```
 1 # -*- coding: utf-8 -*-
 2 '''Created by Tai Dinh
 3 This file is used for k-CMM algorithm: clustering mixed
   numeric and categorical data with missing values
 4 '''
 5
 6 from __future__ import division
 7 import sys
 8 import numpy as np
 9 import evaluation
10 from collections import defaultdict
11 from scipy import *
12 import c45
13 import math
14 import matplotlib.pyplot as plt
15 import pandas as pd
16 import itertools
17 # For measuring the time for running program
18 # source: http://stackoverflow.com/a/1557906/6009280
19 # or https://www.w3resource.com/python-exercises/python-
   basic-exercise-57.php
20 # import atexit
21 from time import time, strftime, localtime
22 from datetime import timedelta
23
24 # For measuring the memory usage
25 import tracemalloc
26
27
28 def get_max_value_key(dic):
29     '''Fast method to get key for maximum value in dict'''
30     v = list(dic.values())
31     k = list(dic.keys())
32     return k[v.index(max(v))]
33
34 '''
35 This function is used to calculate the dissimilarity
   between 2 attributes x and y at the iattr(d)
36 using the global_attr_freq in which the global_axttr_freq[i
   ][x] is the frequency of the attribute at
37 i with value x in the whole samples:
38 dis(x, y) = 1 - 2 * log(P{x, y}) / (log(P{x}) + log(P{y}))
39 '''
40
41
42 def attr_dissim(x, y, iattr, global_attr_freq):
43     '''
44     Dissimilarity between 2 categorical attributes x and y
   at the attribute iattr, i.e
45     dis(x, y) = 1 - 2 * log(P{x, y}) / (log(P{x}) + log(P{y
```

```
45 }))
46     '''
47     if (global_attr_freq[iattr][x] == 1.0) and (
   global_attr_freq[iattr][y] == 1.0):
48         return 0
49     if x == y:
50         numerator = 2 * math.log(global_attr_freq[iattr][x]
   )
51     else:
52         numerator = 2 * math.log((global_attr_freq[iattr][x
   ] + global_attr_freq[iattr][y]))
53     denominator = math.log(global_attr_freq[iattr][x]) +
   math.log(global_attr_freq[iattr][y]) #Noted by Tai Dinh,
   Equation 21, page 124
54     return 1 - numerator / denominator
55 
56 '''information theoretic based similarity measure
57 This function is used to calculate the dissimilarity
   between a centroid and a vector a
58 '''
59 
60 def vector_matching_dissim(centroid, categorical, a,
   global_attr_freq):
61     # Get distance between a centroid and a
62     distance = 0.
63     attrIndex = 0
64     for ic, curc in enumerate(centroid):
65         if ic in categorical:
66             keys = curc.keys()
67             for key in keys:
68                 distance += curc[key] * attr_dissim(key, a[
   ic], attrIndex, global_attr_freq)
69             attrIndex += 1
70         else:
71             tmp = float(curc)-float(a[ic])
72             distance += pow(tmp,2)
73     return distance
74 
75 '''
76 This function is used to calculate the distances between
   centroid clusters and a data point, using the
   global_attr_freq.
77 global_axttr_freq[i][x] is the probability of the attribute
    at position i and value x on the whole samples.
78 categorical is the set of categorical attributes in the
   data point.
79 '''
80 
81 def vectors_matching_dissim(vectors, categorical, a,
   global_attr_freq):
```

```
 82      '''Get nearest vector in vectors to a'''
 83      min = np.Inf
 84      min_clust = -1
 85      for clust in range(len(vectors)):
 86          distance = vector_matching_dissim(vectors[clust],
    categorical, a, global_attr_freq)
 87          if distance < min:
 88              min = distance
 89              min_clust = clust
 90      return min_clust, min
 91
 92  '''
 93  This function is used to transfer a vector point from this
      cluster (from_clust) to another cluster (to_clust)
 94  ipoint is the index of vector in the samples.
 95  membership[cluster_index, ipoint] = 1 means vector with
    the index ipoint belongs to the cluster_index.
 96
 97  cl_attr_freq[cluster_index][iattr][curattr] is the
    frequency of the attribute having value curattr at iattr
    in cluster cluster_index.
 98  In fact, k are kept in the cl_attr_freq instead of k/N
    such that k is the number of appearance of attribute at
    the iattr with value curattr,
 99  N is the number of data objects in the cluster. The reason
      is k and N are probably change, in this case
    recalculating k/N is more complex than k.
100
101 Note that global_attr_freq stores frequency (k/N) because
    it only needs one time to calculate and values keep
    permanently.
102 '''
103
104 def move_point_between_clusters(point, ipoint, to_clust,
    from_clust,
105     cl_attr_freq, membership, categorical):
106     '''Move point between clusters, categorical attributes
    '''
107     membership[to_clust, ipoint] = 1
108     membership[from_clust, ipoint] = 0
109     # Update frequencies of attributes in clusters
110     attrIndex = 0
111     for iattr, curattr in enumerate(point):
112         if iattr in categorical:
113             cl_attr_freq[to_clust][attrIndex][curattr] +=
    1
114             cl_attr_freq[from_clust][attrIndex][curattr]
    -= 1
115             attrIndex +=1
116     return cl_attr_freq, membership
```

```
117
118 def matching_dissim(a, b):
119     '''Simple matching dissimilarity function'''
120     return np.sum(a != b, axis=1)
121
122 '''
123 Ramdomly initialize vectors in X into clusters
124 '''
125
126 def _init_clusters(X, centroids, n_clusters, nattrs,
    npoints, verbose, categorical):
127     # __INIT_CLUSTER__
128     # if verbose:
129     #     print("Init: Initalizing clusters")
130     membership = np.zeros((n_clusters, npoints), dtype='
    int64')
131     # cl_attr_freq is a list of lists with dictionaries
    that contain the
132     # frequencies of values per cluster and attribute.
133     cl_attr_freq = [[defaultdict(int) for i in categorical
    ]
134                     for _ in range(n_clusters)]
135     for ipoint, curpoint in enumerate(X):
136         # Initial assignment to clusters
137         clust = np.argmin(matching_dissim(centroids,
    curpoint))
138         membership[clust, ipoint] = 1
139         # Count attribute values per cluster
140         attrIndex = 0
141         for iattr, curattr in enumerate(curpoint):
142             if iattr in categorical:
143                 cl_attr_freq[clust][attrIndex][curattr] +=
     1
144                 attrIndex += 1
145
146     # Move random selected point from largest cluster to
    empty cluster if exists
147     for ik in range(n_clusters):
148         if sum(membership[ik, :]) == 0:
149             from_clust = membership.sum(axis=1).argmax()
150             choices = \
151                 [ii for ii, ch in enumerate(membership[
    from_clust, :]) if ch]
152             rindex = np.random.choice(choices)
153             # Move random selected point to empty cluster
154             cl_attr_freq, membership =
    move_point_between_clusters(
155                 X[rindex], rindex, ik, from_clust,
    cl_attr_freq, membership, categorical)
156
```

```
157     return cl_attr_freq, membership
158 
159 '''
160 This function is used to calculate the lambda as the
    formula in slide page 15
161 cl_attr_freq[iattr][curattr] is the probability of
    attribute at the iattr having value curattr in cluster
    clust
162 clust_members is the number of data objects in the cluster
163 '''
164 
165 def cal_lambda(cl_attr_freq, clust_members):
166     '''Re-calculate optimal bandwitch for each cluster'''
167     if clust_members <= 1:
168         return 0.
169 
170     numerator = 0.
171     denominator = 0.
172 
173     for iattr, curattr in enumerate(cl_attr_freq):
174         n_ = 0.
175         d_ = 0.
176         keys = curattr.keys()
177         for key in keys:
178             n_ += math.pow(1.0 * curattr[key] /
    clust_members,2)
179             d_ += math.pow(1.0 * curattr[key] /
    clust_members, 2)
180         numerator += (1 - n_)
181         denominator += (d_ - 1.0 / (len(keys)))
182 
183     # print denominator
184     # assert denominator != 0, "How can denominator equal
    to 0?"
185     if clust_members == 1 or denominator == 0:
186         return 0
187     result = (1.0 * numerator) / ((clust_members - 1) *
    denominator)
188     if result < 0:
189         return 0;
190     if result > 1:
191         return 1
192     return (1.0 * numerator) / ((clust_members - 1) *
    denominator)
193 
194 def _cal_global_attr_freq(X, npoints, nattrs, categorical)
    :
195     # global_attr_freq is a list of lists with
    dictionaries that contain the
196     # frequencies of attributes.
```

```
197     global_attr_freq = [defaultdict(float) for _ in
    categorical]
198     # global_attr_freq = {}
199     # for i in categorical:
200     #     global_attr_freq[i] = defaultdict(float)
201 
202     for ipoint, curpoint in enumerate(X):
203         attrIndex = 0
204         for iattr, curattr in enumerate(curpoint):
205             if iattr in categorical:
206                 global_attr_freq[attrIndex][curattr] += 1.
207                 attrIndex += 1
208     attrIndex = 0
209     for iattr in range(nattrs):
210         if iattr in categorical:
211             for key in global_attr_freq[attrIndex].keys():
212                 global_attr_freq[attrIndex][key] /=
    npoints
213             attrIndex += 1
214     return global_attr_freq
215 
216 '''
217 This function is used to calculate the centroid center at
    each attribute.
218 
219 * ldb is lambda
220 * cl_attr_freq_attr is cl_attr_freq[clust][iattr], is the
    number of attribute at the index iattr in the cluster
    clust.
221 * clust_members is the number of data objects in the
    cluster
222 * global_attr_count is the number of attribute at the
    index iattr in the whole dataset X.
223 '''
224 
225 def cal_centroid_value(lbd, cl_attr_freq_attr,
    cluster_members, attr_count):
226     '''Calculate centroid value at iattr'''
227     assert cluster_members >= 1, "Cluster has no member,
    why?"
228 
229     keys = cl_attr_freq_attr.keys()
230     vjd = defaultdict(float)
231     for odl in keys:
232         vjd[odl] = lbd / attr_count + (1 - lbd) * (1.0 *
    cl_attr_freq_attr[odl] / cluster_members) #Noted by Tai
    Dinh equation 12, page 121
233     return vjd
234 
235 def cal_mean_value(X, indexAttr):
```

```
236     # print(X.iloc[:,indexAttr])
237     meanValue = mean(np.asarray(X.iloc[:,indexAttr], dtype
    = float))
238     return meanValue
239 
240 '''
241 This function is the loop for the k-CMM algorithm
242 For each vector curpoint with the index ipoint in X, the
    purpose is to find the nearest centroid with this vector.
243 '''
244 
245 def _k_CMM_iter(X, categorical, centroids, cl_attr_freq,
    membership, global_attr_freq, lbd, use_global_attr_count):
246     '''Single iteration of the k_CMM clustering algorithm
    '''
247     moves = 0
248     for ipoint, curpoint in enumerate(X):
249         clust, distance = vectors_matching_dissim(
    centroids, categorical, curpoint, global_attr_freq)
250         if membership[clust, ipoint]:
251             # Sample is already in its right place
252             continue
253 
254         # Move point and update old/new cluster
    frequencies and centroids
255         '''
256         moves is the number of moving vectors between
    cluster
257         old_clust is the old index of vector curpoint
258         '''
259         moves += 1
260         old_clust = np.argwhere(membership[:, ipoint])[0][
    0]
261 
262         '''
263         Move vector with index ipoint from old_clust to
    clust, meanwhile recalculate the probability of attributes
     in the corresponding clusters.
264         '''
265         cl_attr_freq, membership =
    move_point_between_clusters(
266             curpoint, ipoint, clust, old_clust,
    cl_attr_freq, membership,categorical)
267 
268         # In case of an empty cluster, reinitialize with a
     random point
269         # from the largest cluster.
270         '''
271         After moving vectors from old_clust to new_clust,
    if the old_clust is empty,
```

```
272         then get an arbitrary vector from the largest
    cluster to this cluster to avoid empty clusters.
273         '''
274         if sum(membership[old_clust, :]) == 0:
275             from_clust = membership.sum(axis = 1).argmax()
276             choices = \
277                 [ii for ii, ch in enumerate(membership[
    from_clust, :]) if ch]
278             rindex = np.random.choice(choices)
279 
280             cl_attr_freq, membership =
    move_point_between_clusters(
281                 X[rindex], rindex, old_clust, from_clust,
    cl_attr_freq, membership,categorical)
282 
283         # Re-calculate lambda of changed centroid
284         for curc in (clust, old_clust):
285             lbd[curc] = cal_lambda(cl_attr_freq[curc], sum
    (membership[curc, :]))
286 
287         # Update new and old centroids by choosing mode of
     attribute.
288         attrIndex = 0
289         for iattr in range(len(curpoint)):
290             if iattr in categorical:
291                 for curc in (clust, old_clust):
292                     cluster_members = sum(membership[curc,
     :])
293                     if use_global_attr_count:
294                         centroids[curc][iattr] =
    cal_centroid_value(lbd[curc], cl_attr_freq[curc][attrIndex
    ], cluster_members, len(global_attr_freq[attrIndex]))
295                     else:
296                         attr_count = len(cl_attr_freq[curc
    ][attrIndex].keys())
297                         centroids[curc][iattr] =
    cal_centroid_value(lbd[curc], cl_attr_freq[curc][attrIndex
    ], cluster_members, attr_count)
298                 attrIndex += 1
299             else:
300                 comSetDF = pd.DataFrame(X)
301                 centroids[curc][iattr] = cal_mean_value(
    comSetDF,iattr)
302 
303     return centroids, moves, lbd
304 
305 
306 '''
307  This function is used to calculate the sum of distances
    between vectors inside X and centroids of clusters after
```

```
307 each step.
308  Labels is the label of vector in X, labels[x] = c means
    vector with index is x is belonged to the cluster that has
     its index is c.
309  Cost is the sum of dissimilarity.
310 '''
311
312 def _labels_cost(X, categorical, centroids,
    global_attr_freq):
313     '''
314     Calculate labels and cost function given a matrix of
    points and
315     a list of centroids for the k-CMM algorithm.
316     '''
317
318     npoints, nattrs = X.shape
319     cost = 0.
320     labels = np.empty(npoints, dtype = 'int64')
321     for ipoint, curpoint in enumerate(X):
322         '''
323         For every vector ipoint (its value is curpoint) in
     X, find out nearest cluster with it by using the function
     vectors_matching_dissim.
324         Then calculate the distance by using the
    dissimilarity for a mixed object and cluster)
325         '''
326         clust, diss = vectors_matching_dissim(centroids,
    categorical, curpoint, global_attr_freq)
327         assert clust != -1, "Why there is no cluster for
    me?"
328         labels[ipoint] = clust
329         cost += diss
330
331     return labels, cost
332
333 # This function is used to split numeric and categorical
    attributes in the complete dataset to two subsets.
334 def splitNumCat(x, categorical):
335     xNum = x.loc[:, [ii for ii in range(x.shape[1])
336                                     if ii not in categorical]]
337     xCat = x.loc[:, categorical]
338     return xNum, xCat
339
340 # This function is used to impute numeric attributes
341 def numericImputation(inSetDFNum, comSetDFNum, IDList):
342     inObjectNum = inSetDFNum.loc[0]
343     # listOfObjects = []
344     # for i in range(0, len(IDList)):
345     #     listOfObjects.append(comSetDFNum.iloc[[IDList[i
    ]]])
```

```
346     listOfObjects = comSetDFNum.iloc[IDList]
347     imputeValues = []
348     # Calculate the mean of each numeric attributes in the
     list of objects
349     for j in range(0, comSetDFNum.shape[1]):
350         imputeValues.append(round(mean(pd.to_numeric(
    listOfObjects.iloc[:, j])), 2))
351     indexList = inObjectNum.index.T.values
352     l=0
353     for k in range(0, len(indexList)):
354         if inObjectNum[indexList[k]] == '?':
355             inObjectNum[indexList[k]] = imputeValues[l]
356             l += 1
357     return inObjectNum
358 
359 # This function is used to concatenate the incomplete and
    complete objects into a mixed object
360 def concatenationTwoObjects(inObject, comObject):
361     mixedObject = inObject.append(comObject,ignore_index=
    False)
362     # mixedObjectDF = mixedObject.reset_index(drop=False)
363     indexList = np.sort(mixedObject.index.T.values)
364     mixedObject = mixedObject.loc[indexList]
365     return mixedObject
366 
367 # This function is used to calculate Euclidean distance
    for numeric attributes.
368 def euclidean_dissim(a, b, **_):
369     """Euclidean distance dissimilarity function"""
370     # if np.isnan(a).any() or np.isnan(b).any():
371     #     raise ValueError("Missing values detected in
    numerical columns.")
372     return np.sum((a - b) ** 2, axis=1)
373 
374 # This function is to calculate the means of all numeric
    columns in the complete numeric dataset.
375 def meansColumn(comSetDFNum, column):
376     return round(mean(pd.to_numeric(comSetDFNum.iloc[:,
    column])), 2)
377 
378 '''
379 The k-CMM algorithm for clustering a mixed dataset with
    missing values X into k clusters.
380 * init is the initial method for the algorithm (in this
    case, we used randomness)
381 * n_init is the number of runing algorithm with different
    initialization
382 * max_iter is the number of executing the algorithm
383 * verbose == 1 print information, 0 is otherwise
384 '''
```

```
385
386 def perform_kCMM(X, categorical, n_clusters, init, verbose
    , use_global_attr_count):
387     '''k-CMM algorithm'''
388     inSetDF, comSetDF = readData(X)
389     # inSet = np.asanyarray(inSetDF)
390     comSet = np.asanyarray(comSetDF)
391
392     inSetDFNum, inSetDFCat = splitNumCat(inSetDF,
    categorical)
393     comSetDFNum, comSetDFCat = splitNumCat(comSetDF,
    categorical)
394
395     npoints, nattrs = comSet.shape
396     assert n_clusters < npoints, "More clusters than data
    points?"
397
398     all_centroids = []
399     all_labels = []
400     all_costs = []
401
402     if init == 'random':
403         seeds = np.random.choice(range(npoints),
    n_clusters)
404         centroids = comSet[seeds]
405     else:
406         raise NotImplementedError
407
408     itr = 0
409     converged = False
410     cost = np.Inf
411
412     # For each categorical object in incomplete dataset
    S_2^c do
413     # for idex in range(0,len(inSetDF)):
414     while len(inSetDFCat)!= 0:
415         inObjectCat = inSetDFCat.loc[0]
416         #Create a set that contains all missing attributes
     of an object, initialize an empty set
417         inSetAttr = []
418         comSetAttr = []
419         # Create a set that contains decision tree for
    missing attributes
420         setDT = {}
421         # Create a set that contains complete dataset
    after change index of columns
422         setComSetDFCatAfterChangeIndex = {}
423         #Put missing attributes of object into the setAttr
424         tmpCount = 0
425         inObjectCatAfterChangeIndex = {}
```

```
426         indexList = inObjectCat.index.T.values
427         for k in range(0, len(indexList)):
428             if inObjectCat[indexList[k]] == '?':
429                 inSetAttr.append(indexList[k])
430             else:
431                 inObjectCatAfterChangeIndex[str(tmpCount)]
    = inObjectCat[indexList[k]]
432                 tmpCount +=1
433                 # Store index of complete attribute to
    build the decision tree
434                 comSetAttr.append(indexList[k])
435         # print("Finish finding missing attributes for
    object "+ str(index))
436         # Check if the object contains no missing
    categorical values. In case of missing values in numeric
    attributes
437         if (len(inSetAttr) == 0):
438             inObjectNum = inSetDFNum.loc[0]
439             indexList = inObjectNum.index.T.values
440             for k in range(0, len(indexList)):
441                 if inObjectNum[indexList[k]] == '?':
442                     inObjectNum[indexList[k]] =
    meansColumn(comSetDFNum, k)
443         else:
444             for index in inSetAttr:
445                 # This step is to move the missing
    attribute as the class attribute in the comSetDFCat
446                 colsAfterChangeIndex = comSetAttr.copy()
447                 colsAfterChangeIndex.append(index)
448                 tmpList = comSetDFCat[colsAfterChangeIndex
    ]
449                 comSetDFCatAfterChangeIndex = comSetDFCat[
    colsAfterChangeIndex]
450                 # print(comSet)
451                 tmpList = tmpList.reset_index(drop=True).
    values.tolist()
452                 # Build a decision tree for a missing
    attribute of an object in incomplete set
453                 decisionTree = c45.growDecisionTreeFrom(
    tmpList)
454                 # c45.plot(decisionTree)
455                 # Add decision tree to the set of DT
456                 setDT[index] = decisionTree
457                 setComSetDFCatAfterChangeIndex[index] =
    comSetDFCatAfterChangeIndex
458             # print("Finish building DTs for missing
    attributes")
459
460             #Create a table that contains all correlated
    objects in a DT
```

```
461             tableOfObjects = []
462             for key, value in setDT.items():
463                 # c45.plot(value)
464                 # getAllLeavesInDT(value)
465                 comSetDFCatAfterChangeIndex =
    setComSetDFCatAfterChangeIndex[key]
466                 listBestPath = findSuitableLeave(value,
    inObjectCatAfterChangeIndex)
467                 for path in listBestPath:
468                     listOfObject =
    assignCompleteObjectsIntoDT(path,
    comSetDFCatAfterChangeIndex.T.reset_index(drop=True).T)
469                     tableOfObjects.append(listOfObject)
470             # Merge correlated complete objects with
    incomplete object into one collection
471             IDList =list(itertools.chain.from_iterable(
    tableOfObjects))
472             IDList = list(dict.fromkeys(IDList))
473             IDList.sort()
474             listOfObjects =[]
475             for i in IDList:
476                 listOfObjects.append(comSetDFCat.iloc[[i]]
    )
477
478             # This step is to impute missing values for
    categorical attributes
479             imputeValues = IS_MCS_Measure(inObjectCat,
    comSetAttr, inSetAttr, listOfObjects)
480
481             indexList = inObjectCat.index.T.values
482             i = 0;
483             for j in range (0, len(indexList)):
484                 if inObjectCat[indexList[j]] == '?':
485                     inObjectCat[indexList[j]] =
    imputeValues[i]
486                     i +=1
487
488             # This step is to fill in missing values for
    numeric attributes
489             inObjectNum = numericImputation(inSetDFNum,
    comSetDFNum, IDList)
490
491         mixedObject = concatenationTwoObjects(inObjectNum,
    inObjectCat)
492         #This step is to add imputed object into complete
    dataset and remove it to complete datasets
493         comSetDF = pd.concat([comSetDF, mixedObject.
    to_frame().T], ignore_index=True)
494         comSet = np.asanyarray(comSetDF)
495         inSetDFNum = inSetDFNum.drop(inSetDFNum.index[0])
```

```
496         inSetDFNum = inSetDFNum.reset_index(drop=True)
497         inSetDFCat = inSetDFCat.drop(inSetDFCat.index[0])
498         inSetDFCat = inSetDFCat.reset_index(drop=True)
499         inSetDF = inSetDF.drop(inSetDF.index[0])
500         inSetDF = inSetDF.reset_index(drop=True)
501         # Then, this step is to perform clustering process
502         npoints, nattrs = comSet.shape
503         global_attr_freq = _cal_global_attr_freq(comSet,
    npoints, nattrs, categorical)
504         cl_attr_freq, membership = _init_clusters(comSet,
    centroids, n_clusters, nattrs, npoints, verbose,
    categorical)
505 
506         centroids = [[defaultdict(float) for _ in range(
    nattrs)]
507                      for _ in range(n_clusters)]
508         # Perform initial centroid update
509         lbd = np.zeros(n_clusters, dtype='float')
510         for ik in range(n_clusters):
511             cluster_members = sum(membership[ik, :])
512             attrIndex = 0
513             for iattr in range(nattrs):
514                 if iattr in categorical:
515                     centroids[ik][iattr] =
    cal_centroid_value(lbd[ik], cl_attr_freq[ik][attrIndex],
    cluster_members,
516 
    len(global_attr_freq[attrIndex]))
517                     attrIndex += 1
518                 else:
519                     centroids[ik][iattr] = cal_mean_value(
    comSetDF,iattr)
520         # print(comSet)
521         # print("\n")
522         # print(inSetDF)
523         # '''
524         # Bước lặp chính của thuật toán
525         # 1. Tính các vector trung tâm, lambda
526         # 2. Nếu dissimilarity mới (cost) nhỏ hơn thì cập
    nhật và tiếp tục thực
527         # hiện thuật toán lần nữa (từ bước 1).
528         # Nếu lớn hơn thì kết thúc thuật toán.
529         # '''
530         # while itr <= max_iter and not converged:
531         # while not converged:
532         #     itr += 1
533         #     if verbose:
534         #         print("...k-center loop")
535         centroids, moves, lbd = _k_CMM_iter(comSet,
    categorical, centroids, cl_attr_freq, membership,
```

```
535 global_attr_freq, lbd,
536
    use_global_attr_count)
537         labels, ncost = _labels_cost(comSet, categorical,
    centroids, global_attr_freq)
538         cost = ncost
539         # Store result of current run
540         all_centroids.append(centroids)
541         all_labels.append(labels)
542         all_costs.append(cost)
543         # while itr <= max_iter and not converged:
544     converged = (moves == 0) or (ncost >= cost)
545     if not converged:
546         while not converged:
547             all_costs = []
548
549             if verbose:
550                 print("...k-center loop")
551             centroids, moves, lbd = _k_CMM_iter(comSet,
    centroids, cl_attr_freq, membership, global_attr_freq, lbd
    ,
552
    use_global_attr_count)
553             if verbose:
554                 print("...Update labels, costs")
555             labels, ncost = _labels_cost(comSet,
    categorical, centroids, global_attr_freq)
556             converged = (moves == 0) or (ncost >= cost)
557             cost = ncost
558             # if verbose:
559             #     print("Run {}, iteration: {}/{}, moves
    : {}, cost: {}"
560             #         . format(init_no + 1, itr, max_iter
    , moves, cost))
561             # Store result of current run
562             all_centroids.append(centroids)
563             all_labels.append(labels)
564             all_costs.append(cost)
565     '''
566     Return the labels that contains exactly the number of
    data points in the original dataset X
567     '''
568     # print(all_labels[-1])
569     return all_centroids[-1], all_labels[-1], all_costs[-1
    ]
570
571
572 class KCMM(object):
573
574     '''k-CMM clustering algorithm for mixed numeric and
```

```
574 categorical data with missing values
575 
576     Parameters
577     ----------
578     K : int, optional, default: 8
579         The number of clusters to form as well as the
    number of
580         centroids to generate.
581 
582     categorical: ndarray
583                  Indicate the list of categorical
    attributes in a dataset
584 
585     max_iter : int, default: 300
586         Maximum number of iterations of the k-modes
    algorithm for a
587         single run.
588 
589     n_init : int, default: 10
590         Number of time the k-modes algorithm will be run
    with different
591         centroid seeds. The final results will be the best
     output of
592         n_init consecutive runs in terms of cost.
593 
594     init : 'random'
595         'random': choose k observations (rows) at random
    from data for
596         the initial centroids.
597 
598     verbose : boolean, optional
599         Verbosity mode.
600 
601     Attributes
602     ----------
603     cluster_centroids_ : array, [K, n_features]
604         Categories of cluster centroids
605 
606     labels_ :
607         Labels of each point
608 
609     cost_ : float
610         Clustering cost, defined as the sum distance of
    all points to
611         their respective cluster centroids.
612     '''
613 
614     def __init__(self, categorical, n_clusters, init='
    random', n_init=3, max_iter = 10,
615         verbose=1, use_global_attr_count=1):
```

```
616         if verbose:
617             print("Number of clusters: {0}" . format(
    n_clusters))
618             print("Init type: {0}" . format(init))
619             print("Max iterations: {0}" . format(max_iter)
    )
620             print("Use global attributes count: {0}" .
    format(use_global_attr_count > 0))
621 
622         if hasattr(init, '__array__'):
623             n_clusters = init.shape[0]
624             init = np.asarray(init, dtype = np.float64)
625 
626         self.categorical = categorical
627         self.n_clusters = n_clusters
628         self.init = init
629         self.n_init = n_init
630         self.verbose = verbose
631         self.max_iter = max_iter
632         self.use_global_attr_count = use_global_attr_count
633 
634     def fit(self, X, **kwargs):
635         '''Compute k-CMM clustering.
636 
637         Parameters
638         ----------
639         X : array-like, shape=[n_samples, n_features]
640         '''
641 
642         self.cluster_centroids_, self.labels_, self.cost_
    = \
643             perform_kCMM(X, self.categorical, self.
    n_clusters, self.init, self.verbose, self.
    use_global_attr_count)
644         return self
645 
646     def fit_predict(self, X, **kwargs):
647         '''Compute cluster centroids and predict cluster
    index for each sample.
648 
649         Convenience method; equivalent to calling fit(X)
    followed by
650         predict(X).
651         '''
652         return self.fit(X, **kwargs).labels_
653 
654 #This function is used to read a data with missing values
    and then split it into two parts: completed dataset and
    missing dataset
655 def readData(x):
```

```
656     inSet = []
657     comSet = []
658     for i in range(0, len(x)):
659         object = list(x[i])
660         for j in range(0, len(object)):
661             flag = False
662             if(object[j] == '?'):
663                 inSet.append(object)
664                 flag = True
665                 break;
666         if(flag == False):
667             comSet.append(object)
668     inSetDF = pd.DataFrame(inSet)
669     comSetDF = pd.DataFrame(comSet)
670     return inSetDF, comSetDF
671 
672 def getLeafNodes(decisionTree):
673     result = []
674     for key, value in decisionTree.results.items():
675         result.append(key)
676     return result
677 
678 # This function is used to get all paths in a decision
    tree where they have the same categorical values in many
    column
679 def getAllPathsInDT(decisionTree):
680     result = []
681     firstTime = True
682     def recursiveCall(decisionTree, list, firstTime):
683         if decisionTree.results != None:  # leaf node
684             leafNodeList = getLeafNodes(decisionTree)
685             for i in range(len(leafNodeList)):
686                 tmpList = list.copy()
687                 # tmpList.append(str(decisionTree.results
    ).split()[0].split("{'")[1])
688                 tmpList.append(leafNodeList[i])
689                 result.append(tmpList)
690             return
691         else:
692             # index column of each decision node is added
    to distinguish the same categories
693             # appearing in attributes.
694             decision = decisionTree.value
695             decisionTrue = str(decisionTree.col) + '-' +
    str(decision)
696             decisionFalse = str(decisionTree.col) + '-!' +
     str(decision)
697             # if(decisionTree.trueBranch.value == None):
698             if(firstTime == True):
699                 tmpTrue = [decisionTrue]
```

```
700                 tmpFalse = [decisionFalse]
701                 firstTime = False
702             else:
703                 tmpTrue = list.copy()
704                 tmpTrue.append(decisionTrue)
705                 tmpFalse = list.copy()
706                 tmpFalse.append(decisionFalse)
707             recursiveCall(decisionTree.trueBranch, tmpTrue
    , firstTime)
708             recursiveCall(decisionTree.falseBranch,
    tmpFalse, firstTime)
709     recursiveCall(decisionTree, None, firstTime)
710     # print(result)
711     return result
712
713 def splitNodeInDT(node):
714     if node.find('-')== -1:
715         return -1, node
716     else:
717         result = node.split('-')
718     return result[0], result[1]
719
720 def findSuitableLeave(decisionTree, inObject):
721     # print(object)
722     bestPath = []
723     listOfPaths = getAllPathsInDT(decisionTree)
724     for i in range(0,len(listOfPaths)):
725         # path = pd.DataFrame(listOfPaths[i])
726         path = listOfPaths[i]
727         # print(type(path))
728         # print(path)
729         for j in range(0,len(path)):
730             nodeIndex, nodeValue = splitNodeInDT(path[j])
731             if nodeIndex == -1:
732                 bestPath.append(path)
733                 break
734             if(nodeValue[0]!='!'):
735                 if(inObject[nodeIndex]!=nodeValue):
736                     break
737             else:
738                 nodeValue = nodeValue[1:]
739                 if(inObject[nodeIndex]==nodeValue):
740                     break
741     return bestPath
742
743 #This function is to assign objects in the complete
    dataset into leaves in DT
744 def assignCompleteObjectsIntoDT(path, comSet):
745     listOfObject = []
746     for i in range(0,len(comSet)):
```

```
747         comObject = comSet.loc[i]
748         for j in range(0, len(path)):
749             nodeIndex, nodeValue = splitNodeInDT(path[j])
750             if nodeIndex == -1:
751                 listOfObject.append(i)
752                 break
753             if (nodeValue[0] != '!'):
754                 if (comObject[int(nodeIndex)] != nodeValue
    ):
755                     break
756             else:
757                 nodeValue = nodeValue[1:]
758                 if (comObject[int(nodeIndex)] == nodeValue
    ):
759                     break
760     return listOfObject
761 
762 # This function is used to calculate the IS measure that
    is the correlation between set of attributes with non-
    missing values C and set
763 # of attributes with missing values M within a record. The
     IS measure measures the degree of associations between
    two sets of attribute values.
764 # IS = Support(C,M)/sqrt(Support(C)*Support(Q)) where
    Support(C,M) = |C,M|/Q, |C,M| is the number of records
    that contain both C and M, Q is the size of dataset
765 
766 def getFrequency(comSetAttr, inSetAttr, listOfObjects):
767     valueSet = {} # Use to store: key: complete values,
    values: values at corresponding missing attributes in
    complete object
768     #  that match with complete values in incomplete
    object
769     comObjectSet = {} # Use to store: key: complete values
    , values: indices of complete objects
770     # that will be used for calculating the MSC measure
771     freqC = {}
772     freqM ={}
773     freqCM = {}
774     for i in range(0,len(listOfObjects)):
775         comObject = listOfObjects[i]
776         # Note that the variable missingValues is the
    values at the missing attribute in the complete object
777         missingValues = comObject[inSetAttr]
778         missingValuesDF = missingValues.copy()
779         missingIndexList = missingValues.columns.values
780         for index in missingIndexList:
781             missingValuesDF.loc[:,index] = str(index)+'-'+
     missingValues.loc[:,index]
782         missingValuesTuple = [tuple(x) for x in
```

```
782 missingValuesDF.values][0]
783         #
784         completeValues = comObject[comSetAttr]
785         completeValuesDF = completeValues.copy()
786         completeIndexList = completeValues.columns.values
787         for index in completeIndexList:
788             completeValuesDF.loc[:,index] = str(index)+'-'
    + completeValues.loc[:,index]
789         completeValuesTuple = [tuple(x) for x in
    completeValuesDF.values][0]
790         valueSet[completeValuesTuple] = missingValuesTuple
791         if completeValuesTuple in comObjectSet:
792             objects = comObjectSet[completeValuesTuple]
793             if i not in objects:
794                 objects.append(i)
795             comObjectSet[completeValuesTuple] = objects
796         else:
797             objects = [i]
798             comObjectSet[completeValuesTuple] = objects
799         CMTuple = completeValuesTuple + missingValuesTuple
800         if missingValuesTuple in freqM:
801             count = freqM[missingValuesTuple]
802             count += 1
803             freqM[missingValuesTuple] = count
804         else:
805             freqM[missingValuesTuple] = 1
806         if completeValuesTuple in freqC:
807             count = freqC[completeValuesTuple]
808             count +=1
809             freqC[completeValuesTuple] = count
810         else:
811             freqC[completeValuesTuple] = 1
812         if CMTuple in freqCM:
813             count = freqCM[CMTuple]
814             count += 1
815             freqCM[CMTuple] = count
816         else:
817             freqCM[CMTuple] = 1
818
819     return comObjectSet,  freqM, freqC, freqCM, valueSet
820
821 # This function is used to calculate the frequency of
    categorical values in listOfObjects
822 def categoricalValuesFrequency(inObject, comSetAttr,
    listOfObjects):
823     inObjectCat = pd.DataFrame(inObject).T
824     listOfObjects.append(inObjectCat)
825     result = {}
826     # print(listOfObjects[0])
827     # indexList = inObject.index.T.values
```

```
828     for i in range(0,len(listOfObjects)):
829         comObject = listOfObjects[i]
830         for j in range(0, len(comSetAttr)):
831             categoryValue  = str(comSetAttr[j]) + '-' +
    comObject[comSetAttr[j]]
832             categoryTuple = tuple(categoryValue)
833             if categoryTuple in result:
834                 count = result[categoryTuple]
835                 count = count +1
836                 result[categoryTuple] = count
837             else:
838                 result[categoryTuple] = 1
839     return result, len(listOfObjects)
840 
841 def IS_MCS_Measure(inObject, comSetAttr, inSetAttr,
    listOfObjects):
842     resultIS = {}
843     q = len(listOfObjects)
844     comObjectSet, freqM, freqC, freqCM, valueSet =
    getFrequency(comSetAttr, inSetAttr, listOfObjects)
845     for key, value in freqC.items():
846         supC = value/q
847         supM = freqM[valueSet[key]]/q
848         _key = list(key)
849         CM = _key.copy()
850         for j in range(0, len(valueSet[key])):
851             CM.append(valueSet[key][j])
852         CMTuple = tuple(CM)
853         supCM = freqCM[CMTuple]/q
854         IS = supCM/math.sqrt(supC*supM)
855         resultIS[key]= IS
856     # The following code is used to calculate the MCS
    measure that quantifies the similarity between an object
    with missing values
857     #  and an object with no missing values
858     frequency, cardinality = categoricalValuesFrequency(
    inObject, comSetAttr, listOfObjects)
859     del listOfObjects[-1]
860     resultMCS = {}
861     for key, value in comObjectSet.items():
862         msc = 0
863         for j in range(0, len(comSetAttr)):
864             # Note that the variable missValue below is
    the category in complete attribute of incomplete object
865             # print(inObject[comSetAttr[j]])
866             missValue = str(comSetAttr[j]) + '-' +
    inObject[comSetAttr[j]]
867             missValueTuple = tuple([missValue])
868             compValueTuple = tuple([key[j]])
869             if missValueTuple == compValueTuple:
```

```
870                 msc += 1
871             else:
872                 freqMissValue = frequency[missValueTuple]
    / cardinality
873                 freqComValue = frequency[compValueTuple] /
     cardinality
874                 sumFreq = (frequency[missValueTuple] +
    frequency[compValueTuple]) / cardinality
875                 msc += (2 * math.log(sumFreq)) / (math.log
    (freqComValue) + math.log(freqMissValue))
876         resultMCS[key] = msc
877     sumIS_MCS = []
878     # Calculate the affinity degree for the IS and MCS
    measures
879     for key, value in resultMCS.items():
880         sumIS_MCS.append((resultIS[key]+value)/2.0)
881     sumValue = sum(sumIS_MCS)
882     randomSampling = []
883     for i in range(0,len(sumIS_MCS)):
884         randomSampling.append(sumIS_MCS[i]/sumValue)
885     maxIndexValue = randomSampling.index(max(
    randomSampling))
886     tmp = list(resultIS)[maxIndexValue]
887     imputeValues = valueSet[tmp]
888     result = []
889     for value in imputeValues:
890         result.append(value.split('-')[1])
891     return result
892
893 def do_kr(x, y, nclusters, verbose, use_global_attr_count,
     n_init):
894     start_time = time()
895     tracemalloc.start()
896     categorical = [0, 3, 4, 5, 6, 8, 9, 11, 12]
897     kr = KCMM(categorical, n_clusters = nclusters, init='
    random',
898         n_init = n_init, verbose = verbose,
    use_global_attr_count = use_global_attr_count)
899     kr.fit_predict(x)
900     # print(kr.labels_)
901
902     ari = evaluation.rand(kr.labels_, y)
903     nmi = evaluation.nmi(kr.labels_, y)
904     purity = evaluation.purity(kr.labels_, y)
905     end_time = time()
906     elapsedTime = timedelta(seconds=end_time - start_time)
    .total_seconds()
907     memoryUsage = tracemalloc.get_tracemalloc_memory() /
    1024 / 1024
908     if verbose == 1:
```

```
909         print("Purity = {:8.3f}".format(purity))
910         print("NMI = {:8.3f}".format(nmi))
911         print("Adjusted rand index = {:8.3f}" . format(ari
    ))
912         print("Elapsed Time = {:8.3f} secs".format(
    elapsedTime))
913         print("Memory usage = {:8.3f} MB".format(
    memoryUsage))
914         print("Finish all task!")
915     tracemalloc.stop()
916     return [round(purity, 3), round(nmi, 3), round(ari, 3)
    , round(elapsedTime, 3), round(memoryUsage, 3)]
917
918 def run(argv):
919     max_iter = 10
920     ifile = "data/mixed_credit.csv"
921     ofile = "output/credit.csv"
922     use_global_attr_count = 0
923     use_first_column_as_label = False
924     verbose = 1
925     delim = ","
926     n_init = 3
927
928     # Get samples & labels
929     if not use_first_column_as_label:
930         x = np.genfromtxt(ifile, dtype = str, delimiter =
    delim)[:, :-1]
931         y = np.genfromtxt(ifile, dtype = str, delimiter =
    delim, usecols = -1)
932     else:
933         x = np.genfromtxt(ifile, dtype = str, delimiter =
    delim)[:, 1:]
934         y = np.genfromtxt(ifile, dtype = str, delimiter =
    delim, usecols = 0)
935
936     from collections import Counter
937     nclusters = len(list(Counter(y)))
938     result = []
939     for i in range(max_iter):
940         if verbose:
941             print("\n===============Run {0}/{1} times
    ===============\n" . format(i + 1, max_iter))
942         result.append(do_kr(x, y, nclusters, verbose =
    verbose, use_global_attr_count = use_global_attr_count,
    n_init = n_init))
943     resultDF = pd.DataFrame(result)
944     tmpResult = []
945     for i in range(0,5):
946         tmpResult.append(cal_mean_value(resultDF,i))
947     finalResult = [['Purity', 'NMI', 'Adjusted rand index'
```

```
947 , "Elapsed Time", "Memory Usage"]]
948     finalResult.append(tmpResult)
949     import csv
950     with open(ofile, 'w') as fp:
951         writer = csv.writer(fp, delimiter = ',')
952         writer.writerows(finalResult)
953 
954 
955 if __name__ == "__main__":
956     run(sys.argv[1:])
```