K-ANONYMITY IMPLEMENTATION VIA SAMARATI AND MONDRIAN

Jiahui Huang
Electronic Information Engineering
University of Science and Technology of China
Hefei, Anhui, 230026
hjh233@mail.ustc.edu.cn

ABSTRACT

K-anonymity is a property possessed by certain anonymized data. The concept of k-anonymity was first introduced by Latanya Sweeney and Pierangela Samarati in a paper published in 1998 as an attempt to solve the problem: "Given person-specific field-structured data, produce a release of the data with scientific guarantees that the individuals who are the subjects of the data cannot be re-identified while the data remain practically useful." A release of data is said to have the k-anonymity property if the information for each person contained in the release cannot be distinguished from at least k - 1 individuals whose information also appear in the release.[1] Here we implement k-anonymity via two algorithms: Samarati and Mondian and have further discussions about what difference does it make with different parameters, e.g., k and MaxSup.

Keywords K-anonymity · Samarati · Mondrian

1 Introduction

Sweeney[2] demonstrated that releasing a data table by simply removing identifiers can seriously breach the privacy of individuals whose data are in the table. By combinig a public voter registration list and a released medical database of health insurance information, she was able to identify the medical record of the governor of Massachusetts.
This kind of attack is called *linking attack*. To protect data from linking attacks, Samaratia and Sweeney proposed *k-anonymity*[2,3].

Definition 1. ***(k-Anonymity)*** *Given a set of QI attributes $Q_1, \ldots, Q_d$, release candidate D^* is said to be k-anonymous with respect to $Q_1, \ldots, Q_d$ if each unique tuple in the projection of D^* on $Q_1, \ldots, Q_d$ occurs at least k times.*

To implement *k-anonymity*, we have the following two algorithms proposed by Samarati[3], Mondrian[4]. We give the pseudo-code in the following part.

2 Relevant Algorithms

2.1 Samarati

For Algorithm 1 (Samarati) which deals with multiple categotical attributes, basic idea can be illustrated as follows:

1. Define the hierarchy of generealization and Construct a Lattice.
2. For instance, the distance vector corresponding to a generalized table with hierarchy $< X_1, Y_1 >$ is [1,1]
3. Given the constraint that the generalized table must satisfy k-anonymity and that the deleted records is less than MaxSup, we need to set the sum of all the distance vectors as small as possible.

[1] Cited from Wikipedia

Consequently, we use bisection method to complete the process: For a lattice with its maximun height h, we determine whether the nodes with height h/2 is k-anonymous. If so, we proceed with the nodes with height h/4, otherwise we check the nodes with height 3h/4. We repeat the above process until the layer with minimun height satisfying k-anonymity is found.
Detailed algorithm is shown as below:

Algorithm 1: Samarati

Input: Table $T_i = \mathbf{PT}[QI]$ to be generalized, anonymity requirement k, suppression threshold $Maxsup$, lattice $\mathbf{VL}_{DT}$ of the distance vectors corresponding to the domain generalization hierarchy $\mathbf{DGH}_{DT}$, where DT is the tuples of the domains of the quasi-identifier attributes

Output: The distance vector sol of a generalized table $\mathbf{GT}_{sol}$ that is $k-minimal$ generalization of $\mathbf{PT}[QI]$

$low := 0;\ high := height(T, \mathbf{VL}_{DT});\ sol := T;$
while *low*<*high* **do**
  try:=$\lfloor \frac{low+high}{2} \rfloor$
  $Vectors := \{vec | height(vec, \mathbf{VL}_{DT}) = try\}$
  $reach\_k :=$ **false**
  while $Vectors \neq \varnothing \wedge reach\_k \neq$ ***true*** **do**
    Select and remove a vector vec from $vectors$
    if satisfies$(vec, k, T_i, Maxsup)$ **then** $sol := vec;\ reach\_k =$ **true**
    if $reach\_k =$ **true then** $high := try$ **else** $low := try + 1$
  end
end
Return sol

2.1.1 Coding implemetation of Samarati

1. We first load the original data and preprocess it. Since some data are not full, that is, some attributes value is lost and is shown as '?'. We relace them with np.nan and use dropna() to remove them. Since drop is set as True, we will have a new datafram with clean data (30162x15)

```
data_nan = data_original.replace('?', np.nan)
# We replace ? with np.nan, which can be detected and deleted automatically via dropna
data_cleaned = data_nan.dropna().reset_index(drop = True)
```

2. We define Hierarchy = $(Layer\_sex, Layer\_race, Layer\_marital\_status, Layer\_age)$, each item with the range of 0-1, 0-1, 0-2, 0-4. The total generalization number ranges from 0 to 8, 0 meaning no generalizing the data. Therefore, we use a for loop to finish the construction work.

```
for Layer_sex in range(2):
    for Layer_race in range(2):
        for Layer_marital_status in range(3):
            for Layer_age in range(5):
                Lattice[Layer_age + Layer_marital_status + Layer_race + Layer_sex].add
                    ((Layer_sex, Layer_race, Layer_marital_status, Layer_age))
```

3. Since there are only two layers w.r.t. sex and race, we only need to set sex/race to '*' if it needs generalization. As for marital status, we construct three lists representing the hierarchy of it. Since the sequence of each element in a list is fixed, we therefore can construct a n-to-1 mapping in order to generalize the property. Similaily, we define a list containing 3 sets representing 3 hierarchies, each set containing mutiple tuples to which certain ages belong.

```
Sex = ['*']

Race = ['*']

Marital_status = ['Married-spouse-absent','Widowed','Separated','Divorced',
                  'Married-civ-spouse','Married-AF-spouse','Never-married']

Marital_status_G1 = ['Alone','Leave','Married','NM']
```

```
Marital_status_G2 = ['*']

Age = [{(16,20),(21,25),(26,30),(31,35),(36,40),(41,45),(46,50),(51,55),
       (56,60),(61,65),(66,70),(71,75),(76,80),(81,85),(86,90),(91,95)},
       {(16,25),(26,35),(36,45),(46,55),(56,65),(66,75),(76,85),(86,95)},
       {(16,35),(36,55),(56,75),(76,95)}]
```

4. We give the functions generalizing each property as below.

```
def GeneralizeAge(i,G):
    if G != 4:
        for vector in Age[G-1]:
            if vector[0] <= i and vector[1] >= i:
                break
        return str(vector)
    else:
        return star
```

```
def GeneralizeSex(str,G):
    if G == 1:
        return star
    else:
        return str
```

```
def GeneralizeRace(str,G):
    if G == 1:
        return star
    else:
        return str
```

```
def GeneralizeMarital(str,G):
    if G == 2:
        return star
    if G == 1:
        index = LocMaritalStatus(str)
        return Marital_status_G1[int(index/2)]
    else:
        return str
```

where LocMaritalStatus is a function aiming to find the index of marital_status in the list Marital_status.

5. After generalization, we need to find each QI-cluster, which can be solved via groupby function. After clustering, we can find the clusters that does not meet the demand of k-anonymity. Therefore, we derive the suppression number and can decide whether the table can be released or not (We can't release the table if the number of suppressed items is greater than MaxSup, otherwise we can). Certain codes of the function is shown as below.

```
def Cluster(data):
    data_generalized = copy.deepcopy(data)
    Cluster_generalized = data_generalized.groupby(QI, as_index=False).count()
    # We use groupby function to find QI clusters to further discuss whether they satisfy
        k-anonymity
    Cluster_generalized_suppress =
        Cluster_generalized_show[(Cluster_generalized_show['Total_number']<=K)]
    Number_generalized_suppress = Cluster_generalized_suppress['Total_number'].sum()
    print('Total number of QI clusters after generalization that do NOT satisfy
        k-anonymity: %d' % Number_generalized_suppress)
    return Number_generalized_suppress
```

6. We come to implement Samarati at last. That is, we try to find the hierarchy with minimum height. Specific code is given below.

```
def Samarati(data,Lattice):
    low = 0
    high = 8
    while (low < high):
        mid = int((low + high)/2)
        for VECTOR in Lattice[mid]:
            GTC = Generalize(data, VECTOR) # Generalized table candidate
            if Releasable(Cluster(GTC)) == 0:
                low = mid + 1
            else:
                ReleaseVector = copy.deepcopy(VECTOR)
                high = mid

    return ReleaseVector
```

2.2 Mondrian

For algorithm 2 (Mondrian) which deals with multiple numerical attributes, basic idea can be illustrated as folllows:

1. Choose one attribute for every partition, you can choose the attribute with maximum range or just randomly pick one.
2. Find the median of the selected attribute and split the partition. There are 2 ways of doing partitioning, strict partitioning and relax partitioning respectively.
   (a) Say we set k = 2, and we have a dataset = [1,2,3,3,4,5]
   (b) For strict partitioning, we have the split dataset as [1,2,3,3],[4,5]. That is, we place all the median to one bucket.
   (c) For relax partitioning, we have the split dataset as [1,2,3],[3,4,5]. That is, we try to make the partitioned dataset as even as possible.
3. Repeat the process above until every cluster is k-anonymous and no more partitioning is allowed.

Detailed strcict multidimentional partitioning algorithm is shown as below:

Algorithm 2: Mondrian

Anonymize($partition$)
if no allowable multidimensional cut for *partition* **then**
  return $\phi : partition \rightarrow summary$
else
  $dim \leftarrow$ choose_dimension() ;
  $f_s \leftarrow$ frequency_set($partition, dim$) ;
  $splitVal \leftarrow$ find_median(f_s) ;
  $lhs \leftarrow \{t \in partition : t.dim \leq splitVal\}$;
  $rhs \leftarrow \{t \in partition : t.dim > splitVal\}$
end
return Anomymize $(rhs) \cup$ Anonymize (lhs)

There are still some details to be illustrated: After partitioning the maximum cluster, we need to judge whether the partitioned ones are k-anonymous or not. If so, we proceed. If not, we set the table the way it used to be before partitioning and tag the maximum cluster to show that the cluster cannot be partitioned.

2.2.1 Coding implemetation of Mondrian

1. We first load the original data and preprocess it. This is the same as Samarati.

```
data_nan = data_original.replace('?', np.nan)
# We replace ? with np.nan, which can be detected and deleted automatically via dropna
data_cleaned = data_nan.dropna().reset_index(drop = True)
```

2. We add 4 columns in the original data, TempAge and TempEducationNumber respectively to store the generalized formation of age and education_num.If we directly change the value in the original data, we can NOT find the original value of age and education_num once we start generalization. Also, we create DifAge and DifEduNum to store the difference of age and education_num to calculate Loss Metric.

```
# Initialization of TempQI
data['TempAge'] = Native_country
data.loc[:, 'TempAge'] = str((16,95))
data['TempEducationNumber'] = Native_country
data.loc[:, 'TempEducationNumber'] = str((1,20))

# Initialization of DIF
data['DifAge'] = AGE
data.loc[:, 'DifAge'] = 79
data['DifEduNum'] = AGE
data.loc[:, 'DifEduNum'] = 19
```

3. To do the generalization job, we first find the maximum cluster and relevant value of age and education_num. Then we traverse the data, find the items whose value is the same and store the value of age and education_num in two arrays to find MaxAge, MinAge, MidAge, MaxEduNum, MinEduNum, MidEduNum respectively. Then we can do the generalization.

```
max = data_cluster['Total_number'].max()
index = data_cluster[data_cluster.Total_number == max].index.tolist()[0]

ClusterAge = data_cluster.loc[index,'TempAge']
ClusterEducationNumber = data_cluster.loc[index,'TempEducationNumber']

for i in range(30162):
    if data_copy.loc[i,'TempAge'] == ClusterAge and data_copy.loc[i,'TempEducationNumber']
        == ClusterEducationNumber:
        AGE.append(data_copy.loc[i,'age'])
        EDUNUMBER.append(data_copy.loc[i,'education_num'])
    if i == 30161:
        AGE_array = np.array(AGE)
        EDUNUMBER_array = np.array(EDUNUMBER)

def AgeGeneralization(data, AGE_array, ClusterAge, ClusterEducationNumber):
    MinAge = np.min(AGE_array)
    MaxAge = np.max(AGE_array)
    MidAge = np.median(AGE_array)
    print('Maximun, Minimun, Midian of Age is:', MaxAge, MinAge, MidAge)

    for i in range(30162):
        if data.loc[i,'TempAge'] == ClusterAge and data.loc[i,'TempEducationNumber'] ==
            ClusterEducationNumber:
            data.loc[i,'DifAge'] = MaxAge - MinAge
            if data.loc[i,'age'] <= MidAge:
                data.loc[i,'TempAge'] = str((MinAge, MidAge))
            else:
                data.loc[i,'TempAge'] = str((MidAge + 1, MaxAge))
```

4. We randomly choose an attribute and generalize it in the process.

```
    FlipCoin = random.random()
    if (FlipCoin < 0.5):
        EduNumGeneralization(data_copy, EDUNUMBER_array, ClusterAge, ClusterEducationNumber)
    else:
        AgeGeneralization(data_copy, AGE_array, ClusterAge, ClusterEducationNumber)
```

5. Finally, we use recursion to implement Mondrian. Before each iteration, we first come to judge whether the table is k-anonymous after partitioning.

```
global flag
if flag == 0:
    data_cluster.loc[index,'Total_number'] = 0 # TAG the cluster and do not partition it

if Satisfiable(data_copy_cluster) == 1:
    data = copy.deepcopy(data_copy)
    Mondrian(data)

if Satisfiable(data_copy_cluster) == 0:
    flag = 0
    Mondrian(data)
```

3 Experiments and Discussions

3.1 Samarati experiments with different parameters

As is mentioned in the README.txt, to further discuss the impact the parameters exert on the final release table, we set 5 groups of parameters, donated by (K, SupMax): (10,20)(baseline), (5,20), (20,20), (10,10), (10,30).
We give the results of 5 release tables as follows:

(K, SupMax)	Hierarchy	Loss Metric	Suppressed Number	Cluster number
(10,20)	(0,1,2,1)	2.06250	0	30
(5,20)	(1,1,1,1)	2.17675	10	60
(20,20)	(1,1,0,4)	3.00000	0	7
(10,10)	(0,1,2,1)	2.06250	0	30
(10,30)	(0,1,2,1)	2.06250	0	30

Table 1: Measurement of different Release Tables

where Hierarchy = (Layer_sex,Layer_race,Layer_marital_status,Layer_age), Cluster Number is the number of QI clusters satisfying k-anonymity.

We can see that with default parameters (K, SupMax) = (10,20), the hierarchy is (0,1,2,1), loss metric = 2.0625, suppressed number = 0.

It is not hard to see that with fixed K = 10, the generalized table with hierarchy (0,1,2,1) will always satisfy K-anonymity with MaxSup $\in$ [0,20] since not a single record is suppressed. Then it comes as no surprise that with (K, SupMax) = (10, 10), the output remains the same. Moreover, with (K, SupMax) = (10, 30), generalized table with hierarchy (0,1,2,1) undoubtedly is K-anonymous, yet it may not be the hierarchy with minimum loss metric, which will be discussed in the next section.

With fixed MaxSup = 20, we can see that the generalized table with hierarchy (0,1,2,1) will always satisfy K-anonymity with K $\in$ [0,10]. Given K = 5, the release table has a hierarchy of (1,1,1,1), along with its loss metric being 2.17675 and suppressed number being 10. It is apparent that with this hierarchy, the release table is not the best with respect to its utility since the loss metric is larger than 2.0625. From what is given above, we can see that the we may not obtain the 'best' candidate w.r.t. the utility of the release table. Again, we will have further discussion in the next section.

Finally, with (K,MaxSup) being (20,20), we can see that the height of generalized hierarchy is 6, being the greatest among all. Intuitively, we need to generalize more with larger K in that we need to put more records in one cluster, resulting in larger height and greater loss metric. We can see that the experiment result corresonds to our intuition.

3.2 Mondrian experiments with different parameters

Since we use strict partitioning to partition the data, the partitioned data may be quite imbalanced. For instance, if the data is (1,2,2,2,2,2,2,2,2,2), K = 2, then we can't split the data using strict partitioning where the data will be partitioned into (1),(2,2,2,2,2,2,2,2,2). So here we set K as big as possible. We set default K = 100 and three different Ks with their

value being 10,50,1000.
We give the experiment result as follows:

K	Loss Metric	Cluster number
100	0.3459	64
10	0.2887	60
50	0.3166	60
1000	0.4890	17

Table 2: Measurement of different Release Tables

We can see that loss metric increases as the value of K goes up, which corresonds to our intuition discussed in Section 3.1. Also, the cluster numbers are quite close when K = 10,50,100, and the cluster number is way smaller when K = 1000, meaning that the partitioning process may be terminated due to the use of **strict** partitioning, as is discussed at the beginning of the section.

4 Further Discussions

4.1 Select output that maximizes utility with Samarati

Samarati guarantees that we find the final hierarchy with its height minimized. Under such constraint, however, there may still be multiple solutions. Now we seek to find the 'optimal' output with its utility maximized, that is, its loss metric minimized (here we consider it a proper method to evaluate the output). We give our way to find the optimal output as follows:

We initialize 2 lists: HierarchyVector = [], LMHV = [] , to record the hierarchy vectors that can be used to generalize the data satisfying k-anonymity and their loss metric respectively. We only need to find the index of minimum in LMHV, and then we can find the hierarchy that minimize the loss metric simply by getting the value of HierarchyVector[index]. Relevent code is shown as below.

```
def Samarati(data,Lattice,LM,HV): # Hierarchy Vector
    low = 0
    high = 8
    while (low < high):
        mid = int((low + high)/2)
        for VECTOR in Lattice[mid]:
            GTC = Generalize(data, VECTOR) # Generalized table candidate
            if Releasable(Cluster(GTC)) == 0:
                low = mid + 1
            else:
                ReleaseVector = copy.deepcopy(VECTOR)
                HV.append(ReleaseVector) # NEW!
                SupNum = Cluster(data) # NEW!
                LM.append(LossMetric(ReleaseVector, SupNum)) # NEW!
                high = mid

    Min = min(LM) # NEW!
    MinIndex = LM.index(Min) # NEW!

    ReleaseVector = HV[MinIndex] # NEW!

    return ReleaseVector
```

From what is discussed in Section 3.1, we can see that when (K,MaxSup) = (5,20), the release table is definitely not optimal regarding its utility since there is at least one table with smaller loss metric (2.06250). We then run the code and obtain relevant result shown in Table 3.

We can see that for (K,MaxSup) = (10,20), we actually obtain the optimal solution at the begining. Yet for (K,MaxSup) = (5,20), the original solution was quite bad w.r.t. loss metric. The loss metric of optimal solution given by refined Samarati is approximately 0.57 of the original one. We note that the suppressd number rises from 10 to 15, meaning that the improvement of loss metric may be derived at the cost of greater suppressed number. Yet this can also demonstrate

Algorithm	(K, SupMax)	Hierarchy	Loss Metric	Suppressed Number	Cluster number
Original Samarati	(10,20)	(0,1,2,1)	2.06250	0	30
Refined Samarati	(10,20)	(0,1,2,1)	2.06250	0	30
Original Samarati	(5,20)	(1,1,1,1)	2.17675	10	60
Refined Samarati	(5,20)	(0,1,1,2)	1.23991	15	63
Original Samarati	(10,10)	(0,1,2,1)	2.06250	0	30
Refined Samarati	(10,10)	(0,1,2,1)	2.06250	0	30

Table 3: Compare the loss metric of original algorithm and refined algorithm

the superiority of the Refined Samarati since the loss metric goes down even with more suppressed records, meaning the new hierarchy outperforms the original one in the whole dataset to a large extent.

For all the improvements we've made, we may NOT literally obtain the global optimal solution w.r.t. loss metric in that there is a CONSTRAINT in Samarati: We seek to find the solution with MINIMUM HEIGHT. Yet there may exist hierarchies with greater height but smaller loss metric. We give an example in the following table.

Algorithm	(K, SupMax)	Hierarchy	Loss Metric	Suppressed Number	Cluster number
Refined Samarati	(10,20)	(0,1,2,1)	2.06250	0	30
None	(10,20)	(0,1,1,3)	1.36544	19	32

Table 4: An example showing the restriction of Refined Samarati

We can see that when marital status is generalized to level 1, the loss is approximately 0.1129, yet it rises to 1 when generalized to level 2. As for age, the loss for each level is 0, 0.0625, 0.125, 0.25, 1. When we set level_marital_status from 2 to 1, the loss decreases greatly. Even if we set level_age from level 1 to level 3, the total loss metric still decreases.

4.2 Use Mondrian to tackle with categorical attributes

Normally we use Mondrian to tackle with numerical attributes, then how can we use Mondrian to deal with categorical attributes? The first idea we come up with is that we can encode categorical attributes to numerical ones. For example, for attribute 'gender', we can map 'male' to 0 and 'female' to 1. Then we can apply Mondrian to deal with 'gender'. Here we'd better use relax Mondrian in that there are only two numbers in 'gender' and unevenly partitioning can occur quite often.

5 Conclusion

From what is discussed above, we can see that to achieve k-anonymity, we can use generalization via algorithms like Samarati and Mondrian. To obtain better utility, we propose Refined Samarati to generate output with smaller loss metric. We also discuss the restriction of Samarati, that is, we may not find the global optimal output with its loss metric minimized due to the structure of the algorithm. As for Mondrian, we find out that the partitioning process may be terminated due to the use of strict partitioning. Also, we figure out how to employ Mondrian to tackle with categorical attributes.

References

[1] https://en.wikipedia.org/wiki/K-anonymity

[2] Sweeney, Latanya. “Datafly: A System for Providing Anonymity in Medical Data.” Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI: Status and Prospects, 1997, pp. 356–381.

[3] Samarati, P. “Protecting Respondents Identities in Microdata Release.” IEEE Transactions on Knowledge and Data Engineering, vol. 13, no. 6, 2001, pp. 1010–1027.

[4] LeFevre, K., et al. “Mondrian Multidimensional K-Anonymity.” 22nd International Conference on Data Engineering (ICDE’06), 2006, pp. 25–25.