《复杂结构数据挖掘》第一次作业实验报告

191250026 丁云翔

问题和数据集介绍

问题介绍

分别应用Apriori算法、FP-Growth算法和使用穷举搜索的基线方法对超市购物单数据和UNIX命令数据进行数据挖掘，比较性能差异，并挖掘出一些关联规则。

使用的方法和代码实现介绍

项目主程序Assignment1.py为自己原创编写的。在寻找频繁项集的算法上分别使用了Apriori算法、FP-Growth算法和使用穷举搜索的基线方法，其中Apriori算法使用了GitHub上现有的代码实现（apriori.py）；FP-Growth算法基于GitHub上现有的代码实现进行了一些修改，使其支持Python3（fp_growth.py）；使用穷举搜索的基线方法参考了网上的实现进行编写（exhaustive_search.py）。

在Assignment1.py，我对数据进行了一些预处理。对于GroceryStore数据集，直接将其读入即可；而对于UNIX_usage数据集，由于其格式较复杂，且不同用户的数据存储在不同的文件里，我对每个文件都进行了一次预处理，将每个“SOF”与“EOF”间的命令作为一行数据，对于空行，在读取时直接忽略。然后对行内的每一项进行进一步处理，将非字母开头的参数部分与之前的第一个字母开头的项连接，即将命令的参数连接到其所属命令上，以防止参数项对挖掘造成干扰。另外，还对同一行内相同的命令进行了去重（若集合中含有相同项，FP-Growth算法无法运行）。最后再将所有用户的数据连接到一起，对汇总的后的数据进行挖掘。

数据预处理的部分代码如下，首先是GroceryStore，读取完的数据存在data_array中。

```
data = pd.read_csv("dataset/GroceryStore/Groceries.csv")
    data = np.array(data)
    data_array = []
    for item in data:
        row_array = str(item[1])[1:-1].split(',')
        data_array.append(row_array)
```

然后是UNIX_usage，先对数据格式进行处理，将每个“SOF”与“EOF”间的命令作为一行数据。

```
    for i in range(0, 9):
        with open('dataset/UNIX_usage/USER' + str(i) +
'/sanitized_all.981115184025', 'r') as f:
            with open('dataset/UNIX_usage/USER' + str(i) + '/data.csv', 'w') as
g:
                lines = f.readlines()
                begin = False
                for l in lines:
                    t = l.strip('\n')
                    if t == "**SOF**":
                        begin = True
                        continue
                    elif t == "**EOF**":
                        if begin:
                            continue
```

```
                    g.write('\n')
                else:
                    if begin:
                        g.write(t)
                        begin = False
                    else:
                        g.write(',')
                        g.write(t)
```

然后再将所有用户的数据进行汇总，去除空行，并将参数与其所属的命令连接，对同一行内相同的命令进行去重，最后存储在data_array中。

```
data_array = []
    for i in range(0, 9):
        with open('dataset/UNIX_usage/USER' + str(i) + '/data.csv', 'r') as f:
            lines = f.readlines()
            for l in lines:
                row_array = str(l.strip('\n')).split(',')
                real_array = []
                start = True
                temp = ''
                dic = {}
                for item in row_array:
                    if start:
                        temp = item
                        start = False
                    else:
                        if len(item) < 1:
                            continue
                        if 'a' <= item[0] <= 'z':
                            if temp in dic.keys():
                                continue
                            dic[temp] = 1
                            real_array.append(temp)
                            temp = item
                        else:
                            temp = temp + ' ' + item
                real_array.append(temp)
                data_array.append(real_array)
```

下面简单介绍代码中的一些关键函数。Apriori算法的主要实现如下，runApriori函数根据输入数据、最小支持度和最小置信度，返回频繁项集和挖掘出的关联规则。

```
def runApriori(data_iter, minSupport, minConfidence):
    """
    run the apriori algorithm. data_iter is a record iterator
    Return both:
     - items (tuple, support)
     - rules ((pretuple, posttuple), confidence)
    """
    itemSet, transactionList = getItemSetTransactionList(data_iter)

    freqSet = defaultdict(int)
    largeSet = dict()
    # Global dictionary which stores (key=n-itemSets,value=support)
    # which satisfy minSupport
```

```
    assocRules = dict()
    # Dictionary which stores Association Rules

    oneCSet = returnItemsWithMinSupport(itemSet, transactionList, minSupport,
freqSet)

    currentLSet = oneCSet
    k = 2
    while currentLSet != set([]):
        largeSet[k - 1] = currentLSet
        currentLSet = joinSet(currentLSet, k)
        currentCSet = returnItemsWithMinSupport(
            currentLSet, transactionList, minSupport, freqSet
        )
        currentLSet = currentCSet
        k = k + 1

    def getSupport(item):
        """local function which Returns the support of an item"""
        return float(freqSet[item]) / len(transactionList)

    toRetItems = []
    for key, value in largeSet.items():
        toRetItems.extend([(tuple(item), getSupport(item)) for item in value])

    toRetRules = []
    for key, value in list(largeSet.items())[1:]:
        for item in value:
            _subsets = map(frozenset, [x for x in subsets(item)])
            for element in _subsets:
                remain = item.difference(element)
                if len(remain) > 0:
                    confidence = getSupport(item) / getSupport(element)
                    if confidence >= minConfidence:
                        toRetRules.append(((tuple(element), tuple(remain)),
confidence))
    return toRetItems, toRetRules
```

除此之外，还有计算项集中的项的支持度，并返回一个每个项都满足最小支持度的子项集的returnItemsWithMinSupport函数。

```
def returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet):
    """calculates the support for items in the itemSet and returns a subset
    of the itemSet each of whose elements satisfies the minimum support"""
    _itemSet = set()
    localSet = defaultdict(int)

    for item in itemSet:
        for transaction in transactionList:
            if item.issubset(transaction):
                freqSet[item] += 1
                localSet[item] += 1

    for item, count in localSet.items():
        support = float(count) / len(transactionList)
```

```
        if support >= minSupport:
            _itemSet.add(item)

    return _itemSet
```

FP-Growth算法的主要实现如下，find_frequent_itemsets函数根据输入数据和最小支持度，返回满足要求的频繁项集。

```
def find_frequent_itemsets(transactions, minimum_support,
include_support=False):
    """
    Find frequent itemsets in the given transactions using FP-growth. This
    function returns a generator instead of an eagerly-populated list of items.
    The `transactions` parameter can be any iterable of iterables of items.
    `minimum_support` should be an integer specifying the minimum number of
    occurrences of an itemset for it to be accepted.
    Each item must be hashable (i.e., it must be valid as a member of a
    dictionary or a set).
    If `include_support` is true, yield (itemset, support) pairs instead of
    just the itemsets.
    """
    items = defaultdict(lambda: 0) # mapping from items to their supports

    # Load the passed-in transactions and count the support that individual
    # items have.
    for transaction in transactions:
        for item in transaction:
            items[item] += 1

    # Remove infrequent items from the item support dictionary.
    items = dict((item, support) for item, support in items.items()
        if support >= minimum_support)

    # Build our FP-tree. Before any transactions can be added to the tree, they
    # must be stripped of infrequent items and their surviving items must be
    # sorted in decreasing order of frequency.
    def clean_transaction(transaction):
        transaction = filter(lambda v: v in items, transaction)
        transaction = sorted(transaction, key=lambda v: items[v], reverse=True)
        return transaction

    master = FPTree()
    for transaction in map(clean_transaction, transactions):
        master.add(transaction)

    def find_with_suffix(tree, suffix):
        for item, nodes in tree.items():
            support = sum(n.count for n in nodes)
            if support >= minimum_support and item not in suffix:
                # New winner!
                found_set = [item] + suffix
                yield (found_set, support) if include_support else found_set

                # Build a conditional tree and recursively search for frequent
                # itemsets within it.
                cond_tree = conditional_tree_from_paths(tree.prefix_paths(item))
                for s in find_with_suffix(cond_tree, found_set):
```

```
                    yield s # pass along the good news to our caller

    # Search for frequent itemsets, and yield the results we find.
    for itemset in find_with_suffix(master, []):
        yield itemset
```

除此之外，还有FP-Growth算法的核心数据结构FPTree和其上的节点FPNode。

```
class FPTree(object):
    """
    An FP tree.
    This object may only store transaction items that are hashable
    (i.e., all items must be valid as dictionary keys or set members).
    """

    Route = namedtuple('Route', 'head tail')

    def __init__(self):
        # The root node of the tree.
        self._root = FPNode(self, None, None)

        # A dictionary mapping items to the head and tail of a path of
        # "neighbors" that will hit every node containing that item.
        self._routes = {}

    ...

class FPNode(object):
    """A node in an FP tree."""

    def __init__(self, tree, item, count=1):
        self._tree = tree
        self._item = item
        self._count = count
        self._parent = None
        self._children = {}
        self._neighbor = None

    ...
```

使用穷举搜索的基线方法实现如下，枚举所有项集，在主函数中判断是否满足最小支持度（即是否为频繁项集）。

```
def exhaustive(data):
    maxItemLength = max([len(each) for each in data])
    allItems = []
    for i in data:
        for j in i:
            if not j in allItems:
                allItems.append(j)
    dic = {}
    for i in range(maxItemLength):
        nr = i + 1
        for each in combinations(allItems, nr)[0]:
            each = tuple(each)
            if not each in dic.keys():
```

```
                dic[each] = 1
            else:
                dic[each] += 1
    return dic
```

除此之外，还有返回列表L上大小为k的所有子集的集合的函数combinations。

```
def combinations(L, k):
    n = len(L)
    result = []
    for i in range(n - k + 1):
        if k > 1:
            newL = L[i + 1:]
            Comb, _ = combinations(newL, k - 1)
            for item in Comb:
                item.insert(0, L[i])
                result.append(item)
        else:
            result.append([L[i]])
    return result, len(result)
```

实验标准

1. 通过获取频繁项集数组长度（或迭代器遍历后符合支持度要求的项集数），来得到频繁项集数。
2. 通过memory_profiler库监控程序运行过程中的内存消耗，绘制内存占用大小的变化图，可根据时间定位到各算法占用的内存大小。
3. 通过记录调用不同算法函数前后的时间，并计算时间差得到时间开销。

结果和讨论

尝试了多组min_supp和min_conf，实验结果如下。使用穷举搜索的基线方法由于实在是过于朴素，导致在较长时间内也无法完成运行，故无记录。另外在UNIX_usage数据集的挖掘中，Apriori算法和FP-Growth算法出现了频繁项集数不同的情况，尚未想到解决方案。

频繁项集数（GroceryStore）	Apriori算法	FP-Growth算法	使用穷举搜索的基线方法
min_supp=0.2，min_conf=0.2	1	1	——
min_supp=0.2，min_conf=0.1	1	1	——
min_supp=0.1，min_conf=0.2	8	8	——
min_supp=0.1，min_conf=0.1	8	8	——
min_supp=0.1，min_conf=0.05	8	8	——
min_supp=0.05，min_conf=0.1	31	31	——
min_supp=0.05，min_conf=0.05	31	31	——
min_supp=0.05，min_conf=0.01	31	31	——
min_supp=0.01，min_conf=0.05	333	333	——
min_supp=0.01，min_conf=0.01	333	333	——

频繁项集数（UNIX_usage）	Apriori算法	FP-Growth算法	使用穷举搜索的基线方法
min_supp=0.2，min_conf=0.2	4	4	——
min_supp=0.2，min_conf=0.1	4	4	——
min_supp=0.1，min_conf=0.2	17	20	——
min_supp=0.1，min_conf=0.1	17	20	——
min_supp=0.1，min_conf=0.05	17	20	——
min_supp=0.05，min_conf=0.1	71	98	——
min_supp=0.05，min_conf=0.05	71	98	——
min_supp=0.05，min_conf=0.01	71	98	——
min_supp=0.01，min_conf=0.05	3230	4421	——
min_supp=0.01，min_conf=0.01	3230	4421	——

运行时间（GroceryStore）	Apriori算法	FP-Growth算法	使用穷举搜索的基线方法
min_supp=0.2，min_conf=0.2	0.272027s	0.31916s	——
min_supp=0.2，min_conf=0.1	0.329116s	0.030915s	——
min_supp=0.1，min_conf=0.2	0.344855s	0.048511s	——
min_supp=0.1，min_conf=0.1	0.401636s	0.060389s	——
min_supp=0.1，min_conf=0.05	0.365755s	0.047875s	——
min_supp=0.05，min_conf=0.1	0.877172s	0.425229s	——
min_supp=0.05，min_conf=0.05	1.011281s	0.426738s	——
min_supp=0.05，min_conf=0.01	1.405667s	0.566369s	——
min_supp=0.01，min_conf=0.05	11.355311s	1.531197s	——
min_supp=0.01，min_conf=0.01	11.828934s	1.445004s	——

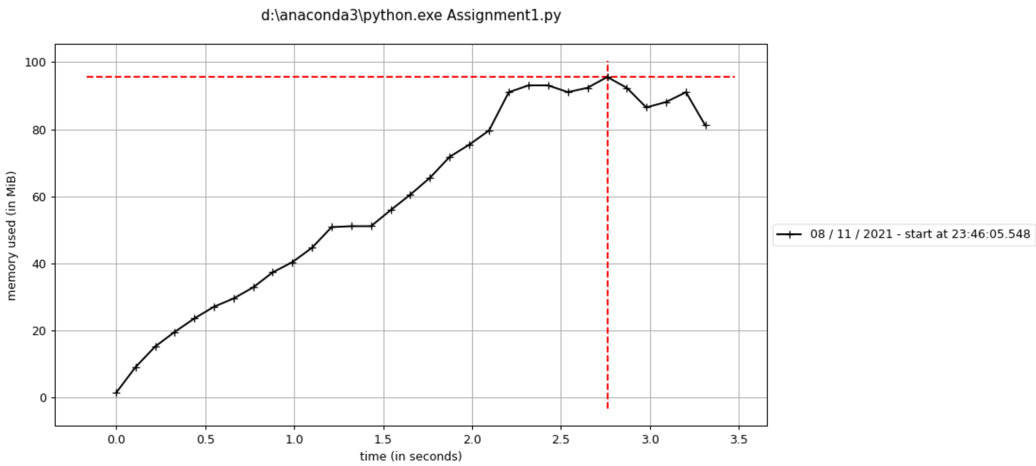
运行时间（UNIX_usage）	Apriori算法	FP-Growth算法	使用穷举搜索的基线方法
min_supp=0.2，min_conf=0.2	14.863954s	0.07384s	——
min_supp=0.2，min_conf=0.1	13.376523s	0.057846s	——
min_supp=0.1，min_conf=0.2	12.975545s	0.069812s	——
min_supp=0.1，min_conf=0.1	13.017332s	0.08178s	——
min_supp=0.1，min_conf=0.05	12.788737s	0.098737s	——
min_supp=0.05，min_conf=0.1	13.491375s	0.347998s	——
min_supp=0.05，min_conf=0.05	15.085267s	0.354785s	——
min_supp=0.05，min_conf=0.01	12.806447s	0.255535s	——
min_supp=0.01，min_conf=0.05	58.2089s	3.396795s	——
min_supp=0.01，min_conf=0.01	59.536372s	3.464092s	——

分析可发现，表格中设置的min_conf项并未起到区分的作用，观察各频繁项集的置信度发现，其均在0.2以上，将min_conf设为才有效果。

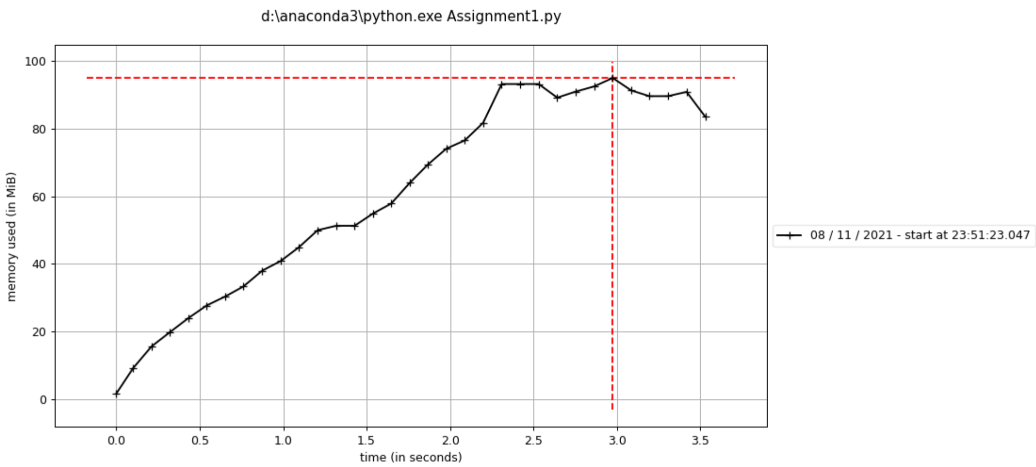
内存占用如下，其中前段高峰为Apriori算法，后段高峰为FP-Growth算法。由于min_conf实际并未起到区分作用，故内存只放min_supp不同时的截图。

GroceryStore数据集：

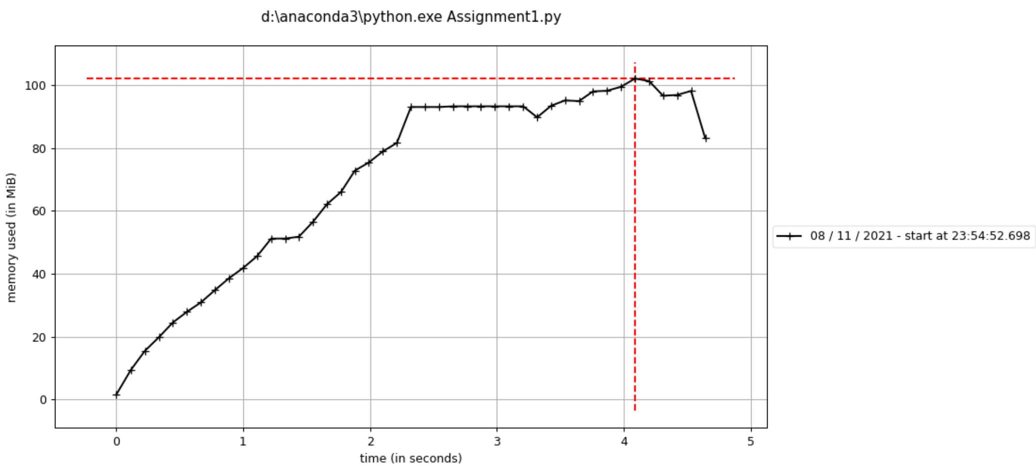
min_supp=0.2：



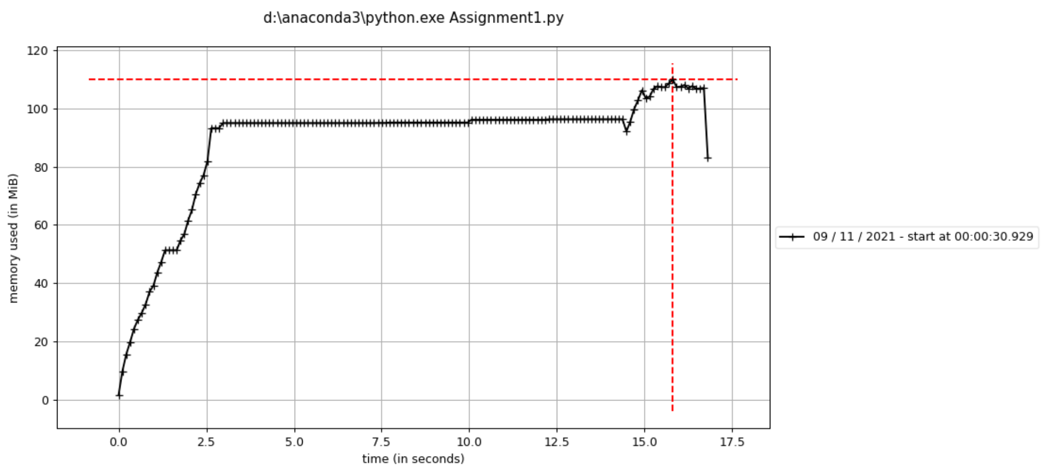
min_supp=0.1：



min_supp=0.05：

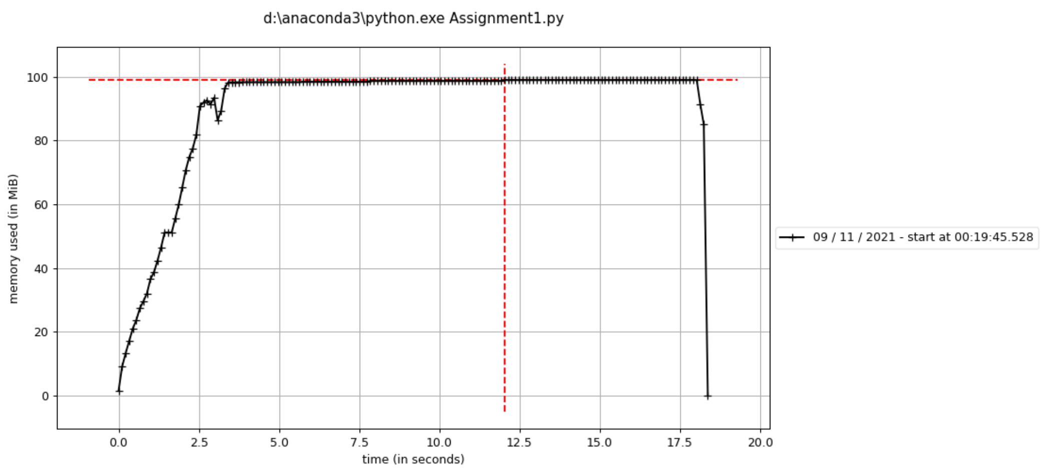


min_supp=0.01：

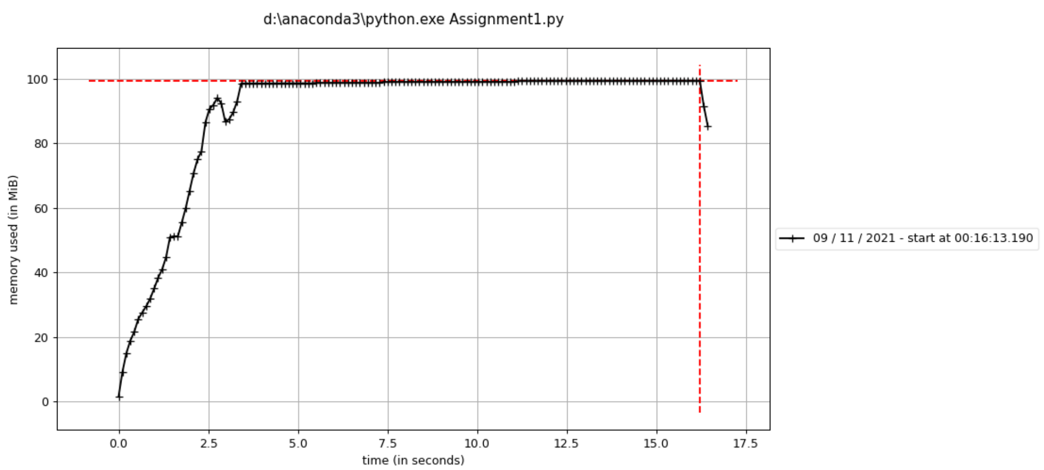


UNIX_usage数据集：

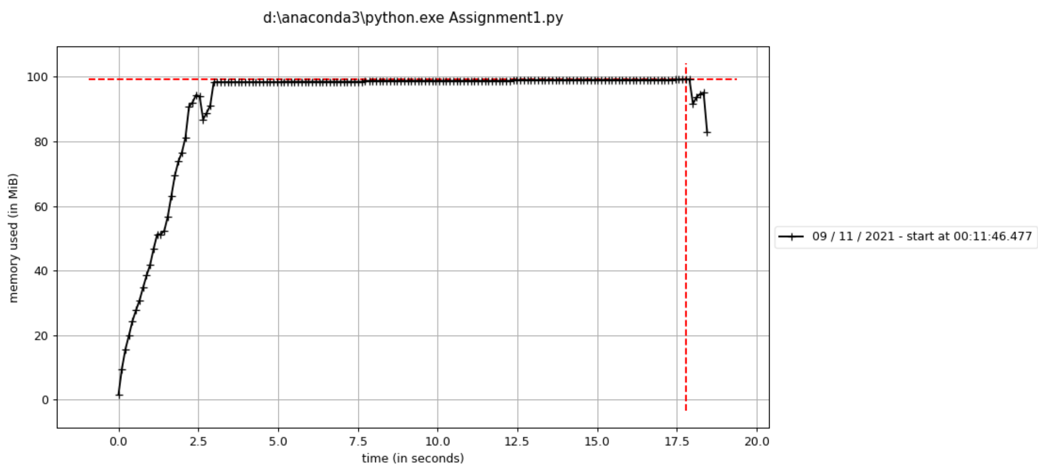
min_supp=0.2：



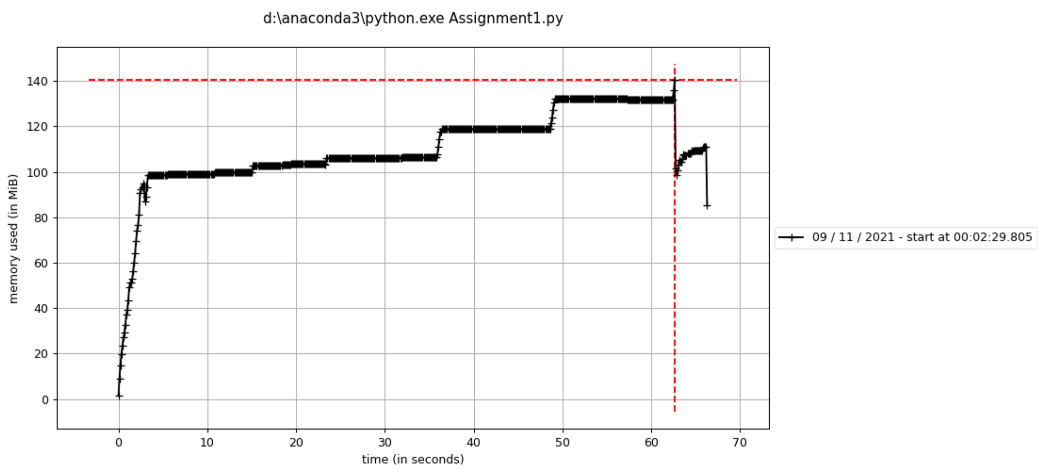
min_supp=0.1：



min_supp=0.05：



min_supp=0.01:



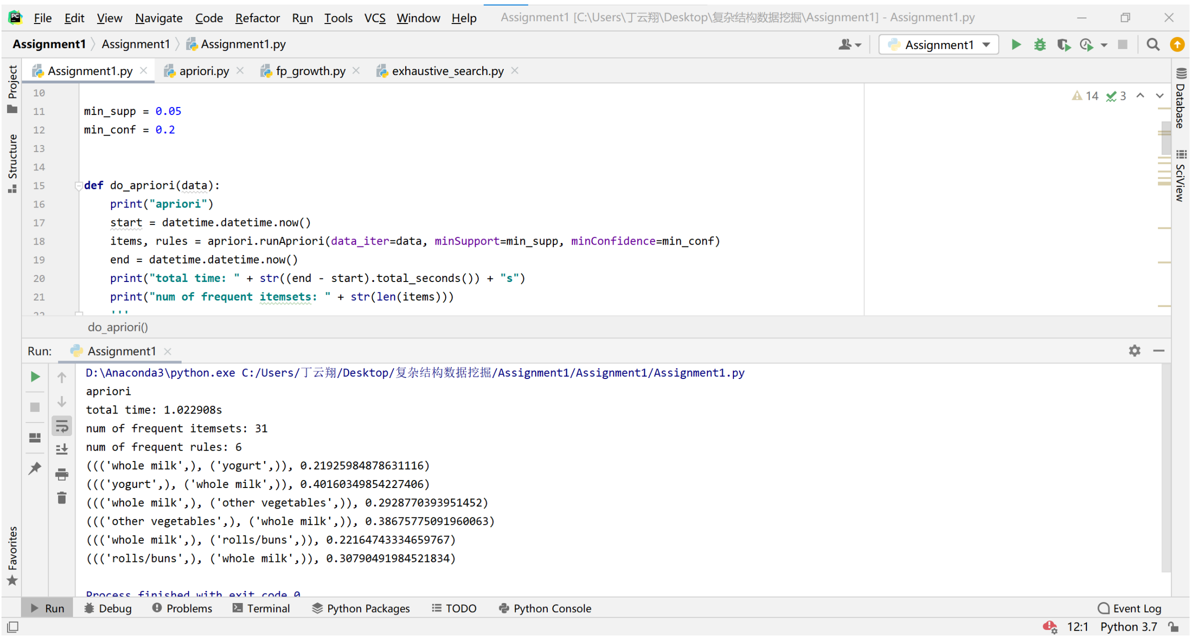
前段更长更低的为Apriori算法，后段更短更高的为FP-Growth算法，可知FP-Growth算法比Apriori算法内存占用更大（因为使用了FPTree结构），且min_supp越小，二者内存占用相差越大。

综合分析可以发现，FP-Growth算法相较于Apriori算法，消耗的内存更大，但是花费的时间大大减少，原因在于FPTree的构建占用了更多内存，但是省去了多次扫描数据列表的时间。

结论

GroceryStore数据集

运行截图



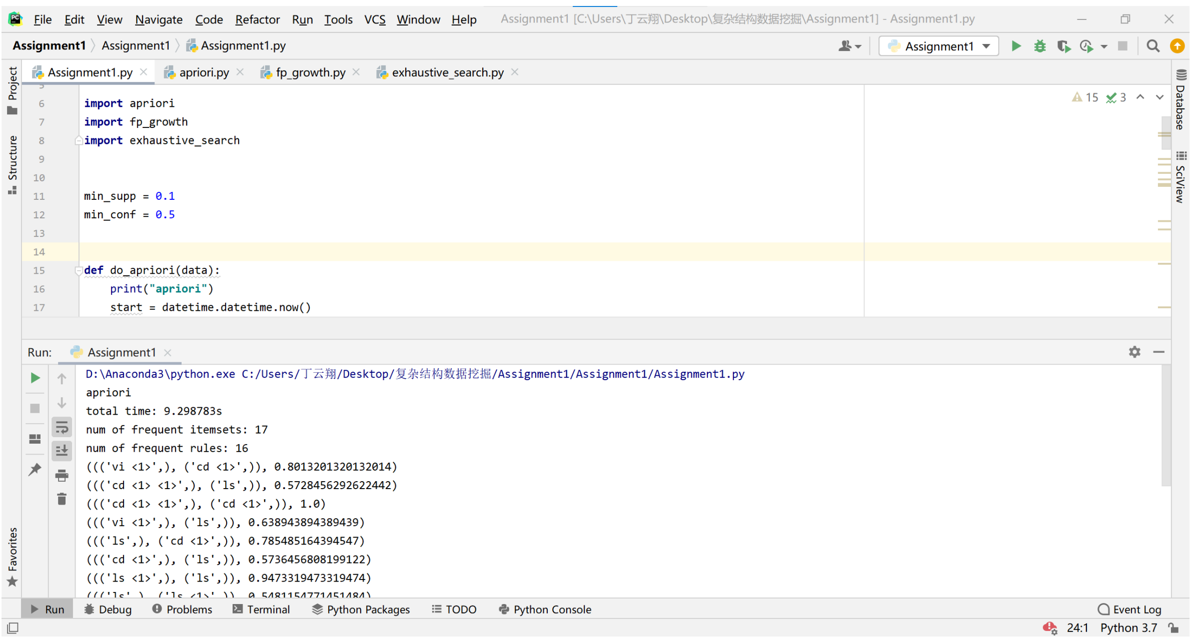
基于min_supp = 0.05、min_conf = 0.2，挖掘到规则如下：

```
((('whole milk',), ('yogurt',)), 0.21925984878631116)
((('yogurt',), ('whole milk',)), 0.40160349854227406)
((('whole milk',), ('other vegetables',)), 0.2928770393951452)
((('other vegetables',), ('whole milk',)), 0.38675775091960063)
((('whole milk',), ('rolls/buns',)), 0.22164743334659767)
((('rolls/buns',), ('whole milk',)), 0.30790491984521834)
```

分析挖掘出的规则可以得出：全脂奶与酸奶、全脂奶与蔬菜、全脂奶与面包卷相伴出现的概率较高，说明顾客很可能会同时买这些组合的商品，将这些商品的货架摆放的近一些有利于利用这些规律提高营业额。

UNIX_usage数据集

运行截图



基于min_supp = 0.1、min_conf = 0.5，挖掘到规则和置信度如下：

```
((('vi <1>',), ('ls',)), 0.638943894389439)
```

```
((('ls <1>',), ('cd <1>',)), 0.8828828828828831)
((('ls <1>',), ('ls',)), 0.9473319473319474)
((('ls',), ('ls <1>',)), 0.5481154771451484)
((('cd <1> <1>',), ('ls',)), 0.5728456292622442)
((('vi <1>',), ('cd <1>',)), 0.8013201320132014)
((('cd <1> <1>',), ('cd <1>',)), 1.0)
((('cd <1>',), ('ls',)), 0.5736456808199122)
((('ls',), ('cd <1>',)), 0.785485164394547)
((('cd <1> <1>',), ('ls', 'cd <1>')), 0.5728456292622442)
((('cd <1>', 'cd <1> <1>'), ('ls',)), 0.5728456292622442)
((('ls', 'cd <1> <1>'), ('cd <1>',)), 1.0)
((('ls <1>',), ('ls', 'cd <1>')), 0.8447678447678448)
((('ls <1>', 'cd <1>'), ('ls',)), 0.956828885400314)
((('ls <1>', 'ls'), ('cd <1>',)), 0.8917337234820776)
((('ls', 'cd <1>'), ('ls <1>',)), 0.6222562531904033)
```

分析挖掘出的规则可以得出：cd、ls和vi相伴出现的概率较高，这也符合我们的使用习惯，使用cd命令进入某个文件夹，然后使用ls查看该文件夹下的文件或使用vi编辑该文件夹下的某个文件；使用vi编辑完后使用ls查看文件夹下的文件或使用cd进入其他文件夹；等等。