Data Mining - ID2222 15 November 2017

Homework assignment 2

Author: Vendramin Nicolò

The task has been fulfilled with a python script made up of the following functions:

A. Function - “contains”:

```python
def contains(itemSet, basket, singletons=False):
    """
    This function defines wether itemSet is contained in basket. If singleton is true
    it means that itemSet only contains one element.
    """
    if singletons:
        return itemSet in set(basket)

    else:
        if len(itemSet) > len(basket):
            return False

        return set(itemSet) <= set(basket)
```

This function is an utility function used to be able to verify the containment of a set of elements or a single element in a given basket. If the element is single we call the function with the singletons parameter at True. The result is a bool indicating wether the containment is confirmed or not.

B. Function - “count_support”:

```python
def count_support(itemSets, baskets, singletons=False):
    """
    This function verifies the support of all the itemSet in itemSets in the baskets.
    If singleton is true it means that itemSet is composed by a single value.
    """
    itemSetsLen = len(itemSets)
    counts = [0 for i in range(0, itemSetsLen)]

    for itemSetIndex in range(0, itemSetsLen):
        itemSet = itemSets[itemSetIndex]

        for basket in baskets:
            if contains(itemSet, basket, singletons):
                counts[itemSetIndex] += 1

    return counts, itemSetsLen
```

This function goes through to a set of itemSets and computes for each of them their support in the baskets. For each itemSet we go through the complete set of baskets and count in how many of the baskets the itemSet is included.
This function is pretty expensive from the computational point of view. As a matter of fact we have to run a double for loop that results in n x m iterations where n is the number of itemSets of which we want to check the support and m is the number of baskets included in the dataset.
The function returns a list including in order the support of each one of the itemSets passed as a parameter, and the length of the itemSets (number of itemSet whose support has been computed).

C. Function - “filter_by_support”:

```python
def filter_by_support(itemSets, baskets, threshold=0, singletons=False):
    """
    Filters the itemSet in itemSets according to the given threshold of miniumum
    required support.
    """
    supports, length = count_support(itemSets, baskets, singletons)
    filtered = []
    counts = []

    for i in range(0, length):
        sup = supports[i]
        if sup >= threshold:
            filtered.append(itemSets[i])
            counts.append(sup)

    return filtered, counts, len(counts)
```

This function simply receives as input a list of itemSets, the dataset, the threshold that we want to use to filter on the basis of the support and a flag to indicate wether the itemSets are singletons or lists. At first the function calls the “count_support” function on the given parameters and afterwards only returns the list of those itemSets whose support resulted bigger or equal than the provided threshold. It also returns the list of the supports of the kept itemSets and their number.

D. Function - “generate_candidates”:

```python
def generate_candidates(itemSets, singletons, size):
    """
    This function is used in the apriori algorithm to generate new possible itemSets starting from
    the ones of the previous step and the filtered singletons of the dataset.
    """
    candidates = []
    singletons_len = len(singletons)

    print("Generating candidates of size {}.".format(size+1))
    if size == 1:
        for i in range(0, singletons_len-1):
            for j in range(i+1, singletons_len):
                candidates.append([singletons[i], singletons[j]])
    else:
        for itemSet in itemSets:
            max_ = max(itemSet)
            for singleton in singletons:
                if singleton > max_:
                    new = list(itemSet)
                    new.append(singleton)
                    flag = True
                    for subset in itertools.combinations(new, size):
                        if list(subset) not in itemSets:
                            flag = False
                            break

                    if flag:
                        candidates.append(new)

    return candidates
```

The generate_candidates function takes as input a set of itemSets, the singletons that are present in the dataset with at least the required support and the size of the itemSets to combine.
The procedure is slightly different for size=1 or other sizes. In the first case it just combines each singleton with all the following ones building only the top part of the diagonal matrix of the combinations between all the singletons. In case the size of the itemsets to combine is bigger than one, the routine goes through all the itemsets and for each one of them attaches all the singletons that are bigger than the bigger element in the items (to avoid to generate multiple times the same candidate, e.g: [1,2,3] and [1,3,2]). Of all of this temporary generated candidates

we keep only those ones for which all the possible subsets of length size are included in the itemsets provided in input (if the support of a subset is not at least the one required then the support of the itemset would surely be not enough).
Finally the function returns the candidates that have been found.

E. Function - “map_all_items”:

```python
def map_all_items(baskets):
    """
    This function extracts the singletons and maps them in a dictionary.
    """
    uniqueItems = set()
    for basket in baskets:
        uniqueItems.update(basket)

    uniqueItems = list(uniqueItems)
    lenUniqueItems = len(uniqueItems)
    uniqueItemsDictionary = {uniqueItems[i]: i for i in range(0, lenUniqueItems)}

    return uniqueItems, uniqueItemsDictionary, lenUniqueItems
```

This function simply extracts some basics information about the dataset to start the mining phase. It goes through the dataset adding all the items contained in each of the baskets to a set. It returns the list of all the unique items, a dictionary mapping them to an int (useful in case they are not numeric) and finally the number of unique items.

F. Function - “apriori”:

```python
def apriori(baskets, threshold=1):
    """
    Implementation of the apriori algorithm to find the frequent itemsets given a certain
    support threshold.
    """
    uniqueItems, uniqueItemsDictionary, lenUniqueItems = map_all_items(baskets)
    filtered_singletons, supports, lenSingletons = filter_by_support(uniqueItems, baskets, threshold, singletons=True)

    itemSets = []
    itemSets.append([[i] for i in filtered_singletons])
    support_dictionary = {}

    for i in range(0, lenSingletons):
        support_dictionary[str(itemSets[0][i])] = supports[i]

    k = 1

    while(True):
        candidates = generate_candidates(itemSets[k-1], filtered_singletons, k)
        num__ = len(candidates)
        candidates, candidate_support, num_ = filter_by_support(candidates, baskets, threshold)

        print("Generated {} candidates, {} remaining after the support filter.".format(num__, num_))

        if num_ > 0:
            itemSets.append(candidates)

            for i in range(0, num_):
                support_dictionary[str(candidates[i])] = candidate_support[i]

            k += 1
            print("Expanding the set with {} ItemSets of {} elements.".format(num_, k))

        else:
            print("Reached maximum expansion of ItemSets.")
            break

    return itemSets, support_dictionary
```

This function implements the a-priori algorithm for the discovery of frequent itemSets in a set of transactions (baskets). It receives as input the baskets in which we want to perform the analysis and the threshold representing the minimum support required to consider a given itemSet as frequent.

First of all the function extracts the items present in the baskets and filters them by support, obtaining the singletons that have at least the required support. After adding the singletons to the itemSets (they are frequent itemSets themselves) we initialise a dictionary that we will use to rapidly fetch the support of the frequent itemSets by using the string representation of the set as a key and its support as associated value.
We start a cycle that at each iteration increases the size of the itemSets that we try to generate, and will only break when no frequent itemSets of a certain size can be generated.
At each iteration the generate_candidates function will be called on the itemSets generated at the previous iteration (at iteration 0, the singletons) to produce candidate sets of frequent itemSets that we filter calling the filter_by_support function on them.
If at least one candidate remains after the filter we proceed to update the dictionary containing the supports of the itemSets, otherwise it means that no bigger itemSets can be produced (if no itemSet with support k can be generated of size i, surely there not exist any with support k of size i+1) and the routine is interrupted.
The function eventually returns the itemSets and the dictionary of supports.

G. Function - “generate_rules”:

```python
def generate_rules(frequentItemSets, supports, baskets_len, confidenceThreshold=0.5, interestThreshold=0.3):
    """
    This function returns the rules with a certain confidence and the the ones with a certain interest
    finding them as subsets of frequent itemsets.
    """
    max_set_size = len(frequentItemSets)

    rules = []
    interesting_rules = []

    for size in range(1, max_set_size):
        itemSets = frequentItemSets[size]
        for itemSet in itemSets:
            itemSet_support = supports[str(itemSet)]
            for item in itemSet:
                set_ = list(itemSet)
                set_.remove(item)
                rule = [set_, item]
                confidence = itemSet_support / supports[str(set_)]
                if confidence >= confidenceThreshold:
                    interest = math.fabs(confidence - (supports[str([item])] / baskets_len))
                    cand = (rule, confidence, interest, itemSet_support)
                    rules.append(cand)
                    if interest >= interestThreshold:
                        interesting_rules.append(cand)

    return rules, interesting_rules
```

This function has been written to solve the bonus task. This function takes as input a list of frequent itemSets (actually a list of lists where at position n all frequent itemSets of size n+1 are stored as lists), the dictionary containing their supports, the length of the basket and the threshold for confidence and interest used to respectively decide wether to keep a rule and to consider it interesting.
Going through all the different itemSets of size at least two we try to extract from each set all the possible rules. Doing the rule extraction only on the frequent itemSets we are already guaranteed that the rule will have at least the minimum required support. For each element in the itemSet a we try to generate the rule (A-{e}) -> {e}, we compute the confidence of the rule, and if it’s more than the required threshold we append it to the generated rules, we compute its interest and basing on the comparison of the latter with the interest threshold we append the rule to the interesting rules or not.
Finally the list of rules R and interesting rules I are returned (I is always subset or equal to R).

H. Function - “discover_association_rules”:

```python
def discover_association_rules(baskets, supportThreshold=1, confidenceThreshold=0.5, interestThreshold=0.3):
    """
    This function simply pipelines the function calls to directly find rules.
    """
    itemSets, support_dictionary = apriori(baskets, supportThreshold)
    return generate_rules(itemSets, support_dictionary, len(baskets), confidenceThreshold, interestThreshold)
```

This utility function simply calls the a-priori algorithm on the baskets with the given threshold and feeds the generate_rules function with the result of that prior phase.

I. Functions - “print_rule” & “print_itemset”:

```python
def print_rule(rule_tuple):
    """
    Just a function to print in a formatted style the rule tuple
    """
    print("[{} -> {}], with confidence of {}, interest of {} and support of {}."
        .format(rule_tuple[0][0], rule_tuple[0][1], rule_tuple[1], rule_tuple[2], rule_tuple[3]))


def print_itemset(itemSets, supports):
    """
    Just a function to print in a formatted style the rule tuple
    """
    for layer in itemSets:
        for itemSet in layer:
            print("[{}], with support of {}."
            .format(itemSet, supports[str(itemSet)]))
```

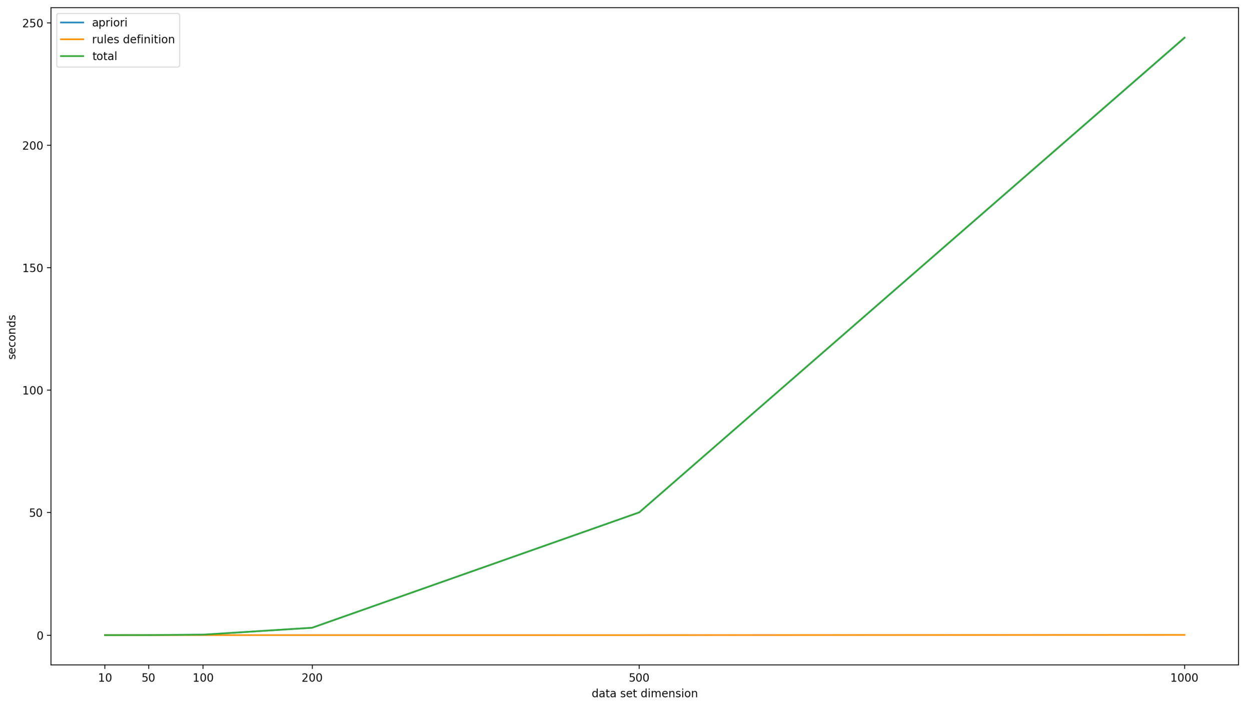
This utility functions simply are defined to print a a rule or an itemSets, along with their related information with a standard formatting for better visualisation.

J. Function - “main”:

```python
def main():

    f = open(args.file_path, "rU")
    words = f.readline()
    baskets = []
    while(len(words)>0):
        baskets.append([int(word) for word in words.split(" ")[:-1]])
        words = f.readline()
    f.close()
    baskets_len = len(baskets)

    if args.rules == "True":
        rules, interesting_rules = discover_association_rules(baskets, args.support, args.confidence, args.interest)
        print("{} rules found. {} of them are kept after the interest filtering.".format(len(rules), len(interesting_rules)))
        if args.verbose == "True":
            for rule in interesting_rules:
                print_rule(rule)

    else:
        itemSets, supports = apriori(baskets, 2)
        print("{} itemSets found.".format(len(supports)))
        if args.verbose == "True":
            print_itemset(itemSets, supports)
```

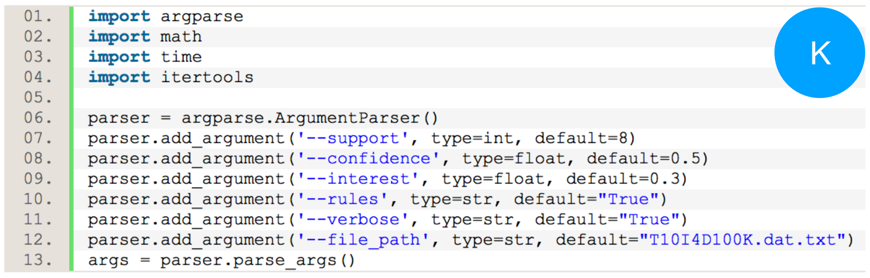
The main function of the program simply parses the datasets containing the baskets and basing on the argument passed to the program it finds the association rules or the itemSets.
The dataset that has been used for this lab contains one hundred thousands baskets and has been fetched at the link provided in the homework description.
After reading the text file containing the baskets we store them in a list of list. Each element of the list is a basket and each element of each basket is an item that was present in that given transaction.
Due to combinatorial complexity of the algorithm to generate and filter pairs, triplets etc. the execution time grows extremely fast with the size of the input as we can see in the graph reported in the next page.

We can also notice that the rules generation phase has almost a constant and really low contribution while the principal components of the computational effort is provided by the



frequent itemSet identification through the a-priori algorithm (total and apriori lines are overlapping in the plot).

K. Imports, arguments and usage:



```
import argparse
import math
import time
import itertools

parser = argparse.ArgumentParser()
parser.add_argument('--support', type=int, default=8)
parser.add_argument('--confidence', type=float, default=0.5)
parser.add_argument('--interest', type=float, default=0.3)
parser.add_argument('--rules', type=str, default="True")
parser.add_argument('--verbose', type=str, default="True")
parser.add_argument('--file_path', type=str, default="T10I4D100K.dat.txt")
args = parser.parse_args()
```

The only packages required to run the code are argparse, math, time and itertools. Apart from argparse they already form part of the standard libraries provided with python. To run the code is just necessary to set up an environment with Python 3.5, install the argparse external library and open a console in the folder of the project and run the command:

"python rules_mining.py"

With the following argument options:

- —support: is an integer value expressing the minimum required support to consider an element or an itemSet as frequent. Default value is 8.
- —confidence: is a float value between 0 and 1 that sets the threshold for the minimum confidence required to consider an association as a rule.
- —interest: is a float value between 0 and 1 that sets the threshold for the minimum interest required to consider a rule as an interesting rule.

- —rules: is a string value that can be either “True” or any other string. In case it’s true the rule mining phase is done, otherwise not. Default value is “True”.
- —verbose: is a string value that be either “True” or any other string. In case it’s true the rules (or the itemSets depending on wether we run it with —rules “True” or not) are explicitly printed otherwise only their count is printed. Default value is “True”.
- —file_path: is a string including the relative path from the directory of the project to the file containing the transactions that we want to mine. Default is “T10I4D100K.dat.txt”.