Learning How to Correct a Knowledge Base from the Edit History
Thomas Tanon, Camille Bourgaux, Fabian Suchanek

To cite this version:
Thomas Tanon, Camille Bourgaux, Fabian Suchanek. Learning How to Correct a Knowledge Base from the Edit History. World Wide Web Conference, May 2019, San Francisco, United States. 10.1145/3308558.331358. hal-02066041

HAL Id: hal-02066041
https://hal-imt.archives-ouvertes.fr/hal-02066041
Submitted on 13 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
Learning How to Correct a Knowledge Base from the Edit History

Thomas Pellissier Tanon
Télécom ParisTech
ttanon@enst.fr

Camille Bourgaux
DI ENS, CNRS, ENS, PSL Univ. & Inria
camille.bourgaux@ens.fr

Fabian Suchanek
Télécom ParisTech
suchanek@enst.fr

ABSTRACT
The curation of a knowledge base is a crucial but costly task. In this work, we propose to take advantage of the edit history of the knowledge base in order to learn how to correct constraint violations. Our method is based on rule mining, and uses the edits that solved some violations in the past to infer how to solve similar violations in the present. The experimental evaluation of our method on Wikidata shows significant improvements over baselines.

KEYWORDS
knowledge base; history; data cleaning; rule mining; Wikidata

1 INTRODUCTION
Knowledge bases (KBs) play a key role in many applications and lay at the core of the Semantic Web. They contain entities (such as persons, cities, etc.) and statements about them (such as relationships between persons, places of residence, etc.). In this work, we focus on RDF-style KBs, such as Wikidata [38], YAGO [35] or DBpedia [7]. The data quality of a KB is crucial for its usability. However, it is usually very costly to check the correctness of the data, because KBs can be huge (Wikidata, e.g., contains about 50 millions entities). Moreover, KBs are often built using methods that are error-prone. For instance YAGO and DBpedia are automatically extracted from Wikipedia. Wikidata, for its part, is a collaborative KB that anyone can edit, with more than 18,000 active contributors.

A way of avoiding or at least detecting some of the flaws in the data is to impose constraints on the KB. Such constraints can enforce that some information must be present (e.g., implying that every human being has a birth date), or that some statements may not occur (e.g., ensuring that a person is not also a city). Constraints are related to, but different from, ontological rules: A constraint imposes a certain condition, whereas an ontological rule infers certain statements. For example, consider a KB that contains the statement "Spinoza is a human being" without knowing any birth date for Spinoza, and consider a rule "All human beings have a birth date". If the rule is taken as an ontological rule, then it would just infer that Spinoza has some birth date. If the rule is taken as a constraint, in contrast, the KB would be considered incorrect. Constraints are thus similar in spirit to database integrity constraints. In practice, constraints often have exceptions. Therefore, it is useful to allow data that does not respect them (in Wikidata, e.g., constraint violations are simply flagged). Nonetheless, by design, most of the constraint violations are not exceptions but actual errors and proposing to repair them is a good starting point when it comes to improving KB quality.

In this paper, we aim at learning how to repair constraint violations. Our goal is to help a KB editor by suggesting how to clean the data locally (providing a solution to a particular constraint violation) or globally (providing rules that can be automatically applied to all constraint violations of a given form once validated by the editor). To do that, we take advantage of the edit history of the KB. We use it to mine correction rules that express how different kinds of constraint violations are usually solved. To the best of our knowledge, this is the first work that builds on past users’ knowledge, this is the first work that builds on past users’ knowledge, this is the first work that builds on past users’ knowledge, this is the first work that builds on past users’ knowledge.

2 RELATED WORK
We start with a brief discussion of works relevant to our problem along three axes: constraints for KBs, KB cleaning, and rule learning.

Constraints. Constraints have long been used in databases and KBs to express rules that the data should follow. Databases typically operate under the closed world assumption, where missing facts are considered to be false. This allows for "completeness" constraints such as tuple generating dependencies. KBs, in contrast, operate under the open world assumption, where missing
facts are not necessarily false. They thus classically have only "correctness" constraints, such as disjointness or functionality axioms (corresponding to special cases of denial constraints and equality generating dependencies in databases).

To express also completeness constraints, several works propose to use description logics, with varying semantics [28, 37]. Another possibility is to use queries that should or should not hold as constraints (see e.g., [24] for methods for writing constraint queries in SPARQL). Other approaches define constraint languages to specify conditions for RDF graph [9] validation, such as SHACL [22] or ShEx [8]. It has been argued in [30] that description logics under the closed world assumption are also suitable for constraint checking in RDF, which can then be implemented with SPARQL queries. In our work, we follow a similar path, using description logic axioms as constraints for RDFS KBs, because it corresponds best to what we observe in current real-world KBs.

Contrary to the above works, we do not aim at expressing constraints, but at repairing their violations. The correction rules we learn for this purpose are similar in spirit to active integrity constraints [11], which specify for each constraint a set of possible repair actions. This type of constraints has recently been applied to description logic KBs as well [32]. Conditioned active integrity constraints add conditions for choosing among the possible actions, and we propose, in a similar spirit, to take into account the context of the constraint violation to correct it. Different from the existing work [11, 32], our goal is to mine correction rules automatically from the edit history of the KB.

Knowledge base cleaning. Several recent approaches have dealt with the interactive cleaning of KBs. The proposed methods detect when a constraint is violated, compute the responsible facts, and work [11, 32], our goal is to mine correction rules automatically.

In this work, we use association rules from a KB [33]. A more recent trend is to use embedding-based rule mining have also been used for induction of new ontological rules from a KB [33]. A more recent trend is to use embedding-based rule mining have also been used for induction of new ontological rules from a KB [33].

usual rule learning approaches is reported in [27] and significant recent works in this area include [19, 39, 40].

In this paper, we use a vanilla rule mining algorithm inspired by [14]. Our contribution is not the rule mining per se, but the application of rule mining to the edit history of a KB in order to mine correction rules. This avenue has, to the best of our knowledge, never been investigated.

3 PRELIMINARIES

In this work, we use description logics (DL) [4] as KB language and as constraint language, because they are the foundation of the Semantic Web standard OWL [16].

Syntax. We assume a set NC of concept names (unary predicates, also called classes), a set NR of role names (binary predicates, also called properties), and a set NI of individuals (also called constants). An ABox (dataset) is a set of concept or role assertions of the form A(a) or R(a, b), where A ∈ NC, R ∈ NR, a, b ∈ NI. A TBox (ontology) is a set of axioms whose form depends on the DL L in question, and expresses relationships between concepts and roles (e.g., concept or role hierarchies, role domains and ranges...). A knowledge base (KB) K = T ∪ A is the union of an ABox A and a TBox T.

In this work, we assume that T is a flat DL TBox [23], i.e., that L differs from the standard RDF Schema (RDFS) [17] only by allowing inverse roles in role inclusions. More precisely, T can contain concept inclusions of the form A1 ⊆ A2 (subclass), 3P ⊆ A (domain or range), and role inclusions P1 ⊆ P2 (subproperty), where A1 ∈ NC and P1 ∈ NI and P1(j) := R | −R with R ∈ NR.

A KB can also be written as a set of RDF triples (s, p, o) where s is the subject, p is the property, and o the object, using special properties to translate concept membership and relationships between concepts and roles [29]. A concept assertion A(a) is written as (a, rdf:type : A), and a role assertion R(a, b) as (a, R, b). Flat DL TBox axioms can also be represented by single triples. For example, A1 ⊆ A2 is written as ⟨A1, rdf:subClassOf : A2⟩ and ∃R ⊆ A is written as ⟨R, rdfs:range : A⟩.

Semantics. We recall the standard semantics of DL KBs. An interpretation has the form I = ⟨ΔI, I⟩, where ΔI is a non-empty set and I is a function that injectively maps each a ∈ NI to aI ∈ ΔI (unique name assumption), ⊑ to ΔI, each A ∈ NC to A1 ⊆ ΔI, and each R ∈ NR to R ⊆ ΔI × ΔI. The function I is straightforwardly extended to general concepts and roles, e.g. (¬B)I = ΔI \ B1I, (R∗)I = {(c, d) | (d, c) ∈ R I}, {a1, . . . , an} = {a1I, . . . , anI}, (∃P ∨ B)I = {c | ∃d ∈ P | d ∈ B I}, (B1 ⊔ B2)I = B1I ∪ B2I, (B1 ⊓ B2)I = B1I ∩ B2I. An interpretation I satisfies an inclusion A ⊆ B; if I satisfies an axiom (func) if I satisfies an axiom (trans) if I is transitive; and it satisfies A(a) (resp. R(a, b)), if aI ∈ A I (resp. (aI, bI) ∈ R I). We write I |= a if I satisfies the DL axiom a.

An interpretation I is a model of a KB K = T ∪ A if I satisfies all axioms in K. A KB K entails a DL axiom a if I |= a for every model I of K.

Queries. A conjunctive query (CQ) takes the form q(σ) = ∃yψ(σ, y), where ψ is a conjunction of atoms of the form A(t) or R(t, t′) or of equalities t = t′, where t, t′ are individual names or variables from
We denote by a (certain) union of CQs is a (certain)
∪®
The domain of I
P
B

which abbreviate is rewritten as B
of inclusions, and not at all in negative inclusions of the form

In this work, we consider two types of constraints: consistency constraints (which express that some state-
ments are contradictory), and completeness constraints (which im-
straints: consistency constraints (which says that everyone can have at most one birth date). If this
body

Example 2: In our running example, the KB K satisfies Γ1 since
∃hasMotherFK = {Zeus, Spinoza} and IΓK |= Person(Zeus) and IΓK |= Person(Spinoza). However, it violates Γ2 because
IΓK |= Person(Marques) ∨ Animal(Marques) while Marques ∈ ∃hasMotherFK. It violates Γ3 and Γ4 for similar reasons. Fi-

Not the semantic difference between the constraints and the ax-
ions of the TBox: The axiom (Human ⊆ Person) in the TBox makes
every human an answer to the query asking for persons. In contrast,
if we had put the axiom in the set of constraints, it would have required all human beings in the KB to be explicitly marked as persons.
As another example, consider the axiom (func hasBirthdate) (which says that everyone can have at most one birth date). If this axiom appears in the TBox, it renders the KB inconsistent whenever a person is given two distinct dates of birth. This has severe consequences on the reasoning capabilities, since everything is entailed from an inconsistent KB. If this axiom is in the set of constraints, in contrast, then distinct dates of birth lead only to the violation of the constraint. This gives us relevant information without having any impact on the usability of the KB.

Checking constraints. We show that our setting allows us to check constraint satisfaction via CQ answering. For this purpose, we use a function π, which maps each constraint Γ ∈ C to a rule of the form ∃ψ(¯x, ¯y) → ∃ψ′(¯x, ¯z). This function is defined recursively as shown in Table 1. The left side of the rule is called the body and its right side the head.

Example 3: In our running example, we obtain the following rules:
Γ0(x) : ∃hasGender(y, x) → x = male v x = female v x = nonbinary
Γ1(x) : ∃hasMother(x, y) → Person(x) ∨ Animal(x)
Γ2(x) : ∃hasMother(y, x) → Person(x) ∨ Animal(x)
Γ3(x) : ∃hasMother(y, x) → ∃hasGender(x, z) ∧ z = female
Γ4(x, y) : hasMother(x, y) → hasChild(y, x)

ABox A provides information on several individuals. Our constraints C state that there are three possible genders (consistency constraint), that those who have a mother or are a mother must be persons or animals, that a mother must have gender female, and that if a has mother b, then b must have child (completeness constraints).
The following proposition shows that this transformation is sound and that the rule body and head can be rewritten as CQ and UCQ.

**Proposition 1:** For every constraint \( \Gamma \in C, \pi(\Gamma) \) can be rewritten as a rule \( \Gamma(\pi) : b(\pi) \rightarrow h(\pi) \) where \( b(\pi) \) is a CQ, \( h(\pi) \) is a UCQ, and for every flat QL KB \( \mathcal{K} \), \( \mathcal{K} \) satisfies \( \Gamma \) iff answers\((h(\pi), \mathcal{K}) \subseteq \text{answers}(h(\pi), \mathcal{K}) \).

**Proof.** By our assumptions on the form of the concepts that occur in the left side of the inclusions or in the right side of a negative inclusion, \( b(\pi) := \exists \varphi(x, y) \) is a CQ. It is easy to show by structural induction that for every concept \( B(\text{resp. } P) \), \( \pi(B, x) \) (resp. \( \pi(P, x, y) \)) can be written as a UCQ \( q(x) \) (resp. \( q(x, y) \)) and that answers\((q(x), \mathcal{K}) = B^K \) (resp. \( \text{answers}(q(x, y), \mathcal{K}) = P^K \)). If \( \pi \) is a completeness constraint of the form \( B_1 \sqsubseteq B_2 \) (resp. \( P_1 \sqsubseteq P_2 \)), or a consistency constraint of the form \( B \sqsubseteq \{a_1, \ldots, a_n\} \), the result follows immediately since \( \mathcal{K} \) satisfies \( \Gamma \) iff \( B^K \sqsubseteq \Gamma \). If \( \pi \) is a consistency constraint of the form \( B_1 \sqsubseteq \neg B_2 \) (resp. \( P_1 \sqsubseteq \neg P_2 \)), answers\((b(\pi), \mathcal{K}) = B^K \cap B_2^K \) (resp. \( \text{answers}(b(\pi), \mathcal{K}) = P^K \cap P_2^K \)) and is empty iff \( \Gamma \) is satisfied. Since in this case \( h(\pi) := \text{false}, \text{answers}(h(\pi), \mathcal{K}) = \emptyset \), and the desired relation holds.

If \( \Gamma \) is of the form (func \( P \)), since the answers of \( h(x, y, z) := y = z \) over \( \mathcal{K} \) are all possible tuples of the form \( (a, b, b) \), \( P^K \) is functional if \( \text{answers}(b(\pi), \mathcal{K}) \subseteq \text{answers}(h(\pi), \mathcal{K}) \). Finally, if \( \Gamma \) is of the form (trans \( P \)), \( b(x, y, z) := \pi(P, x, y) \wedge \pi(P, y, z) \) and \( h(x, y, z) := \pi(P, x, z) \), and it is easy to see that \( P^K \) is transitive if \( \text{answers}(b(\pi), \mathcal{K}) \subseteq \text{answers}(h(\pi), \mathcal{K}) \).

**Constraint violations.** A constraint instance \( \Gamma(\alpha) \) of a constraint \( \Gamma(\pi) \) is obtained by replacing the variables \( \pi \) by the individual names \( \alpha \) in \( \Gamma(\pi) \). This notion allows us to define constraint violations:

**Definition 2 (Constraint violation):** A violation of a constraint \( \Gamma(\pi) \) in \( \mathcal{K} \) is a minimal subset \( V \subseteq \mathcal{K} \) such that there exists \( \alpha \) such that \( V \) violates \( \Gamma(\alpha) \) and \( \mathcal{K} \) violates \( \Gamma(\alpha) \). We denote by \( \text{Violations}(\mathcal{K}, \Gamma(\pi)) \) the set of violations of \( \Gamma(\pi) \).

In this definition, the requirement that \( \mathcal{K} \) violates \( \Gamma(\alpha) \) may seem superfluous. Yet, if \( \Gamma \) is a completeness constraint, it may be the case that some \( V \subseteq \mathcal{K} \) violates \( \Gamma(\alpha) \), while \( \mathcal{K} \) satisfies it.

**Example 4:** In our running example, it is easy to see that the subset \( V_0 = \{\text{hasGender}(\text{Zeus, masculine})\} \) is a violation of \( \Gamma_1 \). Consider now \( V = \{\text{hasMother}(\text{Spinoza, Marques})\} \). \( V \) is a violation of \( \Gamma_2, \Gamma_3 \) and \( \Gamma_4 \). Indeed, it violates \( \Gamma_2(\text{Marques}), \Gamma_3(\text{Marques}), \) and \( \Gamma_4(\text{Spinoza, Marques}) \) and \( \mathcal{K} \) does not satisfy any of these constraint instances. However, even if \( V \) violates \( \Gamma_1(\text{Spinoza}) \), \( V \) is not a violation of \( \Gamma_1 \) because \( \{\text{Human}(\text{Spinoza}), \text{Human} \sqsubseteq \text{Person}\} \subseteq \mathcal{K} \) satisfies the head of \( \Gamma_1(\text{Spinoza}) \).

If a constraint instance \( \Gamma(\alpha) \) is violated, \( \alpha \in \text{answers}(h(\pi), \mathcal{K}) \) and \( \alpha \notin \text{answers}(h(\pi), \mathcal{K}) \), so its violations are the minimal subsets of \( \mathcal{K} \) responsible for \( \alpha \in \text{answers}(h(\pi), \mathcal{K}) \). The next proposition relates constraint violations and justifications. A justification (also known as an explanation, axiom pinpointing, or MinAs) for the entailment of a BCQ is a minimal subset of the KB that entails the BCQ [21, 34].

**Proposition 2:** If \( \mathcal{K} \) violates \( \Gamma(\alpha) : b(\alpha) \rightarrow h(\alpha) \), a subset \( V \subseteq \mathcal{K} \) is a violation of \( \Gamma(\alpha) \) iff \( V \) is a justification of \( \mathcal{K} \models b(\alpha) \).

**5 CORRECTIONS**

We now turn to correcting constraint violations.

**Solutions.** We will make use of atomic modifications of the KB to define solutions to constraint violations.

**Definition 3 (Atomic modification):** An atomic modification of a KB \( \mathcal{K} \) is a pair \( m = (M^+, M^-) \) of two sets of assertions or L-axioms that takes one of the following forms:

- **Addition:** \( m = \{((s, p), o), (s, p) \notin \mathcal{K}\} \), \( (s, p, o) \notin \mathcal{K} \)
- **Deletion:** \( m = \emptyset, ((s, p), o) \notin \mathcal{K} \), \( (s, p, o) \notin \mathcal{K} \)

**Replacement:** \( m = \{(s, p, o), ((s', p', o'), (s', p', o')) \}, (s, p, o) \notin \mathcal{K} \), \( (s', p', o') \notin \mathcal{K} \), \( (s, p, o) \notin \mathcal{K} \)

A replacement is equivalent to a sequence of a deletion and an addition. We chose to keep it as an atomic modification because it corresponds to common knowledge base curation tasks, such as correcting an erroneous object for a given subject and predicate, or fixing a predicate misuse. Atomic modifications can be used to solve a constraint violation, as follows:

**Definition 4 (Solution):** A solution to a violation \( V \) of a constraint instance \( \Gamma(\alpha) \) in \( \mathcal{K} \) is an atomic modification \( (M^+, M^-) \) such that there exists \( \mathcal{K}' \subseteq \mathcal{K} \) such that \( \Gamma(\alpha) \models \mathcal{K}' \) and \( \mathcal{K}' \models V \). We call \( (M^+, M^-) \) a solution to \( V \) for \( \Gamma(\alpha) \) in \( \mathcal{K} \).

Note that \( \Gamma(\alpha) \) can still be violated in \( (\mathcal{K} \cup M^+) \setminus M^- \) if \( \mathcal{K} \) contains other violations of \( \Gamma(\alpha) \) for which \( M^+, M^- \) is not a solution. For example, if \( \Gamma(\alpha) : \exists x \text{hasGender}(x) \rightarrow A(a) \rightarrow \text{false} \), and \( \mathcal{K} = (R(a, b), R(a, c), A(a)) \), the deletion of \( R(a, b) \) is a solution to the violation \( (R(a, b), A(a)) \), but \( (R(a, c), A(a)) \) still violates \( \Gamma(\alpha) \). Note also that every constraint violation has at least one solution, which consists of the deletion of any of its elements. Solutions may also be additions or replacements, as in the following example:

**Example 5:** In our running example, the deletion \( (\emptyset, \text{hasGender}(\text{Zeus, masculine})) \) and the replacement \( (\text{hasGender}(\text{Zeus, male}), \text{hasGender}(\text{Zeus, male})) \)
We also prefer replacements over deletions as long as they fulfill the same condition. For instance, in our running example, the replacement hasGender(Zeus, masculine)) is better than the deletion (θ, hasMother(Spinoza, Marques)), because it corrects erroneous information instead of simply erasing it.

Algorithm 1 Construction of PCDataset

Input: set of constraints C, current TBox Tp, history (Kr)0≤i≤p
Output: set of relevant past corrections PCDataset

// Construct correction seed patterns
for all Γ in C such that Γ(®x): h(®x) → h(®x)
do
Patterns(Γ) := {((®x), A(®x)) | A(®x) ∈ b′(®x), b′(®x) ∈ rewrite(b(®x), Tp)}
if Γ is a completeness constraint then
Patterns(Γ) := {((®x), A(®x)) | A(®x) ∈ h′(®x), h′(®x) ∈ rewrite(h(®x), Tp)}
// Extract past corrections
for 0 ≤ i ≤ p − 1 do
if (M′ i, M″ i) such that Kr,i+1 = (Kr ∪ M′ i) \ M″ i matches some pattern in Patterns(Γ) then
PCDataset := {((M′ i, M″ i), Γ(®x), V, i) | V ∈ Violations(Kr,i, Γ(®x)) \ Violations(Kr,i+1, Γ(®x))}
// Remove reversed past corrections
for ((M′ i, M″ i), Γ(®x), V, i) ∈ PCDataset do
if M′ i ∉ Kr or M″ i ∉ Kr or M′ ′ i ∩ M″ i ≠ 0 then
PCDataset := {((M′ ′ i, M″ ′ i), Γ(®x), V, i)}

Let us explain our algorithm with our running example. Consider the constraint I0(x) : ∀y hasGender(y, x) → y = male ∨ y = female ∨ y = nonbinary. Assume that (Zeus, hasGender, masculine) was added between K3 and K2, but then replaced by (Zeus, hasGender, male) between K100 and K101.

The first goal of the algorithm is to find out that the removal of (Zeus, hasGender, masculine) between K100 and K101 (as part of the replacement) may be part of a relevant past correction. We call this deletion a correction seed. Formally, a correction seed is a deletion (θ, M″) or an addition (M′, θ) such that (i) there exists 0 ≤ i ≤ p − 1 such that Kr,i+1 = (Kr ∪ M′ i) \ M″ i with M′ ′ i = M″ i.

Good solutions. Our goal is to find “good” solutions to constraint violations, i.e., solutions that make the KB as close to the real world as possible. The basic requirement for a “good” solution is that it deletes only erroneous facts, and that it adds only true facts. We also prefer replacements over deletions as long as they fulfill this condition. For instance, in our running example, the replacement hasGender(Zeus, masculine) is better than the deletion (θ, hasMother(Spinoza, Marques)), because it corrects erroneous information instead of simply erasing it.

In some cases, there may be no “good” solution that consists of a single atomic modification. Consider for example a completeness constraint of the form A ⊆ B violates by (A(a)). If A(a) is true, we should actually add both P(a, b) and B(b) for some b. We choose to define solutions as atomic modifications nevertheless to simplify the problem by reducing the size of possible solutions. This is a limitation of our approach since we will not be able to learn solutions that are not atomic. However, we will still be able to learn to add B(b) to solve the aforementioned constraint violation in the case where P(a, b) is already present.

The main difficulty in finding good solutions to constraint violations is that we do not have access to an oracle that knows the validity of all facts. This is the problem that all KB cleaning approaches face (cf. Section 2). Our idea is to exploit the history of the KB modifications to learn how to correct constraint violations.

Definition 5 (Edit history): The edit history of a KB is a sequence of KBs (Kr)0≤i≤p = (Kr∪A)0≤i≤p such that Kr+1 = (Kr∪M′ i) \ M″ i, where (M′ i, M″ i) is an atomic modification.

The edit history allows us to pinpoint how constraint violations have been corrected in the past. In order to avoid learning from vandalism or mistakes, we consider only those corrections that have not been reversed:

Definition 6 (Past correction): A past correction is a solution (M′, M″) to a violation V of a constraint instance Γ(®x) in Kr such that there exist B and D such that the current KB Kr,p = (Kr∪B) \ D with M′ ⊆ B, M″ ⊆ D, M′ ∩ D = ∅, M″ ∩ B = ∅.

Intuitively, (B, D) corresponds to the sequence of additions and deletions that leads from Kr,i to the current state of the KB Kr,p, that contains the solution, and that does not “undo” it.

Relevant past corrections. During the history of a KB, users can change not just the assertions of the KB, but also the TBox. However, the TBox is typically much smaller and more stable than the ABox. Therefore, the edit history of the TBox is not a rich ground for correction rule mining. Moreover, we are interested in learning solutions that correct constraint violations in the current KB Kr,p. We thus consider only those past corrections that would have been corrections also under the current TBox. For example, assume that the TBox contained C ⊆ B. Assume that C(a) was added to correct a violation of the constraint A ⊆ B. If, in the meantime, the inclusion C ⊆ B has been removed, we do not want to learn from this past correction. The following definition formalizes these requirements.

Definition 7 (Relevant Past Correction): A relevant past correction (M′, M″) to a violation V of a constraint instance Γ(®x) in Kr is a past correction such that (i) M′ ⊇ M″ contains only assertions, and (ii) V ∪ A∪T contains a violation V′ of Γ(®x) such that (M′, M″) is also a solution to V′ in A∪T. We will now see how we can use the relevant past corrections to mine correction rules.

6 FROM HISTORY TO CORRECTION RULES

In this section, we propose an approach based on rule mining to learn correction rules for building solutions to constraint violations.

6.1 Extraction of the Relevant Past Corrections

Algorithm 1 constructs the set of relevant past corrections from the KB history. It consists of three main steps. First, it constructs patterns to spot KB modifications that could be part of a relevant past correction. Then it uses these patterns to extract atomic modifications that solved some violation in the past. Finally, the relevant past corrections are obtained by pruning those that have been reversed.
We now present our approach to mine correction rules. We collect the patterns for the constraint \( \Gamma \) that we only consider past corrections that involve assertions, and want relevant past corrections (the PCDataset, exemplified in Table 2). The result is thus the set of relevant past corrections.

To find such correction seeds efficiently, the first step of the algorithm precomputes for each constraint a set of atomic modification patterns that the possible correction seeds would match. In the example there would be only one pattern: the deletion pattern \( (_-, (?\text{, hasGender, ?}) \rangle \) where \( _- \) can be anything so that it matches both the deletion of \( (?\text{, hasGender, ?}) \) and its replacements. Since we only consider past corrections that involve assertions, and want them to be relevant for the current TBox, computing the correction seed patterns can be done via query rewriting of the CQs in the body \( b(\bar{x}) \) and the head \( h(\bar{z}) \) of the constraint w.r.t. \( T_r \). Indeed, if \( T \) is a flat QL TBox, any CQ \( q(\bar{x}) \) can be rewritten w.r.t. \( T \) into a UCQ \( q'(\bar{x}) \) such that for every ABox \( A \), answering \( q'(\bar{x}) \) over \( T \cup A \) amounts to answering \( q(\bar{x}) \) over \( A \) [23]. Each atom that occurs in the rewriting of the body of a constraint corresponds to a deletion pattern, and each atom that occurs in the rewriting of the head of a completeness constraint corresponds to an addition pattern. We collect the patterns for the constraint \( \Gamma \) in the set \( \text{Patterns}(\Gamma) \). Note that it is not possible to solve a consistency constraint with an addition, which is why such constraints have only deletion patterns.

The second step of the algorithm verifies, for each correction seed, whether it solved some constraint violation in the past – i.e., whether \( \mathcal{K}_i \) contains some violations of some constraint instances that are not in \( \mathcal{K}_{i+1} \). If so, the modification between \( \mathcal{K}_i \) and \( \mathcal{K}_{i+1} \) is a solution that solved these violations in \( \mathcal{K}_i \). In the example we would have found the violation \( \langle (\emptyset, \text{hasGender, masculine}) \rangle \) of \( \text{I}_0(\text{masculine}) \) in \( \mathcal{K}_{100} \), which is not in \( \mathcal{K}_{101} \). So we would have extracted that \( \langle (\text{Zeus}, \text{hasGender, male}), (\text{Zeus}, \text{hasGender, masculine}) \rangle \) is a solution that solved the violation \( \langle (\text{Zeus}, \text{hasGender, masculine}) \rangle \) of \( \text{I}_0(\text{masculine}) \) in \( \mathcal{K}_{100} \). We store this information as a tuple in the relevant past corrections dataset (the PCDataset), as shown in Table 2. Finding the constraint instances violated in \( \mathcal{K}_i \) or \( \mathcal{K}_{i+1} \) is done via CQ answering (Proposition 1), and computing their violations amounts to computing BCJ justifications (Proposition 2).

The final step of the algorithm removes corrections that have been reversed. The result is thus the set of relevant past corrections.

### 6.2 Correction Rule Mining

**Correction rules.** The previous algorithm has given us a list of relevant past corrections (the PCDataset, exemplified in Table 2). We now present our approach to mine correction rules from this dataset and the KB history.

**Definition 8** (Correction rule): A correction rule is of the form

\[
  r := [\Gamma(\bar{x})] : E(\bar{x}, \bar{y}, \bar{z}) \rightarrow (M^+(\bar{x}, \bar{y}), M^-(\bar{x}, \bar{y})),
\]

where

- \( \Gamma(\bar{x}) \) is a constraint that can be partially instantiated, i.e., some of its variables have been replaced by constants,
- \( (M^+(\bar{x}, \bar{y}), M^-(\bar{x}, \bar{y})) \) is a pair of sets of at most one triple,
- \( E(\bar{x}, \bar{y}, \bar{z}) \) is a set of atoms called the context of the violation such that \( M^-(\bar{x}, \bar{y}) \subseteq E(\bar{x}, \bar{y}, \bar{z}) \),

and both \( (M^+(\bar{x}, \bar{y}), M^-(\bar{x}, \bar{y})) \) and \( E(\bar{x}, \bar{y}, \bar{z}) \) are built from \( N_C \cup N_R \cup \{ \text{rdf:} \text{type} \} \cup N_I \cup R \cup \bar{x} \cup \bar{y} \cup \bar{z} \).

A correction rule can be applied to a KB \( \mathcal{K} \) when there exist tuples of constants \( \bar{a}, \bar{b} \) such that \( \mathcal{K} \) violates \( \Gamma(\bar{a}) \) (recall that this can be decided via CQ answering by Proposition 1) and \( \mathcal{K} \models \exists \Xi E(\bar{a}, \bar{b}, \bar{z}) \).

The result of the rule application is then \( (M^+(\bar{a}, \bar{b}), M^-(\bar{a}, \bar{b})) \).

Note that while the variables from \( E(\bar{x}, \bar{y}, \bar{z}) \) that do not appear in \( \Gamma(\bar{x}) \) or in the head of \( r \) can be existentially quantified, those that occur in the head of \( r \) have to be free: they have to be mapped to individuals occurring in the KB in order to construct the result.

**Example 6:** In our running example, we would like to learn the following correction rules:

\[
  r_1 := [\Gamma_0(\text{masculine})] : \{\text{hasGender(\bar{y}, \text{masculine})}\}
  \rightarrow (\text{hasGender(\bar{y, male}), hasGender(\bar{y, masculine})})

\[
  r_2 := [\Gamma_2(\bar{x})] : \{\text{hasMother(\bar{y, x}), Human(\bar{y})}\}
  \rightarrow (\text{Human(\bar{x}, \emptyset})
\]

The context of the second rule says that if \( x \) is the mother of a human, then \( x \) must also be a human. The rule obtained by replacing Human by Animal would express how to solve a violation of \( \Gamma_2 \) in the context where \( y \) is an animal.

**Mining correction rules.** We mine correction rules with Algorithm 2. This algorithm is an adaptation of the algorithm in [13, 14] to our context, where we learn rules not from a KB but from the PCDataset and the KB history. We first adapt the definitions of the confidence and support from [13, 14] to our case. The support of the body of a correction rule \( r \) for a constraint \( \Gamma \) is the number of violations of \( \Gamma \) stored in the PCDataset that could have been corrected by applying \( r \). Such violations are associated with an instance \( \Gamma(\bar{a}) \) of the partially instantiated \( \Gamma(\bar{x}) \) that appears in \( r \) and with an index \( i \) such that \( \mathcal{K}_i \models \exists \Xi E(\bar{a}, \bar{b}, \bar{z}) \) for some \( \bar{b} \). These two conditions imply that \( r \) could be applied to the KB \( \mathcal{K}_i \). Moreover, we need to check that the result of applying \( r \) to \( \mathcal{K}_i \) actually gives a solution to \( \mathcal{V} \). For example, consider the case where PCDataset contains both \( (\emptyset, \text{R(a, b)}), \Gamma(a), \text{R(a, b)}, A(a), ) \) and \( (\emptyset, \text{R(a, c)}), \Gamma(a), \text{R(a, c)}, A(c), ) \) for \( \Gamma(a) : \exists x \text{R(x, a)} \land A(a) \rightarrow \) false. Both violations count for the support of the body of \( \Gamma(a) \) : \text{R(a, c)} → (\emptyset, \text{R(a, c)}), even in the case where \( \mathcal{K}_i \models \text{R(a, c)} \). Formally, \( \text{sup}_{\text{body}(r)}(r) = |B\text{Sup}| \) where

\[
  B\text{Sup} = \{ \mathcal{V} \mid \subseteq \Gamma(\bar{a}), \nu, i \in \text{PCDataset}, \exists \bar{b} \mathcal{K}_i \models \exists \Xi E(\bar{a}, \bar{b}, \bar{z}) \}
\]

and the result of the application of \( r \) to \( \mathcal{K}_i \) is a solution to \( \mathcal{V} \).
The support of the rule \( r \) measures when the past correction is exactly the result of the application of the rule in the cases where it could be applied. Formally, \( \text{sup}_{\text{rule}}(r) = |R\text{Sup}| \), where

\[
R\text{Sup} = \{ V \mid ((M^+(\bar{a}, \bar{b}), M^-(\bar{a}, \bar{b})), \Gamma(\bar{a}), V, i) \in \text{PCDataset}, \text{ and } \mathcal{K}_i \models \exists \mathcal{E}(\bar{a}, \bar{b}, \bar{z}) \}.
\]

Finally, the confidence of a correction rule \( r \) is \( \text{conf}(r) = \frac{\text{sup}_{\text{rule}}(r)}{\text{sup}_{\text{rule}}(r) + \theta} \).

Algorithm 2 Correction rule mining

Input: PCDataset, \((\mathcal{K}_i)_{0 \leq i \leq p}, \text{minsup}, \text{minconf}, \theta\)

Output: correction rules

// Generate basic rules
BasicR := ∅

for all \( ((M^+(\bar{a}), \bar{b}), (M^-(\bar{a}), \bar{b})), \Gamma(\bar{a}), V, i) \in \text{PCDataset} \)

\( r_0 := [\mathcal{G}(\bar{a})] : M^-(\bar{a}, \bar{b}) \rightarrow (M^+(\bar{a}, \bar{b}), M^-(\bar{a}, \bar{b})) \)

BasicR := \{ \sigma(r_0) \mid C \subseteq \bar{a} \cup \bar{b}, \sigma : C \rightarrow \text{Var}, \text{ sup}_{\text{rule}}(\sigma(r_0)) \geq \text{minsup}, \text{conf}(\sigma(r_0)) \geq \text{minconf} \}

// Refine the context part of the rules
\( q := ∅, q.\text{enqueueAll}(\text{BasicR}) \)

while \( q \) is not empty do

\( r := q.\text{dequeue}() \)

Output \( r \)

for all operators \( op \) do

for all \( r' \in op(r) \) do

if \( \text{sup}_{\text{rule}}(r') \geq \text{minsup} \) and \( \text{conf}(r') \geq \text{conf}(r) + \theta \) then

\( q.\text{enqueue}(r') \)

Applying correction rules. When all rules have been mined, they are sorted by decreasing confidence, breaking ties by help of the support (as it is done in [26] to build classifiers from rules). This set of rules then forms a program that can be used to fix constraint violations as follows. Given a violation \( V \) of a constraint \( \Gamma \) in \( \mathcal{K} \), choose the first rule \( r \) in the program that is relevant for \( \Gamma \) (i.e., that contains \( [\mathcal{G}(\bar{x})] \) where \( \Gamma(\bar{x}) \) is a partially instantiated version of \( \Gamma \)). Then check whether \( r \) can be applied to \( V \). The correction is the result of the rule application.

Example 7: Assume we mined the rules \( r_1 \) and \( r_2 \) of the preceding example with confidence 0.9 and 0.8 respectively, and another rule \( r_3 := [\mathcal{G}(\bar{x})] : [\text{hasGender(x, y)}] \rightarrow (\emptyset, \text{hasGender(x, y)}) \) with confidence 0.5. The correction program is \((r_1, r_2, r_3)\). To correct a violation of \( \Gamma_0 \), a wrong value for the hasGender property, the program first checks whether \( r_1 \) is applicable. If so, it replaces masculine by male. Otherwise, it falls back to \( r_3 \) and removes the wrong value. To correct a violation of \( \Gamma_2 \), it ignores \( r_1 \) that is not related to \( \Gamma_2 \) and either applies \( r_2 \) if the context matches or does nothing.

7 EXPERIMENTS ON WIKIDATA

This section describes CorHist, which implements the framework introduced for Wikidata, and presents its experimental evaluation.

7.1 Wikidata

Wikidata is a generalist collaborative knowledge base. The project started in 2012, and as of July 2018, it has collected more than 500M statements about 50M entities. The data about each entity is stored in a versioned JSON blob, and there are more than 700M revisions. Wikidata encodes facts not in plain RDF triples but in a reified representation, in which each main (s, p, o) triple can be annotated with qualifiers and provenance information [38].

Wikidata knows the property instanceOf which is similar to rdf:type. It does not have a formally defined TBox, but knows properties such as subClassOf, subPropertyOf, and inverseOf. However, only the property subClassOf is used to flag the constraint violations. Therefore, we use only this property in our TBox, which thus contains simple concept inclusions.

We consider the set \( C \) of constraints built from ten types of Wikidata property constraints (see Table 3). They are the top Wikidata property constraints that can be expressed in DL, covering the majority of the most used constraints, as well as 71% of Wikidata property constraints. The remaining constraints are mainly about string format validation with regular expressions (52% of the remaining constraints) and about qualifiers (31% of them).

7.2 Dataset Construction

We stored the RDF version [10, 18] of the Wikidata edit history in an RDF quad store. We used named graphs for the global state of Wikidata after each revision, and for the triple additions and deletions. Our dataset stores 390M annotated triples about 49M items extracted from the July 1st, 2018 full database dump.

We extracted the relevant past corrections as explained in Section 6.1. Wikidata revisions do not correspond exactly to atomic modifications in our sense. For example, Wikidata bots are able to change multiple unrelated facts about the same entity at the same
Table 3: Wikidata property constraints. $R$ is the property for which the constraint is given. A constraint has several lines when it uses a property whose set of values may be specified or not. $\#$constr. is the total number of constraints of the given type in Wikidata. $\#$triples is the sum for all these constraints of the numbers of triples with the property $R$ on which they apply. $\#$violations is the number of violations for this constraint in Wikidata on July 1st, 2018. $\#$past cor. is the number of past corrections we extracted from Wikidata history. t.o. indicates that we were not able to extract all past corrections because of timeout so that we sample them (we then indicate the number of corrections we extracted).

<table>
<thead>
<tr>
<th>Name in Wikidata</th>
<th>DL form</th>
<th>Rule form</th>
<th>$#$constr.</th>
<th>$#$triples</th>
<th>$#$violations</th>
<th>$#$past cor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 4</td>
<td>$\exists R \subseteq A_1 \sqcup \cdots \sqcup A_n$</td>
<td>$\exists y R(x, y) \rightarrow A_1(x) \lor \cdots \lor A_n(x)$</td>
<td>2575</td>
<td>249M</td>
<td>3465k</td>
<td>t.o.(&gt;16M)</td>
</tr>
<tr>
<td>Value type 4</td>
<td>$\exists R \subseteq A_1 \sqcup \cdots \sqcup A_n$</td>
<td>$\exists y R(y, x) \rightarrow A_1(x) \lor \cdots \lor A_n(x)$</td>
<td>696</td>
<td>67M</td>
<td>3062k</td>
<td>t.o.(&gt;19M)</td>
</tr>
<tr>
<td>One-of</td>
<td>$\exists R^\rightarrow \subseteq {a_1, \ldots, a_n}$</td>
<td>$\exists y R(x, y) \rightarrow x = a_1 \lor \cdots \lor x = a_n$</td>
<td>104</td>
<td>36M</td>
<td>4k</td>
<td>14k</td>
</tr>
<tr>
<td>Item requires</td>
<td>$\exists R \subseteq \exists R'$, ${a_1, \ldots, a_n}$</td>
<td>$\exists y R(x, y) \rightarrow R'(x, a_1) \lor \cdots \lor R'(x, a_n)$</td>
<td>3102</td>
<td>255M</td>
<td>3710k</td>
<td>t.o.(&gt;15M)</td>
</tr>
<tr>
<td>Statement</td>
<td>$\exists R \subseteq \exists R'$</td>
<td>$\exists y R(x, y) \rightarrow \exists z R'(x, z)$</td>
<td>243</td>
<td>85M</td>
<td>1345k</td>
<td>t.o.(&gt;6M)</td>
</tr>
<tr>
<td>Value requires</td>
<td>$\exists R' \subseteq \exists R$, ${a_1, \ldots, a_n}$</td>
<td>$\exists y R(y, x) \rightarrow R'(x, a_1) \lor \cdots \lor R'(x, a_n)$</td>
<td>601</td>
<td>449M</td>
<td>142k</td>
<td>465k</td>
</tr>
<tr>
<td>Conflict with</td>
<td>$\exists R \subseteq \neg \exists R'$, ${a_1, \ldots, a_n}$</td>
<td>$\exists y R(x, y) \land (R'(x, a_1) \lor \cdots \lor R'(x, a_n)) \rightarrow \text{false}$</td>
<td>146</td>
<td>6M</td>
<td>409k</td>
<td>2989k</td>
</tr>
<tr>
<td>Single value</td>
<td>(func $R$)</td>
<td>$R(x, y) \land R(z, x) \rightarrow y = z$</td>
<td>2772</td>
<td>85M</td>
<td>334k</td>
<td>389k</td>
</tr>
<tr>
<td>Distinct values</td>
<td>(func $R^\rightarrow$)</td>
<td>$R(x, y) \land R(z, x) \rightarrow y = z$</td>
<td>2728</td>
<td>56M</td>
<td>189k</td>
<td>7432k</td>
</tr>
</tbody>
</table>

Time. Wikidata users also sometimes prefer to delete a statement then add another one with the same property instead of directly modifying the value, in order to clear the existing qualifiers and references. Therefore, we artificially created a replacement modification for every deletion with a neighboring addition by the same user, which shares at least two components of the triple (analogously for additions). For example, if the correction seed is the deletion of (Zeus, hasGender, masculine), and if this revision or a neighboring one adds (Zeus, hasGender, male), then we consider this a replacement. However, if the same revision added the triple (Zeus, hasMother, Rhea), then we would not consider this a replacement, because it does not share two components with the first one.

Since the TBox consists of simple concept inclusions and the constraint bodies contain only roles, the deletion patterns for correction seeds correspond directly to the atoms of the constraint body. In the same vein, only atoms in the head of the Type or Value type constraints need to be rewritten. To find the constraint violations solved by a correction seed, we make use of the fact that the correction seed allows us to know the constraint instance $\Gamma(\delta)$, and we look for matches of the constraint body.

To speed up the execution for the four constraint types which have the highest numbers of past corrections, Type, Value type, Item requires statement and Value requires statement, we did not extract all the past corrections but sample them as follows. We compute only the relevant past past corrections where applied between $\mathcal{K}_i$ and $\mathcal{K}_{i+1}$ where $i$ is a multiple of $s := \max(1, N/10^6)$ with $N$ the number of triples with the property $R$ of the constraint at hand. This sampling allows us to get a sufficient ground for rule mining for each constraint. In practice, it affects only the most frequent 0.9% of Type, 2% of Value type, 0.5% of Item requires statement, and 3% of Value requires statement constraints.

7.3 Mining Rules

The output of our method is a set of correction rules that form a program (Section 6.2). To evaluate such a program, we apply it to each of the constraint violations stored in the PCDataset, using the associated stage of the KB to evaluate the part of the context which is not the deletion part of the correction. Then we check whether the correction we compute is exactly the same as the one associated to the constraint violation in the PCDataset. The precision $p$ of the program is given by the fraction of the corrections computed by the program that are actually the same as those that have been applied. The recall $r$ of the program is the fraction of the constraint violations stored in PCDataset for which the program gives some correction. The $F1$ score is $F_1 = 2\frac{p \cdot r}{p + r}$.

CorHist mines rules as explained in Section 6.2. In order to decrease the computation time, we only allow one atom $p(x, o)$ in $E(\tilde{x}, \tilde{y}, \tilde{z}) \setminus \text{Body}(\tilde{x}, \tilde{y})$, where $\text{Body}(\tilde{x}, \tilde{y})$ corresponds to the part of the context that matches part of the constraint body, such that $s$ is a variable of $\tilde{x} \cup \tilde{y}$ and $o$ is a fresh variable or a constant.

Rules were mined per constraint. For each constraint, we split the set of extracted past corrections into a 70% training set, a 10% cross-validation set, and a 20% test set. The training set is used to mine the rules, the cross-validation set is used to determine the confidence threshold that maximizes the $F1$ score of the obtained program, and the test set is used to evaluate the final program.

Table 4 gives examples of rules mined by CorHist. Several of these rules show the crucial importance of the instantiation of the constraint and/or of the context to be able to choose the correction. For instance, the rule for the Single value constraint uses the fact that an entity involved in a property “member of sport team” is

---

1. The Wikidata constraint Type can be qualified to modify its meaning. We ignore these cases, which are marginal: they concern less than 6% of the Type constraints. The same goes analogously for Value type.
2. Inverse and Symmetric are two distinct kinds of constraints in Wikidata but we treat them together since Symmetric is actually a special case of Inverse.
**Table 4: Example of mined rules.**

<table>
<thead>
<tr>
<th>Constr. type</th>
<th>Constraint $\Gamma$</th>
<th>Correction rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>$\exists$aListOf $\supseteq$ List</td>
<td></td>
</tr>
<tr>
<td>Value type</td>
<td>$\exists$foundInTax $\supseteq$ Taxon</td>
<td></td>
</tr>
<tr>
<td>One-of</td>
<td>$\exists$mannerDeath $\supseteq$ [ \ldots ]</td>
<td></td>
</tr>
<tr>
<td>Item req. stm.</td>
<td>$\exists$heritageStatus $\supseteq$ $\exists$country</td>
<td></td>
</tr>
<tr>
<td>Val. req. stm.</td>
<td>$\exists$residence $\supseteq$ $\exists$Country</td>
<td></td>
</tr>
<tr>
<td>Conflict</td>
<td>$\exists$filmId $\supseteq$ ¬$\exists$filmId</td>
<td></td>
</tr>
<tr>
<td>Inv./Sym.</td>
<td>$\exists$geneticAssoc $\supseteq$ $\exists$geneticAssoc</td>
<td></td>
</tr>
<tr>
<td>Single val.</td>
<td>(func sexOrGender)</td>
<td></td>
</tr>
<tr>
<td>Distinct val.</td>
<td>(func ncbiLocusTag)</td>
<td></td>
</tr>
</tbody>
</table>

$[\Gamma(s)]$ : isALisOf(s, o) ∧ WikiDisambiguationPage(s) → (\emptyset, isALisOf(s, o))

$[\Gamma(s)]$ : isALisOf(s, o) ∧ WikiDisambiguationPage(s) → (\emptyset, isALisOf(s, o))

$[\Gamma(s)]$ : foundInTax(s, human) ∧ hasPart(s, c) → (foundInTax(s, homoSapiens), foundInTax(s, human))

$[\Gamma(s)]$ : mannerDeath(s, trafficAcc) → (causeDeath(s, trafficAcc), mannerDeath(s, trafficAcc))

$[\Gamma(s)]$ : heritageStatus(s, monarchInFromminnesregistrer) → (country(s, Sweden), \emptyset)

$[\Gamma(s)]$ : diplomaticRelation(s, v) → (country(s, e), \emptyset)

$[\Gamma(s)]$ : filmId(s, o) → (\emptyset, filmId(s, o))

$[\Gamma(s)]$ : sexOrGender(s, maleOrg) ∧ sportsTeam(s, v) → (\emptyset, sexOrGender(s, maleOrg))

$[\Gamma(s)]$ : ncbiLocusTag(o, s) ∧ molecularFunction(o, v) → (\emptyset, ncbiLocusTag(o, s))

**Table 5: Evaluation of the correction rules mined by CorHist with a minimal support of 10, a minimal confidence between 0.3 and 1 and a regularization threshold of 0.05, and comparison with the baselines. Best precision and F1 scores in bold.**

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Micro average</th>
<th>Macro average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec.</td>
<td>Rec.</td>
</tr>
<tr>
<td>Type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>0.35</td>
<td>0.33</td>
</tr>
<tr>
<td>delete</td>
<td>0.04</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.84</td>
<td>0.70</td>
</tr>
<tr>
<td>Value type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>0.09</td>
<td>0.21</td>
</tr>
<tr>
<td>delete</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.81</td>
<td>0.61</td>
</tr>
<tr>
<td>One-of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.26</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.77</td>
<td>0.83</td>
</tr>
<tr>
<td>Item requires statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.017</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.94</td>
<td>0.35</td>
</tr>
<tr>
<td>Value requires statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.041</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.91</td>
<td>0.53</td>
</tr>
<tr>
<td>Conflict with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.39</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.92</td>
<td>0.55</td>
</tr>
<tr>
<td>Inverse/Sym.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>0.91</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>0.072</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td>Single value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.34</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.95</td>
<td>0.093</td>
</tr>
<tr>
<td>Distinct values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>0.036</td>
<td>1</td>
</tr>
<tr>
<td>CorHist</td>
<td>0.99</td>
<td>0.020</td>
</tr>
</tbody>
</table>

For example, for Type constraints of the form $\exists y R(x, y) \rightarrow A(x)$ it applies the correction $(A(x), \emptyset)$. Value type constraints are handled in the same way. However, this baseline is not able to figure out what is the relevant addition correction for a constraint of the form $\exists y R(x, y) \rightarrow A_1(x) \lor A_2(x)$ because there is no way to know a priori if $A_1$ or $A_2$ should be added. For Item requires statement constraints of the form $\exists y R(x, y) \rightarrow R'(x, a)$, the “add” baseline applies $R'(x, a)$ (similarly for Value requires statement constraints). However, it cannot find a correction if there are multiple $a_i$. 

**7.4 Evaluation against the Test Set**

Table 5 presents the results of the evaluation of the mined programs against the test set. We computed both the micro and macro average of the precision, recall and F1 score per kind of constraint. The micro average aggregates over the whole set of relevant past corrections for the given kind of constraint, whereas the macro average computes the scores for each constraint of this given kind, and then computes the average. Both numbers are important: The micro average gives more weight to correction rules that fix many violations. It thus measures the overall impact of the correction rules on the dataset. However, if few rules had a large impact, then it would be easier to formulate these rules by hand. Our method, in contrast, can also find rules that by themselves solve less violations, but together contribute a large mass of corrections. To illustrate this, we also report the macro average: It measures the average performance across different constraints.

We compare our approach with two baselines: The first one, called “delete”, is the most basic one and uses the fact that all Wikidata constraint bodies contain an atom of the form $R(x, y)$ and the TBox contains only concept inclusions, so that all constraint violations contain an assertion that matches $R(x, y)$. The “delete” baseline simply deletes this assertion. For the completeness constraints we define an additional baseline, “add”, which tries to add a new triple to solve the constraint violation. For Inverse and Symmetric constraints this baseline adds the missing reverse edge and performs very well. For Item requires statement, Value requires statement, Type and Value type, it adds the missing triple only if it is possible to know the expected value from the constraint rule. 

\(^6\)Computed from a sample of the set of relevant past corrections. One Type constraint, six Value type constraints and one Val. req. stm constraint were omitted due to time-out.

\(^7\)The actual value is greater than 0.995 and rounded to 1 for consistency.
An entity should not have a statement for country of citizenship if it also has a statement for sex or gender with value male non-human organism.

Possible correction

Edit statement (Q57082102, sex or gender, male non-human organism). Setting value to: male

Figure 1: Example of a replacement correction suggested by CorHist for a violation of the constraint \( \exists \text{countryOfCitizenship} \sqsubseteq \neg \exists \text{sexOrGender} \cdot \{ \text{maleOrg}, \text{femaleOrg} \} \).

Table 6: Human evaluation of the suggested corrections.

<table>
<thead>
<tr>
<th>Constraint type</th>
<th>Suggested</th>
<th>“Apply”</th>
<th>“Wrong”</th>
<th>Approval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>9908</td>
<td>252</td>
<td>312</td>
<td>0.45</td>
</tr>
<tr>
<td>Value type</td>
<td>2374</td>
<td>195</td>
<td>208</td>
<td>0.45</td>
</tr>
<tr>
<td>One-of</td>
<td>239</td>
<td>14</td>
<td>47</td>
<td>0.23</td>
</tr>
<tr>
<td>Item requires stmt.</td>
<td>41</td>
<td>8</td>
<td>32</td>
<td>0.2</td>
</tr>
<tr>
<td>Value requires stmt.</td>
<td>1024</td>
<td>790</td>
<td>178</td>
<td>0.82</td>
</tr>
<tr>
<td>Conflict with</td>
<td>3254</td>
<td>1717</td>
<td>203</td>
<td>0.89</td>
</tr>
<tr>
<td>Inverse/Symmetric</td>
<td>28138</td>
<td>20247</td>
<td>1720</td>
<td>0.92</td>
</tr>
<tr>
<td>Single value</td>
<td>3264</td>
<td>41</td>
<td>71</td>
<td>0.37</td>
</tr>
<tr>
<td>Distinct values</td>
<td>921</td>
<td>8</td>
<td>23</td>
<td>0.26</td>
</tr>
</tbody>
</table>

As shown in Table 5, the precision of our approach significantly outperforms the two baselines—often by a very high margin. Regarding the recall, we manage to keep a reasonable, and sometimes even good, recall (see best F1 scores in Table 5), except for Single and Distinct value. The very low recall obtained for these two kinds of constraints is easily explainable because they are mostly used on predicates that link Wikidata to other databases (91% of the Single and 95% of the Distinct constraints), and we cannot get meaningful information about the target database to mine corrections.

7.5 User Evaluation

To see whether our corrections are accepted by the community, we designed a user study. We created a tool that suggests our corrections for random violations of constraints of this type (Figure 1). The violations for which corrections are suggested are provided by query.wikidata.org, which limits their number for performance reasons. For each proposed correction, the user has to choose between three options: apply the proposed correction to Wikidata, tag it as wrong, or get another correction to review. We ran the experiment for 3 months and 47 Wikidata users participated.

Table 6 presents the results. The number of corrections reviewed is highly unbalanced between the kinds of constraints, mainly because a few users evaluate a lot of suggestions, and have a predilection for some kinds of constraints. It is thus difficult to draw conclusions for those kinds of constraints for which very few corrections have been evaluated. However, we can still make some interesting observations. In particular, the proposed corrections marked as wrong give us insights about possible weaknesses of our approach.

For the constraints which got a significant number of evaluations, our approach seems to perform well for Inverse and Symmetric, Conflict with and Value requires statement constraints, with approval rates above 80%. The other approval rates are lower. This is partly due to biases in the data. For example, when a gender is missing, our approach proposes the value “male” by default, because of the over-representation of men in Wikidata. Another issue is the quality of the constraints, which in Wikidata are sometimes questionable or difficult to understand (e.g., an incomplete set of possible types or values for completeness or One-of constraints).

However, even lower approval scores do not mean that our approach would be useless: Psychological research [12] shows that people find it much easier to choose from given options than to come up with an answer by themselves. The actual time needed to come up with an answer may vary, but if it takes just 3 times longer to come up with an answer than to accept or reject our proposed correction, then achieving a precision of 40% is already useful. If we have a precision of 40%, and if a free-form answer takes time \( t \), then the expected answer time with our tool is \( 40\% \times \frac{3}{4} \times t + 60\% \times \frac{1}{4} \times t < t \).

8 CONCLUSION AND FUTURE WORK

We have introduced the problem of learning how to fix constraint violations from a KB history. We have also presented a method based on rule mining to this end. Our experimental evaluation on Wikidata shows significant improvement over baselines. Our tool is live on Wikidata and has already allowed users to correct more than 23k constraint violations. While our evaluation focused on Wikidata for which the whole edit history was available, we believe that our method can be applied in other settings, for example using edits done during the partial cleaning of an automatically extracted KB.

For future work, it would be interesting to evaluate the impact of parameters such as the size of the context part of the correction rule in terms of rule quality. We also plan to extend the learning dataset with external knowledge (such as other KBs), or with information extracted from other sources (for instance from Wikipedia). We believe that this will allow finding even more precise correction rules, thus making KBs ever more precise and more useful.

ACKNOWLEDGMENTS

Partially supported by the grant ANR-16-CE23-0007-01 (“DICOS”).