강화학습을 사용한 도서 추천 알고리즘

201821387 민경민
202121633 송지윤
202121656 장서연
202120828 조세은

Catholic University of Korea
2023 Reinforcement Learning

Contents

Feedback

01 Feedback

콜드 스타트 문제

- 새로운 유저에 대한 정보 부족

  → 추천이 어려운 상태

Recommendation System with Machine Learning

Contents-based Collaborative Filtering

Collaborative Collaborative Filtering

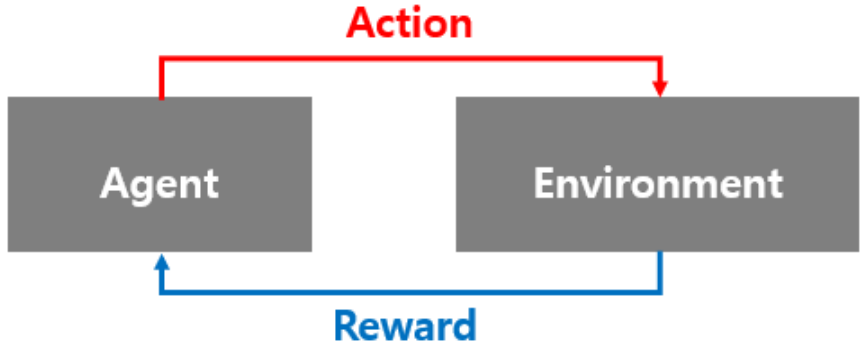
1. 사용자 통계 정보 활용
2. 인기 아이템 우선 추천

강화학습을 사용한 추천 시스템

- 지속적인 탐색
- 사용자 피드백 수집

→ 최적의 추천 전략 학습

✈ 장기적 이익 극대화 및 개인화 추천 가능



Recommendation System with Reinforce Learning

01 Feedback

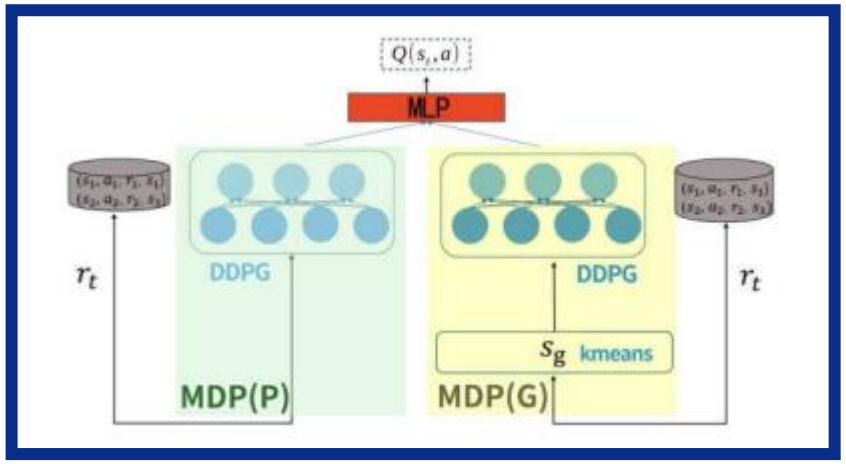
"개인 사용자 및 그룹 선호도를 고려한 강화학습 기반 추천 시스템"

1. 개인 사용자 기반 DDPG Network

   → 개인 사용자의 선호를 파악하고 변화를 캡처

2. 사용자 그룹 기반의 DDPG Network

   → 현재 유행중인 이슈, 트렌드를 파악하고 추천에 반영



Recommendation System with Reinforce Learning

- 콜드 스타트 문제
- 데이터 희소성 문제

✈ 능동적으로 사용자에게 맞춤형 추천

MDP Modeling

02

MDP Modeling

도서 추천 알고리즘

User가 다음에 읽을 도서 **선택**

User가 과거에 관심을 보였던 **도서**

추천한 도서 목록에 대해 User가 남긴 **Rating**

- 4일 때 0.5, 5일 때 1, 나머지 경우일 때 -1
- 추천의 만족도를 높이기 위해 4점 이상만 고려

Preprocessing

03 Preprocessing

데이터셋 전처리

- Amazon: Book.csv Dataset of Amazon Review Data (2018)

전처리 전 : 83386
전처리 후 : 30382

```python
def history_upper20(data):
    a = pd.DataFrame(data['user'].value_counts())
    user_indices = a[a['user']>=20].index
    data_20 = data[data['user'].isin(user_indices)]

    return data_20
```

history_upper20

→ 책을 20권 이상 읽은 User

```python
def item_upper10(data):
    item_counts = pd.DataFrame(data['item'].value_counts())
    item_indices = item_counts[item_counts['item']>=10].index
    data_10 = data[data['item'].isin(item_indices)]
    return data_10
```

item_upper10

→ User가 10번 이상 읽은 책

✈ metadata 활용

✈ user와 item에 해당하는 title, category, timestamp를 매칭

Implementation

04 Implementation

코드 구성

- 전체 구조

01 data_prep

history_upper20

item_upper10

02 meta_applied

user, item, title, category, timestamp

→ 30382

03 DDPG_recsys

1. Data Preprocessing
2. Indexing & Embedding
3. State Representation
4. Actor-Critic Model
5. PER Buffer

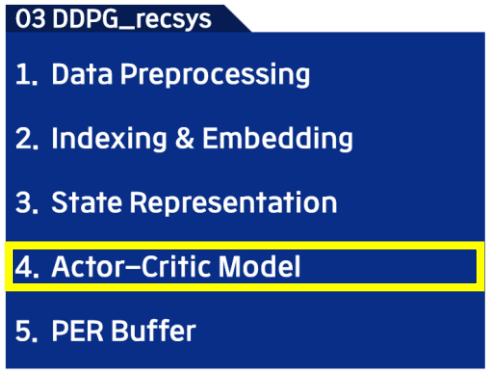
→ offline Env2 → OUNoise

04 Implementation

코드 구성

- Actor Network

  → Action 선택




```python
class Actor(torch.nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super(Actor, self).__init__()

        self.drop_layer = nn.Dropout(p=0.3)
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, state):
        x = F.relu(self.linear1(state.to(device)))
        x = self.drop_layer(x)
        x = F.relu(self.linear2(x))
        x = self.drop_layer(x)
        x = self.linear3(x)  # in case embeds are standard scaled / wiped using PCA whitening
        return x  # state = self.state_rep(state)
```

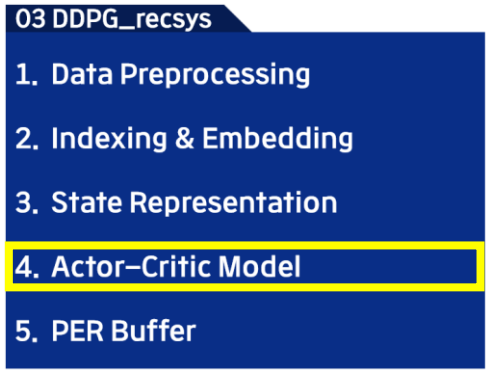
- ReLU 추가
- Dropout 추가

04 Implementation

코드 구성

- Actor Network

  → Action 평가




```python
class Critic(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim):
        super(Critic, self).__init__()

        self.drop_layer = nn.Dropout(p=0.3)
        self.linear1 = nn.Linear(input_dim + output_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = nn.Linear(hidden_dim, 1)

    def forward(self, state, action):
        x = torch.cat([state.to(device), action.to(device)], 1)
        x = F.relu(self.linear1(x))
        x = self.drop_layer(x)
        x = F.relu(self.linear2(x))
        x = self.drop_layer(x)
        x = self.linear3(x)
        return x
```

- State, Action 결합
- ReLU 추가
- Dropout 추가

04 Implementation

코드 구성

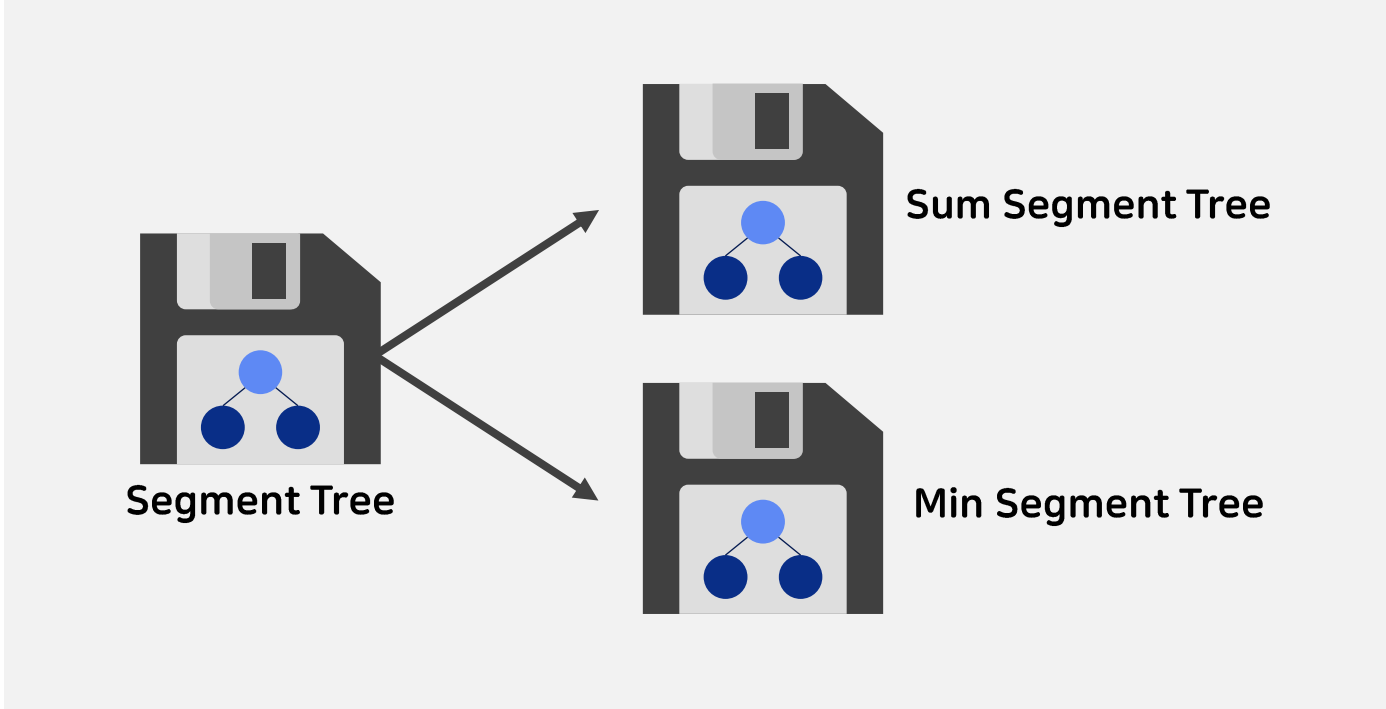
- PER Buffer

  → 경험 재사용

03 DDPG_recsys

1. Data Preprocessing
2. Indexing & Embedding
3. State Representation
4. Actor-Critic Model
5. PER Buffer

- State, Action 결합
- ReLU 추가
- Dropout 추가



04 Implementation

코드 구성

- PER Buffer

  → 경험 재사용

03 DDPG_recsys

1. Data Preprocessing
2. Indexing & Embedding
3. State Representation
4. Actor-Critic Model
5. PER Buffer

state, action, reward, next_state를 저장

```
class ReplayBuffer:
    def store(self,
              state: np.float32,
              action: np.float32,
              reward: np.float32,
              next_state: np.float32,
              done: np.int8):
        self.state_buffer[self.save_count] = state
        self.action_buffer[self.save_count] = action
        self.reward_buffer[self.save_count] = reward
        self.next_state_buffer[self.save_count] = next_state
        self.done_buffer[self.save_count] = done

        self.save_count = (self.save_count + 1) % self.buffer_size
        self.current_size = min(self.current_size + 1, self.buffer_size)

    def batch_load(self):
        indices = np.random.randint(self.current_size, size=self.batch_size)
        return dict(
            states=self.state_buffer[indices],
            actions=self.action_buffer[indices],
            rewards=self.reward_buffer[indices],
            next_states=self.next_state_buffer[indices],
            dones=self.done_buffer[indices])
```

- State, Action 결합
- ReLU 추가
- Dropout 추가

04 Implementation

코드 구성

- PER Buffer

  → 경험 재사용

03 DDPG_recsys

1. Data Preprocessing
2. Indexing & Embedding
3. State Representation
4. Actor-Critic Model
5. PER Buffer

```
class ReplayBuffer:
    def store(self,
              state: np.float32,
              action: np.float32,
              reward: np.float32,
              next_state: np.float32,
              done: np.int8):
        self.state_buffer[self.save_count] = state
        self.action_buffer[self.save_count] = action
        self.reward_buffer[self.save_count] = reward
        self.next_state_buffer[self.save_count] = next_state
        self.done_buffer[self.save_count] = done

        self.save_count = (self.save_count + 1) % self.buffer_size
        self.current_size = min(self.current_size + 1, self.buffer_size)

    def batch_load(self):
        indices = np.random.randint(self.current_size, size=self.batch_size)
        return dict(
            states=self.state_buffer[indices],
            actions=self.action_buffer[indices],
            rewards=self.reward_buffer[indices],
            next_states=self.next_state_buffer[indices],
            dones=self.done_buffer[indices])
```

- state, action 결합
- ReLU 추가
- dropout 추가

저장된 내용을 dictionary 형태로 반환

04 Implementation

코드 구성

- offline Env2

  → Simulation

Data: userID, DataLoader
Result: recommended list, updated memory, updated reward

```
Data = next(iter(Dataloader))
memory = Data['item']
User history = Data['userID']
items = User history['item']
ratings = User history['rating']
related books = [ ]
for item, ratings in zip(items, ratings):
    if ratings >= 4:
        related books.append(items)
```

책 제목

사용자

평점

04 Implementation

코드 구성

- offline Env2

  → Simulation

```
10 while until checking the last state of Data do
11     state list = [ ]
12     if ratings >= 4:  → 시간 순으로 정렬 → book_ix 생성
13         book= related books(User history["item"].sort(by=timestamp))
14         book ix= book.index
15             if ratings['book ix'] == 5:
16                 reward = 1
17             if ratings['book ix'] == 4:
18                 reward = 0.5
19         state list.append(user history["item"])
20         update memory(by=action)
21     else if category matchs:  → 카테고리가 같으면 reward 0
22         reward = 0
23     else:
24         reward = -1
```

04 Implementation

결과 소개

- State가 동일하더라도 Action이 달라짐을 발견

  → DDPG의 특성이 반영된 것으로 예상

User	Action
20	[11616, 5475, 25856, 82644, 70125]
20	[5475, 11616, 70125, 25856, 82644]
20	[5475, 11616, 70125, 82644, 7628]
20	[11616, 5475, 82644, 25856, 81851]
20	[11616, 5475, 82644, 81772, 81851]

- **Off policy**
  - 빠른 피드백 후 고치며 학습 가능
- **Replay Buffer**
  - 과거의 유의미한 경험으로 학습 가능

✈ DDPG를 사용해 총 8명의 적은 User로도 충분한 학습 가능

04 Implementation

결과 소개

- Hyperlarameter Tuning

  → Dropout & Category 추가 후 성능 상승

Dropout & Category	DCG	NDCG	Precision	Precision_Topk
Dropout 0.25	64.31	0.89	0	0.9
Dropout 0.3	73.64	0.95	0	0.8
Dropout 0.25 Add Category	65.97	0.93	0	0.85
Dropout 0.3 Add Category	71.29	0.93	0	0.85

04 Implementation

결과 소개

- 적은 양의 데이터셋으로도 좋은 성능을 내는 RL 추천 시스템

  → 중복 제거 후 총 8명의 user로 이루어진 데이터셋으로 학습

User_index	DCG		NDCG		Precision_rating_over_4	
	ML	RL	ML	RL	ML	RL
16	72.56	80.58	0.93	0.98	0.76	1
0	69.41	75.01	0.91	0.99	0.71	0.8
20	47.13	49.06	0.79	0.83	0.55	0.68
12	35.97	37.88	0.71	0.72	0.51	0.6

✈ 모든 항목에서 머신러닝보다 강화학습의 성능이 뛰어남

04 Implementation

기대 효과 및 아쉬운 점

✓ **콜드 스타트 문제 해결**

- 지속적인 탐색 및 사용자 피드백 수집
- 최적의 추천 전략 학습을 통한 개인화 추천

✓ **데이터 희소성 문제 해결**

- 적은 데이터로는 머신러닝 학습 불가능

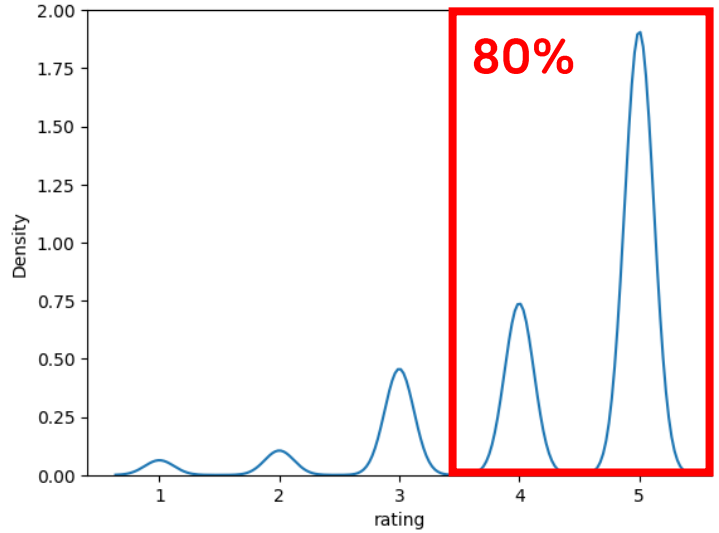
✓ **변화하는 취향 문제 해결**

- 강화학습은 이를 고려할 수 있음

✓ 리소스 부족 문제

✓ Rating 분포 문제

평점 5 : 17726
평점 4 : 6854
평점 3 : 4239
평점 2 : 975
평점 1 : 588



Citation

Citation

Amazon review data (nijianmo.github.io)

shani05a.dvi (jmlr.org)

https://dcollection.ewha.ac.kr/public_resource/pdf/000000201851_20231127103016.pdf

Thank You