Apriori 알고리즘 구현

컴퓨터소프트웨어학부
2016025996
조성진

Apriori 알고리즘을 구현하기 위하여 Itemset 과 support, size 를 가지는 Item 클래스를 생성했다. 가장 기본적인 Apriori 알고리즘의 방법에 따라 먼저 데이터베이스를 1 scan 한다. 첫 스캔 때 Itemset 의 크기가 1 인 모든 frequent pattern 을 구하고, 그 다음부터는 더이상 frequent pattern 이 나오지 않을때까지 Length=k 인 frequent pattern 으로부터 Length=k+1 인 frequent pattern 을 generate 한다. 각 함수의 설명과 함께 설명을 보충하겠다.

우선 알고리즘의 흐름이 아닌 그 알고리즘 내에서 도와주는 함수들부터 설명하겠다. isSubset 함수는 parameter1 이 parameter2 의 부분집합에 포함되는지를 판단하는 함수이다. 부분집합에 포함되는지 확인하는 방법으로, 두 itemset 의 교집합으로 만들어진 벡터가 포함되는지 확인하고자 만드는 벡터와 동일하면 부분집합이 맞고, 다르다면 부분집합이 아니다.

```
bool isSubset(Item a, Item b) {
    // a의 사이즈가 항상 작게 만들어줌
    if(a.getSize() > b.getSize()) {
        swap(a, b);
    }
    vector<int> v(a.getSize() + b.getSize());
    vector<int>::iterator it;
    vector<int> v1 = a.getItemSet();
    vector<int> v2 = b.getItemSet();
    it = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v.begin());
    v.resize(it - v.begin());
    if(v == v1) return true;
    else return false;
}

bool isSubset(Item a, vector<int> v2) {
    vector<int> v(a.getSize() + v2.size());
    vector<int>::iterator it;
    vector<int> v1 = a.getItemSet();
    it = set_intersection(v1.begin(), v1.end(), v2.begin(), v2.end(), v.begin());
    v.resize(it - v.begin());
    if(v == v1) return true;
    else return false;
}
```

유사한 방법을 사용하는 unionItem 함수는 인자로 들어오는 두 파라미터를 합치는 역할을 한다. 만약 length =k 인 두 itemset 을 합쳐 length=k+1 인 itemset 을 만들지 못한다면 빈 아이템셋을 만들어 실패했음을 의미하고, 아니면 합쳐진 아이템셋을 리턴한다.

```
Item unionItem(Item a, Item b) {
    vector<int> v(a.getSize() + b.getSize());
    vector<int>::iterator it;
    vector<int> v1 = a.getItemSet();
    vector<int> v2 = b.getItemSet();
    it = set_union(v1.begin(), v1.end(), v2.begin(), v2.end(), v.begin());
    v.resize(it - v.begin());
    if(v.size() == v1.size() + 1) {
        sort(v.begin(), v.end());
        return Item(v, 0);
    } else {
        return Item();
    }
}
```

printVector 함수는 벡터를 과제의 명세에 맞게 출력변형해주는 함수이다.

```
void printVector(vector<int>& v, ostream& ofs) {
    ostringstream oss;
    if(!v.empty()) {
        oss << "{";
        copy(v.begin(), v.end()-1,
        ostream_iterator<int>(oss, ","));
        oss << v.back();
        oss << "}\t";
    }
    ofs.write(oss.str().c_str(), oss.str().size());
}
```

Search 는 재귀적으로 모든 부분집합을 만드는 함수이다.

```
void search(set<Item>& sI, vector<int>& total, vector<int>& subset, int k, ostream& ofs) {
    if(k == total.size()) {
        seperate(sI, total, subset, ofs);
        return;
    } else {
        subset.push_back(total[k]);
        search(sI, total, subset, k+1, ofs);
        subset.pop_back();
        search(sI, total, subset, k+1, ofs);
    }
}
```

Separate 는 부분집합을 만들고 나서 해당 부분집합을 출력하고 support 와 confidence 를 계산해 출력한다.

```
void seperate(set<Item>& sI, vector<int>& total, vector<int>& sub, ostream& ofs) {
    if(sub.empty() || total == sub) return;
    vector<int> v(total.size() + sub.size());
    vector<int>::iterator it;
    it = set_difference(total.begin(), total.end(), sub.begin(), sub.end(), v.begin());
    v.resize(it - v.begin());
    printVector(v, ofs);
    printVector(sub, ofs);
    char p1[10], p2[10];
    sprintf(p1, "%.2f\t",sI.find(total)->getSupport()*100);
    ofs.write(p1, strlen(p1));
    int u = sI.find(total)->getSupportCount();
    int d = sI.find(v)->getSupportCount();
    sprintf(p2, "%.2f\n",(double)u / d * 100);
    ofs.write(p2, strlen(p2));

}
```

이제 알고리즘에 대한 설명을 추가로 하자면 먼저 apriori 함수에서 firstScan 이라는 함수를 호출해서 length=1 인 모든 frequent pattern 을 만든다. 방법은 다음과 같이 구현하였다.

```
void firstScan(double min_support, vector<Item>& v, set<Item>& sI) {
    set<int> s;
    for(int i = 0; i < database.size(); ++i) {
        for(int j = 0; j < database[i].size(); ++j) {
            int f = database[i][j];
            vector<int> one_len_v = vector<int>(1, f);
            Item newItem(one_len_v);
            if(s.find(f) == s.end()) {
                s.insert(f);
                sI.insert(newItem);
            } else {
                auto it = sI.find(newItem);
                int now_sup = (*it).getSupportCount();
                sI.erase(newItem);
                sI.insert(Item(one_len_v, now_sup+1));
            }
        }
    }
    for(auto it = sI.begin(); it != sI.end(); ++it) {
        v.push_back(*it);
    }
}
```

그리고 Apriori 알고리즘을 구현하기 위해서 apriori 함수 내에 vector, set 과 multimap 을 활용하였는데, set 은 최종적인 모든 frequent item 을 모두 모와놓기 위해 사용하였고, multimap 은 itemset 을 합쳐서 길이가 1 증가한 itemset 을 만들때 그 새로 만들어진 itemset 보다 길이가 1 짧은 모든 부분집합이 존재해야 하는데, 그 집합을 카운트할때 사용한다. 끝까지 multimap 을 돌면서 모든 부분집합이 존재하는 Iteme 들을 벡터 L 에 보관한다.

```
void apriori(double min_support, ofstream& ofs) {
    int idx = 1;
    vector<Item> L;
    set<Item> sI;
    firstScan(min_support, L, sI);
    for(int k = 1; !L.empty(); ++k) {
        vector<Item> nextL;
        multimap<Item, int> mm;
        for(int m = 0; m < L.size(); ++m) {
            for(int n = m+1; n < L.size(); ++n) {
                Item newItem = unionItem(L[m],L[n]);
                vector<int > nv = newItem.getItemSet();
                if(newItem.getSize() == k+1) {
                    multimap<Item, int>::iterator it = mm.find(newItem);
                    if(it == mm.end()) {
                        mm.insert(make_pair(newItem, 1));
                    } else {
                        it->second += 1;
                    }
                } else {
                    continue;
                }
            }
        }
        for(auto it = mm.begin(); it != mm.end(); ++it) {
            if(it->second == (k*(k+1))/2) {
                nextL.push_back(it->first);
            }
        }
```

그 후 데이터베이스를 scan 하면서 pattern 이 얼마나 나왔는지 count 를 해준다. 여기서 addSupport 가 데이터베이스의 스캔카운트를 증가시키는 함수이다. 그리고 스캔이 완료되었을때 min_support 보다 커지면 set 에 진짜 frequent pattern 으로 추가를 하고, 위에 설명한 search 함수로 모든 부분집합을 구해 답을 출력한다.

```
for(int dbIdx = 0; dbIdx < database.size(); ++dbIdx) {
    for(auto it = nextL.begin(); it != nextL.end(); ++it) {
        Item* comp = &(*it);
        if(isSubset(*comp, database[dbIdx])) {
            comp->addSupport();
        }
    }
}
for(auto it = nextL.begin(); it != nextL.end();) {
    if(it->getSupport() >= min_support) {
        sI.insert(*it);
        it++;
    } else {
        it = nextL.erase(it);
    }
}
for(auto it = nextL.begin(); it != nextL.end(); ++it) {
    Item item = *it;
    vector<int> total = item.getItemSet();
    vector<int> subset;
    search(sI, total, subset, 0, ofs);
}
L = nextL;
```

이러한 루프를 반복해 L 이 empty 가 될때까지 반복하면 모든 frequent pattern 을 구할 수 있다.

컴파일 환경은 다음과 같다.

macOS Big Sur 11.2.3

gcc –version:

```
Apple clang version 12.0.0 (clang-1200.0.32.29)
Target: x86_64-apple-darwin20.3.0
Thread model: posix
```

컴파일 방법으로는 Makefile 을 만들었으나 만약 실패한다면 g++ -o assignment assignment1.cpp -std=c++11 로 컴파일하면 된다.

1)

```
└─ make
c++ -std=c++11   -c -o assignment1.o assignment1.cpp
g++ -o assignment assignment1.o
```

2)

```
└─ g++ -o assignment1 assignment1.cpp -std=c++11
```