Anonymizing transaction databases for publication: (h,k,p)-coherence

Data Protection and Privacy final assignment

Marco Zucca   Chiara Cossu

Università degli studi di Genova
Computer Science

January 2023

Introduction

What is hkp-coherence?

- Anonymizing algorithm for transactional data
- Used for anonymizing databases for publication
- " Our scenario requires publishing sensitive information, but hiding the identity of data subjects "
- "we model attackers' prior knowledge as subsets of public items."

Glossary

- **P**: “Power of the attacker”
- β: Set of items (public or not)
- β-**cohort**: Set of all transactions containing beta
- **H**: Percentage of private items for a β-cohort (?)
- **K**: Similar to the K in *k-anonimity*
- **Sup** (=Support): Number of transactions for a β-cohort
- **P-breach**: The max `sup(p)` among all p restricted to a β-cohort
- **Mole**: β such that $sup(\beta) < K \vee P_{breach} > H$
- δ: Percentage of public items
- **MM(e)**: The number of moles that contains *e*
- **IL(e)**: Information loss by removing *e* from D

Main algorithm: simple overview

Algorithm Anonymization overview

```
suppress all size-1 moles from D
find moles of size in $[2, p]$
build the mole tree and its score table
while there are minimal moles in D do
    suppress the public item e with the maximum MM(e)/IL(e) from D
end while
```

Implementation and features

- Written in Python 3.9
- Extensive use of Pandas data structures like `DataFrames`
- 2 main blocks: `hkp_anon.py` and `mole_tree.py`
- takes as input a sparse matrix of 1s and 0s
- we tried to stick to paper implementation as much as possible
- additional data manipulation utilities:
    - preprocessing
    - synthetic data generator
    - density evaluation

hkp_anon.py: Usage

```
usage: anon_hkp.py [-h] [-d] [-H H] [-K K] [-P P] [-L L] [-s SENSITIVE [SENSITIVE ...]]
[--delta DELTA] [--seed SEED] [-rmt {mmil,1il,mm,rmall}] [-df DF] [-o OUTPUT] [--stat STAT]
[--preprocess PREPROCESS]

options:
    -h, --help            show this help message and exit
    -d, --debug           print debug info
    -H H
    -K K                  like k-anonymity
    -P P                  power of the attacker
    -L L                  early stop(?)
    -s SENSITIVE [SENSITIVE ...], --sensitive SENSITIVE [SENSITIVE ...]
                            List of sensitive items
    --delta DELTA         Percentage of public items
    --seed SEED           Seed used for selecting private item
    -rmt {mmil,1il,mm,rmall}
                            select the removing method
    -df DF                Dataset to anonymize
    -o OUTPUT, --output OUTPUT
                            Anonymized dataset filename
    --stat STAT           save info for statistics
    --preprocess PREPROCESS
                            create *only* the preprocessed csv
```

Finding moles: `sup()` & `p_breach()`

Algorithm sup(beta)

```
if len(beta) == 1 then
    return sum of D[beta]                    ▷ number of 1s in column D[beta]
end if
row_sums = list: for each row index in D_beta, sum(row)
return number of items in row_sums with value len(beta)
```

Algorithm p_breach(beta)

```
prob = empty list
for each e in sensitive_items do
    $s = sup(beta \cup e)$                   ▷ occurrencies of e in beta-cohort
    append s/sup(beta) to prob
return max(prob)
```

hkp_anon: Find and suppress size 1 moles

Algorithm suppress_size1_moles()

```
size1_moles = empty list
for each cmole in public_items do                              ▷ candidate mole
    if sup(cmole) $<$ K or p_breach(cmole) $>$ H then           ▷ check if mole
        size1_moles.append(cmole)
        drop cmole from D                                      ▷ drop column
        remove cmole from public_items
return size1_moles                                             ▷ list of s1 moles
```

`hkp_anon`: Find minimal moles ($1 < \text{size} \leq L$)

Algorithm find_minimal_moles()

```
$M^*$ = empty list
F = set of public_items                                  ▷ (without size-1 moles)
while $i < L$ & $F$ not empty do                         ▷ i starts from 1
    temp = set of all items in F                         ▷ explode moles sets
    c = set of all size i+1 combinations of temp
    remove_subsets(c)           ▷ remove sets that has a mole subset from c
    F_temp = empty list
    for each cmole in c do                               ▷ candidate mole
        if $sup(cmole) < K$ or $p\_breach(cmole) > H$ then
            append cmole in $M^*$                        ▷ cmole is a mole
        else
            append cmole in F_temp                       ▷ cmole is not a mole
F = set(F_temp)
return $M^*$                                   ▷ list of all moles —— =0
```

hkp_anon: Suppress minimal moles

Algorithm suppress_minimal_moles()

```
supp_item = empty set
MM = create_MM($M^*$)                          ▷ Compute MM for all labels
IL = create_IL()                               ▷ Compute IL for all labels
$M^*$ = sort_tuple($M^*$, MM)                  ▷ order by MM
tree = init_tree(...,$M^*$,...)                ▷ create tree root
build_tree()                                   ▷ Populate tree with $M^*$ values
supp_item = suppress_moles(MM,IL)
drop supp_item from D
                                               ▷ drop columns return supp_item
```

`mole_tree.py`: Overview and data structures

Library for `anon_hkp.py` used for:

- Build the mole tree and score table from scratch
- Find the items to remove based on score table
- Utility methods to explore and debug mole tree

Every node maintain:

- level
- list of visible moles
- children
- father
- label
- #moles passing the node

The score table is a dictionary having labels as keys and `score_list` objects as values that contain:

- MM
- IL
- Linked nodes

mole_tree: Build tree recursively by levels from a node obj

Algorithm build_tree()

```
for each mole in $M^*$ do
    if level < length(mole) then                                    ▷ if false, leaf reached
        new_label = item at position level in mole
        if new_label is in children then                        ▷ child node already exists
            increment by 1 that child's mole_number
        else                                                       ▷ add a new child node
            new_$M^*$: moles in $M^*$ with new_label at position level        ▷ **
            create child node (new_$M^*$, level+1, new_label, ...)
            append new child to children
for each child in children do                                         ▷ recursion by levels
    call build_tree() from child
```

** Create M^* for the child: a node can see only its own moles

mole_tree.py: Suppress moles

Algorithm suppress_moles()

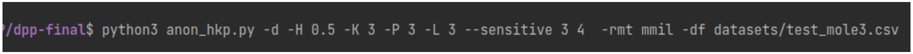
```
score_table = build_score_table(MM, IL)
supp_item = empty set
while score_table is not empty do
    e = key with max MM/IL in score_table          ▷ best item to delete
    add e to supp_item
    for each node of e in score_table link list do
        remove node subtree in mole_tree
    end for
    remove e from score_table
    for each item in score_table do
        if item has MM == 0 then                   ▷ additional check
            remove all linked nodes' subtrees from mole_tree
            remove item from score_table
return supp_item                                   ▷ items to delete from D
```

mole_tree.py: Recursively remove subtree from root

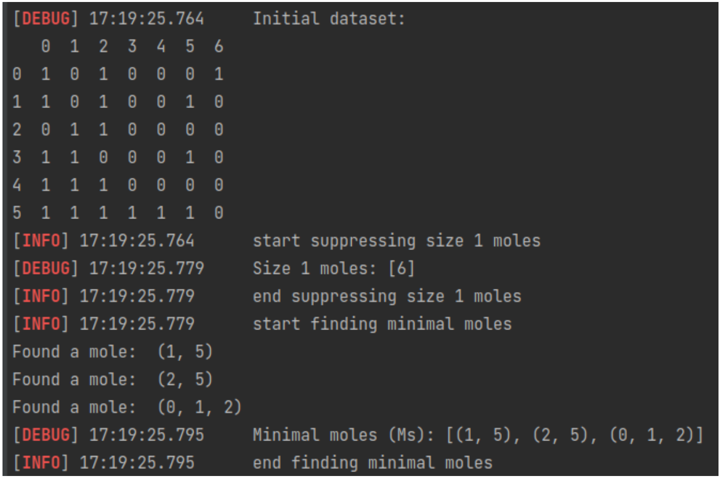
Algorithm remove_subtree(root)

```
if $label \neq root$ then
    remove current node from score table links associated to label
for each child in children do
    call remove_subtree() from child                     ▷ recursive remove
remove all children of current node
if label == root then                        ▷ no more children = no update
    for each ancestor in ancestors do
        ancestor's mole_num -= current node mole_num
        ancestor's MM in score_table -= current node mole_num
    end for
    remove current node (root) from father's children list
label's MM in score_table -= current node mole_num
self-delete current node
=0
```

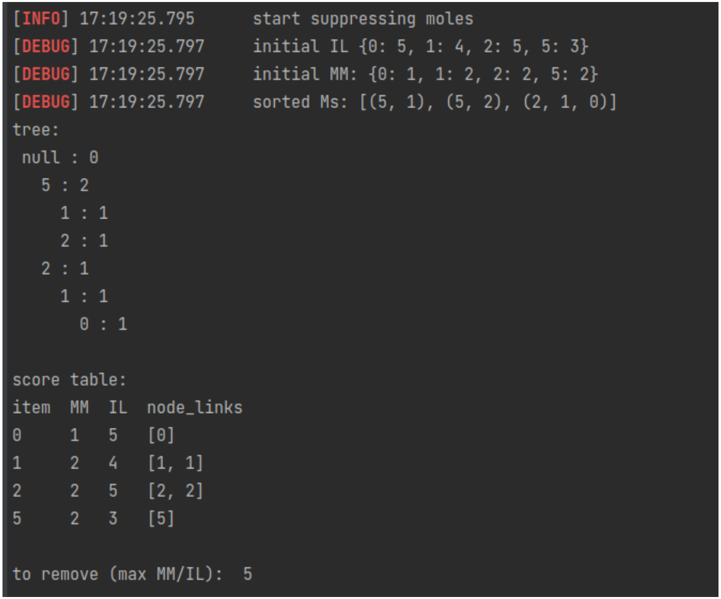
Simple execution example



```
/dpp-final$ python3 anon_hkp.py -d -H 0.5 -K 3 -P 3 -L 3 --sensitive 3 4  -rmt mmil -df datasets/test_mole3.csv
```

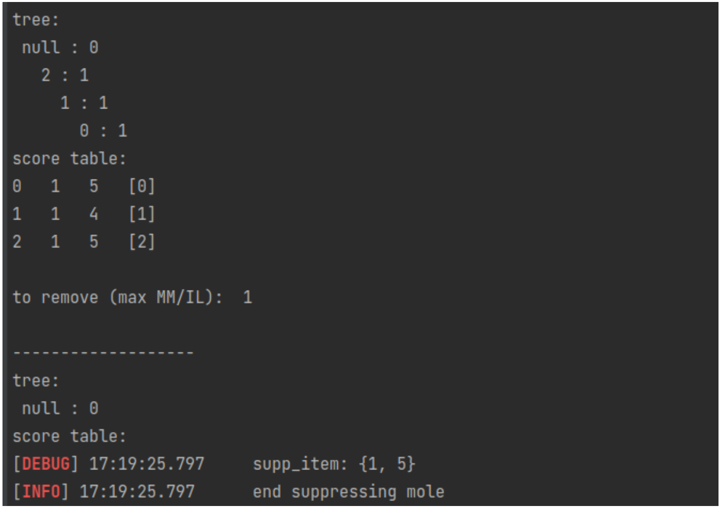


```
[DEBUG] 17:19:25.764     Initial dataset:
   0  1  2  3  4  5  6
0  1  0  1  0  0  0  1
1  1  0  1  0  0  1  0
2  0  1  1  0  0  0  0
3  1  1  0  0  0  1  0
4  1  1  1  0  0  0  0
5  1  1  1  1  1  1  0
[INFO] 17:19:25.764      start suppressing size 1 moles
[DEBUG] 17:19:25.779     Size 1 moles: [6]
[INFO] 17:19:25.779      end suppressing size 1 moles
[INFO] 17:19:25.779      start finding minimal moles
Found a mole:  (1, 5)
Found a mole:  (2, 5)
Found a mole:  (0, 1, 2)
[DEBUG] 17:19:25.795     Minimal moles (Ms): [(1, 5), (2, 5), (0, 1, 2)]
[INFO] 17:19:25.795      end finding minimal moles
```


```
[INFO] 17:19:25.795      start suppressing moles
[DEBUG] 17:19:25.797     initial IL {0: 5, 1: 4, 2: 5, 5: 3}
[DEBUG] 17:19:25.797     initial MM: {0: 1, 1: 2, 2: 2, 5: 2}
[DEBUG] 17:19:25.797     sorted Ms: [(5, 1), (5, 2), (2, 1, 0)]
tree:
 null : 0
   5 : 2
     1 : 1
     2 : 1
   2 : 1
     1 : 1
       0 : 1

score table:
item  MM  IL  node_links
0     1   5   [0]
1     2   4   [1, 1]
2     2   5   [2, 2]
5     2   3   [5]

to remove (max MM/IL):  5
```

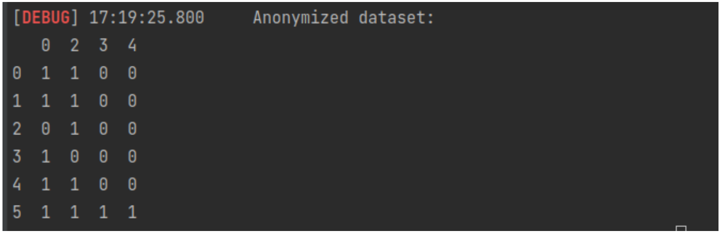


```
tree:
 null : 0
   2 : 1
     1 : 1
       0 : 1
score table:
0   1   5   [0]
1   1   4   [1]
2   1   5   [2]

to remove (max MM/IL):  1


-------------------
tree:
 null : 0
score table:
[DEBUG] 17:19:25.797     supp_item: {1, 5}
[INFO] 17:19:25.797      end suppressing mole
```

Result:



```
[DEBUG] 17:19:25.800     Anonymized dataset:
   0  2  3  4
0  1  1  0  0
1  1  1  0  0
2  0  1  0  0
3  1  0  0  0
4  1  1  0  0
5  1  1  1  1
```

Dataset has been anonymized by removing columns: 1, 5, 6.
This represents the optimal solution: the choice max(MM/IL) for supp_items finds the element e that “appear” in most moles but also minimizes the information loss.

Experimental results

To evaluate the efficiency of our algorithm we relied on the metrics presented on the paper:

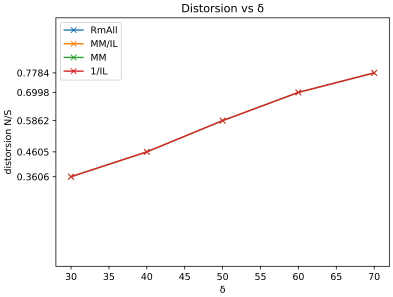
- We visualized the percentage of information loss after the anonymization (IL(supp_item)/total information in D) at the variation of different parameters: **K**, **P** and δ
- We compared different criteria for suppressing an item: **MM/IL**, **1/IL**, **remove_all_public**

We generated synthetic datasets and we evaluated the different performance results on the variation of some features: **density** of information and row/columns proportion.

Note: we choosed to plot the graphs for a single execusion with a fixed random seed to get consistent results.

Results on dataset Connect: a failure

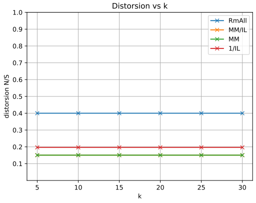
Contrary to what is reported on the paper, the results on the dataset Connect did not show any benefits of using different elimination methods.
Connect size: 10000x20, k=10, p=l=5, h=0.4, δ=0.4



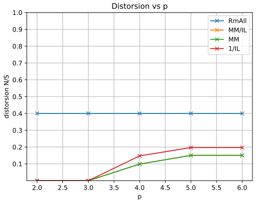
Why? This behaviour may be caused by the distribution of the dataset: we noticed that support values of the columns are highly unbalanced, by random sampling the public items is then pretty easy to find a bad combination that brings up the number of the moles (p_breach() only needs to find a max between many private items).

Connect: Synthetic simulation

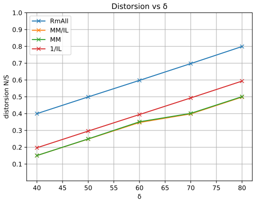
Synthetic matrix size: $10000x20$, k=10, p=l=5, h=0.4, δ=0.4
The density of the datset Connect is 0.3. We reproduced a more balanced, smaller synthetic version by keeping the same density and proportion row/columns:



(a) K



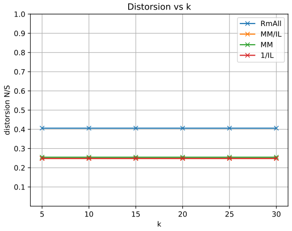
(b) P

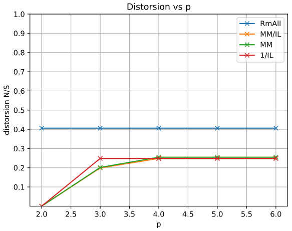


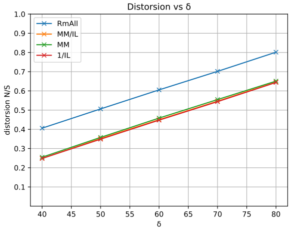
(c) δ

Results for different densities (same parameters):

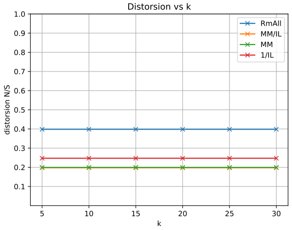
Density 0.1:

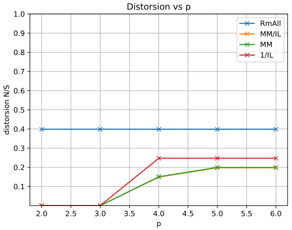


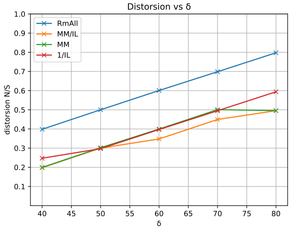




Density 0.2







Conclusion

- It can be seen that the optimal elimination method is MM/IL since it takes in consideration both MM and IL
- The algorithm and the speed of our implementation are strongly affected by the characteristics of the input dataset: dimensions, density, information distribution
- An accurate tuning of all the parameters is needed to get a good result