**FINDING FREQUENT ITEMSET AND GENERATING STRONG ASSOCIATION RULES**

NAME:BINJU THOMAS

ROLL NO: 175214149

PROBLEM DESCRIPTION:

**Apriori** is an algorithm for frequent item set mining and [association rule learning](https://en.wikipedia.org/wiki/Association_rule_learning) over transactional [databases](https://en.wikipedia.org/wiki/Databases). It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets as long as those item sets appear sufficiently often in the database. The frequent item sets determined by Apriori can be used to determine [association rules](https://en.wikipedia.org/wiki/Association_rules) which highlight general trends in the [database](https://en.wikipedia.org/wiki/Database): this has applications in domains such as [market basket analysis](https://en.wikipedia.org/wiki/Market_basket_analysis).

**Code:**

AprioriFrequentItemsetGenerator.java:

package net.coderodde.mining.arg;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Objects;

import java.util.Set;

public class AprioriFrequentItemsetGenerator<I>

{

public FrequentItemsetData<I> generate(List<Set<I>> transactionList,

double minimumSupport) {

Objects.requireNonNull(transactionList, "The itemset list is empty.");

checkSupport(minimumSupport);

if (transactionList.isEmpty()) {

return null;

}

Map<Set<I>, Integer> supportCountMap = new HashMap<>();

List<Set<I>> frequentItemList = findFrequentItems(transactionList, supportCountMap, minimumSupport);

Map<Integer, List<Set<I>>> map = new HashMap<>();

map.put(1, frequentItemList);

int k = 1;

do {

++k;

List<Set<I>> candidateList = generateCandidates(map.get(k - 1));

for (Set<I> transaction : transactionList) {

List<Set<I>> candidateList2 = subset(candidateList,

transaction);

for (Set<I> itemset : candidateList2) {

supportCountMap.put(itemset,

supportCountMap.getOrDefault(itemset,

0) + 1);

}

}

map.put(k, getNextItemsets(candidateList,

supportCountMap,

minimumSupport,

transactionList.size()));

} while (!map.get(k).isEmpty());

return new FrequentItemsetData<>(extractFrequentItemsets(map),

supportCountMap,

minimumSupport,

transactionList.size());

}

private List<Set<I>>

extractFrequentItemsets(Map<Integer, List<Set<I>>> map) {

List<Set<I>> ret = new ArrayList<>();

for (List<Set<I>> itemsetList : map.values()) {

ret.addAll(itemsetList);

}

return ret;

}

private List<Set<I>> getNextItemsets(List<Set<I>> candidateList,

Map<Set<I>, Integer> supportCountMap,

double minimumSupport,

int transactions) {

List<Set<I>> ret = new ArrayList<>(candidateList.size());

for (Set<I> itemset : candidateList) {

if (supportCountMap.containsKey(itemset)) {

int supportCount = supportCountMap.get(itemset);

double support = 1.0 \* supportCount / transactions;

if (support >= minimumSupport) {

ret.add(itemset);

}

}

}

return ret;

}

private List<Set<I>> subset(List<Set<I>> candidateList,

Set<I> transaction) {

List<Set<I>> ret = new ArrayList<>(candidateList.size());

for (Set<I> candidate : candidateList) {

if (transaction.containsAll(candidate)) {

ret.add(candidate);

}

}

return ret;

}

private List<Set<I>> generateCandidates(List<Set<I>> itemsetList) {

List<List<I>> list = new ArrayList<>(itemsetList.size());

for (Set<I> itemset : itemsetList) {

List<I> l = new ArrayList<>(itemset);

Collections.<I>sort(l, ITEM\_COMPARATOR);

list.add(l);

}

int listSize = list.size();

List<Set<I>> ret = new ArrayList<>(listSize);

for (int i = 0; i < listSize; ++i) {

for (int j = i + 1; j < listSize; ++j) {

Set<I> candidate = tryMergeItemsets(list.get(i), list.get(j));

if (candidate != null) {

ret.add(candidate);

}

}

}

return ret;

}

private Set<I> tryMergeItemsets(List<I> itemset1, List<I> itemset2) {

int length = itemset1.size();

for (int i = 0; i < length - 1; ++i) {

if (!itemset1.get(i).equals(itemset2.get(i))) {

return null;

}

}

if (itemset1.get(length - 1).equals(itemset2.get(length - 1))) {

return null;

}

Set<I> ret = new HashSet<>(length + 1);

for (int i = 0; i < length - 1; ++i) {

ret.add(itemset1.get(i));

}

ret.add(itemset1.get(length - 1));

ret.add(itemset2.get(length - 1));

return ret;

}

private static final Comparator ITEM\_COMPARATOR = new Comparator() {

@Override

public int compare(Object o1, Object o2) {

return ((Comparable) o1).compareTo(o2);

}

};

private List<Set<I>> findFrequentItems(List<Set<I>> itemsetList,

Map<Set<I>, Integer> supportCountMap,

double minimumSupport) {

Map<I, Integer> map = new HashMap<>();

// Count the support counts of each item.

for (Set<I> itemset : itemsetList) {

for (I item : itemset) {

Set<I> tmp = new HashSet<>(1);

tmp.add(item);

if (supportCountMap.containsKey(tmp)) {

supportCountMap.put(tmp, supportCountMap.get(tmp) + 1);

} else {

supportCountMap.put(tmp, 1);

}

map.put(item, map.getOrDefault(item, 0) + 1);

}

}

List<Set<I>> frequentItemsetList = new ArrayList<>();

for (Map.Entry<I, Integer> entry : map.entrySet()) {

if (1.0 \* entry.getValue() / map.size() >= minimumSupport) {

Set<I> itemset = new HashSet<>(1);

itemset.add(entry.getKey());

frequentItemsetList.add(itemset);

}

}

return frequentItemsetList;

}

private void checkSupport(double support) {

if (Double.isNaN(support)) {

throw new IllegalArgumentException("The input support is NaN.");

}

if (support > 1.0) {

throw new IllegalArgumentException(

"The input support is too large: " + support + ", " +

"should be at most 1.0");

}

if (support < 0.0) {

throw new IllegalArgumentException( "The input support is too small: " + support + ", " +

"should be at least 0.0");

}

}

}

**FrequentItemsetData.java:**

import java.util.List;

import java.util.Map;

import java.util.Set;

public class FrequentItemsetData<I> {

private final List<Set<I>> frequentItemsetList;

private final Map<Set<I>, Integer> supportCountMap;

private final double minimumSupport;

private final int numberOfTransactions;

FrequentItemsetData(List<Set<I>> frequentItemsetList,

Map<Set<I>, Integer> supportCountMap,

double minimumSupport,

int transactionNumber) {

this.frequentItemsetList = frequentItemsetList;

this.supportCountMap = supportCountMap;

this.minimumSupport = minimumSupport;

this.numberOfTransactions = transactionNumber;

}

public List<Set<I>> getFrequentItemsetList() {

return frequentItemsetList;

}

public Map<Set<I>, Integer> getSupportCountMap() {

return supportCountMap;

}

public double getMinimumSupport() {

return minimumSupport;

}

public int getTransactionNumber() {

return numberOfTransactions;

}

public double getSupport(Set<I> itemset) {

return 1.0 \* supportCountMap.get(itemset) / numberOfTransactions;

}

}

Demo.java

package net.coderodde.mining.arg;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class Demo {

public static void main(String[] args) {

AprioriFrequentItemsetGenerator<String> generator =

new AprioriFrequentItemsetGenerator<>();

List<Set<String>> itemsetList = new ArrayList<>();

itemsetList.add(new HashSet<>(Arrays.asList("a", "b")));

itemsetList.add(new HashSet<>(Arrays.asList("b", "c", "d")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "c", "d", "e")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "d", "e")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c", "d")));

itemsetList.add(new HashSet<>(Arrays.asList("a")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "c")));

itemsetList.add(new HashSet<>(Arrays.asList("a", "b", "d")));

itemsetList.add(new HashSet<>(Arrays.asList("b", "c", "e")));

FrequentItemsetData<String> data = generator.generate(itemsetList, 0.2);

int i = 1;

for (Set<String> itemset : data.getFrequentItemsetList()) {

System.out.printf("%2d: %9s, support: %1.1f\n",

i++,

itemset,

data.getSupport(itemset));

}

}

}

PROJECT SUMMARY:

On finding frequent itemsets and generating strong association rules to develop an

implementation using java for Apriori algorithm to find all frequent itemsets. Also

we,generate all Strong Association Rules from the frequent itemsets.