Web 信息处理与应用实验 2 报告

翁屹禾 PB20000017
傅申 PB20000051
侯博文 PB20000054

目录

1. 阶段一: 图谱抽取

该阶段的代码结构如下:

```
stage-1
├── data
│   ├── douban2fb.txt
│   ├── freebase_douban.gz
│   ├── Movie_id.csv
│   ├── Movie_tag.csv
│   └── kg_extracted.txt    # 提取的图谱
└── kg_extraction.py        # 图谱抽取代码
```

1.1. 获取豆瓣电影对应的实体

在给出的链接信息文件 `douban2fb.txt` 中, 提供了豆瓣电影 ID 到图谱实体 ID 之间的映射关系. 因此, 可以通过该文件获取豆瓣电影对应的实体.

```python
def load_movie_entities() -> set[str]:
    # load movie entities
    entities = set()
    with open(LINK_INFO_PATH, "r") as f:
        for line in f:
            line = line.strip()
            entities.add(f"{ENTITY_PREFIX}{line.split()[-1]}>")
    return entities
```

1.2. 抽取一跳子图

首先, 获取所有以电影实体作为头实体的三元组, 作为一跳子图:

```
movie_entities = load_movie_entities()
first_hop = extract_subgraph(movie_entities)
```

其中 `extract_subgraph()` 函数的实现如下, 它约束了实体的前缀, 以保证抽取的知识图谱更加精简:

```
def extract_subgraph(entities: set[str]) -> kg_t:
    # load knowledge graph with only provided entities
    kg = set()
    with gzip.open(KG_PATH, "rb") as f:
        for line in process(f, KG_SIZE):
            line = line.strip()
            triplet = tuple(line.decode().split()[:3])
            if triplet[0] in entities and triplet[2].startswith(ENTITY_PREFIX):
                kg.add(triplet)
    return kg
```

然后过滤一跳子图, 只保留至少出现在 20 个三元组中的实体, 同时只保留出现超过 50 次的关系:

```
entity_count, relation_count = kg_info(first_hop)
first_hop = filter(
    first_hop,
    lambda triplet: entity_count[triplet[0]] > 20
                    and entity_count[triplet[2]] > 20
                    and relation_count[triplet[1]] > 50,
)
```

其中 `kg_info()` 函数用于统计实体和关系的出现次数, `filter()` 函数用于过滤子图:

```
def kg_info(kg: kg_t):
    """get the entity and relation count of a knowledge graph"""
    entity_count: dict[str, int] = {}
    relation_count: dict[str, int] = {}
    for start, relation, end in kg:
        entity_count[start] = entity_count.get(start, 0) + 1
        entity_count[end] = entity_count.get(end, 0) + 1
        relation_count[relation] = relation_count.get(relation, 0) + 1
    return entity_count, relation_count


def filter(kg: kg_t, filter_fn: Callable[[kg_entry_t], bool]):
    filtered = set()
    for triplet in process(kg, len(kg)):
        if filter_fn(triplet):
            filtered.add(triplet)
    return filtered
```

在实际运行中, 得到的子图包含了 771 个实体和 31 个关系, 共 24851 个三元组.

1.3. 抽取二跳子图

首先, 使用 `second_hop = hop(first_hop)` 来获取二跳子图, 其中 `hop()` 通过将子图中所有出现过的实体作为头实体, 重新抽取子图:

```
def hop(kg: kg_t):
```

```
    entities = {triplet[0] for triplet in kg}
    entities.update({triplet[2] for triplet in kg})
    return extract_subgraph(entities)
```

然后, 对二跳子图进行处理, 先过滤掉出现超过两万次的实体和出现少于 50 次的关系:

```
entity_count, relation_count = kg_info(second_hop)
second_hop = filter(
    second_hop,
    lambda triplet: entity_count[triplet[0]] < 20000
    and entity_count[triplet[2]] < 20000
    and relation_count[triplet[1]] > 50,
)
```

然后再次过滤, 只保留出现大于 15 次的实体和出现大于 50 次的关系:

```
entity_count, relation_count = kg_info(second_hop)
second_hop = filter(
    second_hop,
    lambda triplet: entity_count[triplet[0]] > 15
                    and entity_count[triplet[2]] > 15
                    and relation_count[triplet[1]] > 50,
)
```

最后将得到的二跳子图写入文件:

```
with open(OUTPUT_KG_PATH, "w") as f:
    for triplet in process(second_hop):
        f.write(" ".join(triplet) + "\n")
```

1.4. 运行结果

程序的运行过程如下图所示, 最终得到的子图包含 1938 个实体, 56 个关系, 共 43711 个三元组:

```
Loading movie entities ...
100%|| 578/578 [00:00<00:00, 1245150.34it/s]
Done, size: 578

First hop ...
100%|| 395577070/395577070 [13:06<00:00, 503176.11it/s]
Done, size: 125372
Filtering first hop ...
100%|| 125372/125372 [00:00<00:00, 1246916.27it/s]
Done, 771 entities, 31 relations, size: 24851

Second hop ...
100%|| 395577070/395577070 [14:52<00:00, 443100.84it/s]
Done, size: 104742330
Filtering second hop ...
100%|| 104742330/104742330 [01:16<00:00, 1361481.01it/s]
100%|| 762594/762594 [00:01<00:00, 671881.79it/s]
Done, 1938 entities, 56 relations, size: 43711

Saving extracted knowledge graph ...
43711it [00:00, 637306.76it/s]
```

图 1　阶段一运行结果

2. 阶段二: 图谱推荐

该阶段的代码结构如下:

```
stage-2
├── data
│   ├── Douban
│   │   ├── entity_map.txt    # 实体映射
│   │   ├── kg_final.txt      # 映射后的知识图谱
│   │   ├── relation_map.txt  # 关系映射
│   │   ├── test.txt
│   │   └── train.txt
│   ├── douban2fb.txt
│   ├── kg_extracted.txt      # 阶段一中提取的知识图谱
│   ├── movie_id_map.txt
│   └── user_id_map.txt
├── data_loader/*
├── model/*
├── parser/*
├── trained_model/*
├── utils/*
├── main_Embedding_based.py
├── main_GNN_based.py
├── main_KG_free.py
└── mapping.py                # 实体和关系映射代码
```

2.1. 映射知识图谱中的实体和关系

首先, 加载知识图谱 `kg_extracted.txt`、链接信息 `douban2fb.txt` 和电影 ID 到索引值的映射关系 `movie_id_map.txt`.

```
raw_kg = load_raw_kg()
id_to_entity = load_link_info()
id_to_index = load_id_map()
```

然后, 先将电影 ID 对应的实体映射到相应的索引值:

```
entity_to_index = {}
relation_to_index = {}
for id, entity in id_to_entity.items():
    entity_to_index[entity] = id_to_index[id]
```

再将剩余的实体和关系映射到相应的索引值:

```
entity_index = max(id_to_index.values()) + 1
relation_index = 0
for start, relation, end in raw_kg:
    if start not in entity_to_index:
        entity_to_index[start] = entity_index
        entity_index += 1
    if relation not in relation_to_index:
        relation_to_index[relation] = relation_index
        relation_index += 1
    if end not in entity_to_index:
        entity_to_index[end] = entity_index
        entity_index += 1
```

最后, 将实体映射、关系映射和映射后的知识图谱写入文件:

```
with open(ENTITY_MAP_PATH, "w") as f:
    for entity, index in entity_to_index.items():
        f.write(entity + " " + str(index) + "\n")
with open(RELATION_MAP_PATH, "w") as f:
    for relation, index in relation_to_index.items():
        f.write(relation + " " + str(index) + "\n")

with open(FINAL_KG_PATH, "w") as f:
    for triplet in map_kg(raw_kg, entity_to_index, relation_to_index):
        f.write(" ".join(triplet) + "\n")
```

2.2. 基于图嵌入的模型

2.2.1. 数据加载部分

数据加载部分位于 `stage-2/dataloader/loader_Embedding_based.py` 文件内, 需要补全 `construct_data()` 函数.

一、添加逆向三元组: 使用 `rename()` 函数将深拷贝的三元组中的头实体和尾实体交换, 再给它们的关系加上 `n_relations`, 最后将源三元组和逆向三元组连接起来.

```
inverted_kg_data = copy.deepcopy(kg_data)
inverted_kg_data = inverted_kg_data.rename({"h": "t", "t": "h"}, axis=1)
inverted_kg_data["r"] += max(kg_data["r"]) + 1
self.kg_data = pd.concat(
    [kg_data, inverted_kg_data],
    axis=0,
    ignore_index=True
)
```

二、计算关系数, 实体数和三元组数量.

```
self.n_relations = self.kg_data["r"].max() + 1
self.n_entities = self.kg_data["h"].max() + 1
self.n_kg_data = self.kg_data.shape[0]
```

三、构建字典 `kg_dict` 和 `relation_dict`:

```
self.kg_dict = collections.defaultdict(list)
self.relation_dict = collections.defaultdict(list)
for _, (h, r, t) in self.kg_data.iterrows():
    self.kg_dict[h].append((t, r))
    self.relation_dict[r].append((h, t))
```

2.2.2. 模型搭建部分

模型搭建部分位于 `stage-2/model/Embedding_based.py` 文件内, 需要补全相关代码, 实现 TransE 和 TransR 算法.

`calc_kg_loss_TransE()` 函数和 `calc_kg_loss_TransR()` 函数分别根据 TransE 算法和 TransR 算法计算嵌入的损失函数, 需要进行补全.

TransE 算法

TransE 算法的基本思想是: 将实体和关系分别映射到对应空间中, 使得头实体和关系的和接近尾实体, 即 $\boldsymbol{h}+\boldsymbol{r}\approx\boldsymbol{t}$. 具体而言, 需要最小化下面的损失函数:

$$\mathscr{L}=\sum_{t_r\in T_r}\sum_{t_r'\in T_r'}\max(0,\gamma+f(t_r)-f(t_r'))$$

其中, 得分 $f(\cdot)$ 计算了 $\boldsymbol{h}+\boldsymbol{r}$ 和 $\boldsymbol{t}$ 的距离. 换而言之, 我们需要尽可能地使得负样例的得分大于正样例的得分. 如果采用 BPR 损失函数, 则有:

$$\mathscr{L}_{\text{BPR}}=-\sum_{t_r\in T_r}\sum_{t_r'\in T_r'}\log(\sigma(f(t_r')-f(t_r)))$$

基于上述思想, 可以完成 `calc_kg_loss_TransE()` 函数.

一、计算头实体、关系和正负样例尾实体在空间中的嵌入:

```
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)
```

二、对这些嵌入进行 L2 归一化:

```
r_embed = r_embed / torch.norm(r_embed, dim=1, keepdim=True)
h_embed = h_embed / torch.norm(h_embed, dim=1, keepdim=True)
pos_t_embed = pos_t_embed / torch.norm(pos_t_embed, dim=1, keepdim=True)
neg_t_embed = neg_t_embed / torch.norm(neg_t_embed, dim=1, keepdim=True)
```

三、计算正负样例的得分 $f(\cdot)$, 采用 L2 距离:

```
pos_diff = h_embed + r_embed - pos_t_embed  # (kg_batch_size, embed_dim)
neg_diff = h_embed + r_embed - neg_t_embed  # (kg_batch_size, embed_dim)
pos_score = torch.norm(pos_diff, dim=1) ** 2  # (kg_batch_size)
neg_score = torch.norm(neg_diff, dim=1) ** 2  # (kg_batch_size)
```

四、计算 BPR 损失函数:

```
kg_loss = -F.logsigmoid(neg_score - pos_score).mean()
```

TransR 算法

TransR 算法的基本思想与 TransE 算法类似. TransR 算法在实体空间和**多个**关系空间中建模实体和关系. 对于每个三元组 (h,r,t), 将实体空间中的实体 $\boldsymbol{h}$ 和 $\boldsymbol{r}$ 通过矩阵 $\boldsymbol{W}_r$ 投影到 r 关系空间中, 即

$$\boldsymbol{h}_r=\boldsymbol{h}\boldsymbol{W}_r,\boldsymbol{t}_r=\boldsymbol{t}\boldsymbol{W}_r$$

然后, 在 r 关系空间中, 尽可能使得 $\boldsymbol{h}_r+\boldsymbol{r}\approx\boldsymbol{t}_r$. 即尽可能使负样例得分大于正样例的得分.

一、计算头实体、关系和正负样例尾实体在空间中的嵌入, 获取 r 关系空间的投影矩阵 $\boldsymbol{W}_r$:

```
r_embed = self.relation_embed(r)  # (kg_batch_size, relation_dim)
W_r = self.trans_M[r]  # (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)
```

二、将头实体和尾实体投影到 r 关系空间中:

```
def trans_r_mul(W_r, entity_embed):
    """
    计算 TransR 中的投影嵌入
    """
    return torch.matmul(entity_embed.unsqueeze(1), W_r).squeeze(1)

r_mul_h = trans_r_mul(W_r, h_embed)  # (kg_batch_size, relation_dim)
r_mul_pos_t = trans_r_mul(W_r, pos_t_embed)  # (kg_batch_size, relation_dim)
r_mul_neg_t = trans_r_mul(W_r, neg_t_embed)  # (kg_batch_size, relation_dim)
```

三、对这些嵌入进行 L2 归一化:

```
r_embed = r_embed / torch.norm(r_embed, dim=1, keepdim=True)
r_mul_h = r_mul_h / torch.norm(r_mul_h, dim=1, keepdim=True)
r_mul_pos_t = r_mul_pos_t / torch.norm(r_mul_pos_t, dim=1, keepdim=True)
r_mul_neg_t = r_mul_neg_t / torch.norm(r_mul_neg_t, dim=1, keepdim=True)
```

四、计算正负样例的得分 $f(\cdot)$, 采用 L2 距离:

```
pos_diff = r_mul_h + r_embed - r_mul_pos_t  # (kg_batch_size, relation_dim)
neg_diff = r_mul_h + r_embed - r_mul_neg_t  # (kg_batch_size, relation_dim)
pos_score = torch.norm(pos_diff, dim=1) ** 2  # (kg_batch_size)
neg_score = torch.norm(neg_diff, dim=1) ** 2  # (kg_batch_size)
```

五、计算 BPR 损失函数:

```
kg_loss = -F.logsigmoid(neg_score - pos_score).mean()
```

在后面的计算中, 还需要为**物品嵌入**注入**实体嵌入**的语义信息, 可以采用相加、逐元素相乘和拼接等方法:

```
def inject(item_embed, item_kg_embed):
    """
    为 物品嵌入 注入 实体嵌入 的语义信息, 可以采用相加/逐元素相乘/拼接等方式
    """
    return item_embed + item_kg_embed
    # return item_embed * item_kg_embed
    # return torch.cat([item_embed, item_kg_embed], dim=1)
```

如果使用拼接的方式, 则还需要对 `calc_cf_loss()` 和 `calc_score()` 中的 `user_embed` 扩充维度, 以保证维度一致

```
user_embed = torch.cat([user_embed, user_embed], dim=1)
```

2.3. 实验结果

我们运行了图谱嵌入模型的训练代码, 对比了不同的算法和注入语义信息方式对模型性能的影响, 同时与 Baseline (MF) 进行比较, 结果如下表所示 (表中为训练过程中最好的结果):

算法 + 语义信息注入方式		Recall@5	Recall@10	NDCG@5	NDCG@10
图谱嵌入 w/ TransE	相加	0.0676	0.1156	0.3104	0.2904
	逐元素相乘	0.0608	0.0999	0.2824	0.2557
	拼接	0.0676	0.1159	0.3102	0.2905
图谱嵌入 w/ TransR	相加	0.0658	0.1126	0.3118	0.2849
	逐元素相乘	0.0627	0.1087	0.2916	0.2690
	拼接	0.0657	0.1127	0.3117	0.2853
Baseline(MF)		0.0660	0.1094	0.3110	0.2829

表 1　实验结果对比

根据结果可以看出, TransE 和 TransR 算法在不同的指标上各有优劣, 而不同的语义信息注入方式对结果的影响比较一致, 即 “相加” ≈ “拼接” > “逐元素相乘”. 在与 Baseline 的比较中, TransE 和 TransR 在采用相加/拼接的语义信息注入方式时, 可以取得稍微好一点的结果.