Apriori Algorithm

December 10, 2018

0.1 Emanuel Becerra Soto

0.2 MDD 01 CIC IPN

1 Introduction

This document contains the Apriori algorithm for generating frequent patterns and strong association rules.

The apriori algorithm uses the apriori property of frequent itemsets that states the following: **All nonempty subsets of a frequent itemset must be also be frequent**.

The apriori algorithm basically can be dived in two parts: 1) generating a set of candidate itemsets and then 2) testing for them if they're frequent or not.

The generating of candidates can be also divided into two parts: 1) The join step and the 2) prune step.

The data structure to represent the transaction database was a list of list, where each element of it represents a transaction.

This code takes advantage of the easiness that the set objects and methods that python provides by default.

As an example database the toy data of the page 250 of the book Data Mining: Concepts and Techniques 3er editions was used and also a function to generate random data for testing was also created.

2 Libraries

```
In [1]: import numpy as np
        import itertools
```

3 Functions

The **generate_random_transaction_db** function was created to generate random data for testing purposes, uses tree distributions to generate random integers numbers, then for each transaction it chooses (randomly) what distribution to use for that particular transaction.

```
In [2]: def generate_random_transaction_db( transactions, avg_items_per_transaction,
                                            max_diff ):
            mean = avg_items_per_transaction
            D = []
```

```
    for entry in range(transactions):
        choice = np.random.choice(['normal','integers','gamma'])
        items = abs(int(np.random.normal(loc=mean,scale=3)))
        if items == 0:
            items = 1
        if choice == 'normal':
            x = np.random.normal(loc=1,scale=10,size=items)
            x = [ abs(int(i)) for i in x ]
            while max(x) > max_diff:
                x = np.random.normal(loc=1,scale=10,size=items)
                x = [ abs(int(i)) for i in x ]
        elif choice == 'integers':
            x = list(np.random.randint(1,max_diff,size=items))
        else:
            x = np.random.gamma(shape=0.7,scale=20,size=items)
            x = [ abs(int(i)) for i in x ]
            while max(x) > max_diff:
                x = np.random.gamma(shape=0.7,scale=20,size=items)
                x = [ abs(int(i)) for i in x ]
        x = list(np.unique(x))
        x.sort()
        D.append(x)
    return D
```

The **get_frequent** function receives a database as input and some candidate itemsets to test according to some given minimum support, which could be a fraction or an integer.Then counts and gives an etiquette of which patterns are frequent.

```
In [3]: def get_frequent(data_base, candidates,
                        min_sup):
    n_tran = len(data_base)
    if isinstance(min_sup, float) and min_sup <= 1 and min_sup >= 0:
        cut_off = min_sup * n_tran
    elif isinstance(min_sup, int) and min_sup >= 1:
        cut_off = min_sup
    else:
        return 'Error: min_sup must be between 0.0 and 1.0 or a positive integer'
    counts = []
    Freqs = []
    for candidate in candidates:
        candidate_count = 0
        for transaction in data_base:
            is_subset = set(candidate) <= set(transaction)
            if is_subset:
                candidate_count += 1
        if candidate_count >= cut_off:
            Freqs.append(candidate)
            counts.append(candidate_count)
    return {'frequent': Freqs, 'counts': counts}
```

The **get_candidates** function receives a k-1 frequent itemsets and generates the set of k item candidates, performing the joining operation between the k-1 frequent itemsets.

```
In [4]: def get_candidates(Freqs):
            # From the L_k frequent
            # get the C_k+1 candidates
            offset = 1
            k = len(Freqs)
            candidates = []
            # Join Step
            for idx in range(k):
                for idx2 in range( idx + offset, k ):
                    candidate1 = join(Freqs[idx], Freqs[idx2])
                    candidate2 = join(Freqs[idx2], Freqs[idx])
                    if candidate1:
                        candidates.append(candidate1)
                    elif candidate2:
                        candidates.append(candidate2)
                    else:
                        pass
            # Prune Step
            final_candidates = []
            for candidate in candidates:
                 if not prune(candidate,Freqs):
                     final_candidates.append(candidate)
            return final_candidates
```

The **join** function performs the join operation between two itemsets.

```
In [5]: def join(pattern1,pattern2):
            k = len(pattern1)
            candidate_k1 = []
            for idx in range(k):
                if pattern1[idx] == pattern2[idx] and idx + 1 < k:
                    candidate_k1.append(pattern1[idx])
                # The less than equal comparission preserves the
                # lexicographic order
                elif pattern1[idx] < pattern2[idx] and idx + 1 == k:
                    candidate_k1.append(pattern1[idx])
                    candidate_k1.append(pattern2[idx])
                else:
                    candidate_k1 = []
                    break
            return candidate_k1
```

The **prune** function further reduces the candidate list using the apriori property.

```
In [6]: def prune(candidate,Freqs):
            n = len(candidate)
```

```
            subsets = list( itertools.combinations( candidate, n-1) )
            eliminate_candidate = False
            for sub in subsets:
                find_it = False
                for freq in Freqs:
                    is_subset = set(sub) <= set(freq)
                    if is_subset:
                        find_it = True
                        break
                if not find_it:
                    eliminate_candidate = True
                    break
            return eliminate_candidate
```

The **generate_1k_candidates** function is used only once to generate the size 1 candidates that are just all the 1-itemsets.

```
In [7]: def generate_1k_candidates(database):
            item_union = set()
            candidates_1k = []
            for transaction in database:
                x = set(transaction)
                item_union = item_union | x
            for item in item_union:
                candidates_1k.append([item])
            return candidates_1k
```

4 Global Settings

Setting seed for reproducibility.

```
In [8]: np.random.seed(442)
```

5 Main Program

```
In [17]: def main(data_base, min_sup, min_conf):
             n = len(data_base)
             #
             C1 = generate_1k_candidates(data_base)
             Ck = C1
             FREQUENT = {}
             i = 0
             #
             while Ck: # Candidate Set is not Empty
                 i += 1
                 Lk = get_frequent( data_base, Ck, min_sup )
                 FREQUENT[i] = Lk
```

```
    Ck = get_candidates( Lk['frequent'] )
#
print( '\n\nPrinting Frequent Itemsets\n'.format() )
print( 'With support of {}\n'.format(min_sup) )
print( 'From a data base with {} transactions\n'.format(n) )
print( '###################\n'.format(n) )
#
for key in range( 1, len(FREQUENT)+1 ):
    print('Size {} Frequent Itemsets:\n'.format(key))
    for idx,freq in enumerate( FREQUENT[key]['frequent'] ):
        count = FREQUENT[key]['counts'][idx]
        relative = round( (count / n) * 100, 2 )
        print('Set {} with count: {} or {}%'\
              .format( freq, count, relative ))
    print( '\n###################\n'.format(n) )
# Indexing the counts
pattern_counts = {}
for size in FREQUENT:
    for idx,key in enumerate(FREQUENT[size]['frequent']):
        key = str(key)
        value = FREQUENT[size]['counts'][idx]
        pattern_counts[key] = value
#
print('Printing Strong Assosiation Rules\n\n')
n = len(FREQUENT)
for key in range(2,n+1):
    for pattern in FREQUENT[key]['frequent']:
        subsets = [ list(itertools.combinations( pattern, i ))\
                    for i in range(1,len( pattern )) ]
        for cardinality in range(len(subsets)):
            for subset in subsets[cardinality]:
                l = set(pattern)
                s = set(subset)
                l_count = pattern_counts[str(list(l))]
                s_count = pattern_counts[str(list(s))]
                confidence = l_count / s_count
                per_confidence = round( (l_count / s_count) * 100, 2 )
                l_minus_s = list(l - s)
                if confidence >= min_conf:
                    print('Rule {} -> {} has confidence of: {}%'.\
                          format( list(s), l_minus_s, per_confidence ))
```

6 Running

6.0.1 Testing the book example.

```
In [15]: D = [
             [1,2,5],
```

```
                [2,4],
                [2,3],
                [1,2,4],
                [1,3],
                [2,3],
                [1,3],
                [1,2,3,5],
                [1,2,3],
                ]
```

```
In [18]: main(data_base = D, min_sup = 2/9, min_conf = 0.70)
```

```
Printing Frequent Itemsets

With support of 0.2222222222222222

From a data base with 9 transactions

###################

Size 1 Frequent Itemsets:

Set [1] with count: 6 or 66.67%
Set [2] with count: 7 or 77.78%
Set [3] with count: 6 or 66.67%
Set [4] with count: 2 or 22.22%
Set [5] with count: 2 or 22.22%

###################

Size 2 Frequent Itemsets:

Set [1, 2] with count: 4 or 44.44%
Set [1, 3] with count: 4 or 44.44%
Set [1, 5] with count: 2 or 22.22%
Set [2, 3] with count: 4 or 44.44%
Set [2, 4] with count: 2 or 22.22%
Set [2, 5] with count: 2 or 22.22%

###################

Size 3 Frequent Itemsets:

Set [1, 2, 3] with count: 2 or 22.22%
Set [1, 2, 5] with count: 2 or 22.22%
```

```
##################

Printing Strong Assosiation Rules


Rule [5] -> [1] has confidence of: 100.0%
Rule [4] -> [2] has confidence of: 100.0%
Rule [5] -> [2] has confidence of: 100.0%
Rule [5] -> [1, 2] has confidence of: 100.0%
Rule [1, 5] -> [2] has confidence of: 100.0%
Rule [2, 5] -> [1] has confidence of: 100.0%
```

6.0.2 Testing the code with a random generated database.

Generating a database with 100 transactions with an average of 4 elements per transactions and at maximum 6 (0-5) different transactions.

```
In [19]: DB = generate_random_transaction_db(100,4,5)
         main(data_base = DB, min_sup = 0.10, min_conf = 0.50)
```

```
Printing Frequent Itemsets

With support of 0.1

From a data base with 100 transactions

##################

Size 1 Frequent Itemsets:

Set [0] with count: 33 or 33.0%
Set [1] with count: 58 or 58.0%
Set [2] with count: 48 or 48.0%
Set [3] with count: 50 or 50.0%
Set [4] with count: 45 or 45.0%
Set [5] with count: 21 or 21.0%

##################

Size 2 Frequent Itemsets:

Set [0, 1] with count: 16 or 16.0%
Set [0, 2] with count: 17 or 17.0%
Set [0, 3] with count: 10 or 10.0%
Set [0, 4] with count: 13 or 13.0%
```

```
Set [0, 5] with count: 14 or 14.0%
Set [1, 2] with count: 26 or 26.0%
Set [1, 3] with count: 32 or 32.0%
Set [1, 4] with count: 29 or 29.0%
Set [1, 5] with count: 14 or 14.0%
Set [2, 3] with count: 25 or 25.0%
Set [2, 4] with count: 24 or 24.0%
Set [2, 5] with count: 10 or 10.0%
Set [3, 4] with count: 24 or 24.0%

##################

Size 3 Frequent Itemsets:

Set [1, 2, 3] with count: 17 or 17.0%
Set [1, 2, 4] with count: 14 or 14.0%
Set [1, 3, 4] with count: 17 or 17.0%
Set [2, 3, 4] with count: 13 or 13.0%

##################

Size 4 Frequent Itemsets:

Set [1, 2, 3, 4] with count: 10 or 10.0%

##################

Printing Strong Assosiation Rules


Rule [0] -> [2] has confidence of: 51.52%
Rule [5] -> [0] has confidence of: 66.67%
Rule [2] -> [1] has confidence of: 54.17%
Rule [1] -> [3] has confidence of: 55.17%
Rule [3] -> [1] has confidence of: 64.0%
Rule [1] -> [4] has confidence of: 50.0%
Rule [4] -> [1] has confidence of: 64.44%
Rule [5] -> [1] has confidence of: 66.67%
Rule [2] -> [3] has confidence of: 52.08%
Rule [3] -> [2] has confidence of: 50.0%
Rule [2] -> [4] has confidence of: 50.0%
Rule [4] -> [2] has confidence of: 53.33%
Rule [4] -> [3] has confidence of: 53.33%
Rule [1, 2] -> [3] has confidence of: 65.38%
Rule [1, 3] -> [2] has confidence of: 53.12%
Rule [2, 3] -> [1] has confidence of: 68.0%
Rule [1, 2] -> [4] has confidence of: 53.85%
Rule [2, 4] -> [1] has confidence of: 58.33%
```

```
Rule [1, 3] -> [4] has confidence of: 53.12%
Rule [1, 4] -> [3] has confidence of: 58.62%
Rule [3, 4] -> [1] has confidence of: 70.83%
Rule [2, 3] -> [4] has confidence of: 52.0%
Rule [2, 4] -> [3] has confidence of: 54.17%
Rule [3, 4] -> [2] has confidence of: 54.17%
Rule [1, 2, 3] -> [4] has confidence of: 58.82%
Rule [1, 2, 4] -> [3] has confidence of: 71.43%
Rule [1, 3, 4] -> [2] has confidence of: 58.82%
Rule [2, 3, 4] -> [1] has confidence of: 76.92%
```