Qualitative Cleaning: Data Profiling

Data Cleaning & Integration
CompSci 590.01 Spring 2017

Some contents were based on: Abedjan, Golab, and Naumann’s ICDE 2016 tutorial slides
Before you clean data...

You need to define what’s clean or “typical”

• Sometimes you can get these from domain experts or prior knowledge

• Other times you need to discover them from data itself ⇒ “data profiling” helps
  
  • We have seen how statistics help in quantitative data cleaning earlier in class
  
  • Now let’s turn to how logic helps for qualitative data cleaning
A real-life example (project idea)

• An actively updated database of Duke MBB stats
• Already very clean (manually curated), but still not without quality issues
Take-away points

• Sometimes you cannot tell whether a cell is dirty or not without looking at other cells
  • In the same row,
  • In the same column of different rows,
  • Or in different tables!

• “Constraints” help a lot
  • Specified by logic and checked by database queries
  • But lots of possibilities
  • Can we spot them automatically from data?
Keys

A set of attributes $K$ is a **key** for a relation $R$ if

- No two tuples in $R$ agree on the values of $K$
  - That is, $K$ uniquely identifies a tuple in $R$ (we call $K$ a “superkey” in this case)
- No proper subset of $K$ satisfies the above condition
  - That is, $K$ is minimal

Examples

- In *game*, $\{\text{gid}\}$ is a key, and so is $\{\text{date}\}$
- In *pgstats*, $\{\text{gid}, \text{pid}\}$ is a key
Functional dependencies

- A **functional dependency (FD)** has the form $X \rightarrow Y$, where $X$ and $Y$ are sets of attributes in a relation $R$
- $X \rightarrow Y$ means that whenever two tuples in $R$ agree on all the attributes in $X$, they must also agree on all attributes in $Y$

\[
\begin{array}{|c|c|c|}
\hline
X & Y & Z \\
\hline
a & b & c \\
\hline
a & b & ? \\
\hline
\end{array}
\]

Must be $b$ \quad Could be anything

- Note that $K$ is superkey $\iff K \rightarrow \text{all attributes of } R$ (and assuming no duplicate rows)
FD examples

• In game:
  • $gid \rightarrow$ all attributes
  • $date \rightarrow gid$
  • $season, oid \rightarrow score$?
  • $season, score, opp\_score, loc \rightarrow oid$?

• Question: are FDs discovered from a particular instance “reliable”?

Note:

• It suffices to list FDs of the form $X \rightarrow A$, where is $A$ is a singleton and $X \cap A = \emptyset$

• Also, if you list $X \rightarrow A$, there is no need to list $Y \rightarrow A$ where $Y \supseteq X$
Checking FDs by SQL

$X_1 X_2 \cdots X_k \rightarrow A$ where $X_1, \ldots, X_k, A$ are attributes

```sql
SELECT COUNT(DISTINCT($X_1, \ldots, X_k$)),
       COUNT(DISTINCT($X_1, \ldots, X_k, A$))
FROM R;
```

• FD holds if the two counts are the same
  • What if they are close? You have an “approximate” FD, still useful to data cleaning
• Many other formulations are possible
Efficient evaluation

• Sort or hash by the attributes
  • No need to carry along unnecessary attributes
• If data doesn’t fit in memory, use multiple passes
  • ... to merge (if sort)
  • ... to partition (if hash)
• Can you check multiple FDs with one sort?
• Can you leverage previous sorts?
Specialized algorithms

  - “Stripped partitions” of equivalence classes of tuples
  - Bottom-up lattice exploration

- **FastFDs**: Wyss et al. *DaWaK* 2001
  - “Agree/difference” sets from pairs of tuples
  - DFS search
TANE: stripped partitions

- $\pi_X$: partition tuples of $R$ by a set of attributes $X$
  - $\pi_{X \cup Y}$ is a refinement of both $\pi_X$ and $\pi_Y$
    - Assuming data fits in memory, refinement requires one pass
    - Start with singleton $X$’s; combine to get larger $X$’s as needed
  - $X \rightarrow A$ holds iff $|\pi_{X \cup A}| = |\pi_X|$

- Optimization: stripped partitioning $\hat{\pi}_X$: throw away partitions in $\pi_X$ with just a single tuple
  - Singleton partitions won’t lead to violations
  - $X \rightarrow A$ holds iff $|\hat{\pi}_{X \cup A}| - |\hat{\pi}_{X \cup A}| = |\hat{\pi}_X| - |\hat{\pi}_X|$, where $|\cdot|$ is the total number of tuples covered by all (non-stripped) partitions
TANE: search strategy

- Organize attribute subsets into a lattice
- Search bottom-up
  - To ensure minimality
- At each node $X$, check $X \rightarrow A$ for each edge to $X \cup A$ in the next level
  - Unless, e.g., we already know $Y \rightarrow A$ for some $Y \subset X$
    (other pruning conditions are also checked)
FastFDs: difference sets

“Data-driven”

• For each pair of tuples in $R$, compute the set of attributes for which they differ; let $D_R$ be the set of all such sets

<table>
<thead>
<tr>
<th></th>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$a_2$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_2$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$a_1$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_1$</td>
</tr>
</tbody>
</table>

$D_R = \{AD, BC, ABCD\}$

$D^A_R = \{D, BCD\}$

• Let $D^A_R = \{D \setminus A | D \in D_R \land A \subseteq D \}$

• $X \rightarrow A (A \cap X = \emptyset)$ holds iff $X$ intersects with every set in $D^A_R$
**FastFDs: efficient computation**

All-pairs comparison is expensive

- Instead of computing difference sets, compute “agree” sets and complement them
- Use stripped partitions to generate agree sets
  - All-pairs comparison only needed within each partition, not across partitions

Searching for FDs in $D_R$ is expensive

- Use DFS (depth-first traversal)
- Adjust attribute ordering according to how many difference sets each covers

---

**Definition 2.2.** A CFD $\Pi$ on $R$ is a pair $(\Pi, t)$, where:

- $X, Y \subseteq R$; $X \subseteq Y$ is an FD, called embedded FD in the context of CFD;
- $T_p$ is called a pattern tableau of $\Pi$, where for every attribute $A \in X \setminus Y$ and each pattern tuple $t_p \in T_p$, either $t_p[A]$ is a constant in the domain $\text{Dom}(A)$ of $A$, or $t_p[A]$ is a wild card '-'.
Inclusion dependencies

• Given lists of attributes $X$ and $Y$ from relations $R_1$ and $R_2$, an inclusion dependency $R_1[X] \subseteq R_2[Y]$ means that for every $R_1$ tuple, its combination of $X$ values must appear as $Y$ values in some $R_2$ tuple
  • Referential integrity (aka foreign key) constraint in databases is a special (but common) case

• Examples
  • $game.oid$ references $opponent.oid$
  • $pgstats.gid$ references $game.gid$
  • $pgstats.pid$ references $player.pid$
Checking inclusion by SQL

\[ R_1 [X_1 X_2 \cdots X_k] \subseteq R_2 [Y_1 Y_2 \cdots Y_k] \] where \( X_i \)'s and \( Y_i \)'s are attributes

\[
\text{SELECT} \ * \ \text{FROM} \ R_1 \\
\text{WHERE} \ (X_1, X_2, \ldots, X_k) \ \text{NOT IN} \\
(\text{SELECT} \ Y_1, Y_2, \ldots, Y_k \ \text{FROM} \ R_2) ;
\]

• Efficient evaluation?
  • Again, sort or hash
  • Sort (with duplicate elimination) + merge (with early termination) for unary inclusion dependencies (*SPIDER*): Bauckmann et al., *ICDE Workshops*, 2006
Specialized algorithms


- Divide-and-conquer (*BINDER*): Papenbrock et al. *PVLDB* 2015
**MIND: inverted lists**

- Build inverted lists
- Given attribute $A$, intersect all lists containing $A$
- For any surviving attribute $B$ we have $A \subseteq B$
- For multiple attributes, proceed bottom-up
  - Because $R_1[X_1, X_2] \subseteq R_2[Y_1, Y_2]$ implies $R_1[X_1] \subseteq R_2[Y_1]$ and $R_1[X_2] \subseteq R_2[Y_2]$
**BINDER: divide and conquer**

- Use hash partitioning to avoid comparison across partitions
- Validation skips attribute pairs for which inclusion already fails
More constraints

• **Conditional FDs (CFDs):** FDs + additional constraints on value combinations specified by patterns

• **Denial constraints (DCs):** universally quantified first-order logic

• Matching dependencies, metric/numeric FDs, editing/fixing/Sherlock rules, ...
  • More in Ilyas & Chu survey, *FnTdb* 2015
CFDs

\[ (R: X \rightarrow Y, T_p) \]

- \( X \rightarrow Y \) is a standard FD (“embedded” in the CFD)
- \( T_p \) is a “pattern tableau” with attributes of \( X \) and \( Y \), where each row is a pattern with constant values and wildcards (“-”)

For \( (R: X \rightarrow Y, T_p) \) to hold

- If two tuples match on \( X \), check each pattern—if they match the pattern’s LHS, they must also match its RHS, and with each other
- Suffices to consider only CFDs with a singleton RHS
**CFD example**

\( T_p \) for the CFD (\( \{\text{name, type, country}\} \rightarrow \{\text{price, tax}\}, T_p \))

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>country</th>
<th>price</th>
<th>tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>clothing</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>book</td>
<td>France</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>UK</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- **Embedded FD needs to hold**
  - for clothing, for UK, and for book in France
- **For book in France, tax must be 0**

### Table 2.2: CFD example

<table>
<thead>
<tr>
<th>TID</th>
<th>name</th>
<th>type</th>
<th>country</th>
<th>price</th>
<th>tax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Harry Potter</td>
<td>book</td>
<td>France</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Harry Potter</td>
<td>book</td>
<td>France</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Harry Potter</td>
<td>book</td>
<td>France</td>
<td>10</td>
<td>0.05</td>
</tr>
<tr>
<td>4</td>
<td>The Lord of the Rings</td>
<td>book</td>
<td>France</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>The Lord of the Rings</td>
<td>book</td>
<td>France</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Algorithms</td>
<td>book</td>
<td>USA</td>
<td>30</td>
<td>0.04</td>
</tr>
<tr>
<td>7</td>
<td>Algorithms</td>
<td>book</td>
<td>USA</td>
<td>40</td>
<td>0.04</td>
</tr>
<tr>
<td>8</td>
<td>Armani suit</td>
<td>clothing</td>
<td>UK</td>
<td>500</td>
<td>0.05</td>
</tr>
<tr>
<td>9</td>
<td>Armani suit</td>
<td>clothing</td>
<td>UK</td>
<td>500</td>
<td>0.05</td>
</tr>
<tr>
<td>10</td>
<td>Armani slacks</td>
<td>clothing</td>
<td>UK</td>
<td>250</td>
<td>0.05</td>
</tr>
<tr>
<td>11</td>
<td>Armani slacks</td>
<td>clothing</td>
<td>UK</td>
<td>250</td>
<td>0.05</td>
</tr>
<tr>
<td>12</td>
<td>Prada shoes</td>
<td>clothing</td>
<td>USA</td>
<td>200</td>
<td>0.05</td>
</tr>
<tr>
<td>13</td>
<td>Prada shoes</td>
<td>clothing</td>
<td>USA</td>
<td>200</td>
<td>0.05</td>
</tr>
<tr>
<td>14</td>
<td>Prada shoes</td>
<td>clothing</td>
<td>France</td>
<td>500</td>
<td>0.05</td>
</tr>
<tr>
<td>15</td>
<td>Spiderman</td>
<td>DVD</td>
<td>UK</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>Star Wars</td>
<td>DVD</td>
<td>UK</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>Star Wars</td>
<td>DVD</td>
<td>UK</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>Terminator</td>
<td>DVD</td>
<td>France</td>
<td>25</td>
<td>0.08</td>
</tr>
<tr>
<td>19</td>
<td>Terminator</td>
<td>DVD</td>
<td>France</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>Terminator</td>
<td>DVD</td>
<td>France</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Can you detect any violations?
CFD discovery

• Added challenge: too many possibilities for the pattern tableau

• Overall strategy
  • Build on FD discovery algorithms
  • A pattern in the tableau is similar to an association rule (Agrawal et al. SIGMOD 1993 and VLDB 1994) with 100% confidence
Optimal CFD tableau problem

• For a pattern:
  • LS (local support) = fraction of tuples matching its LHS
    • E.g., LS of book/France rule is 5/20
  • LC (local confidence) = max fraction of tuples matching its LHS that can be kept without violating the pattern
    • E.g., LC of book/France rule is 4/5

• Tableau as a whole:
  • GS (global support) = fraction of tuples matching any pattern’s LHS
  • GC (global confidence) = max fraction of matching tuples that can be kept without violating any pattern

• Problem: given the embedded FD, find the smallest tableau with minimum GS and GC
  • NP-complete and hard to approximate
An alternative problem

Given the embedded FD, find the smallest tableau with minimum GS and LC (for each pattern)

• A variant of the partial set cover problem

• Generate all possible patterns with high enough LC

• Greedily choose the pattern with highest marginal improvement to GS
DCs

• Each DC has form $\forall t_1, t_2, t_3, \ldots \in R: \neg (P_1 \land P_2 \land \cdots)$, where each predicate $P_i$ either compares two attribute values of the quantified tuples, or compares one attribute value against a constant
  • Violation = finding a combination of tuples for which all predicates are true

• FDs and CFDs are all special cases of DCs

• Examples
  • $\forall t \in pgstats: \neg (t.fg3 > t.fg3a)$
  • $\forall t_1, t_2 \in R: \neg (t_1.name = t_2.name \land t_1.type = t_2.type \land t_1.country = t_2.country = \text{“UK”} \land t_1.price \neq t_2.price)$
DC discovery

Conceptually extends FastFDs’s difference set idea

• Enumerate all possible predicates $\mathcal{P}$

• Build an evidence set, where each element is a subset of $\mathcal{P}$ that are satisfied by some tuple pair

• If some subset of $\mathcal{P}$ overlaps with every element of the evidence set, then we have a DC
Summary and thoughts

• With a huge space of constraints to explore, even a small database can give you headaches
• Can’t we use sampling or approximation algorithms to speed up search for constraints?
• Next week, we will start looking at how to “repair” data given the constraints