

Web 信息处理与应用实验报告

方驰正 来泽远 周瓯翔

2023 年 12 月 24 日

1 成员

- 方驰正 PB21000163 组长
- 来泽远 PB21000164
- 周瓯翔 PB21000265

2 实验介绍

2.1 实验背景

知识图谱 (KG) 是一种用于表示和组织知识的图形化数据结构,旨在捕获现实世界中的实体、概念、关系和属性。它是一种语义网络,通过连接不同元素之间的关系来呈现信息,并允许计算机系统理解和推理这些信息。

KG 内部包含丰富的实体和关系信息,这些信息不仅能够强化对用户和物品之间关系的建模,同时也揭示了物品之间的多种相关性。举例来说,KG 能展示两部电影是否由同一位导演执导,这种关联可以被用来加强推荐系统对用户品味和喜好的理解。

此外,KG 还有助于解释用户的偏好。例如,系统能够将用户对某部电影的选择归因于该电影的主演或其他与该电影相关的因素。这种解释性的特点使得推荐系统能够更好地理解用户的行为,并为用户提供更具个性化和可信赖的推荐。

2.2 实验目的

本次实验中,我们将利用知识图谱的特性,设计并实现一个基于知识图谱的推荐系统。该系统将利用知识图谱中的实体和关系信息,为用户推荐与其历史行为相关的物品。

在阶段一中,我们将从公开图谱中匹配指定电影对应的实体,并抽取合适的部分图谱,按照规则对抽取到的图谱进行处理。在阶段二中,我们基于豆瓣电影评分数据,结合阶段一所获得的图谱信息,进行可解释的、知识感知的个性化电影推荐。

3 实验环境

- 操作系统: Windows 11
- 开发环境: python 3.7
- 软件平台: visual studio code

4 阶段一：图谱抽取

4.1 建立链接

我们根据给出的连接信息文件 douban2fb.txt，根据电影 ID 匹配对应的实体，核心代码如下：

```

1 movie_tag = pd.read_csv(path + '/../data/Movie_tag.csv')
2 douban2fb = pd.read_csv(path + '/../data/douban2fb.csv')
3 df = pd.merge(movie_tag, douban2fb, how='inner', on='id')
4 df.set_index('entity', inplace=True)
5 df['tag'] = df['tag']. apply(lambda x: x.split(','))
6
7 df.to_json(orient='index',
8           path_or_buf=path + '/../data/movie_entity.json',
9           force_ascii=False,
10          indent=4)

```

直接利用 pandas 的 merge 函数即可完成链接，之后将结果以 json 格式保存。

4.2 一跳子图

之后我们建立一跳子图，核心代码如下：

```

1 with gzip. open(path + '/../data/freebase_douban.gz', 'rb') as f:
2     for line in f:
3         line = line.strip()
4         triplet = line.decode().split('\t')[:3]
5
6         patten = "<http://rdf.freebase.com/ns/"
7         if (patten != triplet[0][: len(patten)] or patten != triplet[2][: len(patten)]):
8             continue
9         item1 = triplet[0][ len(patten):-1]
10        item2 = triplet[2][ len(patten):-1]
11        relation = triplet[1]
12
13        # 一跳子图
14        if (item1 in movie_list):
15            cnt = cnt + 1
16            if (relation not in graph.loc[item1, 'content']):
17                graph.loc[item1, 'content'][relation] = []
18                graph.loc[item1, 'content'][relation].append(item2)
19                graph.loc[item1, 'count'] = graph.loc[item1, 'count'] + 1
20        if (item2 in movie_list):
21            cnt = cnt + 1
22            if (item1 not in graph.index):
23                graph.loc[item1] = [0, 0, {}]
24            if (relation not in graph.loc[item1, 'content']):
25                graph.loc[item1, 'content'][relation] = []
26                graph.loc[item1, 'content'][relation].append(item2)

```

27

```
graph.loc[item1, 'count'] = graph.loc[item1, 'count'] + 1
```

我们使用 DataFrame 作为存储结构，以头节点为索引，每个节点包含一个计数器和一个字典，字典的键为关系，值为一个列表，列表中存储了与该节点有关系的节点，即尾节点。在建立二跳子图前，我们过滤一跳子图，删除出现次数小于 20 的非电影实体，以减小构建二跳子图的压力；对关系的过滤放在二跳子图之后一并进行。

过滤后的一跳子图的大小约为 6MB，其规模如表 1 所示。

#	电影节点	一跳节点	总节点	关系
种类数量	578	146	724	139
出现总数	125488	13266	138754	138754
平均次数	217.11	90.86	191.65	998.23

表 1: 一跳子图规模

4.3 二跳子图

核心代码与一跳子图类似，值得注意的是，二跳子图的数据量远大于一跳子图。经测试，我们发现 DataFrame 的 loc 方法复杂度较高，因此我们替换为使用红黑树维护的字典，以加速子图的构建。

二跳子图建立后，其 json 文件约为 2.94GB。我们删除新引入的且出现次数小于 30 的实体。并且遍历整个子图，删除出现次数小于 50 的关系。

过滤后的二跳子图大小约为 22MB，其规模如表 2 所示。

#	电影节点	一跳节点	二跳节点	总节点	关系
种类数量	578	146	5394	6118	56
出现总数	145966	104376080	182851	104704897	530115
平均次数	252.54	714904.66	33.90	17114.23	9466.34

表 2: 二跳子图规模

5 阶段二：知识感知推荐

5.1 建立映射

根据给出的映射关系，我们将实体 ID 映射到 [0,num of entities) 的范围内，并将关系映射到 [0,num of relations) 范围内，并将结果保存到 data/Douban/kg_final.txt 文件中。核心代码如下：

1

```
for key in graph.keys():
2     if key not in movie_id_dict.keys():
3         movie_id_dict[key] = movie_id_cnt
4         movie_id_cnt += 1
5     u = movie_id_dict[key]
6     for relation in graph[key]['content'].keys():
```

```

7         if relation not in relation_dict.keys():
8             relation_dict[relation] = relation_cnt
9             relation_cnt += 1
10        r = relation_dict[relation]
11        for v in graph[key]['content'][relation]:
12            if v not in movie_id_dict.keys():
13                movie_id_dict[v] = movie_id_cnt
14                movie_id_cnt += 1
15            v = movie_id_dict[v]
16            output.write( str(u) + ' ' + str(r) + ' ' + str(v) + '\n')

```

5.2 图谱嵌入模型

基于给定的 baseline 框架，我们完成了基于图谱嵌入的模型。

1. 实现 KG 的构建。核心代码如下：

```

1  # 1. 为KG添加逆向三元组，即对于KG中任意三元组(h, r, t)，添加逆向三元组 (t, r+n_relations, h)
2  n_relations = kg_data["r"]. max() + 1
3  self.kg_data = kg_data
4  for index, row in kg_data.iterrows():
5      self.kg_data.loc[ len(
6          kg_data.index)] = [row['t'], row['r'] + n_relations, row['h']]
7
8  # 2. 计算关系数，实体数和三元组的数量
9  self.n_relations = self.kg_data["r"]. max() + 1
10 self.n_entities = max(self.kg_data["h"]. max(),
11                        self.kg_data["t"]. max()) + 1
12 self.n_kg_data = len(self.kg_data)
13
14 # 3. 根据 self.kg_data 构建字典 self.kg_dict，其中key为h, value为tuple(t, r),
15 #    和字典 self.relation_dict，其中key为r, value为tuple(h, t)。
16 self.kg_dict = collections.defaultdict( list)
17 self.relation_dict = collections.defaultdict( list)
18 for index, row in self.kg_data.iterrows():
19     self.kg_dict[row['h']].append((row['t'], row['r']))
20     self.relation_dict[row['r']].append((row['h'], row['t']))

```

2. 实现 transE 算法。核心代码如下所示：

```

1 def calc_kg_loss_TransE(self, h, r, pos_t, neg_t):
2     """
3     h:      (kg_batch_size)
4     r:      (kg_batch_size)
5     pos_t:  (kg_batch_size)
6     neg_t:  (kg_batch_size)
7     """
8     r_embed = self.relation_embed(r) # (kg_batch_size, relation_dim)
9

```

```

10     h_embed = self.entity_embed(h) # (kg_batch_size, embed_dim)
11     pos_t_embed = self.entity_embed(pos_t) # (kg_batch_size, embed_dim)
12     neg_t_embed = self.entity_embed(neg_t) # (kg_batch_size, embed_dim)
13
14     # 5. 对关系嵌入，头实体嵌入，尾实体嵌入，负采样的尾实体嵌入进行L2范数归一化
15     r_embed = F.normalize(r_embed, p=2, dim=1)
16     h_embed = F.normalize(h_embed, p=2, dim=1)
17     pos_t_embed = F.normalize(pos_t_embed, p=2, dim=1)
18     neg_t_embed = F.normalize(neg_t_embed, p=2, dim=1)
19
20     # 6. 分别计算正样本三元组 (h_embed, r_embed, pos_t_embed) 和负样本三元组 (h_embed,
21         r_embed, neg_t_embed) 的得分
22     pos_score = torch.norm(h_embed + r_embed - pos_t_embed, p=2,
23                             dim=1) # (kg_batch_size)
24     neg_score = torch.norm(h_embed + r_embed - neg_t_embed, p=2,
25                             dim=1) # (kg_batch_size)
26
27     # 7. 使用 BPR Loss 进行优化，尽可能使负样本的得分大于正样本的得分
28     kg_loss = (-1.0) * F.logsigmoid(pos_score - neg_score)
29     kg_loss = torch.mean(kg_loss)
30
31     l2_loss = _L2_loss_mean(h_embed) + _L2_loss_mean(
32         r_embed) + _L2_loss_mean(pos_t_embed) + _L2_loss_mean(neg_t_embed)
33     loss = kg_loss + self.kg_l2loss_lambda * l2_loss
34     return loss

```

3. 同时，通过相加，逐元素乘积，拼接的方式为物品嵌入注入图谱实体的语义信息。并通过命令行参数的形式选择注入类型。核心代码如下：

```

1 def inject_add(self, item_embed, item_kg_embed):
2     return F.normalize(F.normalize(item_embed) +
3                         F.normalize(item_kg_embed),
4                         p=2,
5                         dim=1)
6
7 def inject_concat(self, item_embed, item_kg_embed):
8     concat_embed = torch.cat(
9         (F.normalize(item_embed), F.normalize(item_kg_embed)), dim=1)
10    return concat_embed
11
12 def inject_multiply(self, item_embed, item_kg_embed):
13    return F.normalize(F.normalize(item_embed) *
14                      F.normalize(item_kg_embed),
15                      p=2,
16                      dim=1)
17
18 if self.inject_embedding_type == "add":
19     inject_embedding = self.inject_add

```

```
20 elif self.inject_embedding_type == "concat":  
21     inject_embedding = self.inject_concat  
22 elif self.inject_embedding_type == "multiply":  
23     inject_embedding = self.inject_multiply
```

6 实验总结

参考资料

- [1] [PSS3E20 EDA which makes sense](#)