CSE 469: Assignment 2
Principle Association Analysis

Peter M. VanNostrand

10/12/2019

ASSIGNMENT 2: ASSOCIATION ANALYSIS

Frequent Itemsets – Gene Dataset

L1 Frequent Itemsets

```
{gene_1}:  sup = 0.83
{gene_12}:  sup = 0.54
{gene_14}:  sup = 0.52
{gene_17}:  sup = 0.55
{gene_21}:  sup = 0.62
{gene_22}:  sup = 0.55
{gene_23}:  sup = 0.54
{gene_25}:  sup = 0.57
{gene_26}:  sup = 0.52
{gene_27}:  sup = 0.51
{gene_3}:   sup = 0.71
{gene_31}:  sup = 0.51
{gene_36}:  sup = 0.61
{gene_37}:  sup = 0.56
{gene_39}:  sup = 0.51
{gene_4}:   sup = 0.5
{gene_43}:  sup = 0.5
{gene_45}:  sup = 0.58
{gene_47}:  sup = 0.66
{gene_48}:  sup = 0.57
{gene_5}:   sup = 0.73
{gene_50}:  sup = 0.5
{gene_53}:  sup = 0.5
{gene_54}:  sup = 0.67
{gene_55}:  sup = 0.55
{gene_56}:  sup = 0.51
{gene_59}:  sup = 0.76
{gene_6}:   sup = 0.66
{gene_60}:  sup = 0.54
{gene_63}:  sup = 0.5
{gene_64}:  sup = 0.5
{gene_66}:  sup = 0.59
{gene_67}:  sup = 0.62
{gene_71}:  sup = 0.58
{gene_72}:  sup = 0.74
{gene_75}:  sup = 0.57
{gene_77}:  sup = 0.58
{gene_78}:  sup = 0.59
{gene_8}:   sup = 0.66
{gene_81}:  sup = 0.58
{gene_83}:  sup = 0.5
{gene_84}:  sup = 0.54
{gene_87}:  sup = 0.67
{gene_89}:  sup = 0.59
{gene_9}:   sup = 0.5
{gene_90}:  sup = 0.52
{gene_91}:  sup = 0.65
{gene_93}:  sup = 0.53
{gene_94}:  sup = 0.62
{gene_98}:  sup = 0.51
{gene_99}:  sup = 0.56
```

L2 Frequent Itemsets

```
{gene_5, gene_72}:  sup = 0.51
{gene_59, gene_5}:  sup = 0.51
{gene_1, gene_59}:  sup = 0.62
{gene_47, gene_3}:  sup = 0.5
{gene_3, gene_72}:  sup = 0.53
{gene_47, gene_5}:  sup = 0.53
{gene_1, gene_67}:  sup = 0.55
{gene_1, gene_3}:   sup = 0.63
{gene_1, gene_8}:   sup = 0.53
{gene_1, gene_84}:  sup = 0.5
{gene_81, gene_1}:  sup = 0.51
{gene_87, gene_59}: sup = 0.51
{gene_1, gene_6}:   sup = 0.59
{gene_1, gene_89}:  sup = 0.52
{gene_1, gene_72}:  sup = 0.61
{gene_87, gene_1}:  sup = 0.56
{gene_1, gene_21}:  sup = 0.53
{gene_87, gene_5}:  sup = 0.51
{gene_91, gene_5}:  sup = 0.5
{gene_1, gene_94}:  sup = 0.54
{gene_59, gene_3}:  sup = 0.56
{gene_1, gene_54}:  sup = 0.58
{gene_91, gene_1}:  sup = 0.55
{gene_1, gene_47}:  sup = 0.59
{gene_5, gene_3}:   sup = 0.59
{gene_59, gene_6}:  sup = 0.51
{gene_1, gene_5}:   sup = 0.65
{gene_6, gene_5}:   sup = 0.52
{gene_59, gene_72}: sup = 0.62
```

L3 Frequent Itemsets

```
{gene_1, gene_72, gene_59}: sup = 0.5
{gene_1, gene_5, gene_3}:   sup = 0.52
```

Length 3 Candidate Itemsets – Gene Dataset

```
{'gene_72', 'gene_3', 'gene_5'}
{'gene_47', 'gene_3', 'gene_5'}
{'gene_59', 'gene_5', 'gene_1'}
{'gene_1', 'gene_91', 'gene_5'}
{'gene_59', 'gene_5', 'gene_6'}
{'gene_1', 'gene_3', 'gene_5'}
{'gene_59', 'gene_5', 'gene_72'}
{'gene_87', 'gene_59', 'gene_1'}
{'gene_87', 'gene_1', 'gene_5'}
{'gene_72', 'gene_3', 'gene_1'}
{'gene_47', 'gene_3', 'gene_1'}
{'gene_72', 'gene_1', 'gene_5'}
{'gene_59', 'gene_1', 'gene_6'}
{'gene_72', 'gene_59', 'gene_1'}
{'gene_59', 'gene_3', 'gene_5'}
{'gene_1', 'gene_47', 'gene_5'}
{'gene_59', 'gene_3', 'gene_1'}
{'gene_1', 'gene_5', 'gene_6'}
{'gene_59', 'gene_3', 'gene_72'}
{'gene_59', 'gene_87', 'gene_5'}
```

Apriori Algorithm Code

Below are the portions of code which I wrote to fill in the template functions provided. These have been nicely formatted to fit in this document. A full copy paste of the code is available in the appendix, read at your own risk.

Apriori Gen

```
def apriori_gen(freq_sets, k):
    n = len(freq_sets)
    if n<2: # Minimum 2 frequent itemsets needed to generate candidates
        return []

    # generate all possible candidate itemsets
    candidate_set = set()
    for i in range(0, n-1): # iterate through each element
        for j in range(i+1, n): # and try to combine it with every element after it
            commonElems = freq_sets[i].intersection(freq_sets[j])
            if len(commonElems) >= k-2: # if k-2 of the items in the sets match
                # combine the sets to make a length k itemset
                newCandidate = freq_sets[i].union(freq_sets[j])
                candidate_set.add(newCandidate) # add that itemset to the list of candidates

    # find candidate itemsets which have k-1 length subsets that are infrequent
    invalidCandidates = set()
    for candidate in candidate_set:
        for elem in candidate:
            k1subset = candidate.difference({elem}) # generate every possible k-1 subset
            subsetIsFrequent = False
            for freqItemset in freq_sets: # check that every k-1 subset is frequent
                if k1subset == freqItemset:
                    subsetIsFrequent = True
                    break
            if not subsetIsFrequent: # if one or more of the k-1 subsets was infrequent
                invalidCandidates.add(candidate) # prune the candidate itemset

    # prune the invalid candidates
    for candidate in invalidCandidates:
        candidate_set.remove(candidate)

    return list(candidate_set)
```

Get Freq

```
def get_freq(dataset, candidates, min_support, verbose=False):
   freq_list = []
    support_data = dict()
    for candidateSet in candidates:
        supportCount = 0
        for transaction in dataset:
            if candidateSet.issubset(transaction):
                supportCount += 1
        support = supportCount / len(dataset)
        if support >= min_support:
            freq_list.append(candidateSet)
            support_data[candidateSet] = support

    return freq_list, support_data
```

Appendix

Full text of Apriori_template.py

```
from __future__ import print_function
import sys

def apriori(dataset, min_support=0.5, verbose=False):
    """Implements the Apriori algorithm.

    The Apriori algorithm will iteratively generate new candidate
    k-itemsets using the frequent (k-1)-itemsets found in the previous
    iteration.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate
        candidate itemsets.

    min_support : float
        The minimum support threshold. Defaults to 0.5.

    Returns
    -------
    F : list
        The list of frequent itemsets.

    support_data : dict
        The support data for all candidate itemsets.

    References
    ----------
    .. [1] R. Agrawal, R. Srikant, "Fast Algorithms for Mining Association
           Rules", 1994.

    """
    C1 = create_candidates(dataset)
    D = list(map(set, dataset))
    F1, support_data = get_freq(D, C1, min_support, verbose=False) # get frequent 1-itemsets
    F = [F1] # list of frequent itemsets; initialized to frequent 1-itemsets
    k = 2 # the itemset cardinality
    while (len(F[k - 2]) > 0):
        Ck = apriori_gen(F[k-2], k) # generate candidate itemsets
        Fk, supK  = get_freq(D, Ck, min_support) # get frequent itemsets
        support_data.update(supK)# update the support counts to reflect pruning
        F.append(Fk)  # add the frequent k-itemsets to the list of frequent itemsets
        k += 1

    if verbose:
        # Print a list of all the frequent itemsets.
        for kset in F:
            for item in kset:
                print("" + "{" + "".join(str(i) + ", " for i in
iter(item)).rstrip(', ') + "}" + ":  sup = " +
str(round(support_data[item], 3)))

    return F, support_data

def create_candidates(dataset, verbose=False):
    """Creates a list of candidate 1-itemsets from a list of transactions.

    Parameters
```

```
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate candidate
        itemsets.

    Returns
    -------
    The list of candidate itemsets (c1) passed as a frozenset (a set that is
    immutable and hashable).
    """
    c1 = [] # list of all items in the database of transactions
    for transaction in dataset:
        for item in transaction:
            if not [item] in c1:
                c1.append([item])
    c1.sort()

    if verbose:
        # Print a list of all the candidate items.
        print("" + "{" + "".join(str(i[0]) + ", " for i in iter(c1)).rstrip(', ')
+ "}")

    # Map c1 to a frozenset because it will be the key of a dictionary.
    return list(map(frozenset, c1))

def get_freq(dataset, candidates, min_support, verbose=False):
    """

    This function separates the candidates itemsets into frequent itemset and infrequent itemsets based on
the min_support,
        and returns all candidate itemsets that meet a minimum support threshold.

    Parameters
    ----------
    dataset : list
        The dataset (a list of transactions) from which to generate candidate
        itemsets.

    candidates : frozenset
        The list of candidate itemsets.

    min_support : float
        The minimum support threshold.

    Returns
    -------
    freq_list : list
        The list of frequent itemsets.

    support_data : dict
        The support data for all candidate itemsets.
    """
    freq_list = []
    support_data = dict()
    for candidateSet in candidates:
        supportCount = 0
        for transaction in dataset:
            if candidateSet.issubset(transaction):
                supportCount += 1
        support = supportCount / len(dataset)
        if support >= min_support:
            freq_list.append(candidateSet)
            support_data[candidateSet] = support
```

```
    return freq_list, support_data

def apriori_gen(freq_sets, k):
    """Generates candidate itemsets (via the F_k-1 x F_k-1 method).

    This part generates new candidate k-itemsets based on the frequent
    (k-1)-itemsets found in the previous iteration.

    The apriori_gen function performs two operations:
    (1) Generate length k candidate itemsets from length k-1 frequent itemsets
    (2) Prune candidate itemsets containing subsets of length k-1 that are infrequent

    Parameters
    ----------
    freq_sets : list
        The list of frequent (k-1)-itemsets.

    k : integer
        The cardinality of the current itemsets being evaluated.

    Returns
    -------
    candidate_list : list
        The list of candidate itemsets.
    """

    n = len(freq_sets)
    if n<2: # Minimum 2 frequent itemsets needed to generate candidates
        return []

    # generate all possible candidate itemsets
    candidate_set = set()
    for i in range(0, n-1): # iterate through each element
        for j in range(i+1, n): # and try to combine it with every element after it
            commonElems = freq_sets[i].intersection(freq_sets[j])
            if len(commonElems) >= k-2: # if k-2 of the items in the sets match
                newCandidate = freq_sets[i].union(freq_sets[j]) # combine the sets to make a length k
itemset
                candidate_set.add(newCandidate) # add that itemset to the list of candidates

    # find candidate itemsets which have k-1 length subsets that are infrequent
    invalidCandidates = set()
    for candidate in candidate_set:
        for elem in candidate:
            k1subset = candidate.difference({elem}) # generate every possible k-1 subset
            subsetIsFrequent = False
            for freqItemset in freq_sets: # check that every k-1 subset is frequent
                if k1subset == freqItemset:
                    subsetIsFrequent = True
                    break
            if not subsetIsFrequent: # if one or more of the k-1 subsets was infrequent
                invalidCandidates.add(candidate) # prune the candidate itemset

    # prune the invalid candidates
    for candidate in invalidCandidates:
        candidate_set.remove(candidate)

    return list(candidate_set)


def loadDataSet(fileName, delim=','):
    fr = open(fileName)
```

```
    stringArr = [line.strip().split(delim) for line in fr.readlines()]
    return stringArr



def run_apriori(data_path, min_support, verbose=False):
    dataset = loadDataSet(data_path)
    F, support = apriori(dataset, min_support=min_support, verbose=verbose)
    return F, support



def bool_transfer(input):
    ''' Transfer the input to boolean type'''
    input = str(input)
    if input.lower() in ['t', '1', 'true' ]:
        return True
    elif input.lower() in ['f', '0', 'false']:
        return False
    else:
        raise ValueError('Input must be one of {T, t, 1, True, true, F, F, 0, False, false}')




if __name__ == '__main__':
    if len(sys.argv)==3:
        F, support = run_apriori(sys.argv[1], float(sys.argv[2]))
    elif len(sys.argv)==4:
        F, support = run_apriori(sys.argv[1], float(sys.argv[2]), bool_transfer(sys.argv[3]))
    else:
        raise ValueError('Usage: python apriori_templete.py <data_path> <min_support> <is_verbose>')
    print(F)
    print(support)

    '''
    Example:

    python apriori_templete.py market_data_transaction.txt 0.5

    python apriori_templete.py market_data_transaction.txt 0.5 True

    '''
```