알고리즘 응용(00) Lab09

201802161 조은빈

1. Goal

- Greedy search 함수 구현하기
- Beam search 함수 구현하기
- 랜덤값으로 결과 비교해보기

2. 코드 분석

```
1    import numpy as np
2    from numpy import argmax
3
4    def greedy_search():
5        score = 1   # score 1로 초기화
6        sequences = list()
7        for i in range(10):
8            np.random.seed(int(score * 10))
9            data = np.random.rand(5)    # 랜덤값 생성
10           index = argmax(data)       # 랜덤값 중 가장 큰 값의 인덱스를 반환
11           score = score * data[index]     # 가장 큰 값과 score을 곱해서 score 계산
12           sequences.append(index)     # 가장 큰 값의 인덱스를 sequences에 추가
13       sequences = [sequences, score]
14       return sequences
```

greedy_search는 5개의 랜덤값 중에서 가장 큰 값과 score를 곱해서 score를 계산한다. 가장 큰 값의 인덱스를 이어 결과를 출력한다. seed에는 실수가 들어갈 수 없기 때문에 score에 10을 곱하고 int로 형변환 시켜서 seed로 사용한다. 10을 곱하는 이유는 score가 0과 1 사이의 실수일 경우 int로 형변환하면 0이 되어버리기 때문이다.

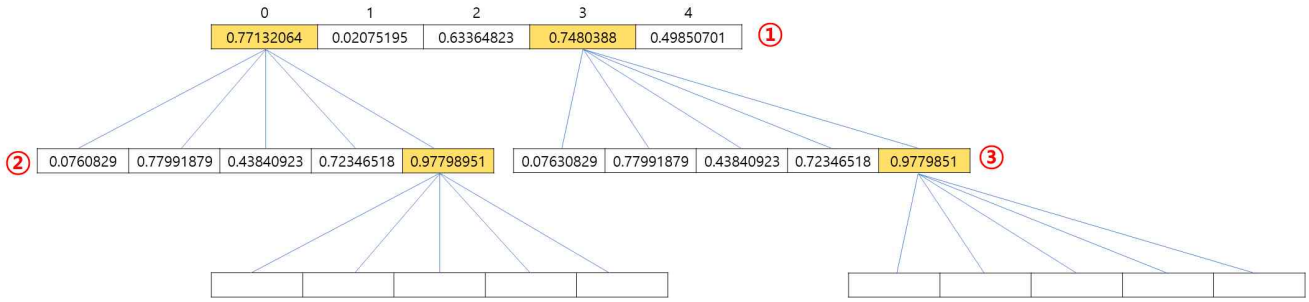
0	1	2	3	4	
0.77132064	0.02075195	0.63364823	0.7480388	0.49850701	sequences = 0
0.07630829	0.77991879	0.43840923	0.7234518	0.97798951	sequences = 0, 4
0.07630829	0.77991879	0.43840923	0.7234518	0.97798951	sequences = 0, 4, 4

과거와 미래를 고려하지 않고 매 순간마다 가장 큰 값을 찾는다.

```python
def beam_search(k):
    score = 1   # score 1로 초기화
    sequences = [[list(), score]]
    for q in range(10):
        all_candidates = list()
        for i in range(len(sequences)):
            seq, score = sequences[i]   # 각 세트의 시퀀스와 score 분리
            np.random.seed(int(score * 10))
            data = np.random.rand(5)     # 각 세트의 score를 seed로 랜덤값 생성
            for j in range(5):
                candidate = [seq + [j], score * data[j]]    # 생성한 랜덤값의 score 계산
                all_candidates.append(candidate)    # score 세트를 all_candidates에 추가
        ordered = sorted(all_candidates, key=lambda tup: tup[1], reverse = True)
        sequences = ordered[:k]      # all_candidate를 내림차순으로 정렬해 k개만 남기고 삭제
    return sequences
```

beam_search는 선택 가능한 모든 경우의 수 중 높은 점수의 k개를 찾는다.

5개의 랜덤값에 score를 곱해서 score를 구한다. score가 큰 순서대로 k개를 뽑는다. k개의 세트에서 각 세트의 score를 seed로 다시 5개의 랜덤값을 뽑아낸다. 그리고 그 랜덤값의 score를 계산한다. 이 과정을 반복한다. 즉 아래와 같다.



k=2라고 가정한다. 우선 초기 score는 1이다. score를 seed로 넣어 5개의 랜덤값을 구한 것이 ①이다. ①의 값에 score를 곱해서 각각의 score를 계산한다. 각각의 인덱스와 score를 하나로 묶어 all_candidates에 추가한다. score를 기준으로 내림차순으로 all_candidates의 요소들을 정렬한다. score가 큰 두 요소를 제외하고 나머지는 삭제한다. 두 요소는 인덱스 시퀀스와 score가 하나로 묶여있는 세트이다. 두 세트가 있는데 여기서 각각의 score를 seed로 넣어 5개의 랜덤값을 구한다. 첫 번째 세트의 score로 구한 랜덤값이 ②이다. 두 번째 세트의 score로 구한 랜덤값이 ③이다. ②의 랜덤값과 score를 곱한 값, ③의 랜덤값과 score를 곱한 값 총 10개의 세트에서 가장 score가 높은 2개의 세트만 남기고 나머지는 삭제한다. 이 과정을 10번 반복한다. 이를 통해 가장 점수가 높은 경로로 이어진다.

score를 seed로 넣어서 랜덤값을 구하는 이유는 greedy_search와 beam_search의 성능의 차이를 보기 위해서이다. 성능의 차이를 확인하기 위해서는 앞에서 선택한 것이 뒤의 선택에도 영향을 주는 데이터여야 한다.

```
33  def main():
34      print("greedy_search")
35      print(greedy_search())
36      print("₩nbeam_search")
37      beam_seq = beam_search(3)
38      for i in beam_seq:
39          print(i)
40
41  if __name__ == '__main__':
42      main()
```

main에서 greedy_search와 beam_search를 실행한다.

3. 실행 결과

- k = 3일 경우

```
C:₩Users₩ChoEunBin₩.conda₩envs₩homework₩python.exe
greedy_search
[[0, 4, 4, 4, 4, 4, 0, 0, 3, 3], 0.4642330111955245]

beam_search
[[0, 4, 4, 4, 4, 4, 0, 0, 1, 2], 0.46593937652594897]
[[0, 4, 4, 4, 4, 4, 2, 3, 3, 2], 0.46516326812927056]
[[0, 4, 4, 4, 4, 4, 0, 0, 3, 3], 0.4642330111955245]
Press any key to continue . . .
```

greedy_search와 beam_search의 점수를 비교하면 beam_search의 성능이 더 좋은 것을 알 수 있다.

- k = 1일 경우, greedy search와 beam search의 결과는 동일하다.

```
C:₩Windows₩system32₩cmd.exe
greedy_search
[[0, 4, 4, 4, 4, 4, 0, 0, 3, 3], 0.4642330111955245]

beam_search
[[0, 4, 4, 4, 4, 4, 0, 0, 3, 3], 0.4642330111955245]
계속하려면 아무 키나 누르십시오 . . .
```