Data Cleaning with Minimal Information Disclosure

DATA CLEANING WITH MINIMAL INFORMATION DISCLOSURE

BY

DHRUV GAIROLA, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Master of Science (2015) McMaster University

(Computer Science) Hamilton, Ontario, Canada

TITLE:	Data Cleaning with Minimal Information Disclosure
AUTHOR:	Dhruv Gairola B.Sc., (Computer Science)
SUPERVISOR:	Dr. Fei Chiang
NUMBER OF PAGES:	xii, 134

Dedicated to my family and friends.

Abstract

Businesses analyze large datasets in order to extract valuable insights from the data. Unfortunately, most real datasets contain errors that need to be corrected before any analysis [1]. Businesses can utilize various data cleaning systems and algorithms to automate the correction of data errors [2] [3] [4] [5]. Many systems correct the data errors by using information present within the dirty dataset itself [3] [4]. Some also incorporate user feedback in order to validate the quality of the suggested data corrections [5] [6]. However, users are not always available for feedback [6]. Hence, some systems rely on clean data sources to help with the data cleaning process [7] [8]. This involves comparing records between the dirty dataset and the clean dataset in order to detect high quality fixes for the erroneous data. Every record in the dirty dataset is compared with every record in the clean dataset in order to find similar records. The values of the records in the clean dataset can be used to correct the values of the erroneous records in the dirty dataset. Realistically, comparing records across two datasets may not be possible due to privacy reasons. For example, there are laws to restrict the free movement of personal data [9]. Additionally, different records within a dataset may have different privacy requirements. Existing data cleaning systems do not factor in these privacy requirements on the respective datasets [7] [8]. This motivates the need for privacy aware data cleaning systems. In this thesis, we

examine the role of privacy in the data cleaning process. We present a novel data cleaning framework that supports the cooperation between the clean and the dirty datasets such that the clean dataset discloses a minimal amount of information and the dirty dataset uses this information to (maximally) clean its data. We investigate the tradeoff between information disclosure and data cleaning utility, modelling this tradeoff as a multi-objective optimization problem within our framework. We propose four optimization functions to solve our optimization problem. Finally, we perform extensive experiments on datasets containing up to 3 million records by varying parameters such as the error rate of the dataset, the size of the dataset, the number of constraints on the dataset, etc and measure the impact on accuracy and performance for those parameters. Our results demonstrate that disclosing a larger amount of information within the clean dataset helps in cleaning the dirty dataset to a larger extent. We find that with 80% information disclosure (relative to the weighted optimization function), we are able to achieve a precision of 91% and a recall of 85%. We also compare our algorithms against each other to discover which ones produce better data repairs and which ones take longer to find repairs. We incorporate ideas from Barone et al. [10] into our framework and show that our approach is 30% faster, but 7% worse for precision. We conclude that our data cleaning framework can be applied to real-world scenarios where controlling the amount of information disclosed is important.

Acknowledgements

I would like to profusely thank Dr. Fei Chiang for her invaluable guidance and patience in helping me with research. I would also like to thank Yu Huang for all the interesting discussions we had.

Contents

List of Figures

Chapter 1

Introduction

Data cleaning is a crucial process for organizations that aim to extract valuable information from raw data. Most raw datasets typically contain erroneous information such as misspellings, missing values, etc [1]. Operating on such dirty datasets can lead to a substantial economic loss. The financial impact of poor data quality is approximated at 600 billion dollars per year on US businesses [11]. Thus, it is essential for businesses to have efficient and effective data quality tools to minimize financial loss. For example, British Telecommunications (BT) estimates that such tools provide a business value of more than £600 million to its organization [12]. In fact, the market for data cleaning and data quality systems is growing by 16% every year, higher that the 7% average for the other IT sectors [13].

Data cleaning systems work by automating the correction of data errors. Many data cleaning systems rely on constraints defined on datasets in order to detect errors [3] [4] [7] [14] [15] [16] [17]. These errors are corrected so that the resulting dataset satisfies the constraints. We provide a small example to illustrate how a simple system might work. Imagine a table containing information about employees, shown in Table

Table 1.1: Employee table

ID	**Employee**	**Designation**
r_1	Ted	Designer
r_2	Ted	Designer
r_3	Ted	Developer

1.1. Suppose that a constraint has been defined on this table that states that every unique employee should have a unique designation. We notice that this table does not satisfy this constraint because the employee "Ted" has two different designations. One correction might be to change the designation of r_3 to be "Designer" while another might be to change the designation of r_1 and r_2 to "Developer". Since the value "Designer" has a higher frequency in the table compared to "Developer", an algorithm might correct the designation of r_3 to be "Designer". However, data cleaning is often subjective [6]. It is possible that the designation of r_1 and r_2 should be changed to "Developer" instead, even though the value "Developer" has a low frequency in Table 1.1. Moreover, if two values have an equal frequency within the table, it is not clear how to correct errors within the table. Some systems solve these problems by involving user feedback to filter out undesirable data repairs [5] [6]. However, users are not always available for feedback [6]. Also, if an algorithm cannot proceed unless it receives user feedback and users are unavailable, then this could negatively impact the efficiency of the cleaning algorithm.

Some systems produce data repairs by performing a *record matching* step along with the *data repairing* step [7] [8]. Record matching involves identifying records that refer to the same real-world entity [18] [19] while data repairing refers to finding

Table 1.2: Clean employee table

Employee	**Designation**
Ted	Designer
Teddy	Database Admin

another dataset that minimally differs from the dirty dataset such that the errors in the dirty dataset were fixed [15] [16]. For example, if we have another table that contains correct values, Table 1.2, then we know that "Ted" is really a "Designer". Hence, we can fix the data values in Table 1.1 by changing the designation of r_3 to be "Designer". In the industrial setting, companies like IBM, SAP, Microsoft and Oracle often maintain tables that contain correct and consistent information [8]. The information contained within such tables is referred to as master data, and we refer to such tables as master tables [20]. In this context, we prefer to address the dirty table as the *target table* [3]. Master tables contain high-quality and reliable data, providing users with a unified and synchronized view of its underlying core business entities. Prior work that uses master data for data cleaning assumes that the master data is publicly available [7] [8]. However, master data is often not disclosed publicly. Hence, data privacy is an important concern for systems that involve master data.

Moreover, different records have different privacy requirements. For example, let us assume that a master table consists of names and the postal code associated with each name. Imagine that two records exist in this table: "John Kerry, L4M 5P3" and "John Doe, M4P 8E8". Revealing "John Kerry, L4M 5P3" might be undesirable because "John Kerry" is a famous politician and we do not want to reveal the postal code associated with the politician "John Kerry". In contrast, revealing "John Doe,

M4P 8E8" might be more acceptable because "John Doe" is not a famous personality.

We present a novel data cleaning framework where data privacy is an important concern. Our framework facilitates cooperation between the master data and the target data so that the master data discloses a minimal amount of information and the target data uses this information to (maximally) clean its values. We provide measures to quantify information disclosure and data cleaning utility. These measures are defined over embedded records, which are records that have been obfuscated in some manner. This ensures that our data cleaning algorithm does not operate over the actual records but only over the embedded records. Our data cleaning algorithm involves solving a multi-objective problem consisting of three objectives.

1. The *pvt* objective involves minimizing information disclosure from the master table.

2. The *ind* objective involves maximizing data cleaning utility on the target table.

3. The *changes* objective involves minimizing the number of data updates that are to be performed on the target data.

We model the interaction between our three objectives using four optimization functions. Finally, we perform extensive experiments on real datasets in order to validate our framework. Our results show that disclosing a larger amount of information within the master data helps in cleaning the target data to a larger extent. We find that with 80% information disclosure (relative to the weighted optimization function), we are able to achieve a precision of 91% and a recall of 85%. We also compare our algorithms against each other to discover which ones produce better quality data repairs and which ones take longer to find repairs. We incorporate ideas

from Barone et al. [10] into our framework and show that our approach is 30% faster, but 7% worse for precision. We conclude that our data cleaning framework can be applied to scenarios where master datasets are not publicly disclosed and different records within the master datasets have different privacy requirements.

1.1 Contributions

1. We present a novel data cleaning framework that supports the cooperation between the master data and the target data such that the master data discloses a minimal amount of information and the target data uses this information to (maximally) clean its values.

2. We use ideas from information theory to model information disclosure (by revealing master data) and data cleaning utility (on the target data). Our information disclosure measure is an extension of the measure proposed by Arenas et al. [21]. The data cleaning utility measure was proposed by Dalkilic et al. [22].

3. We define a multi-objective optimization problem based on the information disclosure measure and the data cleaning utility measure, and utilize four optimization functions to model the optimization problem. The four optimization functions are popular methods of modeling multi-objective optimization problems in optimization literature. The solution to our optimization problem is a set of data updates that can be made to the target dataset in order to clean it.

4. We implement our framework and perform extensive experimental evaluation on datasets containing up to 3 million records.

Note that our data cleaning algorithm operates on obfuscated records, and not actual records. This protects the privacy of individual records.

1.2 Thesis outline

The outline of the thesis is as follows. Chapter 2 provides background information about the relational model, functional dependencies and information theory. Chapter 3 describes research that is related to our work. Chapter 4 gives an overview of our framework. Thereafter, we examine each step of the framework in greater detail between Chapters 5 and 8. In Chapter 9, we provide an implementation of our framework and perform extensive experiments on large datasets. Finally, we present our conclusions and avenues for future research in Chapter 10.

Chapter 2

Background

2.1 Relational model

We assume that the target data and master data is stored in tables known as relations. A single row in the table represents a single tuple (or record), and each tuple can be described by a set of columns known as attributes. For some tuple, each attribute can only be associated with a single value from a set of allowed values, known as the attribute domain. The set of tuples in a relation can be denoted by I and the total number of tuples is given by the cardinality of I, denoted by $N = |I|$. One example of a relation is shown in Table 2.1. This target table contains 5 tuples and 7 attributes: tid, $ClinicId$, $FName$, $LName$, $Gender$, DOB and $Illness$. Another table is shown in Table 2.2. This master table contains 3 tuples and the same 7 attributes as the target table.

Many popular database management systems follow the relational model and are ubiquitous in industrial applications [23]. Conceptually, relations are easy to understand and visualize. Often, business owners define rules over tables in an attempt to

Table 2.1: Target table

tid	**ClinicId**	**FName**	**LName**	**Gender**	**DOB**	**Illness**
t_1	542334	Alex	Smith	Female	20081977	Flu
t_2	542334	Alex	Smith	Male	10081989	Allergies
t_3	882081	Barry	Burns	Male	26082004	Flu
t_4	882081	Barry	Burns	Female	26082004	Allergies
t_5	882081	Barry	Burns	Male	26082004	Flu

Table 2.2: Master table

tid	**ClinicId**	**FName**	**LName**	**Gender**	**DOB**	**Illness**
m_1	542334	Alexis	Smith	Female	20081977	Flu
m_2	542334	Alex	Smith	Male	10081989	Allergies
m_3	882081	Barry	Burns	Male	26082004	Flu

control the quality of the data that is stored in the tables. Incoming data that violate these rules need to be examined carefully. These rules are known as constraints, and a variety of constraints exist in the literature [4] [14] [24] [25]. We focus on constraints known as functional dependencies because they are frequently used to enforce data consistency and are essential in helping to maintain and improve data quality [14] [23] [26].

2.2 Functional dependencies

Let us assume that X and Y are two attribute sets in some relation I. A particular functional dependency (FD) over I can be written as $F : X \rightarrow Y$, where X is referred

to as the antecedent while Y is referred to as the consequent. I is said to satisfy F, denoted as $I \vDash F$ if and only if for every pair of tuples t_1 and t_2 where $t_1, t_2 \in I$, $t_1[X] = t_2[X]$ implies that $t_1[Y] = t_2[Y]$. Note that the notation $t_1[X]$ represents a projection of t_1 onto the attributes X. For example, the following two FDs are defined over both the datasets in table 2.1 and table 2.2.

$F_1 : [FName, LName] \rightarrow [DOB]$

$F_2 : [ClinicId, DOB] \rightarrow [Illness]$

F_1 states that a specific first and last name of a patient corresponds to a unique patient date of birth. In table 2.1, "Alex, Smith" corresponds to "20081977" in t_1 while the same "Alex, Smith" corresponds to "10081989" in t_2. This is a violation of F_1, so table 2.1 does not satisfy F_1. On the other hand, table 2.2 does satisfy F_1. In fact, Table 2.2 satisfies both FDs (denoted by Σ, where $\Sigma = \{F_1, F_2\}$).

2.3 Information theory

In this section, we provide a brief background on two concepts: self-information and entropy.

Let us assume a discrete source of information X that can produce outputs from the set $A = \{a_1, ..., a_n\}$. Each output is associated with a probability of occurring, given by $\{p_1, ..., p_n\}$. Self-information, which is the information contained in a particular output, is defined as $I(X = a_j) = \log \frac{1}{p_j}$, where $a_j \in A$ and p_j is the probability of a_j occurring [27]. Self-information, also known as the Shannon information content of an output [28], satisfies the following four conditions [27].

1. The self-information of an output a_j depends on p_j and not the value of a_j.

2. Self-information is a continuous function.

3. If an output $a_i \in A$ is less likely than another output $a_j \in A$, then observing a_i conveys more information than observing a_j i.e., self-information is higher for a_i compared to a_j.

4. If the information about an output a_j can be broken up into two independent outputs a_{j1} and a_{j2}, then the amount of information gained by revealing a_j is equal to the amount of information gained by revealing a_{j1} plus a_{j2} because a_{j1} and a_{j2} are independent pieces of information.

The *entropy* of the information source X is defined as the weighted average of the self-information of all the outputs. Formally, $H(X) = \sum_{j=1}^{n} p_j \log \frac{1}{p_j}$. Entropy captures the "uncertainty" of the information source X. The higher the entropy, the lower the certainty with which we can predict a value from X.

Chapter 3

Related Work

We propose a privacy aware data cleaning framework. Consequently, our work finds similarities with two areas of research: data cleaning and data privacy.

3.1 Data cleaning

There is a large amount of work related to data cleaning using constraints [3] [4] [7] [14] [15] [16] [17]. These constraints express certain requirements and semantics that the data must satisfy. Bohannon et al. use equality classes to group together tuples for which similar updates will be suggested to fix the erroneous values (i.e., to *repair* the erroneous values) [4]. A variety of heuristics are presented that manipulate the content of each equality class based on the cost of adding tuples to the class. All tuples in final set of equality classes will be updated so that all constraints (a set of functional dependencies and inclusion dependencies) are satisfied. Other systems also incorporate equivalence classes but handle a larger variety of constraints [3]. In fact, a large amount of existing research is concerned with data cleaning with a

variety of constraints such as functional dependencies [25], functional dependencies and inclusion dependencies [4], conditional functional dependencies [24], edit rules [16]. However, previous work does not consider data privacy in relation to data cleaning. We define measures to quantify information disclosure and incorporate this directly into our data cleaning algorithm. For example, unlike Bohannon et al. [4], our cost model incorporates a component for information disclosure. Our algorithms try to find repairs that are maximally beneficial in cleaning dirty data while at the same time, suggested repairs minimize information disclosure. Moreover, we utilize four different optimization functions to find data repairs for the target dataset.

User interaction is another important aspect of data cleaning, and some systems involve user feedback in order to improve their algorithms [5] [6]. For example, Yakout et al. use a classifier in order to predict the quality of data repairs [5]. When users select any suggested repairs, this feedback is channelled back into retraining the classifier. Volkovs et al. are able to find repairs to the constraints too, in addition to data repairs [6]. This is accomplished by leveraging a cost model to determine if the data is dirty, or if the constraint itself is incorrect [26]. Again, user feedback is used to retrain the classifier. Our work can be similarly extended to include a user feedback component. For example, all the data repairs that are suggested can be evaluated by the target dataset. If too many of the suggested data repairs are rejected, then the quality of repairs can be improved by automatically increasing the record matching threshold between the target dataset and the master dataset. This will improve the quality of the record matches which will lead to an improvement in the quality of the data repairs.

There are a variety of algorithms that rely on master data for data cleaning [7] [8].

For example, one algorithm uses editing rules (constraints) in order to perform tuple-by-tuple data repairs [7]. These editing rules are defined over both the master data and the target data. Another algorithm proposed by Fan et al. interleaves the record matching and data repairing steps in order to clean data [8]. Fan et al. examine three algorithms: one that relies on user-defined confidence values over the data, another that uses entropy to quantify the relative certainty of data and finally, a heuristic that produces lower quality repairs compared to the first two algorithms. Unlike Fan et al., we do not use matching dependencies to drive the record matching process [8]. However, our algorithm also interleaves the record matching and data repairing steps. For the confidence-based algorithm, users are required to place confidence scores on the data, whereas none of our algorithms assume heavy user involvement [8]. More importantly, our data cleaning algorithm operates over embedded (obfuscated) records. This ensures that our data cleaning algorithm is unable to directly view the data values that are involved within the algorithm. In addition, our data cleaning algorithm tries to minimize the information that is disclosed from the master data while maximizing the data cleaning utility to the target dataset. This is done by solving a multi-objective optimization problem which involves measures for information disclosure and data cleaning utility.

3.2 Data privacy

Data privacy requires that information must be concealed, but there are various interpretations of what constitutes data privacy and different ways to address these concerns.

In the database community, microdata records contain unperturbed information

about specific individuals and microdata publishing is concerned with the sanitization of this information. One area of research focuses on membership disclosure (i.e., checking if a specific individual is present in the published table) [29] [30]. Another popular area of research is focused on sensitive attribute disclosure, that involves concealing the association of quasi-identifier attributes (i.e., those that can serve as an identifier for an individual) with sensitive attributes (i.e., attributes that contain sensitive values like private medical data). The techniques here include generalizing data values (e.g., a person's specific age can be generalized to a numeric range) or suppressing data values in order to satisfy some notion of privacy like k-anonymity or l-diversity. For example, k-anonymity requires that every record with some specific quasi-identifier values be indistinguishable from at least k-1 other records in the published table [31]. However, if all records within this group are associated with the same sensitive attribute value, then privacy can be breached if an attacker knows that some individual is in that group [32]. Unlike k-anonymity, we are more interested in defining a measure that can model the privacy of individual data values. We aim to incorporate our measures in data cleaning applications where both data privacy and data cleaning utility are important.

Closer to our research is work that examines tradeoffs between privacy and data utility [32] [33] [34]. Kifer et al. propose a way to inject utility into k-anonymous and l-diverse tables by publishing additional contingency tables along with the k-anonymous and l-diverse tables [32]. Rastogi et al. define utility as the ability to accurately estimate statistical queries on published private tables [33]. Finally, Brickell et al. consider utility gain for classification tasks over private tables [34]. It is evident

that data utility can be defined in a variety of ways. We are in agreement with Brickell et al. that utility measurements are dependent on the specific task that is being performed on the dataset [34]. We are interested in data utility in the context of data cleaning. We use the information dependency measure to quantify the amount of data cleaning utility on a dataset [22]. We also propose a model for measuring the amount of information that is disclosed by a dataset. Thereafter, a tradeoff between information disclosure and data cleaning utility is modeled as a multi-objective optimization problem, where the information disclosure is minimized while the data cleaning utility (measured using information dependency) is maximized.

To the best of our knowledge, our work is the first of its kind to examine the tradeoff between information disclosure and data cleaning utility. A comparative experiment is detailed in Chapter 9 where we compare SparseMap embedding [35] (used in our framework) with Bourgain embedding [10]. These embeddings are used to perform record matching in our framework. We show that SparseMap is 30% faster, but 7% worse in terms of precision. Additionally, we note that our information disclosure measure is an extension of the measure proposed by Arenas et al. to measure the quality of a database design [21]. The difference is that we incorporate frequency information into the measure. This will be described in Chapter 6.

Chapter 4

Framework Overview

In this chapter, we describe our novel data cleaning framework that facilitates the cooperation between the target and master datasets so that the master dataset discloses a minimal amount of information and the target dataset uses this information to (maximally) clean its data.

Figure 4.1 shows the steps involved in our framework. Three parties are involved in the framework: the master dataset, the target dataset and the third-party. All three parties are assumed to be semi-honest i.e., these parties will follow the steps outlined in our framework but might try to infer additional information if possible.

4.1 Master dataset

The master dataset has two main responsibilities: (i) embedding the master table, and (ii) building an information content table.

The embedding step is a process that obfuscates the values in the records. We embed the records before sending them to the third-party because we want to preserve

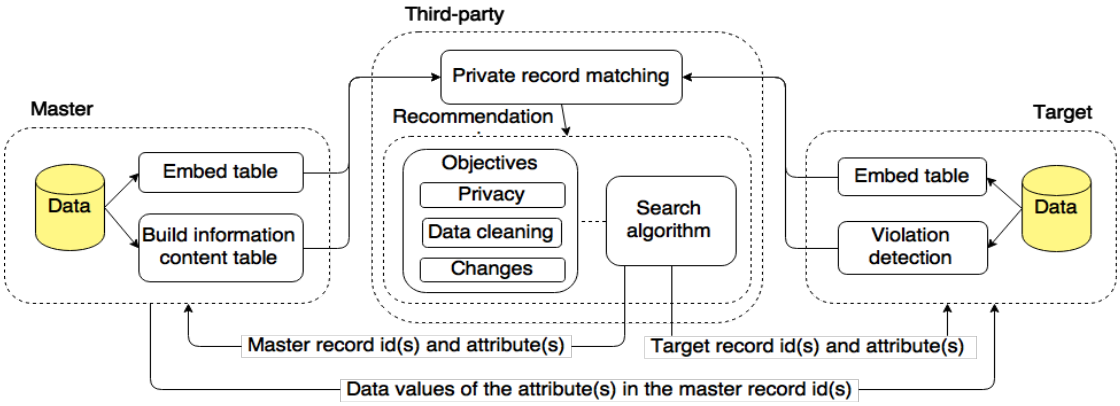

Figure 4.1: Data cleaning framework

the privacy of the records from the third-party (since the third-party is semi-honest).

The master dataset creates an information content table and sends it to the third-party so that the third-party can measure information disclosure when values are revealed from the master table. The information content table duplicates the tuples and attributes in the master table. Every cell in the information content table contains an information content score (a cell is a unit where a row and column intersect). A higher information content score indicates that the corresponding value in the master table contains a high amount of information. Revealing a value with a high information content score represents a high amount of information disclosure by the master dataset. The construction of the information content table is carried out by utilizing our novel privacy model.

4.2 Target dataset

The target dataset has two key responsibilities: (i) embedding the target table, and (ii) detecting violations. The embedding step for the target table is performed in a

similar manner as the embedding step for the master table.

The violation detection step involves finding records that are inconsistent with respect the constraints. The target dataset hopes to clean these records by receiving data value updates.

4.3 Third-party

The third-party consists of two modules: (i) the private record matching module and (ii) the recommendation engine.

Private record matching is performed between the embedded master records and the embedded target violations. This involves comparing the embedded records between the master table and the target table in order to identify embedded records that refer to the same real-world entity. Since the comparison is performed on embedded records (obfuscated records) and not on actual attribute values, privacy is preserved during the record matching phase of the framework. The third-party performs record matching in order to determine which data values in the embedded master table should be sent to the target dataset so that it can clean its data values. However, the third-party does not know the actual data values in either dataset. Only the actual data values can be used to clean the target data. Hence, the third-party has to ask the master dataset to reveal the actual data values directly to the target dataset. The third-party indicates this by sending a set of master record identifiers and the corresponding attributes to the master dataset. This is the central task of the third-party: to determine the set of master record identifiers and attributes that minimize the amount of information disclosed (by the master dataset) while maximizing data cleaning utility (to the target dataset). This is difficult, because the third-party only

knows the embedded records (and the record identifiers) but not the actual record values. We introduce measures to calculate information disclosure and data cleaning utility on the embedded records.

4.3.1 Recommendation engine

The recommendation engine is a module within the third-party whose objective is to find a set of master record identifiers (ids) and attributes that minimize information disclosure (by the master dataset) while maximizing the data cleaning utility (to the target dataset). These record ids and attributes are determined by solving a multi-objective optimization problem that consists of three objectives: an objective to control information disclosure (privacy), an objective to control data cleaning utility, and an objective to control the size of the solution. The search module within the recommendation engine is tasked with finding solutions to the multi-objective problem. Solving the multi-objective problem involves searching within the space of possible master ids and attributes, and finding the set of master record ids and attributes that minimize the three objectives. For a set of master record ids and attributes, information disclosure is determined by referring to the information disclosure table while data cleaning utility is determined by using the information dependency measure. The information dependency measure is based on entropy, so that a lower information dependency is more desirable for data cleaning. By solving the multi-objective optimization problem, the recommendation engine returns a set of master record ids and attributes that should be revealed to the target dataset. The third-party sends the master record ids and attributes to the master dataset, who then reveals the data values directly to the target dataset. However, the target dataset does not know which

tuples should be repaired even after receiving data values from the master dataset. Hence, the third-party indicates the target record ids and attributes that should be updated for each data value that is revealed by the master dataset.

Chapter 5

Private Record Matching

The central contribution of this thesis is our end-to-end data cleaning framework that minimizes the information that is disclosed by the master dataset and maximizes the data cleaning benefit to the target dataset. In this section, we describe how the target dataset discovers violations that need to be cleaned, and how these violations are matched with the master tuples in order to discover similar tuples between the two datasets. By discovering similar tuples, the values of the clean master tuples can be used to clean the errors in the target violations.

5.1 Detecting Violations

Let us assume that some FD $F : X \rightarrow Y$ is defined over some target dataset T. By the definition of an FD, $T \vDash F$ if and only if for every pair of tuples t_1 and t_2, $t_1[X] = t_2[X]$ implies that $t_1[Y] = t_2[Y]$. If $t_1[X] = t_2[X]$ but $t_1[Y] \neq t_2[Y]$, then t_1 and t_2 violate F, and are considered violations. The violation detection procedure is shown in Algorithm 1.

Algorithm 1 calcViolations(T, f): Target dataset T, $f : X \rightarrow Y$
Returns all the violations with respect to F.

1: Define a set V, initialized to $\emptyset$.
2: **for** all pairs $(r_i, r_j) \in T \times T$ **do**
3: **if** $r_i[X] = r_j[X]$ and $r_i[Y] \neq r_j[Y]$ **then**
4: $V \leftarrow V \cup \{r_i, r_j\}$
5: **end if**
6: **end for**
7: **return** V

Algorithm 1 accepts a target dataset T and an FD $f : X \rightarrow Y$ as the input. Note that T here refers to a collection of tuples. In line 1, we define a set V that will contain all the violations. In line 2, we loop over all the pairs of records in T. The violation detection step can be seen in line 3, and all records that pass this check are added to the violation set in line 4. Finally, the set of violations is returned in line 7. In Algorithm 1, we only identify violations where $r_i[X] = r_j[X]$ and $r_i[Y] \neq r_j[Y]$. However, we can easily extend this algorithm to also include the violations that are produced when $r_i[Y] = r_j[Y]$ and $r_i[X] \neq r_j[X]$. This is necessary if we want to discover violations with respect to all the attributes in an FD.

Running example In table 2.1, notice that tuples t_1 and t_2 violate F_1, hence $\{t_1, t_2\} \subset V$, where V contains all the violations. Note that the master dataset M in table 2.2 does not violate any of the constraints because M is a clean dataset.

We have described how T can discover violations with respect to F. We now provide an algorithm to group these violations and order the groups. This ordering determines which groups will be sent for data repair first.

The *orderViolations* algorithm accepts the violations V and an FD $f : X \rightarrow Y$

Algorithm 2 orderViolations(V, f): Violations V, FD $f : X \rightarrow Y$
Orders the violations into violation chunks (groups) by descending size.

1: Define a hashmap D with no keys.
2: Define an empty list G.
3: **for** $r \in V$ **do**
4: $\quad k \leftarrow r[X]$
5: $\quad D[k] \leftarrow D[k] \cup r$
6: **end for**
7: **for** $(k, R) \in D$ **do**
8: $\quad G \leftarrow G \cup R$
9: **end for**
10: sort G in descending order by the size of its elements
11: **return** G

and orders the violations in V into groups. Violations are grouped by the antecedent of the FD and inserted into a hashmap (line 3-6). We refer to each group as a violation chunk. The contents of the hashmap are then added to G (lines 7-9). Finally, G is sorted by the size of its elements (line 10) and returned (line 11). The reasoning is that a larger violation chunk should be processed (by our data cleaning framework) before a smaller violation chunk because T wants to greedily repair as many of its data values as possible.

5.1.1 Ordering a set of FDs

For a set of FDs Σ, we process the FDs in descending order with respect to the number of violations. We process FDs with the most inconsistencies first due to their

possible overlap with other FDs. The FD ordering procedure is shown in Algorithm 3.

Algorithm 3 orderFDs(T, Σ): Target dataset T, a set of FDs Σ
Orders FDs by the number of violations with respect to each FD.

1: Define a hashmap D with no keys.
2: Define an empty list Σ_o.
3: **for** $F \in \Sigma$ **do**
4: $\quad D[F] \leftarrow calcViolations(T, F)$
5: **end for**
6: $D_o \leftarrow$ sort D in descending order by the size of the violations
7: **for** $(f, V) \in D_o$ **do**
8: $\quad \Sigma_o \leftarrow \Sigma_o \cup f$
9: **end for**
10: **return** Σ_o

Algorithm 3 accepts a target dataset T and a set of FDs Σ. In lines 3-5, violations are calculated with respect to every FD F and inserted into a hashmap. The hashmap consists of key-value pairs, where every key is an FD and the value is the number of violations for that FD. In line 6, the hashmap is sorted so that an entry with a higher number of violations is ordered first. In line 7-9, the keys of the ordered hashmap (FDs) are inserted into a list Σ_o and in line 10, this list is returned.

5.2 Record matching for an FD

Once the target dataset has identified the violations with respect to some FD, it would like to fix these violations in order to satisfy the FD. Fixing the violations refers to

updating the data values of the target tuples so that the target dataset is closer to satisfying the current FD. The clean data values that are used to update the target tuples reside in the master dataset. These clean tuples within the master dataset are determined via record matching. For example, imagine that $r_i, r_j \in T$ are violations with respect to some FD $F : X \rightarrow Y$ i.e., $r_i[X] = r_j[X]$ and $r_i[Y] \neq r_j[Y]$. We perform record matching to determine that r_i and r_j are very similar to $r_k \in M$. Hence, if we update $r_i[Y] \leftarrow r_k[Y]$ and $r_j[Y] \leftarrow r_k[Y]$, then we have fixed the violations because $r_i[Y] = r_j[Y]$. These updates are also known as data repairs. Note that it is possible that r_i and r_j may match multiple tuples in the master dataset, and our subsequent examples illustrate this case. In this section, we focus on the record matching process.

Record matching involves comparing records between two datasets in order to detect similar records. This comparison is based on a distance metric, like string edit distance, and involves one or more attributes. Record matching helps us identify the clean master tuples that can be used to repair the violating target tuples. Existing data cleaning research assumes that both the target and master datasets are available publicly [7] [8]. However, there may be data privacy regulations defined over the dataset that restrict the records that are observable by outside parties. For example, consider a hospital that has a master dataset containing patient information for an entire district. Imagine a clinic that wants to clean its target table containing information about its own patients with the help of the hospital records. The hospital cannot simply reveal all its records since they might contain sensitive patient information for specific individuals who are not part of the clinic. Thus, the hospital may not want to disclose all its tuples when performing the comparison in the record matching phase. Instead, the hospital may only want to disclose the tuples

that match across the two datasets while hiding the tuples that do not match. To solve this problem, we introduce a third-party W to help with the record matching. A key condition is that W should not be able to observe the data values in either of the datasets when performing the record matching in order to protect the privacy of the individual records. One way to accomplish this is to embed the data values into a vector space so that the actual values stay hidden to W. Record matching can then be performed on embedded records instead of the actual records.

We apply a metric embedding method known as SparseMap, using a similar approach as Scannapieco et al. [35]. The embedding method is carried out by the target and master datasets and the embedded records are sent to the third-party. The embedding hides the actual data values from the third-party. The embedding method consists of four steps:

1. Building a generator set. This is a set of random strings that are built together by the target and master datasets. The generator set is used to create the reference set in the next step.

2. Building a reference set. The reference set is created using the generator set. Records can be embedded using the reference set. However, embedding will be slower because it involves a large number of string edit distance calculations. Instead, if we embed the reference set first, we can reduce the number of string edit distance calculations by using intermediate euclidean distance calculations.

3. Embedding the reference set. This set is generated from the reference set. Records are embedded using both the reference set and the embedded reference set.

4. Embedding the records. Finally, the target and master records are embedded using both the reference set and the embedded reference set. The target and master datasets send the embedded records to the third-party, who performs record matching using the embedded records in order to protect the privacy of individual records.

5.2.1 Building a generator set

The generator set is a set of strings that will be used to generate a reference set, and all records will be embedded using the reference set. The same generator set is shared by target dataset T and master dataset M. Thus, T and M work together to decide the size of the generator set, N, and the length of each string within the set, x. They then generate a set of N random strings where each random string has a length of x.

Running example Let us assume that T in Table 2.1 wants to embed the violations t_1 and t_2 with respect to the $FName$ attribute and send $t_1[FName]$ and $t_2[FName]$ to W. Similarly, M in Table 2.2 wants to embed m_1-m_3 with respect to $FName$ and send these to W. Let us assume that T and M agree that the generator set size should be equal to 10 and that the length of each random string in the set should be equal to 7. One generator set is (GEtuzDz, DUOcGCk, DZHOgUD, XDtiVZm, JyeruGf, bKLQQID, uOxjCpT, rJmnZaU, OfmUnlC, MJZCIOn). We will use this generator set to build the reference set, and our records will be embedded using the reference set.

5.2.2 Building a reference set

In the second step, the generator set is used to create a reference set so that the records can be embedded using the reference set. Each element of the reference set is a (randomly chosen) subset of the original generator set. The number of subsets is approximated by $\lfloor \log_2 N \rfloor^2$ where N is the size of the generator set. Assuming that the subsets are given by $\{S_1, ..., S_k\}$, each subset S_i has a size 2^q where $q = \lfloor (i-1)/(\log_2 N) + 1 \rfloor$ [35].

Running example One possible reference set corresponding to the earlier generator set is $\{S_1, S_2, ..., S_9\} \leftarrow$ {(DZHOgUD, uOxjCpT), (DZHOgUD, DUOcGCk), ..., (JyeruGf, JyeruGf, bKLQQID, OfmUnlC, DZHOgUD, uOxjCpT, DZHOgUD, DUOcGCk)}. We have 9 elements in the reference set because $\lfloor \log_2 10 \rfloor^2 = 9$.

We now have the reference set and can embed our records using this reference set. Unfortunately, embedding the records using the reference set is an expensive process (known as Bourgain embedding [10]). This is because it involves a large number of string edit distance calculations. An alternative embedding procedure, called SparseMap embedding was proposed so that the records could be embedded in a faster manner [35]. The idea is to embed the reference set itself. Since the embedded reference set consists of numeric vectors while the reference set in Bourgain embedding consists of string vectors, we can use the euclidean distance metric to perform some intermediate calculations and reduce the number of string edit distance calculations needed for embedding the records. In the next step, we describe how the embedded reference sets are created from the reference sets.

5.2.3 Embedding the reference set

The target and master datasets embed the reference set to create embedded reference set because we can use the embedded reference set to speed up computation. For example, we can embed S_1 (from our running example), which is (DZHOgUD, uOxjCpT), as follows:

- (d(DZHOgUD, S_1), d(DZHOgUD, S_2), ..., d(DZHOgUD, S_9)) $\leftarrow (0.0, 0.0, ..., 0.0)$
- (d(uOxjCpT, S_1), d(uOxjCpT, S_2), ..., d(uOxjCpT, S_9)) $\leftarrow (0.0, 6.0, ..., 0.0)$

Another example is embedding S_2 (from our running example), which is (DZHOgUD, DUOcGCk) as follows:

- (d(DZHOgUD, S_1), d(DZHOgUD, S_2), ..., d(DZHOgUD, S_9)) $\leftarrow (0.0, 0.0, ..., 0.0)$
- (d(DUOcGCk, S_1), d(DUOcGCk, S_2), ..., d(uOxjCpT, S_9)) $\leftarrow (6.0, 0.0, ..., 0.0)$

Here d refers to a distance function that accepts a string as the first argument, a set of strings as the second argument and returns the closest distance between its first argument and a member of the second argument. To elaborate, d(DZHOgUD, S_1) is the nearest distance between DZHOgUD and (DZHOgUD, uOxjCpT), which is 0. This is calculated by selecting the minimum edit distance between DZHOgUD and the elements of (DZHOgUD, uOxjCpT). Hence, 0 is the first coordinate of the embedded vector of DZHOgUD. The second coordinate would be d(DZHOgUD, S_2), which is the minimum distance between DZHOgUD and (DZHOgUD, DUOcGCk), which is 0. Thus, we are able to build the vector corresponding to DZHOgUD in S_1. We can follow the same process to calculate the vector corresponding to uOxjCpT in S_1. Since there are no more elements in S_1 to embed, we move on to embedding all

the elements in S_2 as shown, and repeat till $S_1, ..., S_9$ are all embedded to produce $S'_1, ..., S'_9$.

5.2.4 Embedding the records

The final step is for the target and master datasets to embed their records using both the reference set and the embedded reference set. Let us denote A as some attribute within some record r. We embed the value $v = r[A]$ with respect to the embedded reference set by using Algorithm 4.

Algorithm 4 accepts the data value to be embedded v, the reference set $\{S_1, ..., S_n\}$ and the embedded reference set $\{S'_1, ..., S'_n\}$. In line 1, we define an empty array that will contain the embedded data value. In line 2-3, we define two integers used to track some metadata within our algorithm. In line 4, the first coordinate of D is determined by the function $d(v, S_1)$ defined in Section 5.2.3. Lines 5-15 are used to determine the rest of the coordinates of D. In line 5, we loop through each member of the embedded reference set S'_i, starting from the second member. For each numeric vector $o_j \in S'_i$ (line 6), we calculate the euclidean distance between the $i-1$th coordinate of o_j and the $i-1$th coordinate of D and check if it is bigger than integer m (line 7). In simple terms, lines 6-11 are performed to determine the index of the numeric vector p so that the distance between $D[i-1]$ and $o_j[i-1]$ is minimized. In line 12, the string edit distance between v and the string $S_i[p]$ is calculated and added to the embedded data value D. In line 13-14, we just reset the values of the two integers. Finally, the embedded data value D is returned.

Running example From our running example, the target dataset wants to embed $t_1[FName]$ and $t_2[FName]$ in Table 2.1 while the master dataset wants to embed

Algorithm 4 embedValue(v, $\{S_1, ..., S_n\}$, $\{S'_1, ..., S'_n\}$): Data value v, Reference set $\{S_1, ..., S_n\}$, Embedded reference set $\{S'_1, ..., S'_n\}$
Embeds v with respect to $\{S_1, ..., S_n\}$ and $\{S'_1, ..., S'_n\}$.

1: Define an empty array D with n entries.
2: Define integer m as the maximum possible integer value.
3: $p \leftarrow 0$
4: $D[0] \leftarrow d(v, S_1)$
5: **for** $S'_i \in \{S'_2, ..., S'_n\}$ **do**
6: **for** $o_j \in S'_i$ **do**
7: **if** $m > |D[i-1] - o_j[i-1]|$ **then**
8: $p \leftarrow j$
9: $m \leftarrow |D[i-1] - o_j[i-1]|$
10: **end if**
11: **end for**
12: $D[i] \leftarrow$ string edit distance between v and $S_i[p]$
13: $p \leftarrow 0$
14: Reinitialize m to maximum possible integer value.
15: **end for**
16: **return** D

$m_1[FName] - m_3[FName]$ in Table 2.2. We show how to embed $t_1[FName]$, which is "Alex", using the reference set $\{S_1, ..., S_9\}$ and the embedded reference set $\{S'_1, ..., S'_9\}$. The target dataset embeds "Alex" by first computing d("Alex", S_1) i.e., finding the minimum distance between "Alex" and S_1. This is 7.0, following the definition of d in Section 5.2.3. Hence, 7.0 is the first coordinate of the numeric vector that is produced by embedding "Alex". Next, the second coordinate is found by computing:

1. |7.0 - first coordinate of first vector of S'_2| = |7.0 - 0.0| = 7.0
2. |7.0, first coordinate of second vector of S'_2| = |7.0 - 6.0| = 1.0

. The minimum distance of 1.0 was due to the second vector. The second vector in S'_2 corresponds to the second vector in S_2, which is "DUOcGCk". Now we calculate the string edit distance between "Alex" and "DUOcGCk", which is 7.0. This is the second coordinate of the numeric vector that represents the embedded value of "Alex". So far, our embedding of "Alex" has the coordinates (7.0, 7.0). We continue to apply the heuristic by using S'_3, ..., S'_9 and finally obtain the numeric vector for "Alex".

Some of the security properties of the embedding procedure were described by Scannapieco et al. [35]. For example, they showed that the length of the original data values in either T or M are not disclosed to W because W is not involved in creating the reference set and the embedded reference set. Moreover, T and M can control the size of the embedded values that are sent to W by controlling the size of the generator set and the length of each string within the generator set.

5.2.5 Matching

We have described the procedure for embedding data values in the records. T and M embed all the data values in their records and send these to W for record matching.

Note that every embedded record that is sent by T and M is tagged with a record identifier (id) so that W can return information about the matched tuple ids to T and M.

The third-party matches the target violations with the entire master dataset for a set of attributes $\{A_0, ..., A_n\}$. To match a set of violations with the master tuples, W loops through every violation (or symmetrically, every master tuple) and computes the euclidean distance between the embedded data values for the attributes in $\{A_0, ..., A_n\}$. If the distance between the data values for two records for all attributes together in $\{A_0, ..., A_n\}$ is lesser than a user specified similarity threshold τ, then a match has been found. Note that there could be multiple matches for a single target tuple below the matching threshold τ. Formally, let us denote the matches for the attributes of an FD $F : X \rightarrow Y$ by $\mathcal{M}_F = \{(r_m, r_t) | r_m \in M', r_t \in T', match(M', T')\}$ where M' and T' refer to the embedded datasets corresponding to M and T respectively and $match(M', T')$ refers to the matching procedure described in this paragraph.

Running example The matches for the target violations t_1 and t_2 in Table 2.1 with respect to the attributes in $F_1 : FName, LName \rightarrow DOB$ is given by $\mathcal{M}_{F_1} = \{(m_1, t_1) (m_2, t_1), (m_2, t_2)\}$ for $\tau = 0.8$. Here, $(m_1, t_1) \in \mathcal{M}_{F_1}$ means that "Alex, Smith, 20081977" matches "Alexis, Smith, 20081977", although the third-party does not know these values since it only has the embedded values.

We have described how the target and master datasets can embed their records and how the third-party can compute matches on these embedded records. We now present the overall embedding and matching algorithm described in this section. This algorithm is shown in Algorithm 5. Note that in order to improve the running time

of the matching operation, the dimensionality of every embedded data value (which is a numeric vector as shown in our example) in every embedded record is reduced first [35].

Algorithm 5 embeddingAndMatching(T, M, f): Target dataset T, Master dataset M, $f : X \rightarrow Y$

1: $g_0, ..., g_n \leftarrow$ create n random strings of length x

2: $s_0, ..., s_z \leftarrow$ select z random subsets of $g_0, ..., g_n$

3: $s'_0, ..., s'_z \leftarrow$ embed $s_0, ..., s_z$

4: $T' \leftarrow$ embed T w.r.t $s_0, ..., s_z$ and $s'_0, ..., s'_z$ and reduce dimensions from z to k (user defined)

5: $M' \leftarrow$ embed M w.r.t $s_0, ..., s_k$ and $s'_0, ..., s'_k$

6: $\mathcal{M}_F \leftarrow \{(r_m, r_t) | r_m \in M', r_t \in T', match(M', T')\}$

7: **return** $(T', M', \mathcal{M}_F)$

Algorithm 5 accepts the target dataset T, the master dataset M and an FD $f : X \rightarrow Y$. In line 1, the generator sets are created. In line 2, the reference sets are created from the generator sets and subsequently embedded in line 3. In line 4, the target dataset is embedded with respect to the reference set and the embedded reference set. At the same time, the dimensionality of each data value in the target is reduced from z to k following [35]. Next, M is embedded (line 5). In line 6, the embedded datasets are matched with respect to the attributes in f and the matches are produced. Finally, the algorithm returns the embedded target T', embedded master M' and the matches.

After the third-party has computed the matches between the target and master datasets, we want to restrict the matches so that the amount of information disclosure

by the master dataset is minimized while the amount of data cleaning utility to the target is maximized. We decompose the matches into a more discrete form, called *units*. We do this because a single unit is a data update that can be performed on some target tuple and it is easier for us to develop algorithms that manipulate units.

Given some matches $\mathcal{M}_F$ with respect to some FD $F : X \rightarrow Y$, we can convert the matches to units $\mathcal{U}$ where $\mathcal{U} = \{(r_m, r_t, a) | (r_m, r_t) \in \mathcal{M}_F, a \in X \cup Y\}$.

Running example The units corresponding to the earlier matches $\mathcal{M}_{F_1}$ are $\mathcal{U} = \{(m_1,\ t_1,\ FName),\ (m_1,\ t_1,\ LName),\ (m_1,\ t_1,\ DOB),\ (m_2,\ t_1,\ FName),\ (m_2,\ t_1,\ LName),\ (m_2,\ t_1,\ DOB),\ (m_2,\ t_2,\ FName),\ (m_2,\ t_2,\ LName),\ (m_2,\ t_2,\ DOB)\}$. Note that $F_1 : FName, LName \rightarrow DOB$. Each unit is a data update that can be made to a target tuple. For example, a single element in $\mathcal{U}$ e.g., $(m_1,\ t_1,\ FName)$ says that the target tuple t_1 should have its $FName$ attribute value changed to be the same as the $FName$ attribute value in m_1.

Function 6 getUnits($\mathcal{M}_F$, f): Matches $\mathcal{M}_F$, $f : X \rightarrow Y$
Decomposes the matches into discrete units.

```
1: Define a set 𝒰, initialized to ∅.
2: for (r_m, r_t) ∈ ℳ_F do
3:     for a ∈ X ∪ Y do
4:         𝒰 ← 𝒰 ∪ (r_m, r_t, a)
5:     end for
6: end for
7: return 𝒰
```

Function 6 shows how the units can be created from the matches for an FD $f : X \rightarrow Y$. In line 2, we loop through every match, and for every attribute in f, we add the units to a set $\mathcal{U}$ (in line 4). The set of units $\mathcal{U}$ is returned in line 7.

Problem statement Given $\mathcal{U}$, determine $C \in \mathcal{P}(\mathcal{U})$ (where $\mathcal{P}$ is a power set) that minimizes the amount of information disclosure by M and maximizes the data cleaning utility to T (information disclosure and data cleaning utility are described in Chapter 6). Here, we refer to C as a solution that is composed of a set of units. A potential solution is known as a *candidate*, and $\mathcal{P}(\mathcal{U})$ consists of the space of candidates.

In the next chapter, we propose measures for quantifying information disclosure by the master dataset and data cleaning utility to the target dataset and move one step closer to developing an algorithm to solve our problem statement.

Chapter 6

Information Disclosure and Data Cleaning Utility Measures

In this section, we define measures for quantifying information disclosure by the master dataset and data cleaning utility to the target dataset. We need to define these two measures in order to solve our problem statement defined in Chapter 5.

6.1 Information disclosure measure

The master dataset suffers a loss in privacy by revealing its data values. Consequently, the third-party has to calculate the amount of information that is disclosed by the master dataset if the master records in the matches are revealed to the target dataset.

Two popular approaches for measuring data privacy include k-anonymity and l-diversity. These approaches are focused on preventing sensitive attribute disclosure, which involves concealing the association of quasi-identifier attributes (i.e., those that can serve as an identifier for an individual) with sensitive attributes (i.e., attributes

that contain sensitive values like private medical data). k-anonymity requires that every record with some specific quasi-identifier values be indistinguishable from at least k-1 other records in a table [31]. However, if all records within this group are associated with the same sensitive attribute value (i.e., attributes that contain sensitive values like private medical data), then privacy can be breached if an attacker knows that some individual is in that group [32]. To overcome this weakness, a measure called l-diversity was proposed. A table satisfies l-diversity if the probability that any tuple within a quasi-identifier group is linked to a sensitive attribute is a most $\frac{1}{l}$ [36]. However, unlike k-anonymity and l-diversity, we want to model the privacy of individual data values. We want to assign an information content score to each data value so that revealing values with a higher score results in a higher amount of information disclosure and privacy loss. We propose a model for information disclosure by constructing a table that is identical to the master table except that each cell contains an information content score. Since W does not have access to the actual data values in M, M has to build the information content table and send it to W so that the amount of information disclosure can be calculated by W.

The calculation of the information content score for every cell in M is an extension of one method proposed by Arenas et al. [21]. Arenas et al. originally used their information content score to measure the quality of a database design. We are interested in using the information content score to measure the amount of information disclosed by the master dataset.

A	B	C
1	2	3
1	2	4

A	B	C
1	2	3
1	2	4

A	B	C
1	2	3
1	2	4
1	2	5

Figure 6.2: Three tables, I_1(left), I_2 (center) and I_3 (right).

6.1.1 Information content score

We want to calculate an information content score for each data value in the master table. The idea is to calculate how much information is gained if we lose a specific data value in the master table and then some value is replaced back (this could be any value in the attribute domain). For example, in Figure 6.2, let us assume that the FD $F_i : A \rightarrow B$ is defined on all three tables. Let us imagine that the grey cell in I_1 is lost. Since we know that F_i holds on I_1, we can infer that the value of the grey cell should be "2" from the remaining values in attribute B. Hence, the information content of the grey cell in I_1 is low. In contrast, if the grey cell in I_2 was lost, then it is more difficult to infer the value. This is because either "1" or perhaps some other value outside the domain of attribute A could be substituted in and I_2 would still satisfy F_i. Hence, the information content of the grey cell in I_2 should be higher because it is more difficult to infer the value.

Formalizing this idea, for an attribute A, let $adom(A)$ be the active domain of A. The active domain consists of all the values that exist in the master dataset M for the attribute A. As shown in our earlier example, it is possible that some value outside of the active domain can be substituted and the constraints would still hold. Hence, we ensure that at least one value v exists outside the active domain, and use

$adom(A) \cup v$ as the domain of A.

Let $M_{c \leftarrow a}$ denote a database instance constructed from the master dataset M by replacing the attribute value in cell c by a (a cell is a unit where a row and column intersect). We define a probability space for every cell c, denoted by $\mathcal{E}(M, c)$ where $\mathcal{E}(M, c) = (adom(A) \cup v,\ P)$, where P is the probability density function given by :

$$P(a) = \begin{cases} 0 & M_{c \leftarrow a} \not\models F_i \\ \frac{1}{|b|} & otherwise \end{cases}$$

and $b = \{a | M_{c \leftarrow a} \models F_i\}$. The information content of the cell c with respect to $F_i \in \Sigma$ is $inf(c) = H(\mathcal{E}(M, c)) = \sum_{a \in adom(A) \cup v} P(a) \log \frac{1}{P(a)}$.

Running example For the FD F_1, let us consider the cell $m_1[FName]$ in Table 2.2. Here, $adom(FName) \cup v = \{Alexis,\ Alex,\ Barry,\ *\}$, where $v = *$ (some value outside the active domain). For cell $m_1[FName]$, $P(\text{“}Alex''\text{)}$ is 0, while, $P(b) = \frac{1}{3}$ for the rest of the $b \in adom(FName) \cup v$. Hence, $inf(m_1[FName]) = 0 + \frac{1}{3} \log 3 + \frac{1}{3} \log 3 + \frac{1}{3} \log 3 = 1.58$.

The problem with the inf measure is that it does not account for the frequency of the values in the master table. For example, in Figure 6.2, let c_1 denote the grey cell in I_1 while let c_2 denote the grey cell in I_3. Here, $H(\mathcal{E}(I_1, c_1)) = H(\mathcal{E}(I_3, c_2)) = 0$. Intuitively, cell c_2 should contain less information compared to c_1 because I_3 contains more redundancy compared to I_1 [37]. More redundancy means that there is more support (higher frequency) for the values in attributes A and B in I_3 compared to the support for the values in attributes A and B in I_1. We can say with a higher confidence that c_2 contains the value “2” compared to c_1 containing the value “2” because we have higher support in I_3 that c_2 should be “2”. Consequently, it is less

surprising to find out that c_2 contains "2" compared to finding out that c_1 contains "2" because of this higher support. It follows that c_2 contains less information than c_1. Hence, it is important to factor in the effect of frequency of the values into our information content score.

We define a new measure, $einf$, to quantify the amount of information in each cell by accounting for the frequency of the data values in the table. We define a function $freq(a)$ where $freq(a) = \frac{num(a)}{|M|+1}$ where $num(a)$ is the number of cells that contain the value a in the master table M. We add 1 to the denominator because of the value v that lies outside the active domain of the current attribute. Let $P'(a) = freq(a) * P(a)$. Finally, $einf(c) = \sum_{a \in adom(A) \cup v} P'(a) \log \frac{1}{P'(a)}$. Comparing the grey cell c_1 in I_1 with the grey cell c_2 in I_3 in Figure 6.2, $einf(c_1) = \frac{1}{3} \log(\frac{3}{1}) = 0.159$ while $einf(c_2) = \frac{2}{4} \log(\frac{4}{2}) = 0.151$. Since c_2 has a lower $einf$ score, it contains less information compared to c_1, which is what we want.

6.1.2 Information content table

An information content table is calculated by the master dataset owner and sent to the third-party. Every cell of the information content table is associated with an information content score of the corresponding cell in the master table.

Algorithm 7 shows the steps used to construct the information content table. Algorithm 7 accepts the master dataset M, and an FD $f : X \rightarrow Y$. In line 1, the table P is defined as a 2D array, and will eventually be returned by the algorithm. In lines 3-8, the attribute value b is captured in set active domain a while the frequency count for every value is stored in the hashmap D. In line 9, the active domain is augmented with '*'. This ensures that at least one value exists outside the domain.

Algorithm 7 calcInfoContentTable(M, f): Master dataset M, $f : X \rightarrow Y$
Use privacy model to quantify information content of every attribute value in M.

1: Define a $|M| \times |X \cup Y|$ 2-dimensional array P.
2: **for** $b \in X \cup Y$ **do**
3: Define a hashmap D with no keys.
4: Define a set a, initialized to $\emptyset$.
5: **for** $r_m \in M$ **do**
6: $a \leftarrow a \cup r_m[b]$
7: $D[r_m[b]] \leftarrow D[r_m[b]] + 1$
8: **end for**
9: $a \leftarrow a \cup$ '*'
10: D['*'] $\leftarrow 1$

The frequency of the dummy symbol '*' is also added to D in line 10. In line 11, the algorithm loops through the master records. In line 13, the old value of the $r_m[b]$ cell is temporarily stored in a variable p because we are going to be changing the values of $r_m[b]$ and will want to eventually undo all the changes. In lines 14-19, we loop on the active domain, and replace $r_m[b]$ with every value in the active domain one by one. We do this in order to determine the values that can be assigned to $r_m[b]$ so that M will still satisfy f. All these satisfying values are added to a set S in line 17. In line 20, we rollback any changes to $r_m[b]$. In line 22, we loop on the satisfying values. For every satisfying value, we multiply the frequency of that value with $\frac{1}{|S|}$. Then, in line 24, we calculate $p * \log \frac{1}{p}$ and add it to the information content score for the cell $r_m[b]$. Finally, in line 26, we assign the information content score to the cell in the information content table.

Algorithm 7 (continued) calcInfoContentTable(M, f): Master dataset M, $f : X \rightarrow Y$
Use privacy model to quantify information content of every attribute value in M.

for $r_m \in M$ **do**
  Define a set S, initialized to $\emptyset$.
  $p \leftarrow r_m[b]$
  for $v \in a$ **do**
    $r_m[b] \leftarrow v$
    if $M \vDash f$ **then**
      $S \leftarrow S \cup v$
    end if
  end for
  $r_m[b] \leftarrow p$
  $c \leftarrow 0$
  for $v \in S$ **do**
    $p \leftarrow \frac{D[v]}{|M|+1} * \frac{1}{|S|}$
    $c \leftarrow c + p * \log \frac{1}{p}$
  end for
  $P[r_m][b] \leftarrow c$
end for
end for
return P

6.1.3 Measuring information disclosure

We now show how to measure information disclosure by revealing a candidate to the target dataset. For a candidate $C \in \mathcal{P}(\mathcal{U})$, W can calculate the information disclosure for C by referring to the information table. Let us denote information disclosure for C by $pvt(C)$, where $pvt(C) = \sum_{(r_m,a)\in R} einf(r_m[a])$ where $R = \{(r_m, a)|(r_m, r_t, a) \in C\}$.

Running example Recall that we had determined the matches earlier for the tuples t_1 and t_2 in Table 2.1 and then converted these to get the units. The units were given by $\mathcal{U} = \{(m_1, t_1, FName), (m_1, t_1, LName), (m_1, t_1, DOB), (m_2, t_1, FName), (m_2, t_1, LName), (m_2, t_1, DOB), (m_2, t_2, FName), (m_2, t_2, LName), (m_2, t_2, DOB)\}$. One possible candidate $C \in \mathcal{P}(\mathcal{U})$ is C=$\{(m_1, t_1, FName), (m_1, t_1, LName), (m_2, t_2, DOB)\}$. From the definition of $R = \{(r_m, a)|(r_m, r_t, a) \in C\}$, we use C to calculate a set R where R=$\{(m_1, FName), (m_1, LName), (m_2, DOB)\}$. By our definition of information disclosure, $pvt(C) = \sum_{(r_m,a)\in R} einf(r_m[a])$, we can calculate $pvt(C) = einf(m_1[FName]) + einf(m_1[LName]) + einf(m_2[DOB])$.

Function 8 shows how information disclosure is measured. Function 8 accepts a candidate c and the information content table P. In line 3, we iterate over all the units in the candidate. In lines 4-6, if a particular value $r_m[x]$ has not been accounted for yet, then add the information content score $P[r_m][x]$ to p. In line 7, we add (r_m, x) to S, which represents the fact that $P[r_m][x]$ was already accounted for, so we will not account for this value again in the information disclosure score. Finally, in line 9, p is returned as the total information disclosure by revealing c.

Note that the information disclosure measurement is calculated only with respect to the current FD because the $einf$ measure is defined over the current information table (that is built with respect to the current FD). When calculating the information

Function 8 pvt(c, P): Candidate c, InfoContentTable P
Returns the amount of information disclosure by revealing the master data values in the candidate.

```
1: Define a set S, initialized to ∅.
2: p ← 0
3: for (r_m, r_t, x) ∈ c do
4:     if S ∩ (r_m, x) = ∅ then
5:         p ← p + P[r_m][x]
6:     end if
7:     S ← S ∪ (r_m, x)
8: end for
9: return p
```

disclosure with respect to a different FD, the information table will have to be rebuilt with respect to the new FD.

We have formalized the amount of information that is disclosed by the master dataset when a candidate is revealed.

6.2 Data cleaning utility measure

When the third-party reveals master data values to the target dataset, the target dataset uses those data values to clean its own values and move closer to satisfying its constraints. It is incumbent upon the third-party to ensure that the target dataset benefits from the revealed information as much as possible. Hence, the third-party has to quantify the amount of data cleaning utility to the target dataset by revealing master data values. Data cleaning utility is the amount of benefit to the target dataset

A
1
1
1

A
1
1
2

Figure 6.3: Two tables, I_1(left) and I_2 (right).

(in moving closer to satisfying its constraints).

We use the InD (information dependency) measure developed by Dalkilic et al. to quantify data cleaning utility [22]. InD provides a measure of the relationship between the data values in a table. For example, in a weather report, if we know that the current month is "July", then we can be certain that the weather prediction will be "Hot". On the other hand, we are uncertain that the weather prediction will be "Snowing", because snowfall during July is surprising and unexpected. We want to capture this uncertainty about data values from observing related data values.

Entropy is commonly used to capture uncertainty for an attribute in a table. Formally, given a set of data values $A = \{a_1, ..., a_n\}$ with probabilities $\{p_1, ..., p_n\}$, the entropy of attribute A is $H(A) = \sum_{j=1}^{n} p_j \log \frac{1}{p_j}$. Entropy captures the "uncertainty" within A. For example, in Figure 6.3, the entropy of A in I_1 is $H(A) = \log 1 = 0$ while the entropy of A in I_2 is $H(A) = \frac{2}{3} \log \frac{3}{2} + \frac{1}{3} \log \frac{3}{1} = 0.276$. I_2 has a higher entropy because there is more uncertainty about the values in A.

The InD measure uses entropy to quantify the dependency between two attributes, X and Y. It answers the question: how much do we not know about Y, given X? [22]. Formally, the InD for an FD F over T is measured by $i(T) = H(X \cup Y) - H(X)$. If the InD is a non-zero value, this means that T is inconsistent with respect to F.

A higher InD measure indicates that T is more inconsistent with respect to F. Note that the InD of the embedded dataset T' is the same as the InD of the original dataset T. This is possible because the entropy measure uses the distinct values that are present in the table and also the frequencies of those values in the table. Since the embedding process preserves the distances between records in the original space, then theoretically, there is an approximately one-to-one mapping between an original record and the embedded record. Hence, $i(T) = i(T')$.

We define data cleaning utility as the InD on the repaired target table. The third-party simulates the application of data repairs (i.e., applying C to T) and measures the InD of the FD F on the repaired table. For a candidate $C \in \mathcal{P}(\mathcal{U})$, we define a function $apply(C, T')$ that simulates the application of every unit in C to T' and returns the repaired embedded target instance T'_{rep}. We define the data cleaning utility of applying C to T' as $ind(C, T') = i(apply(C, T'))$. The lower the ind, the higher the data cleaning utility.

Running example Continuing our running example, let us consider $C \in \mathcal{P}(\mathcal{U})$ where C=$\{(m_1, t_1, FName), (m_1, t_1, LName), (m_2, t_2, DOB)\}$. We simulate applying C to T in Table 2.1, and this leads to the repaired table shown in Table 6.3. Thus, for F_1, $ind(C, T) = H(FName \cup LName \cup DOB)$ - $H(FName \cup LName) = 0$. This means that $T \vDash F_1$ because ind is 0. Note that in Table 6.3, we show the actual data values in the target table T but in our framework, the algorithm works on the embedded values when calculating ind.

Function 9 shows how ind is calculated. Function 9 accepts a candidate c, the embedded target dataset T' and an FD $f : X \rightarrow Y$. In lines 1-3, we simulate updating $r_t[x]$ to $r_m[x]$, where $r_t[x]$ is an embedded target data value while $r_m[x]$ is an embedded

Table 6.3: Repaired target table where the bold text indicates repaired cells

tid	**ClinicId**	**FName**	**LName**	**Gender**	**DOB**	**Illness**
t_1	542334	**Alexis**	**Smith**	Female	20081977	Flu
t_2	542334	Alex	Smith	Male	**10081989**	Allergies
t_3	882081	Barry	Burns	Male	26082004	Flu
t_4	882081	Barry	Burns	Female	26082004	Allergies
t_5	882081	Barry	Burns	Male	26082004	Flu

Function 9 ind(c, T', f): Candidate c, Embedded target table T', $f : X \rightarrow Y$
Returns the amount of ind on the target by revealing the candidate.

1: **for** $(r_m, r_t, x) \in c$ **do**
2: simulate updating $r_t[x]$ to $r_m[x]$
3: **end for**
4: $i \leftarrow H(X \cup Y)$ - $H(X)$ on the table T'
5: **return** i

master data value in candidate c. In line 4, InD is calculated on the repaired table T' produced by simulating the repairs on T. Finally, the InD is returned in line 5.

We have shown that the data cleaning utility of a candidate can be calculated by utilizing the *ind* measure. We have also shown that the information disclosed by the master dataset can be calculated by using the *pvt* measure. In addition to these two measures, we introduce a third measure, referred to as *changes*, that is defined as $changes = |C|$ where $|C|$ is the number of elements in a candidate set C. Along with minimizing *pvt* and *ind*, we also minimize *changes* because we want the repaired target dataset to be as "close" as possible to the original inconsistent target dataset [38].

Refined problem statement The initial problem statement that was proposed towards the end of Chapter 5 can be refined as follows: Given the units $\mathcal{U}$, determine the candidate set $C \in \mathcal{P}(\mathcal{U})$ such that three objectives are minimized: (i) the *pvt* objective (*pvt*) is minimized for M, (ii) the *ind* objective (*ind*) is minimized for T, and (iii) a minimal number of *changes* are made to T so that it reaches closer to satisfying F. We propose a solution to address this problem via a recommendation engine module, as described in the next chapter.

6.3 Relationship between information disclosure and data cleaning utility

We want to understand the relationship between information disclosure and data cleaning utility in order to identify the conditions that will minimize information disclosure and maximize data cleaning utility.

Information disclosure is minimized when no values are disclosed from the master dataset. However, if no values are disclosed, then the target dataset cannot be repaired. This means that some information has to be disclosed in order to clean the target dataset. Intuitively, a larger amount of information disclosure should lead to a larger amount of data cleaning utility because more values in the target dataset can be repaired if more values are disclosed. However, this relationship is likely to be non-linear, because information disclosure is calculated over the data values in the master table while data cleaning utility is calculated over the entire target table. Practically, we have to perform trade-off analysis to determine how much information has to be disclosed in order to clean the target dataset. We perform experiments

where we vary the amount of information disclosure and observe its effect on the quality of data repairs (Section 9.4.1) and also vary the amount data cleaning utility and observe its effect on information disclosure (Section 9.4.6).

Chapter 7

Finding Optimal Candidates

This chapter describes four methods to solving the multi-objective optimization problem that was described towards the end of Chapter 6. Given the units $\mathcal{U}$, we want to determine the candidate set $C \in \mathcal{P}(\mathcal{U})$ such that three objectives are minimized: (i) the *pvt* objective (*pvt*) is minimized for M, (ii) the *ind* objective (*ind*) is minimized for T, and (iii) a minimal number of *changes* are made to T so that it reaches closer to satisfying the FD F. Our solver finds the candidate C that minimizes the three objectives. Note that C is a set units, and each unit is a data update that can be made to the target dataset. Once the third-party W finds C, it asks M to send the actual data values in C directly to T (since W only has the embedded values, it cannot send actual values to T). T receives the actual data values from M and uses these to clean its erroneous values.

Running example From Table 2.1 and Table 2.2, imagine that our solver minimizes the multi-objective function and finds C={(m_1, t_1, $FName$), (m_1, t_1, $LName$), (m_2, t_2, DOB)} as the optimal solution. For each unit in C, W asks M to send the actual data values to T. For example, for the unit (t_1, m_1, $FName$), M sends $m_1[FName]$

directly to T. W indicates to T that the data value that M sent should be used to update $t_1[FName]$.

The recommendation engine module contains the solver that returns a $C \in \mathcal{P}(\mathcal{U})$ that minimizes all the objectives. Typically, in a multi-objective optimization problem, all the objectives do not reach the minima at the same solution. Moreover, objectives can influence each other. For example, imagine that we have some candidate C that minimizes *ind* but does not minimize *pvt* or *changes*, but we want to also minimize *pvt* and *changes*. We can do so by removing units from C as this will lower *pvt* (since fewer master data values are being disclosed) and will also lower *changes* (since the size of C is smaller). However, imagine that removing any unit or any combination of units from C increases the *ind*. In such a situation, *pvt* and *changes* cannot be decreased without increasing *ind*. Thus, C is a pareto-optimal solution, because the output for a single objective cannot be improved without deteriorating the performance in at least one of the other objectives [39]. A multi-objective problem typically has more than one pareto-optimal solution. Finding all the possible pareto-optimal solutions in the search space yields a pareto-optimal set. For complex problems with a large search space, we are interested in finding a diverse and representative subset of the pareto-optimal set so that a decision maker can decide between a good range of solutions [39]. However, a decision maker can only select a single solution from the set of pareto-optimal solutions that are found [39]. A decision maker performs this selection by using tradeoff analysis to determine which pareto-optimal solution might be preferred. In our context, if our solver finds a set of pareto-optimal candidates $\{C_1, ..., C_n\}$, then we select the pareto-optimal candidate $C_i \in \{C_1, ..., C_n\}$ that has the smallest size because a smaller candidate leads to a

smaller *pvt* output (since fewer values are disclosed) and a smaller *changes* output (since C_i is small) compared to the other candidates in $\{C_1, ..., C_n\}$.

We now describe four methods (four optimization functions) that are used by our solver to model our optimization problem: (i) the weighted method, (ii) the constrained method, (iii) the dynamic method, and (iv) the hierarchical method. We selected the weighted method because it allows us to attach different weights to each of the objectives and control the relative influence of each objective in relation to each other. We selected the constrained method in order to overcome some of the weaknesses of the weighted method (described later). We wanted to explore the dynamic method because it is a variant of the constrained method and requires no user-defined parameters, unlike the constained method. Finally, in the hierarchical method, we arrange the objective functions in order of their importance, where the more important objectives are minimized before less important objectives.

7.1 Weighted method

In the weighted method, all objectives are converted into a single, weighted objective, and the goal is to find a solution that minimizes that weighted objective [39] [40].

$$\begin{aligned} \min \quad & \sum_{i=1}^{k} w_i f_i(x) \\ & x \in S \end{aligned} \tag{7.1}$$

For a set of objectives $\{f_1, ..., f_k\}$, we attach a weight $w_i > 0$ to each objective so that $\sum_{i=1}^{k} w_i = 1$. Note that S refers to the space of possible inputs.

$$\begin{aligned} &\min_{C} && wsum(C) \\ &C \in \mathcal{P}(\mathcal{U}) \end{aligned} \tag{7.2}$$

In the context of our work, the recommendation engine would have to find a candidate C that minimizes the objective $wsum(C)$, where $wsum(C) = \alpha * pvt(C) + \beta * ind(C, T') + \gamma * changes(C)$. The symbols α, β and γ denote the user-defined weights for our three objectives.

One problem with the weighted method is that the whole pareto-optimal set of solutions cannot be found for non-convex problems [39] [40]. This means that there might be some pareto-optimal solutions that can never be reached, no matter how the weights are selected. Our solver does not involve any user interaction so we only require that the pareto-optimal set that is returned is non-empty. As long as the solution set is non-empty, our solver will pick the pareto-optimal solution that has the smallest size because a smaller candidate leads to a smaller *pvt* output (since fewer values are disclosed) and a smaller *changes* output.

Another problem with the weighted method is that deciding on appropriate weights is difficult. The relative influence of one objective on the *wsum* function might be disproportionately higher than another objective, and it is unclear how to precisely set the weights in order to control the relative importance of the objectives. For example, if we only consider two objectives, *pvt* and *ind*, and want to assign both of them equal importance, then we set the weights so that $\alpha = 0.5$ and $\beta = 0.5$ (equal weights). However, imagine that for a given search space of candidates in target T, the *pvt* objective has a mean value of 0.5 and a standard deviation of ± 0.1 while the *ind* objective has a mean value of 0.5 and a standard deviation of ± 0.5. In this case,

the assigned equal weights will not convey equal importance of *pvt* and *ind* because *ind* has a much higher relative influence on the output of *wsum* compared to *pvt* due to its higher standard deviation. We will have to increase the weight for *pvt* and lower the weight for *ind* so that both objectives exert equal influence on *wsum*. A common solution is to test a variety of different weights in order to gauge the relative importance of different objectives and exploring the tradeoffs between the various objectives [40]. In Chapter 9, we have performed experiments where we vary the weights in order to gauge the relative influence of *pvt*, *ind* and *changes* on the accuracy of our results.

7.2 Constrained method

We refer to this method as the constrained method although it is commonly known as the ε-constraint method [39]. In this method, a single objective is chosen to be minimized, while the rest of the objectives are bound by user-defined thresholds.

$$\begin{aligned} &\min && f_l(x) \\ &\text{s.t.} && f_j(x) \leq \varepsilon_j \text{ for all } j = 1, ..., k, j \neq l \\ &x \in S, \varepsilon_j \in \mathbb{R} \end{aligned} \tag{7.3}$$

Here f_l is selected to be minimized while the rest of the objectives are bound by thresholds.

$$\begin{aligned} \min_{C} \quad & pvt(C) \\ \text{s.t.} \quad & ind(C, T') \leq \varepsilon_i \\ & changes(C) \leq \varepsilon_j \\ C \in \mathcal{P}(\mathcal{U}), & \varepsilon_i, \varepsilon_j \in \mathbb{R} \end{aligned} \tag{7.4}$$

In our context, we want to find a candidate C that minimizes $pvt(C)$, subject to $ind(C, T') \leq \varepsilon_i$ and $changes(C) \leq \varepsilon_j$ where ε_i and ε_j are user-defined thresholds.

The advantage of the constrained method over the weighted method is that any pareto-optimal solution (within the pareto-optimal set) can be found for any problem (the problem can be convex or non-convex) [39]. Hence, by changing the constraint thresholds, it is possible to find the entire pareto-optimal set compared to the weighted method where changing the weights does not guarantee that all pareto-optimal solutions will be found for non-convex problems.

The drawback of the constrained method is that for a specific ε_i and ε_j, if there is a good solution really close to ε_i and ε_j but above ε_i and ε_j by a small amount, then it will never be found (unless ε_i and ε_j is increased). In Chapter 9, we have performed experiments where we vary the thresholds and measure their effect on the *pvt* objective.

7.3 Dynamic method

We propose the dynamic method for solving our multi-objective problem, a variant of the adaptive ε-constraint method [41]. In this method, a single objective is chosen to be minimized, while the rest of the objectives are bound by thresholds that are

updated dynamically as the algorithm proceeds.

$$\begin{aligned} &\min && f_l(x) \\ &\text{s.t.} && f_j(x) \leq f_j(x_0) \text{ for all } j = 1, ..., k, j \neq l \\ &x \in S \end{aligned} \tag{7.5}$$

Here f_l is selected to be minimized while the rest of the objectives are bound by thresholds that are updated dynamically.

$$\begin{aligned} &\min_{C} && pvt(C) \\ &\text{s.t.} && ind(C, T') \leq ind(C_0, T') \\ & && changes(C) \leq changes(C_0) \\ &C \in \mathcal{P}(\mathcal{U}) \end{aligned} \tag{7.6}$$

In our context, we want to find a candidate C that minimizes $pvt(C)$, subject to $ind(C, T') \leq ind(C_0, T')$ and $changes(C) \leq changes(C_0)$. We provide an example to illustrate the dynamic method. Imagine that C_0 is the initial starting point of our search algorithm, and $pvt(C_0) = 0.5$, $ind(C_0, T) = 0.5$ and $changes(C_0) = 5$. Hence, the initial threshold on *ind* is set to 0.5 while on *changes*, it is set to 5 i.e., $ind(C, T') \leq 0.5$, $changes(C) \leq 5$. In the next iteration of the search algorithm, imagine that another candidate C_1 is being evaluated, where $pvt(C_1) = 0.4$, $ind(C_1, T) = 0.4$ and $changes(C_0) = 4$. Since C_1 is minimizes all three objectives better than (or equal to) C_0, we update the thresholds so that *ind* is 0.4 while *changes* is 4 i.e., $ind(C, T') \leq 0.4$, $changes(C) \leq 4$. Note that C_1 has to minimize all three objectives more than C_0 in order for the thresholds to be updated.

This advantage of this method over the constrained method is that users do not

have to specify any fixed thresholds because the algorithm is able to update the thresholds dynamically starting from the initial solution.

7.4 Hierarchical method

The hierarchical method is a hybrid method that combines ideas from the constrained method and the lexicographic ordering method described by Fishburn [42]. In the lexicographic ordering method, we have to arrange the objective functions in order of their importance, where the more important objectives are minimized before less important objectives. We start by minimizing the most important objective function. If it has a unique solution, then the solution process stops. If there are multiple solutions, then next most important objective is minimized for the solutions that were found so far. The solution process stops when either a unique solution is found or if all the objectives have been minimized individually one after another.

$$\begin{aligned} &\text{lex min} \quad F(x) = (f_1(x), .., f_k(x)) \\ &x \in S \end{aligned} \tag{7.7}$$

In the lexicographic ordering method, lex min refers to the lexicographic minimization of the k objectives, where f_1 is the most important objective and f_k is the least important objective.

$$
\begin{aligned}
\min_{C} \quad & pvt(C) \\
& \min_{C} \quad ind(C, T') \\
& \text{s.t.} \quad pvt(C) \leq \varepsilon_k \\
& \qquad \min_{C} \quad changes(C) \\
& \qquad \text{s.t.} \quad pvt(C) \leq \varepsilon_k \\
& \qquad \qquad ind(C, T') \leq \varepsilon_l \\
C \in \mathcal{P}(\mathcal{U}), & \varepsilon_k, \varepsilon_l \in \mathbb{R}
\end{aligned}
\tag{7.8}
$$

In our context, we want to minimize the objective $pvt(C)$ first, and then minimize the next objective $ind(C, T')$ subject to $pvt(C) \leq \varepsilon_k$, where ε_k is a user-defined threshold. Lastly, we want to minimize $changes(C)$, subject to $pvt(C) \leq \varepsilon_k$ and $ind(C, T') \leq \varepsilon_l$, where ε_l is also a user-defined threshold. We treat *pvt* as the most important objective since *pvt* is defined over the master table that contains clean and curated data. *ind* is the next important objective, since it is defined over the target dataset that contains dirty data. *changes* is ordered last because it simply restricts the size of the solution that is produced.

The advantage here is that the more important objectives will be minimized before the less important objectives. On the other hand this method runs for a longer time compared to the other methods because the three objectives are minimized one after another, and this requires two additional minimization steps compared to the other methods.

Chapter 8

Repair Algorithm

We described four methods that can be used to model the multi-objective optimization problem outlined in the problem statement. Finally, we have all the background necessary to implement our data repair algorithm.

We recapitulate the steps that we have taken thus far. Our data cleaning framework involves three parties: the target dataset T, the master dataset M and the third-party W. In the first step, T and M embed their records and send them to W. W then performs record matching to match T's violations with M's records. W decomposes these matches into a set of units $\mathcal{U}$. W then has to find the candidate set $C \in \mathcal{P}(\mathcal{U})$ such that three objectives are minimized: (i) the *pvt* objective (*pvt*) is minimized for M, (ii) the *ind* objective (*ind*) is minimized for T, and (iii) a minimal number of *changes* are made to T so that it reaches closer to satisfying F. We proposed four methods to model the minimization of the three objectives. In this chapter, we provide implementations of the four methods so that we can identify the optimal candidate C that will be sent to T for data repairing. Finally, we provide an overall data repairing algorithm that combines all the various components of our

framework described in this paragraph.

8.1 Implementation of the four methods

The four methods that we described in Chapter 7 have to be incorporated into a search algorithm in order to solve the optimization problem. These four methods guide the search algorithm so that the best candidate $C \in \mathcal{P}(\mathcal{U})$ can be found that minimizes our objectives. We select the simulated annealing algorithm as our search algorithm and incorporate the four methods into this search algorithm in order to solve our optimization problem.

8.1.1 Hill climbing algorithm

To motivate our choice of selecting the simulated annealing algorithm, we first briefly describe a related algorithm, called the hill climbing algorithm. The hill climbing algorithm is a simple and popular iterative algorithm that incrementally moves towards better solutions in the search space until a local optima is reached [43]. Starting from an initial input (e.g., a random candidate), the algorithm evaluates every neighbor of that candidate. A neighbor is a candidate that is very similar to the current candidate and all neighbors of the current candidate belong to its neighborhood. If the best neighbor in the neighborhood of the current candidate is better than the current candidate, then it is marked as the current solution. The algorithm then evaluates the neighborhood of the new current solution. This process terminates when no neighbor of the current solution is better than the current solution. A major problem with the hill climbing algorithm is that globally optimal solutions may not be found if

the search process terminates at some local optima. Simulated annealing an algorithm that overcomes this weakness by allowing worse neighbors to be accepted as the current solution with a non-zero probability [44]. Simulated annealing has the same underlying simplicity of the hill climbing algorithm while at the same time, overcomes problem of the search getting trapped in the local optima.

8.1.2 Simulated annealing algorithm

Simulated annealing is inspired by the physical annealing process. Annealing is a metallurgical technique that is used to harden metals by exposing them to a high temperature and then gradually cooling the temperature [43]. This allows the metal to reach a low energy and refine its internal structure. Similarly, simulated annealing accepts a worse solution with a high probability at higher temperatures while gradually lowering this acceptance probability at lower temperatures. The rate that the temperature is lowered is determined by a cooling schedule. If the cooling schedule is lowered slowly enough, the algorithm will find a global optimum with probability approaching 1 [43] [45].

We now describe the simulated annealing algorithm in more detail. Simulated annealing starts with an initial input (e.g., some candidate C), and marks this as the current solution. Next, a single random neighbor C_n of the current solution is selected and compared with the current solution. If C_n is better than C, then it is marked as the current solution. If C_n is a worse than C, then we mark the random neighbor as the current solution with some probability P. Whether C_n is better or worse than C is determined by the objective function output. Each iteration of the algorithm is associated with a temperature K that is determined by a cooling schedule. A

common cooling schedule is $K_n = \beta * K_{n-1}$ where K_n represents the temperature at some timestamp n, while β is a user-defined weight that falls within the range (0,1). If we assume that the objective output of the current solution is c while the output of the random neighbor is n, then P is defined as $P(K, n, c) = exp(\frac{-(n-c)}{K})$. The algorithm iterates until the final (user-defined) temperature is reached (this ensures that the algorithm terminates).

In the simulated annealing algorithm, two important issues have to be addressed: (i) how to initialize a candidate for the simulated annealing algorithm and (ii) how to define a neighborhood for a candidate. Once we address these issues, we can integrate the four proposed methods within the simulated annealing algorithm.

8.1.3 Initializing a candidate

Initialization can affect the quality of the solutions that are found by the simulated annealing algorithm, especially if the search space is very large. Starting at a good initial solution is better that starting at a poor initial solution because the simulated annealing algorithm can only be iterated a fixed number of times (determined by the cooling schedule), and we do not want to waste iterations exploring poorer areas of the search space. Hence, we want to initialize candidates that have a smaller output for the objectives compared to a random candidate since such candidates are good solutions for a minimization problem.

We propose two initialization strategies: greedy initialization and random initialization. The idea with greedy initialization is to create a candidate C_g that is composed of units that involve the most commonly matched master tuple. The intuition is that C_g will likely minimize the objectives more than a random candidate

C. C_g is likely have a lower *ind* output compared to C because when C_g is applied to T, the target violations will be changed to the same values from the single master tuple (this lowers the entropy calculations in the *ind*). Moreover, C_g is likely have a lower *pvt* output compared to C because all the units in C_g involve the same master tuple (information disclosure is low). Random initilization is the same as greedy initialization, except that we randomly remove some units from C_g to create C_r. Since $|C_r| \leq |C_g|$, C_r will have a lower *changes* output compared to C_g by the definition of the *changes* objective.

Running example From our running example, we had determined the units to be $\mathcal{U}$ = $\{(m_1, t_1, FName), (m_1, t_1, LName), (m_1, t_1, DOB), (m_2, t_1, FName), (m_2, t_1, LName), (m_2, t_1, DOB), (m_2, t_2, FName), (m_2, t_2, LName), (m_2, t_2, DOB)\}$. Since m_2 is most commonly matched tuple, greedy initialization will create the candidate $C_g \in \mathcal{P}(\mathcal{U})$, where C_g=$\{(m_2, t_1, FName), (m_2, t_1, LName), (m_2, t_1, DOB), (m_2, t_2, FName), (m_2, t_2, LName), (m_2, t_2, DOB)\}$. Notice that all the units in C_g involve the same master tuple m_2. Random initialization might create a candidate $C_r \in \mathcal{P}(\mathcal{U})$, where C_r=$\{(m_2, t_1, FName), (m_2, t_1, DOB), (m_2, t_2, FName)\}$. Notice that all the units in C_r also involve m_2. $|C_r| < |C_g|$ because C_r was created from C_g by randomly removing some units.

Algorithm 10 shows the greedy initialization procedure. The algorithm accepts units u' and returns a candidate c. In lines 4-6, we compute of how many times a particular master record id is seen and store this information in a hashmap. In line 7, we get the best master record id by determining the key with the highest value in D. In lines 8-12, we loop on the units, and any unit that involves the best master match is added to a set c. Finally, c is returned in line 13.

Algorithm 10 initialize(u'): Units u'
Greedy initialization: returns an initial starting point for the search algorithm.

Define a set c, initialized to $\emptyset$.
Define an empty list z.
Define a hashmap D with no keys.
for $(r_m, r_t, x) \in u'$ **do**
    $D[r_m] \leftarrow D[r_m] + 1$
end for
$r_b \leftarrow$ key from D that has the highest value.
for $(r_m, r_t, x) \in u'$ **do**
    if $r_m = r_b$ **then**
        $c \leftarrow c \cup (r_m, r_t, x)$
    end if
end for
return c

Algorithm 11 initialize(u'): Units u'
Random initialization: returns an initial starting point for the search algorithm.

Define a set c, initialized to $\emptyset$.
Define an empty list z.
Define a hashmap D with no keys.
for $(r_m, r_t, x) \in u'$ **do**
    $D[r_m] \leftarrow D[r_m] + 1$
end for
$r_b \leftarrow$ key from D that has the highest value.
for $(r_m, r_t, x) \in u'$ **do**
    if $r_m = r_b$ and $randDouble > 0.5$ **then**
        $c \leftarrow c \cup (r_m, r_t, x)$
    end if
end for
return c

Algorithm 11 shows the random initialization algorithm. This algorithm is similar to Algorithm 10 except for line 9, where an additional condition is added $randDouble > 0.5$. This condition only allows the unit (r_m, r_t, x) to be added to the initial candidate with probability 0.5.

8.1.4 Defining a neighborhood

The simulated annealing algorithm requires that a random neighbor C_n of the current candidate C is explored and that C_n and C are compared. If C_n minimizes the objectives more than C, then the search algorithm updates the current candidate to C_n. Otherwise, the search algorithm only updates the current candidate to C_n with a probability $P(K, n, c) = exp(\frac{-(n-c)}{K})$, where K is the current temperature, n is the objective output of C_n and c is the objective output of C. In this section, we describe how C_n is determined for a given C.

A neighborhood of C is a set of k candidates $\{C_1, ..., C_k\}$ where each $C_n \in \{C_1, ..., C_k\}$ is a neighbor of C. A neighbor C_n is very similar to C. For example, we define C_n as a candidate that differs from C by only one unit i.e., $C_n \cup U = C$ or $C_n = C \cup U$ where U is a set containing a single unit so that $|U| = 1$.

Running example From our running example, we had determined an initial candidate $C_g \in \mathcal{P}(\mathcal{U})$, where C_g=$\{(m_2, t_1, FName), (m_2, t_1, LName), (m_2, t_1, DOB), (m_2, t_2, FName), (m_2, t_2, LName), (m_2, t_2, DOB)\}$. If the neighborhood of C_g consists of all candidates that differ from C_g by one unit, then one neighbor of C_g is C_g=$\{(m_2, t_1, FName), (m_2, t_1, LName), (m_2, t_1, DOB), (m_2, t_2, FName), (m_2, t_2, LName)\}$. The difference between C_g and C_n is only one unit, which is (m_2, t_2, DOB).

Our present definition of C_n is incomplete, because it ignores a large group of similar candidates to C. It is important to consider all similar candidates to C in its neighborhood because otherwise, some candidates in will never be explored by the simulated annealing algorithm i.e., a large area of the search space will be ignored.

Running example Recall that we had initialized our current candidate to C_g={(m_2, t_1, $FName$), (m_2, t_1, $LName$), (m_2, t_1, DOB), (m_2, t_2, $FName$), (m_2, t_2, $LName$), (m_2, t_2, DOB)}. If the neighborhood of C_g only consists of candidates that differ from C_g by one unit, then C'_n={(m_1, t_1, $FName$), (m_1, t_1, $LName$), (m_1, t_1, DOB), (m_2, t_2, $FName$), (m_2, t_2, $LName$), (m_2, t_2, DOB)} can never be a neighbor of C_g because it differs from C_g by three units, which are (m_1, t_1, $FName$), (m_1, t_1, $LName$), (m_1, t_1, DOB). However, C'_n is clearly a neighbor of C_g because they are very similar to each other. The only difference is that C_g recommends data updates to t_1 using values from m_2 while C'_n recommends data updates to t_1 using values from m_1. If we do not expand our current definition of the neighborhood, then C'_n will never be evaluated by the simulated annealing algorithm.

We define a neighbor C_n of C as follows: (i) C_n differs from C by one unit (C_n is called the unit neighbor) or (ii) C_n and C have the same number of units, except that at most one of the target tuples (in C_n) has update recommendations from a different master tuple (C_n is called the choice neighbor). Before describing our algorithm to find C_n for C, we introduce the notion of *validity*. A candidate is *valid* if and only if every target tuple involved within the candidate is associated with a single master tuple within the candidate. We provide an example to illustrate the importance of validity.

Running example Let us consider C_g from our previous example where C_g={(m_2,

t_1, $FName$), (m_2, t_1, $LName$), (m_2, t_1, DOB), (m_2, t_2, $FName$), (m_2, t_2, $LName$), (m_2, t_2, DOB)}. One of the neighbors of C_g is C_g={(m_1, t_1, $FName$), (m_2, t_1, $FName$), (m_2, t_1, $LName$), (m_2, t_1, DOB), (m_2, t_2, $FName$), (m_2, t_2, $LName$), (m_2, t_2, DOB)} because C_n and C_g differ by only one unit, which is (m_2, t_1, $FName$). However, in C_n, the target tuple t_1 is associated with two different master tuples m_1 and m_2. The two data updates that are recommended by C_n conflict with each other i.e., we do not know if $t_1[FName]$ should be changed to $m_1[FName]$ or $m_2[FName]$. Also, let us imagine that in some iteration of the search procedure, the current candidate is updated so that C= {(m_1, t_1, $FName$)}. One neighbor of C is C'_n= {(m_1, t_1, $FName$), (m_2, t_1, $LName$)}. However, C'_n is an invalid candidate because we are applying data updates to t_1 from different master tuples (even though the updates are on different attributes).

Function 12 checks if a candidate is valid. It accepts a single candidate c as the input and returns $true$ if c is valid and $false$ otherwise. In line 1, we define a hashmap D. Iterating over the units in c (line 2), we first check if there was any master tuple r'_m associated with r_t (line 3). If there was a master tuple associated with r_t but it is not the same as the current master tuple r_m (line 4), then this means that r_t is associated with two different master tuples r'_m and r_m. Hence, c is not valid (line 5). In line 7, we store the current master tuple r_m associated with r_t. Finally, if every r_t is associated with the same r_m, then c is valid.

Finally, we show the algorithm to get a neighbor of a candidate in Algorithm 13. Algorithm 13 accepts a candidate c, and the units u' and returns a neighbor of c. In line 1, we define a set N that will contain the neighborhood of c. In lines 3-7, we add all neighbors of c to N that are smaller than c by one unit. We iterate over the

Function 12 isValid(c): Candidate c
Checks if every target tuple in c is associated with a single master tuple.

1: Define a hashmap D with no keys.
2: **for** $(r_m, r_t, a) \in c$ **do**
3: $r'_m \leftarrow D[r_t]$
4: **if** $r'_m \neq \emptyset$ and $r'_m \neq r_m$ **then**
5: **return** $false$
6: **end if**
7: $D[r_t] \leftarrow r_m$
8: **end for**
9: **return** $true$

elements in c (line 3), and create neighbors (line 4) that are smaller than c by one unit (line 5), and add these to the neighborhood (line 6). In lines 8-14, we add to N all neighbors of c that are bigger than c by one unit. We iterate over the units (line 8), creating a neighbor of c (line 10) and adding the unit to that neighbor so that it is bigger than c by one unit (line 11). If this neighbor is valid (line 12), then we add it to N (line 13). Note that line 9 is for storing information that is used after line 15. In lines 16-25 we add all neighbors of c where every neighbor has the same number of units as c, except that at most one of the target tuples (in each neighbor) has update recommendations from a different master tuple. In line 16, we iterate over D, which contains key-values pairs where a key k is a target tuple, and a value V is the set of master tuples associated with k. For a target tuple k, we iterate on every master tuple $v \in V$ (line 17), create a neighbor of c (line 18), and for each unit in the neighbor (line 19), we find the units where the target tuple is the same as k (line 20), and replace those units with new units that contain the new master tuple $v \in V$

(line 21). We add the neighbor to N in line 24. Finally, we randomly pick a single neighbor from the neighborhood and return it (line 27).

Algorithm 13 getNeighbor(c, u'): Candidate c, Units u'
Returns a random neighbor of c from its neighborhood.

1: Define a set N, initialized to $\emptyset$.
2: Define a hashmap D with no keys.
3: **for** $(r_m, r_t, a) \in c$ **do**
4: $c' \leftarrow$ create copy of c.
5: remove (r_m, r_t, a) from $c'[i]$
6: $N \leftarrow N \cup c'$
7: **end for**
8: **for** $(r_m, r_t, a) \in u'$ **do**
9: $D[r_t] \leftarrow D[r_t] \cup r_m$
10: $c' \leftarrow$ create copy of c.
11: $c' \leftarrow c' \cup (r_m, r_t, a)$
12: **if** $isValid(c')$ **then**
13: $N \leftarrow N \cup c'$
14: **end if**
15: **end for**

The way that we define the neighborhood size in the simulated annealing algorithm influences the search results that are produced. If the neighborhood size is too small, then we will have to iterate the algorithm a higher number of times in order to explore a larger area of the search space. However, if the neighborhood size is too large, then the algorithm essentially performs a random search over the search space [46]. One solution is to compare different neighborhood sizes and measure its effect

Algorithm 13 (continued) getNeighbor(c, u'): Candidate c, Units u'

```
16: for (k, V) ∈ D do
17:     for v ∈ V do
18:         c' ← create copy of c.
19:         for (r_m, r_t, a) ∈ c' do
20:             if r_t = k then
21:                 (r_m, r_t, a) ← (v, r_t, a)
22:             end if
23:         end for
24:         N ← N ∪ c'
25:     end for
26: end for
27: return n ∈ N
```

on the search results. For example, we can simply expand our current definition of a neighbor as follows: (i) C_n differs from C by k unit(s) or (ii) C_n and C have the same number of units, except that at most l of the target tuples (in C_n) have update recommendations from a different master tuple. k and l refer to user defined values that can be adjusted in order to obtain different neighborhood sizes. Although we have not explored the effect of different neighborhood sizes in our study, we hope to perform this experiment at a later date.

8.1.5 Implementation of simulated annealing

We have addressed two important issues related to the simulated annealing algorithm: (i) how to initialize a candidate and (ii) how to define a neighborhood for a candidate. In this section, we implement the simulated annealing algorithm to solve the four objective functions described in Chapter 7. We describe a function called *calcSolns* that contains the implementation of the simulated annealing algorithm. Two flavors of *calcSolns* are provided: Algorithm 14 is for the weighted, constrained and dynamic methods while Algorithm 18 is for the hierarchical method. We separate the two flavors because the simulated annealing algorithm for the hierarchical method requires a slightly different implementation from the other three methods.

Simulated annealing for the weighted, constrained and dynamic methods

Algorithm 14 describes the simulated annealing algorithm for the weighted, constrained and dynamic methods. This algorithm accepts an embedded target table T', an embedded master table M', the information content table P, the units u', an FD $f : X \rightarrow Y$ and returns a set of pareto-optimal solutions s. In line 1, we define a set

Algorithm 14 calcSolns (T', M', P, u', f): Embedded target table T', Embedded master table M', InfoContentTable P, Units u', $f : X \rightarrow Y$
Calculate all solutions to the multi-objective optimization problem for three methods: weighted, constrained, and dynamic.

Define a set s, initialized to $\emptyset$
$c \leftarrow initialize(u')$
for $i \leftarrow 1$ to ∞ **do**
    $t \leftarrow schedule[i]$
    if $t = 0$ **then**
        return s
    end if
    $n \leftarrow getNeighbor(c, u')$
    $c \leftarrow nextCandidate(c, n, t, P)$
    if n is best solution so far **then**
        $s \leftarrow \{n\}$
    end if
    if n is as good as other best solutions **then**
        $s \leftarrow s \cup n$
    end if
end for
return s

that consists of the solutions that will be returned. In line 2, we initialize a candidate (either greedy or random initialization) and begin the search in line 3. In line 4, we assign a temperature to the variable t using the temperature schedule, as per the simulated annealing algorithm. In line 5, if the temperature has cooled to 0, then we can return the set of solutions (line 8). In line 8, we get a neighbor of the current solution. Thereafter, in line 9, *nextCandidate* is called. In the *nextCandidate* method, the neighbor n is evaluated against the current solution c to determine if n should be marked as the current candidate. *nextCandidate* is where we define our objective functions for the weighted method, the constrained method and the dynamic method. In lines 10-12, if n is the best solution, remove all previous solutions from s and add n, and in lines 13-15, if n is as good as the other solutions in s, then add n to s. Deciding if n is the best solution is dependent on comparing the objective output of n with the objective outputs of the solutions in s. The solution with a lower objective output is better than another solution with a higher objective output. For example, s is initially empty. Imagine that a candidate c_1 with objective output o_1 gets added to s. Subsequently, another candidate c_2 with output o_2 was found where $o_2 < o_1$. In this case, c_1 is removed from s and c_2 is added to s because it is the best solution so far. If $o_2 = o_1$, then c_2 is added to s without removing c_1 from s. The objective functions that are used to determine the solutions in s are:

- For the weighted objective function, the objective output for a candidate c is $\alpha * pvt(c, P) + \beta * ind(c, T') + \gamma * changes(c)$.
- For the constrained objective function, the objective output for a candidate c is $pvt(c, P)$.
- For the dynamic objective function, the objective output for a candidate c is

$pvt(c, P)$.

We now describe the *nextCandidate* algorithm used in *calcSolns*. In the *nextCandidate* method, the neighbor n is evaluated against the current solution c to determine if n should be marked as the current solution. This method influences how *calcSolns* traverses through the search space of solutions because *nextCandidate* updates the current solution and the current solution determines the neighborhood that is explored. In its essence, *nextCandidate* captures the objective functions that we described in Chapter 7. We provide three implementations of *nextCandidate*: one for the weighted method, one for the constrained method and one for the dynamic method. We will describe the algorithm for the heirarchical method later.

Algorithm 15 nextCandidate(c, n, t, P): Candidate c, Candidate n, double t, InfoContentTable P
Returns the solution used in the next step of the simulated annealing algorithm. This implementation is for the weighted method.

1: $e_1 \leftarrow \alpha * pvt(n, P) + \beta * ind(n, T') + \gamma * changes(n)$
2: $e_2 \leftarrow \alpha * pvt(c, P) + \beta * ind(c, T') + \gamma * changes(c)$
3: $\Delta E \leftarrow e_2 - e_1$
4: **if** $\Delta E > 0$ or $e^{\Delta E/t} > randDouble$ **then**
5: **return** n
6: **else**
7: **return** c
8: **end if**

Algorithm 15 shows the *nextCandidate* algorithm for the weighted method. *nextCandidate* accepts the current candidate c, the neighboring candidate n, the temperature t and the information content table P and returns the next candidate. In lines 1 and 2, the

weighted objective outputs are calculated for the neighboring candidate and the current candidate respectively. In line 4, if the neighbor produces a more minimal output, or if the neighbor was worse but passed the test $e^{\Delta E/t} > randDouble$, then return n as the next candidate for the next iteration of the simulated annealing algorithm. Otherwise, return c.

Algorithm 16 nextCandidate(c, n, t, P): Candidate c, Candidate n, double t, InfoContentTable P
Returns the solution used in the next step of the simulated annealing algorithm. This implementation is for the constrained method.

1: $\Delta E \leftarrow pvt(c, P) - pvt(n, P)$
2: **if** ($\Delta E > 0$ and $ind(n, T') < \varepsilon_i$ and $changes(n) < \varepsilon_c$) or ($e^{\Delta E/t} > randDouble$) **then**
3: **return** n
4: **else**
5: **return** c
6: **end if**

Algorithm 16 shows the *nextCandidate* algorithm for the constrained method. *nextCandidate* accepts the current candidate c, the neighboring candidate n, the temperature t and the information content table P and returns the next candidate. In line 1, the privacy loss is calculated for both the candidates. In line 2, if the neighbor is more minimal than the current candidate, and both the constraints are satisfied, or if the test $e^{\Delta E/t} > randDouble$ is passed, then return n as the next candidate for the next iteration of the simulated annealing algorithm. Otherwise, return c.

Algorithm 17 nextCandidate(c, n, t, P): Candidate c, Candidate n, double t, InfoContentTable P
Returns the solution used in the next step of the simulated annealing algorithm. This implementation is for the dynamic method.

$\Delta E \leftarrow pvt(n, P) - pvt(c, P)$
if ($\Delta E > 0$ and $ind(n, T') < \varepsilon_i$ and $changes(n) < \varepsilon_c$) or ($e^{\Delta E/t} > randDouble$) **then**
    if $ind(n, T') < \varepsilon_i$ **then**
        $\varepsilon_i \leftarrow ind(n, T')$
    end if
    if $changes(n) < \varepsilon_c$ **then**
        $\varepsilon_c \leftarrow changes(n)$
    end if
    return n
else
    return c
end if

Algorithm 17 shows the *nextCandidate* algorithm for the dynamic method. *nextCandidate* accepts the current candidate c, the neighboring candidate n, the temperature t and the information content table P and returns the next candidate. It is similar to algorithm 16 except for the additional lines 3-8 where the thresholds are updated if the neighbor has a better threshold than the current thresholds.

Simulated annealing for the hierarchical method

We have described *calcSolns* for the weighted, constrained and dynamic methods. In this section, we describe *calcSolns* for the hierarchical method. We have a separate *calcSolns* algorithm for the hierarchical method because it consists of two additional minimization steps compared to the other three methods.

Algorithm 18 is very similar to Algorithm 14. Lines 1-21 are similar to Algorithm 14. However, in lines 22-24, we have added a check where if the neighboring candidate's *pvt* output satisfies ε_p or if its *ind* output satisfies ε_i, then we add the neighbor to j, that is then added to s in line 26. Thereafter, between lines 27-32, we first find a subset of s that minimizes the *ind* objective the most (line 28), and if this subset contains more than one candidate (line 29), then find a subset of s that minimizes the *changes* objective the most. We implement the functionality in line 28 and line 29 by simply iterating through all the solutions in s and finding the set of solutions that minimize *ind* in line 28 and *changes* in line 29. Lines 27-32 represent the two minimization steps in the hierarchical method described in Chapter 7. Finally, we return s (line 33).

Algorithm 18 calcSolns (T', M', P, u', f): Embedded target table T', Embedded master table M', InfoContentTable P, Units u', $f : X \to Y$
Calculate all solutions to the multi-objective optimization problem for hierarchical method.

Define a set s, initialized to $\emptyset$.
Define a set j, initialized to $\emptyset$.
$c \leftarrow initialize(u')$
for $i \leftarrow 1$ to ∞ **do**
    $t \leftarrow schedule[i]$
    if $t = 0$ **then**
        return s
    end if
    $n \leftarrow getNeighbor(c, z, u')$
    $e_1 = pvt(n, P)$
    $e_2 = pvt(c, P)$
    $\Delta E \leftarrow e_1 - e_2$
    if $\Delta E > 0$ or $e^{\Delta E/t} > randDouble$ **then**
        $c \leftarrow n$
    end if

Algorithm 18 (continued) calcSolns (T', M', P, u', f): Embedded target table T', Embedded master table M', InfoContentTable P, Units u', $f : X \to Y$
Calculate all solutions to the multi-objective optimization problem for hierarchical method.

16: **if** n is best solution so far **then**
17: $s \leftarrow \{n\}$
18: **end if**
19: **if** n is as good as other best solutions **then**
20: $s \leftarrow s \cup n$
21: **end if**
22: **if** $pvt(n, P) < \varepsilon_p$ or $ind(n, T') < \varepsilon_i$ **then**
23: $j \leftarrow j \cup n$
24: **end if**
25: **end for**
26: $s \leftarrow s \cup j$
27: **if** $|s| > 1$ **then**
28: $s \leftarrow$ select a subset of s that minimizes the *ind* objective the most
29: **if** $|s| > 1$ **then**
30: $s \leftarrow$ select a subset of s that minimizes the *changes* objective the most
31: **end if**
32: **end if**
33: **return** s

8.2 Overall data repair algorithm

We have implemented the simulated annealing algorithm *calcSolns* for all four of our optimization functions. We now present our entire framework in the *dataRepair* algorithm.

Algorithm 19 accepts a target dataset T, a master dataset M and a set of FDs, Σ and modifies tuples in T so that T is closer to satisfying Σ. We first order the FDs so that FDs that produce a higher number of violations in T are ordered first (line 1). We process FDs with the most inconsistencies first due to their possible overlap with other FDs. We then iterate through all the ordered FDs (line 2). In line 3, T computes all violations with respect to the current FD f. T wishes to repair these violations. T then groups the violations into violation chunks and orders the chunks so that the biggest violation chunk is processed first (line 4). T does this because it wants to repair as many of its data values as possible. Next, T, M, and W perform the embedding and approximate record matching steps (line 5). Record matching is performed so that the clean tuples in M that correspond to the dirty tuples in T can be identified. Data values from the clean master tuples can be used to repair the erroneous values in the target violations. W then decomposes the matches into a set of units denoted by u, where a single unit is a data update that can be performed on some target tuple. Units are easy to visualize as data updates, and consequently, it is easier for us to develop algorithms that manipulate units (compared to manipulating the record matches). In line 7, M computes and returns an information content table P to W that can be used to calculate information disclosure if its data values are disclosed to T. Here, M would like to reveal as little information as possible. We then iterate through the ordered violation chunks to find repairs for the bigger chunks

Algorithm 19 dataRepair(T, M, Σ): Target table T, Master table M, a set of FDs Σ
Our entire framework.

1: $\Sigma_o \leftarrow orderFDs(T, \Sigma)$
2: **for** $f \in \Sigma_o$ **do**
3: $V \leftarrow calcViolations(T, f)$
4: $\{V_0, ..., V_m\} \leftarrow orderViolations(V, f)$
5: $(T', M', \mathcal{M}_F) \leftarrow embeddingAndMatching(T, M, f)$
6: $u \leftarrow getUnits(\mathcal{M}_F, f)$
7: $P \leftarrow calcInfoContentTable(M, f)$
8: **for** $V' \in \{V_0, ..., V_m\}$ **do**
9: Define a set u', initialized to $\emptyset$.
10: **for** $(r_m, r_t, x) \in u$ **do**
11: **if** $r_t \in V'$ **then**
12: $u' \leftarrow u' \cup (r_m, r_t, x)$
13: **end if**
14: **end for**
15: $s \leftarrow calcSolns(T', M', P, u', f)$
16: $c \leftarrow$ pick the candidate in s that has the smallest size
17: **for** $(r_m, r_t, x) \in c$ **do**
18: M sends $r_m[x]$ directly to T for $r_t[x]$.
19: **end for**

Algorithm 19 (continued) dataRepair(T, M, Σ): Target table T, Master table M, a set of FDs Σ
Our entire framework.

```
20:         for (r_m, r_t, x) ∈ c do
21:             P[r_m][x] ← 0
22:         end for
23:     end for
24: end for
```

first (line 8). A bigger violation chunk should be processed before a smaller violation chunk because T wants to greedily repair as many of its data values as possible. We define a set u' (line 9) that will eventually consist of a subset of u i.e., $u' \subseteq u$. In lines 10-14, u' is calculated so that it contains all target record ids present in the current violation chunk V' being repaired. We do this because we only want to work with the relevant units that can be used to repair V'. In other words, the set $u \setminus u'$ cannot be used to repair V' because $u \setminus u'$ does not contain units that involve the target tuples in V'. Our main goal now is to find a candidate $c \in \mathcal{P}(u')$ that minimizes three objectives: (i) the *pvt* objective (*pvt*), (ii) the *ind* objective (*ind*), and (iii) the *changes* objective. We select one of the four optimization functions to model our optimization problem. Since $\mathcal{P}(u')$ is a very large search space, we use the simulated annealing search algorithm to find solutions to our objective function (line 15). This set of pareto-optimal solutions is denoted by s. Only one solution is to be picked from s, so we pick the candidate $c \in s$ that has the smallest size because a smaller candidate leads to a smaller *changes* output compared to the other candidates in s. Intuitively, a smaller candidate should also lead to a smaller *pvt* output because fewer values are disclosed. M then sends the data values in c to T one by one (line 17-19)

and W adjusts the information content table to account for all the master data values that were revealed to T (lines 20-22).

As a final note, we do not guarantee that our data repair algorithm will cause T to satisfy Σ. This is because our algorithm tries to minimize multiple objectives simultaneously, and only the *ind* objective is minimized when $T \vDash \Sigma$.

8.3 Complexity analysis

The complexity of our *dataRepair* algorithm is dominated by the complexity of the *calcSolns* algorithm shown in Algorithm 14. This is because *calcSolns* is the simulated annealing search algorithm and is present in the innermost loop (line 15) of the *dataRepair* algorithm, and is called for every FD (line 2) and for every violation chunk (line 8). We analyze the *getNeighbor* and *nextCandidate* algorithm in order to determine the complexity of *calcSolns*. This is because *getNeighbor* and *nextCandidate* and present in the innermost loop of the *calcSolns* algorithm, and these two functions are called a large number of times (lines 3-7).

8.3.1 Complexity of the *getNeighbor* algorithm

getNeighbor, described in Algorithm 13, returns a random neighbor n from the neighborhood of the current candidate c. In practice, determining the entire neighborhood of c is expensive if it has many neighbors. Thus, our implementation of *getNeighbor* is optimized in Algorithm 20. The difference between *getNeighborOpt* and *getNeighbor* is that in *getNeighborOpt*, we do not need to determine the entire neighborhood of c.

getNeighborOpt accepts a candidate c, and the units u' and returns a neighbor of c. Between lines 1-3, we define a few auxillary data structures. In line 4, we randomly decide if we want a unit neighbor or choice neighbor of c (these were defined in Section 8.1.4). Between lines 4-17, a unit neighbor was selected while between lines 19-35, a choice neighbor was selected. In lines 5-7, we randomly decide to return a unit neighbor that is smaller than c by one unit. In lines 8-16, we randomly decide to return a unit neighbor that is larger than c by one unit. We describe lines 8-16 in more detail. In lines 8-10, we keep track of the master tuple associated with each target tuple in c. In line 11, we iterate over the units in u'. We add a unit to set N (line 13) only if either the target tuple in the unit is not in c, or if the target and master tuple in the unit is in c but the entire unit (including the attribute) is not in c (line 12). In line 15, we select any unit from N and assign it to n. This n is a unit neighbor that is larger than c by one unit. Next, we examine the case where a choice neighbor was selected (lines 19-35). In lines 19-21, we keep track of the master tuple associated with each target tuple in c. Between lines 22-26, we add master tuples and associated target tuples to set K (line 24) which are not present in c (line 23). We then copy c to create n (line 27). Next, we randomly select one element (r'_m, r'_t) in set K (line 28). Iterating on n (line 29), if the target tuple in a unit matches r'_t, the we change the master tuple of that unit to r'_m (line 31). Finally, we return n in line 35. The worst case complexity of *getNeighborOpt* is $O(U)$ where U is the size of u'. Note that by the very defintion of units, the size of u' indicates the amount of matches that were found for a particular FD i.e., the higher the size of u', the higher the number of matches that were found for an FD.

Algorithm 20 getNeighborOpt(c, u'): Candidate c, Units u'
Returns a random neighbor of c from its neighborhood.

Define a hashmap D with no keys.
Define a set N, initialized to $\emptyset$.
Define a set R, initialized to $\emptyset$.
if $randDouble > 0.5$ **then**
    if $randDouble > 0.5$ **then**
        $n \leftarrow$ create copy of c and remove a random unit in c.
    else
        for $(r_m, r_t, a) \in c$ **do**
            $D[r_t] \leftarrow r_m$
        end for
        for $(r_m, r_t, a) \in u'$ **do**
            if $(D[r_t] \neq \emptyset)$ or $(D[r_t] = r_m$ and $(r_m, r_t, a) \notin c)$ **then**
                $N \leftarrow N \cup (r_m, r_t, a)$
            end if
            $n \leftarrow$ create copy of c and add a random unit in N.
        end for
    end if
else

Algorithm 20 (continued) getNeighborOpt(c, u'): Candidate c, Units u'

```
19:     for (r_m, r_t, a) ∈ c do
20:         D[r_t] ← r_m
21:     end for
22:     for (r_m, r_t, a) ∈ u' do
23:         if D[r_t] ≠ ∅ and D[r_t] ≠ r_m then
24:             K ← K ∪ (r_m, r_t)
25:         end if
26:     end for
27:     n ← create copy of c.
28:     (r'_m, r'_t) ← randomly select an element from K.
29:     for (r_m, r_t, a) ∈ n do
30:         if r_t = r'_t then
31:             (r_m, r_t, a) ← (r'_m, r_t, a)
32:         end if
33:     end for
34: end if
35: return n
```

8.3.2 Complexity of the *nextCandidate* algorithm

We now examine the complexity of the *nextCandidate* algorithm. The complexity of all the three flavors of the *nextCandidate* algorithm (shown in Algorithms 15-17) is dominated by the complexity of the *ind* function shown in Function 9 because *ind* is calculated over the embedded target table T', which is very large. In particular, line 4 of Function 9 is an expensive procedure because calculating the entropy on T' involves calculating the frequency of all the embedded values in T'. We do not want to keep recalculating the frequency of the values in T' each time the *ind* function is called because this involves iterating over all the values in T', which is a large table. Instead, we perform the following optimization. When *ind* is first called by *calcSolns*, we calculate the frequency of the values in T'. This frequency information is stored. For any candidate c that is passed as input, the units in c are applied to T' and the stored frequency information only needs to be updated in order to calculate the entropies.

The initial calculation of frequency values runs at $O(N)$ where $N = |T'|$. If *ind* is called N times, then the amortized cost of calculating the initial frequency values is $O(1)$. The cost of applying a unit c to T' and updating the frequency values is $O(U)$ where $U = |u'|$ (note that $|c| \leq |u'|$). Thus, the amortized complexity of *ind* is $O(U)$, and the amortized complexity of *nextCandidate* is also $O(U)$.

8.3.3 Complexity of the *calcSolns* and *dataRepair* algorithms

The number of times that we iterate within the *calcSolns* algorithm (lines 3-7 in Algorithm 14) depends on the temperature schedule. This number is some constant d, and in our experiments we set this constant to 288. Given that the complexity

of $getNeighbor$ is $O(U)$ (where U refers to the size of input u') and the complexity of $nextCandidate$ is also $O(U)$, the complexity of $calcSolns$ is $O(dU)$. Thus, the complexity of $dataRepairs$ is $O(dUXY)$ where X is the number of FDs and Y is the number of violation chunks that were found for each FD. It is clear that the complexity of $dataRepair$ depends on:

1. The number of FDs that are accepted as input.
2. The number of violation chunks that are found for each FD.
3. The number of matches which are found between the target and master datasets for each violation chunk for an FD.
4. The number of iterations performed within the simulated annealing algorithm (this is a user-defined constant).

Our performance experiments in Section 9.5 corroborate our analysis. For example, in Section 9.5.1 and Section 9.5.2, we find that when the number of violation chunks are increased linearly (all other factors being the same), then the running time increases linearly too.

Chapter 9

Experiments

In this section, we describe the experiments that we carried on our implementation of the proposed framework. We divide the experiments into three categories: accuracy experiments, performance experiments and comparative experiments. For the accuracy experiments, we wanted to measure how well our framework is able to recommend the right repairs. This is measured via the quality metrics (precision, recall and F1). For the performance experiments, we wanted to measure the run time performance of our framework as the dataset error rate, the number of tuples, the number of FDs and the record matching similarity threshold parameters are varied. Finally, for the comparative experiments, we compared SparseMap embedding used in our framework with Bourgain embedding used by Barone et al. [10] and found that SparseMap is 30% faster but 7% worse for precision.

9.1 Datasets

We utilized two datasets for our experiments. The IMDB dataset [47] has 14 attributes and 1.2 million tuples. The 14 attributes are : "actor_id", "act_first_name", "act_last_name", "act_gender", "act_film_count", "movie_id", "act_role", "movie_name", "movie_year", "movie_rank", "director_id", "director_first_name", "director_last_name", "movie_genre". The IMDB dataset contains details about the actors and directors involved in a movie. We defined 2 FDs over the IMDB dataset.

- F_{I1} : ["act_first_name", "act_last_name", "movie_year"] $\rightarrow$ "movie_genre"
- F_{I2} : ["act_first_name", "movie_name"] $\rightarrow$ "act_role"

The books dataset [48] has 12 attributes and 3 million tuples. The 12 attributes are : "user_id", "user_age", "book_rating", "isbn", "book_title", "book_author", "publication_yr", "publisher", "img_url", "city", "state", "country". The books dataset contains information about users who have rated a particular book. We defined 6 FDs over the books dataset for our largest experiment, although most of our experiments use only the first two FDs.

- F_{B1} : ["user_id", "user_age"] $\rightarrow$ "city"
- F_{B2} : ["book_title", "publisher"] $\rightarrow$ "book_author"
- F_{B3} : ["city", "state"] $\rightarrow$ "country"
- F_{B4} : ["book_author", "isbn"] $\rightarrow$ "publisher"
- F_{B5} : ["book_title", "publication_yr", "user_id"] $\rightarrow$ "book_rating"
- F_{B6} : ["publisher"] $\rightarrow$ "book_title"

For our experiments, when we describe a dataset, we specify the number of tuples within parenthesis. For example, IMDB(500k) refers to the IMDB dataset with 500,000 tuples while IMDB(1.2m) refers to the IMDB dataset with 1,200,000 tuples.

We now describe how the target and master datasets are prepared for the experiments. We generate the master dataset M from the raw dataset by detecting and removing all violations with respect to every FD. Next, we generate the target dataset T. For some FD $F : X \rightarrow Y$, violations are generated by selecting $r_m \in M$, creating a copy $r_t \leftarrow r_m$, and updating $r_t[Y]$ to be different from $r_m[Y]$. $r_t[X]$ might also be updated to be different from $r_m[X]$. All such updates are tracked and referred to as *erroneous values*. r_m and r_t are then added to T. Error rate is defined as the number of violating tuples with respect to F. We continue adding tuples to T until the desired error rate is reached. For example, for a dataset with 1 million tuples, an 8% error rate amounts to 80,000 tuples which violate the FDs in Σ. We assume the errors are equally distributed among the FDs in Σ.

9.2 Normalization of the objectives

We normalize all three objectives in our experiments so that they have a value in the range [0,1], where 0 is the minimal output and 1 is the maximal output. We normalize the *pvt* output for a candidate $C \in \mathcal{P}(\mathcal{U})$ by dividing $pvt(C)$ by the total information content of all cells in the information content table. We normalize the *ind* output for a candidate $C \in \mathcal{P}(\mathcal{U})$ by dividing $ind(C, T)$ by $ind(\{\}, T)$, where $\{\}$ is a candidate with no units. We normalize the *changes* output for a candidate $C \in \mathcal{P}(\mathcal{U})$ by dividing $changes(C)$ by $|\mathcal{U}|$.

We now discuss the properties of measurement that are satisfied by each of the

three normalized measures. Each scale of measurement satifies one or more of the following properties [49].

1. Identity: each value on the measurement scale has a unique interpretation e.g., the gender "Male" is different from "Female".

2. Magnitude: values on the measurement scale have an ordered relationship to one another e.g., "First", "Second" or "Third" place in a swimming contest.

3. Equal intervals: a unit of measurement has an equivalent interpretation throughout the measurement scale e.g., the difference between two numbers 1 and 2 is equal to the difference between 15 and 16. Note that a unit of measurement is different from our definition of units described in Section 5.2.5.

4. Minimum value of zero: the measurement scale has a minimum point below which no values exist.

Normalized *pvt* satisfies the following properties of measurement: identity, magnitude and a minimum value of zero. Normalized *pvt* satisfies the identity property because every value in [0,1] is unique. Since a larger value in [0,1] corresponds to a larger amount of information disclosure, normalized *pvt* also satifies the magnitude property. The minimum value of zero is reached when *pvt* equals 0.

Normalized *ind* satisfies the following properties of measurement: identity, magnitude and a minimum value of zero. Normalized *ind* satisfies the identity property because every value in [0,1] is unique. Since a lower value in [0,1] corresponds to a larger amount of data cleaning utility, normalized *ind* also satifies the magnitude property. The minimum value of zero is reached when *ind* equals 0.

Normalized *changes* satisfies the following properties of measurement: identity, magnitude, equal intervals and a minimum value of zero. Normalized *changes* satisfies the identity property because every value in [0,1] is unique. Since a lower value in [0,1] corresponds to a lower amount of *changes* (by definition), normalized *changes* also satifies the magnitude property. *changes* merely measures the number of units that are present within a candidate, a very simple measurement of size which satisfies the equal interval property. The minimum value of zero is reached when *changes* equals 0.

9.3 Experimental settings

Our implementation was written using Java 1.7. All accuracy and comparison experiments were run on a server with 4 virtual CPUs (2.1 GHz each) and 32 GB of memory. The performance experiments were run on a server with 8 virtual CPUs (2.1 GHz each) and 32 GB of memory. In addition, we have made all our code publicly available [50].

Unless otherwise stated, the settings for every experiment are:

- The record matching threshold τ is set at 70%.
- The error rate in the target dataset is 8%.
- Greedy initialization is used for generating the initial candidate for the simulated annealing algorithm.
- F_{I1} and F_{I2} are defined on the IMDB dataset and F_{B1} and F_{B2} are defined on the books dataset.

- If a figure is shown for the weighted method, the *pvt* objective weight is set at 0.1, *ind* objective weight is set at 0.895 and *changes* objective weight is set at 0.005.
- If a figure is shown for the constrained, dynamic or hierarchical method, all the thresholds are set at 1.0.
- We perform 288 iterations in our simulated annealing algorithm.

9.4 Repair accuracy experiments

We want to measure how well our framework is able to recommend the right repairs. This is measured via the accuracy measures: precision, recall and F1.

Let r_c refer to the number of recommended data repairs that were correct. These recommendations are made by the recommendation engine. A correct repair refers to a correct data update to an erroneous value. Let r_t refer to the total number of recommended data repairs. Let e_t refer to the total number of erroneous values that exist in the target dataset.

- $precision = r_c/r_t$
- $recall = r_c/e_t$
- $F1 = 2 * (precision * recall)/(precision + recall)$

We vary the following parameters against accuracy: the weights for the weighted method, the record matching similarity threshold, the thresholds in the constrained, dynamic and hierarchical methods, the error rate and the number of tuples.

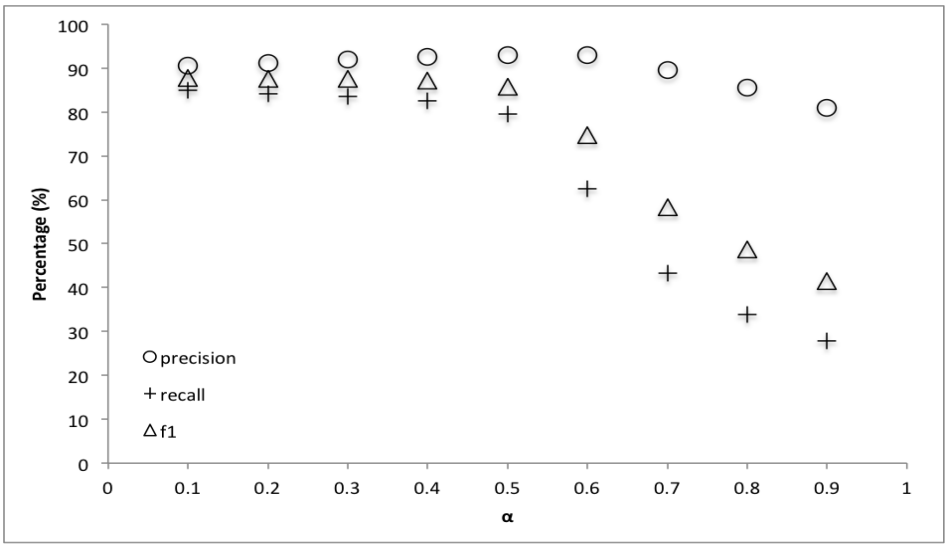

Figure 9.4: Relative importance of *pvt* objective vs accuracy

9.4.1 Varying the weights for the weighted method

We varied the weights for α (controls the *pvt* objective), β (controls the *ind* objective) and γ (controls the *changes* objective) and tested the effect on the accuracy of the solutions for the IMDB(500k) dataset.

In Figure 9.4, we varied the α weight between 0.1-0.9 while conversely varying the β weight between 0.9-0.1. The weight for γ was fixed at 0.05.

When the weight on the *pvt* objective is between 0.1-0.4, there is very little change in any of the three measures. Precision is approximately 90% and recall is approximately 85%. This is due to the relative difference in magnitudes between the *pvt* objective and the *ind* objective. At these lower weights, the influence exerted by the *pvt* objective on the weighted function is negligible compared to the influence exerted by the *ind* objective. This is a well understood trait of the weighted method and was described in Chapter 7 (that different objectives have different magnitudes and influence the weighted objective differently at different weights).

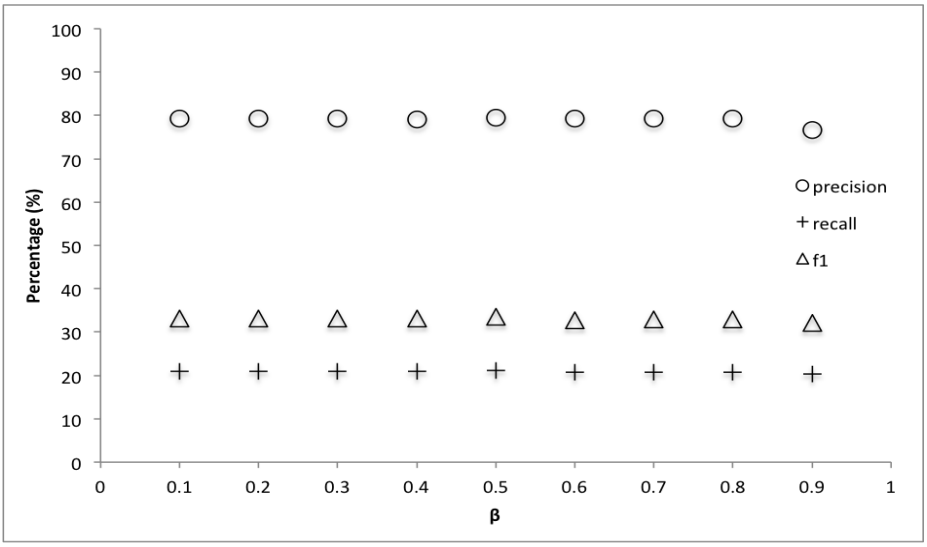

Figure 9.5: Relative importance of *ind* objective vs accuracy (image 1 of 2)

When the weight on the *pvt* objective is increased beyond 0.5 (i.e., when the weight on the *ind* objective is decreased below 0.5), the *pvt* objective exerts more influence on the weighted function. For example, at $\alpha = 0.8$, precision is 85% and recall is 34%. An increased weight on the *pvt* objective causes the precision and recall to decrease, because when *pvt* becomes more important, shorter solutions are found by the recommendation engine. Shorter solutions refer to candidates which are composed of fewer units. Shorter solutions minimize the *pvt* objective better than larger solutions because fewer data values are revealed. Since fewer data values are revealed, many of the errors are not captured by the algorithm, resulting in a sharp decline in recall. Precision also decreases, but at a slower rate, because even though most of the solutions are shorter, they are still mostly correct.

In Figures 9.5 and 9.6, we varied the β weight between 0.1-1.0 while conversely varying the γ weight between 0.9-0. The weight for α was fixed at 0.05.

When the weight on the *ind* objective is between 0-0.98, there is very little change

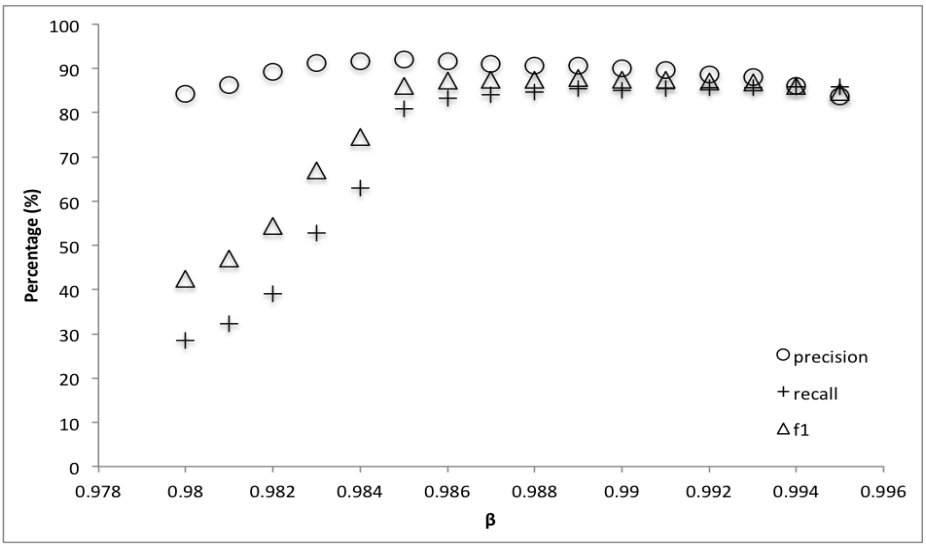

Figure 9.6: Relative importance of *ind* objective vs accuracy (image 2 of 2)

in any of the three measures. This is due to the relative difference in magnitudes between the *ind* objective and the *changes* objective. For example, imagine that we have initialized a candidate C. To minimize our objectives, we input C into our simulated annealing algorithm and find that $ind(C, T) = x_i$ while $changes(C) = x_c$. The simulated annealing algorithm then examines a neighbor C_n, and finds that $ind(C_n, T) = y_i$ while $changes(C_n) = y_c$. It is likely that $|y_i - x_i| << |y_c - x_c|$ because the standard deviation of the *ind* objective is much smaller compared to the standard deviation of the *changes* objective. The standard deviation of *ind* is very small because *ind* is defined over T, and T contains 500,000 tuples. This means that C and C_n will likely have a more similar output for *ind* compared to the output for *changes*. Thus, any weight lesser than 0.98 on the *ind* objective is not significant enough to increase the influence of the *ind* objective on the weighted objective.

When the weight on the *ind* objective is increased beyond 0.98 (i.e., when the weight on the *changes* objective is decreased below 0.02), the *ind* objective exerts

more influence on the weighted function. This causes the precision and recall to increase (up to 88% precision and 85% recall), because the *ind* objective becomes more important compared to the *changes* objective. When the *changes* objective becomes less important compared to the *ind* objective, longer solutions are found by the simulated annealing algorithm because *changes* has less influence on limiting the size of the solutions. A longer solution C has more units, which means that there are more data repairs. Thus, many of the injected errors can be corrected, which results in an increase in precision and recall.

9.4.2 Record matching similarity threshold vs accuracy

We vary the record matching similarity threshold τ between 60 and 100% and observe its effect on accuracy for the four methods for the IMDB(500k) dataset. We use the normalized euclidean distance measure to compare the similarity between two embedded records, where 100% means that two embedded records are exactly the same while 0% means that two embedded records are completely different.

As the record matching similarity threshold is increased, the recall (in Figure 9.8) decreases while precision (in Figure 9.7) remains largely unchanged. At higher similarity thresholds, there are fewer record matches i.e., fewer master tuples are matched against the target's violating tuples. Since our algorithms cannot find data repairs for unmatched target violations, this decreases recall. On the other hand, precision remains unaffected because even at lower thresholds, there is a clear best match for the dirty tuples, so that when the threshold increases, the best matches for the dirty tuples remain the same and are selected by greedy initialization. For example, at a similarity threshold of 60%, let us assume that there is only one match

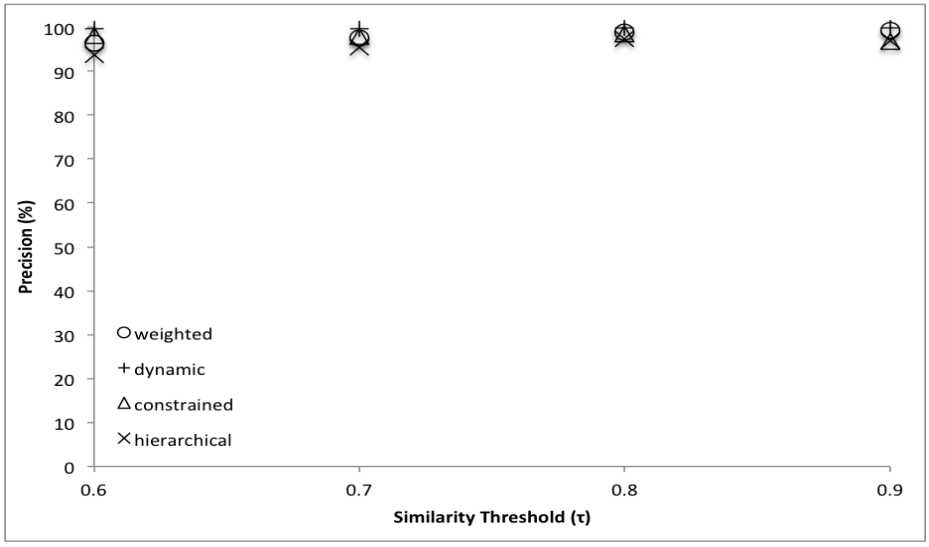

Figure 9.7: Record matching similarity threshold vs precision

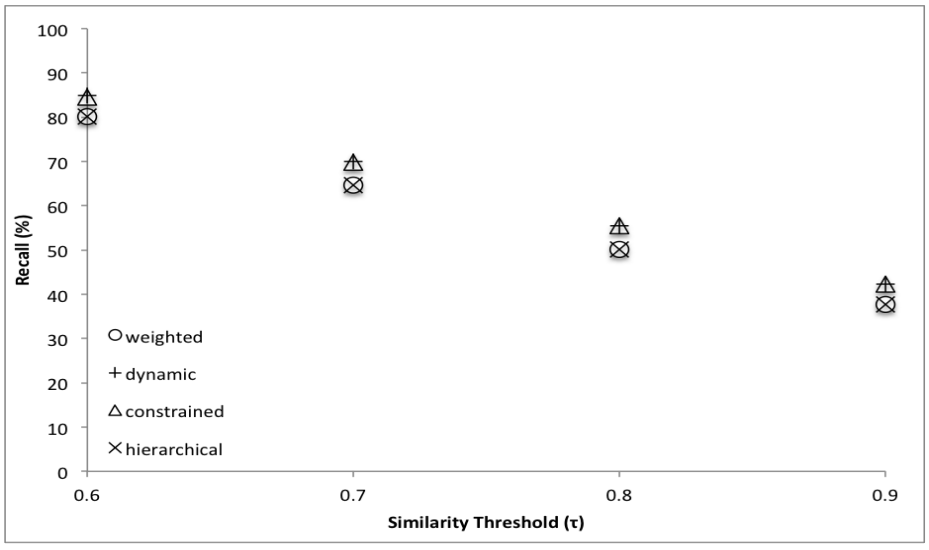

Figure 9.8: Record matching similarity threshold vs recall

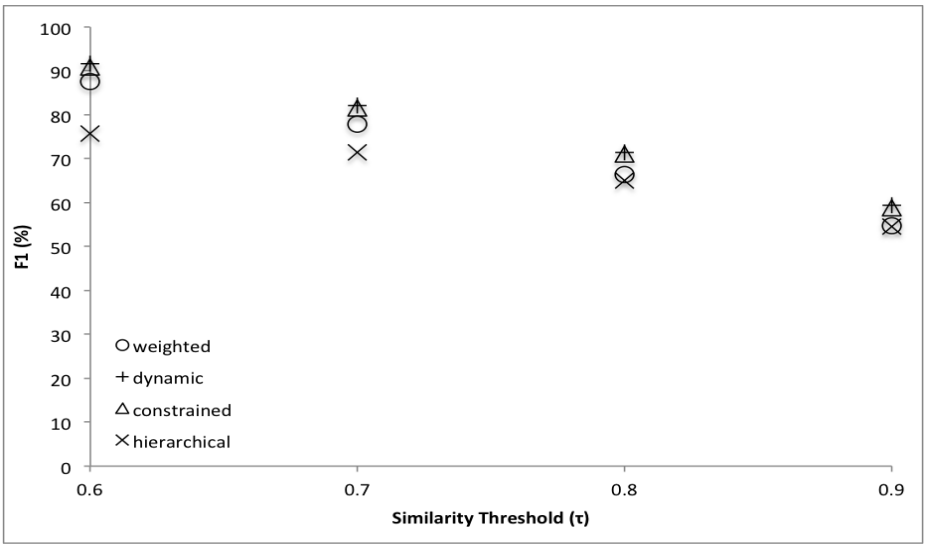

Figure 9.9: Record matching similarity threshold vs F1

for some violation chunk A and ten matches for some violation chunk B. At a 90% similarity threshold, chunk A could have zero matches while chunk B could have only one match. In this case, recall has decreased because the injected error in chunk A will not be fixed (at a 90% similarity threshold) as there are no matches. However precision remains roughly the same because the same best match for chunk B exists even as the threshold is increased. Hence, the number of recommended repairs that are correct remains roughly the same.

9.4.3 Threshold vs accuracy (constrained and hierarchical methods)

We vary the thresholds for the constrained and hierarchical methods and note the change in accuracy for the IMDB(500k) dataset. We do not examine the dynamic method because the thresholds vary dynamically as the candidates are evaluated.

In Figure 9.10, we varied the ε_i parameter for the constrained method. Precision

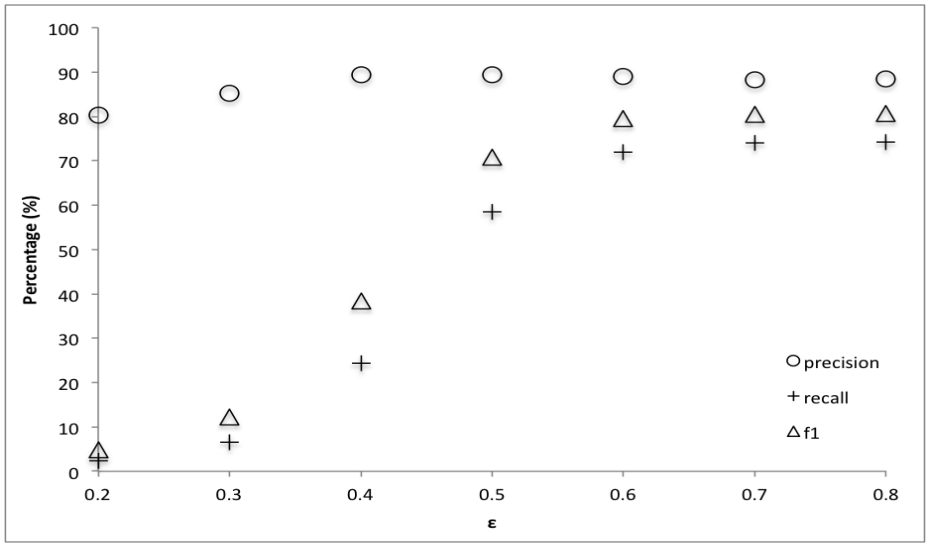

Figure 9.10: Threshold vs accuracy for the constrained method

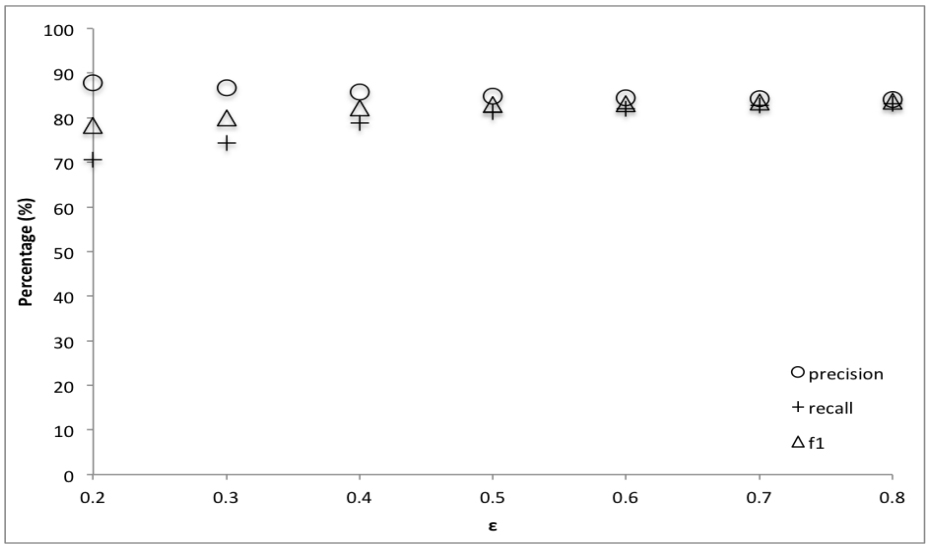

Figure 9.11: Threshold vs accuracy for the hierarchical method

and recall increase as ε_i is increased. When ε_i is between 0.2-0.5, precision is 85% on average and recall is 20% on average, but when ε_i is between 0.5-0.8, precision is 88% on average while recall is 74% on average. This is because when ε_i is low, most of the candidate solutions lie above ε_i and are unable to satisfy the constraints. Hence, few (or no) solutions are found for most of the simulated annealing experiments. Thus, precision and recall is low because few (or no) values are revealed. As ε_i is increased, candidates do not have to minimize *ind* very much in order to satisfy the constraints. This means that more candidates can satisfy the threshold and the simulated annealing algorithm is able to return solutions for most of the experiments, increasing the precision and recall.

In Figure 9.11, we varied the ε_k parameter for the hierarchical method. We see that the increase in recall is much more gradual for the hierarchical method. For example, when ε_k is between 0.2-0.8, precision stays roughly the same at 85% while recall increases gradually from 71% (ε_k=0.2) to 83% (ε_k=0.8). Note that ε_k is defined on the *pvt* objective for the hierarchical method whereas ε_i was defined on the *ind* objective in the constrained method. Since we did not vary ε_l, every simulated annealing experiment is able to return some solution that minimizes the *ind* objective in the hierarchical method. Whereas in the constrained method, most of the simulated annealing experiments did not return any solutions because most of the candidates could not satisfy ε_i when it was low. Hence, the slope for the constrained method is much steeper compared to the hierarchical method.

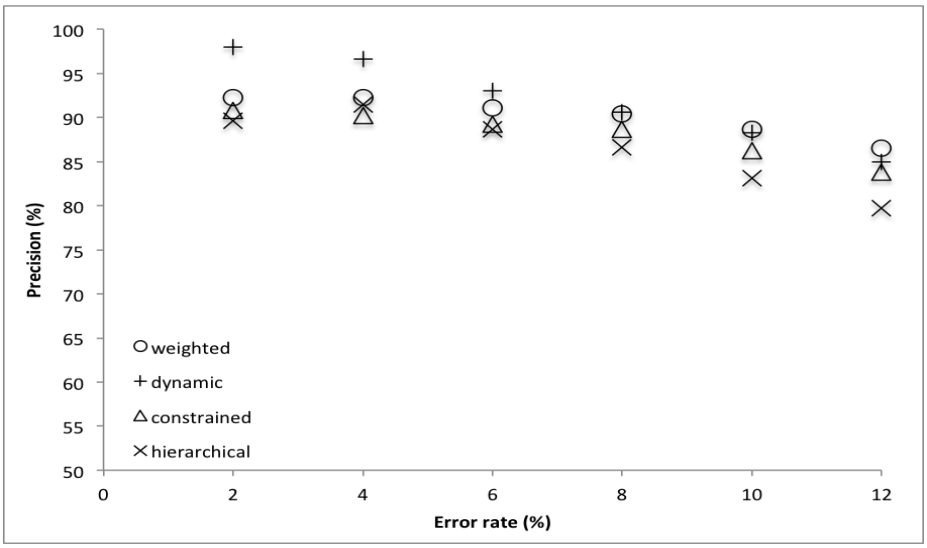

Figure 9.12: Dataset error rate vs precision (greedy initialization)

9.4.4 Error rate vs accuracy

We vary the error rate (between 2-12%) and measure its effect on accuracy for the IMDB(500k) dataset. Both greedy and random initializations were tested for this experiment. We present the results for greedy initialization first.

In Figures 9.12-9.14, the hierarchical method has the poorest precision, recall and F1 results compared to the other methods because the *pvt* objective is minimized first, before the *ind* objective. It is approximately 5% worse on average for precision and approximately 14% worse on average for recall compared to the other methods. The *pvt* objective has the greatest influence on the returned solutions, leading to smaller solutions with fewer data updates because fewer updates normally require fewer values to be disclosed. Smaller solutions lead to a lower recall score because fewer errors are corrected since there are fewer updates. Similarly, the weighted method is approximately 6% worse on average for recall compared to the constrained and dynamic methods because the weighted *pvt* objective influences the output to a

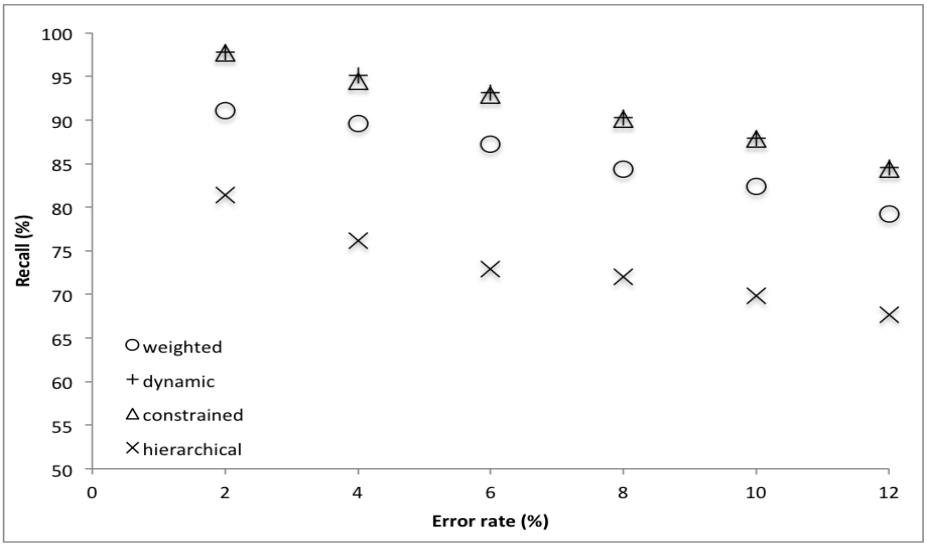

Figure 9.13: Dataset error rate vs recall (greedy initialization)

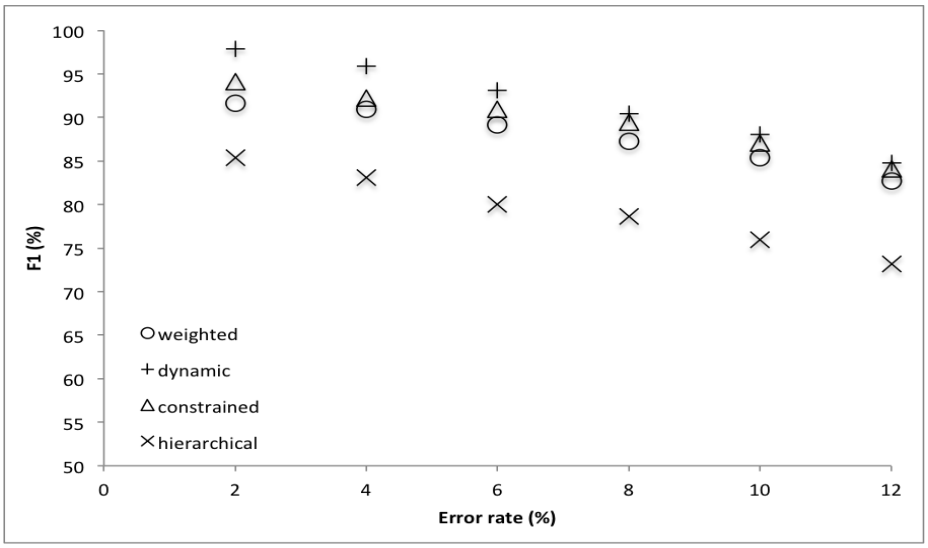

Figure 9.14: Dataset error rate vs F1 (greedy initialization)

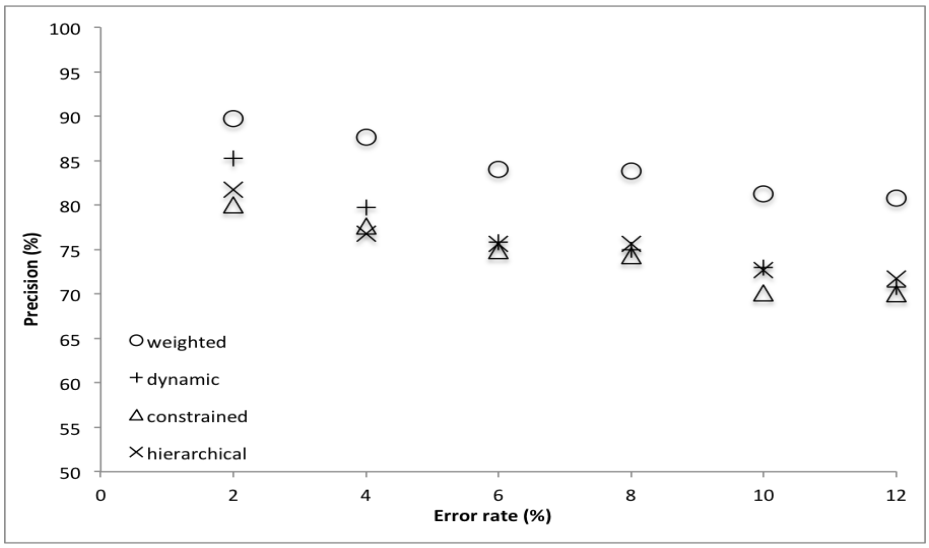

Figure 9.15: Dataset error rate vs precision (random initialization)

higher extent compared to the constrained and dynamic methods.

We now describe the error rate vs accuracy experiment using random initialization.

The precision, recall and F1 values shown in Figures 9.15-9.17 are lower for all the four methods compared to the greedy initialization shown in Figures 9.12-9.14. Precision is approximately 9% lower on average while recall is approximately 5% lower on average. This is expected because starting at a better initial solution forces the simulated annealing algorithm to explore a better search space from the very beginning. With random initialization, it is much more unlikely that the simulated annealing algorithm will be able to explore the better search space for the given number of iterations.

The constrained and dynamic methods have worse precision, recall and F1 with the random initialization when compared with greedy initialization. The constrained method has 13% worse precision and 8% worse recall with the random initialization while the dynamic method has 15% worse precision and 10% worse recall. This is

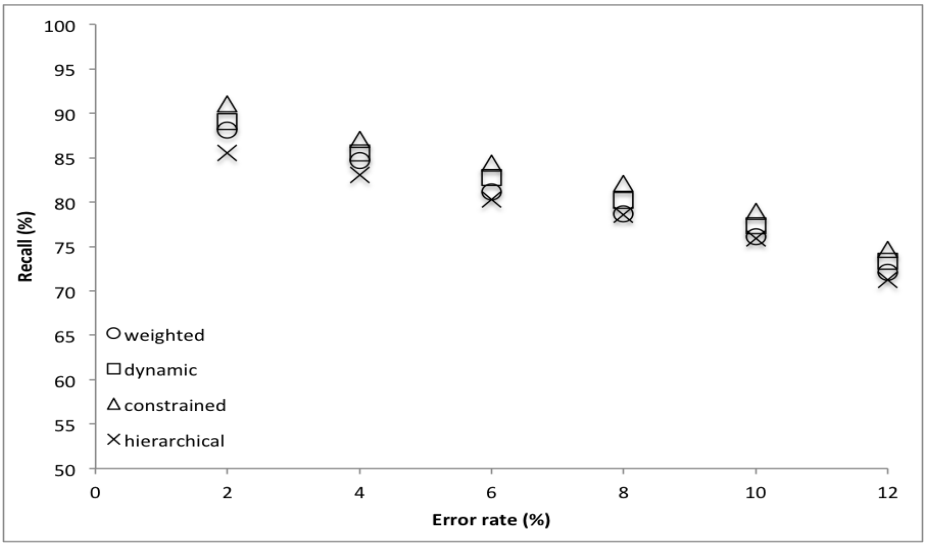

Figure 9.16: Dataset error rate vs recall (random initialization)

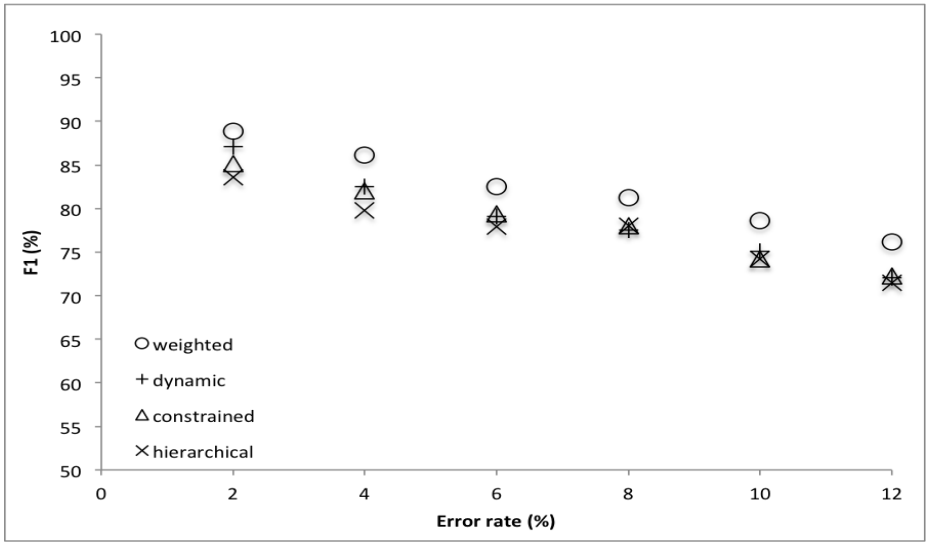

Figure 9.17: Dataset error rate vs F1 (random initialization)

because the thresholds on the constrained and dynamic methods are set based on the initial solution (we do this for the constrained method in our experiments because there is no user to fine tune the initial fixed thresholds). If the initial solution is poorer (i.e., random initialization), then the thresholds will be much larger, because we use the initial solution to determine an appropriate threshold. If the thresholds are much larger, this means that at every step of the the simulated annealing algorithm, many neighbors will be accepted as better solutions. This means that the simulated annealing algorithm has to spend more iterations exploring a worse search space. With greedy initialization, the good initial solution ensures that the thresholds are much tighter. Hence, better solutions are found with the constrained and dynamic methods using the greedy initialization compared with random initialization.

9.4.5 Number of tuples vs accuracy

We vary the number of tuples (between 200,000-1,200,000) and measure its effect on accuracy of the data repairs using the IMDB dataset. Both greedy and random initializations were tested for this experiment. We present the results for greedy initialization first.

The slope of the curves in Figures 9.18-9.20 are not as steep as the slope of the curves in Figures 9.12-9.14. This is a by-product of the way violations were generated on the IMDB dataset. When we increase the number of tuples (Figures 9.18-9.20), the violation patterns remain very similar. However, when we increase the error rate (Figures 9.12-9.14), the violation patterns change significantly. For example, for the FD F_{I2} : [“act_first_name”, “movie_name”] $\rightarrow$ “act_role”, if the tuple patterns “Leonardo, Titanic” $\rightarrow$ “Jack” and “Leonardo, Titanic” $\rightarrow$ “Rose” exist in the

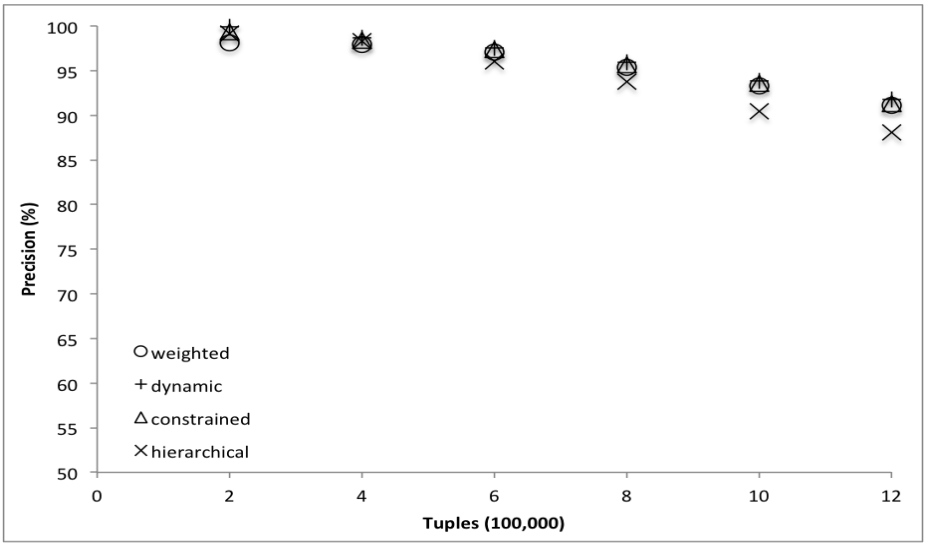

Figure 9.18: Number of tuples vs precision (greedy initialization)

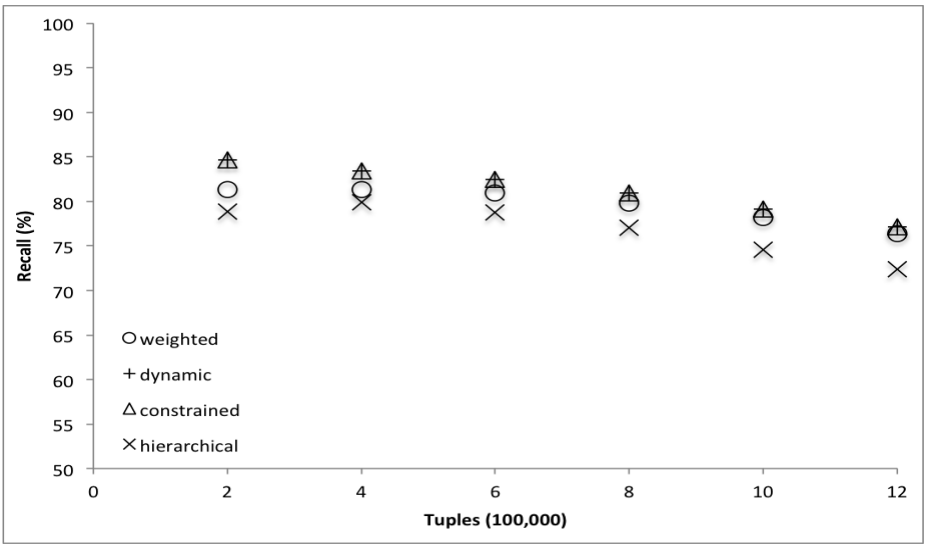

Figure 9.19: Number of tuples vs recall (greedy initialization)

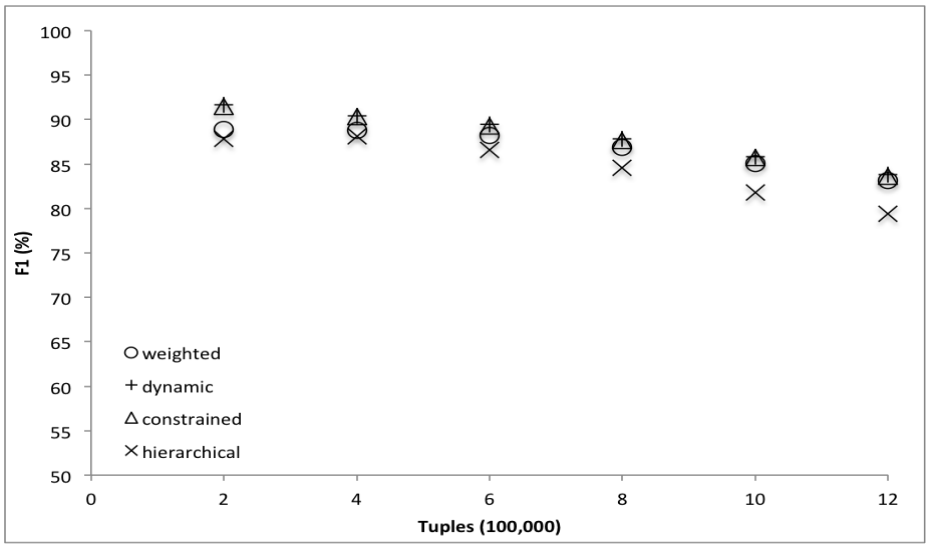

Figure 9.20: Number of tuples vs F1 (greedy initialization)

IMDB(200k) dataset, they would also exist in the IMDB(400k) dataset. The difference is that perhaps we might have a new pattern "Leonardo, Titanic" → "Cal" in IMDB(400k). Since the patterns have the same antecedent ("Leonardo, Titanic"), the total number of violation chunks stays the same even as the number of tuples is increased. This means that if a set of data updates is found for IMDB(200k), it will likely also be discovered for IMDB(400k) because the tuple patterns within a violation chunk are usually similar (this influences record matching and hence, the data repairs). This is not the case with increasing errors. With increasing errors, the IMDB(500k) dataset was used for all the error rates. Hence, when new violations were generated, they would form distinct violation chunks. e.g., with 2% error rate, "Leonardo, Titanic" → "Jack" and "Leonardo, Titanic" → "Rose" might exist, but with 4% error rate, it is more likely that "Keanu, Matrix" → "Neo" and "Keanu, Matrix" → "Morpheus" will be added as violations. In other words, with increasing

error rate, we are not increasing the existing violation chunks but rather, we are introducing new violation chunks with different tuple patterns. However with increasing tuples, we are increasing existing violation chunks with similar tuple patterns. Hence, the same solutions that we found for previous errors on a smaller dataset are found again in the larger dataset. This results in a gentler slope for the experiment with increasing tuples.

Similar to Figures 9.12-9.14, the hierarchical and weighted methods perform worse than the constrained and dynamic method in Figures 9.18-9.20 since they place a higher emphasis on the *pvt* objective. The hierarchical method is approximately 2% worse on average for precision and approximately 5% worse on average for recall compared to the other three methods. The weighted method is approximately the same for precision but 2% worse on average for recall compared to the constrained and dynamic methods.

We now describe the number of tuples vs accuracy experiment using random initialization.

The constrained and dynamic methods have worse precision, recall and F1 with the random initialization (Figures 9.21-9.23) when compared with greedy initialization (Figures 9.18-9.20). Both the constrained and dynamic methods have 22% worse precision and 10% worse recall with the random initialization. This is due to the same reason as explained in Section 9.4.4 for the random initialization experiment. To reiterate, random initialization is worse for the constrained and dynamic methods compared to greedy initialization because the thresholds for these methods are set based on the initial solution. If the initial solution is poorer (random initialization),

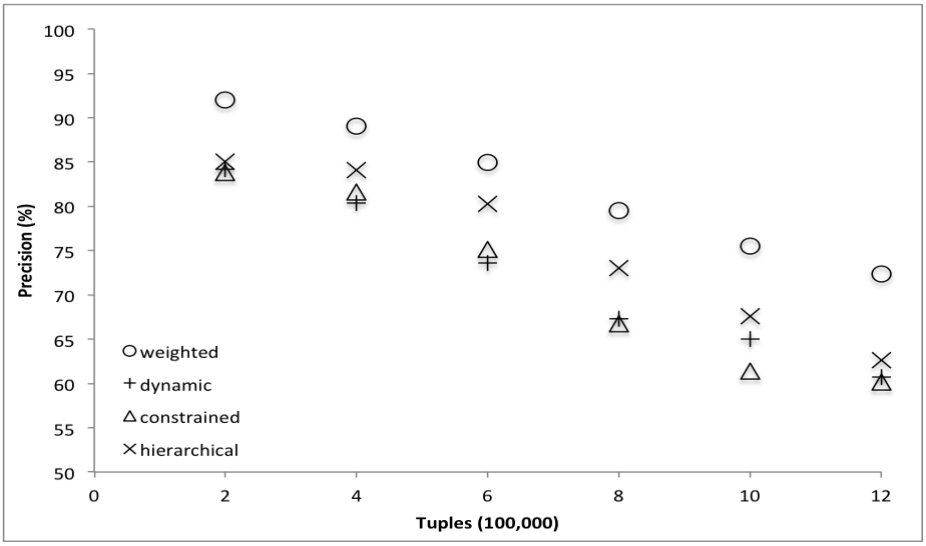

Figure 9.21: Number of tuples vs precision (random initialization)

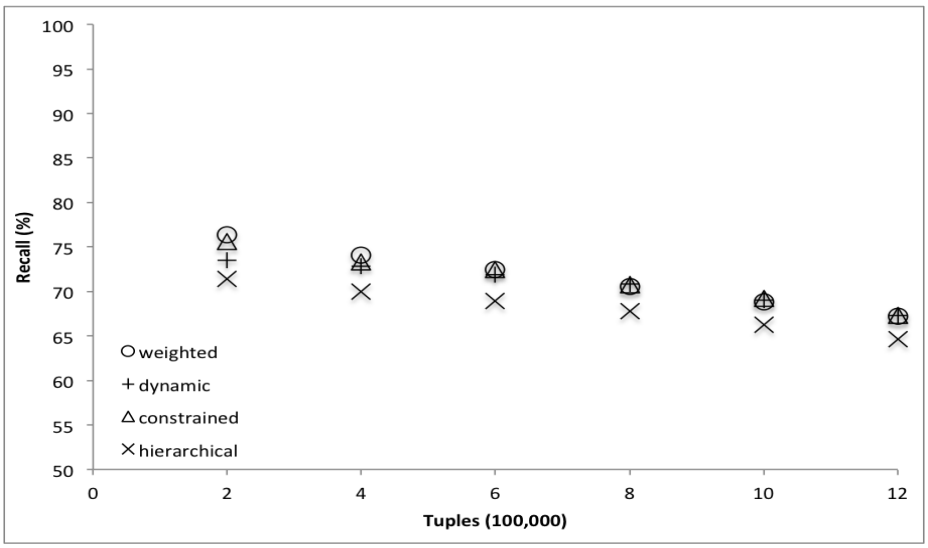

Figure 9.22: Number of tuples vs recall (random initialization)

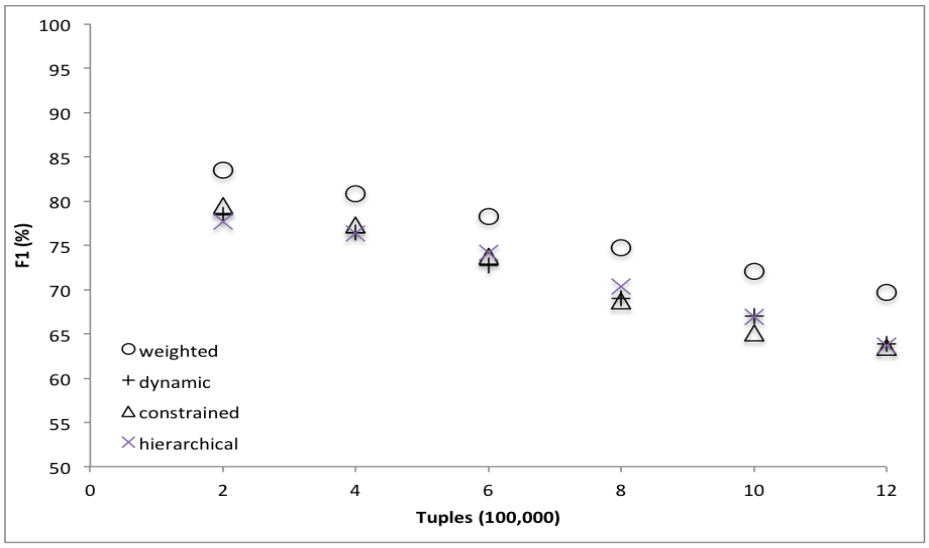

Figure 9.23: Number of tuples vs F1 (random initialization)

then the thresholds will be much larger, because we use the initial solution to determine an appropriate threshold. If the thresholds are much larger, this means that at every step of the the simulated annealing algorithm, many neighbors will be accepted as better solutions. This means that the simulated annealing algorithm has to spend more iterations exploring a worse search space. With greedy initialization, the good initial solution ensures that the thresholds are much tighter. Hence, better solutions are found with the constrained and dynamic methods using the greedy initialization compared with random initialization.

9.4.6 Threshold vs information disclosure (constrained and hierarchical methods)

We vary ε_i (between 0.05-0.95) for the constrained method and ε_k (between 0.05-0.95) for the hierarchical method and measure the amount of information disclosed (pvt) with a 0.1% error rate on the IMDB(500k) dataset.

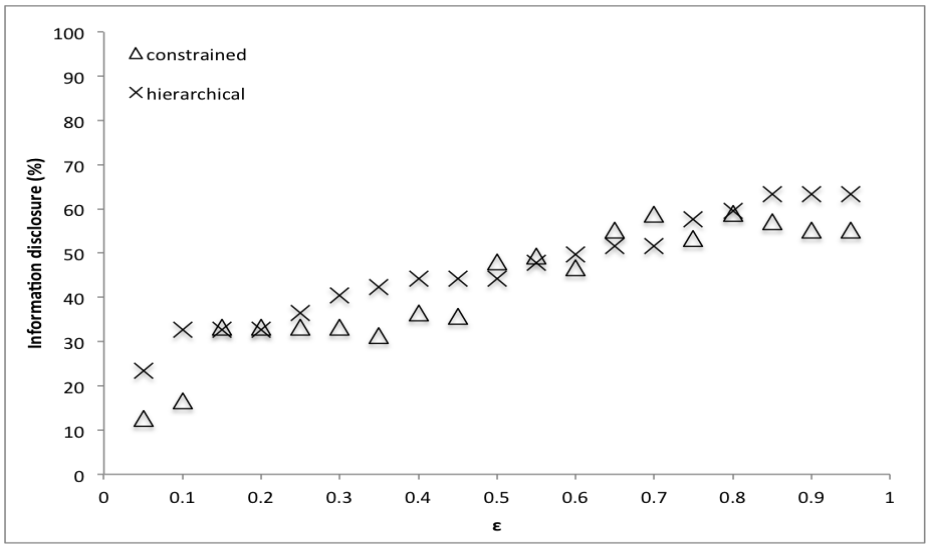

Figure 9.24: Threshold vs information disclosure for constrained and hierarchical methods

We normalize the information disclosure to get the y-axis in Figure 9.24. Normalized information disclosure is the total amount of information content that is disclosed to the target dataset divided by the total information content of all cells in the information content table.

In Figure 9.24, as ε_i is increased for the constrained method, the amount of information disclosed increases. When ε_i is low, most of the candidate solutions lie above ε_i and are unable to satisfy the constraints. Hence, few optimal solutions are found for most of the simulated annealing experiments. When few optimal solutions are found, information disclosure is low because fewer values are revealed. As ε_i is increased, more candidate solutions are able to satisfy ε_i. Hence, the simulated annealing algorithm is able to return optimal solutions for most of the experiments. If more solutions are found by the algorithm, information disclosure is higher because more values are revealed to the target dataset owner.

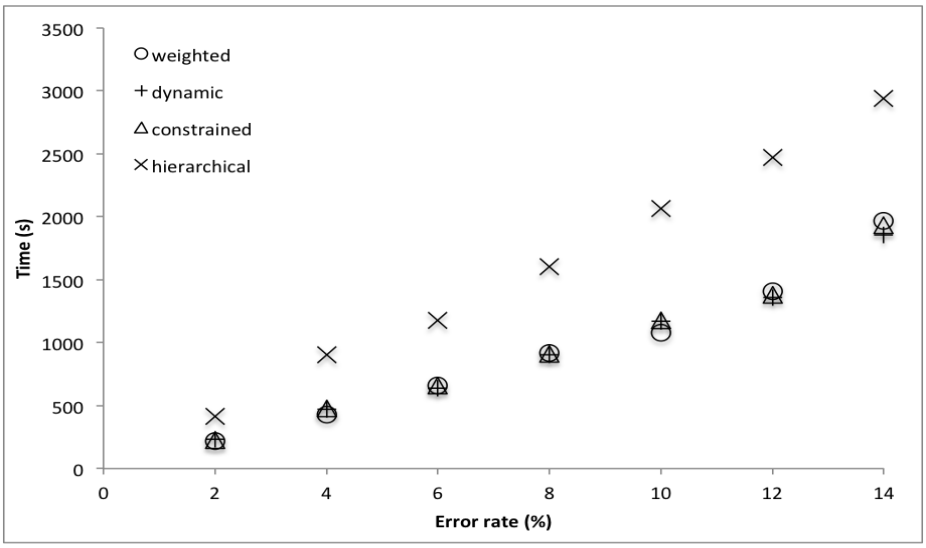

Figure 9.25: Dataset error rate vs running time

In Figure 9.24, as ε_k is increased for the hierarchical method, the amount of information disclosure increases. A lower ε_k means that most of the candidate solutions are not able to satisfy ε_k. Hence, fewer solutions are found, which means that fewer values are disclosed. Thus, information disclosure is lower when ε_k is lower.

9.5 Performance experiments

9.5.1 Error rate vs running time

We vary the error rate between 2-14% for books(500k) and record the running time.

As the error rate increases, the running time for the experiments increases linearly (Figure 9.25). As error rate is increased linearly, the number of violation chunks also increases linearly. For each additional violation chunk, an additional simulated annealing experiment is needed in order to find a solution. Hence, the number of simulated annealing experiments also increases linearly. This causes the running time to

also increase linearly, because the running time of our framework is heavily dependent on the number of simulated annealing experiments that need to be performed as seen in Algorithm 19 in Chapter 8.

The weighted, constrained and dynamic methods take approximately the same time to complete for all error rates. Moreover, this observation holds across all the various performance experiments. This is because the simulated annealing search algorithm is run for the same number of iterations for the three methods. However, the hierarchical method takes 70% longer (on average) compared to the other methods because two additional minimization steps are required with this method (described in Chapter 8). In the hierarchical method, the *pvt* objective is hierarchically minimized first using the simulated annealing algorithm, and a set of pareto-optimal solutions are found. If the size of the set is greater than one, then we need to perform an additional minimization step in order to minimize the *ind* objective and return a subset of the pareto-optimal set. If this minimization procedure results in yet another solution set with a cardinality greater than one, then the *changes* objective is hierarchically minimized. These two additional minimization steps are not required in the other methods. Hence, the hierarchical method takes a longer time to terminate.

9.5.2 Number of tuples vs running time

We vary the number of tuples for the books dataset to between 500,000-3,000,000 tuples and measure the running time.

As the number of tuples are increased, the running time also increases linearly (Figure 9.26). This is because when the number of tuples are increased linearly, the number of violations chunks also increase linearly. Every additional violation chunk

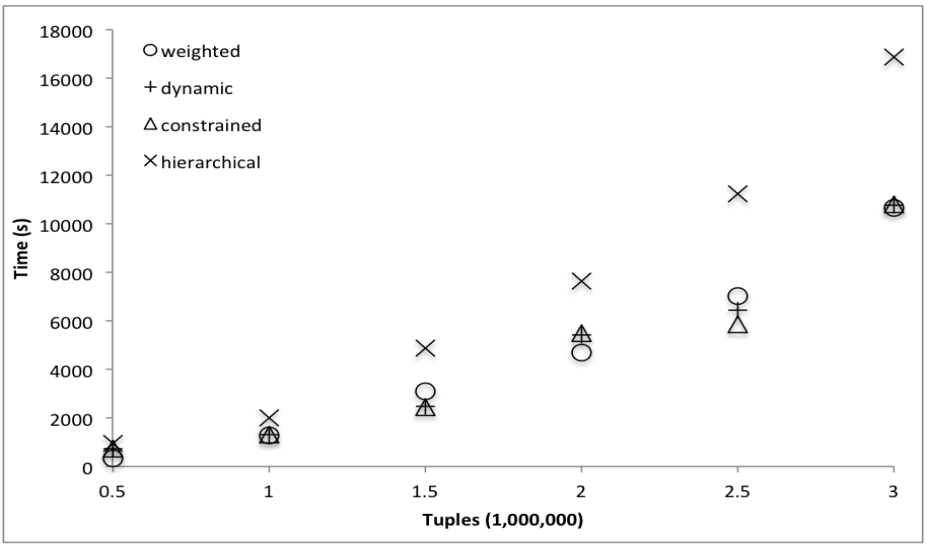

Figure 9.26: Number of tuples vs running time

requires an additional simulated annealing experiment in order to find a solution. This means that more simulated annealing experiments are required, and this results in an increase in the running time of the experiment.

There is a divergence in the running time between the hierarchical method and the other three methods as the number of tuples is increased. This is because even though the number of violation chunks increases with an increased number of tuples, the average size of each violation chunk also increases. When the average size of the violation chunks increases, the search space also increases, and hence, more pareto-optimal solutions are returned by the simulated annealing algorithm (on average). If the size of the pareto-optimal set increases, then the two additional minimization steps that are involved in the hierarchical method take a longer time to complete, because they have to iterate over a larger set.

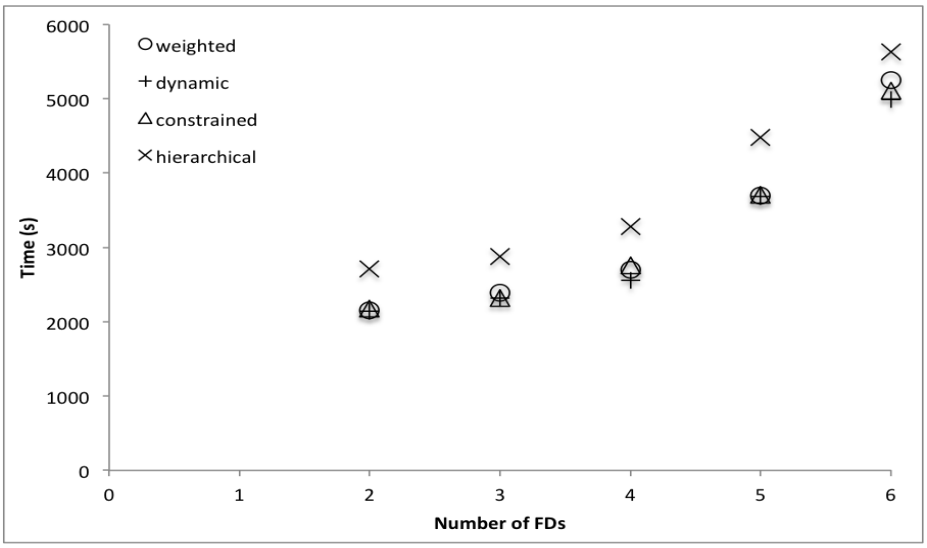

Figure 9.27: Number of FDs vs running time

9.5.3 Number of FDs vs running time

We vary the number of FDs in the books(500k) dataset to between 2-6 FDs and measure the running time.

As the number of FDs increases, the running time increases (Figure 9.27). This is because of how the errors were injected into the book(500k) dataset. Imagine that for an 8% error rate, we have x number of violation chunks for 2 FDs. This requires x number of simulated annealing experiments to be performed (as per the steps of Algorithm 19 in Chapter 8). When we increase the number of FDs, we increase the number of violation chunks, which subsequently increases the number of simulated annealing experiments which are performed. This causes the running time to increase as the number of FDs are increased.

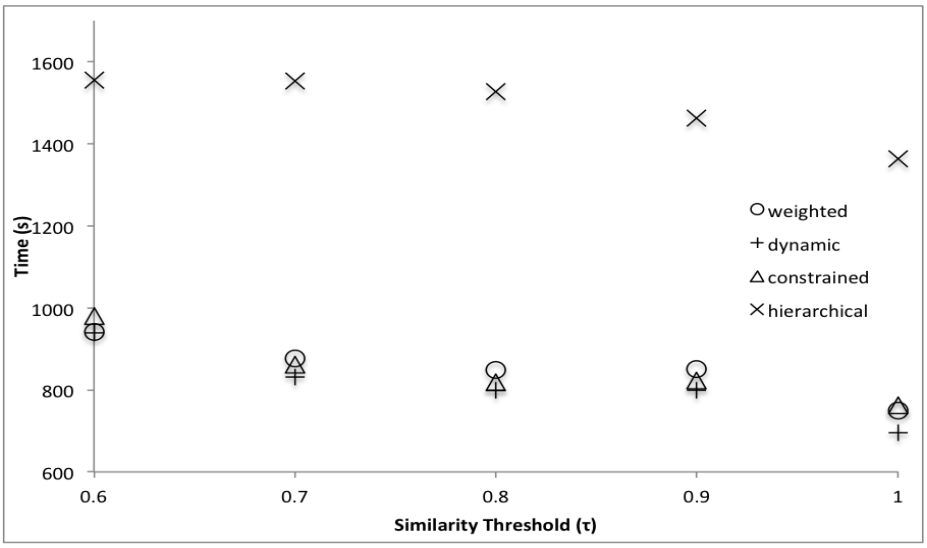

Figure 9.28: Record matching similarity threshold vs running time

9.5.4 Record matching similarity threshold vs running time

We vary the similarity threshold τ to between 60-100% and measure its effect on the running time using the books(500k) dataset (Figure 9.28). We use the normalized euclidean distance measure to compare the similarity between two embedded records, where 100% means that two embedded records are exactly the same while 0% means that two embedded records are completely different.

As the similarity threshold increases, fewer matches are found by the matching algorithm. Fewer matches means that the search space decreases, causing the running time to decrease.

The gradient of the curves in Figure 9.28 are much gentler compared to the other performance experiments in this section because even at the highest similarity threshold, most of the matches were found. Hence, the running time decreases only by 20% on average when we go from 60% to 100% similarity for all the four methods.

Table 9.4: Comparative experiment

	Bourgain embedding	**SparseMap embedding**
Precision of matching (%)	88	82
Time taken for matching (sec)	42	30
Precision for repair process (%)	85	79
Recall for repair process (%)	84	84

9.6 Comparison experiments

In this experiment, we compare the SparseMap embedding technique used in our framework with the Bourgain embedding technique [10] for the IMDB(100k) dataset with the weighted approach. In our framework, the embedding technique is used to embed target and master records that are sent to the third-party. The third-party performs record matching on embedded records and thereafter, calculates information disclosure and data cleaning utility on the embedded records in order to find a set of data repairs for the target dataset.

Our results are shown in Table 9.4. Let us denote record matches that are calculated over actual records by m_a. Let us denote the record matches that are calculated over embedded records by m_e. "Precision of matching" is defined as $\frac{|m_a \cap m_e|}{|m_e|}$. For example, imagine that we have two tables M and T, and their embedded counterparts M' and T'. Matches over actual records can be calculated by comparing all pairs of records across M and T. If the normalized string edit distance between $r_i \in M$ and $r_j \in T$ is below some threshold, then we add (r_i, r_j) to m_a. Matches over embedded records can be calculated by comparing all pairs of records across M' and T'. The SparseMap embedding and matching steps are described in Algorithm 5 in Chapter 5

while the Bourgain embedding and matching steps are described by Barone et al. [10]. All resulting matches (r_i, r_j) where $r_i \in M'$ and $r_j \in T'$ are added to m_e. We then intersect the sets m_a and m_e to determine the "correct matches" in m_e. Any element in the set m_a is defined as a "correct match" because this match was calculated using the normalized string edit distance measure over actual records. Finally we divide the number of "correct matches" in m_e by the total number of matches in m_e, and this is defined as the "precision of matching".

"Precision for repair process" and "Recall for repair process" are defined in Section 9.4.

From Table 9.4, SparseMap embedding has an 82% precision for matching while Bourgain embedding has 88% precision for matching. This is because SparseMap embedding involves two additional heuristics compared to Bourgain embedding. Bourgain embedding calculates the exact distance between the reference set and the data value in a record, whereas SparseMap uses the embedded reference set to approximate the distance between the reference set and the data value (this is described in Chapter 5). Thus, the quality of embedding is lower with SparseMap. Moreover, with SparseMap embedding, the dimensionality of the embedded records is reduced before record matching. This reduces the quality of the matches since fewer dimensions are being compared between the master and target dataset records during the matching phase. However, SparseMap embedding is approximately 30% faster compared to the Bourgain embedding method.

The precision results for the repair process is also higher for the Bourgain embedding method (at 85%) compared to SparseMap embedding (at 79%) as a result of the difference in the precision of matching. However, the recall results are not significantly

different between the Bourgain embedding and the SparseMap embedding. This was because we set the dimensionality reduction parameter to 0.9, and this is only a 10% difference in dimensionality between the two embedding techniques. Consequently, the quality of the record matching is similar across the two techniques, which leads to similar solutions and a similar recall for the solutions.

9.7 Summary of results

- For the weighted method, as the weight on the *pvt* objective is increased, the quality of data repair decreases. This is because disclosing a smaller amount of information helps in cleaning the dirty dataset to a smaller extent. Similarly, when the weight on the *ind* objective is increased, the quality of data repair increases.

- When we lower ε_i for the constrained method and ε_k for the hierarchical method, the quality of data repair decreases because most candidates lie above the thresholds and are not found by our search algorithm.

- For all the four methods, as the record matching similarity threshold is increased, the recall decreases while precision remains largely unchanged. This is because fewer matches are found when the similarity threshold is increased, which means that fewer data repairs are found.

- When measuring error rate against the accuracy of data repairs, the hierarchical method has the poorest results compared to the other methods. This is because it places more emphasis on the *pvt* objective compared to the other methods. More emphasis on *pvt* means that fewer values are disclosed, and hence, data

repairing is not as effective. For this same reason, the hierarchical method also has the poorest results when we measure the number of tuples against the accuracy of data repairs.

- Greedy initialization is better than random initialization in producing better quality data repairs because starting at a better initial solution forces the simulated annealing search algorithm to explore a better search space from the very beginning.

- As we increase the error rate, or the number of tuples, or the number of FDs, the running time of all four methods increases linearly because we have to repair a higher number of violations which requires a higher number of simulated annealing experiments to be performed.

- The weighted, constrained and dynamic methods generally have similar running times because the simulated annealing search algorithm is run for the same number of iterations for the three methods. In contrast, the hierarchical method takes 70% longer (on average) compared to the other methods because two additional minimization steps are required with this method (described in Chapter 8).

- SparseMap embedding is 30% faster than Bourgain embedding, but 7% worse in terms of precision of data repairs. This is because SparseMap embedding uses two heuristics to speed up the embedding and record matching process, at the cost of embedding quality.

Chapter 10

Conclusion

We have presented a complete data cleaning framework to clean a target dataset by using information from the master dataset. Our framework facilitates the cooperation between the two datasets so that the amount of information disclosed by the master dataset is minimized while the amount of data cleaning utility to the target dataset is maximized. We examine measures for quantifying information disclosure and data cleaning utility. Our information disclosure measure is an extension of the measure proposed by Arenas et al. [21] while the data cleaning utility measure was proposed by Dalkilic et al. [22]. Both measures are defined over embedded tuples, and not on actual tuples. Operating on embedded tuples protects the privacy of individual records in the datasets. We use the measures to develop a multi-objective optimization problem where the solution to the problem consists of a set of data repairs that will help to clean the target dataset. We utilize four optimization functions to model the optimization problem and incorporate the optimization functions into the simulated annealing search algorithm in order to find solutions to our optimization problem. The four optimization functions are popular methods of modeling multi-objective

optimization problems in optimization literature.

We perform extensive experiments on datasets containing up to 3 million records by varying parameters such as the error rate of the dataset, the size of the dataset, the number of constraints on the dataset, etc and measure the impact on accuracy and run-time performance for those parameters. Our results demonstrate that disclosing a larger amount of information within the clean dataset helps in cleaning the dirty dataset to a larger extent. We find that with 80% information disclosure (relative to the weighted optimization function), we are able to achieve a precision of 91% and a recall of 85%. In terms of running time, we find that increasing the error rate, or the number of tuples, or the number of FDs linearly causes the running time of our algorithms to increase linearly. This is because we have to repair a higher number of violations, so our algorithms take longer to find all repairs. We also compare our algorithms against each other to discover which ones produce better quality data repairs and which ones take longer to find repairs. We find that the hierarchical method has the poorest results compared to the other methods because it places more emphasis on the *pvt* objective. More emphasis on *pvt* means that fewer values are disclosed, and hence, data repairing is not as effective. Moreover, the hierarchical method takes 70% longer (on average) compared to the other methods because two additional minimization steps are required with this method. Finally, we incorporate ideas from Barone et al. [10] into our framework and show that our approach is 30% faster, but 7% worse for precision.

We conclude that our data cleaning framework can be applied to scenarios where master datasets are not publicly disclosed and different records within the master datasets have different privacy requirements.

10.1 Future research

In our experiments, our data repair algorithm tries to find solutions that minimize information disclosure by the master dataset and maximize data cleaning utility to the target dataset. However, the simulated annealing search algorithm does not guarantee that the solution is indeed the global minimal solution (unless we iterate the search algorithm enough times). Experimentally, we can verify the effectiveness of our search algorithm (in finding the global minima) by comparing its output with the output of a brute force search procedure on a small dataset.

Different privacy operations could be integrated into the framework. For example, we could generalize data values in the master dataset and measure the amount of information disclosed by revealing the generalized values. We could develop heuristics that use these generalized values to help clean the target dataset.

We can integrate different types of constraints into our framework apart from functional dependencies. This would be helpful in cleaning datasets which have different types of constraints defined on them. For example, we could perform record matching based on matching dependencies (a type of constraint [8]) and use these matches to clean the dirty data.

Various search algorithms (e.g., tabu search) could be explored and compared with the simulated annealing algorithm in terms of data repair quality and run-time performance.

Bibliography

[1] T. Redman, "The impact of poor data quality on the typical enterprise," *Communications of the ACM (CACM)*, vol. 41, no. 2, pp. 79–82, 1998.

[2] M. Dallachiesa, A. Ebaid, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang, "Nadeef: A commodity data cleaning system," *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp. 541–552, 2013.

[3] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, "The llunatic data-cleaning framework," *Proceedings of the VLDB Endowment (VLDB)*, vol. 6, no. 9, pp. 625–636, 2013.

[4] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD)*, pp. 143–154, 2005.

[5] M. Yakout, A. K. Elmagarmid, J. Neville, and M. Ouzzani, "Gdr: a system for guided data repair.," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pp. 1223–1226, 2010.

[6] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller, "Continuous data cleaning," *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pp. 244–255, 2014.

[7] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Towards certain fixes with editing rules and master data.," *The VLDB Journal — The International Journal on Very Large Data Bases*, vol. 21, no. 2, pp. 213–238, 2012.

[8] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Interaction between record matching and data repairing," *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pp. 469–480, 2011.

[9] E. Union, "Directive 95/46/ec on the protection of individuals with regard to the processing of personal data and on the free movement of such data.," *Official Journal of the European Communities*, 1995.

[10] D. Barone, A. Maurino, F. Stella, and C. Batini, "A privacy preserving framework for accuracy and completeness quality assessment.," *Emerging Paradigms in Informatics, Systems and Communication*, pp. 83–87, 2009.

[11] W. W. Eckerson, "Data quality and the bottom line: Achieving business success through a commitment to high quality data.," *The Data Warehousing Institute*, 2002.

[12] B. Otto and K. Weber, *From health checks to the seven sisters: The data quality journey at BT.* St. Gallen: University of St. Gallen, Institute of Information Management, 2009.

[13] Gartner, "Forecast : Data quality tools, worldwide, 2006-2011. technical report.," *Gartner*, 2007.

[14] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," *Proceedings of the 33rd international conference on Very large data bases (VLDB)*, pp. 315–326, 2007.

[15] M. Arenas, L. E. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases.," *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pp. 68–79, 1999.

[16] I. Fellegi and D. Holt, "A systematic approach to automatic edit and imputation.," *Journal of the American Statistical Association*, vol. 71, no. 353, pp. 17–35, 1976.

[17] C. Mayfield, J. Neville, and S. Prabhakar, "Eracer: a database approach for statistical inference and data cleaning.," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pp. 75–86, 2010.

[18] T. N. Herzog, F. J. Scheuren, and W. E. Winkler, *Data Quality and Record Linkage Techniques.* Springer-Verlag New York, 1 ed., 2007.

[19] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, pp. 1–16, 2007.

[20] D. Loshin, "Master data management.," *Knowledge Integrity, Inc.*, 2009.

[21] M. Arenas and L. Libkin, "An information-theoretic approach to normal forms for relational and xml data.," *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pp. 15–26, 2003.

[22] M. Dalkilic and E. Robertson, "Information dependencies," *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, no. 245 - 253, 2000.

[23] F. Chiang and R. J. Miller, "Discovering data quality rules," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1166–1177, 2008.

[24] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for capturing data inconsistencies.," *ACM Transactions on Database Systems*, vol. 33, no. 2, pp. 1–48, 2008.

[25] J. Wijsen, "Database repairing using updates.," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 3, pp. 722–768, 2005.

[26] F. Chiang and R. J. Miller, "A unified model for data and constraint repair," *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pp. 446–457, 2011.

[27] J. Proakis and M. Salehi, *Communication Systems Engineering*. Prentice Hall, 2nd ed., 2001.

[28] D. MacKay, *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 4th ed., 2005.

[29] C. Dwork, "Differential privacy," *Proceedings of the 33rd international conference on Automata, Languages and Programming (ICALP)*, pp. 1–12, 2006.

[30] M. Nergiz, M. Atzori, and C. Clifton, "Hiding the presence of individuals from shared database," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD)*, pp. 665–676, 2007.

[31] S. Chawla, C. Dwork, F. McSherry, A. Smith, and H. Wee, "Toward privacy in public databases," *Proceedings of the Second international conference on Theory of Cryptography*, vol. 3378, pp. 363–385, 2005.

[32] D. Kifer and J. Gerkhe, "Injecting utility into anonymized datasets," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD)*, pp. 217–228, 2006.

[33] V. Rastogi, D. Suciu, and S. Hung, "The boundary between privacy and utility in data publishing," *Proceedings of the 33rd international conference on Very large data bases (VLDB)*, pp. 531–542, 2007.

[34] J. Brickell and V. Shmatikov, "The cost of privacy: Destruction of data-mining utility in anonymized data publishing," *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pp. 70–78, 2008.

[35] M. Scannapieco, I. Figotin, E. Bertino, and A. Elmagarmid, "Privacy preserving schema and data matching," *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD)*, pp. 653–664, 2007.

[36] A. Machanvajjhala, J. Gerkhe, D. Kifer, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity.," *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, p. 24, 2006.

[37] D. Srivastava and S. Venkatasubramanian, "Information theory for data management," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*, pp. 1255–1256, 2010.

[38] S. Kolahi and L. Lakshmanan, "On approximating optimum repairs for functional dependency violations," *Proceedings of the 12th International Conference on Database Theory (ICDT)*, pp. 53–62, 2009.

[39] J. Branke, K. Deb, K. Miettinen, and R. Slowinski, *Multiobjective Optimization*. Springer, Lecture Notes in Computer Science, 2008.

[40] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[41] M. Laumanns, L. Thiele, and E. Zitzler, "An efficient, adaptive parameter variation scheme for metaheuristics based on the epsilon-constraint method.," *European Journal of Operational Research*, vol. 169, pp. 932–942, 2006.

[42] P. Fishburn, "Lexicographic orders, utilities and decision rules: A survey," *Management Science*, vol. 20, no. 11, pp. 1442–1471, 1974.

[43] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd ed., 2009.

[44] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[45] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images.," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, no. 6, pp. 721–741, 1984.

[46] L. Goldstein and M. Waterman, "Neighborhood size in the simulated annealing algorithm," *American Journal of Mathematical and Management Sciences*, vol. 8, no. 3-4, pp. 409–423, 1988.

[47] M. Stepp, J. Miller, and V. Kirst, "Web Programming Step by Step." `http://www.webstepbook.com/supplements-2ed.shtml`, 2015. [Online; accessed 24-June-2015].

[48] C. Ziegler, "Book-Crossing Dataset." `http://www2.informatik.uni-freiburg.de/~cziegler/BX/`, 2015. [Online; accessed 24-June-2015].

[49] S. Stevens, "On the theory of scales of measurement," *Science*, vol. 103, no. 2684, pp. 677–680, 1946.

[50] D. Gairola, "privacyCleaning repository." `http://github.com/dhruvgairola/privacyCleaning`, 2015. [Online; accessed 24-June-2015].