Implementation of a Data Privacy Protection Tool for Transaction Data

Final Report

Author: Daniel Hendry
Supervisor: Dr Jianhua Shao
Moderator: Dr Vctor Gutirrez Basulto

Module Code: CM3203
Module Title: One Semester Individual Project
Credits: 40

Abstract

An increased amount of data is being gathered and stored and is set to increase into the future. An increasing concern becomes how we can ensure enough privacy without sacrificing the utility of the data.

Existing techniques alter the original data by either generalising or suppressing it. This becomes infeasible with higher dimensional data, as achieving the same level of privacy requires very high information loss.

This dissertation aims to implement a data privacy tool for anonymising high dimensional data through a technique called disassociation. This aims to preserve the original terms whilst hiding identifying combinations of terms in order to protect a users' privacy.

Acknowledgments

I would like to thank my supervisor Dr Jianhua Shao for his continued guidance and advice throughout this project. Your assistance has been greatly appreciated.

Contents

List of Figures

List of Tables

1 Introduction

The increasing amount of data being saved and used is growing exponentially with the technology to store and capture said information becoming cheaper, making it easier to store and mine more of our data.

The challenge comes, as data gets released to a wider audience, to maintain the privacy of each individual whilst maintaining a good degree of utility for data mining operations.

For relational data, the idea of k-anonymity is that each record should be indistinguishable from $k-1$ other records[3], was proposed. This followed with data privacy techniques such as generalisation and suppression, which alter the original data in order to conform to privacy.

For higher dimensional data, transaction data for this project, these techniques can apply, but at a great cost to the data utility since to achieve the same level of privacy requires altering the data so much that the information loss is very high and the data utility is low.

A proposed solution is that of disassociation, which aims to hide infrequent combination of terms that could be used to uniquely identify an individual. This was first proposed in [2] and shows a great increase in data utility over existing techniques.

This project aims to implement that algorithm, based on disassociation, and recreate the information loss results in order to verify that this method produces a higher utility anonymised dataset.

From this implementation, this data privacy tool can then be used in future projects as a comparison between other data privacy tools.

1.1 Project Aims

Implementation of disassociation algorithm - The algorithm will he developed in Java and will focus solely on inputting a transaction dataset and outputting an anonymised dataset.

Replication of Information loss Results - I will aim to replicate as many of the information loss results from [2] as I can in order to verify improved utility to be shown.

2 Background

This section is dedicated to the wider context of this report. Providing support and understanding for concepts used throughout the rest of this report.

2.1 Transaction Data

This project focuses on transaction data over relational data, as the algorithm I intended to implement has been designed to handle transaction data over relational.

Transaction data refers to a record with a variable length record, meaning it is classified as high dimensional data which can be seen as amount of attributes. Therefore, it is possible to map transaction data to relational data, where each value will become an attribute and the value will be either "0" or "1" depending on whether the value exists within that record.

An example of transaction data would be a persons' shopping purchases, since it is very likely each person will buy a different amount of items, leading to the variable length.

ID	Transaction
r1	ketchup, pasta, pencil, paper
r2	paper, pen, chewing gum
r3	ketchup, chips
r4	pasta, pesto
r5	chewing gum, paper

Table 1: Example of transaction data

ID	Ketchup	Pasta	Pencil	Paper	Pen	Chewing Gum	Chips	Pasta	Pesto
r1	1	1	1	1	0	0	0	0	0
r2	0	0	0	1	1	1	0	0	0
r3	1	0	0	0	0	0	1	0	0
r4	0	0	0	0	0	0	0	1	1
r5	0	0	0	1	0	1	0	0	0

Table 2: Transaction data mapped to relational data

It is well known that companies themselves, or out source to third parties, will use various data mining techniques to extract patterns within data that can be used by said company to predict outcomes, such as what products will sell during a season or noting what people buy as part of a loyalty card scheme so that retail companies can target products towards your specific purchasing habits.

The problem comes in ensuring privacy within the dataset so that, no individual can be identified within the dataset whilst still having the utility to carry about accurate data mining.

The novel idea to simply remove the most sensitive identifiers, such as name and address, has been shown to not be enough to protect a users' privacy, as a combination of only a few attributes could uniquely identify an individual. This was shown as 87% of the population of the United States of America could be identified using only {5-digit ZIP, gender, date of birth} [4].

2.2 k-Anonymity

One approach to this was proposed by Latanya Sweeney called k-anonymity [3]. The general idea behind this, is that each record within a dataset will be not be distinguishable between k-1 other records.

Datasets are made k-anonymous based on their quasi-identifiers. These are defined as a set of attributes that can combine in such a way to uniquely identify an individual without each attribute themselves being a unique identifier[3].

The example being that gender, birth dates and postal codes can combines to uniquely identify an individual or 87% of individuals in the United States as discovered by Latanya Sweeney in an earlier study[3]. [Might move to intro maybe]

An example of this is shown in Figure 2 where the quasi-identifiers are {Age, Gender, ZIP}[1] and $k = 2$. This means that each combination of the quasi-identifier values occur at least 2 times in the anonymised dataset. The table on the left shows a dataset that is not k-anonymous and each combination of the quasi-identifiers will lead to the identification of an individual while the right hand side leads to, in this case, 2 records of each combination.

ID	Age	Gender	ZIP
r_1	40	Male	02130
r_2	30	Female	02123
r_3	35	Female	02120
r_4	19	Female	02145
r_5	23	Female	02142
r_6	45	Male	02138

ID	Age	Gender	ZIP
r_1	$40 \leq$ Age ≤ 45	Male	0213*
r_6	$40 \leq$ Age ≤ 45	Male	0213*
r_2	$30 \leq$ Age ≤ 35	Female	0212*
r_3	$30 \leq$ Age ≤ 35	Female	0212*
r_4	Age ≤ 20	Female	0214*
r_5	Age ≤ 20	Female	0214*

Figure 1: Example table before and after k-anonymisation. $k = 2$

The two most common techniques used to anonymise the dataset, and used in the table above, are generalisation and suppression.

Generalisation is the process of replacing an attribute value with a broader value, while still retaining meaning. The example in Figure 2 is with the attribute 'Age'. The values are replaced with a range, e.g., the first row Age 40 is replaced with age range $40 \leq$ Age ≤ 45 as this means the first and last row can have identical age values to ensure k-anonymity on the age attribute.

Suppression involves not releasing certain parts of the data by removing them and replacing them with an asterisk '*'. In table 2, the end of the zip-codes have been suppressed and replaced with an asterisk, e.g., ZIP 02130, first row, becomes 0213* in order to match another record, the last one. This is to allow at least k zip-codes to be the same, therefore achieving k-anonymity on the ZIP attribute.

2.3 Problems with k-Anonymity

Figure 2 shows an example of anonymising relational data, but as shown I aimed to use transaction data, which is higher dimensional. As shown in [5], as the number of dimensions

[1] ID is included in the table as reference

increase, it becomes harder to ensure k-anonymity without a high amount of information loss. This is why k-anonymity is not ideal for higher dimensional data.

Another problem with general k-anonymity is that it relies on knowing the quasi-identifiers within the dataset, for example, {Age, Gender, ZIP} in Figure 2 are defined, but within a dataset such as 1, it is not clear what could potentially be a quasi-identifier. As I show with disassociation, this does not consider the differences between sensitive (name, address, etc.) and quasi-identifiers (age, gender, etc.).

2.4 k^m-Anonymity

One solution put forward, that is used in disassociation, is the idea of k^m-anonymity [6]. This is an expansion on k-anonymity concept but now applies for up to m terms. This means that if someone knows up to m terms in a record, they will not be able to distinguish from $k-1$ other records.

Owner	TID	Items Purchased
Alice	T_1	{Beer, Diapers}
Bob	T_2	{Wine, Diapers, Pregnancy Test}
Chris	T_3	{Beer, Wine, Pregnancy Test}
Dan	T_4	{Beer, Wine, Diapers, Pregnancy Test}

Figure 2: k^m-anonymous example - 2^2-anonymity [1]

This example, from [1] shows a dataset that is k^m-anonymous, more specifically, 2^2- anonymous. This means that if an adversary knew terms such as Beer, Diaper, they could only find transactions $T1$ and $T4$ that contain both. There is no unique transaction that could be identified and reduces the probability of an adversary finding the record they want to $1/k$.

The idea proposed of k^m-anonymity was proposed to be used with generalisation [6] [Use the example from the paper]

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$(D_1, ignore \cup \mathsf{a}) \cup$HORPART$(D_2, ignore)$

Figure 3: $HORPART$ - Horizontal Partitioning Pseudo-Code [2]

3 Disassociation Algorithm

In this section, I present the idea of disassociation as first presented in [2] and explain the algorithm that was implemented.

Compared with generalisation and suppression, which alters the original data values, disassociation aims to partition records into smaller sub-records. The reason behind this is to hide combinations of terms that could lead to the identification of an individual.

Disassociation is achieved in three different stages, Horizontal Partitioning, Vertical Partitioning, and Refining.

3.1 Horizontal Partitioning

The general idea behind horizontal partitioning is to partition the dataset into records that contain the most frequent term and the remaining records. The partitioning is applied recursively to each split of the dataset until the dataset size is small enough to become a $cluster$. The term used to split the dataset is not used again in subsequent splitting. The size of these clusters is determined by the input parameter $maxClusterSize$.

The pseudo-code for this part of the algorithm is shown in Figure 3 where it should be noted that the termination condition is if the size of the dataset is *less than* the $maxClusterSize$ meaning that even if the cluster size was selected as 10 for example, the max cluster sizes this part of the algorithm will produce will be of size 9 and below.

Figure 3 mentions the $ignore$ set. This is the set of most frequent terms that were used in the partitioning process and is unique to each dataset partition. This means that when the dataset is split, the subsequent horizontal partition for the dataset that contained the previous most frequent term will have that $ignore$ set contain this term so that it cannot be split again. The dataset of the remaining records will not contain this term. [include example of ignore terms]

This is important as a global $ignore$ set will cause an error due to line 3 in Figure 3 where the set of terms in the dataset takes out all the terms that have been used for splitting before.

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

Cluster 1

ID	Records
r_1	{itunes, flu, madonna, ikea, ruby}
r_2	{madonna, flu, itunes, viagra, ruby, audi a4, sony tv}
r_3	{itunes, madonna, audi a4, ikea, sony tv}
r_4	{itunes, flu, viagra}
r_5	{itunes, flu, madonna, audi a4, sony tv}

Cluster 2

ID	Records
r_6	{digital camera, panic disorder, playboy}
r_7	{iphone sdk, ikea, ruby}
r_8	{iphone sdk, digital camera, playboy}
r_9	{iphone sdk, digital camera, panic disorder}
r_{10}	{iphone sdk, digital camera, ikea, ruby}

Figure 4: Example dataset before and after horizontal partitioning

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 5: $VERPART$ - Vertical Partitioning Pseudo-Code [2]

If a cluster contains all the $ignore$ terms then there will be no more terms left to use for splitting.

An example of horizontal partitioning is shown in Figure 4 where the most frequent term in the original dataset was the term 'iTunes'. Therefore records ($r_1 - r_5$) were moved into the first cluster and all subsequent records (i.e. $r_6 - r_{10}$) were moved into the next cluster. For this example $maxClusterSize = 6$, hence why no subsequent splitting. If the $maxClusterSize$ was smaller, then 'iTunes' would be in the $ignore$ set for cluster one, but not for two. As noted, subsequent splitting would occur till clusters were below the $maxClusterSize$.

3.2 Vertical Partitioning

Vertical partitioning takes each of the clusters from horizontal partitioning and partitions them into two types of *chunks*. The first are called *Record Chunks*, which contain sub-records from

the cluster and each cluster is k^m-anonymous. Each cluster can contain 0 or many record chunks.

The other type of chunk is called a *Term Chunk*. This is a set of terms, no sub-records, from the cluster that contain support $< k$. Each cluster will always have only one term chunk, but it may be empty.

To create said clusters, I followed the pseudo-code outline in Figure 5. Lines 2-5 will initially create the term chunk, by first calculating the support of every term within the cluster. This simply means count the appearances of each term. This set is sorted in descending order and all the terms with support $< k$ are moved to the term chunk and not used in the rest of the algorithm.

The reason behind that term chunk is that any term within this set could potentially be joined with any sub-record to identify an individual.

The remaining terms, which are still sorted in descending order are then place in record chunks (while loop). This effectively builds up terms that will go into each record chunk. Each term is added to a "test chunk" (Lines 10-12), which is just an empty set and then that set of terms is checked to be $k - m$-anonymous. This is checking that every combination up to size m occurs at least k times in all the records in this cluster.

For example. If the set of remaining terms were {a, b, c, d} in that order, then initially the algorithm would add the term 'a' to the "test chunk". There is no need to check if this combination is k^m-anonymous since every term in the remaining set already has support more that k. Since this will always evaluate to true, this term is added to the current record chunk that is being created. The next round would add the term 'b' to the test chunk and again check if k^m-anonymous. Now the test chunk contains {a, b}, and every combination of the terms will have to be checked i.e. {{a}, {b}, {a, b}}.

The same process happens for the rest of the terms. This is not the same a generating a power-set of all the remaining terms as we only need to generate combinations of up to size m in order to prove this set of terms is k^m-anonymous. I.e. For terms {a, b, c} and $m = 2$, then the only combinations to check would be {{a}, {b}, {c}, {a, b}, {a, c}, {b, c}} and not {a, b, c}.

Any single combination that does not occur k times in the cluster dataset will evaluate to false and therefore not add the term from the test chunk to the current record chunk. This term will be added to another record chunk.

This process is completed when all terms have been added to record chunks. The final part of the algorithm, line 16 (line 17 is using the same term chunk that was created at the start of this part of the algorithm), is using the sets of terms created and then creating the sub records using those terms.

An example of the completed vertical partitioning process is shown in Figure 6 where the terms ikea, viagra and ruby did not have enough support $k = 3$ and so are moved into the term chunk. The remaining terms are added one by one and checked for k^m-anonymity. The terms itunes, flu and madonna all have k support for each of their subsets of combinations, up to $m = 2$, and so are placed in record chunk 1. The rest of the terms are placed in the next record chunk since they also have enough support for their respective combinations. The same idea applies to the second term chunk.

	Record Chunks		Term Chunk
	RC_1	RC_2	TT
r_1	{itunes, flu, madonna}		
r_2	{madonna, flu, itunes}	{audi a4, sony tv}	ikea,
r_3	{itunes, madonna}	{audi a4, sony tv}	viagra,
r_4	{itunes, flu}		ruby
r_5	{itunes, flu, madonna}	{audi a4, sony tv}	
	Record Chunk		**Term Chunk**
	RC_1		TT
r_6	{digital camera}		panic disorder,
r_7	{iphone sdk}		playboy,
r_8	{iphone sdk, digital camera}		ikea,
r_9	{iphone sdk, digital camera}		ruby
r_{10}	{iphone sdk, digital camera}		

Figure 6: Vertical partitioning example

Record Chunks		Term Chunk	Shared Chunk
RC_1	RC_2	TT	
{itunes, flu, madonna}			{ikea, ruby}
{madonna, flu, itunes}	{audi a4, sony tv}		{ruby}
{itunes, madonna}	{audi a4, sony tv}	viagra	{ikea}
{itunes, flu}			
{itunes, flu, madonna}	{audi a4, sony tv}		
Record Chunk		**Term Chunk**	
RC_1		TT	
{digital camera}			
{iphone sdk}		panic disorder,	{ikea, ruby}
{iphone sdk, digital camera}		playboy	
{iphone sdk, digital camera}			
{iphone sdk, digital camera}			{ikea, ruby}

Figure 7: Disassociated dataset after refine

3.3 Refine

At this point in the algorithm has already been disassociated and so the refining part of the algorithm aims to keep the k^m-anonymity and improve the quality of the result, i.e. reduce information loss. [2]. This is done by combining clusters together into a *joint cluster* based on whether two or more clusters have common terms in their term chunks.

From this stage, the set of clusters from vertical partitioning are referred to as *simple clusters* consisting of ≥ 0 record chunks and one term chunk, that could be empty. This stage of the algorithm introduces a joint cluster. A collection of simple clusters, with an arbitrary number of *shared chunks* (≥ 0).

A shared chunk is very similar to a record chunk in that it contains a collection of sub-records and each chunk is k^m-anonymous, with the main difference being the sub-records are shared between multiple clusters and not just one cluster. An example of this is shown in Figure 7 where the two clusters from vertical partitioning are joined together to form a joint cluster with a shared chunk. They are joined together on the basis that the terms {ikea, ruby} were in both term chunks. This collection of terms that appear in term chunks of two or more clusters are called *refining terms*.

While the example features only two simple clusters and therefore they can only be joined with each other, bigger datasets with many clusters will need a solution to decide which pairs of clusters should be joined. One way would be to compute every join of all clusters and see which pairing produces the least information loss, but this would be very inefficient, so [2] have opted for the equation shown in Figure 8.

$s(t_1) + ... + s(t_n)$ is the support of each of the refining terms in the original records. It is defined as the support in the shared chunks in [2], but this can be calculated the same way through the support of the term within the simple clusters used to make the joint cluster. For example, looking at Figure 7, the $s(ikea) = 4$ from the shared chunks, while this number can also be calculated from the support of that term in the original cluster as featured in Figure 4.

$|J_{new}|$ is simply the total combined size of all the clusters used to make this joint cluster. For our example since each cluster is of size 5, i.e. they hold 5 records each, $|J_{new}| = 10$.

$u_1 + ... + u_m$ is the number of term chunks of each term $t_1, ..., t_n$ appears in within the clusters used to create this new joint cluster. For our example, the refining term 'ikea' appeared in each of the simple cluster term chunks, therefore $u_1 = 2$.

$|P_1| + ... + |P_m|$ are the size of the simple clusters that contain the refining terms. The difference between this total value and $|J_{new}|$, is the latter totals up the size of all the clusters that will go into that joint cluster, while the former only totals the size of clusters that contain a refining term. A simple example where this could occur would be if a joint cluster made up of 2 simple clusters, tries to merged with another simple cluster and all clusters contain 5 records. If refining term t_1 occurs in only one cluster in the joint cluster and then the other simple cluster, then $|J_{new}| = 15$, add the size of all the simple clusters together, while $|J_{new}|$ $= 10$, as the refining term only occurred in 2 of the 3 simple clusters.

Therefore to form Figure 7, the equation would evaluate to below and therefore form a new joint cluster.

$$\frac{s(ikea) + s(ruby)}{|J_{new}|} \geq \frac{u_1 + u_2}{|P_1| + |P_2|}$$

$$\Rightarrow \frac{4+4}{10} \geq \frac{2+2}{10}$$

This equation gives an option to determining if two clusters (simple or joint) should be merged, but still leaves the problem of having to try every single combination of clusters. Once again as noted in [2], this is computationally infeasible, and so they have opted for a heuristic whereby only adjacent clusters are checked, using the equation in Figure 8, and merged to form a new joint cluster.

This heuristic is shown in Figure 9. To determine the order in which clusters are merged, the algorithm first introduces a *virtual term chunk*, which, as described in Figure 9 is the "union of the term chunks of its simple clusters [2]." At first, every cluster is a simple cluster so each virtual chunk is just the contents of their respective term chunks.

By line 3, to order the clusters, a new support is introduced, the *term chunk support*. This simply takes every single term within each term chunk in each cluster and calculates how many term chunks that term appears in. The contents in the virtual term chunks are sorted in descending order of support because the set of clusters are sorted lexicography. This means each cluster is sorted on descending support first and if a tie, order by the terms alphabetically.

The clusters are ordered as it is gives the best chance of adjacent clusters merging since the clusters with terms with the highest support will be seen first and have a greater chance at merging due to their higher support.

Line 4 is the actual modification of clusters to joint clusters. This means, go through each adjacent pair of cluster and see if equation 1 (Figure 8 returns true, to which a joint cluster is created which an arbitrary number of shared chunks based on common term chunk terms, each shared chunk of which is k^m-anonymous.

The only difference on subsequent iterations of refine is there is now joint clusters among the simple clusters. The termination state means that after going through the list of clusters, no more adjacent virtual chunks could be produced.

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

Figure 8: Equation determine if two clusters are to merge [2]

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 9: $REFINE$ - Refine Pseudo-Code [2]

3.4 Challenges of Algorithm

3.4.1 Horizontal Partitioning

The $maxClusterSize$ variable presents the first challenge with this algorithm, determining this cluster size. There was no given information in [2] about what this value should be. However, a more recent paper [7] featuring disassociation showed they managed to achieve good clusters when $maxClusterSize = 2 * k$. i.e. If $k = 3$, $maxClusterSize = 6$ and so on.

$maxClusterSize$ must always be $> k$ as otherwise no record chunks will be produced, since no term will have support $\geq k$, and all the terms will be in term chunks. Sub-records will be created when shared chunks are created, but since they will be associated across a minimum of two simple clusters, the information loss is greater than if the terms were in record chunks.

Since the $maxClusterSize$ allows for clusters to be less than this size, this means there is the chance of having clusters of size $< k$. The original algorithm does not account for these clusters and so I planned a couple possible improvements to this part of the algorithm.

The first is that when the dataset splits into two with records containing the most frequent term and the remaining terms, if either of those splits contain $< k$ records, then I do not continue the split on this dataset and this becomes a cluster. This will avoid the scenario of having clusters of size $< k$, however could lead to cluster of a size $> maxClusterSize$ meaning those clusters could contribute to higher information loss due to a bigger shared chunk size.

The second improvement, which is independent of the first and would have to be implemented separately, is that when dataset is split, to check the size of each split and carry the algorithm as normal if a data-split is $> maxClusterSize$. If the scenario where the other data-split size is $< k$, then this gets added to separate list of records. This is continued until the algorithm would have reached its natural conclusion that all the data has been moved into clusters, then the algorithm is called again, with an empty $ignore$, on the list of records that had been put aside during the first run of the algorithm. This has the benefit of putting more records into clusters of size $< maxClusterSize$ but $> k$ and only having at most one cluster of size $> k$ since every record would have been placed into clusters.

A lesser know consideration I came across was in finding the most frequent term within a dataset. There is no mention of if there is a tie in the most frequent term to select. Logically,

it makes sense to choose the term that comes first alphabetically and so this is what I used when implementing the algorithm.

3.4.2 Vertical Partitioning

The algorithm as outlined in Figure 5 does not leave much to interpretation with the biggest challenge being to generate all the combinations of terms within the test chunk and evaluate each of them to see if they are k^m-anonymous. This is similar to generating a power-set of combinations of size n, except without having to generate the empty set and I would only have to check combinations up to size m to satisfy k^m-anonymity.

3.4.3 Refine

I believe this part of the algorithm has the most chance for interpretation. At the start Figure 9 mentions to add the virtual term chunk to each of the joint clusters, but fails to mention that at the start of the algorithm there are only simple clusters and that the virtual term chunk is just the term chunk of each of the simple clusters.

Although not explicitly stated in [2], joint clusters are effectively put back into the list of clusters as joint clusters can be merged with other joint clusters to produce a new bigger joint cluster. For example, if there was three simple clusters (C_1, C_2, C_3) after vertical partitioning. If after the first iteration of refining, a new joint cluster (C_1, C_2) is formed. The next iteration, C_1, C_2 may merge with C_3 to form C_1, C_2, C_3 joint cluster. Since there are no more clusters to merge, the resulting clusters with be both C_1, C_2 and C_1, C_2, C_3 each with their own shared chunks.

The reason behind this is to minimize information loss per joint cluster. Since records within a shared chunk could be associated with any of the simple cluster records, the greater the number of records, the greater the number of records in the shared chunk and the higher the information loss since there are more records to associate each shared chunk record with.

4 Implementation of the Algorithm

In this section, I detail the implementation of the disassociation algorithm as outlined in the previous section, the format of the data required for input, the choices for implementing the way I did and any challenges I faced along the way.

4.1 Programming Language

I implemented this algorithm in Java 1.8 as this is the language I am the most comfortable in. Java offers ease of object-oriented programming techniques and scales well with big data and offers a slightly better performance than Python[8].

4.2 Package Control

As is common with many Java projects, I used Maven[2] to handle project dependencies. This means I can easily include various outside dependencies, manage their version and then export this project so that it can easily be integrated with other Maven projects as required.

Existing data mining algorithms such as Weka[3], are open source and developed in Java. A motivation for this project was this could be the start of a "Weka-lite" with different algorithms. This also leaves room to be integrated as part of Weka in the future.

4.3 User Interface

As part of my objectives, I illustrated that the algorithm was always more important than the UI and therefore the interface for this software is the command line arguments where the output will be printed to console.

4.4 Data Input

An aim was to ensure the program could accept a dataset of transactions and so I decided to enforce that the dataset must be in a comma-separated-value (CSV) format. I chose this as there is a dedicated library for reading and writing CSV files into Java, and more specifically a Java POJO (Plain Old Java Object).

The library I chose to use is the open-source OpenCSV[4]. This allows for a CSV file to be read and mapped straight to a Java object, which I had called a *Record*. This simple object has a String ID and a list of String values representing the transaction.

To map from a CSV line to POJO, OpenCSV allows for a binding based on the column position in the CSV file, to say what object to map to, and what delimiter to split on. For this algorithm, each single transaction term is treated as a String object and the delimiter is

[2]https://maven.apache.org/
[3]https://www.cs.waikato.ac.nz/ml/weka/
[4]http://opencsv.sourceforge.net/

10307 10311 12487
12559
12695 12703 18715
10311 12387 12515 12691 12695 12699 12703 12823 12831 12847 18595 18679 18751
10291 12523 12531 12535 12883
12523 12539 12803 12819
12819
12471 12491 12531 12535 12567 12663 12667 12823 18447 18507 18691
12487

Figure 10: Example dataset suitable for algorithm

a single space character. There is one column in the dataset and each row contains the list of terms separated by a single space character.

An example of this data can be seen in Figure 10 which shows only one column for the data consisting of a space-split values. The drawback to having the space-split values is if there was terms that were two words, such as 'Audi A4' then these would be treated as two separate terms. I would ideally prefer this to be universal and split on a comma such that it would not matter what the length of the term was, provided there was a comma.

The datasets that I used to test the algorithm were already space split and so the conversion to CSV, via import to Excel and then export as CSV, allowed for ease of data preprocessing.

4.5 Models

I devised a set of classes, or models, to represent the various clusters and chunks described in section 3. All models are *Final*, which means they cannot be overridden.

4.5.1 Record

As stated the first model is called a 'Record', consisting of a string ID and a list of string terms. A simple string ID was chosen over something more complicated such as an UUID (Universally Unique Identifier) as it is a lot more readable to understand an ID such as 'r1' representing record one, than a 128 bit ID.

String values were used to allow a more varied amount of values to be used. For example, the terms "10333" and "iTunes" can be represented as string values whereas only 10333 can be an integer and 'iTunes' would require a conversion process from string to integer. The benefit of having all integer values however, would be that comparisons would be faster as integer comparison looks at the bits while Java compare method for strings looks at each character and compares individually.

4.5.2 Cluster

From section 3, a (simple) cluster will contain a list of records and then eventually a list of record chunks and a single term chunk after vertical partitioning. I added another simple string ID, which apart from give a simple unique value to the cluster, are used in the formation of a joint cluster as described below.

For record chunks, this is represented as a list of list of records. Since one record chunk consists of a number of sub-records, equal to the cluster size, and as there is an arbitrary number of record chunks, then it made sense to have a list of lists. I could have extracted a separate object for a record chunk, but I felt this was unnecessary abstraction.

For the term chunk, I chose this to represent this as a map object made up of key-value (String-Integer) pairs. A map was used as the support associated with each term is used to determine which terms go into the term chunk (all terms with support $< k$) and which go into record chunks. The term chunk record support values are also used in equation 1 (Figure 8) when determining if two clusters should be merged and it is more efficient to hold onto this support rather than recalculate it again in refine. Java *HashMaps*, the default implementation of a map, do not allow for duplicate keys, effectively allowing to fulfill the requirement of a term chunk being a unique set of values rather than records as shown in record and shared chunks.

4.5.3 Joint Cluster

The joint cluster class consists of a list of the simple clusters that make the joint cluster. The ID is formed by the concatenation of its simple clusters to quickly and easily show which clusters were used to make up the joint cluster. This is primarily for illustrative and debugging purposes as the published dataset should not contain IDs as an attacker could easily rebuild a record with the small exception of knowing which terms from the term chunk match to each record.

The shared chunks are modelled in the exact same way as record chunks described above in a (simple) cluster, with a list of list of records. The reasoning is the same.

I included the virtual chunk object in this class since a joint cluster is the only class to require it and this is used in the refine part of the algorithm. Since this is used to represent term chunk support, this is represented by another map object with the key being String and the value being Integer. At the start of the refine method, each simple cluster is mapped to a joint cluster so that I am working with the same object throughout the refine process without having to switch between simple and joint clusters.

Within refine, this requires the sorting of joint clusters by the contents of the virtual chunk. Since this is a map, this requires a custom comparator as there does not exist one for map objects. Individual comparison objects exist for String and Integer.

Figure 11 shows the custom comparator I created for sorting joint clusters. I first get the virtual chunk objects and compare their support values. I have a variable called *smallestListSize*, which represents which of the two virtual chunks has the smallest list size. This is because two term chunks, in each comparison, may be of different sizes and since I iterate through both at the same time, it was important to put a limit on the index size to prevent an

```java
public static final Comparator<JointCluster> VIRTUAL_CHUNK_COMPARATOR = (o1, o2) -> {
  Map<String, Integer> o1VirtualChunk = o1.getVirtualChunk();
  Map<String, Integer> o2VirtualChunk = o2.getVirtualChunk();

  List<Integer> o1Values = new ArrayList<>(o1VirtualChunk.values());
  List<Integer> o2Values = new ArrayList<>(o2VirtualChunk.values());

  // Compare up-to the smallest list of values.
  int smallestListSize = Math.min(o1Values.size(), o2Values.size());
  int compareSupport = IntStream.range(0, smallestListSize)
          .map(i -> o2Values.get(i).compareTo(o1Values.get(i)))
          .filter(value -> value != 0)
          .findFirst()
          .orElse( other: 0);

  // If compare is zero then support is the same and must check the term.
  if (compareSupport == 0) {
    List<String> o1Terms = new ArrayList<>(o1VirtualChunk.keySet());
    List<String> o2Terms = new ArrayList<>(o2VirtualChunk.keySet());

    return IntStream.range(0, smallestListSize)
            .map(i -> o1Terms.get(i).compareTo(o2Terms.get(i)))
            .filter(value -> value != 0)
            .findFirst()
            .orElse( other: 0);
  }
  return compareSupport;
};
```

Figure 11: Custom comparator for sorting joint clusters

ArrayIndexOutOfBoundsException. Where Java tried to get an object at an index that is greater than the array size.

The first comparison looks at each support value in each virtual chunk with the same index. This compares in descending order, so a greater support will be moved above a lesser support and equal support with return 0 as is custom with comparisons in Java. Only if each support value is equal in the two lists, up to the minimum size, will I then check the terms themselves, else the value, which will be either 1 for if the first integer is greater than that second or -1 if second integer is bigger than the first.

This comparison is the same as the value comparison except comparing two strings where alphabetical order takes precedence.

4.6 Utility Class

There is one "helper" or "utility" class, named *DisassociationUtils* used within this program that uses common methods shared between the three stages of the disassociation algorithm. The following is a description of each method and its purpose.

projectRecords - Used in vertical partitioning and refine. In vertical partitioning, this is used at the end of that algorithm to project the terms that have been validated as k^m-anonymous to new record objects to add to a record chunk. This is also used similarly in refine during the creation of shared chunks where the refining terms are projected into new records. If a record does not contain any of the "allowed terms" (i.e. k^m-anonymous valid terms or any refining terms that are k^m-anonymous), then I return null.

countSupport - This take a list of terms and counts the occurrences of each term. In Java 8 with the introduction of streams, this allows for a different way of iterating through data other than a for loop. In this example I used a collector that groups objects by an mapping function, *Function.identity()*, which means it groups by objects that are identical, and a collector to represent the occurrences of the identical objects.

I created a custom collector, as the one to represent the numeric value of each term is of type *Long* by default. Long objects in Java are represented by 64bit values while integers are represented by 32bit values. The maximum integer value is 2,147,483,647, which is greater than the biggest dataset used in testing and for bigger datasets that would have to have the max integer value in order to exceed this map.

This is used in all three stages of the algorithm. In horizontal partitioning, this is used to find the most frequent term, in vertical partitioning this is used to determine which terms have support $< k$, and therefore go to the term chunk, and which terms will be used in record chunks. In refine, this is used to calculate term chunk support.

sortSupportMap - The default map that the above method is collected to is a HashMap, which does not guarantee order. This sorts the map based first on the support of the term in descending order, with ties being settled in alphabetical order. The result is collected to a LinkedHashMap, which retains the sort order.

For horizontal partitioning, this takes the support map of each term, sorts it and is returned where then the first term is returned, which will be the term with the most support since the map is sorted.

For vertical partitioning, the initial map of terms are sorted, as per the algorithm in descending order of support and the order is retained when the terms go into the projection of record chunks in order to put the most frequent terms together in record chunks.

For refine, this method is used to order the contents of the virtual chunks in descending order of support.

4.7 Horizontal Partitioning

For this section of the algorithm, although the pseudo code mentions this is a recursive algorithm, I chose to implement this using a queue system, since any recursive algorithm can be implemented in an iterative way.

```
// This will happen if a cluster contains only terms that have been used before and have the exact same support.
if (allTerms.isEmpty()) {
    finalClusters.add(new Cluster( clusterID: "C" + clusterCounter, dataSet));
    clusterCounter++;
    return;
}
```

Figure 12: Terminating case for dataset with terms with exxact same support

To plan for multiple methods of data splitting, an abstract class was used that contained the common methods that would be used regardless of implementation. This includes a method to gather all terms within the dataset minus the terms within the ignore set and a method to count the support of those terms and return the most common one. The abstract method would be how each implementation deals with the split data.

If the dataset is not initially less than the $maxClusterSize$, else the dataset is returned as a new cluster, then the queue, *ArrayDeque* in Java, initially will consist of an object called a *QueueTerm*, that consists of the initial dataset and an empty set to represent the *ignore* terms.

A while loop, with the condition until the queue is empty, then goes through each queue term and first gathers all the terms within dataset minus the ignore terms. Then, the support of each term is then calculated and sorted in order to produce the most frequent term in the dataset.

There is a small termination clause that if there is no more terms left after the dataset minus ignore, then carry on in the loop. I found this scenario would only occur if a dataset contains only terms that were in the ignore set and dataset.

In this scenario, since the data cannot be split anymore, I have chosen to just return this as a final cluster as shown in Figure 12. These clusters will always be greater than or equal too the $maxClusterSize$, and therefore greater than k, as otherwise when the data was split then if it was less than the $maxClusterSize$, the data would have been made into a cluster in the previous round.

As mentioned, the most frequent term is found in the dataset and then the data is split into a list of records with the most frequent term and a list without. What happens to this split data then depends on the implementation.

4.7.1 Original Splitting

This implementation is based off the original description of horizontal partitioning. This simply checks if either of the split data lists are less than the $maxClusterSize$, and if so, creates a new cluster object with that split dataset. This condition for the dataset that contains the most frequent term is shown in Figure 13. Whilst only the contains dataset code is shown, the same condition applies to the dataset without the most frequent term.

If the split data is still greater than $maxClusterSize$, then the most frequent term gets added to the ignore set and a new queue item containing said dataset and the ignore set is added to the end of the queue.

```
// If either of the clusters are now small enough, we add then to the final cluster list and not to the queue.
// Else add to a queue and repeat.
if (!containsTermCluster.isEmpty()) {
  if (containsTermCluster.size() < maxClusterSize) {
    finalClusters.add(new Cluster( clusterID: "C" + clusterCounter, containsTermCluster));
    clusterCounter++;
  } else {
    // Add this to the set of used terms so it cannot be used again.
    Set<String> usedTermsCopy = new HashSet<>(usedTerms);
    usedTermsCopy.add(mostFrequentTerm);
    dataSetQueue.add(new QueueTerm(containsTermCluster, usedTermsCopy));
  }
}
```

Figure 13: Original horizontal cluster terminating condition for contains cluster

```
// If either cluster is less than k then don't split and just return the data-set.
if (containsTermCluster.size() < k || otherTermsCluster.size() < k) {
  finalClusters.add(new Cluster( clusterID: "C" + clusterCounter, dataSet));
  clusterCounter++;
  return;
}
```

Figure 14: Cluster terminating condition for less than k splitting

4.7.2 Less than k Splitting

This implementation is very similar to the above except has a preliminary check to check if either of the split datasets sizes are of size less than k, then the original dataset before the split is added as a final cluster. This condition is shown in Figure 14.

4.7.3 Multi-round Splitting

The final implementation still had the original splitting terminating conditions, however, first checks if each data-split contains less than k records, before checking the $maxClusterSize$ condition.

If a data-split list size is less than k, then these records get added to another list of remaining records, else they are processed the same as the original splitting. This cluster terminating condition is shown in Figure 15

```
if (containsTermCluster.size() < k) {
  remainingRecords.addAll(containsTermCluster);
```

Figure 15: Cluster terminating condition for multi-round splitting

Once the original dataset has been split, if there are remaining records, i.e. a data-split that had size ¡ k, then horizontal partitioning is called again on the remaining records with an empty ignore terms. This continues until there is no more remaining records to process.

4.8 Vertical Partitioning

As mentioned as part of the *DisassociationUtils* class, the first part of vertical partitioning create the term frequency map of every term within the cluster provided and sorts it in descending order of support.

To create the term chunk, the term frequency map is filtered for all the support values less than k and collected to a separate map. The values are kept as mentioned they are used in refine. The original term frequency map then removes all the terms with support less than k as the rest of the terms are put into record chunks.

The code for creating the record chunks follows the pseudo-code closely (Figure 5 lines 8-15) as there is little room for interpretation or deviation from this. A while loop, with condition that there are no more terms to sort into record chunks, first creates the current chunk, which is just an empty set to ensure no duplicate terms.[5]

Each remaining term is added to a test chunk, empty set to resemble the current chunk, and checked for k^m-anonymity. Since every single term by this point will have support $\geq k$, there is no need to check if a singular term is k^m-anonymous. This is also pointed out in [2].

To check for k^m-anonymity, this requires generating every combination of the terms in the test chunk up to combinations of size m. This is effectively generating a power-set, except only for combinations of up to size m. Initially, the generation of a power-set of the terms was computationally infeasible since as the number of terms, n to generate combinations grew, the time to generate went up exponentially, 2^n. The empty set does not need to be checked so $2^n - 1$.

On top of this issue, was the fact that after each iteration of the for loop, where a new term gets added to the test chunk provided the previous terms were k^m-anonymous, power-set generation would also generate subsets of terms that had already been generated and checked.

For example, the combinations generated from set of terms containing {{a}, {b} {c}} is shown in table 3, where multiple duplicate combinations are generated, including unnecessary ones such as {a, b, c} since if $m = 2$ there and checked each time. The generation of combinations has the biggest impact on performance and the checking of already k^m-anonymous combinations is just unnecessary.

To overcome this problem, I employed a solution that only generates the new combinations based off the new term that was added to the test chunk and using existing combinations that have already been checked and verified for k^m-anonymity. This first involves a set of sets called *checkedCombinations* that is created per record chunk.

Figure 16 shows the method that generates and checks combinations. A while loop, that terminates when a counter reaches the value of m to show that m sized combinations have

[5]The term frequency map keys are itself a set and so there will never be a duplicate term when creating the record chunks.

Term Added	Subsets Generated	Subsets Checked
a	{a}	{a}
b	{a}, {b}, {a,b}	{a}, {b}, {a,b}
c	{a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a,b,c}	{a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a,b,c}

Table 3: Normal power-set generation for terms {a, b, c}

Term Added	Subsets Generated	Subsets Checked
a		
b	{a,b}	{a,b}
c	{a, c}, {b, c}	{a, c}, {b, c}

Table 4: Improved combination generation for terms {a, b, c}

been generated, goes through each of the existing checked combinations, adds the new term to that existing combination, verifies it is k^m anonymous and then saves the checked combination.

Table 4 shows an example of what combinations are generated and checked for the same of terms as table 3. No singular term is generated, since they are added into this method on each pass of the for loop, and also no single term is checked since they all have enough support.

To check if the new combination is k^m-anonymous, I go through each record within the cluster and if that combination exists in that record, that increases the *recordOccurrence* counter. If the counter reaches the value of k, then the method returns false, since the method is named *noKSupport*, to say that combination has enough support. Checking if the *recordOccurrence* already has reached k saves some unnessarary checking as if $k = 10$, therefore the maximum number of records per clusters is 19[6], then the minimum amount of checks would be 10 instead of checking all 19 records.

Each combination is checked as it is generated, meaning that if one combination is not k^m-anonymous, then *isKMAnonymous* will return false and there will be no more generation of combinations.

A temporary list of the new checked combinations, *tempChecked* from Figure 16, holds all the new combinations since adding to a collection that is being iterated through will cause a *ConcurrentModificationException*.

After each term has been processed per record chunk, this ends up with a set of terms that can go into the record chunk. The final step is to project the record chunks from the terms that have been deemed safe to go together within this record chunk.

This goes through each of the sets of terms that will go into a record chunk, i.e. if the list of sets is of size 3, that means there are 3 record chunks in this cluster, and uses the *projectRecords* method as described in the utility class, to create a list of records for that record chunk. Finally, the cluster is updated with the record chunks and the term chunk.

[6]Assuming original data-splitting was used and no clusters were of greater size than maxClusterSize

```java
private boolean isKMAnonymous(Set<Set<String>> checkedCombinations, String newTerm,
                              List<Record> records) {
  // We start at 1 as every single term has enough support and only need to check combinations up to size m.
  int tempM = 1;
  while (tempM != m) {
    Set<Set<String>> tempChecked = new HashSet<>();
    for (Set<String> checkedCombination : checkedCombinations) {
      // Only add to combinations that have not been checked i.e. combinations that already contain the new term.
      if (checkedCombination.size() != tempM || checkedCombination.contains(newTerm)) continue;
      // Add the new term to the existing checked combination to create the new one to check.
      Set<String> combination = new HashSet<>(checkedCombination);
      combination.add(newTerm);
      // Only return false if this combination does not appear k times in the list of records.
      if (noKSupport(combination, tempChecked, records)) return false;
    }
    checkedCombinations.addAll(tempChecked);
    tempM++;
  }
  // All combinations match so this set is k^m anonymous.
  return true;
}
```

Figure 16: Method to generate and check combinations of n terms up to m size combinations

4.9 Refine

For the final section of the algorithm, I first filtered out clusters that did have any terms in their term chunks. This is because refine is about improving the quality of the data by moving common terms between clusters from term chunks to shared chunks and projecting them to records instead of set terms. If there are no terms to refine, then there is no use for these clusters in the refine method. These clusters are mapped to a joint cluster object, each made up of just a single cluster.

As stated in the description of a joint cluster, each simple cluster is mapped to a joint cluster at the start of the refinement process. This also creates the first initial virtual term chunk, by simply taking all the values in in the term chunk, adding them to the virtual term chunk map and setting the initial support of each term to 0.

To update each virtual term chunk, as described in the utility class, all the terms all the now populated virtual chunks have their term chunk support calculated. Then simply go through each virtual term chunk and update each term with the now calculated term chunk support. This was very slow initially using Java 8 streams, so I managed to improve the performance of this update using parallel streams which allows to split the update over many threads. This code is shown in Figure 17.

As noted in the approach of this stage, after each successful joint cluster merge with shared chunks, the term chunk support needs to be updated for every joint cluster that still holds that term. An initial approach was to employ the calculate and update as just described however, this had a big impact on the overall performance of the refine method since this had to re-calculate each term, even though it is more likely only a few terms got refine, and then go through every single virtual chunk and update those that contain the update terms. To address this problem, I used the combination of a what I called a *term map* and a *cluster*

```
private void calculateAndUpdateTermChunkSupport(Set<JointCluster> jointClusterList) {
  // Calculate term chunk support.
  Map<String, Integer> termChunkSupport = DisassociationUtils.countSupport(jointClusterList.stream() Stream<JointCluster>
          // Get the term chunks.
          .map(JointCluster::getVirtualChunk) Stream<Map<String, Integer>>
          // Get all the the terms.
          .map(Map::keySet) Stream<Set<String>>
          // Flatten to one stream.
          .flatMap(Set::stream) Stream<String>
          // Collect to a list.
          .collect(Collectors.toList()));

  // Update each cluster with the new values.
  termChunkSupport.entrySet().parallelStream().forEach(termChunkEntry -> jointClusterList.parallelStream() Stream<JointCluster>
          .map(JointCluster::getVirtualChunk) Stream<Map<String, Integer>>
          .map(Map::entrySet) Stream<Set<Entry<String, Integer>>>
          .flatMap(Collection::stream) Stream<Entry<String, Integer>>
          .filter(entry -> entry.getKey().equals(termChunkEntry.getKey())) Stream<Entry<String, Integer>>
          .forEach(entry -> entry.setValue(termChunkEntry.getValue())));
}
```

Figure 17: Method to initially update each virtual term chunk

map.

The cluster map is a simple map object (*HashMap*) of each joint cluster with its string ID at the key and the actual cluster as the value. The term map is all the terms in all the term chunks, but a set so no duplicates, and a list of the joint cluster IDs in which that term resides.

When a new joint cluster has been produced, a method called *updateTermChunkSupport* is called. As seen in figure 18, this loops through each of the refining terms getting the list of joint cluster IDs, from the term map, that the term is residing in. Then the method loops through that list of joint cluster IDs and gets the joint cluster object from the cluster map. Provided the term still resides within the virtual chunk of that joint cluster, it could not be there if it was removed during the merging process, then the term chunk support value is updated.

This method has a lesser impact on performance than the other method mentioned due to the use of *HashMaps*, since they have $O(1)$ retrieval time. To prevent a *NullPointerException*, I use Java optional which allow for values to be null or "not present" and a check of *isPresent* will confirm if the value is there.

One approach that could be interpreted from the repeat loop is to find the first pair of adjacent clusters that could be merged, merged to create a new joint cluster and then start the loop again. This initial approach is very inefficient as starting the loop again would induce more comparisons between the joint clusters (Figure 9 Line 3), so I opted to go through the set of clusters and merge every adjacent pair before starting the loop again. This has better performance and in the best case scenario after one iteration, reduce the number of joint clusters down by half. This could continue in a hierarchical way till there was just one joint cluster made up of every single cluster.

Sorting the internal virtual chunk was using the utility class as described, and sorting joint clusters based on their virtual chunks was done by a custom comparator as mentioned. However, an unforeseen problem occurred during the sorting of clusters. Due to the custom comparator

```
private void updateTermChunkSupport(Set<String> usedTerms, List<String> usedClusterIDs) {
  // Update term chunk support only for the clusters that contain the terms that were refined.
  long s2 = System.currentTimeMillis();
  for (String usedTerm : usedTerms) {
    List<String> clusterList = termMap.get(usedTerm);
    // Remove the clusters we have just used as we have already updated them.
    clusterList.removeAll(usedClusterIDs);
    for (String clusterID : clusterList) {
      // Find the cluster that contains this ID and update it's virtual chunk value.
      JointCluster jointCluster = clusterMap.get(clusterID);
      // Could be null if used term has already been removed during merging.
      Optional<Integer> support = Optional.ofNullable(jointCluster.getVirtualChunk().get(usedTerm));
      // Don't update padding terms.
      if (support.isPresent() && support.get() == -1L) continue;
      // Update the term with the new cluster size.
      jointCluster.getVirtualChunk().replace(usedTerm, clusterList.size());
    }
  }
  long f2 = System.currentTimeMillis();
  LOGGER.debug("Time for term chunk update {}ms", (f2 - s2));

  // Remove the old IDs from the cluster map.
  usedClusterIDs.forEach(clusterMap::remove);
}
```

Figure 18: Method to update term chunk support during refine

that looks at lists of varying size, i.e. each virtual chunk, and that this is a multi-level comparator comparing both the support (integer) and each term (string) lead to an error. If list one that a term with greater support than list 2 list 1 is greater than this list, but list 2 has the same support as terms in list 3, but has a term that is alphabetically before a term in list 2 then list 3 is greater than list 2. By transitive property, if list 1 ¿ list 2, and list 2 ¿ list then list 1 ¿ list 3, but they can be equal is both list 1 and 3 have the same terms and support. To solve this, I simply add a padding term of support -1 which ensures this scenario never occurs.

Once sorted, the main for loop retrieves adjacent clusters, starting at index 0 and 1 and first retrieves the refining terms. A simple method, shown in Figure 19, that goes through the virtual term chunk of the two selected joint clusters and finds the common terms between them, with the only exception being if their support is -1 to avoid joining the padding terms.

Provided there is at least one refining term, the two clusters are put through the method to represent equation 1 (Figure 8). $|J_{new}|$ was simply calculated by taking the size of every simple cluster and reducing to one value. The rest of the equation parameters are calculated by looping through each refining term and only adding to the equation parameters if the refining term is present within a simple cluster term chunk.

The term support, which was calculated in vertical partitioning, represents $s(t_1) + ... + s(t_n)$ in the equation and so is simply added together. For example, if $k = 5$ and the refining terms are {pickle, board}, with a record support of 3 and 2 respectively, then $s(t_1) + ... + s(t_n)$

```
private Set<String> getRefiningTerms(Map<String, Integer> virtualChunkOne, Map<String, Integer> virtualChunkTwo) {
  Set<String> refiningTerms = new HashSet<>();
  for (Map.Entry<String, Integer> entryOne : virtualChunkOne.entrySet()) {
    for (Map.Entry<String, Integer> entryTwo : virtualChunkTwo.entrySet()) {
      if (entryOne.getKey().equals(entryTwo.getKey()) && (entryOne.getValue() != -1 || entryTwo.getValue() != -1)) {
        refiningTerms.add(entryOne.getKey());
      }
    }
  }
  return refiningTerms;
}
```

Figure 19: Method to gather refining terms

```
private boolean safeMergeClusters(List<Cluster> clusters, Set<String> refiningTerms) {
  double totalItemSupport = 0;
  double totalTermsInTermChunk = 0;
  double jNew = clusters.stream().map(Cluster::getRecords).map(List::size).reduce( identity: 0, Integer::sum);
  Set<Cluster> clusterSet = new HashSet<>();

  for (String term : refiningTerms) {
    for (Cluster cluster : clusters) {
      for (Map.Entry<String, Integer> termChunkEntry : cluster.getTermChunkItemSupport().entrySet()) {
        if (term.equals(termChunkEntry.getKey())) {
          totalItemSupport += termChunkEntry.getValue();
          totalTermsInTermChunk++;
          clusterSet.add(cluster);
          // Found the term, no need to go through a set to find another copy. Move onto next cluster.
          break;
        }
      }
    }
  }
  double totalContainingClusterSize = clusterSet.stream().map(Cluster::getRecords).map(List::size)
          .reduce( identity: 0, Integer::sum);
  return (totalItemSupport / jNew) >= (totalTermsInTermChunk / totalContainingClusterSize);
}
```

Figure 20: Code for equation if to proceed to merging of clusters

would $= 5$.

For the right hand side of the equation, $u_1 + ... + u_m$ is calculated by simply incrementing a counter each time a refining term is found within a term chunk. $|P_1| + ... + |P_m|$ was the trickiest to calculate and it only concerns the combined size of the clusters that contain a refining term. As shown in Figure 20, if a simple cluster contains a refining term, this cluster is added to a set and then the sizes of all the clusters are added together.

Any calculations are done with the Java primitive *double* which allows more precision than *float*. If the adjacent pair of clusters cannot be merged, then those joint clusters are put into a global set of "unmergedClusters" as this method will only return the clusters that did get merged. The for loop will then iterate to test if the joint clusters at index 1 and 2 can be merged, else, after a successful merge, the index will move to 2 and 3 since the joint clusters at index 0 and 1 have been merged.

```
// Remove those terms that got merged from the term chunks in the simple clusters.
List<Cluster> clusterListOne = new ArrayList<>();
for (Cluster cluster : jointClusterOne.getSimpleClusters()) {
  Cluster tempCluster = new Cluster(cluster);
  Map<String, Integer> tempMap = new HashMap<>(cluster.getTermChunkItemSupport());
  tempMap.keySet().removeAll(refiningTerms);
  tempCluster.setTermChunkItemSupport(tempMap);
  clusterListOne.add(tempCluster);
}

List<Cluster> clusterListTwo = new ArrayList<>();
for (Cluster cluster : jointClusterTwo.getSimpleClusters()) {
  Cluster tempCluster = new Cluster(cluster);
  Map<String, Integer> tempMap = new HashMap<>(cluster.getTermChunkItemSupport());
  tempMap.keySet().removeAll(refiningTerms);
  tempCluster.setTermChunkItemSupport(tempMap);
  clusterListTwo.add(tempCluster);
}
```

Figure 21: Creating new clusters to avoid altering existing

Assuming two clusters are able to be merged, then the shared chunks are created. Since they have to be k^m-anonymous, I re-used vertical partitioning in order to create them. The combined original records and the refining terms are used to create a set of records that only contain the refining terms. This effectively creates the records that could go into a shared chunk. These records are move to a temporary new cluster object, just so it can be used within the same vertical partitioning class.

The result is the temporary cluster with have some record chunks, which are the shared chunks since they created in the same way, and a term chunk if not all of the refining terms could be put into a shared chunk.

Any term that did not end up in a shared chunk is removed from the refining terms, as this collection is then used to update the term chunk support as described in *updateTermChunkSupport*. The simple clusters that make up the joint clusters are also updated to remove the terms that were moved into the shared chunks from the term chunks.

Since joint clusters can merge with other joint clusters, I ran into the problem of pass-by-reference in Java. This is shown when altering the term chunks, as when removing terms from the term chunk, if that object is referenced somewhere else, i.e. another joint cluster that contains that simple cluster with the term chunk, then both objects are affected when only one the cluster being merged should be affected. To solve this issue, I create a new instance of a simple cluster, with the same details as the current one, so that it will not affect other objects. This is shown in Figure 21.

The new joint cluster object is created, with its ID being a concatenation of the simple clusters that made up the new cluster, e.g. C_2 and C_{56} become "C_2, C_{56}" and all the simple clusters being added together. This joint cluster is added to the cluster map, while the old clusters are removed.

```java
// Add the merged clusters to the final list.
int oldSize = finalJointClusterSet.size();
finalJointClusterSet.addAll(mergedClusters);
int newSize = finalJointClusterSet.size();

// if no changes in merged clusters then we are done.
if (oldSize == newSize) {
  finalJointClusterSet.addAll(unmergedClusters);
  LOGGER.info("Formula did not pass {}", failed);
  LOGGER.info("Out of {} rounds", rounds);
  return finalJointClusterSet;
}
```

Figure 22: Terminating condition for refine.

This process is repeated for each adjacent cluster until there are no more clusters to merge. The merge method will return a set of merged clusters. The set is to prevent duplicate clusters being merged. This did require over-ridding the *equals* method in the joint cluster in order to tell Java how to tell if one object equals another. The simple solution was if the IDs of two joint clusters were the same, they were considered an equal object.

The entire merge process, i.e. sort virtual term chunks, sort then merge clusters, is done in a *while(true)* loop. The break condition for this loop is if there are no more new merged clusters produced. If at least one new joint cluster is produced, all merged clusters are added to a *finalJointClusterSet*. This is a list of all the merged clusters that will eventually be return by the refine method, but as mentioned in my approach, merged clusters could merge with other clusters, and therefore they are put back into the starting list of clusters, along with all the unmerged clusters from the first iteration, to begin the next iteration.

Once no-more merged joint clusters are produced, any left unmerged joint clusters are added to the final set and then returned. This set of joint clusters gets added to the list of joint clusters that was produced before refine, because they did had empty term chunks and therefore could not be improved as shown in in Figure 22.

A big issue I faced, especially with the biggest dataset, is I would run out of memory. I had to used 8GB of memory in order to complete the algorithm with the biggest dataset at the greatest k. I managed to at reduce the memory foot print by replacing every empty list in a record or shared chunk to null. I do suspect a memory leak, objects which are needlessly referenced and cannot be collected by the garbage collector, but this is very difficult to trace and I did not have time to investigate.

5 Results and Evaluation

One of the main aims of this project was to be able to replicate the test results as shown in [2] to show information loss. For this I had to follow their methodology as close as I could using the same datasets and the same evaluation metrics. These results focus on Relative Error (re), $tlost$ and performance, each of which is explained and evaluated.

5.1 Reconstructed Datasets

In order to calculate the following information loss metrics, I had to reconstruct a dataset from the anonymised one produced by disassociation. It is noted the authors of [2] say they reconstruct one randomly, but this does present a few problems. I opted for a "flattening" strategy when reconstructing a dataset.

1. Shuffle the list of joint clusters to a random order.
2. For each joint cluster
   (a) For each simple cluster
      i. Randomly shuffle the records within each record chunk.
      ii. Concatenate each record together through each record chunk.
      iii. Randomly add a number of term chunks from the term chunk.
   (b) For each shared chunk.
      i. Randomly shuffle the records within each shared chunk.
      ii. Concatenate each record together through each record chunk.
   (c) Add to each record (from the record chunks) a record from the shared chunks.
   (d) Add each completed record to final reconstructed dataset list.
   (e) Check if reconstructed dataset size is equal to the original. Return dataset if true
3. Return reconstructed dataset.

Since this effectively "flattens" the anonymised dataset, this could create a lot of records that existed within the original dataset since the sub-records are within record chunks.

5.2 Variables

For each of the information loss metrics, I changed the value of k using the same values as [2], $3, 5, 10, 15, 20$. $maxClusterSize$ is always $2k$ as noted and m stays constant at 2, as also stated in [2] since it has a negligible effects on the results. Each run is also run with the three horizontal partitioning strategies, original, multi-round and less than k.

5.3 Datasets

Three commonly used datasets were using in testing this algorithm. Described in Figure 23, where $|D|$ is the size of the dataset, $|T|$ is the number of unique terms, and the rest is self explanatory. $WV1$ and $WV2$ are from two different e-commerce sites about click-stream data, and POS is a Point of Sale transaction log from an electronics retailer[2]. These datasets are used commonly in research with anonymised record values.

5.4 Performance

To measure the performance of the algorithm I took a current time-stamp reading before the start of horizontal partitioning and after refine had completed. This is done by using *System.currentTimeMillis()* which gets the current time in milliseconds. To get the time of the algorithm, I simply take away the end from the start value.

5.5 Relative Error (re)

This metric looks at the relative error between the support of term combinations in a re-constructed dataset [2]. This is defined in Figure 24, where $s_o(a, b)$ is the support of terms (a, b) in the original dataset and $s_p(a, b)$ is the support of terms (a, b) in the reconstructed (or published) dataset. The absolute value between the two supports is divided over the average of the two supports. This is described as normalizing re to $[0, 2]$ and avoiding divisions by zero [2].

As noted in [2], there are too many combinations of terms, even if of size 2, that could be created and not exist in the original dataset that would interfere with the result, so the authors found that combinations of them $200^{th} - 220^{th}$ most frequent terms in the original dataset works for giving a good indication of how semi-frequent combinations are retained in the original data.

There is two different measurements for relative error, called re and $re - a$. The difference is that that reconstructed dataset for $re - a$ only creates records from the record and shared chunks, whilst re takes terms from the term chunk as well. $re - a$ covers records that would appear in any reconstructed dataset, since record and shared chunks are just sub-records of the original data.

I have implemented this metric as a separate class called *Relative Error*. The class takes the original and a reconstructed data dataset, both as a list of records. This first counts the support of every term in the original dataset and orders it by support in descending order. Then a sub-list from the 200th (inclusive), to the 220th (exclusive) index and creates every possible combination of the sub-list of size 2 by looping through the same list twice and adding each term to every other term except itself.

It is assumed, since there are many combinations and it is not explicitly stated, that a relative error is calculated for each combination and then the overall relative error is the average of all these values.

Dataset	$\|D\|$	$\|T\|$	max rec. size	avg rec. size
POS	515,597	1,657	164	6.5
WV1	59,602	497	267	2.5
WV2	77,512	3,340	161	5.0

Figure 23: Datasets used for testing [2]

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

Figure 24: Relative error equation

5.5.1 Tlost

The given definition of Tlost is, *"the percentage of terms that have support more than k in the original dataset but they are placed into term chunks by our [disassociation] method"[2].*

I implemented this by first gathering all terms from every term chunk into a set, to prevent duplicate terms. Then I count the support of each term within the dataset and filter for values $\geq k$.

Then I take the set of terms with support $\geq k$ and filter so that it only contains the terms that are in the set of term chunk terms, and then count this list size. Finally I take this value and divide by the size of the set of terms with support $\geq k$ and multiply by 100 to get a percentage.

5.6 Results

5.6.1 Relative Error (re)

For WV1, the relative error from the original paper as shown in Figure 25 is around 1.1 for $k = 5$, whilst my result managed to average at 0.75, as show in Figure 27 for the same value of k and using the original horizontal partitioning. This is an decrease in information loss between the two values and a better result.

For WV2, the relative error from the original paper is around 0.9 for $k = 5$, whilst my result is around 1.3 (Figure 31) for the same value of k and using the original horizontal partitioning. This is a worse result than the original paper since my result has a higher information loss.

For POS, the relative error from the original paper is around 0.3, whilst, my result is around 1.2 (Figure 35). This is three times the information loss.

To explain these, I investigated the number of term chunk terms being added to each record. I conducted the exact same tests, only with a fixed number of term chunk terms to be added. A minimum of 5 and a maximum of the size of the term chunk. The results are as follows.

WV1 - 0.95, closer to the original value.
WV2 - 0.86, closer to the original value.
POS - 0.62, closer to the original value.

This shows that the authors may have a more regulated way of adding terms from a term chunk. Effectively having a minimum and maximum defined. I then looked at the dataset statistics for the original dataset and compared to the reconstructed dataset I had produced. For WV1 and a random number of term chunk terms, the average record size was always greater (3.2) than the one in the original dataset (2.5). This is similar for the max record size

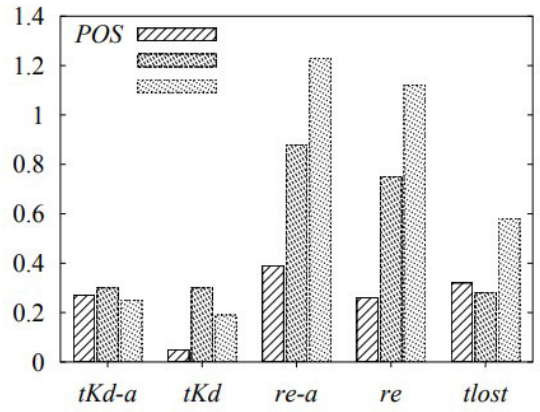

Figure 25: Information loss from [2]. $k = 5, m = 2$

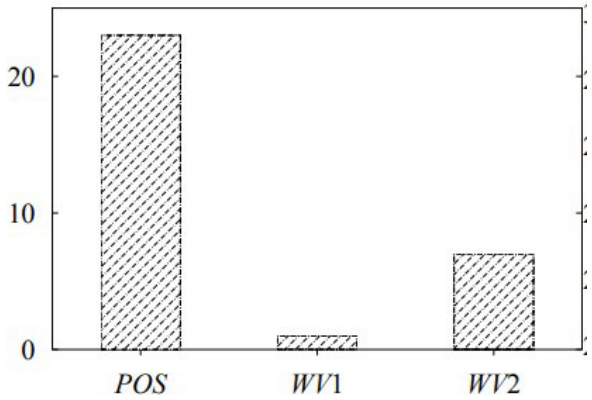

Figure 26: Performance (seconds) of three datasets [2], $k = 5, m = 2$

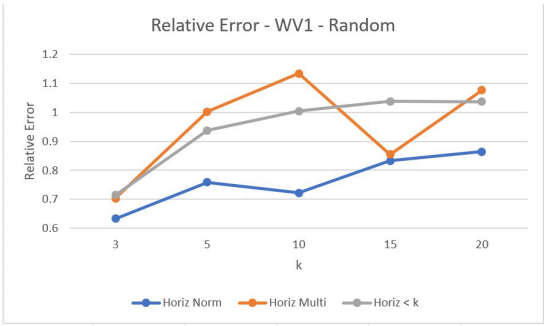

Figure 27: WV1 - re - Random Amount of Term Chunk Terms

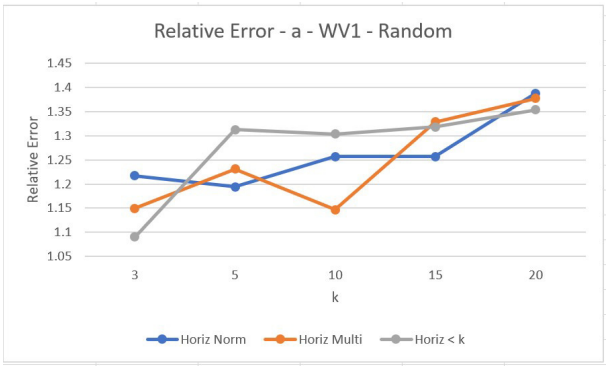

Figure 28: WV1 - $re - a$ - Random Amount of Term Chunk Terms

as well, as my reconstruction was always selecting creating a max record with a greater size than the original. This suggests we are pulling more terms into a record and not regulating the terms.

For WV2, the same applies. The average record length is around 5.5 which is greater than original value of 5, however this is not a big a distinction as WV1 and POS.

For POS, the same applies, which the average record length being pulled around 7.2 compared to the original of 6.5 and it is creating a maximum cluster size of 280.

Horizontal partitioning with multi-round seems to give the better results on average. This is because of preventing clusters with size $< k$ which increases the likelihood that a cluster will contain at least k records that will create at least one record chunk and reduces the number of terms within term chunks.

Horizontal partitioning for $< k$ on average does worse than the original splitting. This is because when the dataset is split and then deemed unsafe, the original dataset size is return as an original cluster. This could potentially be a large cluster with many infrequent terms, which means a large portion of them may end up in a term chunk, despite an increase in record chunks due to more records and a higher chance of terms with at least k support than the original method.

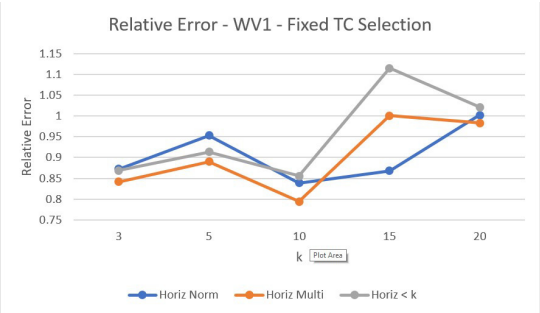

Figure 29: WV1 - re - Fixed Amount of Term Chunk Terms

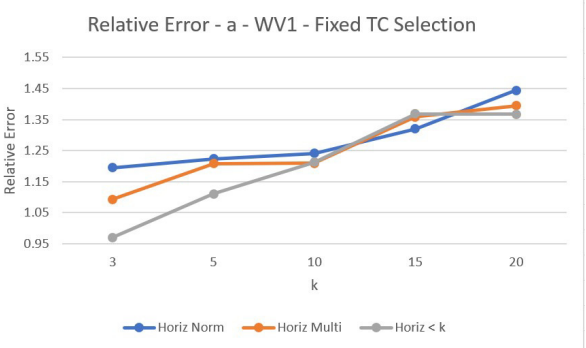

Figure 30: WV1 - $re - a$ - Fixed Amount of Term Chunk Terms

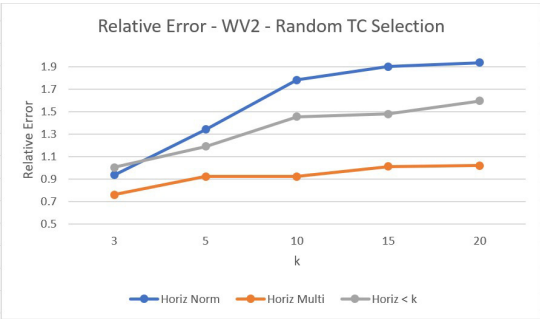

Figure 31: WV2 - re - Random Amount of Term Chunk Terms

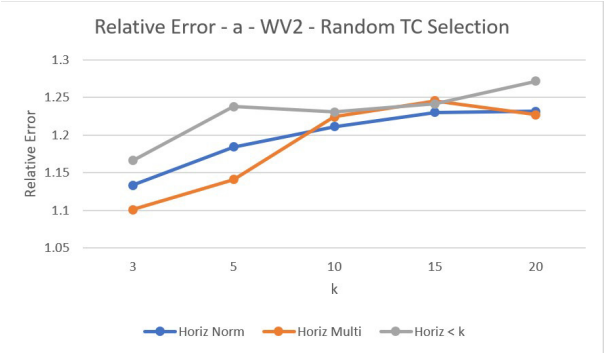

Figure 32: WV2 - $re - a$ - Random Amount of Term Chunk Terms

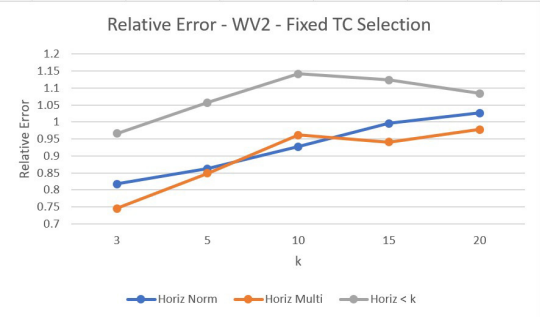

Figure 33: WV2 - re - Fixed Amount of Term Chunk Terms

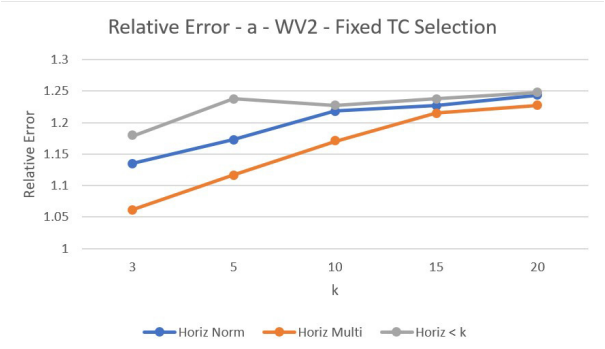

Figure 34: WV2 - $re - a$ - Fixed Amount of Term Chunk Terms

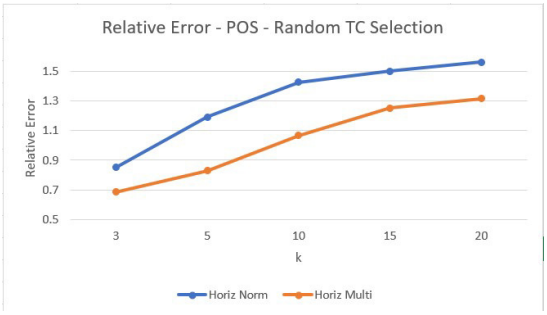

Figure 35: POS - re - Random Amount of Term Chunk Terms

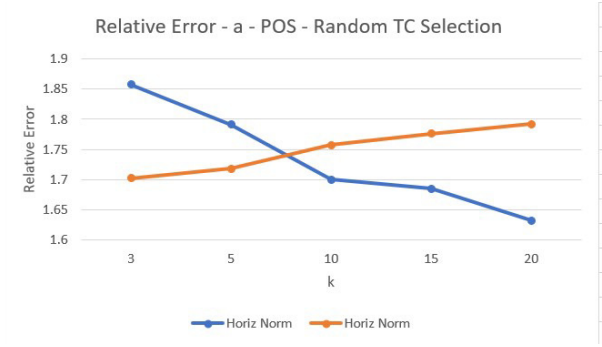

Figure 36: WV2 - $re - a$ - Fixed Amount of Term Chunk Terms

5.6.2 Tlost

For all the three datasets, from my original interpretation of the definition of $tlost$, the result would always be 99% to 100%. I believe this is due to an incorrect interpretation of $tlost$ as their definition could assume that if any term with support $\geq k$ ends up in a term chunk then it is "lost". Take a case where the most popular term in a dataset is "3". Therefore this value will be the most frequent term on the first pass of the original horizontal partitioning and one dataset will contain every record that has the term "3".

However this means that term will never be used again, and assuming the dataset is still $\geq maxClusterSize$ the partitioning will continue splitting all the records that contain the term "3" among different splits. Since this implementation of horizontal partitioning does not consider if a cluster size is $< k$, there is a chance there will be many clusters with $< k$ support, so every term ends up in a term chunk, and could contain even the most frequent term.

Due to this, I ran additional tests on the same datasets with a different interpretation of $tlost$. One considers that, alongside the original definition, that if a term ends up in a shared chunk, even if this term is in a term chunk else where, then the term is not "lost". The same argument could be made for a record chunk and then combined, i.e. a term is "lost" only if it ends up in a term chunk and no where else.

This gave better results that seem to indicate this might have been what the authors intended when they presented this information loss metric. Data is in appendix.

5.6.3 Performance

Due to limited time, the results are limited for just $k = 5$. Their implementation was done in c++. which does allow greater memory control than Java, and with a heuristic solution presented, which allows for many different implementations. As shown in figure 37, this implementation takes longer than the time shown in Figure 26, with a substantial increase for POS. Refine is the slowest method, since it is a heuristic trying to merge thousands of clusters and then even the clusters which have been merged can merge with other clusters, keeping the process going.

k=5	WV1	WV2	POS
Horiz	1338	6050	13049
Vert	143	128	1716
Refine	3003	18732	1342525
Total (ms)	4496	24930	1357290
Seconds	4.4	24.9	1357.29

Figure 37: Time of Disassociation in ms for $k = 5$, original horizontal splitting

6 Future Work

Additional Testing - I would ideally like to test the additional metrics I missed during this paper, namely *Top-K Deviation* tKd which shows how the most frequent item-sets change from the original dataset to the reconstructed one[2]. This allows to look at the most frequent item-sets, since relative error looks at the more semi-frequent item-sets.

Additional to this is to look at the least-frequent item-sets to see how they are preserved through disassociation.

Associative Mining - Although the metrics defined in [2] allow a scientific way of showing the information loss of disassociation, a more practical and real-world way of testing would be to perform associative mining on the original and published dataset.

Then I could show which patterns are retained from the original to the published dataset and which are lost. I believe this would be a more useful metric to show that disassociation is a viable data privacy option that can retain solid privacy whilst still allowing useful data mining.

Additional Horizontal Partitioning Splitting Method - Along with the two additional methods I presented to deal with clusters that have $< k$ records, is another method of handling this situation.

If, after the data-split, either of the datasets contain $< k$ records then this set would be added to the previous dataset in the queue. This is very similar to the method of adding the small datasets to another list to be split after the first iteration of horizontal partitioning is done, except the small datasets will be added to the last item in the queue and not a separate list. I believe that since this is similar to the method described, that the performance will be similar.

Improve Performance for Potential Memory Leak. - The biggest performance issue for this algorithm is how much memory it uses, especially during the refine method. I managed to allow the algorithm to finish with the POS dataset by nullifying the empty array lists in the record and shared chunks, but was still forced to run Java with 8GB of memory to ensure an *OutOfMemoryError* heap exception does not occur.

7 Conclusions

The underlying aim of this project was to implement a data privacy tool for transaction data. This has been accomplished, but the aim to replicate the results has raised the question of how dataset reconstruction is done to achieve the original authors of disassociation. [2].

I have seen that the number of term chunk terms pulled by the reconstruction has an effect on the overall quality of the data and that my reconstruction is, on average, creating records with a greater average length which shows the reconstruction is pulling more terms. However, a reconstructed dataset will never know the metadata and so it is unclear if they are managing the average size of the records as part of their results.

Further investigation is required to see what affects the reconstruction and information loss results.

8 Reflection on Learning

On reflection of this project, despite not achieving the results I had desired, I am pleased with the final implementation of the privacy tool for transaction data. I felt weekly meetings with my supervisor worked very well for keeping me on track and answering any queries I had at the time that helped me advance my progress.

I vastly underestimated the time required for testing and this was mentioned by my supervisor. I believe that whatever time it took to implement the algorithm or any other program, then the time to test it will always be at least double the amount.

This project has taught me above all else about the analytical thinking it requires to study a research paper and more importantly understand it. When I first read [2], which formed the backbone of this project, I did not ask myself why they had implemented their solution that way and simply took it at face value.

One example of this was understanding why the order of terms in vertical partitioning had to be maintained after terms with support $< k$ had been moved to the term chunk. Only to then realise this was because terms with the greatest support have the highest chance of going into a single record chunk and maintain data quality and utility.

Another example of this was understanding that the example of disassociation given in [2] had the incorrect horizontal partitioning. Their example had simply split the data set in two equal clusters when the most frequent term, 'Madonna' was in both clusters. I assumed that their example was correct and did not assess if their algorithm matched up with their examples, but after fully understand horizontal partitioning, I could see that the example was wrong and more importantly why.

My initial approach when planning this project was to do each stage (i.e. Horizontal Partitioning, Vertical Partitioning and Refine) per week and then move onto testing. This approach did work initially as I manage to develop horizontal in a week and vertical in a week, but refine was much more complicated and therefore took longer. I am glad I managed to change my approach to give myself more time for development per stage since although I had developed the total algorithm in reasonable time, I found there were bugs in my code that I then needed development time to address. I believe this has taught me to appreciate that even if I had the pseudo-code for each stage available, that development would not be as simple, especially with the assumptions needed to complete this code.

I believe this project also had an impact on my development skills when it comes to developing complex algorithms or just a complex sub-problem. An example I use is the power-set generation used in vertical partitioning, where the simple approach to generate all combinations and then check each one for k^m-anonymity was infeasible for this project. A recommendation from my supervisor, I changed my approach to the problem differently to only generate new combinations and check each combination as it was generated. The end result is vertical partitioning is the quickest part of disassociation while still performing its function.

Another example would be the custom comparator for the sorting of joint clusters. This is not a common idea to effectively sort by the contents of two map objects and it took me a while to develop it. Even when I had developed the comparator, I ran into the sort error as described in the implementation and had to understand why this was happening and why padding the virtual term chunks fixed the issue.

9 Appendices

Relative Error - WV1					Relative Error A - WV1			
k = 3 | m = 2 | mcs = 6	Horiz Multi	Horiz Norm	Horiz < K			Horiz Multi	Horiz Norm	Horiz < K
	0.841808738	0.872128525	0.868850126			1.093003958	1.195228752	0.969336991
Tlost (Shared)	24.47%	31.01%	17.09%			99.79%	31.01%	
Tlost (Record)	5.49%	14.77%	14.56%					
		4.85%						
k = 5 | m = 2 | mcs = 10	Horiz Multi	Horiz Norm	Horiz < K			Horiz Multi	Horiz Norm	Horiz < K
	0.890048273	0.953181926	0.913294889			1.207744135	1.223717065	1.110332299
Tlost (Shared)	25.66%	45%	17.09%				45%	
Tlost (Record)	6.58%	18%	18.20%					
		6%						
k = 10 | m = 2 | mcs = 20	Horiz Multi	Horiz Norm	Horiz < K			Horiz Multi	Horiz Norm	Horiz < K
	0.79374476	0.838392224	0.855366611			1.209298868	1.241016463	1.21228765
Tlost (Shared)	25.64%	53.85%	12.06%				53.85%	
Tlost (Record)	5.83%	21.21%	21.21%					
		5.36%						
k = 15 | m = 2 | mcs = 30	Horiz Multi	Horiz Norm	Horiz < K			Horiz Multi	Horiz Norm	Horiz < K
	1.000658028	0.867642318	1.114670999			1.357826143	1.31977579	1.368339904
Tlost (Shared)	41.42%	35.54%	11.19%				35.54%	
Tlost (Record)	5.40%	25.74%	25.74%					
		4.90%						
k = 20 | m = 2 | mcs = 40	Horiz Multi	Horiz Norm	Horiz < K			Horiz Multi	Horiz Norm	Horiz < K
	0.983056867	1.001532375	1.020672318			1.39470266	1.443792567	1.367004961
Tlost (Shared)	37.47%	53.92%	11.19%				53.92%	
Tlost (Record)	4.90%	31.39%	30.89%					
		3.54%						

References

[1] Y. He and J. F. Naughton, "Anonymization of set-valued data via top-down, local generalization," *Proc. VLDB Endow.*, vol. 2, pp. 934–945, Aug. 2009.

[2] M. Terrovitis, N. Mamoulis, J. Liagouris, and S. Skiadopoulos, "Privacy preservation by disassociation," *Proc. VLDB Endow.*, vol. 5, pp. 944–955, June 2012.

[3] L. Sweeney, "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, pp. 557–570, Oct. 2002.

[4] L. Sweeney, "Simple demographics often identify people uniquely," *LIDAP-WP4. Carnegie Mellon University, Laboratory for International Data Privacy, Pittsburgh, PA*, 01 2000.

[5] C. Aggarwal, "On k-anonymity and the curse of dimensionality.," *VLDB 2005 - Proceedings of 31st International Conference on Very Large Data Bases*, vol. 2, pp. 901–909, Jan. 2005.

[6] M. Terrovitis, N. Mamoulis, and P. Kalnis, "Privacy-preserving anonymization of set-valued data," *Proc. VLDB Endow.*, vol. 1, pp. 115–125, Aug. 2008.

[7] G. Loukides, J. Liagouris, A. Gkoulalas-Divanis, and M. Terrovitis, "Disassociation for electronic health record privacy," *Journal of biomedical informatics*, vol. 50, pp. 46–61, 2014.

[8] Y. Nader, "Python vs java in 2019 - comparison, features & applications - hackr.io." Available: `https://hackr.io/blog/python-vs-java-2019`, Feb 2019. Accessed 6-May-2019.