Mining temporal data – Frequent sequences

Algorithmic Data Analysis – Coding Assignment 2

Giulia Ortolani

March 25, 2024

Resources: No resources
Collaborations: No collaborations

The Generalized Sequential Pattern Mining (GSP) algorithm is an apriori-like algorithm for solving sequential pattern mining, here I try to implement it in Python and use it for mining frequent patterns from discrete sequences. The algorithm is then applied to the CRSW dataset, which contains two months worth of weather data in Kuopio (January–February 2019).

GSP algorithm implementation idea

The implemented algorithm takes a dataset (list of itemsets) and a minimum support threshold. It first identifies frequent items (items with support greater than or equal to the minimum support threshold). Then, starting from $k = 1$, it iteratively generates candidate sequences of length $k + 1$ by joining pairs of sequences from the frequent patterns of length k. The support of these candidates is calculated and only those with support greater than or equal to the minimum support threshold are retained. This process continues until no more frequent patterns can be found. The output of the code is a dictionary where the keys are the frequent patterns and the values are their corresponding support counts.

Application to the CRSW dataset

The weather time series analysis using the Generalized Sequential Pattern (GSP) mining algorithm has revealed interesting patterns in the occurrences of clouds (C), precipitation (R), sunshine (S), and wind (W).

Setting different minimum support threshold

If we set the minimum support threshold to **30**, we get three frequent patterns:

- `'CRW'`, with a support of 225. This is cloudy, rainy and windy weather.
- `'CR'`, with a support of 99. This is cloudy and rainy weather.
- `'CW'`, with a support of 237. This is cloudy and windy weather.

Setting different values for max gap and max span

If we change the values for max gap and max span we can obtain these different results:

Min supp thresh	Max span	Max gap	Frequent patterns
10	5	10	[('CRW', 225), ('CR', 99), ('CW', 237)]
1	1	1	[('CRW', 75), ('CR', 99), ('CW', 237)]
200	20	20	[('CW', 237)]
100	10	10	[('CRW', 225), ('CW', 237)]

Table 1: Frequent patterns using different parameters

A. Code

```
import copy
## GSP
# implementation of the generalized sequential pattern mining (GSP) algorithm for
    mining frequent patterns from discrete sequences

# def get_sequences_from_file(file_path):
#     sequences = []
#     with open(file_path, 'r') as file:
#         for line in file:
#             timestamp, sequence = line.strip().split()[0], line.strip().split()[1:]
#             sequences.append(sequence)
#     return sequences

def get_sequences_from_file(file_path):
    sequences = []
    with open(file_path, 'r') as file:
        for line in file:
            timestamp, sequence = line.strip().split()[0], line.strip().split()[1:]
            split_sequence = [list(itemset) for itemset in sequence]
            sequences.append(split_sequence)
    return sequences



def join_sequences(frequent_items, maxspan, maxgap):
    # candidate (k + 1)-sequences are generated by combining pairs of frequent k-
    sequences
    candidates = []

    # Fix a canonical order on the sequences and ensure that each candidate (k + 1)-
    subsequence is generated by only one pair of k-sequences ??


    for i in range(len(frequent_items)):
        Sa = list(frequent_items[i])

        for j in range(len(frequent_items)):

            Sb = list(frequent_items[j])

            Sc = []

            if check_candidate_gen(Sa,Sb) != 0:
                # If the first itemset of Sa is a singleton, Sc is generated by
    prepending it to Sb
                if len(list(Sa)[0])==1: # here I'm checking if the list
                    Sa = "".join(Sa)
                    Sb = "".join(Sb)
                    Sc.insert(0, Sa)
                    Sc.append(Sb)
                    Sc = "".join(Sc)
                    candidates.append(Sc)

```

```
                # If the first itemset of Sa is not a singleton, Sc is generated by replacing the first itemset of Sb with it
                elif len(list(Sa)[0])>1:
                    Sa = "".join(Sa)
                    Sb = "".join(Sb)
                    Sc.insert(0,Sa)
                    Sc.append(Sb[1:])
                    Sc = "".join(Sc)
                    candidates.append(Sc)

                # If the last itemset of Sb is a singleton, Sc is generated by appending it to Sa
                elif len(list(Sb)[-1])==1:
                    Sa = "".join(Sa)
                    Sb = "".join(Sb)
                    Sc.append(Sa)
                    Sc.append(Sb[-1])
                    Sc = "".join(Sc)
                    candidates.append(Sc)

                # If the last itemset of Sb is not a singleton, Sc is generated by replacing the last itemset of Sa with it
                elif len(list(Sb)[-1])>1:
                    Sa = "".join(Sa)
                    Sb = "".join(Sb)
                    Sc.append(Sa[0:(len(Sa)-2)])
                    Sc.append(Sb[-1])
                    Sc = "".join(Sc)
                    candidates.append(Sc)

    return candidates

def check_candidate_gen(Sa, Sb):
    # removing an item from the first itemset of Sa and removing an item from the
    # last itemset of Sb should result in the same sequence So

    Sa1 = [list(item) for item in Sa]
    Sb2 = [list(item) for item in Sb]

    Sa_red = copy.deepcopy(Sa1)
    Sb_red = copy.deepcopy(Sb2)

    # first itemset of Sa: Sa_red[0]
    # last itemset of Sb: Sb_red[-1]

    k = 0

    for i in range(len(Sa_red[0])):
        # remove an item from the first itemset of Sa
        if i == len(Sa_red[0]):
            break
        Sa_red[0].pop(i)
        Sa_red = [item for item in Sa_red if item != []]

        for j in range(len(Sb_red[-1])):
        # remove an item from the last itemset of Sb
            if j == len(Sb_red[-1]):
                break
            Sb_red[-1].pop(j)
            Sb_red = [item for item in Sb_red if item != []]

            k += (Sa_red == Sb_red)

            Sb_red = copy.deepcopy(Sa1)
        Sa_red = copy.deepcopy(Sb2)

```

```
    #print(k)
    return k

def is_subsequence(candidate, sequence, max_span, max_gap):

    # CONSTRAINTS
    # 1. maximum span limit the time difference between the first and last itemsets of a subsequence
    # 2. maximum gap limit the number of gaps between successive itemsets of a subsequence

    # Initialize indices for candidate and sequence
    i, j = 0, 0
    span_start = None

    # Iterate through both lists
    while i < len(candidate) and j < len(sequence[0]):
        # If the elements match, move to the next element in both lists
        if candidate[i] == sequence[0][j] and abs(i-j) < max_gap:
            if span_start is None:
                span_start = j  # Start of the span
            i += 1
        # Move to the next element in the larger list
        j += 1

    # If all elements of the candidate list are found in the sequence in the same order, check constraints
    if i == len(candidate):
        span_end = j - 1  # End of the span
        span_length = span_end - span_start

        # Check constraints
        if span_length <= max_span:
            return True

    return False

def count_support(dataset, candidates, maxspan, maxgap):

    support = {}
    # the support of subsequence S in sequence D is the number of occurrences of S in D

    for candidate in candidates:
        candidate = list(candidate)
        for itemset in dataset:
            if is_subsequence(candidate, itemset, maxspan, maxgap):
                itemset = "".join(itemset[0])
                if itemset in support:
                    support[itemset] += 1
                else:
                    support[itemset] = 1

    return support

def gsp(dataset, minsup, maxspan, maxgap):
    k = 1
    item_counts = {}

    # Fk: all frequent items

    # count support of items.... or itemsets?
    for sequence in dataset:
        for itemset in sequence:
            for i in itemset:
                if i in item_counts:
```

```python
                    item_counts[i] += 1
                else:
                    item_counts[i] = 1

    # itemsets:
    # for sequence in dataset:
    #     for itemset in sequence:
    #         key = ''.join(itemset)
    #         if key in item_counts:
    #             item_counts[key] += 1
    #         else:
    #             item_counts[key] = 1

    # Fk:
    frequent_items = {item: count for item, count in item_counts.items() if count >= minsup}

    result = {}

    # while Fk != 0
    while len(frequent_items)!=0:

        # generate C_k+1 by joining pairs of sequences from F_k
        candidates = join_sequences(list(frequent_items.keys()), maxspan, maxgap)
        candidate_support = count_support(dataset, candidates, maxspan, maxgap)

        # S in C_k+1
        frequent_items = {pattern: support for pattern, support in candidate_support.items() if support >= minsup} # add to result if the support is > minsup
        result.update(frequent_items) # F_k+1

        k += 1

    return result # dict object made of 'pattern' and 'support'

## Apply the algorithm on the CRSW dataset:
# `2019-01-03_itemsets-CRSW.txt` Two months worth of weather in Kuopio (January February 2019), obtained from https://en.ilmatieteenlaitos.fi/download-observations#!/
# Each line represents weather events during one hour as an itemset. The first column contains the contextual attribute, i.e. indication of time in Year-Month-Day_Hour format.
# The second column contains the itemset. C, R S and W stand respectively for clouds, precipitation, sunshine and wind.

file_path = "2019-01-03_itemsets-CRSW.txt"
minsup = 100
maxspan = 10
maxgap = 10
dataset = get_sequences_from_file(file_path)
result = gsp(dataset, minsup, maxspan, maxgap)

# Print the frequent patterns
print(f"Minimum support threshold = {minsup}")
print(f"Maximum span = {maxspan}")
print(f"Maximum gap = {maxgap}")
print("Frequent patterns:")
print(list(result.items()))

# Try different values for the mininum support threshold, different constraints such as max gap and max span,
# and considering the data either as one long sequence or with each day as a separate short sequence. You might
# also try replacing repeated occurrences of the same itemset by a single copy, i.e. represent constant weather
# during successive hours with only one itemset.
```

```
# Report on the type and number of patt erns obtained under diff erent conditions.
```