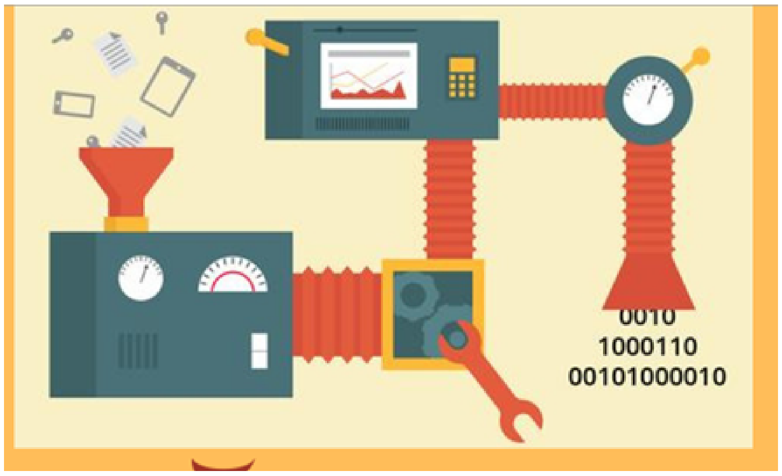
Project PostQL

Closed Frequent Itemset Mining Using FPG and Apriori Algorithms



Vikrant Dewangan **Bhavyajeet Singh**

2018111024 2018111022

INTRODUCTION

To implement the FP Growth algorithm and Apriori algorithm to generate closed frequent itemsets at run-time. Implement Hash-based and Partitioning within Apriori optimizations and analyze performance of each.

FILE STRUCTURE

`./`

- `data/:` The .txt files containing Skin, Mushroom and USCensus data.
- `testing.py`: Testing of 7 algorithms comprising various library implementations and optimizations.
- `apriori.py`: Implementation of Apriori algorithm with Hash based optimization
- `partition.p`: Implementation of Apriori algorithm with Partition based optimization
- `FPGrowth.py`: Implementation of FPGrowth algorithm code.
- `utils.py`: Utilities for finding closed frequent itemsets and data cleaning.

INSTRUCTIONS TO RUN

FP Growth

- To test on datasets, run the file `testing.py`. [0] , [1],and [2] correspond to FPGrowth. The parameters `min_support` and `dataset_name` can be varied above.

Apriori Algorithm

- To test on datasets, run the file `testing.py`. [4], [5], [6], and [7] correspond to FPGrowth. The parameters `min_support` and `dataset_name` can be varied above.

DATA SELECTION AND PRE-PROCESSING

Dataset Choices

We chose the datasets of 2 types majorly -

1. With a small number of unique itemsets but with a large number of rows - to test

the correction and working of our algorithm in theory. With this regard, we used the **SIGN.txt** file.

2. With a large number of unique itemsets - to test the speed. In this regard, we used the

Parsing

Each line corresponds to a transaction, ending with a line denoted by a -2. In a line the items are separated by a -1.

```
def parse_data(file_name):
	file = open(file_name)
	barr = []
	for ind, line in enumerate(file):
	arr = []
	for i in line.split(" "):
		try:
			arr.append(int(i))
		except:
			pass
#	To handle -1 and -2
	barr.append(np.array(arr[::2])[:-1])

#	To handle datasets without -1 and -2
#		barr.append(np.array(arr))
	return np.array(barr)
```

METHOD

FP Growth Algorithm

Theory

Problems with Apriori :- In many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs:

- It may still need to generate a huge number of candidate sets. For example, if there are 104 frequent 1-itemsets, the Apriori algorithm will need to generate more than 107 candidate 2-itemsets.
- It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching. It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

FP-growth, adopts a divide-and-conquer strategy as follows. First, it compresses the database representing frequent items into a frequent pattern tree, or FP-tree, which retains the itemset association information. It then divides the compressed database into a set of conditional databases (a special kind of projected database), each associated with one frequent item or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore, this approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined.

Implementation

On initialization, Conditional Patterns are formed and for each pattern, Conditioned FPTree is built. The ordering of the dataset is given in the ordering named variable.

Initially, a dictionary is built named count for storing the counts of each node.

```
class ConditionedFPTree(object):
      def __init__(self, ordering=None):
      self.ordering = ordering
      self.root = Node(None)
      self.nodes = collections.defaultdict(list)
      self.cond_items = []

      def conditional_tree(self, cond_item, minsup):
      branches = []
      count = collections.defaultdict(int)
      for node in self.nodes[cond_item]:
            branch = node.fine_path_from_root()
            branches.append(branch)
            for item in branch:
                  count[item] += node.count

      items = [item for item in count if count[item] >= minsup]
      items.sort(key=count.get)
      ordering = {item: i for i, item in enumerate(items)}

      cond_tree = ConditionedFPTree(ordering)
      for idx, branch in enumerate(branches):
            branch = sorted([i for i in branch if i in ordering],
                                  key=ordering.get, reverse=True)
            cond_tree.insert_transaction(branch,
self.nodes[cond_item][idx].count)
```

```
      cond_tree.cond_items = self.cond_items + [cond_item]
      return cond_tree
```

Merge Optimization

For merge optimization, while traversing a path, instead of inserting an itemset multiple times, we insert it **single time** into our itemset while considering the data. This greatly reduces our effort of searching and time complexity.

```
      # Generate conditional trees to generate frequent itemsets one item
larger and merge based on path
      if not tree.is_path() and (not max_len or max_len >
len(tree.cond_items)):
      for item in items:
            cond_tree = tree.conditional_tree(item, minsup)
            for sup, iset in compute_fpg(cond_tree, minsup,
                                    colnames, max_len, verbose):
                  yield sup, iset
```

Apriori Algorithm

Hash-based (K=2) Optimization

Theory

A hash-based technique can be used to reduce the size of the candidate k-itemsets, C_k , for k > 1. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, L1, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different buckets of a hash table structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate k-itemsets examined (especially when k= 2).

Implementation

Without optimization:

The apriori starts off by counting the occurrences of itemsets in transactions, subsequently identifying frequent itemsets (which are above the **support** threshold).

```
def iter(records):
      candidates = defaultdict(def_value)
      final = defaultdict(def_value)
      for i in records:
      for element in i :
            candidates[element] += 1

      for key,value in candidates.items():
            if value >= thresh:
                  final[key] = value

      return final
```

Hash based optimization:

The apriori algorithm for hash based optimization can be divided into 2 iterations iter1 and iter2. Each iteration generates the subsequent itemsets from the previous case by considering the **support** threshold.

```
def iter1(records):
      candidates = defaultdict(def_value)
      final = defaultdict(def_value)
      for i in records:
      for element in i :
            candidates[element] += 1

      for key,value in candidates.items():
            if value >= thresh:
                  final[key] = value

      return final

def iter2(final1,records):
      candidates = {}
      final = defaultdict(def_value)
      for i in itertools.combinations(final1,2):
      candidates[i] = 0

      for j in records :
            if set(i).issubset(set(j)):
                  candidates[i]+=1
```

```
    for key,value in candidates.items():
        if value >= thresh:
            final[key] = value

    return final
```

Time Complexity: O(M * N) for where M is the total number of rows and N is the total number of items.

Space Complexity: O(N) for where N is the total number of items.

Partitioning (K >= 2)

Theory

Partitioning technique consists of two phases-

- In phase I, the algorithm divides the transactions of D into *n* non-overlapping partitions. If the minimum relative support threshold for transactions in D is min sup, then the minimum support count for a partition is min_sup × the number of transactions in that partition. For each partition, all the local frequent itemsets (i.e., the itemsets frequent within the partition) are found. A local frequent itemset may or may not be frequent with respect to the entire database, D. However, any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions. 8 Therefore, all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the global candidate itemsets with respect to D.
- In phase II, a second scan of D is conducted in which the actual support of each candidate is assessed to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.

Implementation

The apriori starts off by counting the occurrences of itemsets in transactions, subsequently identifying frequent itemsets (which are above the **support** threshold).

```
def partition(records,n):
    listorec = list(split(records, n))
    cand = []
    candkeys = []
    for rec in listorec :

    print (rec)
    print ("="*30)

    cand.append(apriori(rec,thresh))
    candkeys += list(apriori(rec,thresh).keys())

    new = []
    for i in candkeys:
         new.append(frozenset(i))
    candkeys = set(new)

    print ("candkeys ==== ")
    print (candkeys)

    final = defaultdict(def_value)

    for i in records :
          for element in candkeys:
                if set(element).issubset(set(i)):
                      final[element] += 1

    lol = {}
    for key,value in final.items():
          if value >= thresh:
                lol[key] = value


    return lol
```

Time Complexity: O(P * M * N) for where P is the number of partitions, M is the total number of rows and N is the total number of items.

Space Complexity: O(P*N) for where N is the total number of items.

RESULTS

Please find our analysis of run-times below -

1: Apriori Library

2: Apriori Baseline without optimization

3: Apriori with Hash based

4: Apriori with Partitioning

5: FP Growth Library

6: FP Growth Baseline without optimization

7: FP Growth with Merging

TIME TAKEN

Dataset Used	Minsup Ratio	1	2	3	4	5	6	7
SIGN.txt	0.2					3.57	7.23	6.57
	0.3					4.59	8.48	7.23
	0.4					2.23	7.24	6.38
	0.5					5.72	9.62	5.52
Skin.txt	0.2		2.53	0.255	3.31	0.55	1.50	1.45
	0.3			0.25	2.53	0.51	1.49	1.47
	0.4				0.96	0.51	1.49	1.48
	0.5				0.44	0.49	1.47	1.46

CONCLUSION

We observe the following order of run-times -

7 > **6** > **5** > **4** > **3** > **2** > **1**

This indeed proves that FPGrowth algorithm with merging is the fastest algorithm.

REFERENCES

1. Dataset Used :- http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php
2. Mlxtend Library :- https://github.com/rasbt/mlxtend
3. Data Mining & Concepts by Morgan Kaufman :- http://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf