Apriori algorithm report

- Author: 김영진 2016025241
- Tested build environment: OSX, Ubuntu18

Summary of algorithm

`std::vector<AprioriSolverResultLine> AprioriSolver::Solve(int)` 의 코드에서,

```
std::vector<AprioriSolverResultLine> AprioriSolver::Solve(int
min_support) const {
    ...

    ItemSetList set_list = GetSingleItemSetList();

    ...
}
```

1. 우선 모든 트랜젝션에서 발견된 ID들을 모두 모아서 길이가 1인 ItemSet들로 이루어진 List를 생성합니다.

```
    std::vector<int> support_list = GetSupports(set_list);
    std::vector<ItemSet> new_sets;
    auto it = set_list.begin();
    for (int i = 0; i < support_list.size(); i++, it++) {
        if (support_list[i] > min_support) {
            new_sets.push_back(*it);
            frequent_sets.push_back(*it);
            frequent_supports.push_back(support_list[i]);
            support_map[it->ToString()] = support_list[i];
        }
    }
```

2. 그리고 support value가 일정값 이상인 item set들만 살아남깁니다. 살아남은 모든 item set들은 frequent set으로 등록되며 support value가 cache 됩니다.

```
    set_list = ItemSetList(new_sets).SelfJoin();
```

3. 살아남은 item set들을 self-join해서 새로운 item set들을 생성해냅니다.

위의 2,3번 과정으로 더 이상 살아남은 item set이 존재하지 않을 때 까지 반복합니다.

```
    std::vector<AprioriSolverResultLine> results;
    for (int i = 0; i < frequent_sets.size(); i++) {
        const auto rules = ExtractAssociationRules(frequent_sets[i],
    frequent_supports[i], support_map);
        std::copy(rules.begin(), rules.end(), std::back_inserter(results));
    }
    return results;
```

위의 과정들을 통해 발견한 frequent set들을 통해 association rule을 생성합니다. 같은 frequent set에서 파생된 rule들은 자명하게 같은 support를 가지며 item_set + associative_item_set이 같지만, confidence는 다를 수 있습니다.

Detailed description

Convert minimum support

```
min_support = (int) (((double) records.size() * min_support) / 100.0);
```

유저로부터 받은 minimum support는 상대값이지만, transaction 개수 정보를 반영하여 절대값으로 변형합니다. 앞으로 frequent set은 이 변형된 `min_support` 값 보다 많이 등장해야 합니다.

Getting supports

```
int AprioriSolver::LookupSupport(const ItemSet & set) const {
    int count = 0;
    for (const auto & rec: records) {
        if (set.Included(rec)) ++count;
    }
    return count;
}

bool ItemSet::Included(const std::vector<int> & v) const {
    int count = 0;
    for (int id: v) {
        if (s.find(id) != s.end()) {
            ++count;
        }
    }
    return s.size() == count;
}
```

support value를 확인하기 위해, Naive 한 방법으로 각 transaction 마다 해당 item set 을 포함하고 있는지 확인합니다. 각 확인 절차는 transaction의 길이가 N일 때, O(NlgN) 시간을 소요합니다.

Self-Join

```
ItemSetList ItemSetList::SelfJoin() const {
    const int n = sets.size();
    std::vector<ItemSet> new_sets;
    for (auto i = sets.begin(); i != sets.end(); i++) {
        const int tg_size = i->Size() + 1;
        auto j = i;
        j++;
        for (; j != sets.end(); j++) {
            if (!(i->Similar(*j))) continue;
            new_sets.push_back((*i) + (*j));
        }
    }
    return new_sets;
}

bool ItemSet::Similar(const ItemSet &tg) const {
    if (Size() != tg.Size()) return false;
    const int n = Size();
    auto it1 = begin(), it2 = tg.begin();
    for (int i = 1; i < n; i++) {
        if ((*it1) != (*it2)) return false;
        ++it1;
        ++it2;
    }
    return (*it1) != (*it2);
}
```

item set list의 각 원소쌍에 대해서, 두 item set이 '비슷'하면 합쳐서 다음 회차에 사용합니다.

여기서 '비슷'하다는, 길이 n의 두 ordered set에 대해서, 1~n-1번째 원소까지는 같지만 n번째 마지막 원소만 다른 상태를 의미합니다.

두 item set을 합친 n+1짜리 item set이 frequent 하다면, 위의 과정을 통해 단 한번, [i_1, i_2, ..., i_n, i{n+1}] 이 있을 때, [i_1, i_2, ..., i_n]과 [i_1, i_2, ..., i{n-1}, i_{n+1}] 이 합쳐져 등장하게 됩니다.

Getting association rules from frequent set

우리가 최종적으로 원하는 것은 {item_set, associative_item_set, support, confidence}로 이루어진 record list이므로, 각 frequent set, s가 있을 때, 합집합이 s와 같고 서로 독립인 두 개의 집합으로 나누는 모든 경우를 알아내야 합니다.

```
    for (int a_size = 1; a_size < ids.size(); a_size++) {
        ...
        std::vector<bool> selector(ids.size(), false);
        for (int i = 0; i < a_size; i++) selector[ids.size() - (i+1)] =
    true;
        do {
            ItemSet a, b;
            for (int i = 0; i < ids.size(); i++) {
                if (selector[i]) a.Add(ids[i]);
                else b.Add(ids[i]);
            }
            ...
        } while (std::next_permutation(selector.begin(), selector.end()));
    }
```

이 두 집합 중 하나의 길이를 1~N으로 고정했을 때, `std::next_permutation` 를 이용해서 편하게 모든 경우의 수를 순회할 수 있습니다.

```
    const float confidence = ((float) support) / ((float) a_support);
```

두 집합 A, B가 결정됐을 때, 미리 cache 해둔 support_map의 정보를 활용해서 confidence를 알아낼 수 있습니다.

Instructions for compiling

make를 이용해 빌드할 수 있습니다.

```
CC := g++
CFLAGS := -std=c++17 -O2 -I src/headers/

build:
  $(CC) $(CFLAGS) -c -o src/apriori.o src/apriori.cc
  $(CC) $(CFLAGS) -c -o src/main.o src/main.cc
  $(CC) $(CFLAGS) -c -o src/utils.o src/utils.cc
  $(CC) $(CFLAGS) -c -o src/itemset.o src/itemset.cc
  $(CC) $(CFLAGS) -c -o src/itemsetlist.o src/itemsetlist.cc
  $(CC) -o apriori src/apriori.o src/main.o src/utils.o src/itemset.o
src/itemsetlist.o

clean:
  rm -rf src/*.o

all: build
```

출력되는 바이너리 이름은 apriori 입니다. 현재 레포에서

```
./apriori 5 data/input.txt output.txt
```

로 테스트 할 수 있습니다.