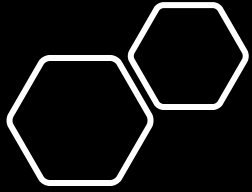


# Implementation of the SPADE Algorithm in a webset



Archishman  
Gupta  
2016A7PS0051H





# Introduction

**Sequential pattern mining** is a topic of data mining concerned with finding statistically relevant patterns between data examples where the values are delivered in a sequence. It is usually presumed that the values are discrete, and thus time series mining is closely related, but usually considered a different activity. Sequential pattern mining is a special case of structured data mining. There are several key traditional computational problems addressed within this field. These include building efficient databases and indexes for sequence information, extracting the frequently occurring patterns, comparing sequences for similarity, and recovering missing sequence members.

# Data Extraction

```
In [3]: file=open('/kaggle/input/msn-dataset/msnbc990928.seq','r')
dataset=[]
for line in file:
    x=line.split()
    dataset.append(x)

In [4]: dataset

Out[4]: [['1', '1'],
['2'],
['3', '2', '2', '4', '2', '2', '2', '3', '3'],
['5'],
['1'],
['6'],
['1', '1'],
['6'],
['6', '7', '7', '7', '6', '6', '8', '8', '8', '8'],
['6', '9', '4', '4', '4', '10', '3', '10', '5', '10', '4', '4', '4'],
['1', '1', '1', '11', '1', '1', '1'],
['12', '12'],
['1', '1'],
['8', '8', '8', '8', '8', '8'],
['6'],
['2'],
['9', '12'],
['3'],
['9']]
```

- The data was extracted in the form a string list of list.
- The macro list contained the collection of lists which represented each user in a span of 24 hours.
- The micro lists in them depicted the hits on a particular website (by indexing) that the user visited in that particular 24 hours.

# Subsequence Checking

```
In [5]: #Checking if a subsequence is a subsequence of the ain sequence

def isSubsequence(mainSequence,subSequence):
    subSequenceClone=list(subSequence)
    return isSubsequenceRecursive(mainSequence,subSequenceClone)

def isSubsequenceRecursive(mainSequence,subSequenceClone,start=0):
    if(not subSequenceClone):
        return True
    firstElem=set(subSequenceClone.pop(0))
    for i in range(start,len(mainSequence)):
        if(set(mainSequence[i]).issuperset(firstElem)):
            return isSubsequenceRecursive(mainSequence,subSequenceClone,i+1)
    return False
```

```
In [6]: sample=[[ '1'],[ '2', '3'],[ '4'],[ '1', '5']]
```

```
In [7]: isSubsequence(sample,[[ '1'],[ '4'],[ '5']])
```

```
Out[7]: True
```

```
In [8]: isSubsequence(sample,[[ '1'],[ '2', '3'],[ '5']])
```

```
Out[8]: True
```

In this **Boolean** function, it was checked if a given sequence is a subsequence of the main sequence or not.

# Computing Support

```
In [12]: #Computing support
def countSupport(dataset,candidateSequence):
    return sum(1 for seq in dataset if isSubsequence(seq,candidateSequence))
```

```
In [13]: dataset
```

```
Out[13]: [['1', '1'],
['2'],
['3', '2', '2', '4', '2', '2', '2', '3', '3'],
['5'],
['1'],
['6'],
['1', '1'],
['6'],
['6', '7', '7', '7', '6', '6', '8', '8', '8', '8'],
['6', '9', '4', '4', '4', '10', '3', '10', '5', '10', '4', '4', '4'],
['1', '1', '1', '11', '1', '1', '1'],
['12', '12'],
['1', '1'],
['8', '8', '8', '8', '8', '8'],
['6'],
['2'],
['9', '12'],
['3'],
['9'],
...]
```

```
In [14]: countSupport(dataset,['2'])
```

```
Out[14]: 264016
```

In this function, the support of a particular sequence with respect to the original dataset was calculated.

# Apriori Candidate generation

```
def generateCandidates(lastLevelCandidates):
    k=sequenceLength(lastLevelCandidates[0])+1
    if (k==2):
        flatShortCandidates=[item for sublist2 in lastLevelCandidates for sublist1
                               result=[[a,b]] for a in flatShortCandidates for b in flatShortCandidates
        result.extend([[a],[b]] for a in flatShortCandidates for b in flatShortCa
        return result
    else:
        candidates=[]
        for i in range(0,len(lastLevelCandidates)):
            for j in range(0,len(lastLevelCandidates)):
                newCand=generateCandidatesForPair(lastLevelCandidates[i],lastLevel
                if (not newCand==[]):
                    candidates.append(newCand)
        candidates.sort()
        return candidates
```

```
In [23]: lastLevelFrequentPatterns =[[['1', '2']], [['2', '3']], [['1'], ['2']], [['1'], ['3']], [[
```

```
In [24]: newCandidates=generateCandidates(lastLevelFrequentPatterns)
newCandidates
```

```
Out[24]: [[['1'], ['2'], ['3']],
           [['1'], ['2', '3']],
           [['1'], ['3'], ['2']],
           [['1'], ['3'], ['3']],
           [['1', '2'], ['3']],
           [['1', '2', '3']],
           [['2'], ['3'], ['2']],
           [['2'], ['3'], ['3']],
           [['2', '3'], ['2']],
```

In this function, the candidates were generated for frequent patterns.

# Candidate Pruning

```
In [26]: #Candidate pruning

def pruneCandidates(candidatesLastLevel,candidatesGenerated):
    return [cand for cand in candidatesGenerated if all(x in candidatesLastLevel for x i
```

```
In [27]: candidatesPruned=pruneCandidates(lastLevelFrequentPatterns,newCandidates)
candidatesPruned
```

```
Out[27]: [[['1'], ['2'], ['3']],
          [['1'], ['2', '3']],
          [['1'], ['3'], ['2']],
          [['1'], ['3'], ['3']],
          [['1', '2'], ['3']],
          [['2'], ['3'], ['3']],
          [['2', '3'], ['3']],
          [['3'], ['2'], ['3']],
          [['3'], ['2', '3']],
          [['3'], ['3'], ['2']],
          [['3'], ['3'], ['3']]]
```

```
In [28]: minSupport=2
candidatesCounts=[(i,countSupport(dataset,i)) for i in candidatesPruned]
resultLvl=[(i,count) for (i,count) in candidatesCounts if(count>=minSupport)]
resultLvl
```

```
Out[28]: (((['1'], ['2'], ['3']], 16048),
          ([['1'], ['3'], ['2']], 14841),
          ([['1'], ['3'], ['3']], 48590),
          ([['1', '2'], ['3']], 11370),
          ([['2'], ['3'], ['3']], 11014),
          ([['3'], ['2'], ['3']], 5173),
          ([['3'], ['3'], ['2']], 9351),
```

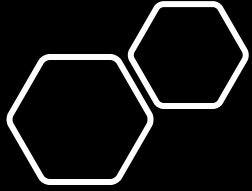
# Sequential Patterns

```
In [29]: #Apriori algorithm

def apriori(dataset,minSupport,verbose=False):
    global numberOfCountingOperations
    numberOfCountingOperations=0
    Overall=[]
    itemsInDataset=sorted(set([item for sublist1 in dataset for sublist2 in sublist1 for item in sublist2]))
    singleItemSequences=[[item] for item in itemsInDataset]
    singleItemCounts=[(i,countSupport(dataset,i)) for i in singleItemSequences if countSupport(dataset,i)>=minSupport]
    Overall.append(singleItemCounts)
    print("Result, lvl 1: "+str(Overall[0]))
    k=1
    while(True):
        if not Overall[k-1]:
            break
        candidatesLastLevel=[x[0] for x in Overall[k-1]]
        candidatesGenerated=generateCandidates(candidatesLastLevel)
        candidatesPruned=[cand for cand in candidatesGenerated if all(x in candidatesLastLevel for x in cand)]
        candidatesCounts=[(i,countSupport(dataset,i)) for i in candidatesPruned]
        resultLvl=[(i,count) for (i,count) in candidatesCounts if (count>=minSupport)]
        if verbose:
            print("Candidates generated, lvl "+str(k+1)+": "+str(candidatesGenerated))
            print("Candidates pruned, lvl "+str(k+1)+": "+str(candidatesPruned))
            print("Result, lvl "+str(k+1)+": "+str(resultLvl))
        Overall.append(resultLvl)
        k=k+1
    Overall=Overall[:-1]
    Overall=[item for sublist in Overall for item in sublist]
    return Overall
```

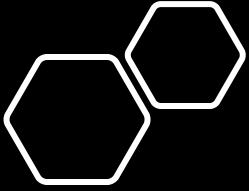
Finally, using the algorithm, we mine for sequential patterns.





# Limitations

- Using this algorithm, too many candidates are generated.
- To mine sequential patterns, an exponential number of short candidates have to be generated. For eg:- A length 100 sequential pattern needs  $10^{30}$  candidate sequences.
- There was a problem regarding some python libraries, hence I uploaded the dataset on **Kaggle** and wrote the code on that notebook. Hence I am attaching a link here for reference  
<https://www.kaggle.com/archi97/kernel121754ab61>



GitHub link:-

<https://github.com/archishman97/SPADE-Implementation>