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，这种关联可以被利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：

```
movie_tag = pd.read_csv(path + '/../data/Movie_tag.csv')
douban2fb = pd.read_csv(path + '/../data/douban2fb.csv')
df = pd.merge(movie_tag, douban2fb, how='inner', on='id')
df.set_index('entity', inplace=True)
df['tag'] = df['tag']. apply(lambda x: x.split(','))

df.to_json(orient='index',
           path_or_buf=path + '/../data/movie_entity.json',
           force_ascii=False,
           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：

```
with gzip. open(path + '/../data/freebase_douban.gz', 'rb') as f:
     for line in f:
         line = line.strip()
         triplet = line.decode().split('\t')[:3]

         patten = "<http://rdf.freebase.com/ns/"
         if (patten != triplet[0][: len(patten)] or patten != triplet[2][: len(patten)]):
             continue
         item1 = triplet[0][ len(patten):-1]
         item2 = triplet[2][ len(patten):-1]
         relation = triplet[1]

         # 一跳子图
         if (item1 in movie_list):
             cnt = cnt + 1
             if (relation not in graph.loc[item1, 'content']):
                 graph.loc[item1, 'content'][relation] = []
             graph.loc[item1, 'content'][relation].append(item2)
             graph.loc[item1, 'count'] = graph.loc[item1, 'count'] + 1
         if (item2 in movie_list):
             cnt = cnt + 1
             if (item1 not in graph.index):
                 graph.loc[item1] = [0, 0, {}]
             if (relation not in graph.loc[item1, 'content']):
                 graph.loc[item1, 'content'][relation] = []
             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 $[0, \text{num of entities})$ 的范围内，并将关系映射到 $[0, \text{num of relations})$ 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
```

```
    u = movie_id_dict[key]
    for relation in graph[key]['content'].keys():
        if relation not in relation_dict.keys():
            relation_dict[relation] = relation_cnt
            relation_cnt += 1
        r = relation_dict[relation]
        for v in graph[key]['content'][relation]:
            if v not in movie_id_dict.keys():
                movie_id_dict[v] = movie_id_cnt
                movie_id_cnt += 1
            v = movie_id_dict[v]
            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。

实现 KG 的构建。核心代码如下：

```
# 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)
n_relations = kg_data["r"]. max() + 1
self.kg_data = kg_data
for index, row in kg_data.iterrows():
    self.kg_data.loc[ len(
        kg_data.index)] = [row['t'], row['r'] + n_relations, row['h']]

# 2. 计算关系数，实体数和三元组的数量
self.n_relations = self.kg_data["r"]. max() + 1
self.n_entities =  max(self.kg_data["h"]. max(),
                       self.kg_data["t"]. max()) + 1
self.n_kg_data =  len(self.kg_data)

# 3. 根据 self.kg_data 构建字典 self.kg_dict ，其中key为h，value为tuple(t, r),
#    和字典 self.relation_dict，其中key为r，value为tuple(h, t)。
self.kg_dict = collections.defaultdict( list)
self.relation_dict = collections.defaultdict( list)
for index, row in self.kg_data.iterrows():
    self.kg_dict[row['h']].append((row['t'], row['r']))
    self.relation_dict[row['r']].append((row['h'], row['t']))
```

实现 transE 算法。核心代码如下所示：

```
def calc_kg_loss_TransE(self, h, r, pos_t, neg_t):
    """
    h:      (kg_batch_size)
    r:      (kg_batch_size)
    pos_t:  (kg_batch_size)
    neg_t:  (kg_batch_size)
    """
    r_embed = self.relation_embed(r)  # (kg_batch_size, relation_dim)
```

```

    h_embed = self.entity_embed(h)  # (kg_batch_size, embed_dim)
    pos_t_embed = self.entity_embed(pos_t)  # (kg_batch_size, embed_dim)
    neg_t_embed = self.entity_embed(neg_t)  # (kg_batch_size, embed_dim)

    # 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
    r_embed = F.normalize(r_embed, p=2, dim=1)
    h_embed = F.normalize(h_embed, p=2, dim=1)
    pos_t_embed = F.normalize(pos_t_embed, p=2, dim=1)
    neg_t_embed = F.normalize(neg_t_embed, p=2, dim=1)

    # 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, r_embed,
        neg_t_embed) 的得分
    pos_score = torch.norm(h_embed + r_embed - pos_t_embed, p=2,
                           dim=1)  # (kg_batch_size)
    neg_score = torch.norm(h_embed + r_embed - neg_t_embed, p=2,
                           dim=1)  # (kg_batch_size)

    # 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
    kg_loss = (-1.0) * F.logsigmoid(pos_score - neg_score)
    kg_loss = torch.mean(kg_loss)

    l2_loss = _L2_loss_mean(h_embed) + _L2_loss_mean(
        r_embed) + _L2_loss_mean(pos_t_embed) + _L2_loss_mean(neg_t_embed)
    loss = kg_loss + self.kg_l2loss_lambda * l2_loss
    return loss
```

我们也实现了 transR 算法，核心代码如下所示：

```
def calc_kg_loss_TransR(self, h, r, pos_t, neg_t):
    """
    h:      (kg_batch_size)
    r:      (kg_batch_size)
    pos_t:  (kg_batch_size)
    neg_t:  (kg_batch_size)
    """
    r_embed = self.relation_embed(r)  # (kg_batch_size, relation_dim)
    W_r = self.trans_M[r]  # (kg_batch_size, embed_dim, relation_dim)

    h_embed = self.entity_embed(h)  # (kg_batch_size, embed_dim)
    pos_t_embed = self.entity_embed(pos_t)  # (kg_batch_size, embed_dim)
    neg_t_embed = self.entity_embed(neg_t)  # (kg_batch_size, embed_dim)

    # 1. 计算头实体，尾实体和负采样的尾实体在对应关系空间中的投影嵌入
    r_mul_h = torch.bmm(h_embed.unsqueeze(1),
                        W_r).squeeze(1)  # (kg_batch_size, relation_dim)
    r_mul_pos_t = torch.bmm(pos_t_embed.unsqueeze(1), W_r).squeeze(
        1)  # (kg_batch_size, relation_dim)
    r_mul_neg_t = torch.bmm(neg_t_embed.unsqueeze(1), W_r).squeeze(
```

```
        1)  # (kg_batch_size, relation_dim)

    # 2. 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
    r_embed = F.normalize(r_embed, p=2, dim=1)
    r_mul_h = F.normalize(r_mul_h, p=2, dim=1)
    r_mul_pos_t = F.normalize(r_mul_pos_t, p=2, dim=1)
    r_mul_neg_t = F.normalize(r_mul_neg_t, p=2, dim=1)

    # 3. 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, r_embed,
        neg_t_embed) 的得分
    pos_score = torch. sum(torch. pow(r_mul_h + r_embed - r_mul_pos_t, 2),
                           dim=1)  # (kg_batch_size)
    neg_score = torch. sum(torch. pow(r_mul_h + r_embed - r_mul_neg_t, 2),
                           dim=1)  # (kg_batch_size)

    # 4. 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
    kg_loss = torch.mean(-1.0 * F.logsigmoid(neg_score - pos_score))

    l2_loss = _L2_loss_mean(r_mul_h) + _L2_loss_mean(
        r_embed) + _L2_loss_mean(r_mul_pos_t) + _L2_loss_mean(r_mul_neg_t)
    loss = kg_loss + self.kg_l2loss_lambda * l2_loss
    return loss
```

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：

```
def inject_add(self, item_embed, item_kg_embed):
    return F.normalize(F.normalize(item_embed) +
                       F.normalize(item_kg_embed),
                       p=2,
                       dim=1)

def inject_concat(self, item_embed, item_kg_embed):
    concat_embed = torch.cat(
        (F.normalize(item_embed), F.normalize(item_kg_embed)), dim=1)
    return concat_embed

def inject_multiply(self, item_embed, item_kg_embed):
    return F.normalize(F.normalize(item_embed) *
                       F.normalize(item_kg_embed),
                       p=2,
                       dim=1)

if self.inject_embedding_type == "add":
    inject_embedding = self.inject_add
elif self.inject_embedding_type == "concat":
    inject_embedding = self.inject_concat
elif self.inject_embedding_type == "multiply":
    inject_embedding = self.inject_multiply
```

此处为了保证注入的语义信息不会过大或过小，我们对注入的语义信息和原本的物品嵌入向量都进行了 L2 范数归一化。事实证明，这确实能够大幅度提升推荐系统的性能。

5.3 对比分析

		precision@5	recall@5	ndcg@5	precision@10	recall@10	ndcg@10
baseline		0.2966	0.06604	0.3110	0.2532	0.1094	0.2829
TransE	**only KG**	0.3226	0.07585	0.3385	0.2743	0.1239	0.3083
	concat	0.3204	0.07696	0.3314	0.2830	0.1319	0.3118
	add	0.3266	0.07494	0.3372	0.2794	0.1263	0.3091
	multiply	0.3177	0.07428	0.3268	0.2765	0.1237	0.3033
TransR	**only KG**	0.3114	0.07438	0.3259	0.2727	0.1212	0.3029
	concat	0.3195	0.07598	0.3272	0.2823	0.1325	0.3087
	add	0.3159	0.07128	0.3295	0.2754	0.1219	0.3051
	multiply	0.3168	0.07508	0.3229	0.2756	0.1233	0.3004

表 3: 不同注入方法及嵌入模型的效果

表 3 中的数据均来自与训练过程中出现的最优结果而非训练的最终结果。因为无论哪种模型均出现了不同程度的过拟合，因此训练的最终结果并不是最优结果。

可以看到，我们通过知识图谱注入的方式，能够相当有效地提升推荐系统的性能。而嵌入模型的对比中，TransE 模型的性能要优于 TransR 模型。这可能是 TransE 模型更简单，参数更少，更容易训练。也可能是因为 TransR 模型虽然更复杂，能够处理一些 TransE 处理不了的复杂关系，但是我们数据集中的关系并不复杂，或者我们数据集太小导致没有足够的数据来学习这些复杂的转换矩阵，那么 TransR 的性能可能就不如 TransE。所以我们下面的讨论主要根据 TransE 开展。

由于训练出的实体向量和数量级上存在差距，我们在进行注入时进行了归一化，这大幅度提高了效果。在 precision@5、recall@5 和 ndcg@5 指标上，通过相加的注入方式表现最好，其次是拼接的注入方式。在 precision@10、recall@10 和 ndcg@10 指标上，又变成了注入方式表现最好，其次是相加注入方式。

此处拼接采用的是直接拼接的方式，使得嵌入向量的维度翻倍，对应的，用户的嵌入向量也需要翻倍，这在某种意义上是增加了嵌入向量的维度，可能对于相乘和相加的方式而言不够公平。但是，我们认为这种方式更加符合直觉。我们也试验了先拼接再通过一个全连接网络将嵌入向量的维度降低一半的方式，但是效果并不好，因此我们最终选择了直接拼接的方式。事实证明，即使将直接相加和直接相乘的嵌入向量的维度翻倍，结果也并没有发生什么变化。

值得注意的是，我们仅仅使用知识图谱的方法就能够将推荐系统的性能提升到接近甚至在部分指标上超过融合了协同过滤和知识图谱算法的推荐系统。这说明，知识图谱的确能够为推荐系统提供有效的信息，能够提升推荐系统的性能。但同时也在侧面说明了知识图谱所能提供的信息和用户交互所能够提供的信息较为接近，导致二者的融合并没有产生预期的”1+1>2” 的效果。

6 实验总结

本次实验中，我们实现了一个基于知识图谱与系统过滤的推荐系统。我们首先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。之后，我们基于用户交互数据进行了个性化电影推荐。

我们通过实验验证了知识图谱的有效性，同时也发现了知识图谱的不足之处。在实验中，我们发现，知识图谱的构建与抽取是一个非常耗费时间耗费计算资源的过程，这也导致了知识图谱的构建成本较高。但与此同时，我们耗费大量计算资源抽取到的有效信息并不多，以至于很容易就能出现过拟合的现象。同时，知识图谱中的实体和关系的质量参差不齐，这导致了知识图谱的有效性受到了一定的影响。这些问题共同导致了没有能够使得加入知识图谱数据后的推荐系统的性能有预期的大幅度提升。