CM3203 – One Semester Individual Project

Final Report

Implementation of a data privacy protection method for transaction data in Python

Lim Chun Kuan – C1855672

Supervisor: Dr. Jianhua Shao

Moderator: Dr. Neetesh Saxena

Abstract

In the modern world, a huge amount of data is being collected, stored, and processed for various purposes, because these data collected from people or the environment often contains valuable information. At the same time, these data may also include sensitive information and quasi-identifier that could lead to identity disclosure, which violates the privacy of the users.

Terrovitis et al. proposed an anonymization technique that focus on protection against identity disclosure in publication of sparse multidimensional data termed *disassociation* [1]. Disassociation is implemented in C++ in the original work. In this work, it aims to implement the proposed anonymization method in a different platform (Python), and evaluates the results and performance.

Acknowledgement

I want to express great appreciation towards my supervisor, Dr. Jianhua Shao, for providing guidance and assistance throughout the project.

Table of Contents

1. Introduction

As technology progress rapidly year by year, many people realize how important data is, in order to progress further. A variety of digital data has been collected by the government, corporations, and individuals for various purposes, e.g. data analytics. Either due to regulations or mutual benefits, data publishing is often required. To make data to be more accessible for either research purpose or commercial purpose, or the data owner wants the data to be published, or to use these data to achieve a greater good for humanity, these data needed to be published. For example, the cases of COVID-19 in each country are mostly published [8], and a large amount of research data such as the demographic of COVID-19 patients [7]. However, publishing data that contains sensitive information of individuals directly violates individual privacy. Hence, privacy-preserving data publishing (PPDP) has been a topic within some research communities, and many approaches (e.g. generalization and suppression) has been proposed.

1.1 Project Aim/ Project Goal

Disassociation is an anonymization technique proposed by Terrovitis et al. [1] to provide protection against identity disclosure in the publication of sparse multidimensional datasets. It protects the user's privacy by disassociating record terms that participate in identifying combinations [1]. In this project, it intends to implement disassociation in Python, evaluates its results and performance on the same dataset introduced in [1], and compares them to the experimental results that are produced in [1].

1.2 Implementation Approach

The project will be implemented step by step with the disassociation algorithm, after completing and testing one part of the algorithm, then only move onto the next one. For example, horizontal partitioning will be implemented and tested first, and after it is completed and ready for deployment, then only the development for vertical partitioning will begin.

1.3 Project Outcome

The project manages to produce several important outcomes:

- The implementation and correct output of horizon partitioning
- The implementation and correct output of vertical partitioning
- The timings of horizon and vertical partitioning
- Information loss measurements

2. Background

We will provide a background context on privacy issues of transaction data and some proposed privacy models for transaction data.

2.1 Privacy issues for transaction data

A large amount of high dimensional data are being processed and published on a daily basis, and transaction data is one of the instances. Transaction data corresponds to a record owner and its respective set of items under a specific context. Some of the examples are web queries, emails, medical notes etc. These types of data often contain useful information and is a great source for data mining. However, they may also contain sensitive information about the record owners and publishing it without any anonymization techniques could result in identity disclosure and privacy breaching.

A previous case exposes the privacy threats caused by publishing transaction data: AOL published a database of web search queries to the public for research purposes [2]. However, by examining query terms, AOL user No. 4417749 was linked to Ms. Thelma Arnold, a 62-year-old widow who lives in Lilburn. Even if a query does not contain any attribute that are detrimental to identity disclosure (e.g. names or address), the combination of query terms that are adequately unique to a record owner can still be used for re-identification of the record owner. This scandal led not only to the disclosure of private information of AOL users, but also affected data publishers' enthusiasm in offering anonymized transaction data for research purposes [3].

2.2 Privacy model

Privacy model can be broadly categorized into two categories.

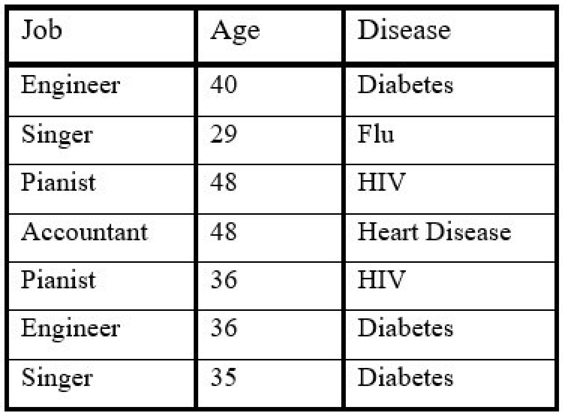
The first category deems that privacy is vulnerable to attack when an adversary is able to link a record owner to a record in a published data table, to a sensitive attribute in a published data table, or to the published data table itself. They are called record linkage, attribute linkage, and table linkage, respectively [3].

The second category tries to achieve the uninformative principle [4]: "The published table should provide the adversary with little additional information beyond the background knowledge. In other words, there should not be a large difference between the prior and posterior beliefs." We consider a probabilistic attack has been performed if there are a large difference between the prior and posterior beliefs to the attacker.

The privacy model we will be focus on falls on the first category. More specifically, the attack model we are providing protection from is record linkage. Our anonymization technique tries to prevent linkage of the records in a published data table to its record owners that leads to identity disclosure.

2.3 Attack Model

Record linkage. In a record linkage attack, some value of an attribute only identifies a small amount of records in a released table. If the victims' QID(quasi-identifier, a set of attributes that could possibly identify record owners) matches them, then the attacker is able to link the victim to the group of records, which narrows down the possibilities for the attacker to find the victim's record. With some additional background knowledge, it becomes possible for the attacker to accurately identify the record that belongs to the victim from those records.



Job	Age	Disease
Engineer	40	Diabetes
Singer	29	Flu
Pianist	48	HIV
Accountant	48	Heart Disease
Pianist	36	HIV
Engineer	36	Diabetes
Singer	35	Diabetes

Figure 1

Example 1. Figure 1.1 is a table of patient records that a hospital wants to publish for research purposes. If an attacker knows that his friend *Bob* is among the patient, with background knowledge of *Bob*'s age is 48 and his job is *pianist*, the attacker can link Bob to the third record, and knows that Bob is a *HIV* patient.

2.4 Anonymization Operations

If a table to be published does not meet a specified privacy requirement, then the table need to be modified to meet the requirement before publishing. A series of anonymization operations are performed to modify the table. There are several types of anonymization operation, here are a few examples: generalization, suppression, anatomization and perturbation.

Generalization. Generalization replaces some value of an attribute with a more "general" value in the taxonomy of that attribute. For example, in Figure 1.1, the job *singer* and *pianist* can be replaced with a more general value *musician*. The reverse operation of generalization is called *specialization*.

Suppression. Suppression removes some values, or replaces some values with a special value (such as asterisk '*'), to hide the content of those specific values or all values of a column.

Anatomization. Anatomization does not change the value of the QID (quasi-identifiers) nor the sensitive attribute, instead it separates them. This approach publishes the data of the QID and the sensitive attribute in two separate tables, with a common attribute that links both tables together. *Disassociation* extends this approach by separating terms of the original data.

Perturbation. This approach focuses on preserving statistical information. The idea is to replace the original data values with some synthetic data values, so the statistical information does not differ too much compared to the original data. Since the perturbated data does not correspond to any real-life individuals, attacker cannot perform linkage attack on the published data. However, the published "synthetic" records could be meaningless to some recipients.

2.5 *k*-Anonymity

k-Anonymity is an approach proposed by Samarati and Sweeney [5] to provide protection against record linkage through QID. A database is a table with *n* rows and *m* columns. A table is *k*-anonymous if one record in the table has some value *qid*, there are at least *k*-1 other records that hold the same value *qid.* In a k-anonymous, each record is indistinguishable from at least *k*-1 other records in terms of QID. So, the probability of an attacker linking an individual through QID is at most 1/k [3].

The common methods for achieving *k*-anonymous is generalization and suppression.

Figure 2 shows an example of *k*-anonymous table using generalization and suppression. Let k=2, and the quasi-identifiers are {Age, State}. The left table is the table with original records, and the right table is the *k*-anonymous table.

ID	Age	State	Disease
R1	58	New York	HIV
R2	39	Texas	Breast Cancer
R3	17	Florida	Flu
R4	34	New York	HIV
R5	13	Alabama	Flu
R6	55	Utah	Heart Disease

ID	Age	State	Disease
R1	$50 \leq Age < 60$	*	HIV
R2	$30 \leq Age < 40$	*	Breast Cancer
R3	$Age < 20$	*	Flu
R4	$30 \leq Age < 40$	*	HIV
R5	$Age < 20$	*	Flu
R6	$50 \leq Age < 60$	*	Heart Disease

Figure 2: Example of *k*-anonymous table using generalization and suppression

Assuming the data is intended to be published to a recipient for the purpose of finding correlations between age and disease, the anonymized table shown in figure 2 should be able to satisfy the intended purpose. In the right table, the ‘age’ attribute values are replaced with more “general” values, replacing specific age value with an age range of 10. While modifying its value, but maintaining its original meaning. The ‘state’ attribute values are suppressed and replaced with ‘*’, because ‘state’ attribute serves no value for the intended purpose of the table, and additionally most of the values in ‘state’ attribute do not have *k* records that holds the same value.

There are several issues with *k*-anonymity. In many real world application, generalization and suppression would often remove valuable information in the data, which results in huge information loss. *k*-Anonymity is also vulnerable to background knowledge attack, by having some background knowledge, the attacker can narrow down the possible records and break *k*-anonymity. For example, Figure 3 shows a *k*-anonymized table where k=2 and the quasi-identifiers are {Age, State, Gender}. Even if the table is *k*-anonymous, if an attacker knows that Bob participates in this table and has background knowledge of Bob is a 58 years old man, then the attacker can narrow it down to R1 and knows that Bob has HIV.

ID	Age	State	Disease	Gender
R1	$50 \le Age < 60$	*	HIV	Male
R2	$30 \le Age < 40$	*	Breast Cancer	Female
R3	$Age < 20$	*	Flu	Male
R4	$30 \le Age < 40$	*	HIV	Male
R5	$Age < 20$	*	Flu	Female
R6	$50 \le Age < 60$	*	Heart Disease	Female

Figure 3

2.6 k^m-Anonymity

k^m-Anonymity is a new version of k-anonymity guarantee proposed by Terrovitis et al. [6]. The anonymization model is proposed for transactional databases. It guarantees that an attacker, who has background knowledge of up to m items, will not be able to distinguish any record from other k-1 records. The formal definition:

An anonymized dataset D^A is k^m-anonymous if no adversary that has a background knowledge of up to m terms of a record can use these terms to identify less than k candidate records in D^A. [1]

In this project, we aim to achieve k^m-Anonymity through *anatomization* instead of *generalization* as proposed in [6]. Unlike k-anonymity problem in relational database, there is no fixed, or well-defined set of quasi-identifier attributes and sensitive data. Additionally, the transactions have variable length and high dimensionality, compared to relational database that only have fixed set of attributes. Figure 4 shows an example of a k^m-Anonymous table, where k = 2, and m = 2.

ID	Web Search Query
R1	basketball, apple, tea, headphone
R2	starbucks, headphone, apple, tea
R3	basketball, starbucks, headphone
R4	basketball, apple, starbucks
R5	tea, basketball, starbucks

Figure 4: 2^2-Anonymous table (k^m-anonymity)

Even if an attacker has a background knowledge of m (2) items in this case, there are at least k (2) records that contain them. For example, if an attacker knows Bob searched 2 terms {basketball, headphone}, the attacker can only narrow it down to R1 and R3, but not able to accurately identify which record belongs to Bob.

3. Specification and Design

The implementation is developed in Python 3.7.5, and the program will require the installation of two Python library: pandas and mlxtend.

A user interface is implemented for the users to choose the dataset they want to run the program with. There will be 4 datasets available to choose, and they will be introduced in the later section.

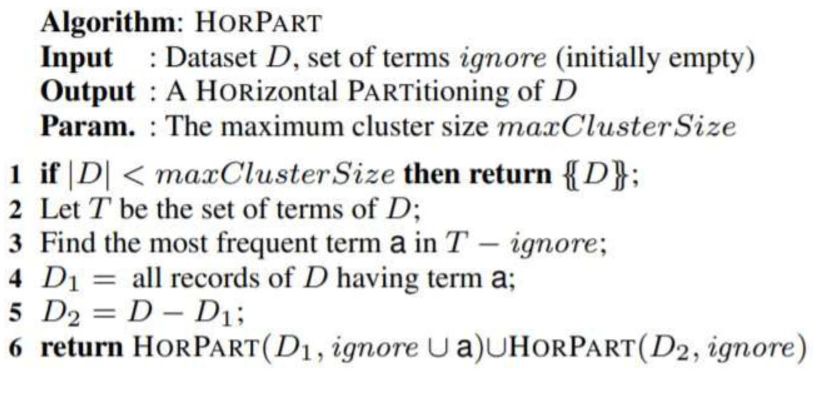
Disassociation algorithm has three stages: horizontal partitioning, vertical partitioning, and refining stage. It transforms the original records into smaller and disassociated sub-records.

3.1 Structures

There is a "structures" python file that contains all the class object that will be used in the disassociation algorithm. There will be three class objects: Record, Cluster, and RecordChunk, which respectively represents record, cluster and record chunk. The object's methods will be defined in the class.

3.2 Horizontal Partitioning

Records of the original dataset D are being grouped into clusters according to their similarity of their contents. Horizontal partitioning essentially transforms a dataset into multiple smaller and independent clusters. The pseudo-code of horizontal partitioning is shown in Figure 5.



```
Algorithm: HORPART
Input    : Dataset $D$, set of terms $ignore$ (initially empty)
Output   : A HORizontal PARTitioning of $D$
Param.   : The maximum cluster size $maxClusterSize$
1 if $|D| < maxClusterSize$ then return $\{\{D\}\}$;
2 Let $T$ be the set of terms of $D$;
3 Find the most frequent term a in $T - ignore$;
4 $D_1 =$ all records of $D$ having term a;
5 $D_2 = D - D_1$;
6 return HORPART($D_1$, $ignore \cup$ a)$\cup$HORPART($D_2$, $ignore$)
```

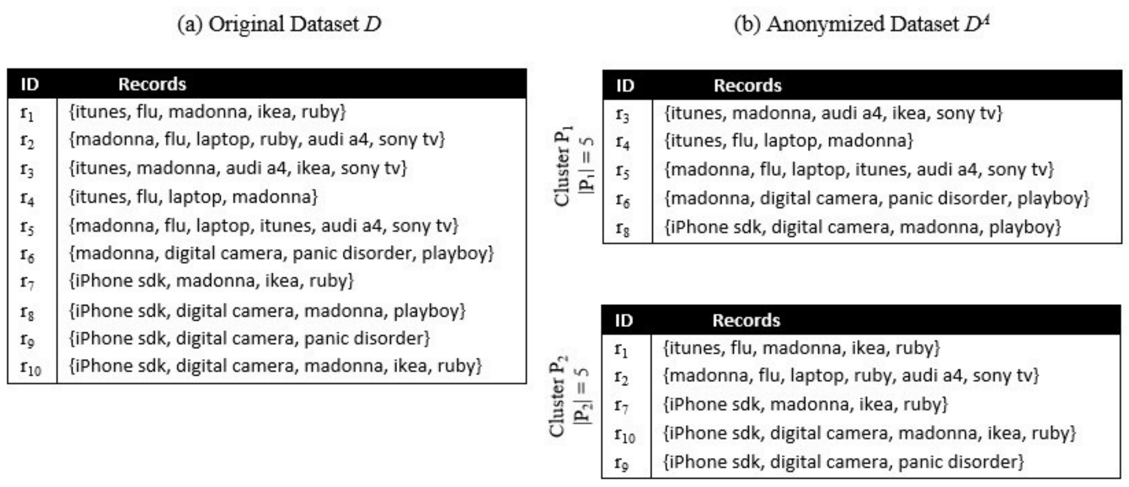
Figure 5: Pseudo-code of Horizontal Partitioning [1]

Horizontal partitioning (HorPart) is a function that split the dataset into clusters of records. The maximum cluster size is defined by the parameter *maxClusterSize*, the final output of the horizontal partitioning algorithm will produce numbers of clusters contains of records less than *maxClusterSize.* The algorithm takes two input: dataset *D*, and a set of terms *ignore*.

There is a set of terms *ignore* that is taken as input in this algorithm as shown in Figure 5. The set *ignore* contains the most frequent terms of accumulated from each partitioning, and the set is unique to each HorPart, it is initially empty.

The algorithm first check if the size of the original dataset *D* is smaller than *maxClusterSize*, if it is then nothing will be done. Otherwise, *D* will be split into two partition (D_1 and D_2). D_1 is the partition with all the records that contains the term *a* (the most frequent term in *D* after excluding all terms in set *ignore*), and D_2 is the remaining records in *D*. If D_1 has a size greater than or equals to *maxClusterSize*, then it will be horizontal partitioned again, the same goes for D_2. Note that D_1 is later partitioned with the *ignore* set that includes *a*, but not for D_2. If D_1 has a size that are smaller than *maxClusterSize*, then the partition D_1 forms a cluster, and the same goes for D_2. HorPart will be applied recursively to each partition until the partition size is smaller than *maxClusterSize*.

However, there is one thing that the paper [1] does not take into account in HorPart, which is what happen if the partition has a size that is smaller than *k.* If HorPart is performed according to the pseudo-code, then there could be clusters with a size that are smaller than *k*, which is problematic because the clusters could not possibly achieve k^m*-anonymity* if there is no *k* items in the cluster. A solution that I come up with is to put all the records of those partitions with the size smaller than *k* into a list of *rejectedRecords*, and later apply HorPart to it. This will ensure that all the clusters have the size less than *maxClusterSize*, at the same time greater than or equals to *k*. If *rejectedRecords* only consists of less than *k* records, loop through each cluster to find a cluster that has the capacity to fit all the records in *rejectedRecords*, and put them in it, this can avoid the scenario of the program endlessly horizontal partitioning *rejectedRecords* if it happens to have less than *k* records.



(a) Original Dataset D

ID	Records
r_1	{itunes, flu, madonna, ikea, ruby}
r_2	{madonna, flu, laptop, ruby, audi a4, sony tv}
r_3	{itunes, madonna, audi a4, ikea, sony tv}
r_4	{itunes, flu, laptop, madonna}
r_5	{madonna, flu, laptop, itunes, audi a4, sony tv}
r_6	{madonna, digital camera, panic disorder, playboy}
r_7	{iPhone sdk, madonna, ikea, ruby}
r_8	{iPhone sdk, digital camera, madonna, playboy}
r_9	{iPhone sdk, digital camera, panic disorder}
r_{10}	{iPhone sdk, digital camera, madonna, ikea, ruby}

(b) Anonymized Dataset D^A

Cluster P_1 $|P_1| = 5$

ID	Records
r_3	{itunes, madonna, audi a4, ikea, sony tv}
r_4	{itunes, flu, laptop, madonna}
r_5	{madonna, flu, laptop, itunes, audi a4, sony tv}
r_6	{madonna, digital camera, panic disorder, playboy}
r_8	{iPhone sdk, digital camera, madonna, playboy}

Cluster P_2 $|P_2| = 5$

ID	Records
r_1	{itunes, flu, madonna, ikea, ruby}
r_2	{madonna, flu, laptop, ruby, audi a4, sony tv}
r_7	{iPhone sdk, madonna, ikea, ruby}
r_{10}	{iPhone sdk, digital camera, madonna, ikea, ruby}
r_9	{iPhone sdk, digital camera, panic disorder}

Figure 6: Horizontal Partitioning Example (*maxClusterSize* = 6, *k*=2)

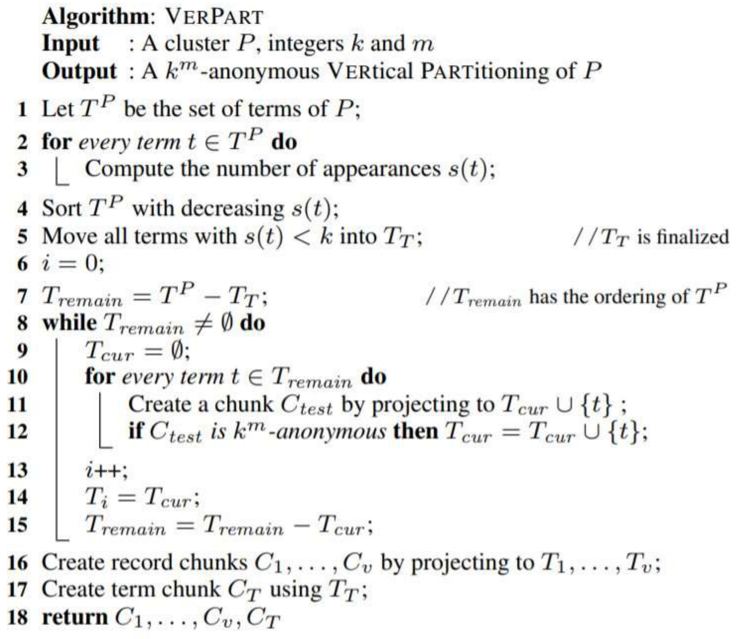
Figure 6 shows an example of horizontal partitioning. The most frequent term in the original dataset *D* is “madonna”, which has 9 records containing it. The algorithm first partitions them into two group. The first group containing the 9 records that contain the term “madonna”, and the second group only contains r_9 because it does not contain “madonna”. Since the first group has a size larger than *maxClusterSize*(=6).

(6), so the first group is partitioned again into two groups, one group containing records that contain the most frequent term other than “madonna” (note that “madonna” is added into the *ignore* set for this group, so it will be ignored when looking for most frequent term in the group), which is “ruby”, since only 4 records contain the term “ruby” and it is smaller than *maxClusterSize*, these records are put into a cluster P_2; the other group contains the remaining 5 records that does not contain “ruby”, since it has a size is smaller than *maxClusterSize* as well, these records are put into a cluster P_1. Now there is only one group left that is not distributed into a cluster, which only contains 1 record r_9, the record size is smaller than *maxClusterSize* but it is also smaller than *k*(=2), so it cannot form its own cluster. Instead we go through each cluster to find a cluster that can fit this group in, which is cluster P_2, because P_1 already has the max size of 5 ($|P| <$ *maxClusterSize*).

3.3 Vertical Partitioning

Vertical partitioning let term combinations that appear many times in a cluster stay together and disassociates terms that create infrequent, thus, identifying combinations. The purpose is to hide the fact that these terms appear together in the same record, and prevent an attacker to easily identify a record with infrequent term combinations.

Each cluster is vertical partitioned into two types of chunks, record chunks and term chunks. Record chunks contain subrecords of the original dataset, and they are k^m*-anonymous*. Term chunks contain terms that appear in a cluster, but are excluded from the record chunks. A term chunk is just a set of terms. Each cluster can have 0 or more records chunks, but only have exactly one term chunk. The pseudo-code of vertical partitioning is shown in Figure 7.



```
Algorithm: VERPART
Input   : A cluster $P$, integers $k$ and $m$
Output  : A $k^m$-anonymous VERtical PARTitioning of $P$
Let $T^P$ be the set of terms of $P$;
for every term $t \in T^P$ do
    Compute the number of appearances $s(t)$;
Sort $T^P$ with decreasing $s(t)$;
Move all terms with $s(t) < k$ into $T_T$;          //$T_T$ is finalized
$i = 0$;
$T_{remain} = T^P - T_T$;          //$T_{remain}$ has the ordering of $T^P$
while $T_{remain} \neq \emptyset$ do
    $T_{cur} = \emptyset$;
    for every term $t \in T_{remain}$ do
        Create a chunk $C_{test}$ by projecting to $T_{cur} \cup \{t\}$ ;
        if $C_{test}$ is $k^m$-anonymous then $T_{cur} = T_{cur} \cup \{t\}$;
    $i$++;
    $T_i = T_{cur}$;
    $T_{remain} = T_{remain} - T_{cur}$;
Create record chunks $C_1, \ldots, C_v$ by projecting to $T_1, \ldots, T_v$;
Create term chunk $C_T$ using $T_T$;
return $C_1, \ldots, C_v, C_T$
```

Figure 7: Pseudo-code of vertical partitioning [1]

Vertical partitioning is applied independently to each cluster. It takes a cluster and integers *k* and *m* as input. First, it computes the support (number of appearances) for each term that appears in the cluster, and sort them in a descending order. Every term that has support less than *k* are inserted into term chunk T_T, as they are infrequent and record chunks are unable to achieve k^m-anonymous with these terms.

T_{remain} is the list of remaining terms that are not assigned to term chunk in descending order, these terms will participate in record chunk. The algorithm computes sets of terms $T_1,\ldots,T_v$

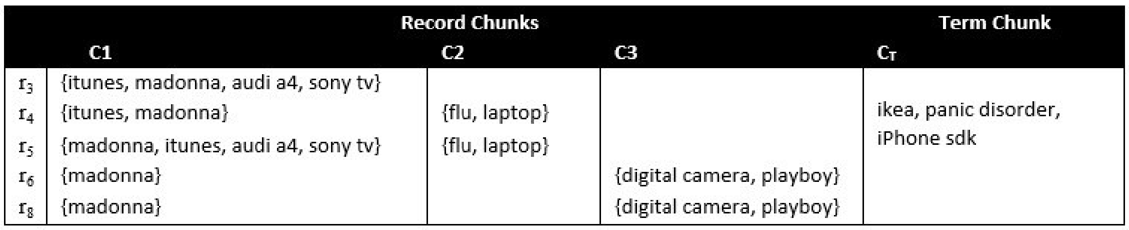
(while loop of line 8 to 15), T_i ($1 \leq i \leq v$) is a set of terms that participates in a record chunk C_i. In the while loop, T_{Cur} is a set that contains the terms that will be assigned to the current set of terms. For each term t in T_{remain}, t will be inserted into a test chunk C_{test} ($T_{Cur} \cup \{t\}$), if C_{test} is k^m-anonymous then t will be added into T_{Cur}, but the first execution of this for loop will always add t into T_{Cur} since $\{t\}$ is k^m-anonymous. To check if C_{test} is k^m-anonymous, we need to check all the combinations of the size up to m and make sure they appear in at least k records, if any of the combinations does not fulfil the requirement above, then t is rejected and will not participate in T_{Cur}. For example, let k=2, m=2, C_{test} = {p,q,r,}, then it will generate the following combinations {{p},{q},{r},{p,q},{p,r},{q,r},{p,q,r}}. However if we want to check if it's k^m-anonymous, we do not have to check all of the combinations, we only need to check the combinations of the size up to m, and we do not have to check the single items ({{p},{q},{r}}) too because they are guarantee to have k support, so we only have to check if every combination in this set {{p,q},{p,r},{q,r}} appears in at least k records.

After finishing testing every term in T_{remain}, the set of terms in T_{Cur} will be assigned to T_i, and all the terms in T_{Cur} will be removed from T_{remain}, then the algorithm moves on to the next set T_{i+1}.

Finally, the algorithm creates record chunks $C_1,\ldots,C_v$ by using $T_1,\ldots,T_v$ and the term chunk C_T using T_T. Each record chunk C_i contains subrecords with terms in T_i. For instance, cluster P_1 has 3 records: {p,q,x,y,z}, {p,q,z}, {p,q,y,z}, if T_i = {y,z}, then C_i contains 3 subrecords: {y,z}, {z},{y,z}.

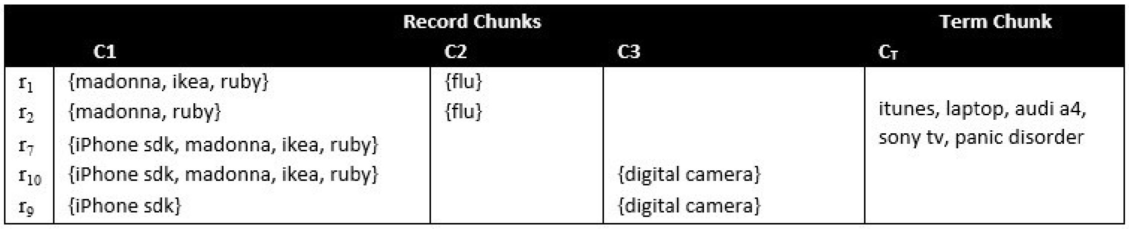
Figure 8 shows the vertical partitioning of the dataset in Figure 6. Take cluster P_1 for example, every term combination in C_1, C_2, C_3 appears in at least k records. {ikea, panic disorder, iPhone sdk} is placed in the term chunk, because they have less than k support.

Cluster P_1



	Record Chunks			Term Chunk
	C1	C2	C3	C_T
r_3	{itunes, madonna, audi a4, sony tv}			
r_4	{itunes, madonna}	{flu, laptop}		ikea, panic disorder, iPhone sdk
r_5	{madonna, itunes, audi a4, sony tv}	{flu, laptop}		
r_6	{madonna}		{digital camera, playboy}	
r_8	{madonna}		{digital camera, playboy}	

Cluster P_2



	Record Chunks			Term Chunk
	C1	C2	C3	C_T
r_1	{madonna, ikea, ruby}	{flu}		
r_2	{madonna, ruby}	{flu}		itunes, laptop, audi a4, sony tv, panic disorder
r_7	{iPhone sdk, madonna, ikea, ruby}			
r_{10}	{iPhone sdk, madonna, ikea, ruby}		{digital camera}	
r_9	{iPhone sdk}		{digital camera}	

Figure 8: Vertical Partitioning Example (*k*=2, *m*=2)

3.4 Refining

Refining is the final stage of the disassociation algorithm, however at the end of vertical partitioning, the dataset is already disassociated. Refining is the stage that improve the quality of the result while keeping the anonymity guarantee. In this stage, it focus on the terms in the term chunks. A term T_1 can appear in term chunks in multiple clusters because their support in those clusters are low, e.g. Figure 9 shows the clusters produced by vertical partitioning if *k*=3. The term *ikea* is in the term chunk in both cluster because its support is low, but the support of the term *ikea* considering both cluster P_1 and P_2 is now low enough to jeopardize user privacy.

To address such issues, [1] proposes *joint clusters* that allows different cluster to share record chunks. Given a set T^S of refining terms (e.g. ikea), which appear in the term chunks of two or more clusters (e.g. P_1 and P_2), a joint cluster is defined by **(a)** creating one of more shared chunks after projecting records of the initial clusters to T^S and **(b)** removing all T^S terms from the term chunks of the initial clusters [1]. Figure 10 shows a joint cluster, created by joining clusters P_1 and P_2 of Figure 9, based on T^S = {audi a4, laptop, panic disorder, sony tv, ikea}.

Cluster P_1

	Record Chunks			Term Chunk
	C1	C2	C3	C_T
r_2	{madonna, ruby}			{audi a4, laptop, panic disorder, sony tv, flu, playboy, ikea}
r_6	{madonna}	{digital camera}		
r_7	{madonna, ruby}		{iPhone sdk}	
r_8	{madonna}	{digital camera}	{iPhone sdk}	
r_{10}	{madonna, ruby}	{digital camera}	{iPhone sdk}	

Cluster P_2

	Record Chunks	Term Chunk
	C1	C_T
r_1	{madonna, itunes, flu}	audi a4, laptop, panic disorder, sony tv, digital camera, iPhone sdk, ikea, ruby
r_3	{madonna, itunes}	
r_4	{madonna, itunes, flu}	
r_5	{madonna, itunes, flu}	
r_9		

Figure 9: Vertical Partitioning of D^A (k=3, m=2)

	Record Chunks			Term Chunk	Shared Chunk
	C1	C2	C3	C_T	
Cluster P_1					
r_2	{madonna, ruby}			flu, playboy	{audi a4, laptop, sony tv}
r_6	{madonna}	{digital camera}			{panic disorder}
r_7	{madonna, ruby}		{iPhone sdk}		{ikea}
r_8	{madonna}	{digital camera}	{iPhone sdk}		
r_{10}	{madonna, ruby}	{digital camera}	{iPhone sdk}		{ikea}
Cluster P_2					
r_1	{madonna, itunes, flu}			digital camera, iPhone sdk, ruby	{ikea}
r_3	{madonna, itunes}				{audi a4, ikea, sony tv}
r_4	{madonna, itunes, flu}				{laptop}
r_5	{madonna, itunes, flu}				{audi a4, laptop, sony tv}
r_9					{panic disorder}

Figure 10: Disassociation with Shared Chunk

We may form a higher-level joint cluster by joining joint clusters with simple clusters. A simple cluster is the cluster produced by vertical partitioning, e.g. cluster P_1. The output of vertical partitioning is a set of k^m-anonymous clusters *P*. Refining improves the quality of *P* by iteratively constructing joint clusters until no more improvement can be made. To perform refining stage on *P*, a naïve way of doing includes computing the information loss (e.g. using the metrics in section 5) for all the possible combinations of joint clusters and choose the combination with the best quality (i.e. least information loss). However, that would be too inefficient, and probably will take too much time and memory space, so a refining criterion is defined in [1]. Let us consider two clusters J_1 and J_2, these clusters are joined into cluster J_{new} if the following inequality holds:

$$\frac{s(t_1) + \cdots + s(t_n)}{|J_{new}|} \geq \frac{u_1 + \cdots + u_m}{|P_1| + \cdots + |P_m|}$$

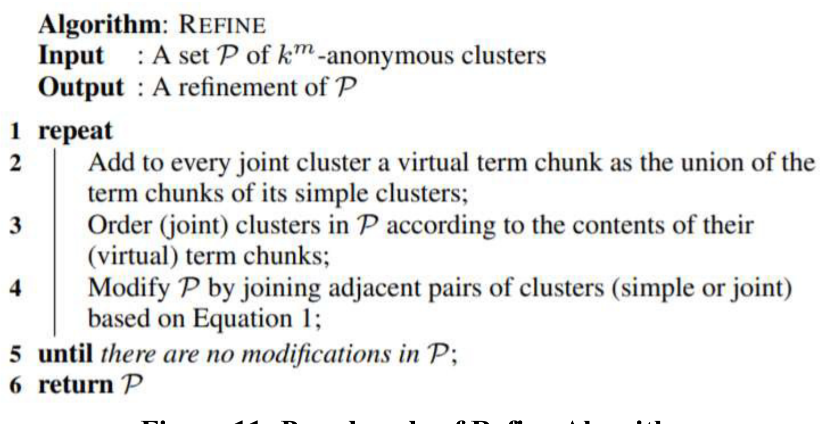
where **(a)** $t_1 ,\ldots, t_n$ are the refining terms T^S, (b) $s(t_1),\ldots, s(t_n)$ are the supports of $t_1 ,\ldots, t_n$ respectively in the shared chunks of J_{new}, (c) $P_1,\ldots, P_m$ are the simple clusters of J_1 and J_2 that contain $t_1 ,\ldots, t_n$ and (d) $u_1,\ldots, u_m$ are the number of terms $t_1 ,\ldots, t_n$ that appear in the term chunk of each of clusters $P_1,\ldots, P_m$ respectively [1]. For example, if J_1 and J_2 are clusters P_1 and P_2 of Figure 9, and J_{new} is the joint cluster of Figure 10, then the refining terms would be {audi a4, laptop, panic disorder, sony tv, ikea}. We will have:

$$\frac{s(audi\ a4)+s(laptop)+s(pan\quad disorder)+s(sony\ tv)+s(ikea)}{|Jnew|} \geq \frac{u1+u2}{|P1|+|P2|}$$

$$\Rightarrow \frac{3+3+2+3+4}{10} \geq \frac{5+5}{5+5} \qquad \rightarrow \frac{15}{10} \geq \frac{10}{10}$$

Thus J_1 and J_2 will be replaced by J_{new}. The left part of the equation calculates the probability of assigning one of $t_1 ,\ldots, t_n$ to the records of the joint cluster J_{new}, while the right part of the equation calculates the probability of assigning one of $t_1 ,\ldots, t_n$ to the records cluster J_1 and J_2 [1].

However, even with the equation provided, it is still computationally infeasible to look at all the combinations of clusters and find the best one. A REFINE algorithm is defined in [1] that merges only two existing clusters at a time to form a new joint cluster. Figure 11 shows the pseudocode of the REFINE algorithm.



```
Algorithm: REFINE
Input   : A set $\mathcal{P}$ of $k^m$-anonymous clusters
Output  : A refinement of $\mathcal{P}$
1 repeat
2     Add to every joint cluster a virtual term chunk as the union of the
      term chunks of its simple clusters;
3     Order (joint) clusters in $\mathcal{P}$ according to the contents of their
      (virtual) term chunks;
4     Modify $\mathcal{P}$ by joining adjacent pairs of clusters (simple or joint)
      based on Equation 1;
5 until there are no modifications in $\mathcal{P}$;
6 return $\mathcal{P}$
```

Figure 11: Pseudocode of Refine Algorithm

Due to the COVID-19 situation, face-to-face meeting is cancelled. It makes explanation and discussion on how the refining stage works and how it should be implemented more difficult for me and my supervisor. After careful consideration, my supervisor kindly suggests that I do not implement the refining stage in this project.

4. Implementation

We will go into details of the implementation of the disassociation algorithm using Python in this section, including the design, explanation of the code and challenges of implementation.

4.1 User Interface

A simple user interface is implemented which lets the user choose which dataset to run the disassociation on. After choosing one of the given 4 dataset, the program will run itself, and perform the disassociation algorithm and the evaluation methods, then it will output the time spent on performing each algorithm and the evaluation metrics results. I try to keep the user interface simple and easily understandable to avoid confusion, at the same time showing all the work that I have done.

4.2 Program Design

All the object class that represents something such as record, cluster etc is under "structures.py". Every method that relates to horizontal partitioning are in the class *HorPart*. Similarly, every method that relates to vertical partitioning are in the class *VerPart*. Python allows multiple classes to be defined under one python file helps to group similar classes and methods together. Evaluation methods are all defined under "evaluation.py" and data reconstruction method is defined in the class *DatasetReconstruct*. At the end, we just use the main method to import every classes and run all the necessary methods to retrieve the information we are interested.

4.3 Structures

Under "structures.py" file, I created three object classes (*Record*, *Cluster* and *RecordChunk*) representing record, cluster and record chunk respectively. Each class contains relevant setter and getter method, as well as string object that returns useful information.

4.3.1 Record

A class Record consists of a unique identifier *recID* and its record values *recVal*. A record's *recID* is given when parsing the dataset into records, e.g. the first record is given the *recID* "R1", the second record's *recID* is "R2", the 300th record's *recID* is "R300" etc. It is implemented because it's easier to count and differentiate each record for a human in this way. The record values are stored in a Set, the reason of it is because it is significantly faster to check if an item is in a set than a List, because time complexity for querying a Set is O(1) compare to a List O(n). The order of the items in a record does not matter, and here I assume there will be no duplicate items in a record. Since there are many checking operations performed on the record values, hence using Set over List for storing record values gives us a better performance.

4.3.2 Record Chunk

Similarly, a class *RecordChunk* consists of a unique identifier *recordChunkID* and its subrecords value *recordChunkValue,* each record chunk ID starts with "C". The class has setter and getter method for both variables. A string method that returns the record chunk ID and its subrecords. The record chunk value is a list of Record objects, it is implemented as a List because it needs to keep the order of the subrecords, while also allow multiple identical subrecords.

4.3.3 Cluster

A class *Cluster* consists of a unique identifier *clusterID,* which starts with “P”, it also consists of a list of records *records*, a list of RecordChunk that associates with the cluster *recordChunks,* and a set of term chunk that associates with the cluster *termChunk*. Its String method will return the records that are in the cluster, the record chunks and their subrecords and its term chunk. We use a List to represent the records and record chunks in the cluster, so it preserves the order of the records, it is also slightly faster to iterate over the records than using set. The term chunk is a Set because the order does not matter and there should not be any duplicate in the term chunk, and the time complexity for querying a Set is O(1), so it would be faster to check if an item is in the term chunk than using a List.

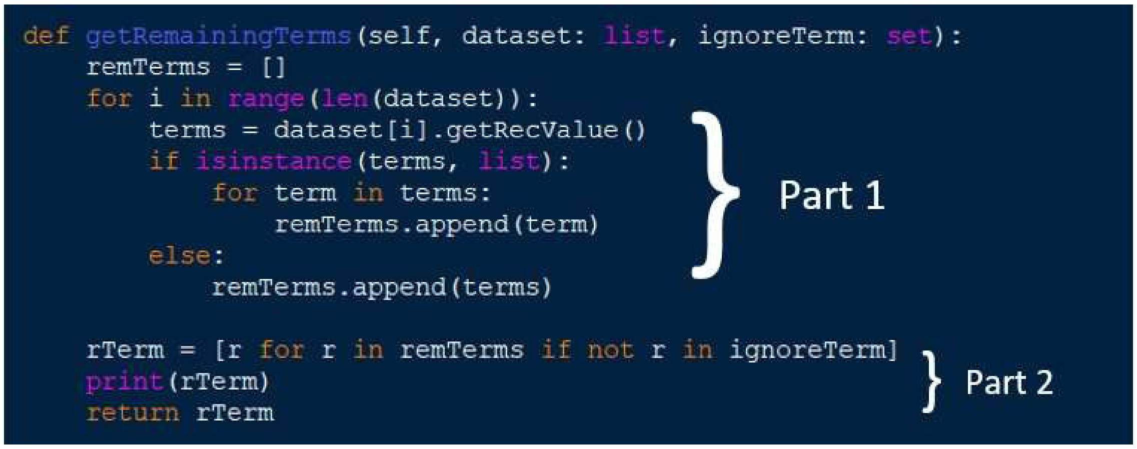
4.4 Horizontal Partitioning

The entire horizontal partitioning algorithm is implemented inside the class *HorPart*. The class expects 2 arguments (*maxClusterSize* and *k*) when creating a *HorPart* object. *maxClusterSize* is needed in horizontal partitioning because horizontal partitioning is the algorithm that partitions the dataset into clusters, hence the maximum size of cluster needed to be defined. *k* needed to be defined too, in order to check and prevent clusters size smaller than *k*.

When the class is initialized, the constructor method will initialize 7 attributes. *k* and *maxClusterSize* has already been explained in section 2 and 3.2 respectively. *clustersList* is the list of clusters produced by horizontal partitioning, this is the output of horizontal partitioning. It is a list because I consider the order of cluster important and there are a good amount of iteration of its content in the program. *clusterCounter* is a just counter for the cluster, it is used for assigning *clusterID,* the clusterID starts with “P1”. *queueListData* and *queueListIgnore* are a queue implemented in a list, a detail explanation will be given below. Finally, *rejectedRecords* is also explained in section 3.2, a list is used for the performance of iteration.

4.4.1 getRemainingTerms

getRemainingTerms is a method that return the remaining terms in a dataset after removing all the terms that appears in the *ignore* set. It takes 2 argument, the *dataset*, which is a list of records and *ignore*, a set of strings to be ignored. The code is shown in Figure 12.



```
def getRemainingTerms(self, dataset: list, ignoreTerm: set):
    remTerms = []
    for i in range(len(dataset)):
        terms = dataset[i].getRecValue()
        if isinstance(terms, list):
            for term in terms:
                remTerms.append(term)
        else:
            remTerms.append(terms)

    rTerm = [r for r in remTerms if not r in ignoreTerm]
    print(rTerm)
    return rTerm
```

Figure 12: getRemainingTerms code

Part 1 iterate over each term of each record in the dataset and append every term in the record into the list *remTerms*. Part 2 uses list comprehension to filter out all the terms in *ignoreTerm* and return the remaining terms that are stored in the list *rTerm*. The reason of using a list for the remaining term and store all the duplicate terms, instead of using a set and only store once for each term and ignoring the duplicate, is because *rTerm* will be used to count the frequency of each term in the future. List comprehension is used for filtering because it's one of the most efficient way of performing this operation.

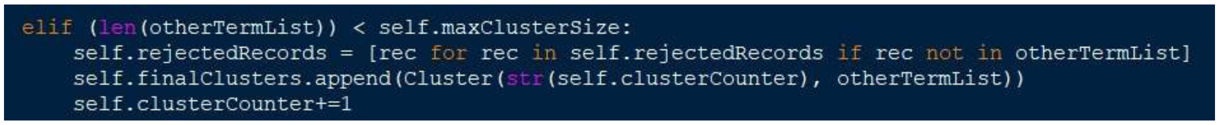
4.4.2 mostFrequentTerm

This method returns the most frequent term of the remaining terms. It uses Dictionary to store each term and its occurrence count. It iterates over the List *remTerms* and updates the occurrence count of each term, then updates the most frequent term if the current term has an occurrence count more than the current most frequent term.

4.4.3 horizonPartition

This method performs horizontal partitioning on the *dataset* just like the algorithm shown in Figure 5. First, it gets the remaining terms, if the remaining terms are empty, that means that the every records in the dataset only contains terms that are previously the most frequent term, in this case, all the records in the dataset are inserted into *rejectedRecords* and the HorPart for this partition ends here.

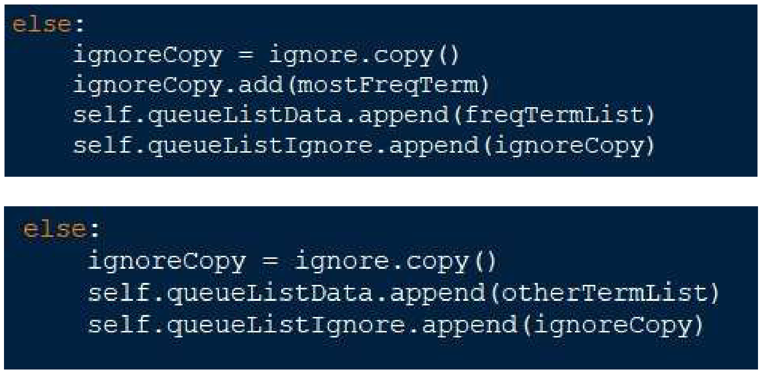
If the remaining terms are not empty, it finds the most frequent term of the dataset, and begin the partitioning. The records that contain the most frequent term will be inserted into a List *freqTermList*, and the others will be inserted into a List *otherTermList*. As mentioned above, if one of the lists has a size smaller than *k*, all its records will be inserted into *rejectedRecords*, for a reapplication of HorPart.



```
elif (len(otherTermList)) < self.maxClusterSize:
    self.rejectedRecords = [rec for rec in self.rejectedRecords if rec not in otherTermList]
    self.finalClusters.append(Cluster(str(self.clusterCounter), otherTermList))
    self.clusterCounter+=1
```

Figure 13: Creating cluster

Figure 13 shows if the list has the size smaller than *maxClusterSize* and greater than *k*, then the records in the list will become a cluster and added into *clustersList.* At the same time, it removes the records that are in this list from the *rejectedRecords*.



```
else:
    ignoreCopy = ignore.copy()
    ignoreCopy.add(mostFreqTerm)
    self.queueListData.append(freqTermList)
    self.queueListIgnore.append(ignoreCopy)
```

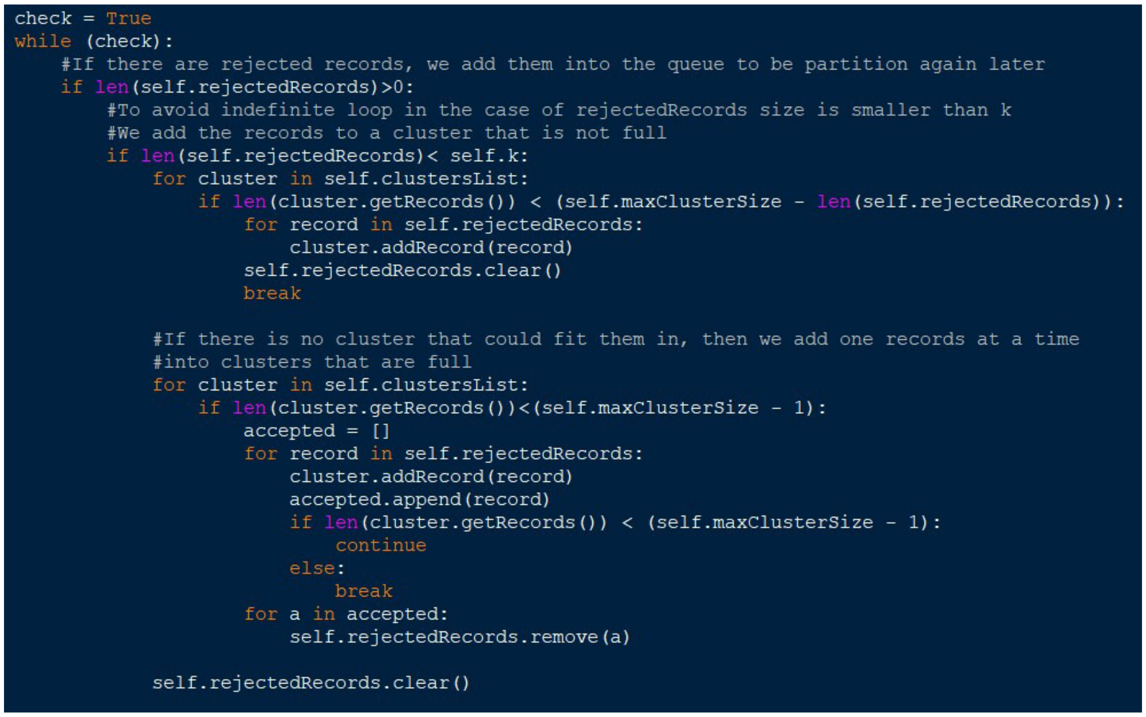
```
else:
    ignoreCopy = ignore.copy()
    self.queueListData.append(otherTermList)
    self.queueListIgnore.append(ignoreCopy)
```

Figure 14: Queueing up larger than k partition

Figure 14 shows what happen if the list has the size greater than *maxClusterSize*. The code above applies to *freqTermList* and the bottom applies to *otherTermList*. It creates a copy of the set *ignore* to prevent alteration. The code above will add the most frequent term to the *ignore* set because the *freqTermList* contains the records that contain the most frequent term. Finally, it appends the list and the *ignore* set to the end of the lists *queueListData* and *queueListIgnore* respectively, the lists acts as a queue, so both of them are inserted into the back of the queue.

4.4.4 horizonPartitioning

This method is the main method of horizontal partitioning. It integrates all the methods and produce the final result of horizontal partitioning. First, it checks if the size of the original dataset is smaller than *maxClusterSize*, if it is then the dataset forms the one and only Cluster. Otherwise, it inserts the dataset and an empty *ignore* Set to the back of the queue (which is empty now). Then, a while loop is performed, and exits under the condition of *rejectedRecords* being empty at the end of the loop. Before that, it enters another while loop with the exit condition being *queueListData* is empty, this is the first full horizontal partitioning, which means that by the end of this loop, the original dataset is fully horizontal partitioned. However, there could be rejected records as mentioned above due to invalid clusters or partitions.



```
check = True
while (check):
    #If there are rejected records, we add them into the queue to be partition again later
    if len(self.rejectedRecords)>0:
        #To avoid indefinite loop in the case of rejectedRecords size is smaller than k
        #We add the records to a cluster that is not full
        if len(self.rejectedRecords)< self.k:
            for cluster in self.clustersList:
                if len(cluster.getRecords()) < (self.maxClusterSize - len(self.rejectedRecords)):
                    for record in self.rejectedRecords:
                        cluster.addRecord(record)
                    self.rejectedRecords.clear()
                    break

            #If there is no cluster that could fit them in, then we add one records at a time
            #into clusters that are full
            for cluster in self.clustersList:
                if len(cluster.getRecords())<(self.maxClusterSize - 1):
                    accepted = []
                    for record in self.rejectedRecords:
                        cluster.addRecord(record)
                        accepted.append(record)
                        if len(cluster.getRecords()) < (self.maxClusterSize - 1):
                            continue
                        else:
                            break
                    for a in accepted:
                        self.rejectedRecords.remove(a)

            self.rejectedRecords.clear()
```

Figure 15: Handling rejected records

As shown in Figure 15, if *rejectedRecords* is not empty by the end of the full horizontal partitioning, then we check if its size is smaller than *k*, if it is then we go through each Cluster and find a Cluster that can fit the records in. However, if there is no cluster that could fit all the records, then we iterate over the clusters again and insert one record at a time, and remove it from *rejectedRecords* after it is inserted into a Cluster. At the extremely unlikely cases of every cluster is full and there are still rejected records left, then these records are abandoned, because all the clusters should be k^m*-anonymous*.

If *rejectedRecords* has a size greater or equals to *k*, then all the records are inserted into the queue (which is empty now) , the *rejectedRecords* will be emptied, and these records will be fully horizontal partitioned again.

When both the queue and *rejectedRecords* are empty by the end of the loop, *clustersList* will be returned as the final result of HorPart.

4.5 Vertical Partitioning

The entire vertical partitioning algorithm is implemented inside the class *VerPart*. The class expects 2 arguments (*k* and *m*) when creating a *VerPart* object. *k* and *m* needed to be defined because the vertical partitioning algorithm will perform a k^m-anonymous check for the test chunk. When the class is initialized, the constructor method will initialize 2 attributes, which are *k* and *m*. Vertical partitioning is only performed on one cluster at a time.

4.5.1 sortSupportTerm

This method sorts the terms in the Cluster with decreasing support and returns a list that contains each term and its support, it accepts a Cluster object as an argument. The method first iterates over the Records in the Cluster, and appends every terms in the records into the List *allTerm*, which is similar to part 1 of Figure 12. Then, it uses a method (Counter([iterable].most_common([n]) imported from Python Library *collections* to count and sort the term support. The method returns a list of all the elements in *allTerm* and their counts from the most common to the least (descending order).

4.5.2 createTermChunk

This method move the terms with less than *k* support to a Set *termChunkSet*, which will be the term chunk for the cluster. It accepts the list of terms support produced by *sortSupportTerm* as argument. It uses the counts in the list to filter the terms with less than *k* support.

4.5.3 kMAnonymous

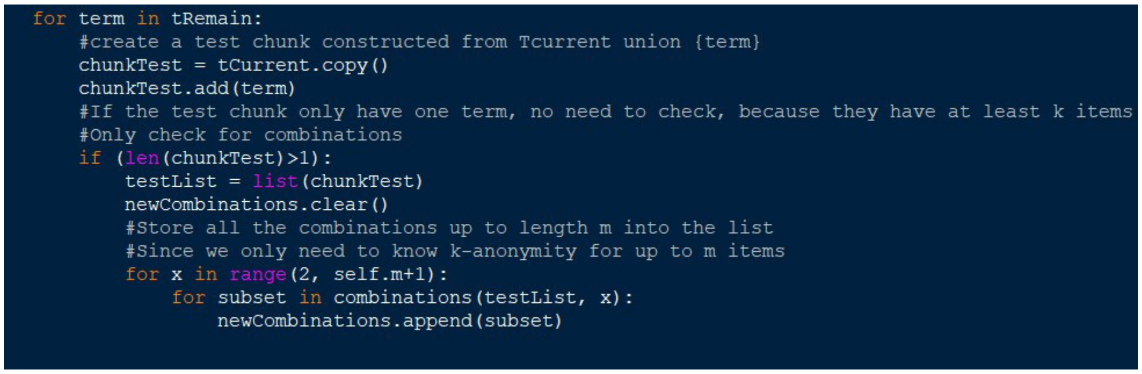
This method checks if the terms in *combinationToCheck* is k^m*-anonymous*, which means that every combination in *combinationToCheck* must appear in at least *k* records. If there is combination that does not meet the requirement, then it return False. The List *combinationToCheck* contains terms combinations up to size *m*. The method iterate over the combinations and check if all the terms in the combination appears in a record, once it finds out that there are *k* records that contain all the term in the combination, it stops the checking for the current combination and move onto the next combination. Once every combinations in *combinationToCheck* pass the check, then it returns True, which means that the terms in the test chunk *chunkTest* are k^m*-anonymous.*

4.5.4 verticalPartition

This method performs the vertical partitioning algorithm from line 8 to line 15 of the pseudocode shown in Figure 7. The List *termSetList* is a list for storing the set of terms $T_1,\ldots,T_v$ that are k^m*-anonymous*. *tRemain* is a list to store the remaining terms T_{remain} , initially contains the remaining terms in the cluster after excluding the terms in term chunk.

The most important task of this method is to generate combinations of terms and check if they are k^m*-anonymous.* We use a module function *combinations()* of a module *itertools* imported from the Python Standard Library to generate combinations. *combinations(iterable, r)* returns combinations of size *r* length The List *newCombinations* is used to store the new combinations generated with the terms in the test chunk *chunkTest*, and *oldCombinations* is used to store the combinations that passed the k^m*-anonymous* check. The reason to store the old combinations is for *newCombinations* is to filter the previously checked combinations, thus reducing the number of combinations to check.

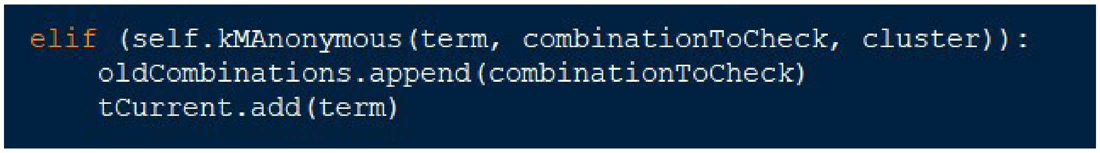
T_{cur} (or *tCurrent* in code) is a Set that stores the terms that will be assigned to T_i. Use the for loop to go through each term in *tRemain*. At the start of each loop, *chunkTest* is assigned a copy of *tCurrent* (use the copy() function to avoid alteration of *tCurrent*), adding the current term.



```
for term in tRemain:
    #create a test chunk constructed from Tcurrent union {term}
    chunkTest = tCurrent.copy()
    chunkTest.add(term)
    #If the test chunk only have one term, no need to check, because they have at least k items
    #Only check for combinations
    if (len(chunkTest)>1):
        testList = list(chunkTest)
        newCombinations.clear()
        #Store all the combinations up to length m into the list
        #Since we only need to know k-anonymity for up to m items
        for x in range(2, self.m+1):
            for subset in combinations(testList, x):
                newCombinations.append(subset)
```

Figure 16: Generating term combinations

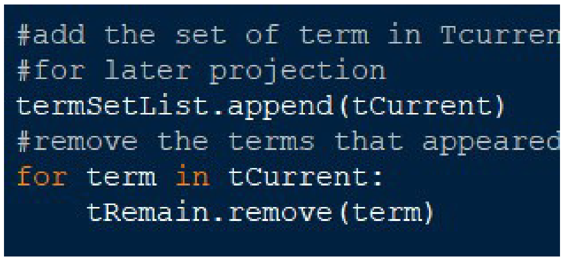
Figure 16 shows the lines of code that generate the term combinations. There is no need to generate combinations if there is only one term in *chunkTest*, so the first execution of the for loop always add a term to *tCurrent*. *chunkTest* is converted to a List because the argument of *combinations*() needs to be an *iterable.* Since we are checking for k^m-anonymity, means that we only need to generate combinations up to size *m,* because k^m-anonymity assumes that an adversary may have background knowledge of up to *m* terms for any record. For example, if m = 3, and there are 4 terms in the test chunk {“A”,”B”,”C”,”D”}. Then we only need to generate combinations of size up to 3 excluding single item combinations, which are {“A”, “B”}, {“A”,”C”}, {“A”,”D”}, {“B”,”C”}, {“B”,”D”}, {“C”,”D”}, {“A”,”B”,”C”}, {“A”,”B”,”D”}, {“A”,”C”,”D”} and {“B”,”C”,”D”}.



```
elif (self.kMAnonymous(term, combinationToCheck, cluster)):
    oldCombinations.append(combinationToCheck)
    tCurrent.add(term)
```

Figure 17: k^m-anonymity check

If the combinations are k^m-anonymous, then these combinations will be inserted into *oldCombinations*, and the current term will be added to *tCurrent*.



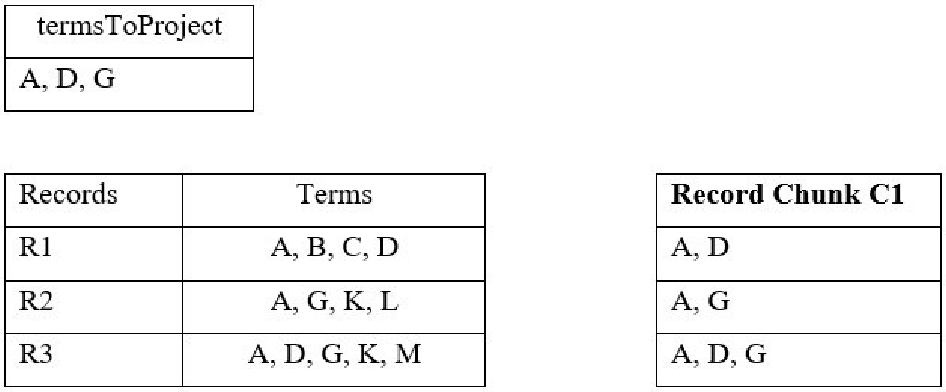
```
#add the set of term in Tcurren
#for later projection
termSetList.append(tCurrent)
#remove the terms that appeared
for term in tCurrent:
    tRemain.remove(term)
```

Figure 18: $T_{remain} = T_{remain} - T_{Cur}$

At the end of the loop, insert the terms in *tCurrent* into *termSetList*, and remove the terms that are in *tCurrent* from *tRemain*.

4.5.5 projectRecordChunk

This method creates a RecordChunk object by projecting the records to the Set of terms *termsToProject* (a set of terms from *termSetList* returned by *verticalPartition* method). The method checks if a record contains the terms in *termsToProject*, and create a new record containing only those terms. Finally adding the new record to the RecordChunk. Figure 19 shows an example of how the records are projected to the terms.



termsToProject
A, D, G

Records	Terms
R1	A, B, C, D
R2	A, G, K, L
R3	A, D, G, K, M

Record Chunk C1
A, D
A, G
A, D, G

Figure 19: Record Chunk Example

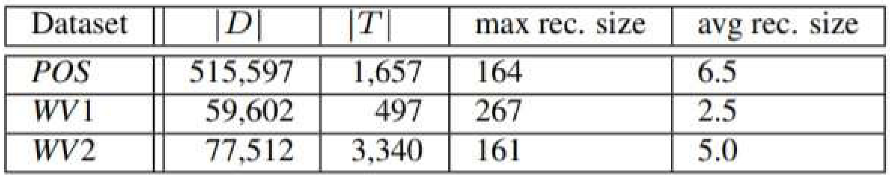
4.5.6 createRecordChunk

This method creates all the record chunks in the cluster and returns them.

5.0 Results and Evaluation

5.1 Dataset

We use the 3 datasets (*POS, WV1* and *WV2)* that are introduced in [1] that are described in Figure 9. Dataset *POS* is a transaction log from an electronics retailer. Datasets *WV1* and *WV2* contain click-stream data from two e-commerce websites, collected over a period of several months [1]. The fourth dataset is a small dataset that only contains 15 records, which can be used for simple and quick testing. As shown in Figure 20, *POS* is the largest dataset with 515,597 records and consists of 1,657 unique terms. WV2 being the second largest dataset and WV1 is the smallest. WV2 has the most unique terms (3,340), and WV1 has the biggest maximum record size (267 items in a record).



Dataset	$\|D\|$	$\|T\|$	max rec. size	avg rec. size
POS	515,597	1,657	164	6.5
*WV*1	59,602	497	267	2.5
*WV*2	77,512	3,340	161	5.0

Figure 20: Dataset

The dataset that I used are all in txt file, *WV1* and *WV2* has a different format than *POS*. *WV1* and *WV2* uses "-1" and "-2" as separator, where "-1" acts like a comma, and "-2" acts as the end of line; while *POS* uses "," to separate items. Since each dataset has different format, I implemented different file handling operation for each dataset.

5.2 Evaluation Parameters

There are 3 parameters that should be considered before performing the disassociation algorithm, which are *k*, *m* and *maxClusterSize*. We vary *k* with the values of 5, 10 ,15 and 20 as these k values are also tested in [1] and we can compare our result to [1] to see the differences. *m* is fixed on 2 as it was mentioned in [1] that the effect of m > 2 is insignificant. It is not mentioned anywhere in [1] what value *maxClusterSize* should be, after some testing, I found *maxClusterSize* = 2*k* to be the best in terms of performance, so I set *maxClusterSize* to be 2*k* in all the evaluations unless stated explicitly otherwise.

5.3 Performance

First, compare our time spent on disassociation algorithm to [1] with the parameter *k*=5, *m*=2, *maxClusterSize*=11. Figure 21 illustrates the performance of the disassociation algorithm in terms of CPU time (in seconds), the left chart is the results achieved by [1], and the right chart is our results.

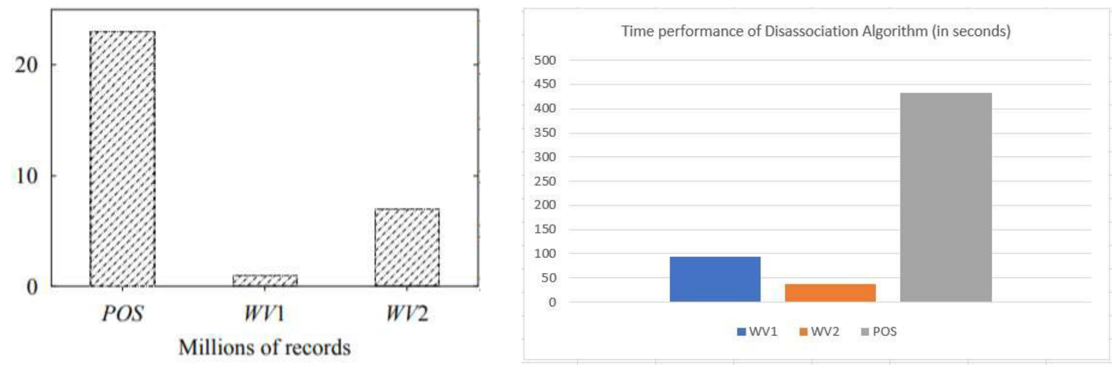
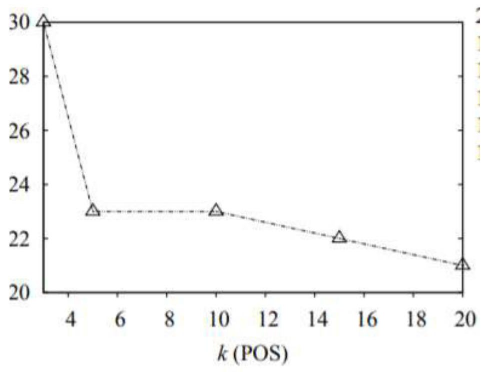


Figure 21: Performance on dataset (left from [1])

The timing is the average timing results over 5 runs of each dataset, a more detailed information can be found in the excel file inside the Appendix folder. Our results are significantly slower than the performance of [1], and even slightly different in characteristics. In the left chart, the performance on dataset *WV1* is the best, however, the best performance of our results is achieved by dataset *WV2*. One of the reason that our performance is much slower than [1] is that the disassociation algorithm is implemented in C++ in [1], while ours are implemented in Python. C++ is compiled, while Python is interpreted, and the interpretation of code is always slower than compilation, which is why Python loses out on performance in this case against C++.

Figure 22 shows the performance of disassociation algorithm with different *k* values (5, 10, 15, 20, *m*=2, *maxClusterSize* = 2*k* +1). The performance in general scale linearly downward as the value of *k* increases, but the effects are mostly insignificant. It is almost unremarkable for *WV2*, however noticeable for a large dataset such as *POS*. There is also a small spike when *k*=10 for *WV1*. Note that the performance on *WV1* is worse than *WV2* until *k* increases to 15, then only the performance is better than *WV2*, which is a larger dataset than *WV1*.



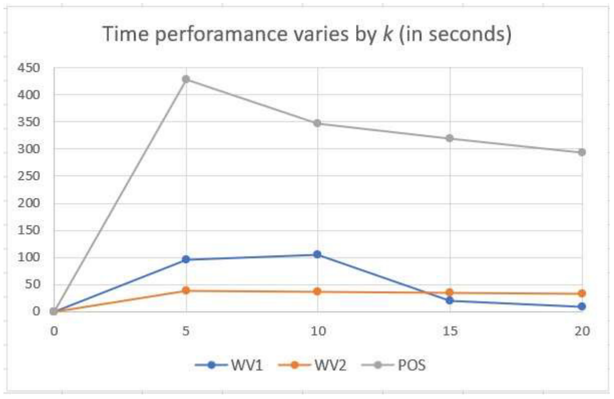


Figure 22: Time performance varies by k (left from [1])

5.4 Relative Error

Relative error is used the measure the relative error in the support of term combinations in the reconstructed data [1]. However, considering the size of the dataset, the amount of possible combinations is too large to compute, it will take too long to generate the combinations and likely run out of memory to store them. Here we only focus on generating combinations of size two as an indication of the dataset quality, since the larger combinations are normally rare. The relative error is defined as below [1]:

$$re = \frac{|s_o(a,b) - s_p(a,b)|}{AVG(s_o(a,b),\ s_p(a,b))},$$

Where $s_o(a,b)$ is the support of the combination of two terms (*a*,*b*) in the original dataset and $s_p(a,b)$ is the support of the combination of two terms (*a*,*b*) in the reconstructed dataset. Since reconstructing dataset will introduce new term combinations that does not exists in the original dataset, this should be taken into account when choosing a denominator, that’s why instead of choosing $s_o(a,b)$ as denominator, [2] opted to the average of both supports. For example, the term combination {“C”,”D”} does not exists in the original dataset, but it does exists in the reconstructed dataset, if we use $s_o(a,b)$ as denominator, we will get a division by 0.

Calculating the average relative error on all the combinations of size two does not indicate the relative error accurately in the cases of skewed distributions and large domains [1]. A majority of those combinations would be infrequent or not even exists in the original dataset, but they would dominate the result. To avoid this situation, the domain of the dataset are ordered by descending term support and a small sequential terms are used to trace relative error, the 200th-220th most frequent term are used to measure relative error, which in this case relative error is a more accurate indicator of how well the less frequent but not extremely rare combinations are kept in the reconstructed dataset [1].

The parameter defined in [1] is k=5, m=2. Here, I tried a few experiments to compare my results to the experiment results of [1]. The Figure 23a shows the information loss obtained by [1] on the three datasets. Figure 23b illustrates the average relative error I obtained

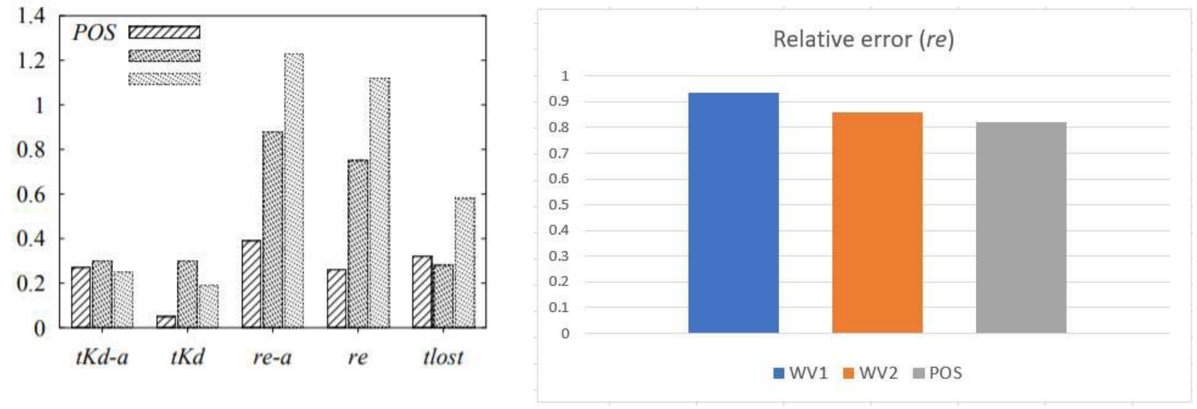


Figure 23: Comparison of relative error (*re*) (a-b) (left by [1])

on each dataset over 5 runs, details are available in the Excel file inside the Appendix folder.

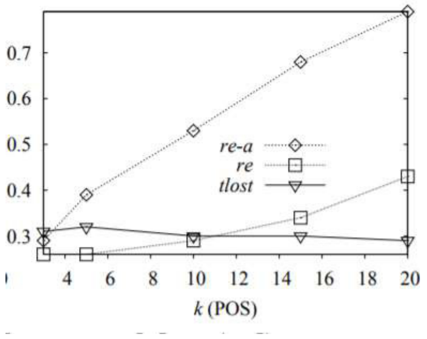
Average relative error over 5 runs:

WV1: 0.93362

WV2: 0.86014

POS: 0.82142

The characteristics of our results are much different than the results obtained in [1], Figure 23a shows that the relative error of dataset *POS* is significantly lower than the other two datasets. Although it is not explicitly stated, we can assume that the middle column represents *WV1*, and the right column represents *WV2*. It appears that WV2 should have the most relative error among the three datasets, but my result (Figure 23b) appears differently. While *POS* still has the least relative error, *WV1* has the most relative error with an average of 0.93362. The ratio difference between the values are much smaller when compare to Figure 23a. However, it appears that both the values of *WV1* is close, and the relative error of our result on *WV2* is superior to the left chart.



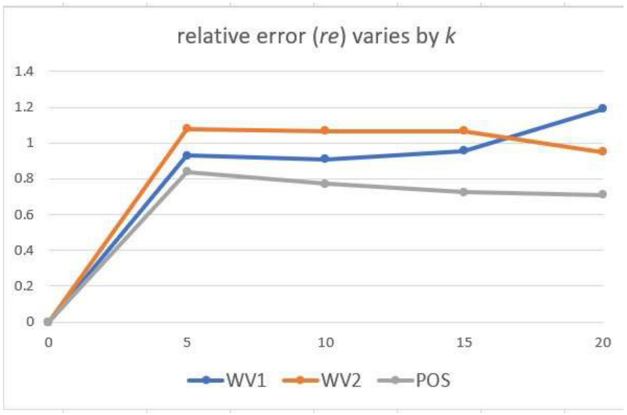


Figure 24: relative error varies by *k* (a-b) (left by [1])

Figure 24a shows the relative error scales linearly with *k* in the left graph. On Figure 24b, our results indicate that only the relative error of *WV1* scales upward, while other datasets scales downward, which is different than what is shown in the left graph. The differences between different *k* values in Figure 24b also appears to be insignificant.

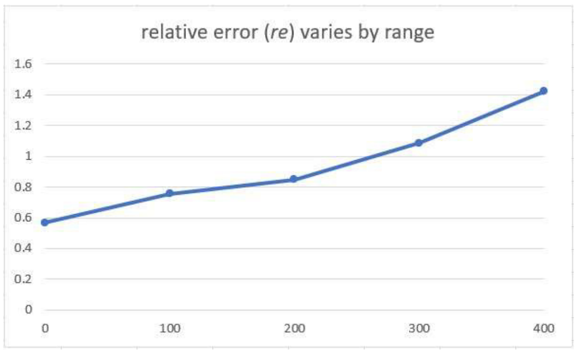


Figure 25: relative error varies by range

We measure relative error on the combination of the 0-20th, 100-120th, 200-220th, 300-320th and 400-420th most frequent terms in *POS*. Relative error scales linearly with the frequency order of the terms as shown in Figure 25.

5.5 Top-K Deviation

Top-K deviation (*tKd*) is a metric that measures how the top-K frequent itemsets of the original dataset differs in the reconstructed disassociated dataset. Let *FI* and *FI'* be the top-K frequent itemsets in the original dataset and the reconstructed dataset, *tKd* is defined as below [1]:

$$tKd = 1 - \frac{|FI \cap FI'|}{|FI|}$$

tKd expresses the ratio of the top-K frequent itemsets of the original dataset that also appear in the top-K frequent itemsets of the reconstructed dataset. Here we measure the 1000 most frequent itemsets in the datasets.

To find the top-K frequent itemsets, I import an Apriori function that extract frequent itemsets for association rule mining using Apriori algorithm from mlxtend (machine learning extensions, a Python library) [9]. The *min_support* argument is a user specified support threshold, for instance, if *min_support* is 0.02, the apriori function only returns a set of items that occur together in at least 0.2% of all the records in the dataset. However, the lower the *min_support*, the longer it takes to compute the frequent itemsets, but setting it too high might not generate enough (1000) frequent itemsets. After some testing, it appears that when *k*=5, m=2, by setting the *min_support* to 0.01 for *WV1*, 0.03 for *WV2* and 0.08 for *POS* provides the best performance and still generate enough frequent itemsets.

Although not explicitly stated in [1], single item itemsets are not included, only itemsets with two items or above is a valid itemset.

Figures below shows my *tKd* results compare to the experiment results in [1]. Figure 26a contains *tKd* obtained in [1] where k=5, m=2. Figure 26b demonstrates our *tKd* results with the same parameter.

Although it is not stated in the left chart, we can safely assume that the middle column represents *WV1*, and the right column represents *WV2*. We can see that the *tKd* of the three datasets of both charts behave similarly, but our result has a generally higher *tKd*. *POS* has a significantly lower *tKd* that the other two datasets, because it is the largest dataset and its records have the longest average length. This reflects the fact that disassociation managed to create multiple record chunks for *POS* [1].

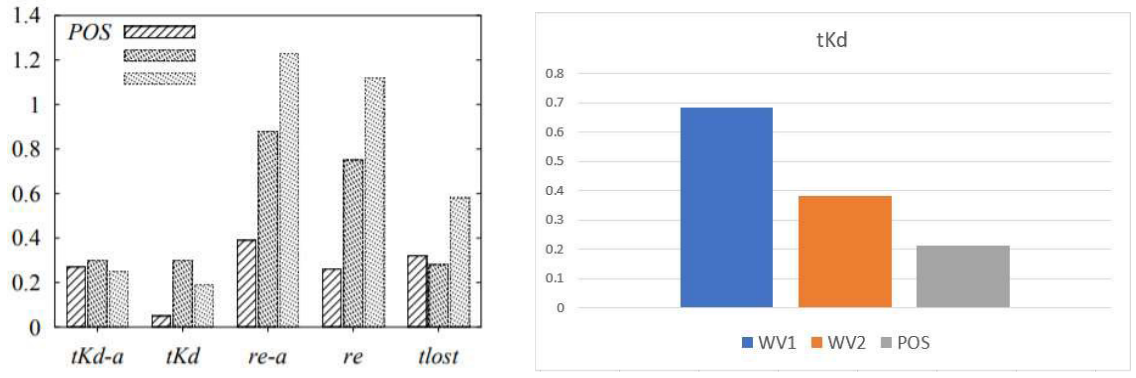


Figure 26: Comparison of tKd (a-b) (left from [1])

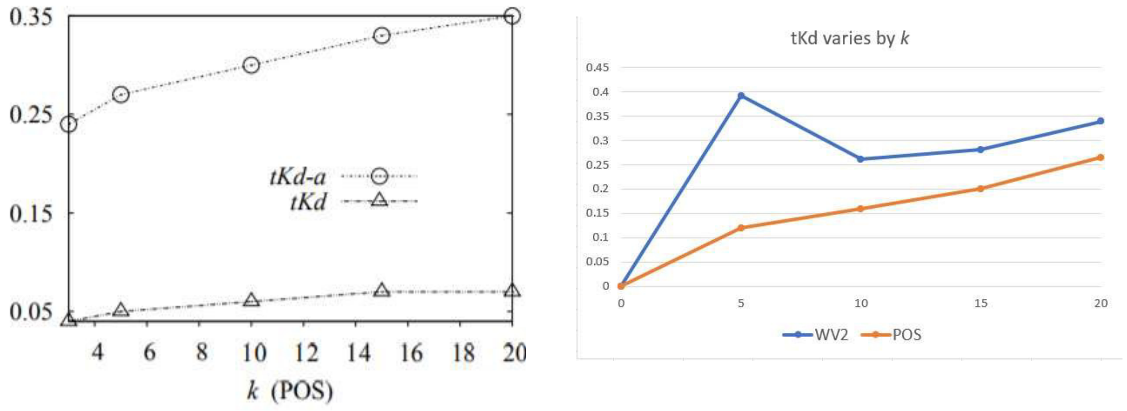


Figure 27: Comparison of tKd with growing *k* (a-b) (left from [1])

From Figure 27, we can observe how information loss accelerates as the power of the guarantee, expressed by *k* parameter, grows [1]. *tKd* is a measure that depend on the most frequent item and itemsets are only slightly affected by *k*, since they are mostly preserved in the record chunks, that reflects in Figure 27a where the increase of *tKd* is barely noticeable as *k* grows. Figure 27b shows that the margin of increase of tKd is slightly higher than Figure 27a, and the reason may be lack of refining algorithm. I also notice that tKd of WV2 when *k*=5 is oddly high, and I could not provide a reasonable explanation.

5.6 Reconstruction of Dataset

Possible original datasets can be constructed by combining the subrecords of record chunks and padding some terms in term chunk. We called it *reconstructed dataset*. Reconstructed datasets should have similar statistical properties to the original dataset. The reconstructed datasets will be used to compare and evaluate the similarity of reconstructed datasets to disassociated datasets. The dataset is reconstructed by reconstructing the disassociated dataset. The paper [1] does not explicitly state the method of reconstructing the dataset, so I perform the dataset reconstruction by shuffling the subrecords in the record chunks of the clusters and combining them together to form a collection of records. Every subrecord in the record chunks in a cluster will be shuffled and combined into records, then finally randomly padding 0 or more terms in term chunk into the records. For example, {madonna, ruby, digital camera, itunes, audi a4} is a reconstructed record from cluster P_2, taking {madonna, ruby} from C_1, {} from C_2, {digital camera} from C_3, and {itunes, audi a4} from C_T.

6. Future Work

6.1 Refining

I would like to finish implementing the refining stage of the disassociation algorithm. The current evaluation results does not reflect accurately compare to evaluations that are performed on a dataset disassociated by the completed disassociation algorithm. I would like to see if my results will be similar to the evaluation results in [1].

6.2 Performance Improvement

I would like to try to improve the performance of my code, and see if I could make the execution time shorter. The execution time of *POS* is almost 15 times higher than the performance displayed in [1]. The time spent on vertical partitioning of the dataset *WV1* is oddly long, I would also like to find out the problem and see if I could fix that as well. Other than the time performance, I would also like to improve the memory management of the code, as running the program on the large dataset sometimes cause memory issues.

6.3 Top-K multiple level mining loss tKd – ML^2

A version of *tKd*, called the *top-k multiple level mining loss tKd – ML^2*, that compares disassociation with generalization-based methods. Mining a dataset at a multiple levels of a generalization hierarchy in an established technique, allows the detection of frequent association rules and frequent itemsets that might not show in the most detailed level of the data [1]. A generalized frequent itemset is considered lost if it contains terms that have been generalized at a higher level during the anonymization process [1]. If I have more time, I would like to implement this, so I could learn some new knowledge about associative rule mining.

6.4 re-a and tKd-a

re-a and *tKd-a* is the relative error and top-K deviation for the disassociated datasets computed by only taking into account the subrecords that appear inside the record and shared chunks [1]. However, to implement these evaluation metrics requires refining stage to be implemented.

7. Conclusions

The project has successfully implemented the horizontal partitioning and vertical partitioning of the disassociation algorithm proposed in [1] using Python. While the disassociation algorithm is not fully implemented, we can see the difference between the results before and after the refining stage by looking at the results produced by this project and results in [1] respectively, although I have not reached a concrete conclusion on why some of the results may differ significantly. The evaluation metrics evaluates the information loss of the disassociated dataset.

8. Reflection

Reflecting on the whole process of completing this project, first I want to thank my supervisor for guiding me through this project. The weekly meeting really helps me progressing on the project in a timely manner. However, later due to the COVID-19 situation, face-to-face meeting is cancelled, and Skype meeting took place. However, the meeting frequency sometimes went from weekly to bi-weekly, which I regret a lot. At the end of the project, I almost ran out of time because at some point I was lost on what to work on, which now I thought that maybe I would do better if I speak to my supervisor more often, and he could provide me clear guidance to solve the problem I was facing.

This project taught me the importance of analytical thinking and creative thinking when studying and trying to understand a research paper. Many times, I thought I understand the algorithm, only to be corrected by my supervisor. After being corrected, I always find that my initial interpretation is so obviously wrong, yet I still did not see it. When I reflect on it, I realized that I did not analyse the problem enough and think about what the algorithm is trying to achieve. After you fully picture the problem it is trying to solve, and understand what the algorithm is trying to achieve, it becomes easier to understand the algorithm and the implementation of it. The ability to "think outside the box" also plays a huge role here in my opinion.

This program helps me develop my programming skill and problem-solving skill significantly. I have to consider the performance of the program and routinely improve my code quality to achieve a better performance. The data type that I use, the approach that I take to implement the algorithm, all matters significantly when it comes to memory and time performance. For example, I was not careful about choosing the data type to store a Record's value, and I use a list initially because that's what I instinctively thought was the best, and did not do enough research to see the difference between List and Set not just in terms of performance, but in general. I later found out that using a set to store the record value would be much better, because there are a decent amount of item checking (check if a record value contains an item), e.g. check if the record value contains a term combination, and using a set for that would be significantly faster than using a list. Implementing the algorithm requires a good amount of problem solving, regularly I have to figure out a way to implement what the algorithm is trying to achieve, and personally I think that is the fun part.

A big problem that I encounter is my poor planning skill and poor time management. The occasional procrastination, some part of the algorithm took longer time than expected to implement and too little time at the end to test the results, makes me realized the importance of proper planning and execution. While planning, I should never assume everything to go perfectly, and should always plan for the worst-case scenario.

9. Reference

[1] Terrovitis, M., Liagouris, J., Mamoulis, N., & Skiadopoulos, S. (2012). Privacy preservation by disassociation. *arXiv preprint arXiv:1207.0135*.

[2] Barbaro, M., & Zeller, T. (2006, August 9). A Face Is Exposed for AOL Searcher No. 4417749. Retrieved from https://www.nytimes.com/2006/08/09/technology/09aol.html

[3] Fung, B. C., Wang, K., Chen, R., & Yu, P. S. (2010). Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys (Csur)*, *42*(4), 1-53.

[4] Machanavajjhala, A., Kifer, D., Gehrke, J., & Venkitasubramaniam, M. (2007). l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, *1*(1), 12-13.

[5] Samarati, P., & Sweeney, L. (1998). Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression.

[6] Terrovitis, M., Mamoulis, N., & Kalnis, P. (2008). Privacy-preserving anonymization of set-valued data. *Proceedings of the VLDB Endowment*, *1*(1), 115-125.

[7] South Carolina Department of Health and Environmental Control (SC DHEC). (n.d.). SC Demographic Data (COVID-19) | SCDHEC. Retrieved May 31, 2020, from https://www.scdhec.gov/infectious-diseases/viruses/coronavirus-disease-2019-covid-19/sc-demographic-data-covid-19

[8] World Health Organization (WHO). (n.d.). WHO Coronavirus Disease (COVID-19) Dashboard. Retrieved May 31, 2020, from https://covid19.who.int/

[9] Raschka, S. (2020). *Apriori - mlxtend*. Rasbt.github.io. Retrieved 4 June 2020, from http://rasbt.github.io/mlxtend/user_guide/frequent_patterns/apriori/.