FP_growth

February 22, 2021

[1]:
```
from collections import OrderedDict
from pprint import pprint
from math import ceil
from itertools import  combinations
import pandas as pd
```

1 Group members

Irfan Sheikh

Nishant Yadav

Vineet Kumar

2 FP-growth algorithm

Discovering frequent itemsets without candidate generation.

Input * `D` : transaction database containing `N` transactions. * `min_sup` : minimum support threshold in percentage

Output * The complete set of frequent patterns.

2.1 Construction of FP-tree

This takes only **two** scans of the database.

1. In the first scan, derive the set of frequent items (1-itemsets) and their support counts (frequencies), and, sort it in nonincreasing order of support count to obtain `L`. This is implemented in the `derive_L` function which returns `L`.

2. In the second scan, we create the root of FP-tree, and process the items in each transaction in `L`-order to create the branches of the tree. This is implemented in the `generate_FP_tree` function.

2.1.1 Deriving L

[2]:
```
def derive_L(D_items):
  # We maintain a dictionary to store the support counts and node links of each␣
 ↪frequent item.
```

```python
    # The dictionary is indexed by the item ID.

    item_scn = dict()

    for i, trans in D_items.items():
        t_items = list(set(trans.split(',')))

        for item in t_items:
            if item not in item_scn:
                item_scn[item] = {'sc' : 1, 'node_link' : []}
            else:
                item_scn[item]['sc'] += 1

    # remove items with support count < min_sup_count to obtain frequent␣
↪1-itemsets, F
    F = { k : v for k, v in item_scn.items() if v['sc'] >= min_sup_count}

    # sort in nonincreasing order of support count to obtain L
    L = OrderedDict(sorted(F.items(),
                          key=lambda kv: kv[1]['sc'],
                          reverse=True))
    return L
```

2.1.2 Constructing the actual FP-tree

[3]:
```python
class Node:
  def __init__(self, item_name, parent, count = 1):
    self.item_name = item_name
    self.parent = parent
    self.children = []
    self.count = count

# p_P -> transaction itemset
# T -> FP tree
# L -> frequent 1-itemset
def insert_tree(p_P, T, L):
  if p_P == []:
    return

  p = p_P[0]
  p_P.pop(0)
  P = p_P

  N = None
  child_present = False
  for child in T.children:
    if child.item_name == p:
```

```
      child.count += 1
      child_present = True
      N = child
      break

  if not child_present:
    N = Node(p, T)
    L[N.item_name]['node_link'].append(N)
    T.children.append(N)

  insert_tree(P, N, L)


def generate_FP_tree(D_items, L):

  T = Node("null", None)

  for i, trans in D_items.items():

    # sort the transaction in L-order
    trans = trans.split(',')

    # select only the items that are in L - i.e., are frequent
    trans = [item for item in trans if item in L]
    trans = list(set(trans))

    # sort these items in L-order
    trans.sort(key=lambda item: L[item]['sc'], reverse=True)

    """
     trans = [p|P], p is the first element and P is the remaining list
     Call insert_tree([p|P], T).
     If T has a child N such that N.item-name = p.item-name:
       increment N's count by 1;
     else:
       create a new node N, and let its count be 1,
       link its parent link to T, and
       link its node-link to the nodes with the same item-name via the␣
 ↪node-link structure.
     If P is nonempty, call insert tree(P, N) recursively.
    """
    insert_tree(trans, T, L)
  return T

def print_tree(T):
    if (T == []):
        return
```

```
    Q = [T]

    while (Q != []):
        n = len(Q)
        while (n > 0):
            node = Q.pop(0)
            if (node.parent != None):
              print(f'{node.item_name}: {node.count} ({node.parent.
↪item_name})', end="  ")
            else:
              print(f'{node.item_name}: {node.count}', end="  ")
            for child in node.children:
                Q.append(child)
            n -= 1;
        print()
```

2.2 Mining the FP-tree

Now, we generate the frequent patterns using the FP-tree that we generated. This is implemented in the `mine_FP_tree` function.

[4]:

```
"""
generate_conditional_pattern_base returns conditional pattern base for the item␣
↪'i'.
Input:
1. T : FP Tree
2. L : frequent 1-itemsets with their nodelinks
3. i : item for which we need to find conditional pattern base

Output:
1. result, a list of all the conditional patterns i.e. all the paths in the␣
↪FP-tree that goes from item 'i' to root node
2. item_sc, support count of all the items that occur in the conditional␣
↪pattern base of item 'i'
"""
def generate_conditional_pattern_base(T, L, i):
  result = []
  link = L[i]['node_link']
  item_sc = {}
  for node in link:
    path = []
    temp = node.parent
    sc = node.count
    while temp.item_name != "null":

      if temp.item_name in item_sc.keys():
        item_sc[temp.item_name] += sc
```

```
      else:
        item_sc[temp.item_name] = sc

      path.insert(0, (temp.item_name, sc))
      temp = temp.parent

    if path:
      result.append(path)

  return result, item_sc


"""
generate_conditional_FP_tree generates conditional the FP tree given conditonal
pattern base of item 'i'.
Input:
CPB : conditional pattern base
item_sc : the support counts of items occuring in the CPB

Output:
C_FP_Tree, the list of all the paths in the conditional FP tree
"""
def generate_conditional_FP_tree(L, i, CPB, item_sc, min_sup):

  #inner function
  # generates all the paths in the conditional FP tree
   def get_path(T, path):
    for child in T.children:
      path.append((child.item_name, child.count))
      get_path(child, path)

    if path and T.children == []:
      C_FP_tree.append(path.copy())

    if path:
      path.pop()

  #inner function
  # insert a conditional pattern 'p_P' in the conditional FP tree
   def insert_tree(p_P, T, i = 0):
     if i == len(p_P):
       return

     p, sc = p_P[i]
     while item_sc[p] < min_sup:
       i += 1
       if i < len(p_P):
```

```
          p, sc = p_P[i]
        else:
          break


      if i == len(p_P):
        return

      N = None
      child_present = False
      for child in T.children:
        if child.item_name == p:
          child.count += sc
          child_present = True
          N = child
          break

      if not child_present:
        N = Node(p, T, sc)
        T.children.append(N)

      insert_tree(p_P, N, i+1)



    T = Node("null", None)
    # inserting all the patterns in CPB in conditional FP tree
    for pattern in CPB:
      insert_tree(pattern, T)

    C_FP_tree = []
    path = []
    get_path(T, path)

    return C_FP_tree

# generates all the frequent patterns given all the paths in conditional FP␣
 ↪tree of the item 'item'
def generate_frequent_patterns(paths, item):
  fp = []
  sc = []
  for path in paths:
    for i in range(1, len(path) + 1):
      for comb in combinations(path, i):
        x = [i for i, j in comb]
        x.append(item)
```

```
        y = [j for i, j in comb]

        temp = min(y)

        if x in fp:
          ind = fp.index(x)
          sc[ind] += temp
        else:
          fp.append(x.copy())
          sc.append(temp)
  return fp, sc

"""
mine_FP_tree mines the FP tree.
Input:
1. L : frequent 1-itemset
2. T : FP tree
3. min_sup_count

Output:
1. The conditional pattern bases
2. The condition FP trees.
3. The frequent patterns.
"""
def mine_FP_tree(L, T, min_sup_count):
  data_CPB = []
  data_CFP_tree = []
  data_FP_patterns = []

  for item in L:
    # print(item)
    CPB, item_sc = generate_conditional_pattern_base(T, L, item)

    # print("conditional pattern base:")
    paths = generate_conditional_FP_tree(L, item, CPB, item_sc, min_sup_count)
    fp, sc = generate_frequent_patterns(paths, item)

    gather_data(data_CPB, data_CFP_tree, data_FP_patterns, item, CPB, paths,␣
↪fp, sc)

  return create_dataframes(data_CPB, data_CFP_tree, data_FP_patterns)

def gather_data(data_CPB, data_CFP_tree, data_FP_patterns, item, CPB, paths,␣
↪fp, _sc):
  x = 1
  for pattern in CPB:
    temp = [i for i, j in pattern]
```

```
    sc = pattern[0][1]
    data_CPB.append([item, x, temp, sc])
    x += 1

  x = 1
  for path in paths:
    data_CFP_tree.append([item, x, path])
    x += 1

  x = 1
  for i in range(len(fp)):
    data_FP_patterns.append([item, x, fp[i], _sc[i]])
    x += 1

def create_dataframes(data_CPB, data_CFP_tree, data_FP_patterns):
  df1 = pd.DataFrame(data_CPB, columns=['item', 'p_num', 'pattern', 'sc'])
  df1.set_index(['item', 'p_num'], inplace = True)

  df2 = pd.DataFrame(data_CFP_tree, columns = ['item', 'p_num', 'path'])
  df2.set_index(['item', 'p_num'], inplace = True)

  df3 = pd.DataFrame(data_FP_patterns, columns= ['item', 'p_num', 'frequent␣
↪pattern', 'support count'])
  df3.set_index(['item', 'p_num'], inplace = True)
  return df1, df2, df3
```

3 Running the FP-growth algorithm on a dataset

[5]:
```
D = pd.read_csv('https://raw.githubusercontent.com/genericSpecimen/college-work/
↪master/data-mining/python/transactions-data.tsv', sep='\t')
N = len(D)

min_sup = 22 # %age
min_sup_count = ceil(min_sup / 100 * N)

D
```

[5]:

	TID	Item_IDs
0	T100	I1,I2,I5
1	T200	I2,I4
2	T300	I2,I3
3	T400	I1,I2,I4
4	T500	I1,I3
5	T600	I2,I3
6	T700	I1,I3
7	T800	I1,I2,I3,I5

```
8  T900     I1,I2,I3
```

DERIVING FREQUENT-1 ITEMSETS

[6]:
```
L = derive_L(D['Item_IDs'])
pprint(L)
```

```
OrderedDict([('I2', {'node_link': [], 'sc': 7}),
             ('I1', {'node_link': [], 'sc': 6}),
             ('I3', {'node_link': [], 'sc': 6}),
             ('I5', {'node_link': [], 'sc': 2}),
             ('I4', {'node_link': [], 'sc': 2})])
```

GENERATING FP-TREE

[7]:
```
T = generate_FP_tree(D['Item_IDs'], L)

# (.) denotes parent
print_tree(T)
```

```
null: 1
I2: 7 (null)  I3: 2 (null)
I1: 2 (I2)  I4: 1 (I2)  I3: 4 (I2)  I1: 2 (I3)
I5: 1 (I1)  I4: 1 (I1)  I1: 2 (I3)
I5: 1 (I1)
```

MINING FP-TREE

[8]:
```
df1, df2, df3 = mine_FP_tree(L, T, min_sup_count)
```

[9]:
```
print('Conditional pattern base')
df1
```

```
Conditional pattern base
```

[9]:

item	p_num	pattern	sc
I1	1	[I2]	2
	2	[I3]	2
	3	[I2, I3]	2
I3	1	[I2]	4
I5	1	[I2, I1]	1
	2	[I2, I3, I1]	1
I4	1	[I2]	1
	2	[I2, I1]	1

[10]:
```
print('Conditional FP tree')
df2
```

```
Conditional FP tree
```

[10]:
```
                              path
item p_num
I1   1      [(I2, 4), (I3, 2)]
     2               [(I3, 2)]
I3   1               [(I2, 4)]
I5   1      [(I2, 2), (I1, 2)]
I4   1               [(I2, 2)]
```

[11]:
```
print('Frequent patterns')
df3
```

```
Frequent patterns
```

[11]:
```
           frequent pattern  support count
item p_num
I1   1             [I2, I1]              4
     2             [I3, I1]              4
     3         [I2, I3, I1]              2
I3   1             [I2, I3]              4
I5   1             [I2, I5]              2
     2             [I1, I5]              2
     3         [I2, I1, I5]              2
I4   1             [I2, I4]              2
```

4 Another dataset

[12]:
```
D = pd.read_csv('https://raw.githubusercontent.com/genericSpecimen/college-work/
↪master/data-mining/python/monkey.tsv', sep='\t')
N = len(D)

min_sup = 60 # %age
min_sup_count = ceil(min_sup / 100 * N)

D
```

[12]:
```
    TID items_bought
0  T100  M,O,N,K,E,Y
1  T200  D,O,N,K,E,Y
2  T300      M,A,K,E
3  T400    M,U,C,K,Y
4  T500  C,O,O,K,I,E
```

DERIVING FREQUENT-1 ITEMSETS

[13]:
```
L = derive_L(D['items_bought'])
pprint(L)
```

```
OrderedDict([('K', {'node_link': [], 'sc': 5}),
```

```
                ('E', {'node_link': [], 'sc': 4}),
                ('M', {'node_link': [], 'sc': 3}),
                ('O', {'node_link': [], 'sc': 3}),
                ('Y', {'node_link': [], 'sc': 3})])
```

GENERATING FP-TREE

[14]:
```python
T = generate_FP_tree(D['items_bought'], L)

# (.) denotes parent
print_tree(T)
```

```
null: 1
K: 5 (null)
E: 4 (K)   M: 1 (K)
M: 2 (E)   Y: 1 (E)   O: 1 (E)   Y: 1 (M)
O: 1 (M)   O: 1 (Y)
Y: 1 (O)
```

MINING FP-TREE

[15]:
```python
df1, df2, df3 = mine_FP_tree(L, T, min_sup_count)
```

[16]:
```python
df1
```

[16]:

item	p_num	pattern	sc
E	1	[K]	4
M	1	[K, E]	2
	2	[K]	1
O	1	[K, E, M]	1
	2	[K, E, Y]	1
	3	[K, E]	1
Y	1	[K, E, M, O]	1
	2	[K, E]	1
	3	[K, M]	1

[17]:
```python
df2
```

[17]:

item	p_num	path
E	1	[(K, 4)]
M	1	[(K, 3)]
O	1	[(K, 3), (E, 3)]
Y	1	[(K, 3)]

[18]:
```python
df3
```

[18]:

```
                frequent pattern  support count
item p_num
E    1                    [K, E]              4
M    1                    [K, M]              3
O    1                    [K, O]              3
     2                    [E, O]              3
     3                 [K, E, O]              3
Y    1                    [K, Y]              3
```

[18]: