frequent-pattern

December 10, 2021

0.1 Homework 1 - Frequent Pattern Analysis

Name: <insert name here> ***

Remember that you are encouraged to discuss the problems with your instructors and classmates, but **you must write all code and solutions on your own**.

The rules to be followed for the assignment are:

- Do **NOT** load additional packages beyond what we've shared in the cells below.
- Some problems with code may be autograded. If we provide a function or class API **do not** change it.
- Do not change the location of the data or data directory. Use only relative paths to access the data.

[271]:

```
import argparse
import pandas as pd
import numpy as np
import random
import pickle
from pathlib import Path
from collections import defaultdict
```

0.1.1 [10 points] Problem 1 - Apriori Implementation

A sample dataset has been provided to you in the './data/dataset.pickle' path. Here are the attributes for the dataset. Use this dataset to test your functions.

- Dataset should load the transactions in the form of a python dictionary where each key holds the transaction id and the value is a python list of the items purchased in that transaction.
- An example transaction will have the following structure. If items A, C, D, F are purchased in transaction T3, this would appear as follows in the dictionary.

```
transactions = {
   "T3": ["A", "C", "D", "F"]
}
```

Note: - A sample dataset to test your code has been provided in the location "./data/dataset.pickle". Please maintain this as it would be necessary while grading. - Do not change the variable names of the returned values. - After calculating each of those values, assign them to the corresponding value that is being returned.

- If you are encountering any errors while loading the dataset, the following lines of code should help. Please delete the cells before submitting, to reduce any potential autograder issues.

```
!pip install pickle5
```

```
import pickle5 as pickle
```

[272]:

```
import itertools

def findsubsets(s, n):

#   A helper function that you can use to list of all subsets of size n. Do not␣
↪make any changes to this code block.
#   Input:
#       1. s - A python list of items
#       2. n - Size of each subset
#   Output:
#       1. subsets - A python list containing the subsets of size n.

    subsets = list(sorted((itertools.combinations(s,n))))
    return subsets
```

[273]:

```
def items_from_frequent_itemsets(frequent_itemset):

#   A helper function that you can use to get the sorted items from the␣
↪frequent itemsets. Do not make any changes
#   to this code block
#   Input:
#       1. Frequent Itemsets
#   Output:
#       1. Sorted list of items

    items = list()
    for keys in frequent_itemset.keys():
        for item in list(keys):
            items.append(item)
    return sorted(list(set(items)))
```

[274]:

```
def generate_frequent_itemsets(dataset, support, items, n=1, frequent_items={}):

#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. support - A floating point variable representing the min_support␣
 ↪value for the set of transactions.
#       3. items - A python list representing all the items that are part of␣
 ↪all the transactions.
#       4. n - An integer variable representing what frequent item pairs to␣
 ↪generate.
#       5. frequent_items - A dictionary representing k-1 frequent sets.
#   Output:
#       1. frequent_itemsets - A dictionary representing the frequent itemsets␣
 ↪and their corresponding support counts.

    frequent_items = {}
    len_transactions = len(dataset)
    temp_dict = {}
    if n == 1:
        # your code here
        for item in items:
            temp_dict[item] = 0
        for key, item_list in dataset.items():
            for sub_item in item_list:
                temp_dict[sub_item] = temp_dict[sub_item] + 1
        for item, supp_perc in temp_dict.items():
            if (temp_dict[item] / len_transactions) >= support:
                frequent_items[item] = supp_perc

    else:
        # your code here
        subset_combos = findsubsets(items, n)
        for i in subset_combos:
            temp_dict[i] = 0
        for key, item_list in dataset.items():
            for i in subset_combos:
                if set(i).issubset(item_list):
                    temp_dict[i] = temp_dict[i] + 1
        for item, supp_perc in temp_dict.items():
            if (temp_dict[item] / len_transactions) >= support:
                frequent_items[item] = supp_perc

    return frequent_items
```

[275]:

```
# This cell has hidden test cases that will run after you submit your␣
 ↪assignment.
```

[276]:

```
import unittest

class TestX(unittest.TestCase):
    def setUp(self):
        self.min_support = 0.5
        self.items = ['A', 'B', 'C', 'D', 'E']
        self.dataset = dict()
        self.dataset["T1"] = ['A', 'B', 'D']
        self.dataset["T2"] = ['A', 'B', 'E']
        self.dataset["T3"] = ['B', 'C', 'D']
        self.dataset["T4"] = ['B', 'D', 'E']
        self.dataset["T5"] = ['A', 'B', 'C', 'D']

    def test0(self):
        frequent_1_itemsets = generate_frequent_itemsets(self.dataset, self.
 ↪min_support, self.items)
        print (frequent_1_itemsets)
        frequent_1_itemsets_solution = dict()
        frequent_1_itemsets_solution['A'] = 3
        frequent_1_itemsets_solution['B'] = 5
        frequent_1_itemsets_solution['D'] = 4

        print ("Test 1: frequent 1 itemsets")
        assert frequent_1_itemsets == frequent_1_itemsets_solution

        frequent_2_itemsets = generate_frequent_itemsets(self.dataset, self.
 ↪min_support, self.items, 2, frequent_1_itemsets)
        print (frequent_2_itemsets)
        frequent_2_itemsets_solution = dict()
        frequent_2_itemsets_solution[('A', 'B')] = 3
        frequent_2_itemsets_solution[('B', 'D')] = 4

        print ("Test 1: frequent 2 itemsets")
        assert frequent_2_itemsets == frequent_2_itemsets_solution

        frequent_3_itemsets = generate_frequent_itemsets(self.dataset, self.
 ↪min_support, self.items, 3, frequent_2_itemsets)
        print (frequent_3_itemsets)
        frequent_3_itemsets_solution = dict()

        print ("Test 1: frequent 3 itemsets")
        assert frequent_3_itemsets == frequent_3_itemsets_solution

tests = TestX()
tests_to_run = unittest.TestLoader().loadTestsFromModule(tests)
unittest.TextTestRunner().run(tests_to_run)
```

```
.
{'A': 3, 'B': 5, 'D': 4}
Test 1: frequent 1 itemsets
```

```
{('A', 'B'): 3, ('B', 'D'): 4}
Test 1: frequent 2 itemsets
{}
Test 1: frequent 3 itemsets

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

[276]:

```
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

0.1.2 [10 points] Problem 2 - FP-Growth Implementation

A sample dataset has been provided to you in the './data/dataset.pickle' path. Here are the attributes for the dataset. Use this dataset to test your functions.

- Dataset should load the transactions in the form of a python dictionary where each key holds the transaction id and the value is a python list of the items purchased in that transaction.
- An example transaction will have the following structure. If items A, C, D, F are purchased in transaction T3, this would appear as follows in the dictionary.

```
transactions = {
    "T3": ["A", "C", "D", "F"]
}
```

Note: - A sample dataset to test your code has been provided in the location "./data/dataset.pickle". Please maintain this as it would be necessary while grading. - Do not change the variable names of the returned values. - After calculating each of those values, assign them to the corresponding value that is being returned.

[277]:

```
def item_support(dataset, min_support):

#   A helper function that returns the support count of each item in the dataset. The dictionary is further sorted
#   based on maximum support of each item and pruned based on min_support.
#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. items - A python list representing all the items that are part of all the transactions.
#       3. min_support - A floating point variable representing the min_support value for the set of transactions.
#   Output:
#       1. support_dict - A dictionary representing the support count of each item in the dataset.

```

```
    len_transactions = len(dataset)
    support_dict = dict()
    for key, value in dataset.items():
        # your code here
        for i in value:
            if i in support_dict.keys():
                support_dict[i] = support_dict[i] + 1
            else:
                support_dict[i] = 1

    sorted_support = dict(sorted(support_dict.items(), key=lambda item:␣
↪item[1], reverse=True))
    pruned_support = {key:val for key, val in sorted_support.items() if val/
↪len_transactions >= min_support}

    support_dict = sorted_support
    return support_dict
```

[278]:

```
# This cell has hidden test cases that will run after you submit your␣
↪assignment.
```

[279]:

```
def reorder_transactions(dataset, min_support):

#   A helper function that reorders the transaction items based on maximum␣
↪support count. It is important that you finish
#   the code in the previous cells since this function makes use of the support␣
↪count dictionary calculated above.
#   Input:
#       1. dataset - A python dictionary containing the transactions.
#       2. items - A python list representing all the items that are part of␣
↪all the transactions.
#       3. min_support - A floating point variable representing the min_support␣
↪value for the set of transactions.
#   Output:
#       1. updated_dataset - A dictionary representing the transaction items in␣
↪sorted order of their support counts.

    pruned_support = item_support(dataset, min_support)
    updated_dataset = dict()

    # This loop sorts the transaction items based on the item support counts
    for key, value in dataset.items():
        updated_dataset[key] = sorted(value, key=pruned_support.get,␣
↪reverse=True)
```

```
    # Update the following loop to remove items that do not belong to the␣
 ↪pruned_support dictionary
    for key, value in updated_dataset.items():
        updated_values = list()
        for item in value:
            # your code here
            if pruned_support[item] >= min_support * len(dataset):
                updated_values.append(item)
        updated_dataset[key] = updated_values

    return updated_dataset
```

[ ]:

```

```

[280]:

```
def build_fp_tree(updated_dataset):

#   Input:
#       1. updated_dataset - A python dictionary containing the updated set of␣
 ↪transactions based on the pruned support dictionary.
#   Output:
#       1. fp_tree - A dictionary representing the fp_tree. Each node should␣
 ↪have a count and children attribute.
#
#   HINT:
#       1. Loop over each transaction in the dataset and make an update to the␣
 ↪fp_tree dictionary.
#       2. For each loop iteration store a pointer to the previously visited␣
 ↪node and update it's children in the next pass.
#       3. Update the root pointer when you start processing items in each␣
 ↪transaction.
#       4. Reset the root pointer for each transaction.
#
#   Sample Tree Output:
#   {'Y': {'count': 3, 'children': {'V': {'count': 1, 'children': {}}}},
#    'X': {'count': 2, 'children': {'R': {'count': 1, 'children': {'F':␣
 ↪{'count': 1, 'children': {}}}}}}}

    fp_tree = dict()
    for key, value in updated_dataset.items():
        root = value[0]
        current_node = None
        previous_node = None
        # your code here
        for i in value:
            if i not in fp_tree:
                fp_tree[i] = {'count': 0, 'children': {}}
            if i in fp_tree:
                fp_tree[i]['count'] = fp_tree[i]['count'] + 1
            if i != root:
                for j in range( value.index(i) - 1, -1, -1):
                    previous_node = j
                    fp_tree[value[previous_node]]['children'] = { value[j+1]:␣
↪fp_tree[value[j + 1]]}



    return fp_tree
```

[281]:

```
example1 = {1: ['A', 'C', 'D', 'G', 'F', 'M', 'P', 'X'], 2: ['A', 'B', 'C',␣
↪'F', 'M', 'O', 'X'], 3: ['B', 'F', 'H', 'J', 'O', 'W'], 4 : ['B', 'C', 'K',␣
↪'P', 'S'], 5: ['A', 'C', 'E', 'F', 'M', 'N', 'P', 'X']}
new_set = reorder_transactions(example1, min_support = 0.6)
build_fp_tree(new_set)
```

[281]:

```
{'C': {'count': 4,
  'children': {'F': {'count': 4,
    'children': {'A': {'count': 3,
      'children': {'M': {'count': 3,
        'children': {'P': {'count': 3,
          'children': {'X': {'count': 3, 'children': {}}}}}}}}}}}},
 'F': {'count': 4,
  'children': {'A': {'count': 3,
    'children': {'M': {'count': 3,
      'children': {'P': {'count': 3,
        'children': {'X': {'count': 3, 'children': {}}}}}}}}}},
 'A': {'count': 3,
  'children': {'M': {'count': 3,
    'children': {'P': {'count': 3,
      'children': {'X': {'count': 3, 'children': {}}}}}}}},
 'M': {'count': 3,
  'children': {'P': {'count': 3,
    'children': {'X': {'count': 3, 'children': {}}}}}},
 'P': {'count': 3, 'children': {'X': {'count': 3, 'children': {}}}},
 'X': {'count': 3, 'children': {}},
 'B': {'count': 3,
  'children': {'P': {'count': 3,
    'children': {'X': {'count': 3, 'children': {}}}}}}}
```

[ ]: