BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Master's Thesis

Robust Machine-Learning Approaches for Efficient Functional Dependency Approximation

by

Philipp Jung

Philipp Jung
Matriculation Number: 872855
16.03.2019

Advisors:
Prof. Dr. Biessmann
Prof. Dr. Schoeneberg

ABSTRACT. Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

Contents

1 Introduction

Data-driven methods change the way computer scientists approach algorithmic problems. Rather than designing and implementing complex algorithms themselves, recent advances in machine learning have allowed for learned algorithms. While these learned algorithm come with their own limitations and problems, e.g. lack of explainability, some of them have proven to solve classic algorithmic problems in a more performant fashion.

Kraska et al. showed in their 2018 publication "The case for Learned Index Structures" that different index structures can be replaced by learned ones, greatly improving performance.[Kra+18]

In the field of data cleaning and data enrichment, HoloClean lead the way for machine-learning approaches in the domain of data cleaning.[Hei+19] HoloClean is agnostic of the way data is structured, making it versitile for many different domains of application.

In this work, machine-learning techniques are applied to the field of relational database theory — more precisely, functional dependency detection.

Stemming from the early days of relational database theory, functional dependencies were introduced to formalize schema normalization.

In this works' theory section, basic relational database terminology is introduced. The application of functional dependencies in normalization is presented. Furthermore, limitations of canonical functional dependencies are mentioned and relaxed functional dependencies are introduced. With reference to Koudas et al., functional dependencies' robustness is discussed. A way of measuring robustness is proposed. Ultimately, machine-learning classification theory necessary for understanding the basic functionality of Datawig[Bie+18] is discussed.

Several experiments are conducted to explore machine-learning techniques working with functinoal dependencies. **continue here with description of experiments**.

2 Theory

In this section, *functional dependencies* (FDs) and the theoretical foundation necessary to put them into context, are introduced. Common normal forms are briefly defined. Building on this foundation, *relaxed functional dependencies* are established as a way of adapting FDs for use-cases other than database normalization.

The term *robustness* of FDs is discussed and defined. A way of measuring robustness with cross-validation techniques is presented. Lastly, machine learning classifier theory is reviewed, introducing *Datawig Imputer*.

2.1 Introduction of Functional Dependencies

FDs are a way of expressing "a priori knowledge of restrictions or constraints on permissible sets of data". [Mai83, p. 42] In order to give a definition of FDs, they need to be put in context to the domain they stem from: relational database theory.

Definition (Relation Scheme, Attribute Names, Domain, Relation, Tuples) A *relation scheme*[1] $\boldsymbol{R}$ is a finite set of *attribute names* $\boldsymbol{R} = \{A_1,\ A_2,\ \ldots,\ A_n\}$, where to each attribute name A_i corresponds a set D_i, called *domain* of A_i, $1 \leq i \leq n$. Let $\boldsymbol{D} = D_1 \cup D_2 \cup \cdots \cup D_n$, then a *relation* r on relation scheme $\boldsymbol{R}$ is a finite set of mappings $\{t_1, t_2, \ldots, t_p\}$ from $\boldsymbol{R}$ to $\boldsymbol{D}$:

$$t_i : \boldsymbol{R} \rightarrow \boldsymbol{D}$$

We call those mappings *tuples* under the constraint that [Mai83, p.2]

$$t(A_i) \subseteq D_i.$$

In application, attribute names are commonly called *column name* or *column attribute*. One can think of them as labels of data that is stored in the respective column.

Definition (Functional Dependency) Consider a relation r on scheme $\boldsymbol{R}$ with subset $X \subseteq \boldsymbol{R}$ and a single attribute $A_i \in \boldsymbol{R}$. A FD $X \rightarrow A$ is said to be *valid* in r, if and only if

$$t_i[X] = t_j[X] \Rightarrow t_i[A] = t_j[A] \tag{1}$$

holds for all pairs of distinct tuples $t_i, t_j \in r$. [Abe+19, p. 21] We say that X *functionally determines* A [Mai83, p. 43] and write X → A. X is called the *left hand side* (LHS) of a FD, whilst A is called the *right hand side* (RHS) of the same FD. A FD X → Y is called *trivial*, if $Y \subseteq X$.[Stu16, p. 163]

[1] also called *relational schema* in literature[Abe+19, p.21]

Example Considering table 1, one can see that every tuple in the left hand side subset of the relation uniquely determines the right hand side. We say that Id, Prenameame, Surname, Town *functionally determines* Zip or write {Id, Prename, Surname, Town } → Zip. [Mai83, p. 43]

left hand side				right hand side
Id	Prename	Surname	Town	Zip
1	Alice	Smith	Munich	19139
2	Peter	Meyer	Munich	19139
3	Ana	Parker	Munich	19139
4	John	Pick	Berlin	12055
5	John	Pick	Munich	19139

Table 1: Example for a FD.

If inspected closely, one can discover even more FDs in table 1. For example, Town → Zip and Id → Zip. Since Town and Id are subsets of {Id, Prename, Surname, Town}, we call the FD {Id, Prename, Surname, Town} → Zip *non-minimal*.

Definition (Minimal FD) A FD X → A is *minimal*, if no subset of X functionally determines A. [Pap+15, p. 2] Thus, Id → Zip and Town → Zip are *minimal FDs*.

Definition (Logical Implication) Let F be the set of FDs on $\boldsymbol{R}$. X → Y is *logically implied* by F, or algebraically expressed

$$F \models \mathrm{X} \to \mathrm{Y}, \tag{2}$$

if every relational instance r of $\boldsymbol{R}$, which satisfies all dependencies in F, also satisfies X → Y. [Stu16, p. 166]

Thus, if $F = \{\mathrm{A} \to \mathrm{B},\ \mathrm{A} \to \mathrm{C},\ \mathrm{BC} \to \mathrm{D}\}$, the following logical implications are true:

$$F \models \mathrm{A} \to \mathrm{B}$$
$$F \models \mathrm{A} \to \mathrm{BC}$$
$$F \models \mathrm{A} \to \mathrm{D}$$

Definition (Transitive Closure) Let F be a set of FDs. The *transitive closure* F^+ of F is defined by

$$F^+ := \{\mathrm{X} \to \mathrm{Y} \mid F \models \mathrm{X} \to \mathrm{Y}\} \tag{3}$$

Definition (Full Dependency, Determinant) Let **R** be a relation scheme and let Y $\in$ **R** be an attribute on **R**. Furthermore, let X $\subseteq$ **R** be a set of attributes on **R**. The attribute Y is called *fully dependent* on X, if Y is functionally determined by X and X is minimal. [Sch17, p. 61] We call a *determinant* a set of attributes that fully functionally determines another attribute.

2.2 Additional Relational Database Theory

When real-world data used by an application is stored on a machine according to the relational model, it is usually stored in a relational database. *Database normalization* is the original domain of application for FDs. [Cod70, p. 381] In the following section, concepts necessary for introducing databases are defined. In addition, terms used in database normalization are defined.

Definition (Superkey) Let r be a relation and let $\boldsymbol{R} = \{A_1, A_2, \ldots, A_n\}$, $n \in \mathbb{N}$, be a relation scheme on which r is defined. Let K be a subset of R, such that $K = \{A_1, A_2, \ldots, A_m\} \subseteq \boldsymbol{R}$, where $m \leq n$, $m \in \mathbb{N}$. The subset K is called *superkey*, if for any tuple $t_i \in r$ the relation

$$t_i(A_k) = t_j(A_k) \Rightarrow t_i \equiv t_j$$

holds for any single $A_k \in K$. [Mai83, p. 4] In other words, if K is a superkey, any K-value of a tuple identifies that tuple uniquely. [Sch17, p. 32]

Definition (Candidate Key) A superkey K is called *candidate key*, if it is *minimal*. [Sch17, p. 32] The notion of K being minimal means that K is no longer a superkey, if any attribute $A_k \in K$ is removed from K.

Definition (Prime Attributes) An attribute A on a relational scheme $\boldsymbol{R}$ is called *prime*, if A is part of a key of $\boldsymbol{R}$. Otherwise, A is called *non-prime*.

Definition (Relational Database) Following the definition of a relation scheme $\boldsymbol{R}$, one can formally introduce databases and database schemes:

We assume that $\boldsymbol{R}$ is composed of two parts, S and $\boldsymbol{K}$. We call S a *set of attributes* and $\boldsymbol{K}$ a *set of designated keys* and describe this composition by writing $R = (S, \boldsymbol{K})$. A *relational database scheme* $\mathcal{R}$ over $\boldsymbol{U}$ can now be defined as a collection of relation schemes

$\mathcal{R} = \{R_1, R_1, \ldots, R_p\}$, where $R_i = (S_i, \boldsymbol{K}_i)$, $1 \leq i, j \leq p$,

$$\boldsymbol{U} := \bigcup_{i=1}^{p} S_i$$

We demand that $S_i \neq S_j$ if $i \neq j$.

A *relational database* d on a *database scheme* $\mathcal{R}$ is a collection of relations $d = \{r_1, r_2, \ldots, r_p\}$ such that for each relation scheme $R = (S, \boldsymbol{K})$ in $\mathcal{R}$ there is a relation r in d such that r is a relation on S that satisfies every *key* in $\boldsymbol{K}$. [Mai83, p. 94]

2.3 Database Normalization

In 1970, Edgar F. Codd defined the *relational model* [Cod70] and pioneered many important concepts in database theory. When introducing the relational database model in his 1970 article "A relational model of data for large shared data banks", Codd formalized database normalization alongside. [Cod70]

Describing what will be know to academia as *First Normal Form* (1NF), Codd states that "problems treated [when normalizing databases] are those of *data independence*", aiming to protect future users of large databases "from having to know how the data is organized in the machine". [Cod70, p. 1]

Databases at the time were structured hierarchically or navigationally. This design was centered on efficiency, optimized for handling queries as fast as possible. While this design yielded good performance in an era when computing time was cost-intensive, it came with a heavy cost of complexity: "Teams of programmers were needed to express queries to extract meaningful information. [. . .] Such databases [. . .] were absolutely inflexible[y]". [IBM03]

Codd's relational model shifted the focus of database architecture away from efficiency towards a new design, centered around the user of a database. However, this new approach of database architecture came with new challenges. How does one create a database-scheme with favorable properties?

Generally, redundantly stored data is an indicator [Wat14, p. 61] that so-called *anormalies* might occur, leading to inconsistency when operating the database. [Stu16, p. 162] Kleuker writes that "with a certain 'database-sense' " one could sense when splitting tables up is necessary to prevent anormalies. [Kle11, p. 76]

This 'database-sense' was formalized when Codd introduced database normalization as part of the relational model. [Cod70, p. 381] Different *normal forms* were defined. Codd used three normal forms, which he called First-, Second- and Third Normal Form.[2]

Subsequently, Fourth-, Fifth- and even higher-order Normal Forms were defined by other authors. However, these normal forms find little reception in real-world applications. [Sch17, p. 58] Today, FDs are primarily used in database normalization. [CDP16, p. 1] Thus, the derivation of the most commonly used normal-forms is of interest within the scope of this work.

Definition (First Normal Form) A relation scheme R is in *First Normal Form* (1NF), if values in $dom(A)$ are atomic for every attribute $A \in R$. [Mai83, p. 96] Likewise, a database scheme $\mathbf{R}$ is in 1NF if every relation scheme $R \in \mathbf{R}$ is in 1NF.

Example Consider table 2 which represents two relational instances on two different relational schemes. It serves as an example of what is called atomic- and compound data in the Relational Database model. [Cod90, p. 6]

	Compound Scheme		Atomic Scheme			
	NAME	ADRESS	PRENAME	SURNAME	TOWN	STREET
1	Alice Smith	Munich, Flurstr.	Alice	Smith	Munich	Flurstr.
2	Peter Smith	Munich, Flurstr.	Peter	Smith	Munich	Flurstr.
3	Ana Parker	Munich, Anastr.	Ana	Parker	Munich	Anastr.
4	John Pick	Berlin, Flurstr.	John	Pick	Berlin	Flurstr.

Table 2: The compound attributes ADRESS and NAME can be split into their atomic components TOWN and STREET as well as PRENAME and SURNAME, respectively.

Note that the compound scheme's attributes can be decomposed into several other attributes. The atomic scheme's attributes however cannot be further split into any meaningful smaller components.

1NF is the very foundation of the Relational Model. It demands, that the only type of compound data is the relation itself. [Cod90, p. 6]

[2]The Third Normal Form was later newly formulated by Boyce and Codd and named Boyce-Codd Normal Form (BCNF). Due to its more elegant definition, BCNF is introduced rather than Codd's original 3NF.

Definition (Second Normal Form) A relation scheme R is said to be in *Second Normal Form* (2NF) in respect to a set of FDs F, if it is in 1NF and every nonprime attribute is fully dependent on every key of R. [Mai83, p. 99] This definition can be extended for databases: A database scheme $\mathbf{R}$ is in 2NF with respect to F if every relation scheme $R \in \mathbf{R}$ is in 2NF with respect to F.

It can be shown that a database scheme in 2NF is also in 1NF. [Sch17, p. 58]

Definition (Boyce-Codd Normal Form) A relation scheme R is in *Boyce-Codd Normal Form* (BCNF), if every determinant in R is a candidate key. [Sch17, p. 65] Equally, a relational database scheme $\mathbf{R}$ is in BCNF if every relation scheme $R \in \mathbf{R}$ is in BCNF.

If a database scheme is in BCNF, it is also in 2NF. [Sch17, p. 58] A database in BCNF effectively eliminates redundancies apart from candidate keys, freeing a database in BCNF from anormalies. [Sch17, p. 67]

2.4 Relaxed Functional Dependencies

When Edgar F. Codd introduced the relational model in 1970, the concept of keys and thus the concept of FDs was already present. [Mai83, p. 70] However, in 1972 Codd separated the definition of a FD from keys, persuing the specific goal of normalizing a relational database scheme.

In the ensuing time, researchers abstracted the concept of FDs and transferred the approach. FDs were used in research to solve problems other than schema normalization. [CDP16, p. 161] Many of these use-cases, such as 'approximate classification' or 'source merging', arose due to the fact that real-world datasets are almost always *noisy*.

Noise in databases has many faces: Entries might be corrupted by missing data, wrongly entered data or incomplete data. A database might have been merged with identical entries formatted in different ways. [Kou+09, p. 1] In these cases, functionally dependent column-combinations are not detected as such by a FD detection algorithm searching for FDs as defined in 1. This may result in misleading insights when searching for FDs or normalizing a database.

Taking these limitations of *canonical FDs*[3] into account, a multitude of new dependencies was defined, "aiming to solve specific problems". [CDP16, p. 147] Since all of these new

[3]To accentuate the difference between FD and RFD, FDs are also called canonical FDs. This notion refers to FDs as defined in equation 1.

kinds of dependencies *relax* the conditions in equation 1, they are called *relaxed functional dependencies* (RFDs).

Caruccio et al. classified and compared 35 different RFDs. [CDP16, p. 151] They state that each RFD was "based on its underlying relaxation criteria". [CDP16, p. 149] They proceed to define two relaxation criteria by which they distinguish all RFDs.

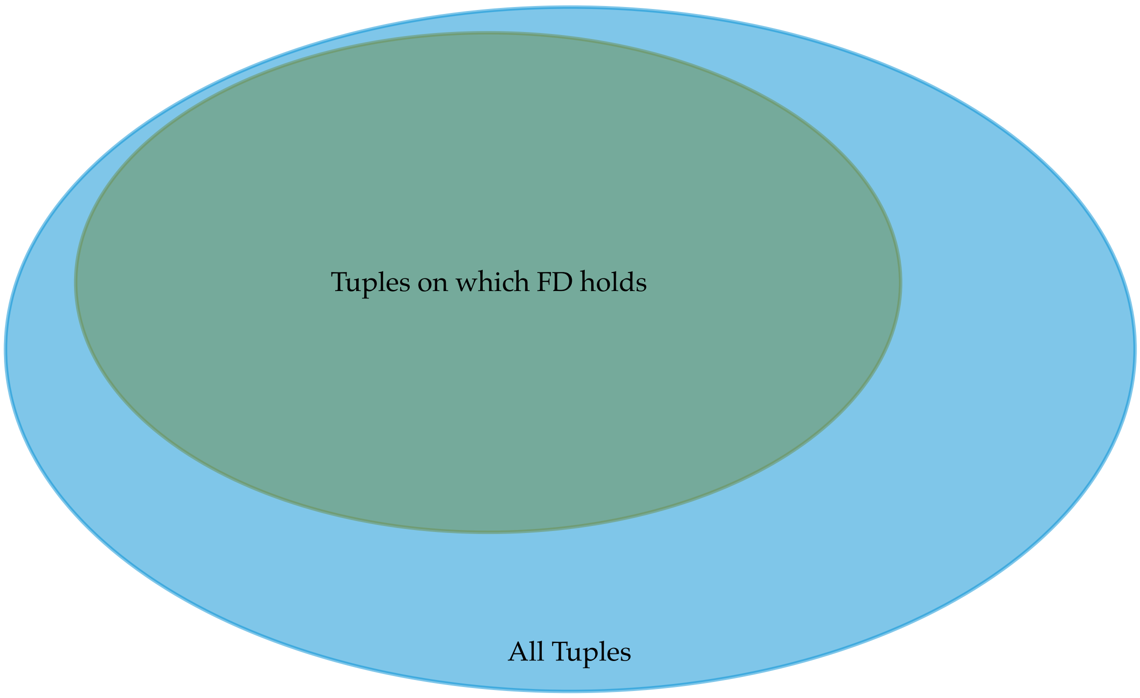


Figure 1: Venn diagram representing the extent relaxation criterion.

The first relaxation criterion is called *attribute comparison*. It refers to "the type of tuple comparison used on the LHS and RHS" in equation 1.

The relaxation criterion *extent* implies a relaxation of the set of tuples for which a FD is valid.

2.4.1 FDs relaxing on the Extent

The term *extent* is used to describe how a RFD relaxes on the subset of tuples for which the RFD is satisfied. While the definition of a FD demands that a FD's condition has to be valid for all tuples in a relational table, RFDs relaxing on the extent only hold on a subset of all tuples.

This idea is implemented by different RFDs in different ways. In the following section, a selection of RFDs relaxing the extent of the FD is briefly presented.

Approximate Functional Dependency (AFD) AFDs improve the applicability of FDs by "holding on *almost*[4] every tuple"[CDP16, p. 151]. To illustrate this, table 3 shows an example of noisy data. The potential FD Town → Zip is not captured by the definition of a canonical FD. Due to a typing error, the potential FD is invalidated in the row where ID = 2. To still capture the relation, a different dependency-measure than given in the definition of the canonical FD is required.

Id	First name	Last name	Town	Zip
1	Alice	Smith	Munich	19139
2	**Peter**	**Meyer**	**Muinch**	**19139**
3	Ana	Parker	Munich	19139
4	John	Pick	Berlin	12055

Table 3: Even though column Zip determines the content of column Town (and vice-versa), a FD is not capable of displaying this fact — a typing error invalidates the FD.

Tuples that do not correspond to the canonical FD are measured as fraction of the total of tuples on a relation r as follows:

$$\Psi(X, Y) = \frac{\min\left(|r_1| \mid r_1 \subseteq r \text{ and } \mathrm{X} \rightarrow \mathrm{Y} \text{ hold in } r \setminus r_1\right)}{|r|} \tag{4}$$

Here, function $\Psi(X, Y)$ is called *coverage measure* of a RFD X → Y. Ψ "quantifies the satisfiability degree of an RFD [...] on r" [CDP16, p. 150] and is used in the definition of an AFD to be compared to a threshold $\epsilon \in [0, 1]$.

If $\Psi(X, Y)$ is smaller or equal to ϵ, the AFD is said to hold on a relation r. Applied to table 3, the AFD Town → Zip holds, if $\epsilon \geq 0.25$.

Conditional Functional Dependencies (CFDs) *Conditional Functional Dependencies* employ conditions to define the subset on which a dependency holds. Originally, those conditions exclusively allow the definition of constraints using the equality operator. [CDP16, p. 152] Applied to table 3, a CFD Town → Zip holds under the condition that a) entries in column Town = Munich or b) entries in column Town = Berlin.

Other RFDs based on the definition of CFDs include *extended conditional functional dependencies* (ECFDs) by Bravo et al. that allow the disjunction-operator as well as the inequality-operator for defining conditions. [Bra+08]

[4]Highlighting by the author.

Also, Chen et al. defined CFD^{p}s, which introduce $<, \leq, >, \geq$ and $\neq$ for defining conditions. [CFM09]

2.4.2 FDs Relaxing on the Attribute Comparison

Instead of specifying subsets of tuples for which a FD is valid, FDs relaxing on the attribute comparison alter the condition under which a dependency is said to be valid. One common RFD relaxing on the attribute comparison is the metric functional dependency.

Metric Functional Dependencies (MFDs) First introduced by Koudas et al. in 2009, *Metric Functional Dependencies* were defined to adress violations of canonical FDs due to "small variations [. . .] in data format and interpretation". [Kou+09, p. 1]

According to equation 1, a FD Zip → Town is said to be valid if

$$t[\textsc{Zip}] = t'[\textsc{Zip}] \Rightarrow t[\textsc{Town}] = t'[\textsc{Town}] \tag{5}$$

holds for all pairs of distinct tuples $t_i, t_j \in r$, where r is a relation on scheme $\boldsymbol{R}$. The definition of a MFD replaces this equation by demanding that if and only if for all pairs of distinct tuples $t_i, t_j \in r$ the metric condition

$$t[\textsc{Zip}] = t'[\textsc{Zip}] \Rightarrow d(t[\textsc{Town}], t'[\textsc{Town}]) \leq \delta \tag{6}$$

holds, the MFD Zip → Town is said to be valid on r. [Kou+09, p. 2] Here, δ is called a *tolerance parameter*, indicating how far the result of a *metric* $d(t[\mathrm{Y}], t'[\mathrm{Y}])$ can deviate from exact equality.

One such metric d would be the absolute difference between two values, such that $d(t[\mathrm{Y}], t'[\mathrm{Y}]) = |t[\mathrm{Y}] - t'[\mathrm{Y}]|$. Another popular metric for measuring the difference between two sequences is the Levenshtein distance, where $d(t[\mathrm{Y}], t'[\mathrm{Y}]) = \mathrm{lev}_{t,t'}(|t|, |t'|)$ and

$$\mathrm{lev}_{t,t'} = \begin{cases} \max(i,j), & \text{if} \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Regarding the example given in table 3, the Levenshtein distance between `Munich` and `Muinch` is 2. Thus the MFD Zip → Town is valid, if $d = \mathrm{lev}$ and $\delta = 2$.

2.5 Robustness of FDs

Koudas et al., when introducing the MFD, write that the definition of the canonical FD was "not *robust* enough to capture functional relationships on data obtained from merging heterogenous sources [...]"[5]. [Kou+09, p. 1] However, they do neither provide a definition of robustness, nor a way to measure it.

Bleifuß et al. [Ble+16, p. 3] define a measure similar to robustness called *Correctness*. They do so by comparing a set of RFDs *Out* to a set of FDs *Gold* on the same relational instance r as follows:

$$\text{Correctness} = \frac{|Out \cap Gold^+|}{|Out|}$$

$Gold^+$ represents the transitive closure of *Gold*, meaning that it includes all non-minimal FDs implied by *Gold* as defined in equation 3.

By this definition, the notion 'Correctness' considers *Out* to be more or less 'correct' depending on how many members it has in common with $Gold^+$. If a RFD is used to find relations on noisy data, this definition of 'Correctness' does not meaningfully measure dependencies that try to work around the noise. This means that a RFD that takes the typing error in table 3 into account would not be more 'correct' than a RFD that does not.

In summary, Koudes et al. do not provide a definition of robustness and 'Correctness' cannot be used for measuring robustness. Drawing inspiration from Machine Learning theory, the application of cross-validation techniques to FDs order to measure robustness is proposed.

Definition (Train Set, Test Set, Split Ratio) Let $r = \{t_1, t_2, \dots, t_p\}$ be a relation on a relation scheme $\boldsymbol{R}$ where $p \in \mathbb{N}$ is the number of tuples in r. Let $s \in [0, 1]$ be the *split ratio* and let $m = \lfloor s \cdot p \rfloor$, then we define:

$$r_{train} = \{t_1, t_2, \dots, t_m\} \tag{7}$$

$$r_{test} = \{t_{m+1}, t_{m+2}, \dots, t_p\}, \tag{8}$$

such that $r = r_{train} \dot\cup r_{test}$ is split into two disjunct subsets. We call r_{train} the *train set* and r_{test} the *test set*.[6] [SV08, p. 56]

[5]Highlighting added by the author.

[6]The naming 'test set' is ambiguous in literature. Bishop [Bis06] and Smola et al. [SV08] call 'validation set' what is called 'test set' by Burkov [Bur19, ch. 5, p. 8-9]. In this work, the notation from [Bur19] is adopted.

Definition (Imputation Derived by a FD) Let r be a relational instance on a relational scheme $\mathbf{R}$. In addition, let $\mathrm{X} \subseteq \mathbf{R}$ and $\mathrm{A} \in \mathbf{R}$, $\mathrm{A} \notin X$. Furthermore, let $\mathrm{X} \rightarrow \mathrm{A}$ be a non-trivial, minimal FD of $r = r_{train} \dot{\cup} r_{test}$. We then call the RHS of all tuples $t_{train} \in r_{train}$ where

$$t_{train}[\mathrm{X}] = t_{test}[\mathrm{X}] \tag{9}$$

imputation derived by the FD $\mathrm{X} \rightarrow \mathrm{A}$ of $t_{test} \in r_{test}$. In consequence, we call

$$r^i_{imp} = \{t[A] \mid t[\mathrm{X}] = t_i[\mathrm{X}], t_i \in r_{test}, t \in r_{train}\} \tag{10}$$

set of imputations of the ith tuple in the test set.

Definition (Robustness) Let $r = \{t_1, t_2, \ldots, t_p\}$ be a relational instance on a relational scheme $\mathbf{R}$, $p \in \mathbb{N}$. Moreover let $\mathrm{X} \subseteq \mathbf{R}$ and $\mathrm{A} \in \mathbf{R}$, $\mathrm{A} \notin X$. Let $\mathrm{X} \rightarrow \mathrm{A}$ be a non-trivial, minimal FD of r. Furthermore, let $r = r_{train} \dot{\cup} r_{test}$ be split with split ratio $s \in [0, 1]$ and let $m = \lfloor s \cdot p \rfloor$.

For each tuple $t_i \in r_{test}$, $i \in \{m+1, m+2, \ldots, p\}$ exists a corresponding set of imputations r^i_{imp}. If $r^i_{imp} \neq \emptyset$, we arbitrarily choose one tuple t^i_{imp} from r^i_{imp}. We call

$$r_{imp} = \{t^i_{imp} \mid r^i_{imp} \neq \emptyset, i \in \{m+1, m+2, \ldots, p\}\} \tag{11}$$

set of imputations of the test split.

Treating the tuples in r_{imp} as predicted labels and the tuples in r_{test} as labels, we compute the F1-Score or the MSE, depending on the type of data the RHS contains. We call this Score 'Robustness' of FD $\mathrm{X} \rightarrow \mathrm{A}$.

2.6 FD Imputer: Measuring Robustness

In table 4 a splitting is examplified for $s = 0.5$ and $p = 4$. The approach and terminology involved in the definition of robustness are based on methods from statistical cross-validation used in model selection [Hay08, p. 172]. When measuring robustness, FDs are first detected on the train set using a FD detection algorithm. In this work, the HyFD algorithm is chosen. [PN16] The algorithm called *FD Imputer* is then used for measuring robustness of each FD detected on the train set.

FD Imputer FD Imputer iterates over all tuples in the test set and searches for a tuple in the train set with the exact same LHS. If one or more such tuples in the train set are found, FD Imputer randomly selects one of these tuples and saves the RHS of that tuple.

A	B	C	D
Blue	Car	Portugal	Lisbon
Yellow	Car	Portugal	Lisbon
Green	Bus	Spain	Madrid
Grey	Car	Portugal	Lisbon

(a) Original relation r.

A	B	C	D
Green	Bus	Portugal	Lisbon
Yellow	Car	Portugal	Lisbon

(b) Train set r_{train}.

A	B	C	D
Yellow	Bus	Spain	Madrid
Blue	Bus	Portugal	Lisbon

(c) Test set r_{train}.

Table 4: The relation r is split into train- and test sets with a split-ratio $s = 0.5$.

As defined in the previous section, this RHS value is called *imputation* of the RHS value on the train set.

This procedure is repeated for all tuples in the train set. In a last step, FD Imputer compares the imputations with the actual RHS values in the train set. This allows a classification of the imputations. If the FD's RHS contains categorical data, one can now calculate the F1-measure for each FD. Elsewise, in case the FD's RHS contains sequential data, the MSE is chosen as measure of robustness.

Consider table 4. The FD C $\rightarrow$ D, when used for finding a right hand side value on r_{test}, will correctly return 'Lisbon'. However, FD A $\rightarrow$ D for example will lead to FD Imputer yielding 'Madrid', which is an incorrect imputation.

The implementation of FD Imputer leverages SQL join clauses to retrieve imputations. FD Imputer performs an inner join on all LHS columns, joining train-set and test-set. A successive second left join clause concatenates the original test-set with the column of imputed tuples stemming from the first join. The result is a set of imputations as described in equation 11.

Robustness as a F1-Score The imputed values retrieved by FD Imputer can be leveraged to create a measure for robustness. First, each imputed value is interpreted as a label. These labels are then each classified as in the binary classification case (see figure 2). Next, precision and recall are calculated and the F1-Score is derived according to equation 16.

Lastly, the F1-Score weighted according to the number of true instances for each label is calculated.[7]

Algebraically, this means that to a set $L = \{l_1, l_2, \dots, l_n\}$ of $n \in \mathbb{N}$ labels, a set $X = \{x_1, x_2, \dots, x_n\}$ of n F1-Scores corresponds. Let W be a set of weights $W = \{w_1, w_2, \dots, w_n\}$, each being the number of true instances for each label. Then the weighted F1-Score of a dependency can be calculated as follows:

$$\text{F1-Score} = \frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i} \tag{12}$$

This F1-Score, calculated for each FD on a relation r, is named robustness of a FD.

Robustness as Mean Squared Error Depending on the kind of data imputed by FD Imputer, the measure used to express robustness is adapted. As shown in the previous paragraph, performance when imputing categorical data can be measured by the F1 Score. If the RHS's content is of a sequential datatype, using a classification performance measure is pointless.

Therefore, FD Imputer distinguishes between sequential and categorical data. When imputing sequential data, the *mean squared error* (MSE) is chosen as measure.

Let $\hat{Y} \in \mathbb{R}^n$ be the n-dimensional vector of imputed values and $\hat{Y} \in \mathbb{R}^n$ be the vector of actual RHS values. Then the MSE is defined as [Moo+11, p. 597]

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i\right)^2 \tag{13}$$

[7]See sklearn documentation for `sklearn.metrics.f1_score()` with weighted average for more details on the implementation.

2.7 Machine Learning Classifier Theory

Once a model has been trained and validated, it needs to be tested in order to determine whether or not overfitting occured during training. [Bis06, p. 32] This is usually done by measuring the model's performance on a separate dataset not involved in training, the so called test set X_{test}. Performance is measured according to the type of data and the kind of model involved. To visualize the performance of a classifier, a *confusion matrix* can be created. [Tha18, p. 2]

Prediction	Ground Truth	
	Positive	Negative
Positive	True Positive	False Positive
Negative	False Negative	True Negative

Figure 2: Illustration of a binary confusion matrix. 'Prediction' refers to predicted labels $y_{pred}(x)$ while 'Ground Truth' represents the actual labels $y(x)$.

The simplest case of a confusion matrix can be constructed when measuring the performance of a binary classifier. Figure 2 shows such a 2×2 binary confusion matrix. Here, 'Ground Truth' describes the label $y(x)$ of some data point $x \in X_{test}$, where $y \in \{0, 1\}$. 'Prediction' identifies the predicted labels $y_{pred}(x)$ that the model generates after is has been executed on the test-set X_{test} prior unknown.

Whenever $y_{pred}(x) = y(x),\ x \in X_{test}$ holds, the predicted label can be assigned to be either a *True Positive* (TP) or a *True Negative* (TN). The opposite holds as well, such that a falsely predicted label will be either a *False Negative* (FN) or a *False Positive* (FP). [Tha18, p. 2]

Using the classification introduced by the binary confusion matrix, all predicted labels y_{pred} are assigned to the four sets TP, TN, FN and FP. Using these four sets, we can introduce measures for classification performance.

Precision is a measure that depicts the proportion of correctly classified positive samples to the total amount of samples classified as positive.[Tha18, p. 4] This can be algebraically expressed as

$$Precision = \frac{|TP|}{|TP| + |FP|} \tag{14}$$

where $|A|$ is the cardinality of a set A. Precision measures how many elements classified as positive are True Positives.

Recall, also called *sensitivity*, represents the share of positive correctly classified samples to the total amount of positive samples.[Tha18, p. 3] This can be formalized as

$$Recall = \frac{|TP|}{|TP| + |FN|} \tag{15}$$

Recall measures how many of the positive labelled elements were actually selected.

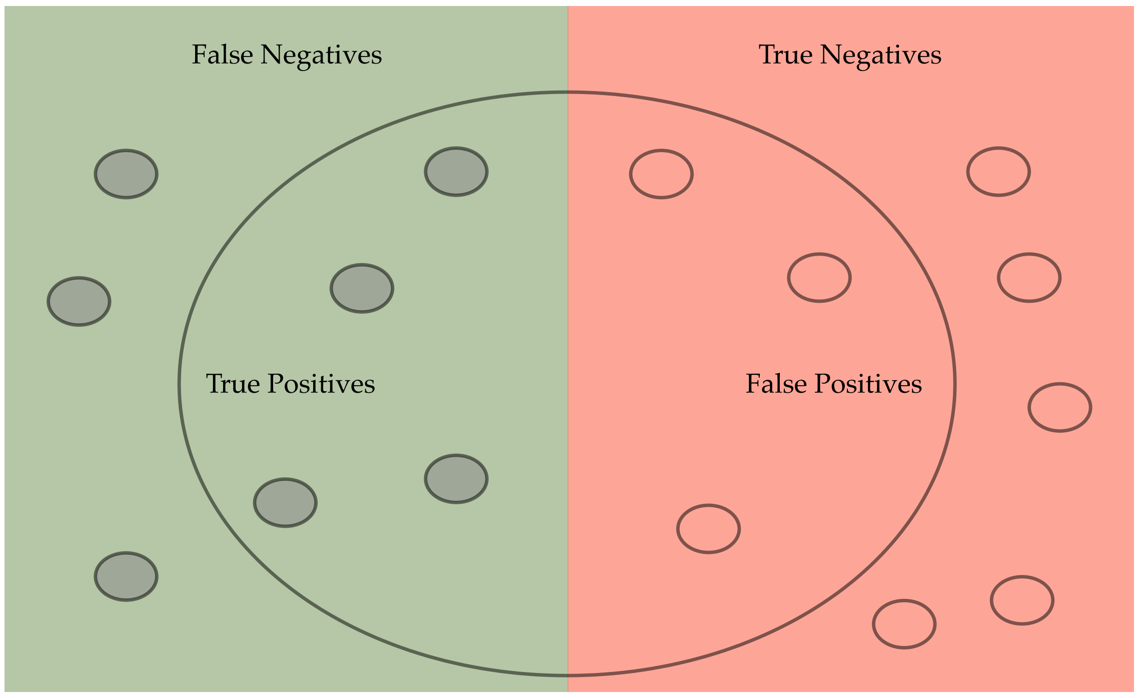


Figure 3: Each predicted label y_x is represented by a circle. Hollow circles stand for negative labels and full circles for positive lables.

The harmonic mean of precision and recall is called *F1-Score* or *F1-Measure*:

$$\text{F1-Score} = \left(\frac{Recall^{-1} + Precision^{-1}}{2}\right)^{-1} \tag{16}$$

The F1-Score was derived under the name of *MUC-4* by Nancy Chinchor in 1992. [Chi92]. Chinchor based her work on the book 'Information Retrieval' by Van Rijsbergen from 1979. [Rij79]

2.7.1 Datawig Imputer

Datawig Imputer is an imputation method based on *supervised learning*. [Bie+18, p. 1] It is used in this work for evaluating the robustness of FDs under the name 'ML Imputer'. In the DepDetector algorithm, the Datawig Imputer is used to train models for dependency detection.

The way Datawig Imputer interprets inputs, observations are a set of tuples $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots\}$. [SV08, p. 10] We call $\boldsymbol{x}_i \in \mathbb{F}$ *feature-vector* in a *feature-space* $\mathbb{F} \subseteq \mathbb{R}^n$ and $y_i \in S$ *output-attributes* or *labels*. [DHS00, p. 7]

Datawig Imputer solves a classification problem. A function

$$f : \mathbb{F} \to S$$

is being approximated, where $\mathbb{F}$ is the feature-space and $S = \{a_1, a_2, \dots, a_M\}$ is a set of labels.

Datawig Imputer assumes that the column to be imputed, the so-called *label comumn*, contains data that can be transformed to obtain attributes $a_1, \dots, a_M$. [Bie+18, p. 2018] All columns on a database table that are *not* the label column are called *feature columns*. These feature columns are transformed by Datawig Imputer to obtain feature-vectors.

The transformation performed to obtain learnable data assign to the `string` data of each column c a numerical representation x^c. [Bie+18, p. 2020] These functions are called *encoders*. Datawig Imputer separates data into two categories: *categorical data* and *sequential data*. [Bie+18, p. 2017] Different encoders are chosen to transform *categorical data* and *sequential data*.

Categorical data are data that consist of *categorical variables*. "A categorical variable places a case into one of serveral groups of categories," define Moore et al.[Moo+11, p. 4] An example-column c_{cat} containing categorical variables can be representend by the following set of colors:

$$c_{cat} = \{\texttt{blue, red, yellow, blue, blue, red}\} \tag{17}$$

Datawig Imputer creates a histogram of column c and uses the histogram's index $x^c \in \{1, 2, \dots, M_c\}$ to generate a numerical representation. Thus the encoded column c_{cat} would have the following form:

$$c^c_{cat} = \{0, 1, 2, 0, 0, 1\}$$

Sequential data is data where the sequece in which individual datapoints are stored in contains information. An example for sequential data would be a column containing non-categorical strings, like Usernames:

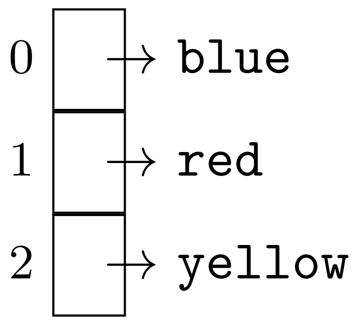


Figure 4: State diagram showing the histogram established by Datawig Imputer to encode example 17

$$c_{seq} = \{\texttt{ItalyPaleAle},\ \texttt{sisou},\ \texttt{primos63}\}$$

The numerical representation $c^c_{seq} \in \{0, 1, 2, \dots, A_c\}^{S_c}$ of the sequential column "is a vector of length S_c, where S_c denotes the length of the sequence or `string` in column c_{seq}." [Bie+18, p. 2020] This numerical representation is called *n-gram representation*. For further details on the n-gram implementation, consider [Bie+18, p. 2020].

Finally, the Datawig-Imputer learns to predict the label distribution from $y \in 1, 2, \dots, D_y$ from the feature-vector $\boldsymbol{x}$. It therefore models $p(y|\boldsymbol{x}, \boldsymbol{\Theta})$, the D_y - dimensional probability vector over all possible values in the to-be imputed column in function of feature-vector $\boldsymbol{x}$ and *learned model parameters* $\boldsymbol{\Theta}$. [Bie+18, p. 2021]

Datawig uses a logistic regression type output layer to achieve this:

$$p(y|\boldsymbol{x}, \boldsymbol{\Theta}) = \texttt{softmax}[\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}]$$

Here, the learned model parameters $\boldsymbol{\Theta} = (\boldsymbol{W}, \boldsymbol{z}, \boldsymbol{b})$ depends on the learned *weights* $\boldsymbol{W}$ and *biases* $\boldsymbol{b}$ as well as $\boldsymbol{z}$, representing all parameters of the column-specific feature extraction. Parameters $\boldsymbol{\Theta}$ are learned by minimizing cross-entropy loss between predicted and observed labels y by computing

$$\boldsymbol{\Theta} = \min_{\Theta} \textstyle\sum_1^N -\log\left(p\left(y|\boldsymbol{x}, \boldsymbol{\Theta}\right)\right)^\top \texttt{onehot}(y). \tag{18}$$

Here, $\log()$ represents the element-wise logarithm. $\texttt{onehot}(y) \in 0, 1^{D_y}$ stands for a one-hot encoding of one label y.

Equation 18 is applying the principle of *Empirical Risk Minimization*. [Vap92, p. 832] The right-hand term is the *empirical risk-function* for the classification problem. By minimizing the risk, in this case cross-entropy loss, weights $\boldsymbol{\Theta}$ are 'learned' that lead to an optimized classification performance.

Training is done using standard backpropagation and stochastic gradient-descent on mini-batches. The exact network layout is described in the paper "Deep Learning for

Missing Value imputation in Tables with Non-Numerical Data” [Bie+18, p. 2022] in full detail.

2.7.2 ML Imputer

When Datawig Imputer is employed in the experimental section of this work, it is referred to as ‘ML Imputer’. This name is chosen to emphasize the internal mode of operation similar to FD Imputer. Both imputers use cross-validation to find impute-values on a test set. They derive either a F1-Score as in equation 12 for classifiable data or a MSE as in equation 13.

2.8 DepDetector: RFD Discovery

In the field of data profiling, an extensive body of theory and algorithms for FD discovery has been created in the past decades. [Abe+19, p. 39] New techniques such as massive parallelization or hybrid algorithms, combining different discovery strategies, lowered the time needed to detect FDs on a relational table continuously. [Abe+19, p. 40]

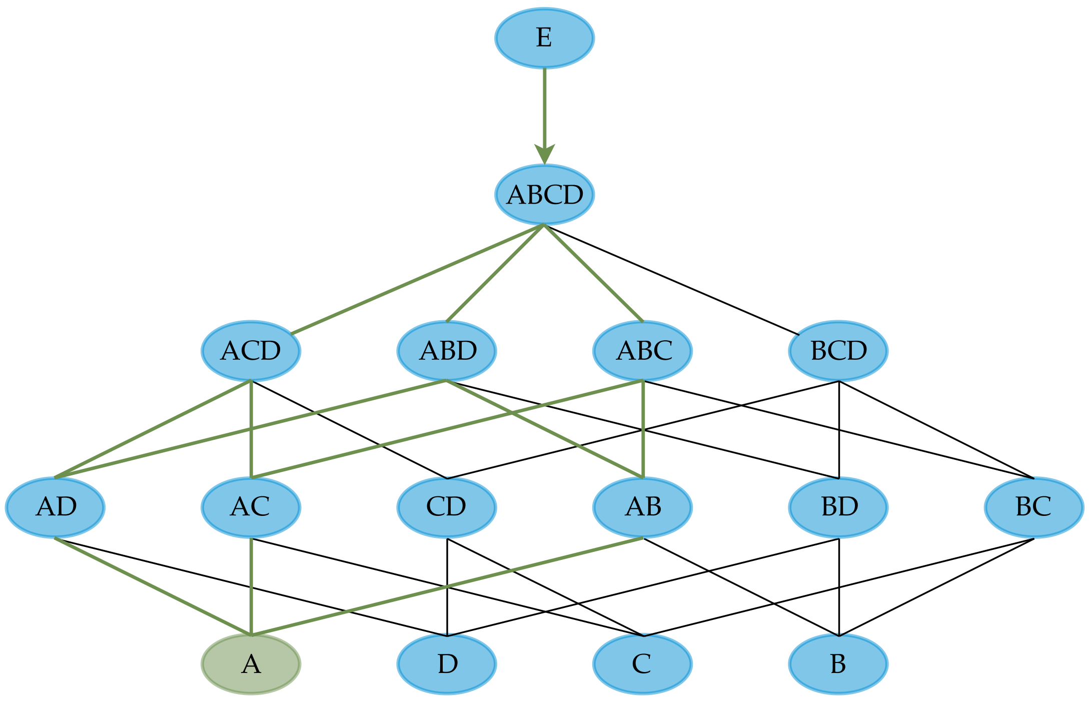


Figure 5: Hasse diagram of the search lattice when discovering minimal, non-trivial FDs on a relational schema with five columns. Column E is the RHS for which a minimal LHS is searched. Green edges show the candidate-LHSs on differnt paths to the minimal LHS A.

Notably the TANE algorithm developed by Huhtala et al. discovered FDs faster than any other algorithm at the time. [Huh+99] It paved the way for other algorithms that use the same search strategy called 'lattice traversal', such as FUN [NC08], FD_MINE [YHB02] and DFD [ASN14]. [Abe+19, p. 39]

In an effort to leverage the imputation-capabilities of ML Imputer for dependency discovery, the algorithm called *DepDetector* is introduced. DepDetector finds RFDs that relax both on the attribute comparison as well as on the extent. When discovering dependencies, DepDetector searches all non-trivial, minimal dependencies. Similar to FD discovery algorithms, the search space can be modeled as a graph labeling problem. [Abe+19, p. 24] Here, every possible combination of attributes, called *power set* of attribute combinations, from the nodes of the graph. Due to the properties of the graph, the problem can be modelled as a *lattice*. [Abe+19, p. 24] This has been visualized using a Hasse diagram in figure 5.

2.8.1 DepDetector Algorithm

DepDetector searches for dependencies by traversing the search lattice as depicted in figure 5. In a first step, ML Imputer is run, imputing the RHS with all other columns on the relational scheme as LHS. This yields the model's robustness. For the algorithm to continue traversing the search lattice in search of dependencies, this model's robustness must exceed an initial threshold value. The threshold value depends on the data type of the potential RHS. The model is required to either achieve an F1-Score equal or greater than 0.9 or an MSE of less than $0.2 \cdot \overline{y}$, where $\overline{y}$ is the arithmetic mean of all RHS values in the dataset. If the initial model does not exceed the threshold, we can assume that no dependency is present on the relational instance.

However, if the initial threshold is surpassed, the search-algorithm traverses the search lattice in a 'top-down' manner as depicted in figure 5. For each node in the search lattice a model is trained and evaluated. Again the search is only proceeded, when the model surpasses a threshold: either a smaller MSE than the parent-node, or a F1-Score bigger than 98% of the parent node's F1-Score. This is repeated until the threshold is not exceeded by any new model or until a minimal LHS is reached.

DepDetector Relaxing on the Extent The threshold-driven search-strategy is effectively a relaxation on the extent of the canonical FD when a potential RHS is of a classifiable data type. It is similar to an AFDs approach of "holding on almost every tuple" — the imputation needs to be correct on almost every instance, but not on every instance. In addition, ML Imputer is capable of learning conditions, similiar to CFDs. As shown in the experimental section of this work, ML Imputer learns the order of entries in a relational instance as well as thresholds, such as a CFD^p would do. Compared to CFDs, ECFDs and CFD^{p}s, it is important to keep in mind though that these conditions are never manually set but learned.

The behavior of DepDetector as described in the paragraph above is also displayed in listing 1. Note that DepDetector is implemented with two search-modes called 'greeedy' and 'complete'. The 'complete' search-mode behaves as described above. It is optimized to find all dependencies on a relational scheme. Meanwhile the 'greedy' search-mode is optimized to detect just *one* dependency. This approach converges faster than 'complete' and can be used if a single minimal dependency is sufficient. However, it neither guarantees to find minimal dependencies, nor does it guarantee to find the most robust dependency.

```python
def get_complete_candidates(root):
    convergence = True
    most_recent_nodes = root.get_newest_children()
    for node in most_recent_nodes:
        if root.is_continuous:  # sequential data
            if (node.score < node.parent.score) and (len(node.name) > 1):
                convergence = False
                pot_lhs = node.name
                for col in pot_lhs:
                    candidate_lhs = [c for c in pot_lhs if c != col]
                    add_node(candidate_lhs,
                             parent=node,
                             score=None)

        elif not root.is_continuous:  # classifiable data
            if (node.score > node.parent.score*0.98) \
                    and (len(node.name) > 1):
                convergence = False
                pot_lhs = node.name
                for col in pot_lhs:
                    candidate_lhs = [c for c in pot_lhs if c != col]
                    add_node(candidate_lhs,
                             parent=node,
                             score=None)
```

Listing 1: 'Complete' candidate generation in the DepDetector algorithm

DepDetector Relaxing on Attribute Comparison A fundamental difference that manifests itself when comparing dependencies discovered by DepDetector with MFDs or FDs is due the fact that DepDetector does not derive the existence of a dependency based on attribute comparison as in equations 1 or 6. Instead, DepDetector uses ERM-strategies when executing ML Imputer to measure a dependency's robustness. Thus, DepDetector optimizes for maximum robustness and a minimum number of LHS attributes.

3 Experiments

To examine robustness as it is defined in the previous chapter, a number of experiments are conducted. In the following subsection, in order to evaluate the capabilities of empirical risk minimization (ERM) techniques for FD discovery, the DepDetector algorithm is run on a number of datasets. For the experiments, datasets from the UCI Machine Learning Repository are used. [DG17].

Since the algorithms analyzed in this section handle missing values differently, rows containing missing values are excluded from the datasets due to possible inconsistencies when comparing results.

3.1 FD Imputer

FD Imputer is run for every FD found on a train subset of a dataset. Figure 6 shows the performance of FD Imputer on the Adult dataset. The analyzed columns contain classifiable data, thus the F1-Score represents robustness. The two top performing FDs have a perfect F1-Score of 1 each. An explanation for this circumstance can be found when analyzing the content of columns 4 and 5.

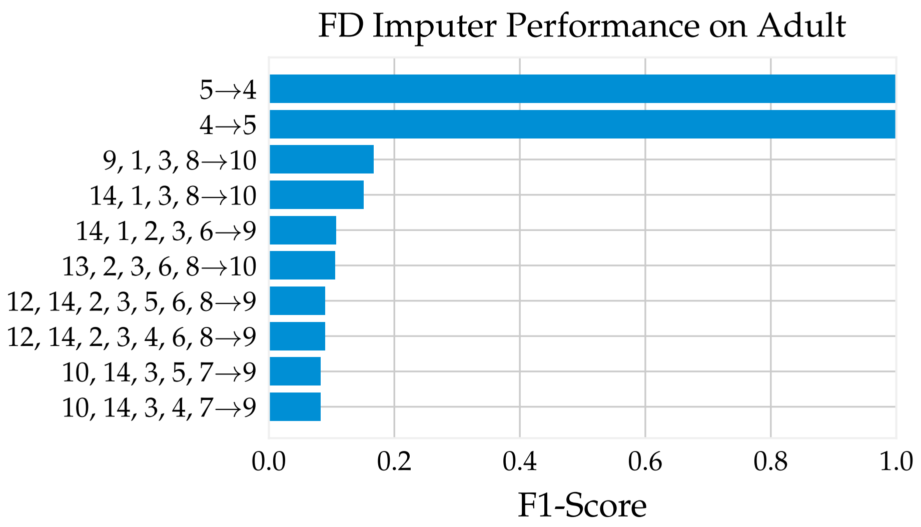


Figure 6: F1-Scores of the 10 most-robust FDs on the Adult dataset when imputing classifiable values.

Column 4 contains information about the highest educational level achieved. There are 16 different categories of educational level defined. Each category is assigned an integer in a range from 0 to 15. This integer is the content of column 5. Thus, the relation between column 4 and column 5 can be modeled by a bijective function, projecting the domains of each attribute onto the other.

All other FDs found on the Adult dataset lead to F1-Scores smaller than 0.2, being substantially less robust than the two top-performing FDs. It can be derived that only the two top-performing FDs can be used to meaningfully impute data. If new data was added to the dataset, it can thus safely be assumed that these two FDs still held. In reverse, no value in a column other than 4 or 5 can be meaningfully imputed using FDs detected on the Adult dataset due to the small robustness of these FDs.

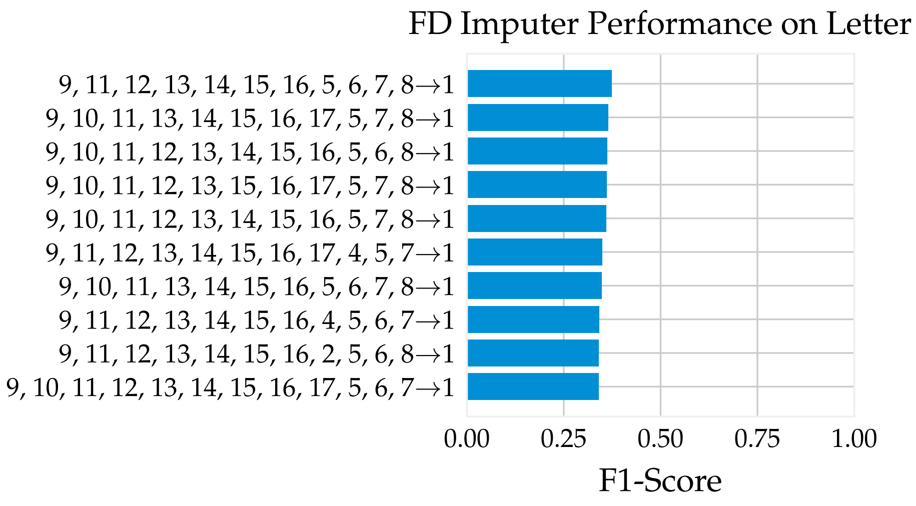


Figure 7: Robustness of the 10 top-performing FDs on the Letter dataset when imputing classifiable RHSs.

Figure 7 displays the result of running FD Imputer on the Letter dataset. From the 80 FDs found on the train split, the 10 most robust ones all share the same RHS, namely '1'. At the same time, the LHSs of those top-performing FDs show minimal destinctions. While they are, by the definition of the canonical FD, distinct minimal dependencies, they have similar robustness.

All datasets other than Adult show no FD with a perfect score. Only a FD on the Breast Cancer Wisconsit dataset achieves a score higher than 0.75 Table 5 provides a summary of how FD Imputer performs on a range of datasets.

Column #FDs indicates how many FDs were found on the complete dataset. #FDS$_{\text{train}}$ contains the number of FDs that were detected on the train subset used for imputations. F1$_{\text{mean}}$ and F1$_{\text{max}}$ indicate the arithmetic mean and maximal F1 score achieved on each dataset respectively. The last column named #(F1 = 0) provides the number of FDs that scored a F1 score of 0.

The results displayed in table 5 show how strongly the FD Imputer's performance depends on the dataset considered. On a dataset as normalized as nursery, FD Imputer cannot leverage FDs to impute unknown entries.

			Classification Performance		
Dataset	#FDs	#FDs$_{\text{train}}$	#FD (F1 = 0)	F1$_{\text{mean}}$	F1$_{\text{max}}$
Abalone	175	193	45	0.0008	0.0048
Adult	93	88	10	0.0669	1.0000
Balance Scale	7	7	6	0.0000	0.0000
Breast Cancer Wisconsin	57	77	10	0.2198	0.7539
Chess	9	9	8	0.0000	0.0000
Iris	9	8	1	0.1274	0.2252
Letter	78	80	17	0.2347	0.3737
Nursery	11	11	10	0.0000	0.0000

Table 5: Performance of the FD Imputer on a selection of UCI datasets.

3.2 Datawig Imputer

Datawig Imputer is run with the same set of FDs as FD Imputer. The 10 best performing FDs' F1 scores are displayed in figure 8. The two top scoring FDs are the same for FD

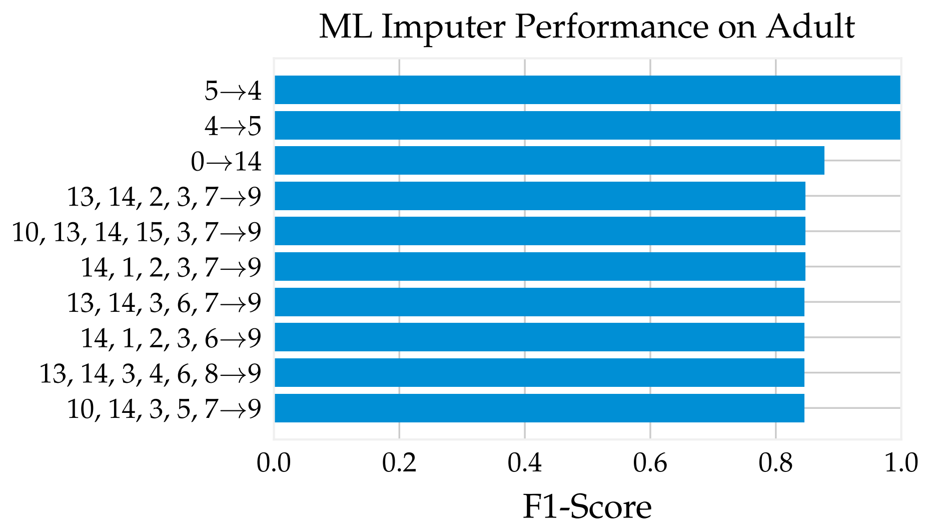


Figure 8: The figure compares the f1-score of the FD Imputer compared to the f1-score of the Datawig Imputer. Each point represents one FD.

Imputer and Datawig Imputer. The third most performant FD is a relation between row ID and nationality. Since there is no appearent dependency between row ID and nationality, it is safe to assume that Datawig Imputer always guesses the most frequent nationality, US-American, for no matter what row ID.

This shows that Datawig Imputer is capable of learning Conditional Dependencies.

			Classification Performance			
Dataset	#FDs	#FDs$_{\text{train}}$	#(F1 = 0)	F1$_{\text{mean}}$	F1$_{\text{max}}$	F1$_{\text{min}}$
Abalone	175	193	0	0.3697	0.5664	0.0619
Adult	93	88	0	0.7720	1.0000	0.0409
Balance Scale	7	7	0	0.4670	0.9443	0.0255
Chess	9	9	0	0.3744	0.9276	0.1162
Iris	9	8	0	0.9243	1.0000	0.1275
Letter	78	80	0	0.7207	0.9134	0.0058
Nursery	11	11	0	0.9915	0.4531	0.1138

Table 6: Performance of the Datawig Imputer.

Table 6 shows a generally higher performance of Datawig Imputer compared to FD Imputer. There are no FDs for which Datawig Imputer scores 0. This can be explained by to Datawig Imputer's behavior of always returning some imputation value, even if certainty is very low. This is contrasted by FD Imputer, which frequently returns no imputation value at all.

3.3 Overfitting the Datawig Imputer

3.4 Comparing Datawig Imputer with FD Imputer

As discussed in the previous section, Datawig Imputer and FD imputer differ fundamentally in the way they function. When comparing the two, metrics need to be computed bearing those differences in mind.

FD Imputer cannot approximate numerical values. Due to the nature of the definition of a FD, Data is always assumed to be classifiable. Meanwhile, Datawig Imputer is able to perform regression, predicting a continuous label for a given input with an uncertainty.

Naturally, this circumstance leads to a far superior performance of Datawig Imputer when imputing continuous labels. FD Imputer usually does not find any values on the train set to impute with and cannot return a meaningful result. Taking the above into account, rows that aren't imputed by FD Imputer are not considered when computing a performance measure on columns containing continuous values.

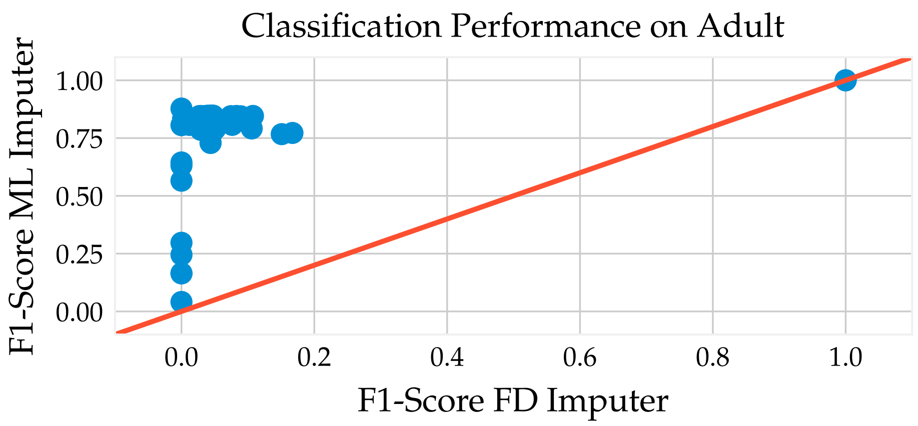


Figure 9: The figure compares the f1-score of the FD Imputer compared to the f1-score of the Datawig Imputer. Each point represents one FD.

Figure 9 compares the F1-scores of both Datawig Imputer and FD Imputer on the Adult dataset. One can observe that for almost all FDs, the Datawig Imputer performs better than the FD Imputer.

FD Imputer performance and Datawig Imputer performance seem to be proportional. If the ML imputers F1-score is lower than 0.7, the FD Imputers F1-score for the same FD is 0. However, for FDs where the Datawig Imputer scores are larger than 0.7, the FD Imputer scores better than 0.0.

There are two FDs for which the FD Imputer performs equally good as the Datawig Imputer. These FDs were identified in the previous section.

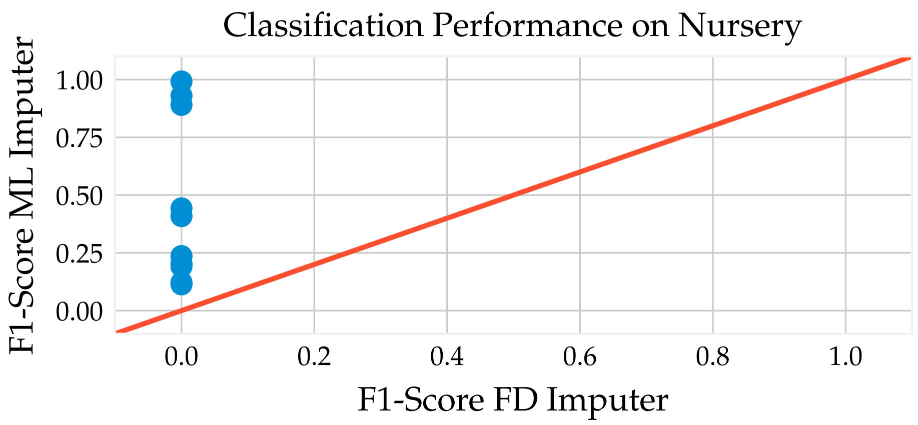


Figure 10: The FD imputers classification performance compared to the performance of the Datawig Imputer on the nursery dataset.

The same experiment was run on the Nursery dataset.[DG17] Results are displayed in figure 10. On this dataset, the ML Imputer performs better for every single FD. This is explainable by the fact that the Nursery dataset is strongly normalized. It contains only 11 FDs, most of them are the key, functionally determining a single column.

3.5 Dependency detection

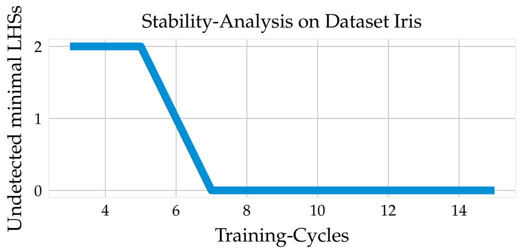
Using the approach described in the theory section to detect a generalized imputation dependency, DepDetector is run on number of datasets. In addition, the stability of discovered minimal dependencies is investigated.

3.5.1 Dependency stability

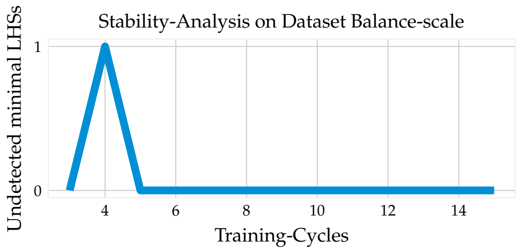
Stability of the found minimal dependency depends on the number of cycles used for training the classifier. DepDetector is run with different numbers of maximal training cycles. Minimal dependencies were searched following the 'complete' search strategy.

The number of training cycles τ is increased stepwise from 3 to 15. The number of undetected minimal LHSs is calculated by selecting the result of the run with $\tau = 15$ training cycles as a reference. For every time DepDetector is run to find minimal dependencies, the obtained LHSs are compared to the ones obtained with $\tau = 15$. For each LHS found when $\tau = 15$ that is not contained in the result of a run, the number of undetected minimal LHSs is increased by one.

Figure 11a shows that the Iris dataset contains two minimal LHS combinations and thus two Dependencies. For $\tau \in [7, 15]$ training cycles, DepDetector discovered all minimal dependencies.



(a) Analysis on the Iris dataset reveals that results become continuously more stable for larger τ.



(b) On the dataset Balance-scale, the analysis of the result shows a peak at $\tau = 4$.

Figure 11: Stability analysis of the minimal LHSs found by DepDetector.

An analysis on the dataset Balance-scale shows the potentially non-linear behavior of obtained results in function of τ: Even though the result obtained for $\tau = 3$ is minimal, the result found when $\tau = 4$ is in fact not minimal.

For further analysis, DepDetector models are trained with $\tau = 10$ training cycles, aiming to detect dependencies that are stable.

3.5.2 Results on various Datasets

Dependencies were detected on a number of well-known datasets.

					Greedy	Complete
Dataset	Cols	Rows	# FDs	# $\text{FDs}_{\text{train}}$	# $\text{F1}_{\text{LHS}} > 0.90$	# $\text{F1}_{\text{LHS}} > 0.90$
adult	16	32561	93	88	99 (100s)	99
nursery	11	11	99	99	99	99
abalone	10	4177	99	99	99	99
balance-scale	6	625	99	99	99	99
chess	8	28056	99	99	99	99
iris	6	150	99	99	99	99
letter	18	20000	99	99	99	99

Table 7: Result of running dependency detection on selected datasets. Values in brackets are the respective algorithm's runtime in seconds.

4 Discussion

The experiments conducted in the previous section explored the characteristics of FDs. When applied for database scheme normalization, FDs serve a well-defined purpose. However, when it comes to obtaining insights about non-static data, FDs seem to provide little insights about what to expect from new rows that might be added in the future.

FDs do not seem fit for obtaining human-readable information about a (relational) database that is being used in a deployed application. FDs show no resistance to noisy data due to the static nature of their definition. In the field of data profiling, it was proposed that "any dependency can be turned into a rule to check for errors in the data".[Abe+19, p. 9] This does not seem to be the case in general, but only in a noise-free, static environment.

References

[Abe+19] Ziawasch Abedjan et al. *Data Profiling*. 2019. ISBN: 9781681734477. DOI: https://doi.org/10.2200/S00878ED1V01Y201810DTM052.

[ASN14] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. "DFD: Efficient Functional Dependency Discovery". In: *CIKM*. 2014.

[Bie+18] Felix Biessmann et al. ""Deep" Learning for Missing Value Imputation in Tables with Non-Numerical Data". In: *ACM International Conference on Information and Knowledge Management* 27 (2018), p. 9. DOI: https://doi.org/10.1145/3269206.3272005.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Ble+16] Tobias Bleifuß et al. "Approximate Discovery of Functional Dependencies for Large Datasets". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: ACM, 2016, pp. 1803–1812. ISBN: 978-1-4503-4073-1. DOI: 10.1145/2983323.2983781. URL: http://doi.acm.org/10.1145/2983323.2983781.

[Bra+08] Loreto Bravo et al. "Increasing the Expressivity of Conditional Functional Dependencies without Extra Complexity". In: *Proceedings - International Conference on Data Engineering* (May 2008), pp. 516–525. DOI: 10.1109/ICDE.2008.4497460.

[Bur19] Andriy Burkov. *The Hundered-Page Machine Learning Book*. Andriy Burkov, 2019. ISBN: 978-1999579500.

[CDP16] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. "Relaxed Functional Dependencies - A Survey of Aproaches". In: *IEEE Transactions on Knowledge and Data Engineering* (2016), pp. 147–165.

[CFM09] Wenguang Chen, Wenfei Fan, and Shuai Ma. "Analyses and Validation of Conditional Dependencies with Built-in Predicates". In: (2009), pp. 576–591.

[Chi92] Nancy Chinchor. "MUC-4 Evaluation Metrics". In: *Proceedings of the 4th Conference on Message Understanding*. MUC4 '92. McLean, Virginia: Association for Computational Linguistics, 1992, pp. 22–29. ISBN: 1-55860-273-9. DOI: 10.3115/1072064.1072067. URL: https://doi.org/10.3115/1072064.1072067.

[Cod70] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[Cod90] Edgar F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Publishing Company, 1990. ISBN: 0-201-14192-2. URL: https://dl.acm.org/citation.cfm?id=77708.

[DG17] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience New York, 2000. ISBN: 0471056693.

[Hay08] Simon Haykin. *Neural Networks and Learning Machines Third Edition*. Pearson Prentice Hall, 2008. ISBN: 9780131471399.

[Hei+19] Alireza Heidari et al. “HoloDetect: Few-Shot Learning for Error Detection”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. ACM, 2019, pp. 829–846. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3319888. URL: http://doi.acm.org/10.1145/3299869.3319888.

[Huh+99] Ykä Huhtala et al. “TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies”. In: *The Computer Journal* 42 (1999), pp. 100–111.

[IBM03] Research News IBM. “Former IBM Fellow Edgar (Ted) Codd passed away on April 18”. In: (Apr. 23, 2003). URL: https://web.archive.org/web/20190425094107/https://www.ibm.com/ibm/history/exhibits/builders/builders_codd.html (visited on 06/26/2019).

[Kle11] Stephan Kleuker. *Grudkurs Datenbankentwicklung*. Vieweg+Teubner Verlag, 2011. ISBN: 978-3-8348-9925-5. URL: https://link.springer.com/book/10.1007/978-3-8348-9925-5.

[Kou+09] N. Koudas et al. “Metric Functional Dependencies”. In: (Mar. 2009), pp. 1275–1278. ISSN: 1063-6382. DOI: 10.1109/ICDE.2009.219.

[Kra+18] Tim Kraska et al. “The Case for Learned Index Structures”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: ACM, 2018, pp. 489–504. ISBN: 978-1-4503-4703-7. DOI: 10.1145/3183713.3196909. URL: http://doi.acm.org/10.1145/3183713.3196909.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Pr, 1983. ISBN: 0914894420. URL: http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html.

[Moo+11] David S. Moore et al. *The Practice of Statistics for Business and Economics*. W. H. Freeman and Company, 2011.

[NC08] Noel Novelli and Rosine Cicchetti. “FUN: An Efficient Algorithm for Mining Functional and Embedded Dependencies”. In: vol. 1973. Jan. 2008, pp. 189–203. DOI: 10.1007/3-540-44503-X_13.

[Pap+15] Thorsten Papenbrock et al. “Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms”. In: *Proc. VLDB Endow.* 8.10 (2015), pp. 1082–1093. ISSN: 2150-8097. DOI: 10.14778/2794367.2794377. URL: https://doi.org/10.14778/2794367.2794377.

[PN16] Thorsten Papenbrock and Felix Naumann. “A Hybrid Approach to Functional Dependency Discovery”. In: SIGMOD ’16 (2016), pp. 821–833. DOI: 10.1145/2882903.2915203. URL: http://doi.acm.org/10.1145/2882903.2915203.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval*. 2nd. Newton, MA, USA: Butterworth-Heinemann, 1979. ISBN: 0408709294.

[Sch17] Edwin Schicker. *Datenbanken und SQL*. Springer Vieweg, 2017.

[Stu16] Thomas Studer. *Relationale Datenbanken*. Springer Vieweg, 2016.

[SV08] Alex Smola and S.V.N. Vishwanathan. *Introduction to Machine Learning*. Cambrisdge University Press, 2008. ISBN: 521825830. URL: svn://smola@repos.stat.purdue.edu/thebook/trunk/Book/thebook.tex.

[Tha18] Alaa Tharwat. "Classification assessment methods". In: *Applied Computing and Informatics* (2018). ISSN: 2210-8327. DOI: https://doi.org/10.1016/j.aci.2018.08.003. URL: http://www.sciencedirect.com/science/article/pii/S2210832718301546.

[Vap92] V. Vapnik. "Principles of Risk Minimization for Learning Theory". In: *Advances in Neural Information Processing Systems 4*. Ed. by J. E. Moody, S. J. Hanson, and R. P. Lippmann. Morgan-Kaufmann, 1992, pp. 831–838. URL: http://papers.nips.cc/paper/506-principles-of-risk-minimization-for-learning-theory.pdf.

[Wat14] Adrienne Watt. *Database Design*. BCcampus, 2014. URL: https://opentextbc.ca/dbdesign01/.

[YHB02] Hong Yao, Howard Hamilton, and Cory Butz. "FD_Mine: Discovering Functional Dependencies in a Database Using Equivalences." In: Jan. 2002, pp. 729–732. DOI: 10.1109/ICDM.2002.1184040.

5 Appendix

```python
def get_greedy_candidates(root):
    convergence = True
    threshold = root.most_recent_nodes()[0].parent.score
    highscore_node = None

    for node in most_recent_nodes():
        if root.is_continuous:  # sequential RHS
            if (node.score <= threshold) and (len(node.name) > 1):
                highscore_node = node
        elif not root.is_continuous:  # classifiable RHS
            if (node.score >= highscore) and (len(node.name) > 1):
                highscore_node = node

    if highscore_node is not None:
        convergence = False
        pot_lhs = highscore_node.name
        for col in pot_lhs:
            candidate_lhs = [c for c in pot_lhs if c != col]
            add_node(candidate_lhs,
                     parent=highscore_node,
                     score=None)
```

Listing 2: 'Greedy' Candidate Generation in DepDetector