知识图谱问答系统对比实验
Seq2Seq 模型，基于 NER,RE 的三元组提取模型

按学号排序

人工智能学院　12 班　蔡嘉骏　2020212287

人工智能学院　12 班　陈东升　2020212288

人工智能学院　12 班　李沅昕　2020212295

2023 年 7 月 3 日

目录

摘要

本项目为 dbpedia2016 的知识图谱问答系统，用户在网页输入自然语言问题，输出该问题对应的在知识图谱里的答案词条。本项目用两种模型实现了该目标，一种是直接生成 SPARQL 的 Seq2Seq 模型，一种是使用 NER+RE 进行实体关系抽取，组装成三元组，并使用 ElasticSearch 搜索实体关系，最终组装成 Sparql 的三元组提取模型。

1 项目介绍

1.1 项目简介

本项目对比了多种模型在知识图谱问答系统的效果，其中包括基于 Encoder-Decoder 的 Seq2Seq 直接输出 SPARQL 模型，和基于 POS_Tagging 的 NER 和 RE 任务的图神经网络的三元组抽取模型，间接转化为 SPARQL 语句模型。

通过对比两种模型在 bleu 指标的优劣，说明每个模型的优缺点。本项目使用了多种模型，包括 Encoder-Decoder 模型,POS_Tagging 模型和图神经网络模型等，编写了网页前端，运用了多种数据处理引擎，包括 ElasticSearch 和 Neo4j 图数据库等

1.2 小组分工

各小组成员贡献比相同

- 蔡嘉骏：负责前端编写，ElasticSearch+Neo4j 大数据处理, 使用 BERT 模型完成 RE 任务，将三元组进而转化为可执行 sparql 或 cypher, 尝试构建图神经网络进行实体关系抽取
- 陈东升：负责 LSTM+CRF+viterbi 模型将自然语言问题通过序列标注进行 NER 任务，标注实体
- 李沅昕：负责 Encoder-Decoder 的 Sep2Seq 模型对自然语言直接输出 sparql 或 cypher

1.3 使用知识点

使用到的课内知识点

1. **BiLSTM**: 使用 LSTM 对变长序列输出句子向量
2. **CRF+Viterbi+Beam Search**: 使用 CRF+Viterbi+Beam Search 对输出句子向量进行编码解码
3. **Encoder+Decoder**：解决 Seq2Seq 问题，使用该模型直接输出 sparql
4. **TF-IDF 和 BM25**:ElasticSearch 的倒排索引实现原理, 在 Elasticsearch 中，每个查询都被转换成一个词项列表，计算其 TF 和 IDF。BM25 在 TF-IDF 的基础上, 加入了文档长度的归一化因子, 通过考虑更多信息并进行归一化处理, 获得了更为准确的相关度评分。使用到的其他知识点
   (a) **网页 +Flask 框架**: 网页展示

(b) **ElasticSearch 作为搜索引擎**: 作为实体识别的模糊搜索，使用 BM25 相似度进行计算

(c) **Neo4j 作为图数据库**: 作为 dbpedia2016 的本地数据库和可视化工具

1.4 文件架构

- NLP2SPARQL 文件夹负责该项目的 Seq2Seq 模型
- NER_BiLSTM+CRF.py 负责 NER 模型
- BERTRelationExtraction 文件夹负责 RE 模型
- elasticsearch 文件夹负责 elasticsearch 处理
- neo4j 文件夹负责 neo4j 图数据库
- front_end.py 负责前端后端通信
- template 文件夹负责存放静态网页文件

1.5 部署步骤

1. 从 https://www.rdfhdt.org/datasets/dbpedia 下载 dbpedia2016.hdt(大小 14g) 文件
2. 用 https://github.com/rdfhdt/hdt-cpp.git 克隆 hdt 处理库，使用其 hdt2rdf 命令将 hdt 文件转化为 N-triples 文件 (大小 150g), 并使用 dumpDictionary 命令将该 hdt 文件转化为 label-uri 的文本文件
3. 下载 neo4j 图数据库，安装 n10s 插件，使用其命令将该 N-triples 文件存储到 neo4j 图数据库中
4. 使用提供的 elasticsearch_mapping.py 将该 label-uri 的文本文件索引到 elasticsearch 中，分别创建实体索引 (dbpedia201604e) 和谓词索引 (dbpedia201604p)
5. 运行 front_end.py，将自动下载本项目要用到的预训练模型，最终能在 localhost:5000 看到本项目网页

1.6 项目结构

- Seq2Seq 模型：用 LC-QuAD 数据集进行训练，使用 Encoder-Decoder 结构，输入自然语言，直接输出 Sparql 语句。该模型作为对比模型
- 基于 POS_Tagging 模型：
  - NER 模型：使用数据 CoNLL-2003 进行训练，使用 BiLSTM+CRF 模型
  - RE 模型: 使用 CNN 数据集进行训练, 使用带 Entity Marker 和 Mention Pooling 的 Encoder-Decoder 结构。

2 数据处理

2.1 数据集

- dbpedia2016:dbpedia2016 是一个包含维基百科信息的大型知识库，它包含了大量的实体、属性和关系。**该数据集用于索引到 ElasticSearch 搜索引擎和存储到 Neo4j 图数据库中，建立知识图谱**

- LC-QuAD-1.0：LC-QuAD-1.0 数据集是用于问答系统评估的数据集，其中问题和答案都涉及到 Linked Data 资源（包括 RDF 图、Ontology 和实例数据）。该数据集包含 5000 个问题和答案对，覆盖了 14 个主题，包括人物、组织机构、电影、音乐等。**该数据集用于 Seq2Seq 模型，图神经网络**

- conllp-2003 conllp-2003 是一个用于命名实体识别（NER）任务的数据集。它由路透社语料库中的新闻文章组成，包括四种语言：英语、西班牙语、荷兰语和德语。该数据集用四种类型的实体进行了标注：人物（PER）、地点（LOC）、组织（ORG）和杂项（MISC）。**该数据集用于训练用于 NER 的 BiLSTM+CRF 模型**

2.2 ElasticSearch 数据处理

下载 dbpedia2016, 将其 label 和 uri 索引到 ElasticSearch 中。

ElasticSearch 是一个高度可扩展的开源全文搜索和分析引擎。它允许用户快速地存储、搜索和分析大量数据。ElasticSearch 通常用于日志和事件数据分析、全文搜索和其他数据处理场景。

我们对 ElasticSearch 的 Setting 部分进行如下设置

- number_of_shards: 分片数。这里设置为 4，表示使用 4 个分片，分别负载在每个组员的电脑上。分片可以帮助提高数据存储和查询的性能。将一个索引拆分成多个分片可以实现数据的水平拆分，使得数据分布在多个节点上。这有助于提高查询性能，因为查询可以并行地在多个分片上执行。

- number_of_replicas: 副本数。这里设置为 1，表示使用副本数量为 1。副本可以提高数据的可用性和查询性能。在查询时，Elasticsearch 可以根据负载均衡策略选择使用哪个副本分片。

- max_ngram_diff: 最大 n-gram 差值，这里设置为 50。此设置用于限制 n-gram 分词器中最大和最小 n-gram 之间的差值。将文本划分为长度为 n 的连续字符序列。例如，假设我们设置最小 n-gram 为 3，最大 n-gram 为 6，那么最大 n-gram 差值就是 3（6-3）。这意味着在文本分析过程中，我们会将输入文本分割为长度为 3、4、5、6 的连续字符序列。这有助于提高搜索的准确性和召回率，因为较短和较长的 n-gram 可以捕捉到不同程度的文本相似性。然而，最大 n-gram 差值过大可能导致索引体积增加和查询性能下降。

- analysis: 定义自定义分析器和分词器。
  - analyzer: 定义四个自定义分析器：
    - default_analyzer: 默认分析器，使用 label_tokenizer 分词器和 lowercase、asciifolding 过滤器。
    - snowball_analyzer: 使用 label_tokenizer 分词器和 lowercase、asciifolding、snowball 过滤器。
    - shingle_analyzer: 使用 label_tokenizer 分词器和 shingle、lowercase、asciifolding 过滤器。
    - ngram_analyzer: 使用 ngram_tokenizer 分词器和 lowercase、asciifolding 过滤器。
  - tokenizer: 定义两个自定义分词器：
    - label_tokenizer: 空白字符分词器，按空格分词。
    - ngram_tokenizer: n-gram 分词器，根据设置的最小和最大 n-gram 进行分词，这里为 3 和 6。

在创建索引时，我们需要定义索引的映射。映射定义了索引中的字段和它们的类型。在本实验中，我们将使用以下映射：

- uri 字段：该字段是一个 keyword 类型，用于存储实体的 URI。这个字段适用于精确匹配查询，不会进行分词处理。

- id 字段：该字段是一个 keyword 类型，用于存储实体的唯一 ID。这个字段适用于精确匹配查询，不会进行分词处理。
- label 字段：该字段是一个 text 类型，用于存储实体的标签，并根据映射中的 analyzer 设置进行分词处理。此字段包含四个子字段：
  - label：使用 default_analyzer 进行分词处理，采用 BM25 相似度算法。
  - snowball：使用 snowball_analyzer 进行分词处理，采用 BM25 相似度算法。这个分析器还包括一个 snowball 过滤器，用于词干提取。
  - shingles：使用 shingle_analyzer 进行分词处理，采用 BM25 相似度算法。这个分析器还包括一个 shingle 过滤器，用于生成词语的组合。
  - ngrams：使用 ngram_analyzer 进行分词处理，采用 BM25 相似度算法。这个分析器还包括一个 ngram 分词器，用于生成词语的 n-gram。

```
curl -X PUT "localhost:9200/dbpedia201604p" -H 'Content-Type: application/json' -d'
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0,
    "max_ngram_diff" : 50,
    "analysis": {
      "analyzer": {
        "default_analyzer": {
          "type": "custom",
          "tokenizer": "label_tokenizer",
          "filter": ["lowercase", "asciifolding"]
        },
        "snowball_analyzer": {
          "type": "custom",
          "tokenizer": "label_tokenizer",
          "filter": ["lowercase", "asciifolding", "snowball"]
        },
        "shingle_analyzer": {
          "type": "custom",
          "tokenizer": "label_tokenizer",
          "filter": ["shingle", "lowercase", "asciifolding"]
        },
        "ngram_analyzer": {
          "type": "custom",
          "tokenizer": "ngram_tokenizer",
          "filter": ["lowercase", "asciifolding"]
        }
      },
      "tokenizer": {
        "label_tokenizer": {
          "type": "whitespace"
        },
        "ngram_tokenizer": {
          "type": "ngram",
          "min_gram": 3,
```

```
            "max_gram": 6,
            "token_chars": ["letter", "digit"]
          }
        }
      }
    },
    "mappings": {
      "properties": {
        "label": {
          "type": "text",
          "fields": {
            "label": { "type": "text", "similarity": "BM25", "analyzer": "default_analyzer" },
            "snowball": { "type": "text", "similarity": "BM25", "analyzer": "snowball_analyzer" },
            "shingles": { "type": "text", "similarity": "BM25", "analyzer": "shingle_analyzer" },
            "ngrams": { "type": "text", "similarity": "BM25", "analyzer": "ngram_analyzer" }
          }
        },
        "label_exact": { "type" : "keyword" },
        "uri": { "type" : "keyword" },
        "id": { "type" : "keyword" }
      }
    }
  }
  '
```

2.3 Neo4j 数据处理

将 dbpedia2016 转化为 N-triple 文件，使用 ns10 插件将 nt 文件导入 neo4j 里，效果如下

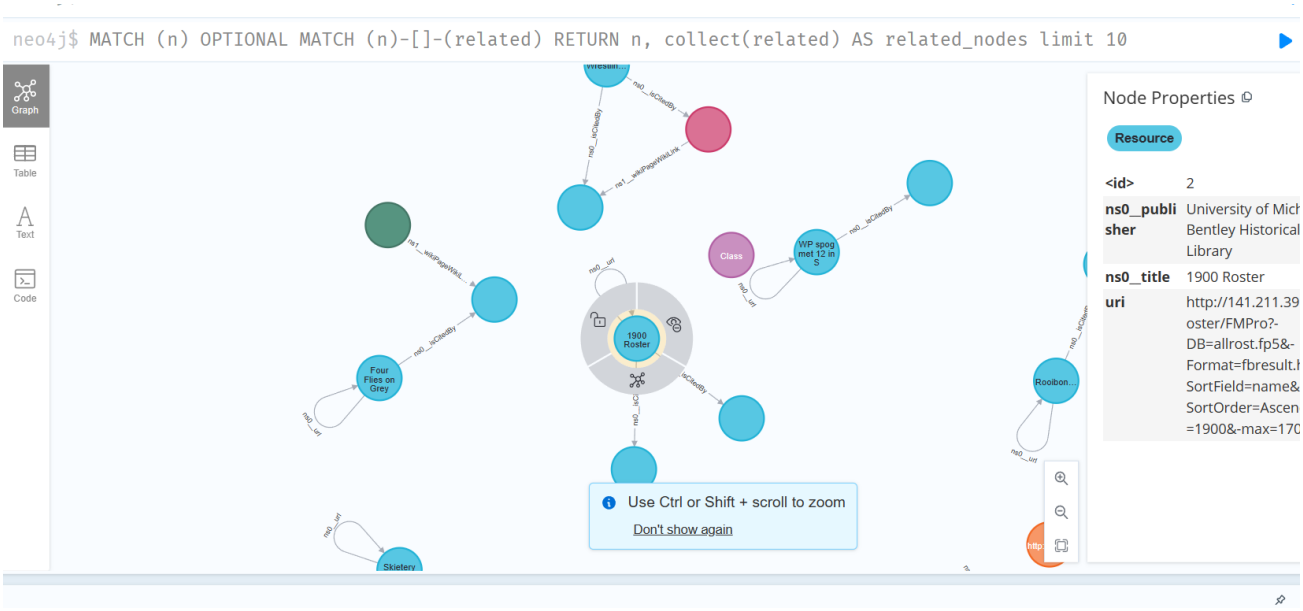


图 1: neo4j 网页展示

3 Seq2Seq 模型

3.1 模型结构与原理

3.2 Encoder-Decoder 结构

这个模型由编码器 (encoder) 和解码器 (decoder) 组成。编码器可以是 BERT 或 RoBERTa 等, 解码器可以是 Transformer 等。在本项目中采用都是如下参数

Encoder：基于 bert-based-cased 的预训练大模型。bert-based-cased 模型是一种基于 BERT 的语言模型，它在英语语料上使用了掩码语言建模（MLM）的目标进行了预训练。它在这篇论文中被提出，并首次在这个仓库中发布。这个模型是区分大小写的：它能够区分 english 和 English。bert-based-cased 模型可以用于掩码语言建模或下一句预测。

Decoder：基于 sparql-mim-zero 的预训练大模型。sparql-mlm-zero 模型是一种基于 BERT 的语言模型，它在大量的 SPARQL 查询日志上进行了预训练。sparql-mlm-zero 模型可以学习自然语言和 SPARQL 查询语言的通用表示，并利用词语的顺序信息，这对于像 SPARQL 这样的结构化语言非常重要。sparql-mlm-zero 模型可以用于知识图谱上的问答任务，将自然语言问题转换为 SPARQL 查询。

在前向传播的时候，分两种情况:

- 有 target 输入, 用于训练:

```
outputs = self.encoder(source_ids, attention_mask=source_mask)
#编码器编码source_ids,输出outputs
encoder_output = outputs[0].permute([1, 0, 2]).contiguous()
#取编码器最后一层输出,并转置维度为[batch_size, seq_len, hidden_size]
if target_ids is not None:
    # 如果有target输入,进行训练
    attn_mask = -1e4 * (1 - self.bias[:target_ids.shape[1], :target_ids.shape[1]])
    # 构造注意力掩码attn_mask,对角线以外部分填充-1e4
    tgt_embeddings = self.encoder.embeddings(target_ids).permute([1, 0, 2]).contiguous()
    # 得到target_ids的嵌入tgt_embeddings,并转置维度为[batch_size, seq_len, hidden_size]
    out = self.decoder(tgt_embeddings, encoder_output, tgt_mask=attn_mask,
    memory_key_padding_mask=(1 - source_mask).bool())
    # 解码器解码tgt_embeddings和encoder_output,产生out,并使用注意力掩码attn_mask
    ...
    loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1))[active_loss],
    shift_labels.view(-1)[active_loss])
    # 计算loss,采用LabelSmoothingLoss,loss用于反向传播更新encoder和decoder的参数

```

- 通过 beam search 不断生成部分预测序列, 维护当前最优的 beam_size 个序列, 最终得到完整的预测目标序列 preds。

```
  preds = []
# 存储最终的预测结果
for i in range(source_ids.shape[0]):
    # 对每条输入序列进行beam search
    context = encoder_output[:, i:i + 1]
    # 取编码器输出的第i条序列
    context_mask = source_mask[i:i + 1, :]
    # 取输入序列的注意力掩码
```

```
        beam = Beam(self.beam_size, self.sos_id, self.eos_id)
        # 初始化[beam search](poe://www.poe.com/_api/key_phrase?phrase=beam%20search&prompt=Tell%20me%20more%20about%20beam%20search.)
        input_ids = beam.getCurrentState()
        # 获取beam的当前状态,即beam中每个序列的部分预测结果
            for _ in range(self.max_length):
            # 进行max_length步beam search
            if beam.done():
                # 如果beam search提前结束,break
                break
        attn_mask = -1e4 * (1 - self.bias[:input_ids.shape[1], :input_ids.shape[1]])
        # 构造注意力掩码
        tgt_embeddings = self.encoder.embeddings(input_ids).permute([1, 0, 2]).contiguous()
        # 得到部分预测序列的嵌入
        out = self.decoder(tgt_embeddings, context, tgt_mask=attn_mask,
        # 解码器解码部分序列的预测和encoder输出
        memory_key_padding_mask=(1 - context_mask).bool())
        out = torch.tanh(self.dense(out))
        # 过激活函数
        hidden_states = out.permute([1, 0, 2]).contiguous()[:, -1, :]
        # 取解码器最后一层的输出
        out = self.lsm(self.lm_head(hidden_states)).data
        # 应用lm_head和log_softmax计算下一步的token得分
        beam.advance(out)
        # 根据得分更新beam
        ...
        hyp = beam.getHyp(beam.getFinal())
        # 得到beam搜索结束后的每个序列
        pred = beam.buildTargetTokens(hyp)[:self.beam_size]
        # 从每个序列构造target序列
        preds.append(torch.cat(pred, 0).unsqueeze(0))
    # 将各个target序列连接起来
    preds = torch.cat(preds, 0)
    # 最终的预测结果
    return preds

```

反向传播:

loss.backward() 进行反向传播。

对有 target 输入情况, 根据 loss 计算出的梯度更新 encoder 和 decoder 的参数。即进行前向和反向传播, 更新参数对无 target 输入情况, 没有进行反向传播。即只进行 beam search, 不更新参数

3.3 Beam-Search 算法实现

用来在没有 target 的情况下生成预测目标序列。维护 beam_size 个当前最优序列, 在每个时间步根据 encoder 输出和当前最优序列得分选择下一步的 token, 不断扩展生成最终完整的预测目标序列。

主要思想是: 在每一时间步, 维护当前最优的 beam_size 个部分序列, 称为 current beams, 根据 encoder 输出和 current beams 计算下一步所有可能的 token 的得分, 选择得分最高的 beam_size 个 token 扩展 current beams, 得到新的 current beams, 不断重复此过程直到得到完整的预测目标序列。

```
def __init__(self, size, sos, eos):
```

```
        self.size = size  # beam size
        self.tt = torch.cuda   # 用于 cuda 操作
        # The score for each translation on the beam.
        self.scores = self.tt.FloatTensor(size).zero_()
        # beam中每个序列的得分
        # The backpointers at each time-step.
        self.prevKs = []
        # 每一步的前序beams
        # The outputs at each time-step.
        self.nextYs = [self.tt.LongTensor(size)
                           .fill_(0)]
        # 每一步的token
        self.nextYs[0][0] = sos  # 第一个token为sos
        # Has EOS topped the beam yet.
        self._eos = eos
        # 结束token
        self.eosTop = False
        # 是否已有eos为第一个
        # Time and k pair for finished.
        self.finished = []
```

初始化 beam search,size 是 beam size,sos 和 eos 分别是序列的开始和结束 token。

```
def advance(self, wordLk):
        """
        Given prob over words for every last beam `wordLk` and attention
        `attnOut`: Compute and update the beam search.
        """
        numWords = wordLk.size(1)  # 词表大小

        # Sum the previous scores.
        if len(self.prevKs) > 0:
            # 如果不是第一步,为上一步的每个beam加上本步的得分
            beamLk = wordLk + self.scores.unsqueeze(1).expand_as(wordLk)
            # beam_size x 词表大小
            # Don't let EOS have children.
            for i in range(self.nextYs[-1].size(0)):
                if self.nextYs[-1][i] == self._eos:
                    # 如果beam的最后一个token是eos,则得分为-1e20
                    beamLk[i] = -1e20
        else:
            # 第一步,beamLk即为wordLk
            beamLk = wordLk[0]
        flatBeamLk = beamLk.view(-1)  # 展平,获得beam_size*词表大小的得分
        bestScores, bestScoresId = flatBeamLk.topk(self.size, 0, True, True)
                                            # 选出得分最高的beam_size个

        self.scores = bestScores  # 更新得分

        # bestScoresId是beam x word,计算对应beam和word
        prevK = bestScoresId // numWords
        self.prevKs.append(prevK)  # 记录对应前序beams
        self.nextYs.append((bestScoresId - prevK * numWords))
```

```
                                # 记录本步选中的token

        for i in range(self.nextYs[-1].size(0)):
            if self.nextYs[-1][i] == self._eos:
                # 如果有eos,记录对应的得分和位置
                s = self.scores[i]
                self.finished.append((s, len(self.nextYs) - 1, i))

        # End condition is when top-of-beam is EOS and no global score.
        if self.nextYs[-1][0] == self._eos:
            self.eosTop = True    # 如果第一个beam的token是eos,eosTop=True
```

advance 函数在 beam search 的每一步中根据上一步的 beam 状态和本步的 token 得分 wordLk 计算新的 beam 状态。

具体做法是:

1. 为上一步的每个 beam 加上 wordLk 中的得分, 得到 beamLk, 维度是 beam_size x 词表大小。
2. 将 beamLk 展平, 得到 flatBeamLk, 维度是 beam_size x 词表大小。
3. 从 flatBeamLk 中选出得分最高的 beam_size 个, 这就是新的 beam 状态。
4. 记录这些 beams 对应的 token 索引和上一步的 beams 索引, 以用于回溯最终的预测序列。
5. 判断是否搜索完成, 如果 beam 的第一个序列的最后一个 token 为 eos 且没有更高的置信度, 则结束搜索。

```
def done(self):
        return self.eosTop and len(self.finished) >= self.size  # 判断是否搜索完成

def getFinal(self):
    if len(self.finished) == 0:
        # 如果没有eos,取当前得分最高的beam_size个
        self.finished.append((self.scores[0], len(self.nextYs) - 1, 0))
    self.finished.sort(key=lambda a: -a[0])  # 按得分从高到底排序
    if len(self.finished) != self.size:
        # 如果finished长度不够,添加未完成的beam
        unfinished = []
        for i in range(self.nextYs[-1].size(0)):
            if self.nextYs[-1][i] != self._eos:
                s = self.scores[i]
                unfinished.append((s, len(self.nextYs) - 1, i))
        unfinished.sort(key=lambda a: -a[0])
        # 按得分从高到低排序
        self.finished += unfinished[:self.size - len(self.finished)]
    return self.finished[:self.size]  # 返回最终的beam_size个beam

def getHyp(self, beam_res):
    """Walk back to construct the full hypothesis."""
    hyps = []
    for _, timestep, k in beam_res:
        # 遍历最终的beam_size个beam
        hyp = []
```

```
        for j in range(len(self.prevKs[:timestep]) - 1, -1, -1):
            # 遍历beam对应的每一步
            hyp.append(self.nextYs[j + 1][k])  # 添加beam在每一步选择的token
            k = self.prevKs[j][k]  # 找到beam在上一步对应的beam
        hyps.append(hyp[::-1])  # 添加翻转后的序列,即完整的预测序列
    return hyps

def buildTargetTokens(self, preds):
    sentence = []
    for pred in preds:
        tokens = []
        for tok in pred:
            if tok == self._eos:
                break   # 遇到eos则结束
            tokens.append(tok)
        sentence.append(tokens)
    return sentence  # 返回预测token序列
```

getFinal 函数返回最终的 beam_size 个 beam,getHyp 函数通过回溯对应 beam 在每一步选择的 token, 构造出完整的预测序列,buildTargetTokens 将预测序列转换为词表表示的序列。最后, 通过 Predictions 构造目标序列的词表表示。

3.4 模型参数

- 训练数据集：LC-QuAD-1.0 数据集
- Encoder:bert-base-cased，生成句子向量
- Decoder: sparql-mlm-zero，转化为 sparql
- epoch:50 次
- batch_size:6

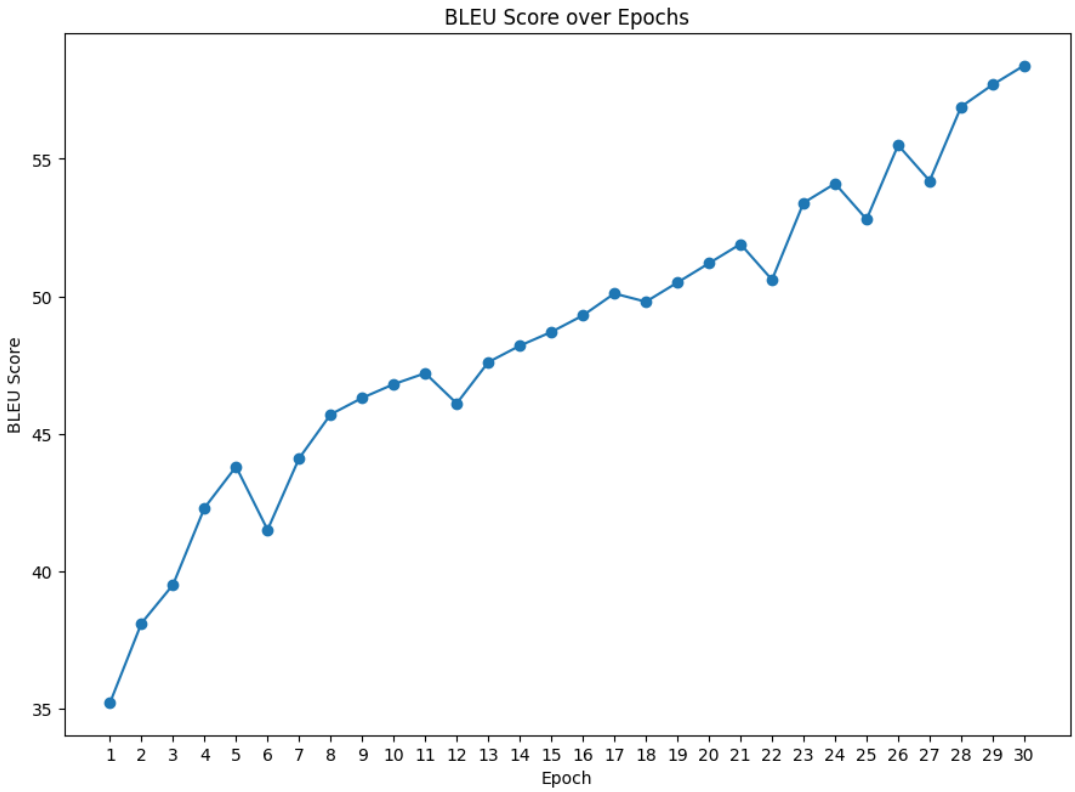
3.5 bleu 指标评估

BLEU 的全名为：bilingual evaluation understudy，即：双语互译质量评估辅助工具。它是用来评估机器翻译质量的工具。BLEU 的设计思想：机器翻译结果越接近专业人工翻译的结果，则越好。BLEU 算法实际上就是在判断两个句子的相似程度。想知道一个句子翻译前后的表示是否意思一致，直接的办法是拿这个句子的标准人工翻译与机器翻译的结果作比较，如果它们是很相似的，说明翻译很成功。

所以,BLEU 的主要步骤是:

1. 计算机器翻译输出与多个人工参考翻译的 n-gram 匹配精度 pn,n 从 1 到 4。
2. 对 pn 加权求和, 得到加权精度和。
3. 对加权和取幂, 得到 0-1 之间的得分, 作为最终的 BLEU 得分。

BLEU 得分越高, 表示机器翻译输出与人工翻译的相似度越高, 效果越好。



由此图可以得知,bleu 指标总体上升，说明翻译结果逐步变好。

3.6 网页展示

图 2: Seq2Seq 网页展示

4 基于 POS_Tagging 的 NER+RE 模型

4.1 主要任务说明

- NER: 使用 BiLSTM+CRF 对输入句子每个单词进行实体标注，对每个单词标注 O,E,E_end 三类标签，其中 O 代表" 不是实体",E 代表 "实体的中间部分" ,E_end 代表" 实体的结尾部分"。对识别出来的实体，使用 ElasticSearch 搜索 dbpedia2016 词条中最相近的链接。
- RE: 对 NER 抽取出来的实体，使用带有 Entity Marker 和 Mention pooling 的 BERT 模型进行关系抽取，该模型输出一个提前预设好的关系的一种。
- Triple Assembler: 将上述实体和关系组装成关系链路，进而转化为 sparql 或 cypher 语句，在知识图谱查询并返回结果

4.2 NER 模型

- BiLSTM（Bidirectional Long Short-Term Memory）是一种深度学习模型，常用于处理序列数据。它是 LSTM 的变种，通过在输入序列的两个方向上分别构建 LSTM 来获取序列中每个位置的上下文信息，从而获得更全面的序列特征表示。
- CRF（Conditional Random Field）是一种概率图模型，用于建模序列标注任务中标签之间的依赖关系。CRF 可以在序列标注过程中考虑整个序列的上下文信息，从而避免了标签之间的矛盾和不一致。

  CRF 的主要思想是: 相邻的标注之间存在依赖关系, 不是独立同分布的。CRF 可以同时模型化特征函数和依赖关系, 进行联合概率匹配, 得到最优的标注序列。

  CRF 包含两个主要部分:

  – 特征函数, 其中,fk 是一个特征函数, k 是对应的权重, 对所有可能的 Y 进行求和。

  $$score(X, Y) =_k f_k(Y_{m-1}, Y_m, X, m)$$

  – 指数线性模型: 定义条件概率分布 p(Y|X), 其中分母是规范化因子。Z(X) 使得 p(Y|X) 的值归一化。

  $$p(Y|X) = (1/Z(X)) * exp^{score(X,Y)}$$

- Viterbi 算法是一种动态规划算法，用于在 CRF 中找到最优的标注序列。它通过递推地计算每个位置上每个标签的最优路径分数，从而找到整个序列的最优路径。Viterbi 算法是 CRF 解码过程使用的算法。它可以有效地解码得到最优的标注序列。主要思想是:

  1. 定义状态转移分数转移矩阵和观测概率矩阵。
  2. 初始化状态转移矩阵第一列, 代表向各个状态转移的概率得分。
  3. 递推计算各个时间步转移到每个状态的最优得分, 并记录最优前驱状态。
  4. 回溯最优前驱状态, 得到最优路径, 即最优标注序列。

  所以 Viterbi 算法通过动态规划高效求解 CRF, 得到最优的标注序列, 这在 NLP 任务中很有用。

- 将识别出来的实体,使用 ElasticSearch 搜索出其在图数据库中最相似的节点,为后续组装成可执行 SPARQL 作准备

定义 BiLSTM_CRF 模型, 包含以下主要部分:

```
self.embedding = nn.Embedding(vocab_size, embedding_dim)
# 词embedding

self.char_embedding = nn.Embedding(char_vocab_size, char_embedding_dim)
# 字符embedding

self.char_lstm = nn.LSTM(char_embedding_dim, char_hidden_dim, bidirectional=True, num_layers=1)
# 双向LSTM编码字符

self.lstm = nn.LSTM(embedding_dim + char_hidden_dim * 2, hidden_dim // 2, bidirectional=True,
    num_layers=1)
# 双向LSTM编码词和字符信息

self.hidden2tag = nn.Linear(hidden_dim, tag_vocab_size)
# 线性层转换为标注得分

self.crf = CRF(tag_vocab_size)
# CRF解码层

```

4.3 F1-score 评估

将 conllp-2003 划分成训练集和验证集，参考验证集的 F1-score，说明 F1-score 逐步变好

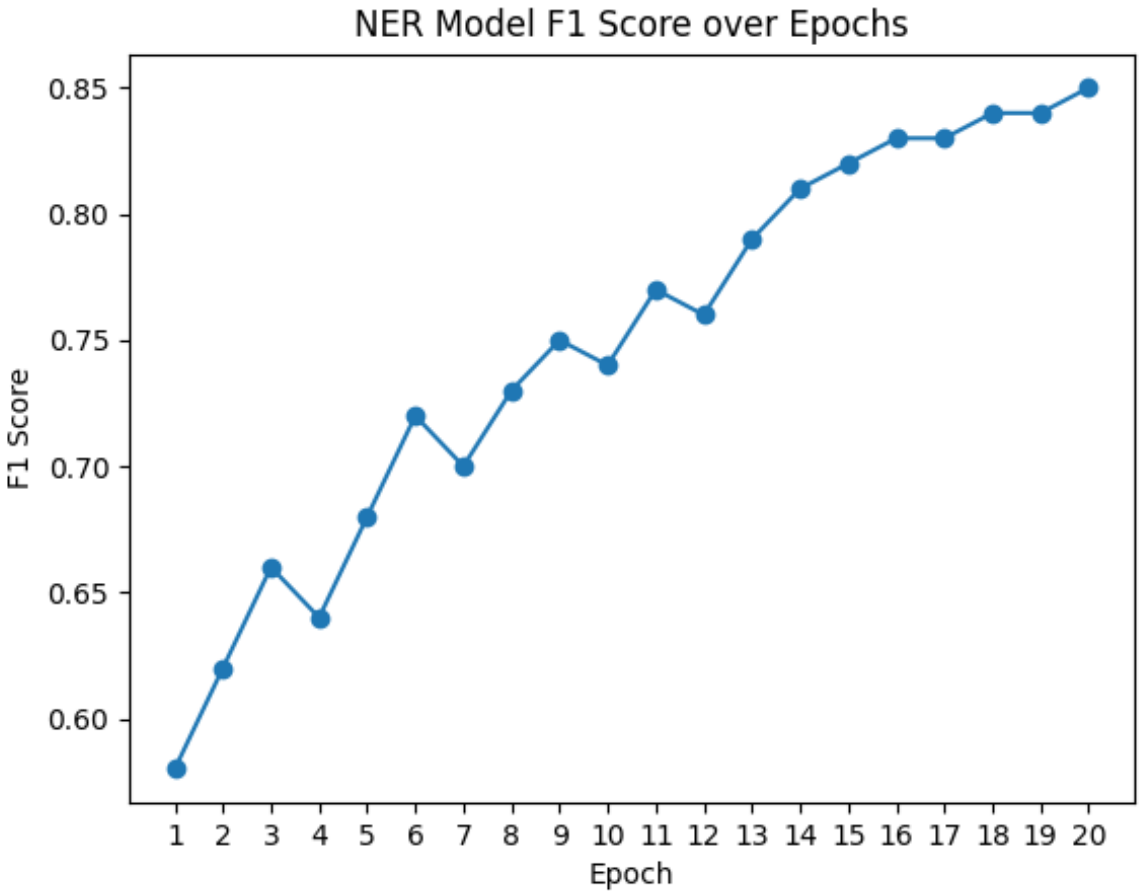


图 3: NER 模型评估

4.4 RE 模型

使用 en 使用带 Entity-Marker 和 Mention Pooling 都 BERT 模型进行关系抽取

4.4.1 en_core_web_lg

en_core_web_lg 是一个 spaCy 提供的大型英语模型。
它具有以下主要功能:

- 标注词性、依存关系等语法信息
- 实体识别
- 向量表示 - 可以用于文本相似度计算等
- 支持 nlp 相关任务的 pipeline

所以, 加载这个模型可以使我们的代码获取到词性、依存关系、实体识别等丰富的语言信息。

```
if self.detect_entities:
    self.nlp = spacy.load("en_core_web_lg")
```

4.4.2 Entity Marker

是一种在输入序列中插入特殊标记来表示实体边界和类型的方法。这些标记可以帮助 PLM 捕捉实体的语义信息，并区分不同的实体类别。例如，给定一个句子：

$$Tom \quad Hanks \quad starred \quad in \quad Forrest \quad Gump.$$

如果我们要抽取其中的人物（PER）和电影（MOV）实体，我们可以用以下方式添加标记：

$$[PER \quad TomHanks \quad PER]starredin[MOV \quad ForrestGump \quad MOV].$$

这样，PLM 就可以根据标记来识别实体，并为每个实体生成一个向量表示。Entity_Marker 的主要流程是:

1. 加载 spaCy 或 NER 模型, 比如 en_core_web_lg 和上述的
2. 调用 nlp(text) 对输入文本进行实体识别
3. 遍历识别出的实体, 判断实体类型是否在感兴趣的类型列表中, 如果是则进行标注
4. 标注的形式是在实体首尾添加 [E1],[/E1] 等标签
5. 返回标注后的文本

```
class Entity_Marker(object):
    def __init__(self, model="en_core_web_lg"):
        self.nlp = spacy.load(model)
        self.entities_of_interest = ["PERSON", "NORP", "FAC", "ORG", "GPE", "LOC", "PRODUCT", "EVENT"
    ,
                                     "WORK_OF_ART", "LAW", "LANGUAGE", 'PER']

    def mark_entities(self, text):
        doc = self.nlp(text)
        result = ''
        for ent in doc.ents:
            if ent.label_ in self.entities_of_interest:
                result += "[%s]" % ent.label_ + ent.text + "[/%s]" % ent.label_ + ' '
            else:
                result += ent.text + ' '
        return result.strip()
```

4.4.3 Mention-Pooling

是一种用于从多个实体提及中聚合一个单一的实体表示的方法。这对于处理多句或跨句的关系抽取任务是有用的，因为一个实体可能在不同的句子中以不同的形式出现。由于 BERT 的输出为序列特征, 但是我们需要一个分类预测, 所以需要进行 pooled output。这里使用的是 mention-pooling 方法。

具体实现在模型的 forward 方法中:

```python
def forward(self, input_ids, token_type_ids, attention_mask, Q, e1_mask=None, e2_mask=None, infer=True, e1_e2_start=None):
    outputs = self.bert(input_ids, token_type_ids, attention_mask, Q, e1_mask, e2_mask)
    sequence_output = outputs[0]

    if infer:
        # mention-pooling
        e1_start, e2_start = e1_e2_start[:,0], e1_e2_start[:,1]
        e1_output = sequence_output[torch.arange(sequence_output.size(0)).unsqueeze(1), e1_start]
        e2_output = sequence_output[torch.arange(sequence_output.size(0)).unsqueeze(1), e2_start]
        pooled_output = torch.cat((e1_output, e2_output), dim=1)
    else:
        pooled_output = self.pooler(sequence_output[:,0,:])
```

当 infer=True 时, 使用 mention-pooling, 具体步骤:

1. 从 e1_e2_start 中取出 e1 和 e2 的 start id, 分别为 e1_start 和 e2_start
2. 根据 start id, 从 BERT 的 sequence_output 中取出 e1 和 e2 对应的输出, 分别为 e1_output 和 e2_output
3. 将 e1_output 和 e2_output 在 dim=1 上拼接, 作为 pooled_output
4. 所以 pooled_output 包含了两个实体的特征, 这为关系分类提供了两个实体的语义信息
5. 当 infer=False 时, 使用 BERT 的 Pooler 层得到 pooled_output

所以,mention-pooling 的思想是直接从 BERT 的序列输出中取出两个实体对应的向量表示, 然后拼接作为最终的 pooled 输出, 这 retain 了两个实体的语义信息, 为关系分类提供有利信息。

所以,get_e1e2_start 用于获取两个实体的位置信息,meta_labels 代表每个样本的标签,mention-pooling 是一种简单高效的 pooling 方法, 它直接从 BERT 的序列输出中提取两个实体的特征, 并拼接作为分类器的输入, 这为关系分类任务提供了重要的实体语义信息。

4.5 三元组->SPARQL 组装器

将上述提取出的实体-关系树使用 ElasticSearch 找到对应的实体，关系在 dbpedia2016 里的 uri，封装好后用组装成 sparql

```python
def generate_sparql_query(triples_list):
    """
    将三元组列表转换为DBpedia的SPARQL查询语句。

    参数:
    triples_list (list): 一个列表，每个元素是一个长度为3的子列表，分别为主语，谓语，宾语。

    返回:
    str: 生成的SPARQL查询语句。
    """
    # 初始化查询语句的基本结构，包括前缀定义和选择子句
    query = """
    PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX dbo: <http://dbpedia.org/ontology/>
```

```
    PREFIX dbr: <http://dbpedia.org/resource/>
    PREFIX dbp: <http://dbpedia.org/property/>

    SELECT DISTINCT * WHERE {
    """

    # 遍历三元组列表，将每个三元组转换为SPARQL查询语句的内容
    for triple in triples_list:
        subject, predicate, object_ = triple
        query += f"    {subject} {predicate} {object_} .\n"

    # 结束查询语句
    query += "}"

    return query
```

4.6 网页展示

图 4: NER 网页展示

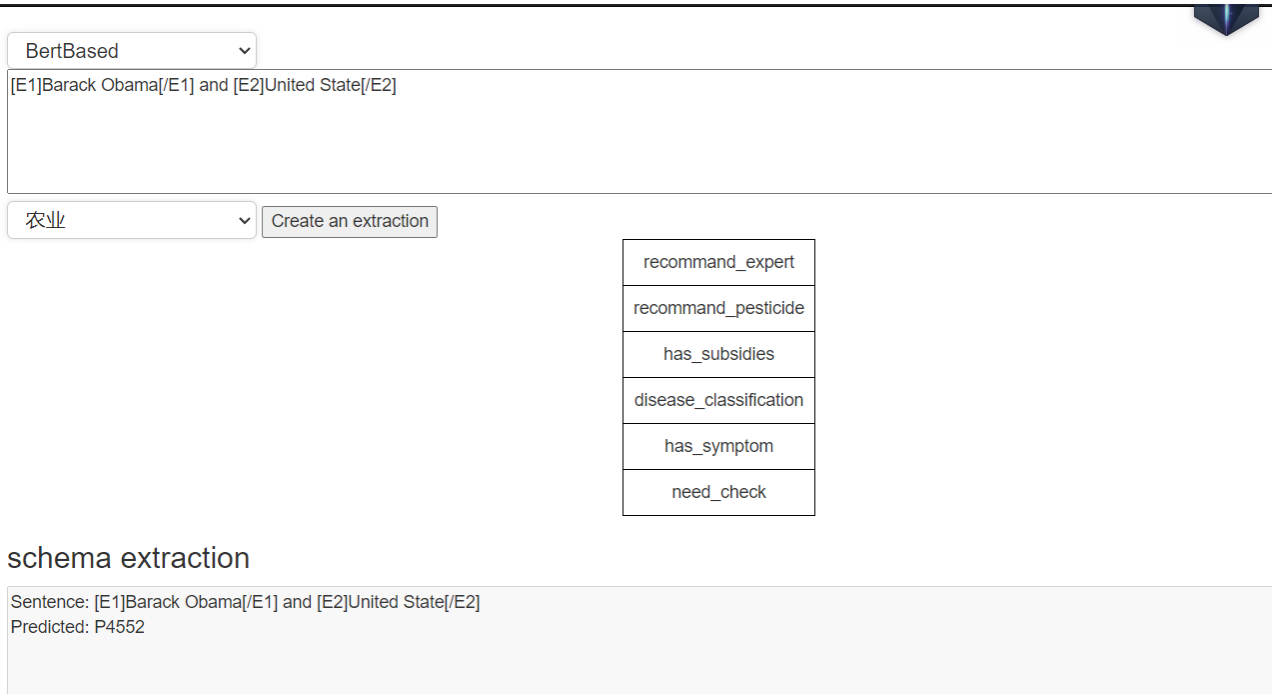

BertBased
[E1]Barack Obama[/E1] and [E2]United State[/E2]
农业
Create an extraction
recommand_expert
recommand_pesticide
has_subsidies
disease_classification
has_symptom
need_check
schema extraction
Sentence: [E1]Barack Obama[/E1] and [E2]United State[/E2]
Predicted: P4552

图 5: RE 网页展示

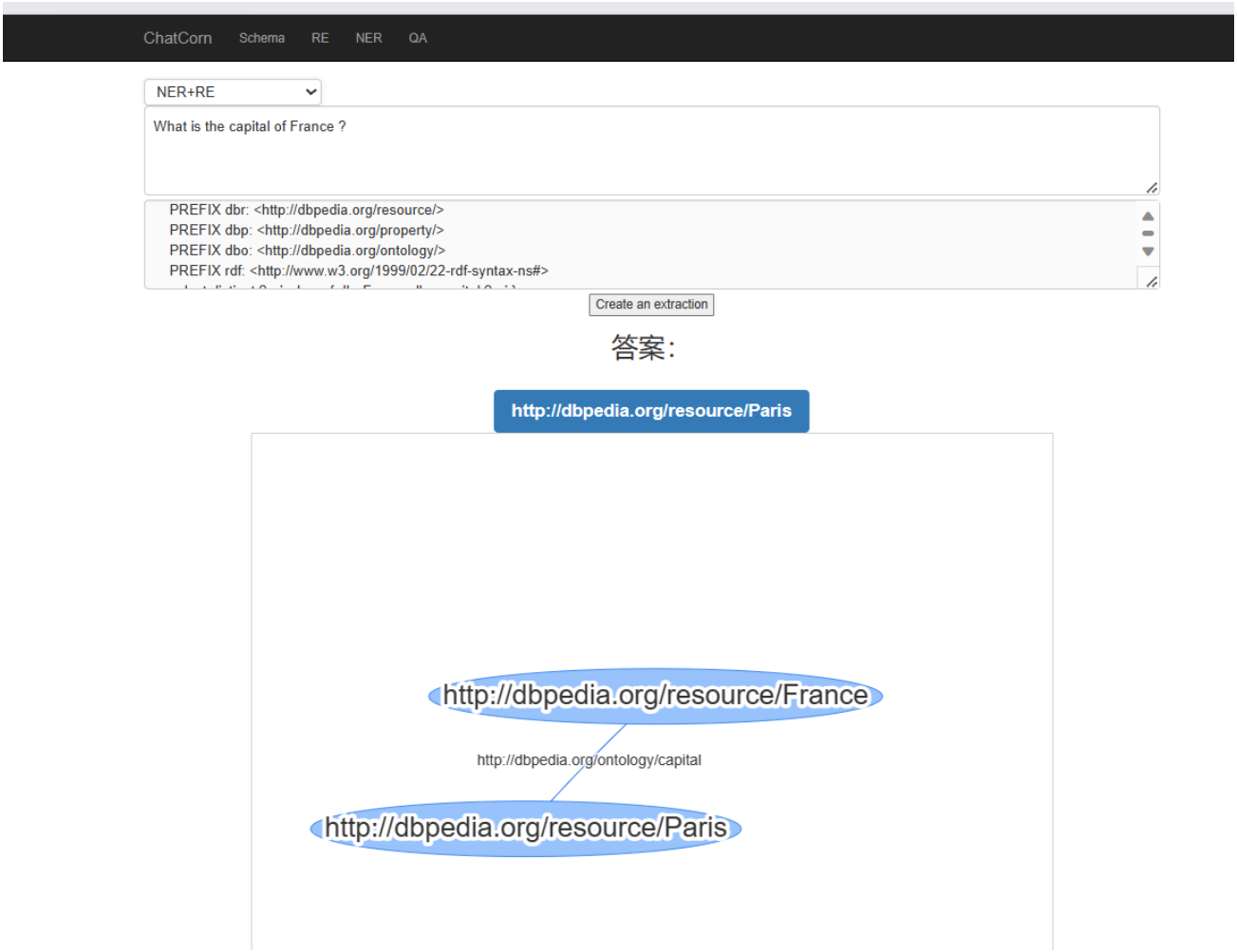


图 6: QA 网页展示

5 总结

使用 Seq2Seq 模型直接用 LC-QuAD 训练集进行训练，可能出现实体，谓词识别不准确，SPARQL 语句格式错误的问题

将原任务划分为多个子任务，使用 POS_Tagging 进行实体抽取，使用 BERT 进行关系抽取，再使用 ElasticSearch 搜索该实体，最后在 Neo4j 图数据库进行查询和可视化，能提高模型的鲁棒性，提升任务质量，确保 SPARQL 语句格式正确。

由于 NER+RE 模型所识别的关系有限，因此无法在整个 LC-QuAD 数据集进行评估。并且 NER 和 RE 两个分模型所使用的训练数据集不同，因此每个分模型的 epoch 无法对齐。因此，在保持 HuggingFace 所使用的 RE 模型不变的情况下如图是我们小组成员表人工抽取了 100 个 question-sparql 对进行评估，测试了 NER 的 epoch 和最终 bleu 指标关系，得到如下图

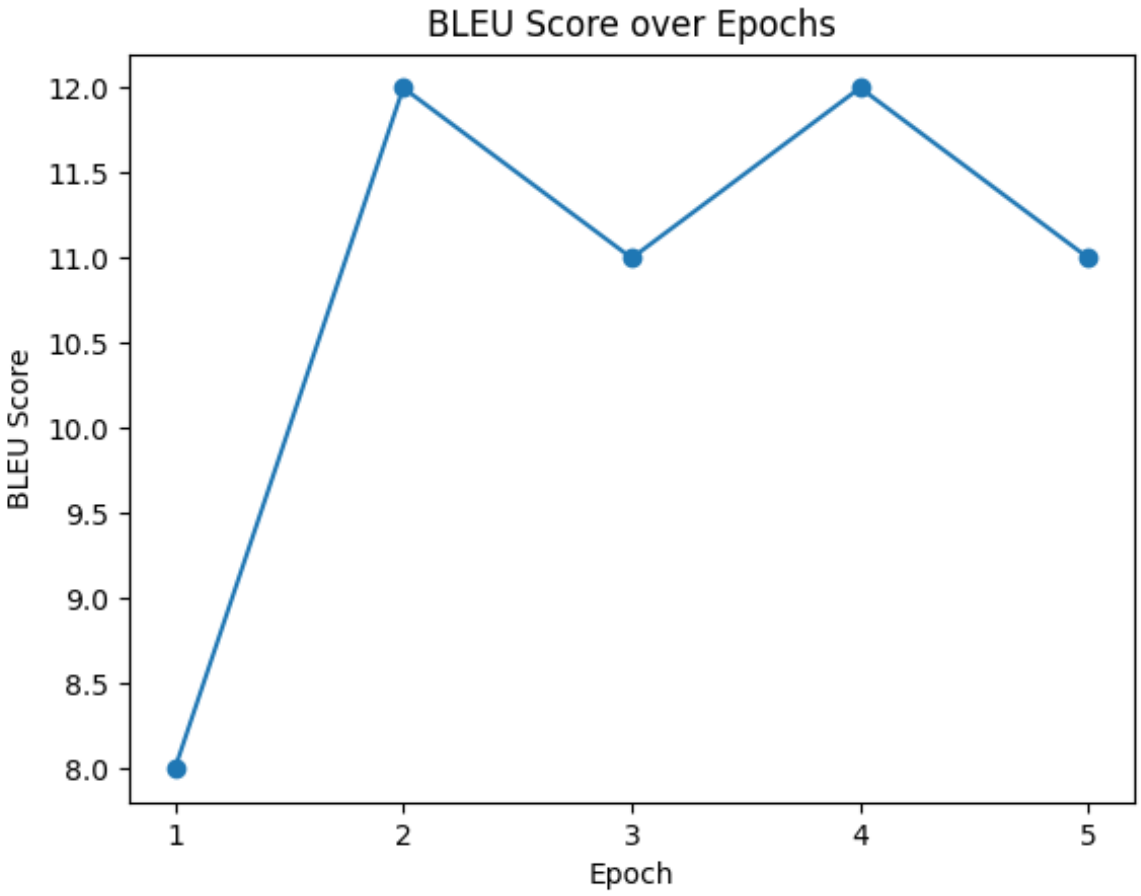


图 7: NER+RE 模型评估

虽然 NER+RE 在 LC-QuAD 数据集表现远远不如直接用该数据集训练的 Seq2Seq 模型，但 NER+RE 有更强的鲁棒性，当跳出 LC-QuAD 数据集时，NER+RE 能表现得更好，因为 NER+RE 的实体关系都是搜索出来的，能保证实体关系的存在性。