|  |  |
| --- | --- |
| **Course Code: CSL603** | **Course Name: DWM LAB** |
| **Class: TE-CO** | **Batch: 3** |
| **Roll no: 18CO63** | **Name: SHAIKH TAUSEEF MUSHTAQUE ALI** |

**Experiment :08**

**Aim:** Implementation of Apriori algorithm

**Code:**

|  |
| --- |
| import sys |
|  |
| from itertools import chain, combinations |
| from collections import defaultdict |
| from optparse import OptionParser |
|  |
|  |
| def subsets(arr): |
| """ Returns non empty subsets of arr""" |
| return chain(\*[combinations(arr, i + 1) for i, a in enumerate(arr)]) |
|  |
|  |
| def returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet): |
| """calculates the support for items in the itemSet and returns a subset |
| of the itemSet each of whose elements satisfies the minimum support""" |
| \_itemSet = set() |
| localSet = defaultdict(int) |
|  |
| for item in itemSet: |
| for transaction in transactionList: |
| if item.issubset(transaction): |
| freqSet[item] += 1 |
| localSet[item] += 1 |
|  |
| for item, count in localSet.items(): |
| support = float(count) / len(transactionList) |
|  |
| if support >= minSupport: |
| \_itemSet.add(item) |
|  |
| return \_itemSet |
|  |
|  |
| def joinSet(itemSet, length): |
| """Join a set with itself and returns the n-element itemsets""" |
| return set( |
| [i.union(j) for i in itemSet for j in itemSet if len(i.union(j)) == length] |
| ) |
|  |
|  |
| def getItemSetTransactionList(data\_iterator): |
| transactionList = list() |
| itemSet = set() |
| for record in data\_iterator: |
| transaction = frozenset(record) |
| transactionList.append(transaction) |
| for item in transaction: |
| itemSet.add(frozenset([item])) # Generate 1-itemSets |
| return itemSet, transactionList |
|  |
|  |
| def runApriori(data\_iter, minSupport, minConfidence): |
| """ |
| run the apriori algorithm. data\_iter is a record iterator |
| Return both: |
| - items (tuple, support) |
| - rules ((pretuple, posttuple), confidence) |
| """ |
| itemSet, transactionList = getItemSetTransactionList(data\_iter) |
|  |
| freqSet = defaultdict(int) |
| largeSet = dict() |
| # Global dictionary which stores (key=n-itemSets,value=support) |
| # which satisfy minSupport |
|  |
| assocRules = dict() |
| # Dictionary which stores Association Rules |
|  |
| oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet) |
|  |
| currentLSet = oneCSet |
| k = 2 |
| while currentLSet != set([]): |
| largeSet[k - 1] = currentLSet |
| currentLSet = joinSet(currentLSet, k) |
| currentCSet = returnItemsWithMinSupport( |
| currentLSet, transactionList, minSupport, freqSet |
| ) |
| currentLSet = currentCSet |
| k = k + 1 |
|  |
| def getSupport(item): |
| """local function which Returns the support of an item""" |
| return float(freqSet[item]) / len(transactionList) |
|  |
| toRetItems = [] |
| for key, value in largeSet.items(): |
| toRetItems.extend([(tuple(item), getSupport(item)) for item in value]) |
|  |
| toRetRules = [] |
| for key, value in list(largeSet.items())[1:]: |
| for item in value: |
| \_subsets = map(frozenset, [x for x in subsets(item)]) |
| for element in \_subsets: |
| remain = item.difference(element) |
| if len(remain) > 0: |
| confidence = getSupport(item) / getSupport(element) |
| if confidence >= minConfidence: |
| toRetRules.append(((tuple(element), tuple(remain)), confidence)) |
| return toRetItems, toRetRules |
|  |
|  |
| def printResults(items, rules): |
| """prints the generated itemsets sorted by support and the confidence rules sorted by confidence""" |
| for item, support in sorted(items, key=lambda x: x[1]): |
| print("item: %s , %.3f" % (str(item), support)) |
| print("\n------------------------ RULES:") |
| for rule, confidence in sorted(rules, key=lambda x: x[1]): |
| pre, post = rule |
| print("Rule: %s ==> %s , %.3f" % (str(pre), str(post), confidence)) |
|  |
|  |
| def to\_str\_results(items, rules): |
| """prints the generated itemsets sorted by support and the confidence rules sorted by confidence""" |
| i, r = [], [] |
| for item, support in sorted(items, key=lambda x: x[1]): |
| x = "item: %s , %.3f" % (str(item), support) |
| i.append(x) |
|  |
| for rule, confidence in sorted(rules, key=lambda x: x[1]): |
| pre, post = rule |
| x = "Rule: %s ==> %s , %.3f" % (str(pre), str(post), confidence) |
| r.append(x) |
|  |
| return i, r |
|  |
|  |
| def dataFromFile(fname): |
| """Function which reads from the file and yields a generator""" |
| with open(fname, "rU") as file\_iter: |
| for line in file\_iter: |
| line = line.strip().rstrip(",") # Remove trailing comma |
| record = frozenset(line.split(",")) |
| yield record |
|  |
|  |
| if \_\_name\_\_ == "\_\_main\_\_": |
|  |
| optparser = OptionParser() |
| optparser.add\_option( |
| "-f", "--inputFile", dest="input", help="filename containing csv", default=None |
| ) |
| optparser.add\_option( |
| "-s", |
| "--minSupport", |
| dest="minS", |
| help="minimum support value", |
| default=0.15, |
| type="float", |
| ) |
| optparser.add\_option( |
| "-c", |
| "--minConfidence", |
| dest="minC", |
| help="minimum confidence value", |
| default=0.6, |
| type="float", |
| ) |
|  |
| (options, args) = optparser.parse\_args() |
|  |
| inFile = None |
| if options.input is None: |
| inFile = sys.stdin |
| elif options.input is not None: |
| inFile = dataFromFile(options.input) |
| else: |
| print("No dataset filename specified, system with exit\n") |
| sys.exit("System will exit") |
|  |
| minSupport = options.minS |
| minConfidence = options.minC |
|  |
| items, rules = runApriori(inFile, minSupport, minConfidence) |
|  |
| printResults(items, rules) |