Implementation of Apriori Algorithm

2017029807 성창호

Index

1. Introduction

This program is implemented to find association rules using the Apriori algorithm. Receive transactions(Transaction DB, TDB) consisting of $item\_id$ and minimum support value (Sup_{min}) as input, and output association rules with support greater than Sup_{min}. This is implemented using the Apriori algorithm and the appropriate data sturctures are used in the implementation process to increase performance.

2. Algorithm

The Apriori algorithm can be divided into the two process, generated candidates and calculate support from candidates. These process need to scan the input(TDB) as a whole to achieve each goal. Scanning the TDB has a high time complexity. So we should try to lower the time complexity of this process.

First, sort the items in the transactions by size. This eliminates unnecessary operations while performing the apriori algorithm. And find an item set with a length of 1 in the TDB. This is the same as finding all kinds of items in the TDB. The item sets that are found are the candidates with a length of 1, C_1.

The support of itemset in C_1 is calculated by scanning the TDB. The support Sup is calculated as follow:

$$Sup(X) = \frac{n(X)}{N_{transaction}}$$

Based on this, candidates with support smalller than the Sup_{min} are eliminated. The remaining candidates are referred to as frequent 1-itemset, L_1. We save thre frequent itemsets and their support in the Red-Black tree as pair for later when prunning or generating association rules. This ca reduce the time to find a specific frequent-itemset from $O(N_k)$ to $O(\lg N_k)$ when the number of frequent itemset with length of k is called N_k.

To generate C_2, we have to self-join L_1. If the length of frequent itemset is more than 2, then verify that thee candidate of the self-join result is satisfied the Apriori prunning principle. The subset of the itemset must be frequent in order to satisfy the Apriori prunning principle. This can be proved inductive, so to verify that the itemset with length of k is frequent, check if the subset with length of $k-1$ is present in frequent itemset L_{k-1}. With C_2, self-join in the same way as before, calculate support to generate L_2. These processes of this program done in `make_Candidate` and `get_Supports` functions in `Apriori` class.

Repeat these processes until there is no candidate or frequent itemset. And save the frequent itemsets, L_k and thier support as pair in the Red-Black Tree.

When all of the frequent itemsets are generated, we generate make association rules. The number of assocation rules that can be made with a frequent itemet with a length of k is $2^k - 2$. There are 2^k rules for each item to be located on the left side and right side, but if all items are on one side, subtract 2 rules. This process is recursively implemented and implemented in `make_Association` function in `Apriori` class.

3. Usage

```
$ make
$ ./apriori.exe 10 input/input_file.txt output/output_file.txt
```

Makefile

```
TARGET = apriori.exe dataGenerator.exe

all: $(TARGET)

apriori.exe : apriori.o
    g++ -std=c++11 -o apriori.exe apriori.o

dataGenerator.exe : dataGenerator.o
    g++ -std=c++11 -o dataGenerator.exe dataGenerator.o

apriori.o : apriori.cpp
    g++ -std=c++11 -c -o apriori.o apriori.cpp

dataGenerator.o : dataGenerator.cpp
    g++ -std=c++11 -c -o dataGenerator.o dataGenerator.cpp

clean:
    rm $(TARGET) apriori.o dataGenerator.o
```

4. Implementation

We implemented this program by divding it into `Input`, `Output`, and `Apriori` class.

`main` fucntion

```
typedef vector<int> VI;
typedef vector<double> VD;
typedef vector<vector<int>> VV;
typedef vector<tuple<VI, VI, double, double>> VT;

int main(int argc, char* argv[]) {
    clock_t start = clock(), end;

    if (argc != 4) {
        printf("E: You should input 3 parameters! => min_sup(%%), input_file_name, output_file_name\n");
        return 0;
    }

    double min_sup = stod(string(argv[1]));
    string input_file(argv[2]), output_file(argv[3]);

    Input input(input_file);
    Apriori apriori(input.make_TDB(), min_sup);
    Output output(output_file);

    output.print(apriori.process());

    end = clock();
    printf("Total Running Time : %.3lfsec\n", (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

In the `main` function, the arguments inputed as `argv` are formatted and created objects for each class. The execution time is also measured to verify the performance of this program.

`Input` class

```
class Input {
private:
    FILE * fo;
    VV TDB;

public:
    Input(string input_file) {
        fo = fopen(input_file.c_str(), "r");
        if (fo == NULL) {
            printf("E: Cannot open input file!\n");
            exit(0);
        }
    }

    VV make_TDB() {
        VI T;
        int num;
        char space;
        bool flag = true;

        while (fscanf(fo, "%d%c", &num, &space) != EOF) {
            T.push_back(num);
            if (space == '\n') {
                sort(T.begin(), T.end());
                TDB.push_back(T);
                T.clear();
                flag = true;
            }
            else
                flag = false;
        }
        if (!flag) {
            sort(T.begin(), T.end());
            TDB.push_back(T);
        }

        return TDB;
    }
};
```

In `Input` class, input transactions from files received by argument and create a TDB based on them. The constructor defines the file pointer and creates a TDB in the `make_TDB` function. Unlike input, the TDB sorts the items in each transaction in ascending order. The variable `TDB` that stores the TDB is declared `vector<vector<int>>` is equal `VV`.

`Apriori` class

Member Variables

```
class Apriori {
private:
    VV TDB, C, L;
    vector<map<VI, double>> Freq;
    double min_sup;
    int step;
    VT AR;
};
```

`TDB` is a variable that stores a TDB generated through `make_TDB` function in `Input` class. `C` and `L` are the candidates and frequent itemsets for the length of `step`. `Freq` is a C++ STL `map` variable that store L_k that is frequent itemest for the legnth of k, in the Red-Black Tree. (`map` is implemented as a Red-Black Tree.) `min_sup` is the minimum support received by input, and `step` is the length of the candidate in the current state. `AR` stores generated assocation rules.

Constructor

```
Apriori(VV _TDB, double _min_sup) {
    TDB = _TDB;
    min_sup = _min_sup;
    step = 0;

    // make 1-candidate
    set <int> C1;
    for (auto& T : TDB) {
        for (auto& item : T)
            C1.insert(item);
    }
    for (auto& item1 : C1)
        C.push_back({ item1 });
    Freq.push_back({});
}
```

In Constructor, we generate candidate C_1 for the length of 1 using TDB. C_1 is as same as the kinds of items in the TDB, we generate these using C++ STL `set`.

`process` fucntion

```
VT process() {
    while (1) {
        step++;

        if (step > 1)
            C = make_Candidate();

        if (C.size() == 0)
            break;

        VD C_sup = get_Supports();
        L.clear();

        map<VI, double> now_Freq;
        for (int i = 0; i < C_sup.size(); i++) {
            if (C_sup[i] < min_sup)
                continue;

            L.push_back(C[i]);
            now_Freq[C[i]] = C_sup[i];
        }
        if (L.size() == 0)
            break;

        Freq.push_back(now_Freq);
    }

    for (auto& now_items : Freq) {
        for (auto& item : now_items) {
            make_Association(item.first, item.second, {}, {}, 0);
        }
    }

    return AR;
}
```

In the `process` function, the candidates for current step are generated by `make_Candidate` fucntion and their support is calculated by `get_Supports`. At this point, the `process` function does not generate candidates for the length of 1, because it was created by the constructor when `step` is 1.

Candidates with support greater than or equal to the minimum support `min_sup` in `C_sup` calculated by `get_supports` fucntion append to `L`. And these and their support store as pairs in the `map` variable `now_Freq`. `now_Freq` is append to vector `Freq`.

Finally, calculate the association rules with `Freq` that all of the frequent itemset are stored, as the argument of `make_Associatoin`.

`make_Candidate` function

```
VV make_Candidate() {
    set<VI> check;
    VV candidate;

    // Self-Join
    for (int i = 0; i < L.size(); i++) {
        for (int j = i + 1; j < L.size(); j++) {
            auto item = merge(L[i], L[j]);
            if (item.size() != step)
                continue;
            if (check.find(item) == check.end()) {
                check.insert(item);
                // Pruning
                int k;
                for (k = 0; k < item.size(); k++) {
                    auto tmp = item;
                    tmp.erase(tmp.begin() + k);
                    if (Freq[step - 1].find(tmp) == Freq[step - 1].end())
```

```
                        break;
                    }

                    if (k == item.size())
                        candidate.push_back(item);
                }
            }
        }

        return candidate;
    }
```

In `make_Candidate` funtion, we generate candidate for the length of `step`. We self-join the frequent itemset `L` for the length of `step` -1. We use nested loops for doing self-join, and merge each itemset `L[i]`, `L[j]`. If the length of merged itemset is lower than `step`, it can not be frequent itemset for the length of `step`.

For a merged itemset `item` to be frequent itemset, it must satisfy the Apriori prunning principle. Therefore, subset itemset of `item` with length of `step` -1 is checked through that it exist in `Freq[step - 1]`. Because we use `map`, it is possible to reduce the time to find subset of `item` from $O(N)$ to $O(\lg N)$.

`merge` function

```
VI merge(VI A, VI B) {
    VI C;

    for (int i = 0; i < A.size(); i++) {
        if (i < A.size() - 1) {
            if (A[i] == B[i])
                C.push_back(A[i]);
            else
                // If we merge these, it will be overlaped
                return {};
        }
        else {
            C.push_back((A[i] < B[i]) ? A[i] : B[i]);
            C.push_back((A[i] < B[i]) ? B[i] : A[i]);
        }
    }

    return C;
}
```

In `merge` function, we merge two itemsets for the lengh of `step` with no duplicate. Since all itemset are sorted, in order to increase the length of merged itemsets by 1, only the last element must be different. If other elements are different, they can not be merged.

`get_Supports` function

```
VD get_Supports() {
    VD supports;

    for (auto& item : C) {
        int support = 0;
        for (auto& T : TDB) {
            if (T.size() < item.size()) continue;

            int j = 0;
            for (int i = 0; i < T.size() && j < item.size(); i++) {
                if (T[i] == item[j])
                    j++;
            }

            if (j == item.size())
                support++;
        }
        supports.push_back((double)support / TDB.size() * 100);
    }

    return supports;
}
```

In `get_Supports` function, we calculate the support of each candidates. The return value `supports` is changed to the % (percent) format.

`make_Association` function

```
void make_Association(VI item, double AB_sup, VI A, VI B, int idx) {
    if (idx == item.size()) {
        if (A.size() == 0 || B.size() == 0)
            return;
        double A_sup = Freq[A.size()][A];

        AR.push_back(make_tuple(A, B, AB_sup, AB_sup / A_sup * 100));
        return;
    }

    A.push_back(item[idx]);
    make_Association(item, AB_sup, A, B, idx + 1);
    A.pop_back();
    B.push_back(item[idx]);
    make_Association(item, AB_sup, A, B, idx + 1);
}
```

As explained in **Algorithm**, the assocation rule is recursively generated. The generated assocation rule stores each set of items, support, and confidence in `AR` as a tuple. The support and confidence are calculated as follow:

$$Sup(X \to Y) = \frac{n(X \cup Y)}{N_{transaction}}$$
$$Conf(X \to Y) = \frac{n(X \cup Y)}{n(X)} = \frac{Sup(X \to Y)}{Sup(X)}$$

`Output` class

```
class Output {
private:
    FILE * fp;

public:
    Output(string output_file) {
        fp = fopen(output_file.c_str(), "w");
        if (fp == NULL) {
            printf("E: Cannot make output file!\n");
            exit(0);
        }
    }

    void print(VT ARs) {
        for (auto AR : ARs) {
            auto A = get<0>(AR), B = get<1>(AR);
            double sup = get<2>(AR), conf = get<3>(AR);

            print_vector(A);
            print_vector(B);
            fprintf(fp, "%.2lf\t%.2lf\n", sup, conf);
        }
    }

    void print_vector(VI A) {
        fprintf(fp, "{");
        for (int i = 0; i < A.size(); i++) {
            if (i == A.size() - 1)
                fprintf(fp, "%d", A[i]);
            else
                fprintf(fp, "%d,", A[i]);
        }
        fprintf(fp, "}\t");
    }
};
```

In `Output` class, define a file pointer to save as the `output_file` received by the constructor. And print the result according to format.

5. Experiments

Running Environment

- OS : Ubuntu 16.04.6 (WSL)
- CPU : Intel(R) Core(TM) i7-7660U CPU 250Hz
- RAM : 16GB
- Language : G++ 5.4.0

Results

N_i and N_t represent the number of itemset and transactions. The result below are the time to execute data made from N_i and N_t. The unit is $\mathbf{sec}$.

	$(N_i, N_t) = (10, 300)$	$(N_i, N_t) = (10, 1000)$	$(N_i, N_t) = (10, 10^5)$	$(N_i, N_t) = (20, 500)$ **(example)**	$(N_i, N_t) = (20, 1000)$	$(N_i, N_t) = (30, 1000)$	$(N_i, N_t) = (50, 1000)$
$Sup_{min} = 4$	0.062	0.078	3.266	0.125	5.484	-	-
$Sup_{min} = 5$	0.031	0.047	3.172	0.094	1.672	13.625	-
$Sup_{min} = 9$	0.031	0.031	1.641	0.031	0.391	1.562	10.906
$Sup_{min} = 10$	0.016	0.016	1.609	0.031	0.203	0.828	7.719

Analysis

Results show that as the Sup_{min} increases, and as the N_i and N_t decreases, the execution time decreases. An increase in Sup_{min} results in a decrease in the number of candidates, and so does the execution time. And as N_i and N_t increase, the time to scan the TDB increases, so does the execution time. However, the increase in N_i has had a greater impact on the increase in execution time than the increase in N_t. This is expected to be largely influenced by self-join which generates candidates.