ANMAT: Automatic Knowledge Discovery and Error Detection through Pattern Functional Dependencies

Abdulhakim Qahtan  
QCRI, HBKU  
aqahatan@hbku.edu.qa

Nan Tang  
QCRI, HBKU  
nptang@hbku.edu.qa

Mourad Ouzzani  
QCRI, HBKU  
mouzzani@hbku.edu.qa

Yang Cao  
University of Edinburgh  
yang.ca@ed.ac.uk

Michael Stonebraker  
CSAIL, MIT  
stonebraker@csail.mit.edu

ABSTRACT
Knowledge discovery is critical to successful data analytics. We propose a new type of meta-knowledge, namely pattern functional dependencies (PFDs), that combine patterns (or regex-like rules) and integrity constraints (ICs) to model the dependencies (or meta-knowledge) between partial values (or patterns) across different attributes in a table. PFDs go beyond the classical functional dependencies and their extensions. For instance, in an employee table, ID “F–9–107”, “F” determines the financial department, and “9” determines one’s grade. Moreover, a key application of PFDs is to use them to identify erroneous data; tuples that violate some PFDs in this demonstration, attendees will experience the following features: PFD discovery – automatically discover PFDs from (dirty) data in different domains; and Error detection with PFDs – we will show errors that are detected by PFDs but cannot be captured by existing approaches.

1 INTRODUCTION
Patterns (or regex-like rules) are widely used to discover meta-knowledge in a given domain, e.g., a Year column should contain only four digits. In addition, integrity constraints (ICs) have been extensively studied to model data dependencies across columns, e.g., Postal Code uniquely determines City, which can then be used for error detection, query optimization, and data modeling, among others. Our key observation is that by relaxing the limitation of previous ICs, namely the

Table 1: D1: A Name Table

<table>
<thead>
<tr>
<th>name</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>M</td>
</tr>
<tr>
<td>John</td>
<td>M</td>
</tr>
<tr>
<td>Susan</td>
<td>F</td>
</tr>
<tr>
<td>Susan</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 2: D2: A Zip Table

<table>
<thead>
<tr>
<th>zip</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>90001</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>90002</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>90003</td>
<td>Los Angeles</td>
</tr>
<tr>
<td>90004</td>
<td>New York</td>
</tr>
</tbody>
</table>

need to operate on entire attribute values, we can specify a new type of data dependencies that can capture partial attribute values that follow some patterns.

Consider two datasets D1 and D2, for two tables Name and Zip, respectively. Table Name (Table 1) is defined over the schema (name, gender), and table Zip (Table 2) is defined over the schema (zip, city). Errorneous cells, r_{3} [gender] in D_1 and s_{4} [city] in D_2, are highlighted. Their correct values (or ground truth) are F and Los Angeles, which are also shown in the tables, below the erroneous values.

Our Methodology. Our proposed ICs are based on patterns of partial attribute values, as shown below:

\[
\lambda_1 : \text{Name} (\{ \text{name} = \text{John} \} \rightarrow \{ \text{gender} = \text{M} \})
\]

\[
\lambda_2 : \text{Name} (\{ \text{name} = \text{Susan} \} \rightarrow \{ \text{gender} = \text{F} \})
\]

\[
\lambda_3 : \text{Zip} (\{ \text{zip} = 900 \} \rightarrow \{ \text{city} = \text{Los Angeles} \})
\]

where \(\lambda_1/\lambda_2\) says that if someone’s first name is John/Susan, then the gender is M/F (\(A\) matches any alphabet and \(A^+\) matches any string, which will be defined in Section 2); and \(\lambda_3\) says that if a five-digit zip code starts by 900, then the city is Los Angeles (\(D\{2\}\) matches any two consecutive digits). Clearly, \(\lambda_2\) can detect error \(r_{4} [\text{gender}]\) in D_1 and \(\lambda_3\) can detect error \(s_{4} [\text{city}]\) in D_2.

Alternatively, consider two other constraints as follows:

\[
\lambda_4 : \text{Name} (\{ \text{name} = \text{LU} \{LL^+\} \rightarrow \{ \text{gender} \})
\]

\[
\lambda_5 : \text{Zip} (\{ \text{zip} = \{D\{3\}\} \rightarrow \{ \text{city} \})
\]

where \(\lambda_4\) says that one’s first name uniquely determines one’s gender for table Name (\(LU\) matches any upper case letter and \(LL^+\) matches any consecutive lower case letters); and \(\lambda_5\) states that the first 3 digits of a 5-digit zip code determines
the city for table Zip. These two PFDs ($\lambda_4$ and $\lambda_5$) are defined over two tuples: for example, two tuples match the LHS of $\lambda_4$, if they both satisfy the pattern $\text{[LU]\[LL+\]\[A*]}$, and their first names are the same, which is enforced by $\text{[LU]\[LL+\]}$.

$\lambda_4$ can detect the error $r_4[\text{gender}]$ by comparing tuples $r_3$ and $r_4$: $r_3$ and $r_4$ have the same first name Susan but different gender, which identifies a violation consisting of four cells ($r_2[\text{name}], r_4[\text{gender}], r_2[\text{name}], r_4[\text{gender}]$). Similarly, $\lambda_5$ can detect the error $s_4[\text{city}]$ by comparing $s_4$ with either $s_1$, $s_2$, or $s_3$.

The Limitations of the Prior Art. The fundamental limitation of previous ICs (e.g., FDs [1] and CFDs [2]) is that they enforce data dependencies using the entire attribute values. Consequently, they cannot specify the fine-grained semantics found in partial attribute values.

Our Proposed Demonstration. This demo implements ANMAT\footnote{From the Arabic word, patterns.}, a system to discover PFDs directly from dirty data, and to use them for error detection as a key application for such ICs. The audience will be able to see PFDs discovered from diverse domains. It will also see how new (i.e., cannot be detected by other ICs) data errors can be detected.

2 PATTERN FUNCTIONAL DEPENDENCIES

Before demonstrating the discovery of PFDs, we need to formally define them. We first discuss the (regex-like) patterns that we use for modeling the partial attribute values. While the class of general regular expressions can be used, it is actually too large for our purpose. In addition, it complicates the problems (i.e., high time complexity) of discovering and applying PFDs, e.g., checking the equivalence of two regular expressions is FSPACE-complete [6]. Fortunately, for the purpose of data cleaning, simple patterns are typically sufficient, as it has been shown in recent works [3, 5].

We use the generalization tree, which is a tree defined over an alphabet $\Sigma$, where each leaf node is a character in $\Sigma$ and each intermediate node is a generalization of its child nodes, depicted in Figure 1. It contains upper case letters [A-Z], lower case letters [a-z], digits [0-9], and other symbols. Here, $\epsilon$ represent the empty string.

Patterns. A pattern $P$ is a sequence of characters defined over the generalization tree. For strings $a$ and $\beta$, $\alpha$ having $N$ repetitions of $a$, $\alpha\beta$ is the logical and of $\alpha$ and $\beta$, $\alpha\ast$ means one-or-more repetitions, and the Kleene star operator $\alpha\ast$ denotes zero-or-more repetitions. We do not consider recursive patterns such as $(\alpha\ast)\ast$.

Employing a simple definition of patterns, in contrast to complicated regular expressions, has many benefits as they are: (1) easy to specify, (2) easy to discover, (3) easy to apply, (4) easy to reason about, and (5) most importantly, enough to detect most errors that more general regular expressions can capture in practice.

We say that a string $s$ matches (or satisfies) a pattern $P$, denoted by $s \mapsto P$, if $s$ is evaluated to be true by $P$.

Given two patterns $P$ and $P'$, we say that $P$ is contained by $P'$, denoted by $P \subseteq P'$, iff for any string $s$, $s \mapsto P$ implies $s \mapsto P'$. In other words, $P'$ is more general than $P$.

Example 1: [Patterns] Consider zip code 90001 and two patterns $P_1 = \{\text{D}\{5\}\}$ and $P_2 = \{\text{D}\}$. We have 90001 $\mapsto P_1$, 90001 $\not\mapsto P_2$, and $P_1 \subseteq P_2$.

Constrained Patterns. A constrained pattern $\overline{Q}$ is a concatenation of several patterns where at least one is constrained (or annotated) by the symbol $\psi\ast$. We call $Q$ the embedded pattern of the constrained pattern $\overline{Q}$. Given a string $s$, $s$ matches a constrained pattern $\overline{Q}$, denoted by $s \mapsto \overline{Q}$, iff $s \mapsto Q$.

Given two constrained patterns $\overline{Q}$ and $\overline{Q}'$, we say that $\overline{Q}$ is a restricted pattern of $\overline{Q}'$, denoted by $\overline{Q} \subseteq \overline{Q}'$, if for any two strings $s, s' \mapsto \overline{Q}$, implies $s \mapsto \overline{Q}'$.

Example 2: [Constrained Patterns.] One example constrained pattern is $\overline{Q}_1 = \{\text{LU}\}\{\text{LL*}\}\{\text{A*}\}$ from the constraint $\lambda_4$ presented in the introduction. It is used on the name attribute to enforce the matching over the first name. Another example is $\overline{Q}_2 = \{\text{LU}\}\{\text{LL*}\}\{\text{A*}\}\{\text{LU}\}\{\text{LL*}\}$, which can be used to enforce the matching over both the first name and the last name, but with an arbitrary number of middle names.

The embedded patterns of $\overline{Q}_1$, and $\overline{Q}_2$ are $\{\text{LU}\}\{\text{LL*}\}\{\text{A*}\}$ and $\{\text{LU}\}\{\text{LL*}\}\{\text{A*}\}\{\text{LU}\}\{\text{LL*}\}$, respectively. Obviously, $\overline{Q}_2 \subseteq \overline{Q}_1$, i.e., pattern $Q_2$ is contained by $Q_1$, and $\overline{Q}_1 \subseteq \overline{Q}_1$, i.e., $\overline{Q}_1$ is a restricted constrained pattern of $\overline{Q}_1$.

Consider two names in Table 1, $r_1[\text{name}] = \text{John Charles}$ and $r_2[\text{name}] = \text{John Bosco}$. We have $r_1[\text{name}] \mapsto \overline{Q}_1$, $r_2[\text{name}] \mapsto \overline{Q}_1$. Moreover, we have $r_1[\text{name}] \equiv_{\overline{Q}_1} r_2[\text{name}]$, because $r_1[\text{name}] = \{\text{John}\}$, $r_2[\text{name}] = \{\text{John}\}$, and $r_1[\text{name}] \cap r_2[\text{name}] = \{\text{John}\}$.

Pattern Functional Dependencies (PFDs). A PFD $\psi$ defined over schema $R$ is a pair $R(X \rightarrow Y, T_p)$, where:

1. $X$ and $Y$ are sets of attributes from $R$.
2. $X \rightarrow Y$ is a standard FD, called an embedded FD, and
3. $T_p$ is a tableau with all attributes in $X$ and $Y$, where for attribute $A$ in $X$ or $Y$ and each tuple $t_p \in T_p$, $t_p[A]$ is either a constrained pattern that matches values in $\text{dom}(A)$, or an unnamed variable ‘$\bot$’ that serves as a wildcard.

Please refer to $\lambda_1$–$\lambda_3$ in Section 1 for PFD examples.
3 DISCOVERY AND ERROR DETECTION

PFD Discovery. The PFD Discovery algorithm is shown in Figure 2. Given a table and a function to decide whether a set of value pairs forms a PFD as input, it outputs a set of PFDs. The algorithm first profiles the data to prune attributes for which PFDs cannot be found (line 1). For example, we drop all columns with pure numerical values. We then assume that all column pair combinations are potential dependencies for the PFDs. Then for each candidate dependency, the algorithm checks whether there are patterns that can be used to form a PFD (lines 3–14). The same process can be used to work on tokens (obtained using the function Tokenize) or n-grams (using the function NGrams) (lines 6–7). Then for each token or n-gram of t[A] (line 6), the algorithm inserts a key-value pair for the token or n-gram into an inverted list, where the key is the token or n-gram of t[A], and the value is a triple consisting of tuple id, position of the token or n-gram in t[A], and t[B] (line 8). Afterwards, it will scan all entries in the inverted list (line 10), and decide which entry can form a meaningful pattern tuple based on a predefined function (lines 11–12).

Error Detection using PFDs. Given a PFD ψ defined over schema R as (A → B, tp), we consider two cases for error detection: constant PFDs, i.e., the constrained parts of the tableau in the B attribute contains only constants, and variable PFDs, i.e., the value related to B attribute in the tableau contains a wildcard. For each constant PFD, we simply do the following scan the table and check, for each tuple t, if t[A] → tp[A] and t[B] ≠ tp[B], then there is a violation. In this case, if we assume that the LHS value is correct then the RHS could be repaired by changing it to tp[B]. For better performance, we create an index supporting regular expressions for each column present on the LHS of the PFDs. In this case, the search for violations will be limited to those tuples that match tp[A]. For variable PFDs, i.e., tp[B] = ∅, the brute force approach would be to enumerate all possible tuple pairs (t_i, t_j) and check for violations, i.e., t_i[A] = t_j[A] = tp[A] and t_i[B] ≠ t_j[B]. Again, we create an index supporting regular expressions for each column present on the LHS of the PFDs to limit the check to only tuples matching tp[A]. However, this is still quadratic. The quadratic time complexity can be avoided using blocking [4].

4 DEMONSTRATION OVERVIEW

Datasets. We will use real-world datasets, from data.gov and ChEMBL (https://www.ebi.ac.uk/chembl/downloads), as well as anonymized private datasets from the MIT data warehouse and local companies in Qatar. The audience is also encouraged to bring its own data and test it using ANMAT.

Algorithm Discover PFDs

Input: a relational table T, a function f and a minimum coverage threshold γ to make PFD decisions

Output: a set Ψ of PFDs

1. Φ := CandidateDependencies(T)
2. Ψ := ∅ /* the set of discovered PFDs */
3. for each FD ϕ : (A → B) ∈ Φ do
   4.   H := ∅ /* a hash-based inverted list */
   5.   for each tuple t ∈ T do
   6.     for each s ∈ Tokenize(t[A])|NGrams(t[A]) do
   7.       for each u ∈ Tokenize(t(B)|NGrams(t(B)) do
   8.         H.insert(s, (id(t), pos_u, u, pos_u))
   9.       T_p = ∅ for a new PFD ψ : (A → B, T_p)
   10.      for each entry h ∈ H do
   11.         if f(h) is true then
   12.           add a tuple t_p to T_p, w.r.t. entry h
   13.          if coverage(T_p) ≥ γ then
   14.             Ψ := Ψ ∪ {ψ}
   15.         return Ψ

Figure 2: Algorithm for Discovering PFDs

<table>
<thead>
<tr>
<th>Data</th>
<th>Dependency</th>
<th>Pattern Tableau</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_1</td>
<td>Phone Number</td>
<td>→ State</td>
<td></td>
</tr>
<tr>
<td>D_2</td>
<td>Full Name</td>
<td>→ Gender</td>
<td></td>
</tr>
<tr>
<td>D_3</td>
<td>ZIP → CITY</td>
<td>→ STATE</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Discovered PFDs and Detected Errors

Parameter Setting. ANMAT accepts two user input parameters, namely: (1) the minimum coverage and (2) the ratio of allowed violations. The minimum coverage represents the ratio of the records that participate in a PFD to the total number of records in the attribute. The participation is determined by checking all the records containing at least one of the patterns that appear in the tuples of the tableau. Since we assume the data is dirty, we tolerate a specific ratio of violations, which are reported as errors. The minimum coverage and the allowed violations give the user the ability to control the number of discovered dependencies. Both parameters represent a trade-off between discovering more dependencies and reducing the rate of false positives. For example, using smaller percentage for the coverage will allow to report more dependencies but it will report more dependencies which are false positives.
**System Interface.** We have implemented Anmat with two interfaces for different users: a GUI for lay users as shown in Figure 3, and a Jupyter Notebook for programmers. We will mainly demonstrate the GUI.

**Dataset Specification.** The user of Anmat will select the project and the dataset to work on from drop-down menus as shown at the top of Figure 3. New users can create their own projects and upload the datasets that need to be processed. After uploading the dataset and setting the minimum coverage and allowed violations, the system will automatically profile the dataset, extract the PFDs, and store the results in a MongoDB database.

**PFD Discovery.** An example of the extracted patterns is shown in Figure 3. The set of patterns are then used to extract the PFDs, and the PFDs that satisfy the minimum coverage will be reported. The user of Anmat will be able to display the tableau of each dependency and confirm whether that discovered dependency is valid for the dataset at hand (Figure 4). The displayed patterns have the form "pattern::position, frequency", where the position represents the token number at which the combination of tokens form the pattern start, assuming that the position of the first token is 0. The frequency represents the number of tuples that contain the pattern. When the patterns are extracted using n-grams, the position represents the position of the character at which the n-gram starts. Please note that n-grams are mainly used to extract patterns from attributes that contain single token which could be a code or ids.

**Error Detection using Discovered PFDs.** Based on the confirmed dependencies, Anmat will run them through the corresponding columns and return all violations, which are highly likely to be erroneous values. Since easy validation of the reported errors increases data cleaning tools’ usability, it is important for Anmat to provide techniques to validate the errors. The user of Anmat can display the violated rule(s) in the tableau and the full violating records to have more insights about the violations and confirm whether it is an error. Figure 5 shows examples of reported violations for the dependency Full Name → Gender. More examples for errors discovered from different datasets are shown in Table 3.

**REFERENCES**