Ghent University - TELIN - DDCM
2021-2022

Lab session: Categorical Edit Rules

1 Introduction

During this lab session, we will investigate concepts related to the (in)consistency of categorical data. In order to do this effectively, we will use the framework of edit rules, which is introduced by Fellegi & Holt[1] and originally used to discover and enhance inconsistent responses in demographic surveys. In the following, we will not focus on datasets that persist responses of demographic surveys, but we will focus on datasets persisting data related to the design of clinical trials. More specifically, we will investigate whether the data related to the masking of clinical trials are consistent over two different data sources (EudraCT[2] and ClinicalTrials.gov[3]). In order to avoid solving the exercises manually, we will use our own rulebox tool, which is developed in Java, and the PostgreSQL database management system.

2 First steps

In order to successfully go through the workshop and complete the exercises, make sure that you have downloaded, installed and tested all necessary software (listed in the following task) in advance.

1. Download and install Java (version 8+)[4].

[1] Fellegi and Holt, A Systematic Approach to Automatic Edit and Imputation, 1976.
[2] `https://eudract.ema.europa.eu`
[3] `https://clinicaltrials.gov`

2. Download the `rulebox.zip` file, which you can find on Ufora (Content > Data Quality > Edit rules > `rulebox.zip`). Extract the content (`rulebox.jar`) of this .zip file.

3. Open a command line, navigate to the location of the `rulebox.jar` file and execute the following command: `java -jar rulebox.jar`. As output, you should see information related to the usage of the rulebox tool.

4. Download and install PostgreSQL (version 13+)[5]and pgAdmin 4[6]. Information on how to install PostgreSQL and pgAdmin is provided on Ufora (Content > Introduction > `Installing and testing PostgreSQL.pdf`).

If you have any issues with the previous task, please contact the teaching staff.

3 Data exploration

Once you have successfully downloaded, installed and tested all necessary software, you can start exploring the data that will be used in the remainder of this workshop. These data are persisted in a PostgreSQL database that you only can access within the UGent network (or by means of a VPN-connection to the UGent network) with the following connection parameters.

- Hostname/IP-address: ddcmstud.ugent.be
- Port: 8088
- Database: clinicaltrials
- Username: sql_exerciser
- Password: 7UCVuJeLCGcQbk2M[7]

As you learned during the lectures on relational databases, connecting to a database can be done by using a database client (e.g. pgAdmin 4 for PostgreSQL). Besides that, you can also use the rulebox tool to apply specific quality operations (e.g. exploration, constraint detection, error localization,...) on data persisted in a database quite easily. In the remainder of this section, you will explore the data persisted in the `design` table of the `clinicaltrials` database (i.e. the clinical trials dataset) by

[4] `https://www.java.com/en/download/manual.jsp`
[5] `https://www.postgresql.org`
[6] `https://www.pgadmin.org`
[7] Do not copy the password from this file, but insert it manually to avoid any mistakes.

means of both a database client and rulebox. For most of the following exercises, we will indicate whether you have to complete it by using a database client and SQL, or by using the rulebox tool, by adding the abbrevation DB (resp. RB) after the corresponding exercise. A brief overview on how to use rulebox is given in the Appendix of this workshop.

Connect to the `clinicaltrials` database and inspect all data in this database. (DB)

Besides simply selecting all data in a database table, you can use the rulebox tool to explore (part of) the persisted data. This can be done by means of the `explore` command sequence. As stated in the Appendix, you always have to pass a databinder file after the `--d` option for each `explore`-based command sequence of rulebox. Fortunately, we have already provided such a .json databinder file for the `clinicaltrials` database, which you can use during the remainder of this workshop, on Ufora (Content > Data Quality > Edit rules > `clinicaltrials.json`).

- Get the total number of rows and attributes in the `design` table (DB & RB).
- Get the datatype of all attributes in the `design` table (DB & RB).
- Get the total number of unique, non-`NULL` values per attribute in the `design` table (DB & RB).

If you have completed the previous exercise correctly, you should see that the data in the `design` table consist of 35945 rows and 4 attributes. Each row contains data related to the masking/blindness of a clinical trial that is adopted in both EudraCT and ClinicalTrials.gov. The four attributes are `open`, `single_blind`, `double_blind` and `masking`. The first three attributes originate from EudraCT and can contain two different string values (‘Yes’ and ‘No’). The last attribute can contain four different string values (‘0’, ‘1’, ‘2’, ‘>2’) and originates from ClinicalTrials.gov. Below, we have listed the meaning of each attribute in short.

- `open`: a trial in which both the subjects and the investigators are aware of the treatment assignment.
- `single_blind`: a trial in which only one of the parties (mostly investigators) is aware of the treatment assignment.
- `double_blind`: a trial in which none of the parties is aware of the treatment assignment.

- `masking`: categorical attribute related to the masking/blindness of the clinical trial (0 = open, 1 = single blind, 2 = double blind, >2 = triple/quadruple).

Now you know which data are stored, you can explore data which comply with a certain boolean proposition.

- Get the total number of rows in which attribute `single_blind` takes value 'Yes' and attribute `masking` takes values '1' in the `design` table (DB & RB).
- Get the total number of rows in which attribute `masking` takes a value containing number '2' (so '2' or '>2') in the `design` table (DB & RB).

Finally, `NULL` indicates that the value of a certain row for the corresponding attribute is missing. The more missing values, the more difficult it is to accurately analyze data.

- Get the total number of rows containing at least one `NULL`-value in the `design` table (DB & RB).
- Get, for each attribute, the total number of rows containing a `NULL`-value for the corresponding attribute in the `design` table (DB & RB).

4 Edit rules

Now we have explored the clinical trials dataset, it is time to assess the consistency of these data. We have already mentioned that the data originate from two different sources and therefore, we will assess the consistency of these data (1) within each source (intra-consistency) and (2) between the two sources (inter-consistency).
To assess the consistency (or more general, the quality) of the data, we will follow a rule-based approach, which is highly connected with the following (general) workflow.

- First, one starts by defining a set of data quality rules. These rules feature constraints on attribute values, on attribute(-value) combinations within one data object (row), on attribute(-value) combinations over multiple data objects or even over multiple datasets.
- Each data object is validated against each rule and results in a boolean value indicating satisfaction of the rule by the data object.

- The fewer rules that are satisfied by a data object, the lower the quality of the data object. If all rules are satisfied by a data object, the quality of the data object cannot be improved. If all data objects satisfy all rules, the quality of the dataset cannot be improved.

4.1 Definition

Edit rules provide an elegant way to represent row-level constraints on different types of data (continuous, categorical, integer,...). During this lab session, we will focus on *categorical* edit rules. The general form of a categorical edit rule E is

$$E_1 \times E_2 \times \ldots \times E_k$$

in which each $E_i \subseteq A_i$ where A_i is the domain of attribute a_i and consists of all values that a_i can take (with the exception of the `NULL`-value). An edit rule E specifies a subset of the space of all possible value combinations $A_1 \times A_2 \times \ldots \times A_k$. If a data object (row) is an element in the subset represented by E, the row *fails* or *violates* E. Otherwise, the row *satisfies* E.
Considering the clinical trials dataset, we have already defined 7 edit rules, listed in Table 1. As an example, consider edit rule E^1: $\{\text{Yes}\} \times A_{sb} \times \{\text{Yes}\} \times A_m$. This rule explic-

Table 1: 7 edit rules defined on the clinical trials dataset.

edit rule	`open`	`single_blind`	`double_blind`	`masking`
E^1	Yes	A_{sb}	Yes	A_m
E^2	A_o	Yes	Yes	A_m
E^3	Yes	Yes	A_{db}	A_m
E^4	A_o	No	A_{db}	1
E^5	A_o	A_{sb}	No	2
E^6	A_o	A_{sb}	Yes	0,1
E^7	A_o	Yes	A_{db}	0

itly states that value 'Yes' of attribute `open` and value 'Yes' of attribute `double_blind` are not permitted to appear together in a data object, regardless of the values of attributes `single_blind` and `masking`. Indeed, it is not possible that the masking of a clinical trial is both open and double blind. Recall from the theory lecture that attributes `open` and `double_blind` are *involved* in E^1, because $E^1_o \subset A_o$ and $E^1_{db} \subset A_{db}$.

Before continuing, try to understand the underlying semantics of all edit rules defined in Table 1. Give, for each edit rule, which attributes are involved in the corresponding rule.

List, for each of the following rows, which edit rules defined in Table 1 the rows fail or potentially fail (in case of missing values).

- `open`: Yes, `single_blind`: No, `double_blind`: No, `masking`: 2
- `open`: No, `single_blind`: No, `double_blind`: Yes, `masking`: >2
- `open`: Yes, `single_blind`: Yes, `double_blind`: No, `masking`: 0
- `open`: Yes, `single_blind`: No, `double_blind`: Yes, `masking`: 2
- `open`: Yes, `single_blind`: No, `double_blind`: `NULL`, `masking`: 0

In order to avoid manual validation of each row in the clinical trials dataset, we can use rulebox and PostgreSQL again. As stated in the Appendix, if you want to use the error detection functionality of rulebox, you have to construct a .rbx file containing all rules defined on the data and pass it after the `--c` parameter of any `detect errors`-based command sequence. We have uploaded a .rbx file with the name `clinicaltrials.rbx` in which edit rules E^1 and E^6 are already defined on Ufora (Content > Data Quality > Edit rules > `clinicaltrials.rbx`). If you inspect the content of this file, you see that on lines 4–7 (resp. lines 10–13), the datatypes (resp. domains) of the attributes are defined. After this, it is possible to add edit rules.

Complete the file `clinicaltrials.rbx` by defining all edit rules listed in Table 1.

Now, we can analyze which rows in the clinical trials dataset fail which edit rules. Notice in the following task that the constraints on the domains of the attributes are also handled as edit rules by rulebox.

- Print all rows in the `design` table that fail at least one edit rule (DB & RB).
- Get the total number of rows in the `design` table failing edit rule E^1 (DB & RB).

- Get the total number of rows in the `design` table failing both edit rules E^1 and E^6 (DB & RB).
- Print the rows in the `design` table that fail most edit rules (RB).
- Get the total number of rows in the `design` table failing at least two edit rules (RB).

4.2 Error localization

Now we have validated which rows in the clinical trials dataset fail the given set of edit rules and investigated how many times an edit rule is failed, we can shift our attention to *error localization*. During the process of error localization, we are trying to find a set of attributes for which the values in row r can be changed such that the adapted row r' does not fail any rules. Such a set of attributes is called a *solution* S of r.

Recall from the theory lecture that finding solutions comes down to finding *set covers* of failing rules. As an example, consider the following row r: `open`: Yes, `single_blind`: No, `double_blind`: Yes, `masking`: 2. This row only fails edit rule E^1, which involves attributes `open` and `double_blind`. Therefore, in order to make r satisfy E^1, at least the value of attributes `open` or `double_blind` should be adapted. Adapting the value of `single_blind` or `masking` will not influence satisfaction of E^1. As a result, {`open`}, {`double_blind`} and each superset of these sets are set covers of $\{E^1\}$.

Moreover, value adaptation comes at a certain cost $w(a)$, which, in its most simple form, depends on an attribute a. A minimal solution is a solution for which the sum of the costs to change the attributes in this solution is minimized. If we reconsider the example above and suppose that the cost to change any attribute is 1, set covers of $\{E^1\}$ for which the cost is minimized are only {`open`} and {`double_blind`} (cost 1). Any other set cover of $\{E^1\}$ has a cost that is greater than 1 and is thus not minimal.

Suppose the cost to change each attribute in the clinical trials dataset is 1. Give all minimal set covers of the edit rules in Table 1 failed by the following rows.

- `open`: Yes, `single_blind`: No, `double_blind`: No, `masking`: 2
- `open`: No, `single_blind`: No, `double_blind`: Yes, `masking`: >2
- `open`: Yes, `single_blind`: Yes, `double_blind`: No, `masking`: 0
- `open`: Yes, `single_blind`: No, `double_blind`: Yes, `masking`: 2

What are the minimal set covers if we adapt the cost to change attributes `open` and `single_blind` to 2?

4.3 Implication of edit rules

As explained during the theory lecture, the strategy of finding set covers of failing rules is not guaranteed to work. More precisely, a (minimal) set cover of failing edit rules is not always a correct (minimal) solution to the error localization problem.
As an example, consider again row r: `open`: Yes, `single_blind`: No, `double_blind`: Yes, `masking`: 2. As we saw in the previous, this row only fails rule E^1 and minimal set covers of $\{E^1\}$ are {`open`} and {`double_blind`}. If we choose {`open`} as our solution, then we have to change the value of `open` from 'Yes' to 'No' to get row r'. As r' does not fail any edit rules (check this!), {`open`} is a correct solution. On the other hand, if we change the value of `double_blind` from 'Yes' to 'No' to get row r', we can easily see that r' satisfies E^1, but it does not satisfy E^5. Therefore, the set cover {`double_blind`} is not a correct solution of row r.

Earlier, you have listed all set covers of edit rules in Table 1 failed by the given rows (for two cost functions). Check now, manually, for each set cover, whether it is a correct solution of the row or not.

To avoid the problem explained above, it is important that the set of edit rules captures *all* information, even if it is implicit. Indeed, from the combination of E^1 and E^5, information can be derived that is not explicitly listed in the given set of edit rules, stating that value 'Yes' of `open` is not permitted to appear together with value '2' of `masking`. More formally, we can make all implicit information explicit, by repeatedly applying the following implication procedure.

- Consider any set of $n > 1$ edit rules, which we will denote by $\mathcal{E}_c$ and call the contributing set.
- Consider any attribute a_g, which we will call the generator.
- Consider all value sets of attribute a_g for each rule in $\mathcal{E}_c$. Take the union of these value sets to construct E_g^*
- Consider, for each attribute $a_i \neq a_g$, the value sets of a_i for each rule in $\mathcal{E}_c$. Take the intersection of these value sets to construct E_i^*.
- The edit rule $E^* = E_1^* \times \ldots \times E_g^* \times \ldots \times E_k^*$ is called an *implied edit rule* if no $E_i^* = \emptyset$. If any $E_i^* = \emptyset$, E^* is called a tautology, because it can never be failed.

Moreover, if a_g is involved in each rule in $\mathcal{E}_c$, but not in E^*, E^* is called an *essentially new* edit rule.
As an example, take $\mathcal{E}_c = \{E^1, E^5\}$ and $a_g = \texttt{double\_blind}$. The union of the value sets of the rules in $\mathcal{E}_c$ for a_g equals the domain of `double_blind`, A_{db}. The intersection of the value sets of the rules in $\mathcal{E}_c$ for attributes `open`, `single_blind` and `masking` are resp. {Yes}, A_{sb} and {2}. In the end, we have the implied rule $E^* = \{\text{Yes}\} \times A_{sb} \times A_{db} \times \{2\}$, that is also essentially new.

Give for each given contributing set $\mathcal{E}_c$ and generator a_g the implied edit rule E^* starting from the rules listed in Table 1. Validate also whether the rule E^* is essentially new or not.

- $\mathcal{E}_c = \{E^1, E^3\}$, $a_g = \texttt{double\_blind}$
- $\mathcal{E}_c = \{E^4, E^5, E^6\}$, $a_g = \texttt{masking}$
- $\mathcal{E}_c = \{E^5, E^6\}$, $a_g = \texttt{double\_blind}$
- $\mathcal{E}_c = \{E^3, E^4\}$, $a_g = \texttt{single\_blind}$
- $\mathcal{E}_c = \{E^4, E^5\}$, $a_g = \texttt{masking}$

4.4 Complete set generation

In the previous section, you learned how to generate implied edit rules from a given set $\mathcal{E}$. Additionally, we state that, if one generates all essentially new edit rules, starting from a given set $\mathcal{E}$, and add these rules to $\mathcal{E}$, a *complete set* of edit rules is obtained, denoted by $\Omega(\mathcal{E})$. A complete set of edit rules has the important property that finding all (minimal) solutions to the error localization problem for row r comes down to finding all (minimal) set covers of rules in $\Omega(\mathcal{E})$ that are failed by r.
A final problem that we have to solve is the problem of generating a complete set efficiently. By repeatedly applying the implication procedure, introduced in 4.3, one can generate all implied edit rules (including all essentially new edit rules), but this can be very inefficient, as many combinations of edit rules exist which can lead to an essentially new edit rule. Therefore, a more efficient algorithm is the Field Code Forest (FCF) algorithm, which is first reported by Garfinkel, Kunnathur and Liepins[8] and later enhanced by Boskovitz[9].
Before we are going to study the algorithm, an important property of the algorithm is that it does not necessarily generate all essentially new edit rules, but only the

[8] Garfinkel, Kunnathur and Liepins, Optimal Imputation of Erroneous Data: Categorical Data, General Edits, 1986.
[9] Boskovitz, Data Editing and Logic: The Covering Set Method from the Perspective of Logic, 2008.

essentially new edit rules which are non-*redundant*. The reason for this is that edit rules that are redundant, are not necessary to generate in order to solve the error localization problem correctly by means of the set cover method. Formally, an edit rule E is redundant to an edit rule E' if $E_1 \times \ldots \times E_k \subseteq E'_1 \times \ldots \times E'_k$. We say in this case that E' *dominates* E.

List, for each of the following edit rules, all edit rules given in Table 1 that are dominating the given edit rule.

- $\{\text{Yes}\} \times \{\text{No}\} \times \{\text{Yes}\} \times \{1\}$
- $\{\text{Yes}\} \times A_{sb} \times A_{db} \times \{1\}$
- $A_o \times \{\text{Yes}\} \times \{\text{Yes}\} \times \{1,2\}$

In summary, the FCF algorithm is going to generate a *sufficient set* of edit rules, denoted by $\underline{\Omega}(\mathcal{E})$, from a given set $\mathcal{E}$. This set consists of (1) the non-redundant edit rules in the given set $\mathcal{E}$ and (2) the non-redundant, essentially new edit rules generated from $\mathcal{E}$. The algorithm can be subdivided in the following steps.

1. Generate a Field Code Forest for attributes $a_1, \ldots, a_k$.
2. Visit each node in the Field Code Forest in depth-first order and in each node N:
   a) Select the rules $\mathcal{E}_N$ which belong to node N;
   b) Generate all essentially new rules from $\mathcal{E}_N$;
   c) Clean the rules in the nodes that are already visited.
3. After visiting the last node, the union of the cleaned rules attached to all nodes equals $\underline{\Omega}(\mathcal{E})$.

In the following, we will explain each step in detail.

4.4.1 Field Code Forest (step 1)

Suppose that our dataset consists of k attributes $a_1, \ldots, a_k$ (in random order). A Field Code Forest is a tree data structure in which each node is labeled with a concatenation of attribute indices $1, \ldots, k$, such that

- The root node is labeled with the empty string.
- The labels of the child nodes of a node N are labeled with the label of N concatenated with one of the higher indices. One child node per higher index exists and as a result, the label of the leaf nodes ends always with k.

An example of an FCF data structure for 4 attributes is given in Figure 1.

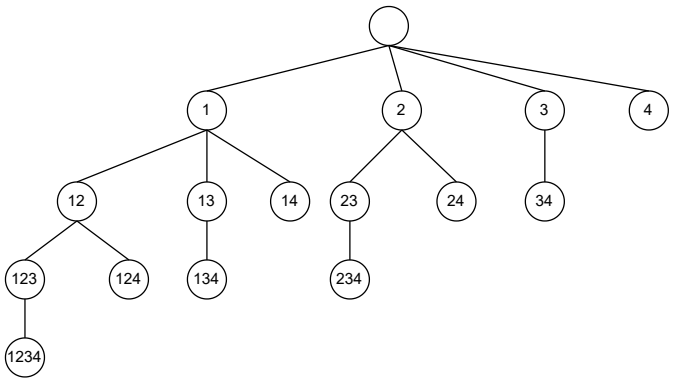

Figure 1: Field Code Forest data structure based on 4 attributes

4.4.2 Generating rules

Once the Field Code Forest is defined, we are going to iterate through the nodes of the tree in depth-first order (go as deep in the tree as possible before backtracking). In each node N with label $(j_1,\ldots,j_g)$, all rules generated in ancestor nodes (nodes higher than the current node in the same branch) of N, in which attribute a_{j_g} is involved, but attributes $a_{j_1},\ldots,a_{j_{g-1}}$ not, are selected in the set $\mathcal{E}_N$. Only in the root node, no rules are selected, but the initial set $\mathcal{E}$ is generated. From $\mathcal{E}_N$, all essentially new rules are generated with a_{j_g} as generator. If no essentially new rules (both redundant and non-redundant) are generated in a node, its children will not be visited anymore!
As an example, consider the edit rules defined on the clinical trials dataset and attribute order 1: `open`, 2: `single_blind`, 3: `double_blind` and 4: `masking`. In node (1) of the Field Code Forest, all rules are selected in which attribute `open` is involved from the initial set (E^1 and E^3) and all essentially new edit rules are generated from this selection. In node (12), all rules are selected from the initial set of (cleaned) rules and the set of (cleaned) essentially new rules generated in node (1) in which `single_blind` is involved, but `open` not, but only if there is an essentially new rule generated in node (1).

4.4.3 Cleaning rules

After the essentially new edit rules are generated in node N, we are visiting each previously visited node again, as well as the node currently under investigation (so, not only the previously visited nodes in the current branch!). If we encounter a rule that is redundant to any of the other rules, we are substituting this rule with any maximal dominating rule.
As an example, suppose that an essentially new edit rule E^d is generated in node

(13) which dominates an essentially new edit rule E^r generated in node (1), then we substitute E^r in node (1) with E^d during the cleaning step of node (13). If an essentially new edit rule E^d is generated in node (1) which dominates an essentially new edit rule E^r generated in node (2), than we substitute E^r in node (2) with E^d during the cleaning step of node (2). If two essentially new edit rules E^d and E^r are generated in the same node, and E^d dominates E^r, discard E^r in this node.

4.4.4 FCF in rulebox

The FCF algorithm is part of the reasoning functionality of rulebox (cfr. Appendix), so you can use it to generate a sufficient set of edit rules. However, make sure that you understand the algorithm entirely, because we can test this during the assignment on edit rules or on the exam.

1. Execute the FCF algorithm manually given the set of edit rules $\mathcal{E}$ listed in Table 1 to construct a sufficient set $\underline{\Omega}(\mathcal{E})$. Consider the order of the attributes as 1 = `open`, 2 = `single_blind`, 3 = `double_blind` and 4 = `masking`.
2. Validate whether your result of the previous exercise is correct by executing FCF in rulebox.
3. Again, list for each of the rows given in the first task on page 6, which edit rules in $\underline{\Omega}(\mathcal{E})$ the rows fail or potentially fail. List all minimal set covers (assuming the cost to change an attribute is 1) and check whether these minimal set covers are correct solutions to the error localization problem.

After generating a sufficient step, the next step in the process is to adapt, per row r, the values for each attribute in its solution such that the adapted row r' does not fail any edit rules. This step is called *imputation*, but will not be covered during this lab session as there are more advanced courses during the education that deal with imputation methods.

Extra: Consider the schema $\mathcal{R}$ with following attributes and corresponding domains.

- a_1: {1, 2, 3}
- a_2: {1, 2}
- a_3: {1, 2}

- a_4: {1, 2, 3}

Besides that, a set of six edit rules $\mathcal{E}$ defined over $\mathcal{R}$ is given.

- E^1: $\{1\} \times A_{a_2} \times A_{a_3} \times \{1, 3\}$
- E^2: $A_{a_1} \times \{2\} \times A_{a_3} \times \{2, 3\}$
- E^3: $A_{a_1} \times \{1\} \times A_{a_3} \times \{1\}$
- E^4: $A_{a_1} \times A_{a_2} \times \{1\} \times \{3\}$
- E^5: $A_{a_1} \times A_{a_2} \times \{2\} \times \{1, 2\}$
- E^6: $\{2, 3\} \times A_{a_2} \times A_{a_3} \times \{2\}$

Generate a sufficient set $\underline{\Omega}(\mathcal{E})$ from $\mathcal{E}$ by applying the FCF algorithm.

Appendix: `rulebox`

During this lab session, we will extensively use our rulebox data quality tool[10], which is developed in Java. In this appendix, we will briefly explain how this tool can be used.

Command structure

The rulebox tool is bundled as a .jar file (i.e. a Java executable), which means that you have to run it as a Java command on the command line (or terminal) of your computer[11]. Each command has the following (general) structure.

```
java -jar rulebox.jar <command sequence> <parameters>
```

First, you can substitute `<command sequence>` with any of the available sequences. To list the different command sequences, just run `java -jar rulebox.jar` on the command line. As the names suggest, each command sequence features a certain (quality) operation. Second, each command sequence comes with different parameters. These parameters (and their corresponding values) can be added at the end of each command (therefore, substitute `<parameters>` with the desired parameters and values). In the remainder, we will briefly introduce the command sequences and corresponding parameters that are most important for completing this lab session successfully.

Exploration

Each command sequence to explore data in a dataset starts with the word `explore`. Moreover, when exploring data, you always have to specify where the data that you want to explore is located, by means of the `--d` parameter. After this parameter, you have to type the path to a .json databinder file containing information about the location of the data. This can either be a .csv file or a JDBC data source (i.e. connection to a relational database). More information on the structure of these .json files can be found here.
Command sequences that can be useful for data exploration during this lab session are the following.

- `explore header`: shows the attributes of the dataset and their datatypes.
- `explore stats`: compute basic statistics of a dataset.

As an example, the following command can be used to show all attributes and their datatypes of a table in the `clinicaltrials` database, for which connection information is stored in the `clinicaltrials.json` databinder file.

[10] `https://gitlab.com/ledc/ledc-rulebox/-/blob/main/docs/index.md`
[11] Make sure to navigate on the command line to the location where the `rulebox.jar` file is located.

```
java -jar rulebox.jar explore header --d clinicaltrials.json
```

Notice that, when you want to explore data stored in a relational database, you are asked to write a `SELECT`-query upon executing each `explore`-based command. This is because data can only be explored in the result table of a passed `SELECT`-query. For example, if you want to explore *all* data in the `design` table of the `clinicaltrials` database, you can type `SELECT * FROM design;`.

Error detection

Each command sequence that can be used to retrieve information related to rule-based error detection starts with the words `errors detect`. Again, you have to specify where the data on which you want to apply error detection is located by means of a .json databinder file after the `--d` parameter (cfr. exploration). Moreover, you have to provide the path to a .rbx file containing the data quality rules defined on the data after the `--c` parameter. More information on the structure of these .rbx files can be found here. Command sequences that can be useful for error detection during this lab session are the following.

- `errors detect rows`: reports detected errors per row in the dataset in an aggregated way. Other useful (optional) parameters of this command sequence are
  - `--se`: explicitly show rows with errors and their violations, and
  - `--srf x`: only show rows if they fail at least x rules (can only be used in combination with the `--se` parameter).
- `errors detect sigma`: reports violations of σ-rules (edit rules are a specific case of σ-rules).

As an example, the following command can be used to show all rows in the `clinicaltrials` database, for which connection information is stored in the `clinicaltrials.json` databinder file, that fail at least 3 rules defined in `clinicaltrials.rbx`.

```
java -jar rulebox.jar errors detect rows
    --d clinicaltrials.json
    --c clinicaltrials.rbx
    --srf 3
    --se
```

Again, you are asked to write a `SELECT`-query in order to construct a result table on which error detection is applied.

Reasoning

A final subset of the rulebox functionality that is relevant for solving this lab session is reasoning about data quality rules. Each command sequence that can be used for this starts with the word `reason`. For this command sequence, no databinder file is expected, but you have to provide the path to a .rbx file containing the data quality rules about which rulebox is going to apply reasoning after the `--c` parameter (cfr. error detection). The only `reason`-based command sequence that will be useful during this lab session is `reason fcf`. This command is going to apply the FCF algorithm (4.4) on edit rules defined in the passed .rbx file. Besides the `--c` parameter, it also expects a `--of` parameter, specifying the path to a .rbx outputfile. After completing FCF, rulebox is going to write all rules generated by FCF to this outputfile.
As an example, the following command can be used to apply the FCF algorithm on the edit rules defined in `clinicaltrials.rbx` and to write all rules generated by FCF to `clinicaltrials_sufficient.rbx`.

```
java -jar rulebox.jar reason fcf
    --c clinicaltrials.rbx
    --of clinicaltrials_sufficient.rbx
```

Redirect output

In order to write the output of a rulebox command that does not have a `--of` parameter to a file, you can add `> path/to/file` after the command. As an example, to write the output of `java -jar rulebox.jar explore header --d clinicaltrials.json` to a file with name `output.txt`, you can execute

```
java -jar rulebox.jar explore header --d clinicaltrials.json > output.txt
```