




숙제 1의 Feedback

2023년



1. 숙제 개요

- 입력:
 - 파일 이름, Target 사용자, 참고인 수, 추천 항목 수? small.txt target n k
 - 추천할 사용자 이름: target \leftarrow int
 - 참조 사용자 수 n \leftarrow int
 - 추천할 항목의 수 k \leftarrow int
- 출력
 - Target 사용자의 (콘텐츠, 정규화된 점수)를 콘텐츠의 오름차순으로 출력 (10점)
 - Target과 유사도가 가장 높은 n명에 대해 (id, 유사도)를 유사도의 내림차순으로 출력 (15점)
 - Target과 유사도가 가장 높은 n명이 구매한 항목 중에서 target이 구매하지 않은 콘텐츠에 대해 점수의 내림차순으로 (콘텐츠, 점수)를 k개 출력. 점수가 같을 경우에는 콘텐츠의 이름이 작은 것을 먼저 출력. (25점)



2. 숙제에서 구현할 내용

- (사용자, 구매 리스트)를 저장할 데이터 구조 결정
- 별점을 정규화(normalization)
- **Target** 사용자의 정규화 결과를 출력 (정렬)
- 사용자들간의 유사도 계산 후 상위 **n**명을 결정
- **n**명의 구매 리스트에서 **k**개의 추천 콘텐츠 결정

3. (사용자, 구매 리스트)의 데이터 구조 결정

- 합리적인 구조
 - `ArrayList<HashMap<String, Double>>`
 - `User[] list ← class User {HashMap<String, Double> ...}`
 - `HashMap<Integer, HashMap<String, Double>>`

```
{1, {(A0: 1.667), (A1: 0.667), (B0: -2.333)}}
{2, {...}, {...}, {...}}
```



비효율적인 구조들(1)

- **HashMap<Integer, TreeMap<String, Double>>**
 - TreeMap을 사용하는 이유
 - Target 사용자의 (콘텐츠, 평가점수)를 "정렬"해서 출력
 - TreeMap을 사용하면 안되는 이유
 - Target을 제외한 나머지 99,999명의 사용자에게 대해서는 정렬할 필요가 없다.
 - 그런데 왜 콘텐츠의 이름으로 정렬해서 Map에 저장해야 하는가?
 - 사용자 수가 많을수록 입력 지연이 발생
- TreeMap을 사용하는 다른 예들
 - HashMap<Integer, User> → User class에 TreeMap<String, Content>
 - TreeMap<String, Double>[], TreeMap<String, UserReview>[]
 - ArrayList<TreeMap<String, Integer>> ...

비효율적인 구조들(2)

- **List user[], user[i] = new LinkedList<Contents>**
 - 유사도 검사시 선형 검색을 사용

```
for (int j = 0; j < user[target].size(); j++) {  
    s1 = (Contents) user[target].get(j);  
    sumTarget += (s1.score * s1.score);  
    for (int k = remem; k < user[i].size(); k++) {  
        s2 = (Contents) user[i].get(k);  
        cmp = al.compare(s1, s2);  
        if (cmp == 0) {  
            value += (s1.score * s2.score);  
            remem = k + 1;  
            break;  
        }  
    }  
}
```

Map으로 구현

```
Map<String, Double> map1 = ulist.get(target);  
for (Map.Entry<String, Double> entry : ulist.get(u2).entrySet()) {  
    if (map1.containsKey(entry.getKey()))  
        cp += entry.getValue() * map1.get(entry.getKey());  
}
```

일반화되지 않은 구조

- 일반화되지 않은 (이번 숙제에만 사용할 수 있는) 정보로 구성된 **class**

```
class People {  
  
    Map<String, Integer> Contents;  
  
    double normalized_score = 0 ;  
    int contents_count = 0, id;  
    double avg;  
    double similarity;  
  
    ...  
}  
  
User = new People[TotalNumber];
```

```
class Save{  
    Map<String, Integer> map = new LinkedHashMap<>();  
    Map<String, Double> normal = new TreeMap<>();  
    Map<Integer, Double> similarity = new HashMap<>();  
}  
  
Save user[] = new Save[Integer.parseInt(size)];
```

```
class User{  
    Double average;  
    Double scalar;  
    HashMap<String,Double> rating = new HashMap<>();  
}  
  
User users[]=new User[Nofuser];
```

4. 별점을 정규화

- 일반적인 방법

```
for (u = 0; u < N; u++) {    // normalize
    double sum = 0.0;
    int length = ulist.get(u).size();

    for (Map.Entry<String, Double> entry : ulist.get(u).entrySet())
        sum += entry.getValue();
    double average = sum / length;
    for (Map.Entry<String, Double> entry : ulist.get(u).entrySet())
        entry.setValue(entry.getValue() - average);
}
```

- **Stream**을 이용

- sum =
ulist.get(u).values().stream().mapToDouble(Double::doubleValue).sum();

다른 방법들

- **HashMap<Integer, HashMap<String, Integer>> userMap, normalization_userMap** 두 개를 유지
- **map.keySet()**과 **map.get()**을 별도로 사용

```
void nomalization() {  
    int count=0;  
    double sum=0;  
    for(Double j : map.values()) {  
        sum+= j;  
        count++;  
    }  
    sum=sum/count;  
    for(String str : map.keySet()) {  
        map.put(str, map.get(str)-sum);  
    }  
}
```

5. Target 사용자의 정규화 결과를 출력 (정렬)

■ **Comparator** 이용

```
class MyComparator implements Comparator<String>{  
    public int compare(String str1,String str2) {  
        int result=str1.substring(0,1).compareTo(str2.substring(0,1));  
        if(result==0) {  
            int num1=Integer.parseInt(str1.substring(1));  
            int num2=Integer.parseInt(str2.substring(1));  
            return num1-num2;  
        }  
        return result;  
    }  
}
```

결과를 추출하고 출력하는 방법

- HashMap에서 keySet() 만 추출한 후, 정렬 순서로 get()한 결과를 출력

```
List<String> normalSort = new ArrayList<>(user[target].normal.keySet());
normalSort.sort(new Comparator<String>() { ...});
for(String key: normalSort) {
    System.out.printf("(%s, %.3f)", key, user[target].normal.get(key));
}
```

- HashMap에서 (key, value)를 다른 클래스에 저장한 후, 정렬하여 출력

```
ArrayList<IRPair> irlist = new ArrayList<>(ulist.get(target).size());
for (Map.Entry<String, Double> entry : ulist.get(target).entrySet())
    irlist.add(new IRPair(entry.getKey(), entry.getValue()));
Collections.sort(irlist, new IRCompItem());
System.out.println("1. 사용자 " + target + "의 콘텐츠와 정규화 점수: ");
System.out.println("\t" + irlist + "\n");
```

다른 방법

- 전체 데이터를 정렬하자!

```
User[] matrix=new User[user_num];
...
while((line=buf.readLine())!=null) {
    buffer=line.split(" ");
    matrix[Integer.parseInt(buffer[0])].map.put(buffer[1],Double.parseDouble(buffer[2]));
}
for(int i=0;i<user_num;i++)
    matrix[i].nomalization();
...
for(int i=0;i<user_num;i++) {
    matrix[i].cosSimilality=matrix[user].CalCos(matrix[i]);
}

Arrays.sort(matrix);
```

```
class User implements Comparable<User>{
    Comparator a;
    int id;
    Map<String, Double> map;
    double cosSimilality=0;
    ...
}
```

6. 사용자들간의 유사도 계산 후 상위 n명을 결정

- 일반적인 방법

```
ArrayList<USPair> us = new ArrayList<>(N-1);  
double t_norm = calculate_norm(target);  
  
for (int u = 0; u < N; u++) {  
    if (u == target)  
        continue;  
    us.add(new USPair(u, cos_similarity(target, u, t_norm)));  
}  
us.sort();    // Natural order = similarity의 내림차순
```

```
public class USPair implements Comparable<USPair> {  
    int user;  
    double similarity;  
    ...  
}
```

비효율적인 방법들(1)

- **Target** 사용자의 **norm**을 사용자 수만큼 계산

```
for (Map.Entry<Integer, HashMap<String, Double>> entry : map.entrySet()) {  
    ...  
    for (Map.Entry<String, Double> contentEntry : entry.getValue().entrySet()) {  
        if (map.get(target).containsKey(contentEntry.getKey()))  
            molecule += contentEntry.getValue() * map.get(target).get(contentEntry.getKey());  
        num1 += contentEntry.getValue() * contentEntry.getValue();  
        for (Map.Entry<String, Double> targetEntry : map.get(target).entrySet())  
            num2 += targetEntry.getValue() * targetEntry.getValue();  
  
        double cosineSimilarity = 0.0;  
        if (!(num1 == 0 || num2 == 0))  
            cosineSimilarity = molecule / (Math.sqrt(num1) * Math.sqrt(num2));  
        cosineSimilarityMap.put(userNumber, cosineSimilarity);  
    }  
}
```

비효율적인 방법들(2)

- **TreeMap**을 사용하여 **similarity**의 내림차순으로 정렬 수행

```
Map<Double, Integer> similarTree = new TreeMap<>(Comparator.reverseOrder());

for (int i = 0; i < N; i++) {
    if (i == target) {
        continue;
    }
    double a = Similar(data, target, i);
    similarTree.put(a, i);
}
```

정렬할 용도로 **TreeMap**을 사용하는 것은 비추 → **PriorityQueue**를 사용하는 것을 고려

7. n명의 구매 리스트에서 k개의 추천 콘텐츠 결정

- 일반적인 방법

```
HashMap<String, Double> tmap = ulist.get(target);
HashMap<String, Double> irmap = new HashMap<>();
for (int u = 0; u < num_ref; u++) {
    for (Map.Entry<String, Double> entry : ulist.get(us.get(u).user).entrySet()) {
        double similarity = us.get(u).similarity;
        if (tmap.containsKey(entry.getKey())) continue;
        if (!irmap.containsKey(entry.getKey()))
            irmap.put(entry.getKey(), entry.getValue() * similarity);
        else
            irmap.put(entry.getKey(), irmap.get(entry.getKey()) + entry.getValue() * similarity);
    }
}
for (Map.Entry<String, Double> entry : irmap.entrySet())
    irlist.add(new IRPair(entry.getKey(), entry.getValue()));
Collections.sort(irlist, ...);
```


비효율적인 방법

- 집합 연산을 사용

```
TreeMap<String,Float>Recomand_Contents= new TreeMap<>();
TreeMap<String, Float>Target=UserList.get(target);
for(int key : Top_List.keySet()){
    TreeMap<String, Float> each=UserList.get(key);
    Set<String>t1=Target.keySet();    ← for문 밖으로
    Set<String>t2=each.keySet();
    Set<String>t3=new HashSet<>(t2);
    t3.removeAll(t1);                ← HashMap.contains()로 구현 가능하므로 불필요
    float xxx=Top_List.get(key);     ← 변수 이름 수정: similarity
    for(String str : t3){
        float recomand_value= xxx*each.get(str);
        if(Recomand_Contents.containsKey(str))
            Recomand_Contents.put(str, Recomand_Contents.get(str)+recomand_value);
        else Recomand_Contents.put(str,recomand_value);
    }
}
```

총평

- **HashMap**과 **TreeMap**의 성능에 대한 이해 부족
 - TreeMap은 sorted symbol table이 필요한 경우에만 제한적으로 사용할 것
- **Memory**를 무분별하게 사용하는 행위는 금지
 - Set의 남용, Map의 중복 사용 등
- **main()** 함수로만 구성된 프로그램들
- 변수 이름, 함수 이름, ...
 - One, Two, Three, class Function ...
- 불필요하게 중복되는 작업을 줄이는 것은 프로그래머의 의무





채점 결과

- **10점대: 20명**
- **20점대: 9명**
- **30점 이상: 47명**
- 제출하지 않았거나 오류 발생: **84명**
- 프로그램이 아닌 파일을 첨부(**-100점**): **2명**