深度学习的自然语言处理工具——jiagu 来自动完成对网页中的 text 数据进行三元组关系预测。我们使用了1.4 亿中文知识图谱开源数据来对模型进行训练。模型训练完后，我们筛选出需要进行事实抽取的段落，并通过正则表达式过滤去冗余的符号、文字信息，便可以通过 jiagu 进行事实抽取。具体事实抽取代码如下，抽取得到的三元组结构为（头、关系、尾），将“关系”与 infobox 属性名匹配的元组中的“尾”作为属性填充进对应空缺格中。

```
def improved_extract(text):
    text = text.replace('\n','').replace('\r','')
    pattern = r"'''(.*?)=="
    match = re.search(pattern, text)
    if match is not None:
        text = match.group(1)
    pattern2 = re.compile(r'[{<](.*?)[>}]')
    text = re.sub(pattern2, '', text)
    pattern3 = re.compile(r'[\[\]\{\}\|\';,./\\&\?"]')
    text = re.sub(pattern3, '', text)
    extracted_triples = jiagu.knowledge(text)

    return extracted_triples
```

图7以阮玲玉为例，展示了 jiagu 进行事实抽取的具体结果。可以发现，虽然本段句式复杂，格式混乱，但是所抽取的三元组依旧具有很好的表现。

原文本：阮玲玉（ ），原名阮凤根、训名学名阮玉英，祖籍广东省香山县，生于上海县上海，中国无声电影默片时代演员。她是1930年代中国影坛最突出的明星之一，其优秀的演技与于24岁时自杀一事使之成为中国电影的一个时代象征。

提取出的三元组：[阮玲玉, 祖籍, 广东省香山县]
[阮玲玉, 出生地, 上海县]
[阮玲玉, 国籍, 中国]

图 7: 基于深度学习的事实抽取

6 数据清洗

在上一节内容中，我们固然已经抽取出了事实，但是事实较为杂乱，且许多信息隐藏于字符串中，难以利用。我们清洗数据方法可以主要划分为以下几种：

1. 按顿号或者逗号将输入字符串切分为多个字串；
2. 清除无效字符以及链接；
3. 删除无意义词（如“电影”、“电视剧”）；
4. 删除括号；
5. 使用正则表达式对目标内容进行匹配（如“《》”、“奖”）；
6. 对数据进行格式化；

6.1 日期数据格式化

例：’1997 年 2002 年，2010 年至今’，我们需要提取其中的数据，并将至今替换为 2021 年，对数据格式化，变作 [’1997 年-2002 年’, ’2010 年-2021 年’]。

```
def time_norm(str, flag):
    ...
    if len(str_nlist)==0:
        return []
    if flag==1:
        return [str_nlist[0]+'年']

    judge=len(str_nlist)%2
    n=len(str_nlist)//2
    for j in range(n):
        result_list.append(str_nlist[j]+'年-'+str_nlist[j+1]+'年')
    if judge==1:
        result_list.append(str_nlist[2*n]+'年-2021年')

    return result_list
```

6.2 姓名数据格式化

对于’ 小美、叶小美、青儿、牙签黄瓜’ 这样的数据条目，我们在顿号的位置将句子切割，从而将输入字符串转为列表 [’ 小美’, ’ 叶小美’, ’ 青儿’, ’ 牙签黄瓜’]。

```
def name_norm(origin_name):
    def process_single_name(name):
        # 去除多余符号
        name = name.strip().replace('-{', '').replace('}-', '')
        if name=='': return None
        pattern = r'(.+?)([( (].*?[)) ])' # 去除括号
        match = re.search(pattern, name)
```

```
        if match is not None:
            name = match.group(1).strip()
        return name
        ...
    return list(set(result))
```

6.3 作品数据格式化

对于' 月满西楼（1968 年）\n 庭院深深（1971 年）'，我们需要提取《月满西楼》和《庭院深深》两个作品，首先我们要删除其中的无效字符和括号，最后加上书名号并进行格式化。

```
def works_norm(work_list):
    def process_work_list(work):
        ...
        pattern = r'[\< 《](.*?)[\》 ]'
        match = re.findall(pattern, work)
        if len(match):
            whole_list = match
        else:
            kuohao_pattern = r'(.+?)([( (].*?[) ])'
            kuohao_match = re.findall(kuohao_pattern, work)
            # print('kuohao',kuohao_match)
            if '、' in work:
                whole_list= work.split('、')
            ...
            else:
                whole_list=[work]

        for i in range(len(whole_list)):
            whole_list[i] = ' 《'+ whole_list[i]+ '》 '
        return whole_list
    ...
    result = process_work_list(work_list)
    return list(set(result))
```

格式化前出演的电影名称杂糅在一个字符串内，但是格式化之后会将每一部电影分开，从中可以发现演员间的合作关系。在图6.3中，我们可以看到，原来张卫健和聂远出演的《少年张三丰》《西游记》《大帅哥》《倩女幽魂》《三国》五个作品仅仅被划分为两个实体，这显然不对，通过格式化之后，每个作品被单独划分出来，我们从中可以提取出演员的合作关系。

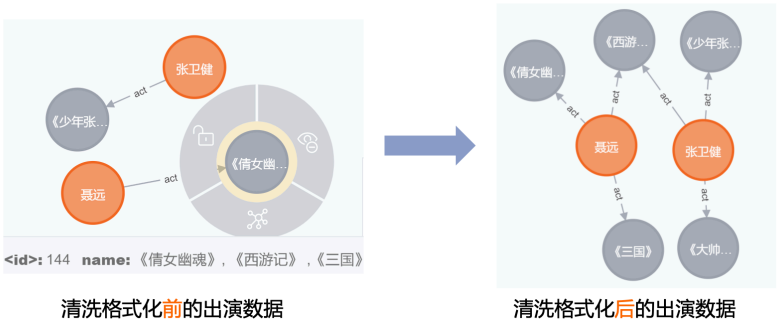


图 8: 清洗前后的出演数据

6.4 奖项数据格式化

对于奖项数据，由于奖项的名字千奇百怪，我们从中寻找到了一定的规律，选择使用正则表达式提取内容中含“奖”和“最佳”的内容。如从’土星奖最佳男主角 1996 年《杀出个黎明》国家评论协会最佳男主角 2007 年《全面反击》’中提取出 [’土星奖最佳男主角’,’国家评论协会最佳男主角’]。

```
def award_norm(award_list):
    def process_award_list(award):
        if award=='':
            return []
        # 判断是否包含中文
        pattern0 = r'[$\backslash$4e00-$\backslash$9fa5]'
        match0 = re.search(pattern0, award)
        if not match0:
            return []
        # 去除括号和书名号中的内容
        ...
        # 去除年份
        ...
        # 删去多余符号
        ...

        whole_list = award.split()
        useless = ['的奖', '颁奖', '项奖', '次奖', '等奖',
                   '此奖', '奖项', '获奖', '该奖', '之奖',
                   '个奖', '冠军', '季军', '亚军', '个']
        useful = ['奖', '最', '小姐', '先生']
        ...
    result = process_award_list(award_list)
    return list(set(result))
```

6.5 地理数据格式化

对于地理数据，我们选择使用工具 geocoder，并利用经纬度将抽取出的地理位置转化为标准格式，并以字典形式返回，便于后续的查询和推理应用。

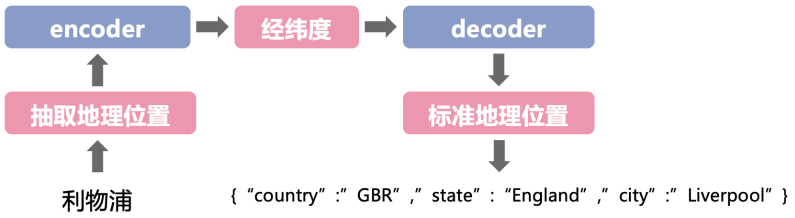


图 9: 地理数据格式化过程

格式化前的地理数据只是简单的字符串，包含的信息较少，但是结构化之后的地理数据是结构化的字典，拥有着更复杂的依赖关系，包含更多的信息。在图6.5中，原本的地理数据中地点依赖关系比较简单，通过格式化之后，节点之间的依赖关系更加丰富，有利于我们获得更多的信息。

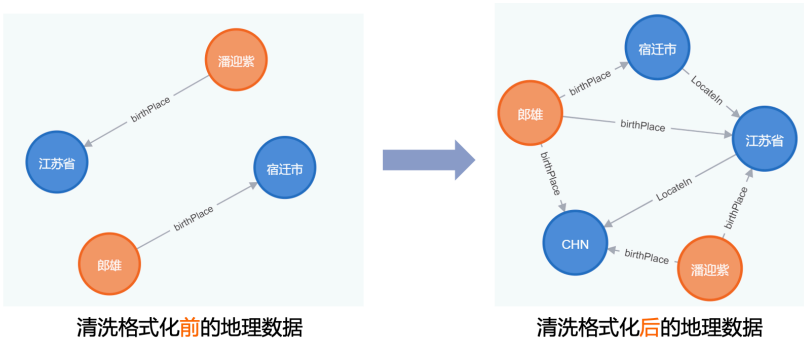


图 10: 清洗前后的地理数据

7 图谱可视化

本项目使用 neo4j 进行图谱的可视化，在 python 中借助 py2neo 库编写 cypher 语句，实现对图谱的操作。

7.1 设计流程

输入文件是一个 csv，每行表示一个实例，每列表示一个属性或者关系。对所有人物，首先将实例及其属性全部导入进 neo4j，而人物关系则在全部导入后进行匹配连接。

7.1.1 导入实例

在属性中，type 属性表征了实例所属类别，用一个列表表示。在创建节点时，读取出 type 列表中的所有元素，并作为节点的 label，表示该节点属于多个类。同时，给每个人物节点添加 person 这一 label，便于人物的查找。

对于 notableWork、debutWork 和 works 这三个属性，在创建人物节点的同时也会创建这些作品节点，并以 notableWork、debutWork 和根据人物职业而定的 act/direct/write 来命名人物与

作品之间的关系。这些作品节点拥有一个 movie 的 label 以及对应作品名的 name。因为作品节点只有作品名一个属性，不需要考虑新创建相同节点时前后属性不同而导致的节点覆盖问题，所以可以在人物节点创建时一并创建作品节点。同时使用 merge 方法创建节点可以避免重复创建相同节点。

```
def works_n(dic,graph):
    dic_help={'actor':'act','director':'direct','screenwriter':'write'}
    keysss=['actor','director','screenwriter']
    works=dic['notableWork']+dic['debutWork']
    for i in keysss:
        if i in dic['type']:
            for j in works:
                graph.run("MATCH(a:Person), (n:Movie) \
                        WHERE a.name=['%s'] AND n.name='%s' \
                        CREATE(a)-[r:%s]->(n)" % (dic['name'][0],j,dic_help[i]))
```

birthPlace、deathPlace 和 nationality 都为地点，在 csv 中以字典形式存储。包含三个键：country、state 和 city。地点节点同样也会在人物节点创建时一并创建。对于一个地点，创建三个节点分别对应于三个键值对。city 与 state、state 与 country 之间用 locatein 关系进行连接，人物节点与这三个节点都进行连接（便于查询）。地点节点的 label 为 place 及其对应的键，name 为对应的值。如 name 为‘南京’的节点的 label 为：place：city。在后续知识补全中会利用 city 节点来补全同一城市的老乡关系。

```
def place(place_type,dic,graph):
    b_place=dic[place_type]
    if b_place=={}:
        return
    if b_place['country']!='':
        graph.run("MERGE(p:Place:country{name:'%s'})"%b_place['country'])
        graph.run("MATCH(a:Person), (n:Place:country) \
                WHERE a.name=['%s'] AND n.name='%s' \
                MERGE(a)-[r:%s]->(n)" % (dic['name'][0],b_place['country'],place_type))
    ......
```

其他属性均以列表形式，作为 Data Property 的值。

7.1.2 创建关系

在所有人物实例导入完成后，进行人物关系的导入。共有四种人物关系：hasSibling、hasSpouse、hasParent 和 hasChild。若关系对应的 object 已经被创建，则将两个节点相连接。若未被创建，则为该 object 创建一个 label 为 person、属性只有 name 的节点（普通人），再进行连接。

四个关系的创建流程相同。首先用 merge 创建 object 节点（存在则合并节点，不存在则新建），再用 match 进行匹配，用 create 连接关系。

```
def person(relation,dic,graph):
    person=dic[relation]
    for i in person:
        graph.run("MERGE(p:Person{name:['%s']})"%i)
        graph.run("MATCH(a:Person), (n:Person) \
```

```
6                 WHERE a.name=['%s'] AND n.name=['%s'] \
7                 CREATE(a)-[r:%s]->(n)" % (dic['name'][0],i,relation))
```

7.2 效果展示

图 11: 最终效果图示例

可以看到，梅小惠与与她的作品之间都有'act' 关系，同时这些作品也是她的成名作。对于籍贯和出生地，将梅小惠与地点的'country'，'state' 和'city' 都进行了连接，地点之间也建立了 LocateIn 关系。梅小惠与她的亲戚们也根据亲属关系进行了连接。

8 知识补全

经过事实抽取、数据的格式化和清洗，我们可以根据已有的图谱建立新的关系，从而进一步提高知识图谱的完备性，并为今后可能的应用奠定基础。

8.1 设计流程

本项目在从 csv 导入到 neo4j 的知识图谱中利用 cypher 语句完成知识补全，一共补全了 3 种关系，分别为老乡关系（fellow）、合作关系（cooperateWith）、指导关系（guide）。并且同样利用 cypher 语句删除重复关系（保证准确性），最后利用 cypher 语句导出 rdf 三元组。

8.2 补全规则

我们根据三种关系的语义分别设计了三种规则来做知识补全：

- 若两个演员参演了同一部电影，即演员 1 和某电影存在参演 (act) 关系并且同时演员 2 和此电影存在参演 (act) 关系，则在两个演员之间添加’cooperateWith’关系。
- 若导演和演员共同参与了一部电影，即某导演和某电影存在导演 (direct) 关系并且同时，某演员和此电影存在参演 (act) 关系，那么这位导演与演员之间创建’guide’关系。
- 若两个人出生地相同，且此出生地的类别是 city，则在两个人之间添加’fellow’关系。

并且在补全中，利用 merge 语句，对已经存在的关系做到不会重复添加。补全效果示例如图所示：

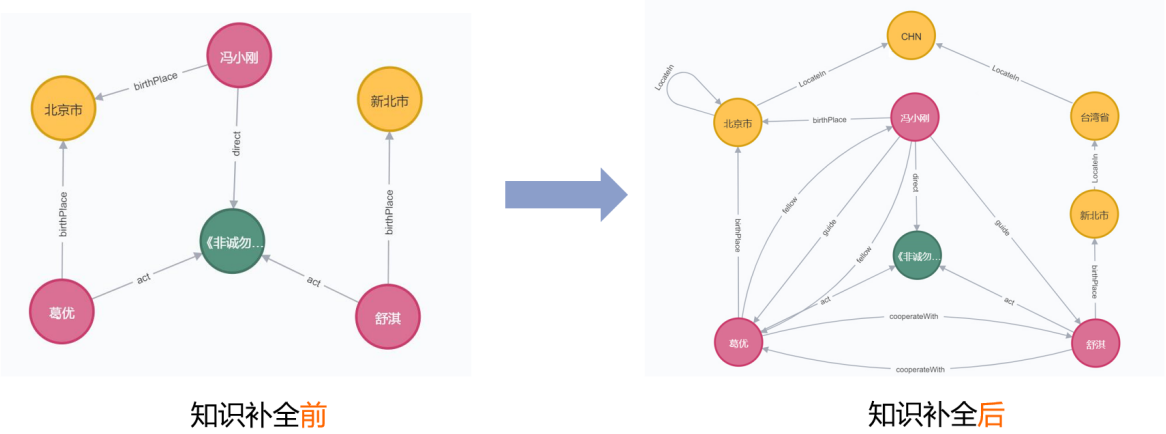


图 12: 知识补全效果图示例

可以看到在图中补全前，演员葛优和演员舒淇都参演过 (actor) 电影《非诚勿扰》，所以给他们互相添加合作 (cooperateWith) 关系；另外导演冯小刚导演了这部电影，所以创建冯小刚对葛优和舒淇的指导 (guide) 关系；以及冯小刚和葛优都出生在 (birthPlace) 北京市，且北京市是 city 类别的，因此给他们之间互相创建老乡 (fellow) 关系。

8.3 cypher 语句

这是我们第一次在 neo4j 里使用 cypher 语句，所以也在一开始使用的时候踩到了一些坑：比如查找匹配的条件放在 WHERE 里导致匹配速度过；在配置 neo4j 的插件时，根据 neo4j 版本配置了

相应的 apoch 和 neosemantics，但是在 neo4j.conf 文件里漏加”dbms.unmanaged_extension_classes=n10s.endpoint=/rdf” 导致在最后导出 rdf 时找不到网页。

解决完这些问题后，我们根据补全的规则，设计了三种不同的 cypher 语句。

补全 cooperateWith:

```
MATCH (a1:actor)-[:act]-(m:Movie),(a2:actor)-[:act]-(m:Movie)
WHERE  a1<>a2
MERGE (a1)-[r:cooperateWith] -> (a2)
```

补全 guide:

```
MATCH (d:director)-[:direct]-(m:Movie),(a:actor)-[:act]-(m:Movie)
WHERE  d<>a
MERGE (d)-[r:guide] -> (a)
```

补全 fellow:

```
MATCH (p1:Person)-[:birthPlace]-(c:city),(p2:Person)-[:birthPlace]-(c:city)
WHERE p1<>p2
MERGE (p1)-[r: fellow] -> (p2)
```

并且在完成 3 个关系的补全后，利用以下 cypher 语句删除重复关系进一步保证准确性：

```
MATCH (a)-[r]->(b)
WITH a, b, TAIL (COLLECT (r)) as rr
FOREACH (r IN rr | DELETE r)
```

最后，利用 cypher 语句导出 rdf 作为输出：

```
:POST /rdf/neo4j/cypher { "cypher":"MATCH (n)-[r]-(m) RETURN *","format": "RDF/XML"}
```

9 小组分工

黄一凡：

- 使用 SAX 解析器将 dump 文件拆分为单个页面，便于后续处理
- 使用轻量化的 DOM 解析器，从单个页面中提取 id，title 以及 text 信息
- 建立类别推断规则，并根据规则编写相关推断代码，从而将各页面分类并提取目标页面
- 根据统计的 Infobox 属性进行属性消歧，并据此进行自底向上的本体构建
- 使用 protégé 对本体进行建模，并根据数据清洗和知识补全的结果对本体进行完善
- 编写事实抽取框架代码。使其可以同时支持 Infobox 抽取、正则表达式抽取以及深度学习方法抽取
- 基于 geocoding 进行地理数据格式化，令原先模糊歧义较多的地理位置转为准确清晰的 country-state-city 三层结构
- 编写数据格式化框架代码，支持所有属性的格式化操作

曹思辰：

- 用 opencc 对原 wiki 数据集做繁简转化。
- 编写正则表达式,在文本内容里抽取了 nationality、ethnicity、agency、hasRelative、hasSpouse、hasParent、hasChild 这六个关系
- 观察了抽取后数据中的亲属关系，即 hasRelative、hasSpouse、hasParent、hasChild 这四个关系，编写了相应的规则，对这四个关系经行数据清洗和格式化。
- 根据 neo4j 的官方教程配置了 neo4j+apoch+neosemantics 的环境，以便之后在 neo4j 中运行 cypher。
- 编写了知识补全中的 cypher 语句，实现了在 neo4j 中对知识图谱的查询、添加和删减。定义了知识补全中三个关系的补全规则，并且利用 cypher 语句完成了知识补全，并且最后导出为 rdf。

唐云龙：

- 观察 activeTime 和 debutTime 两个属性，编写规则对这两个属性进行数据清洗和格式化。
- 对数据清洗后的 csv 中含有特殊符号的属性值进行格式化，以便输入 cypher 语句中顺利运行。
- 使用 py2neo 库对 neo4j 中的图谱进行可视化操作，编写 cypher 语句将 csv 中的实例和关系导入到 neo4j 中。并根据需求调整可视化结构。

徐浩卿：

- 使用 mwparserfromhell 提取文本中的 Infobox 及其所含属性
- 根据提取出的 Infobox 属性，对每个类别（演员、导演、编剧）统计具有每种属性的实体所占比例，供后续构建本体使用
- 编写正则表达式，从正文文本中抽取 deathPlace、award、language、website、motherSchool、education、religion 这 7 个关系
- 根据观察 name、originalName、foreignName、nickName 四个关系已提取的数据的特点，使用基于正则表达式以及字符串解析的算法进行数据清洗和格式化
- 修改事实抽取代码，加入 jiagu 抽取部分并统计各方法抽取结果数量，确认抽取效果
- 运行事实抽取代码得到待清洗的原始图谱数据文件

张妍：

- 编写正则表达式，从正文文本中抽取 name、originalName、foreignName、nickname、romanPinyin 这 6 个关系
- 考察使用 openUE 和 deepKE 进行事实抽取的可行性
- 观察 notableWork 与 debutWork 几个关系已提取的数据特点，使用基于正则表达式以及字符串解析的算法，对其进行数据清洗和格式化

谈笑：

- 编写正则表达式，从正文文本中抽取 job、notableWork、debutWork、activeYear、debutYear、birthPlace 这 6 个关系
- 考察使用 openUE 和 deepKE 进行事实抽取的可行性
- 观察 award 与几个关系已提取的数据特点，使用基于正则表达式以及字符串解析的算法，对其进行数据清洗和格式化