Gottfried Wilhelm Leibniz Universität
Hannover
Fakultät für Elektrotechnik und Informatik

Masterthesis

Supporting Explainable AI on Semantic Constraint Validation

Julian Alexander Gercke

Matriculation number:	10003079
First evaluator:	Maria-Esther Vidal
Seconde evaluator:	Sören Auer
Supervisors:	Philipp D. Rohde and Maria-Esther Vidal
Submission date:	14th June, 2022

DECLARATION OF AUTHORSHIP

I, Julian Alexander Gercke, declare that this thesis titled, 'Supporting Explainable AI on Semantic Constraint Validation' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Place, Date	Julian Alexander Gercke

Abstract

There is a rising number of knowledge graphs available published through various sources. The enormous amount of linked data strives to give entities a semantic context. Using SHACL, the entities can be validated with respect to their context. On the other hand, an increasing usage of AI models in productive systems comes with a great responsibility in various areas. Predictive models like linear, logistic regression, and tree-based models, are still frequently used as they come with a simple structure, which allows for interpretability. However, explaining models includes verifying whether the model makes predictions based on human constraints or scientific facts. This work proposes to use the semantic context of the entities in knowledge graphs to validate predictive models with respect to user-defined constraints; therefore, providing a theoretical framework for a model-agnostic validation engine based on SHACL. In a second step, the model validation results are summarized in the case of a decision tree and visualized model-coherently. Finally, the performance of the framework is evaluated based on a Python implementation.

Contents

Chapter 1

Introduction

AI is called explainable when a user of an AI model can understand the predictions a model makes. The user is enabled to not only interpret the model but trust the model through given reasoning of why a decision has been made. [16, 27]
Knowledge graphs are used to store and connect data, promoting the cooperation between machines and humans. Therefore the data and its entities are stored in a way understandable for both. [7]
Data is extracted from knowledge graphs for statistical analysis [3, 11, 48]. However, the process only uses the context of the extracted entities during the data mining process. On the other hand, embeddings represent the entity through its properties and links to other entities (i.e., latent vectors). Therefore, they incorporate the semantic context for later usage (e.g., clustering by embedding-based similarity measures [18]) but lake for explainability. Finally, there are rule-based systems. Mohamed et al. [25] mine rules for comprehensible cluster labels. Halliwell et al. [30] generate ground truth explanations for the link prediction task based on handcrafted rules. Even handcrafted rules show to suffer from understandability in some cases [30].
In the area of the semantic web, SHACL engines are known for their capability to validate knowledge graphs against a set of constraints (i.e., rules). Therefore rating entities based on their integrity and trustability.
This work aims to exploit SHACL engines to support the explainability of predictions made by machine learning models. The approach is naturally based on handcrafted constraints (i.e., rules), which should facilitate the generation of understandable explanations. Based on the constraints the approach will additionally increase the interpretability of AI models by annotating the model with constraint validation results. The constraint validation results will be based on the semantic context provided by a knowledge graph. Standard interpretability methods (e.g., LIME [56], or Partial Dependence Plots [24]) do not take into consideration the context of the entities used to train the model or make predictions. Therefore Using the semantic context of entities in the described way is novel and closes a hole in interpretability methods.
Recently, machine-learning-based techniques have been used to accelerate COVID-19-related drug discovery during the pandemic [67]. Drugs found through these kinds of techniques need to be validated carefully. On a large scale, this highlights the importance of explainable AI. Most likely, the lack of explainability of the AI model used for drug discovery leads to the need to validate the results by hand. However, given the expertise in validating drugs, there are for sure constraints involving the training data and the assigned targets. These could have been used for automatic validation and a step toward better explainable AI.

weight	person	allergic_to	gender	pregnant	country	vaccinated
$\frac{4}{12}$	:Max	PEG	male	$\bot$	Germany	$\bot$
$\frac{1}{12}$	:Maria		female	$\top$	Germany	$\bot$
$\frac{6}{12}$	:Eva		female	$\bot$	Germany	$\top$
$\frac{1}{12}$	:Laura	PEG	female	$\bot$	Germany	$\bot$

Figure 1.1: Dataset about COVID-19 Vaccination

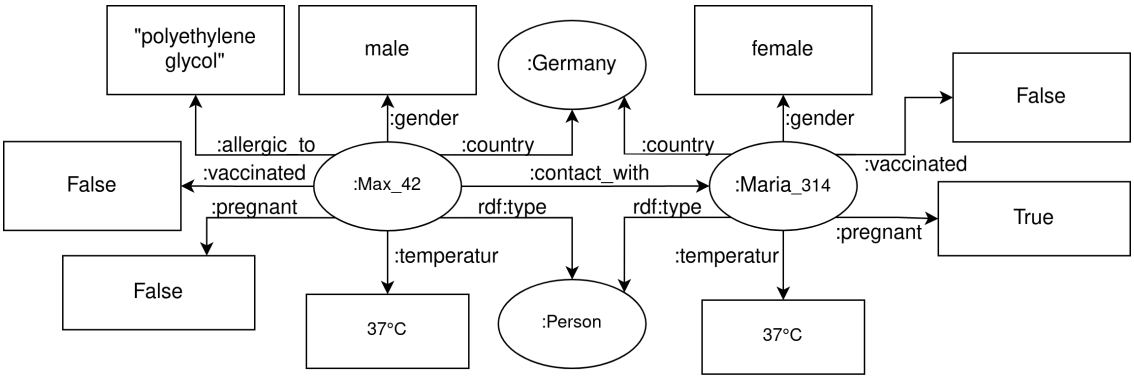


Figure 1.2: Visualization of an RDF Graph containing some Instances of the Data

1.1 Motivating Example

The motivating example is about the social problem of vaccine hesitancy. This is a related problem, which got pronounced during the COVID-19 crisis, when the trust in official sources and scientific studies became a decisive factor in connection with the willingness to vaccinate [52]. To counteract this behavior, one might imagine a predictive model based on a knowledge graph, which gives explainable recommendations to people, whether they should get vaccinated or not. Therefore, a social-themed knowledge graph about people and their COVID-19 vaccination status would be required. Due to privacy reasons, such a knowledge graph is not publicly available. However, one might expect that the introduction of electronic health records in more and more countries [19] will lead to broader availability of this kind of data to specific persons in the future.
For the sake of comprehensibility, a tiny dataset (see figure 1.1) is used for demonstration purposes. The dataset is not meant to be true in reality, but will help explain the approach (see chapter 4). Each row in the table represents a type of sample, which has a weight assigned. This weight gives the proportion of this type of sample in the dataset. As described above, the example assumes a knowledge graph, which was used to extract the data. A small part of the knowledge graph is depicted in Figure 1.2.
In this work, the goal is twofold. On the one hand, the work aims to use the additional context provided by a knowledge graph for a given set of entities. This is to be implemented to discover further insights into the patterns of machine learning models trained on the data extracted from the knowledge graph. On the other hand, a goal is to allow users to validate certain assumptions about the model using the context. To achieve that, user-defined constraints about the target of the machine learning model are validated against a knowledge graph and a model's predictions. When extracting a dataset, as in Figure 1.1, from a knowledge graph, as in Figure 1.2, the graph structure of the data gets lost. In this example, there are the `contact_with` predicates, which get discarded. The ex-

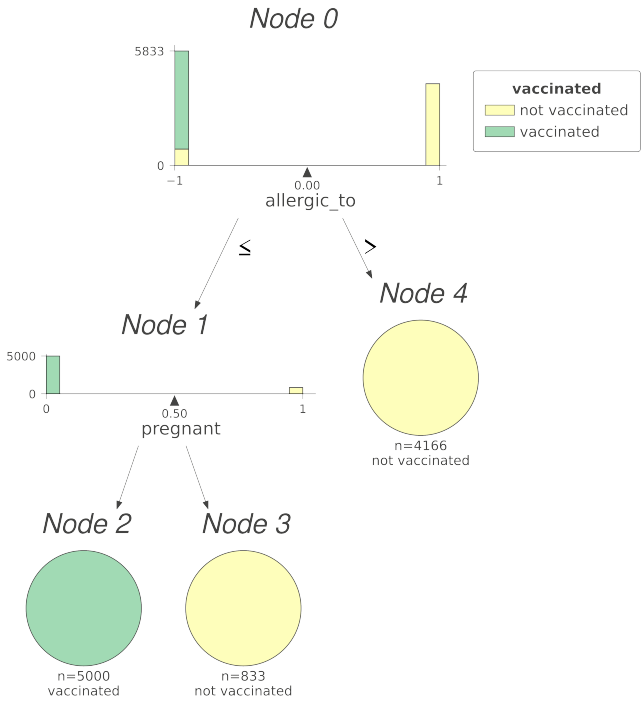


Figure 1.3: Decision tree trained on the dataset from figure 1.1 assuming 9999 samples (n denotes the number of samples getting predictions assigned in the leaf) visualized using dtreeviz [63]

ample will show how this kind of semantic context, exclusively available in the knowledge graph, might be used to explain the model's predictions.
Available knowledge graphs suffer from data quality issues. For example, seven years ago, a paper identified certain substantial constraints over DBpedia [45], which are frequently violated [39]. The situation has not changed until today [58]. This problem gets accelerated when learning from such kind of data. A model trying to generalize to this data will suffer from the same quality constraints as the data. Moreover, it will apply the recognized patterns to new data, which in turn, will also suffer from the same issues.
Both, knowledge graphs as a source of further insights and data quality issues in knowledge graphs give rise to semantic constraint validation. This has the capability of identifying data points, which violate certain constraints and predictions made on the basis of these. Let us assume a decision tree, trained on the given dataset, which recommends a person X, which is not allergic to PEG (included in COVID-19 vaccines) but is pregnant, not to get vaccinated (see Figure 1.3).
Although the decision tree is already inherently explainable, and it is, therefore, known why it recommends not getting vaccinated, it remains unclear whether the decision is rational with respect to human constraints or scientific facts. For example, there might be a constraint stating: „*Every pregnant person in Germany, which has more than 20 contacts with non-vaccinated persons should get vaccinated*“. The decision tree alone cannot be used to validate the constraint. The underlying knowledge graph is needed to count the number of contacts with vaccinated persons. An option would be to include the number of contacts with non-vaccinated persons in the dataset, but that will not guarantee that the decision tree uses this data in a constraint conform way. Further, one could exclude the examples from the dataset which violate the constraint. This

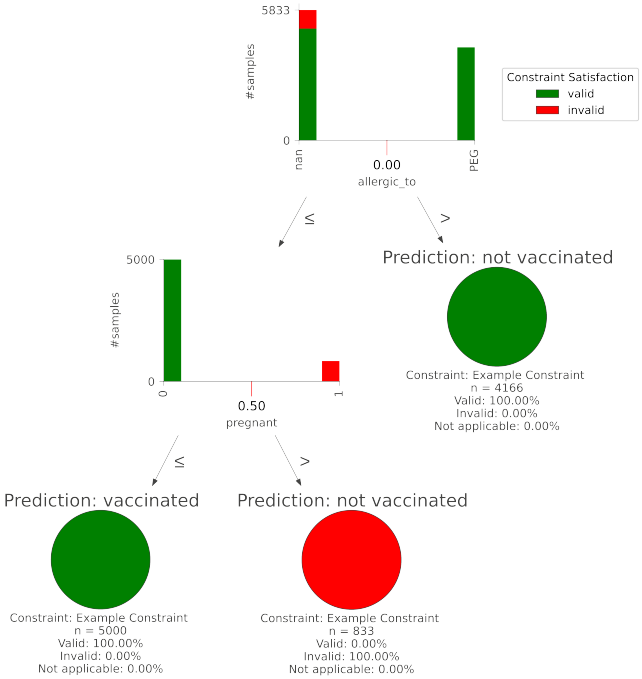


Figure 1.4: The decision tree of figure 1.3 annotated with the validation results of the example constraint. Pregnant persons not allergic to PEG are predicted not to be vaccinated, which violates the example constraint, given the semantic context of the persons in the knowledge graph.

might be a viable approach but will only eliminate the faulty examples from the dataset, which might, in turn, lead to a better-adapted model. However, edge cases not covered by the dataset and a generalized model might continue to give predictions violating the constraint. Therefore, it might be good to learn about the model trained on the available data.
As a result, the approach implicitly validates the model given a set of constraints and the knowledge graph to explain the model's prediction with respect to the constraints. In this context, "implicitly" means that each sample in the dataset is used to conclude the model's behavior. This kind of information is then used to visualize constraint satisfaction. An example of such a visualization is shown in Figure 1.4, which demonstrates that the prediction the model makes for person X violates the constraint. On the other hand, it confirms the other predictions of the model. For example, the model suggests not getting vaccinated if someone is allergic to PEG. The constraint does not contradict these predictions. Therefore, a potential user of the model can be sure that the model's decision is not made based on data for which the model makes invalid predictions.
Performing constraint validation this way has further advantages: One might find errors in the model, which need to be solved, or reject invalid predictions. Furthermore, a model can be checked to apply in a different environment with different constraints.

1.2 Contributions

This work contributes a theoretical framework to efficiently validate predictive models using user-defined constraints based on semantic web technologies. The framework exploits

the SHACL shape schema validation to make the semantic context of entities usable for the validation of predictive models. In this context, two types of constraints arise: (i) *Prediction constraints* to explain and check the model's predictions and (ii) *Data constraints* to measure the trustworthiness of the samples in the dataset used to train a predictive model.
Heuristics are proposed for efficiency: SHACL engine agnostic heuristics allow to minimize the number of SHACL validation results needed to be generated. Join heuristics are presented to reduce the time required to align the SHACL schema validation results with the samples in the dataset used for machine learning.
Based on the constraint validation results generated by the framework, models' predictions can be explained or better interpreted. The way of visualizing decision trees of the `dtreeviz` library [63] with the samples' ground truth distribution is transferred to be usable for the annotation with constraint validation results. The latter allows for increased interpretability. Confusion matrices are decomposed for better interpretability of classifiers' performance. The whole framework is publicly available in the form of a python library[1] and is evaluated with respect to efficiency criteria.

1.3 Document Structure

The thesis is split up into eight chapters. Chapter 1 is the introduction and provides the reader with a motivating example, the main contributions, and the problem tackled in this work. Next, chapter 2 presents the mathematical notations, symbols, and the basic concepts needed to understand the following chapters. Having understood the necessary background, chapter 3 highlights topics related to this work, summarizes the state-of-the-art while referring to suitable papers, and positions the thesis with respect to them. The approach, including the formal problem definition, the proposed solution, and extensions and improvements to it, are discussed formally in chapter 4. For a better understanding of the approach, the motivating example is used to demonstrate the method by using it as a running example. Chapter 5 provides the reader with an implementation of the approach, highlighting the different modules and design decisions. Parallel to the creation of this work, a resource called "InterpretME" has been created as a collaborative project with four Scientific Data Management Research Group members. InterpretME [12] is shortly presented as an application of this work in chapter 6. The implementation is evaluated with respect to its performance based on four benchmarks in chapter 7. Finally, it comes to a conclusion in chapter 8.

1.4 Summary

The rough overall approach presented in the motivating example is summarized in short here. Figure 1.5 shows the sequential process described in the motivating example. The process starts with a knowledge graph (like Figure 1.2), which gets transformed into a dataset (like Figure 1.1). The dataset is in turn used to train a machine learning model like the decision tree in Figure 1.3. Now, constraints like the one about contacts with non-vaccinated persons can be validated over the knowledge graph and the trained machine learning model. Afterward, the constraint satisfaction results can be used for visualizations, meant to explain predictions of the model, show the validity according to

[1] https://github.com/JulianLoewe/Validating_Models

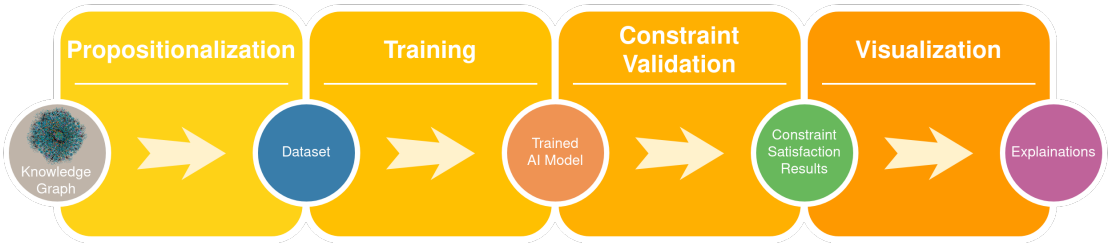


Figure 1.5: Showing the overall approach as a sequential process

the constraint, and might provide inside into patterns used by the model. Figure 1.4 illustrates this visualization.

Chapter 2

Background

This chapter introduces some basic concepts and notations used in this work. Although some basic definitions and lemmas are presented, no claim is made to completeness. The reader should refer to the literature cited for a sound and complete description of the concepts. The background is split into two main parts, referring to the title of the thesis. The first part is about the concepts and the topic of semantic constraint validation, and the second part focuses on introducing approaches related to explainable AI.

2.1 Preliminaries

This section is used to summarize the notation used in this work. Therefore, let A, B and C be sets; D be a tuple; $f : A \to B$ and $g : B \to C$ arbitrary functions and $a \in \mathbb{R}$ to introduce the mathematical symbols listed in Table 2.1.

Symbol	Meaning
$\mathcal{G}_f$	The Graph $\{(x \mapsto f(x)) \mid x \in A \wedge f(x) \in B\}$ of f
$\mathrm{dom}(f)$	The domain $\{x \mid (x \mapsto f(x)) \in \mathcal{G}_f\}$ of f
$\mathcal{P}(A)$	Power set of A
$f \circ g$	Composition of f and g
$\emptyset$	Empty set
tuple	A function transforming a set into a tuple with arbitrary order
$max(A)$	The maximum x of A where $\forall a \in A\ (x \geq a)$
$min(A)$	The minimum x of A where $\forall a \in A\ (x \leq a)$
id	The identity function mapping each instance to itself
$D[i]$	The i-th element in D
$\lvert A \rvert$	Number of elements in A
$\lvert a \rvert$	The absolute value of a
$(a_{i,j})_{\substack{i \in A \\ j \in B}}$	A matrix with elements $a_{i,j}$ and a dimension of $\lvert A \rvert \times \lvert B \rvert$
$E := A$	E is assigned A, such that E equals A after the operation.
$f(.)$	f is marked to be a function of arity one.

Table 2.1: Mathematical Notation

Further, this work makes use of several logical expressions. To define what a logical expression is and how it should be interpreted, the concept of languages and structures is used. In this context, a word in the language is a logical expression, which is built

according to the predicate logic and interpreted using a given structure $\mathbb{S} = (A; \mathcal{R}; \mathcal{F}; \mathcal{C})$. A is called the carrier of the structure and provides the set of available values for variables. $\mathcal{R}$ is the set of relations, $\mathcal{F}$ is the set of functions, and $\mathcal{C}$ is the set of constants. All the syntactically correct words, which are built according to the predicate logic with the symbols, relations, and functions provided by $\mathbb{S}$ are contained in the language, which is denoted by $\mathcal{L}_{\mathbb{S}}$.
Semantically, the symbols used in this work have the usual meaning, if not defined otherwise. In this thesis, the logical expressions do not involve quantification, therefore, variables can only occur unbound. A logical expression $\sigma \in \mathcal{L}_{\mathbb{S}}$ can be evaluated via $\mathcal{I} \models \sigma$ the logical true ($\top$) or false ($\bot$), where $\mathcal{I}$ is an interpretation. An interpretation is a tuple $(\mathbb{S}, \beta)$, where β is a partial function assigning values from the carrier A to the available variables.
To distinguish between this kind of evaluation and the evaluation of other kinds of expressions υ, the second kind uses the Scott brackets $[[\upsilon]]^{\lambda}_{\theta}$ for evaluation with various parameters θ and λ. One can think of it as asking, whether θ in combination with λ models υ. Table 2.2 summarizes the notation explained.

Symbol	Meaning
$[[\upsilon]]^{\lambda}_{\theta}$	The evaluation of υ given θ and λ (Scott brackets)
$var(\sigma)$	Function to retrieve *all kind of* variables in σ
$\mathcal{I} \models \sigma$	$\mathcal{I}$ models σ (semantic entailment)
$\top$	Logical true equivalent to 1
$\bot$	Logical false equivalent to 0
$\phi[a/b]$	ϕ but b is replaced with a

Table 2.2: Logical Symbols

As a reference for the reader, Table 2.3 shows the infinite sets defined in this work:
Some usual namespaces (the meaning of a namespace is defined in section 2.2.1) used in this work are defined in Table 2.4 to shorten the examples, which make use of these namespaces.
Finally highlighting is used in pseudocode and when reporting implementation details for the ease of reading. The following concepts are formatted as their counterparts will be in the thesis: MODULE, *Class*, instances, "paramters", "functions", **pseudocodeFunction**, *pseudocode_variable* and `:ressourceIdentifier`.

2.2 Semantic Constraint Validation

The Semantic Web aims to improve the ability of machines to not only represent but understand the data they are processing and storing. At the same time, the data should stay comprehensible to humans, allowing for the cooperation of humans and machines in making decisions. Therefore, technologies are designed to deal with human-understandable data structures, giving their entities a semantic meaning. [7]
This section introduces the W3C recommended technologies to perform constraint validation over structured data carrying a semantic meaning and, therefore, is called semantic constraint validation.

Symbol	Meaning	Definition
$\mathbf{I}$	Infinite set of IRIs	1
$\mathbf{L}$	Infinite set of literals	2
$\mathbf{B}$	Infinite set of blank nodes	3
$\mathbf{G}$	Infinite set of knowledge graphs	6
$\mathbf{V}$	Infinite set of query variables	7
$\mathbf{T}$	Infinite set of triple patterns	8
$\mathbf{M}$	Infinite set of SPARQL solution mappings	9
$\mathbf{P}$	Infinite set of graph patterns	11
$\mathbf{Q}$	Infinite set of SPARQL `SELECT` queries	12
$\overrightarrow{\mathbf{P}}$	Infinite set of property paths	14
$\mathbf{C_P}$	Infinite set of property path constraints	15
$\mathbf{Q_T}$	Infinite set of target queries	16
$\mathbf{S}$	Infinite set of shapes	17
$\mathbf{SN}$	Infinite set of shape schemas	17
$\boldsymbol{\eta}$	Infinite set of sample-to-node mappings	23
$\mathbf{C}$	Infinite set of constraints	24
$\boldsymbol{\Theta}$	Infinite set of model-validation-result functions	32
$\mathbf{E}$	Infinite set of explanation mappings	34
$\boldsymbol{\Gamma}$	Infinite set of grouping functions	42

Table 2.3: Infinite Sets

Prefix	Namespace
	`http://example.org/`
xsd	`http://www.w3.org/2001/XMLSchema#`
rdf	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
sh	`http://www.w3.org/ns/shacl#`

Table 2.4: Prefixes and their associated namespaces

2.2.1 RDF - Encoding Structured Information

RDF – short for Resource Description Framework – is a data model used to describe resources on the web. Originally, RDF was mainly used for metadata of web resources but was then extended to a general machine-interpretable exchange format to encode structured information. Nowadays, RDF has become the W3C standard for publishing and exchanging structured data over the web. The concepts and the abstract syntax presented in this section are taken from [17], unless stated otherwise. The following builds up the concepts of RDF in a bottom-up fashion.
When talking about something in the world in the context of the web, there are two main types of things called resources. First, there are entities that need to be described uniquely in a global context. These types of resources are assigned an IRI.

Definition 1: IRI
IRIs („Internationalized Resource Identifier“) are a super set of URIs („Unique Resource Identifier“), which additionally allow the use of Unicode characters, while identifying resources uniquely. Generally, they are structured hierarchical [6]:

```
<scheme>://<authority><path>?<query>#<fragment>
```

The infinite set of all possible IRIs is denoted by **I**.

A collection of IRIs used in this context is called an RDF vocabulary. Additionally, a set of IRIs with a common prefix `pre` followed by an arbitrary local part `post` can be abbreviated by defining an RDF namespace `n`. Leading to the shorthand `n:post` to be used for the IRI, which is the concatenation of `pre` and `post`.

Example 1: RDF build-in vocabulary

The built-in RDF vocabulary is a set of IRIs, which, for instance, include the definition of the essential data types. A data type needs to be identified uniquely and, therefore, gets a unique IRI assigned. For example, the data type string has the following IRI.

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#langString
```

Clearly, the IRI can be decomposed into the parts mentioned in definition 1. Another IRI belonging to the same vocabulary is used to state that an entity is of a specific class:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

Both IRIs have the common prefix:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

Usually, `rdf` is chosen as a shorthand so that the IRIs can be shortened to `rdf:langString` and `rdf:type`.

Secondly, there might be things like values, dates and items (e.g., items with a data type) defined. These may be used in different contexts without referring to the same thing and are called literals.

Definition 2: Literal
A literal consists of a Unicode string and a data type IRI, as mentioned in example 1. Further, if the data type is `rdf:langString`, then a language tag should be defined. The infinite set of all possible literals is denoted by **L**.

Literals, IRIs, and a third concept named blank nodes are called RDF terms. Though, there may be string literals, which are also used as IRI, the entities denoted with these concepts are distinct and distinguishable, which means that a literal can never be mixed up with an IRI or a blank node.

Definition 3: Blank Node
A blank node is disjoint from the concept of literals and IRIs. Apart from that, the set of blank nodes is arbitrary, and an identifier of a blank node may only be unique in a local representation. The infinite set of all possible blank nodes is denoted by $\mathbf{B}$.

RDF is a graph-based data model. The goal of using the RDF terms as building blocks is to encode structured information. RDF triples are used to represent the information in the form of facts about resources.

Definition 4: RDF triple
A triple (s, p, o) is called an RDF triple, iff $s \in \mathbf{I} \cup \mathbf{B}$, $p \in \mathbf{I}$ and $o \in \mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$. The three components of the triple are called subject s, predicate p, and object o.

A set of triples (i.e., facts about resources) can then be combined into an RDF graph, which is defined as follows:

Definition 5: RDF graph
An RDF graph G is a set of RDF triples as defined in definition 4.

Given the definition, it is clear that an RDF graph $G \subset (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{B})$ can also be understood as a representation of a labeled directed graph [14], which is the statement of the following lemma.

Lemma 1: Two representations for an RDF graph
An RDF graph G_1 as defined in definition 5 is equivalent to the representation of a labeled directed graph $G_2 = (V_G, E_G)$ where $V_G \subset \mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$ are the vertices and $E_G \subseteq V_G \times \mathbf{I} \times V_G$ are the edges between nodes of V_G labeled with predicates of $\mathbf{I}$.

Proof. To show the equivalence, the conversion from G_1 to G_2 and vice versa needs to be shown. Converting G_1 into G_2 is a matter of setting $V_G = \{s \mid (s, p, o) \in G_1\} \cup \{o \mid (s, p, o) \in G_1\}$ and $E_G = G_1$. The reverse has to be done by setting $G_1 = E_G$, such that two nodes s and o are connected with a directed edge $(s, p, o) \in E_G$, if there is a triple (s, p, o). □

A labeled directed graph $G = (V_G, E_G)$, created as above, can be visualized as shown in Figure 1.2. To distinguish literals from the other nodes, they are drawn rectangular, while the other nodes are drawn round. A blank node, therefore, would be an empty round node. Here, a knowledge graph is defined as a labeled directed graph.

Definition 6: Knowledge Graph [14]
A knowledge graph $G = (V_G, E_G)$ is a labeled directed graph with vertices V_G and edges E_G. The set of all possible knowledge graphs is defined to be $\mathbf{G}$

By encoding structured information in the knowledge graph, the labeled edges give the entities in the knowledge graph a meaning. That is, a labeled edge can be seen as a sentence consisting of a subject (the entity), the predicate (the label of the edge), and the

```
<http://example.org/Max> <http://example.org/gender> "male"@en .
<http://example.org/Max> <http://example.org/contact_with> <http://example.org/Maria> .
<http://example.org/Max> <http://example.org/allergic_to> "polyethylene_glycol"@en .
<http://example.org/Max> <http://example.org/allergic_to> "broccoli"@en .
<http://example.org/Max> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://example.org/Person> .
<http://example.org/Maria> <http://example.org/age>
34^^<http://www.w3.org/2001/XMLSchema#integer> .
```

Listing 2.1: N-Triples serialization of a superset of a subgraph of figure 1.2

object (the literal or another entity). Multiple sentences (or triples in the RDF graph) about an entity sets it in a semantic context, giving it a meaning.

2.2.2 An Example RDF Graph using Turtle

Given the abstract RDF data model, it makes sense to define a concrete textual syntax to serialize RDF graphs. The W3C recommended language for serializing RDF graphs is Turtle. Turtle – short for Terse RDF Triple Language – is built on top of the N-Triples serialization, with the goal of a more compact serialization. Thus, every N-Triples serialization is a Turtle serialization. [5] The Turtle serialization as presented here is defined in [5] and the N-Triples serialization is defined in [4].
Turtle is introduced by serializing an extended subset of the RDF graph given in the motivating example in figure 1.2. First, using N-Triples and building on that in a more compact form using Turtle.

N-Triples As the name "N-Triples" suggests, the serialization of an RDF graph with N different triples is just writing down N triples. A dot is used to mark the end of a triple. Each triple consists of a subject, predicate, and object, which are separated by a space. In listing 2.1 the graph consists of 6 triples. The end of each of them is marked with a dot.
An IRI is always enclosed with "<" and ">" to distinguish it from literals. As defined in definition 2, a literal is just a string associated with a data type. Literals without a predefined data type have their data type annotated with a preceding "^^" and the corresponding data type, expressed as IRI. According to definition 4, each predicate of a triple has to be an IRI. This can also be validated in listing 2.1, where each predicate is marked as an IRI. Furthermore, the last triple has as an object the literal 34. As the 34 is meant to be of the data type integer, it is annotated with the IRI `http://www.w3.org/2001/XMLSchema#integer`.
A particular case of a literal is one surrounded with "", which has the data type string predefined (see example 1) and should instead be annotated with a language tag. This is done by extending the literal with "@" and the language tag, which is a Unicode string without spaces. In listing 2.1 the first and the third triple have a literal of data type string as an object, which is additionally marked with the language tag "en".

Turtle Next, listing 2.1 is inspected to introduce some useful shortcuts defined in Turtle. The first thing one notices when inspecting listing 2 is that it does not use the shorthand already defined in section 2.2.1 for IRIs with a common prefix. To be able to use these namespaces, one first needs to define them. This is done by using the "@prefix" keyword followed by the namespace, ":" and the IRI, which is replaced by the namespace.

```
@prefix : <http://example.org/>
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>

:Max :gender "male"@en ;
     :contact_with :Maria ;
     :allergic_to "polyethylene_glycol"@en, "broccoli"@en ;
     a :Person .
:Maria :age 34^^xsd:integer .
```

Listing 2.2: A Turtle serialization of the graph serialized with N-Triples in listing 2.1

In listing 2.2, there are two prefixes defined. The first maps the empty namespace to `http://example.org/` and the second maps "xsd" to `http://www.w3.org/2001/XMLSchema#`. After defining the prefixes, they can be used as already demonstrated in example 1. There is no longer any need to encapsulate them in "<" and ">". Next, there are four triples, starting with `:Max`, which can also be written short in Turtle by allowing predicate object lists. Such a list of triples with a common subject is separated with a semicolon. The analog is also applied to triples with subject and predicate in common by allowing object lists to be separated with a comma. Finally, Turtle also allows writing "a" instead of `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>` to denote the class of an IRI. All these features of Turtle allow the graph, serialized in listing 2.1, with N-triples to be represented in a more compact and readable form in listing 2.2 with Turtle.
Both serializations allow the use of blank nodes. A blank node with name `b` is referred to with `_:b` and can then be used in subject and object position as defined in definition 4. When a blank node occurs only in the subject position of triples `[(_:b,p1,o1),(_:b,p2,o2),...]` and exactly once in another position, there is an additional shorthand in Turtle. In that case, one can replace the blank node occurring only once with open and closing square brackets. Then inside them have the triples referring to the blank node as the subject with a (possibly empty) predicate-object list. This is heavily used when defining SHACL constraints (see section 2.2.4).

2.2.3 Querying RDF Graphs with SPARQL

Let us assume a knowledge graph using RDF with the introduced Turtle serialization as in listing 2.2. However, over time, the knowledge graph has grown to have thousands of triples, and one would like to know the names of female persons who had contact with persons allergic to broccoli. This scenario is a simple example of the information needed, which can be expressed as a machine-interpretable query formulated in SPARQL. SPARQL – short for SPARQL Protocol and Query Language – is the W3C recommended declarative query language to query data from RDF graphs [54].
This section aims to provide the basic knowledge about SPARQL and the involved SPARQL set algebra used, such that the upcoming extension in section 4.3.3 can be clearly defined, and the reader can understand the basic types of queries used in this work. The notation defined here is mainly taken from Pérez et al. [51] if not otherwise specified.
There are four different types of SPARQL queries called `SELECT`, `CONSTRUCT`, `DESCRIBE` and `ASK`. All query types use graph matching to identify which kind of data the author of the query is interested in. A query can be decomposed into a head and a body. While the head defines the query type containing its parameters and, therefore, defines the structure

of the output, the body includes a graph pattern expression.
Here the procedure is again done in a bottom-up fashion and, therefore, starts with defining the components used in graph pattern expressions. The atomic and pairwise distinct components are IRIs, blank nodes, literals, and variables. Three of them are already defined — the one left is the variables, which are referred to in the head of the query as parameters.

> **Definition 7: Query Variables [54]**
> A query variable is a Unicode string without spaces and prefixed with "?". The infinite set of all possible query variables is denoted by $\mathbf{V}$ and is defined to be distinct with $\mathbf{I} \cup \mathbf{L} \cup \mathbf{B}$.

Generally, a variable will be used in a graph pattern as a placeholder. When the graph pattern is evaluated over a knowledge graph, multiple solution mappings might exist. Each solution mapping assigns the variables the values that need to be substituted for the variables in the graph pattern in order for the graph pattern to match a subgraph of the knowledge graph. The value to be substituted can therefore be an IRI, a literal, or a blank node.
The graph pattern will be defined recursively, and the basic building block is always a triple pattern.

> **Definition 8: Triple Pattern [54]**
> A triple pattern is a 3-tuple such that the infinite set of triple patterns $\mathbf{T}$ is a subset of $(\mathbf{I} \cup \mathbf{B} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{V}) \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{B} \cup \mathbf{V})$. Additional the function $var : \mathbf{T} \to \mathcal{P}(\mathbf{V})$ is defined such that var(t) gives the set of variables occurring in $t \in \mathbf{T}$.

Therefore, a triple pattern can be seen as an extended version of an RDF triple, allowing a variable in each position. At this point, it is helpful to have an idea of what a graph pattern will be. The following example is intended to give this intuition.

Example 2: Visualization of a simple graph pattern

Let us return to the example query mentioned above, which should return the names of the female persons who had contact with persons allergic to broccoli. Analyzing the query gives three variables. The first one will be assigned a node referring to a female person, the second one to a node of the person allergic to broccoli, and the last will refer to the literal, which is the name of the female person. The query only asks for the name, but the other variables are still needed to define the graph pattern. The conjunction of six triple patterns is used to connect the variables. Though the conjunction is not yet defined, it will still make intuitive sense. Figure 2.1 shows the graph pattern visually using the notation explained in section 2.2.1 with the addition of variables. This makes sense because a conjunction of triple patterns is just a set of RDF triples, which allows having variables in each position. A node representing a variable is of a diamond form. If it is a variable that will be returned finally, the node is double lined.
Evaluating the graph pattern against the knowledge graph in figure 2.2 will give a set of solution mappings. In this example the set consists of one solution mapping of ?name to "Maria". Figure 2.2 addi-

tionally depicts the evaluation by highlighting the subgraph, matched by the graph pattern from figure 2.1, used to produce a solution mapping.

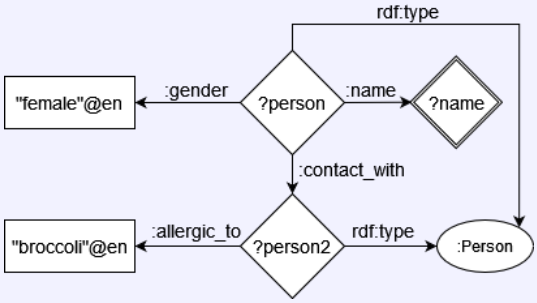


Figure 2.1: An Example Graph Pattern. It will query the names of female persons, which had contact to persons allergic to broccoli

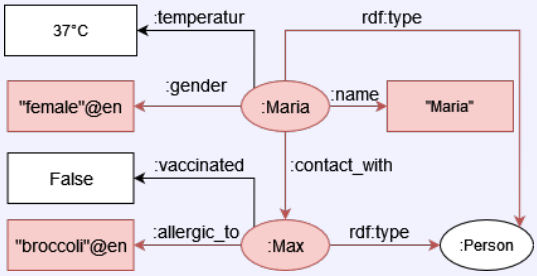


Figure 2.2: A small Knowledge Graph. The red subgraph is matched by the graph pattern from figure 2.1 to produce a solution mapping.

As in the example, evaluating a graph pattern expression P over a knowledge graph G gives a set of solution mappings. Each solution mapping maps at least the variables used outside of an `OPTIONAL` pattern to RDF terms. The `OPTIONAL` pattern will be defined together with the graph pattern expression in definition 11. A single solution mapping is defined as follows.

Definition 9: SPARQL Solution Mapping [54]
A solution mapping is a partial function $\mu : \mathbf{V} \to (\mathbf{B} \cup \mathbf{L} \cup \mathbf{I})$, which maps a subset of the infinite set of variables $\mathbf{V}$ to RDF terms. Two mappings μ_1 and μ_2 are called compatible, denoted with $\mu_1 \sim \mu_2$, when $\forall v \in (\mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_1))\ \mu_1(v) = \mu_2(v)$ is true. In that case the union of two mappings μ_1 and μ_2 is defined as $\mu_1 \cup \mu_2$, which is a shorthand for $\mathcal{G}_{\mu_1} \cup \mathcal{G}_{\mu_2}$. The infinite set of all possible SPARQL solution mappings is defined to be $\mathbf{M}$.

Next, algebraic operations over the sets of solution mappings and filter conditions on a set of solution mappings are needed to recursively extend the notion of the graph pattern with further operators. A filter condition is a logical expression $R \in \mathcal{L}_{\mathbb{S}}$ with $\mathbb{S} = (\mathbf{B} \cup \mathbf{L} \cup \mathbf{I}; <; +, *; 0, 1)$, which makes use of variables from $\mathbf{V}$. There are further functions defined and, moreover, the functions and relations in the structure can only be applied to specific types of literals. IRIs can only be checked for equality as there is no order defined. All this is specified exactly in [54]. Nevertheless, a filter condition R can

be evaluated via $(\mathbb{S}, \mu) \models R$ using the solution mapping μ as an assignment.
The algebraic operations over sets of solution mappings are the content of the following definition.

Definition 10: Algebraic Operations over sets of solution mappings (extended version of [62])
Given two sets of solution mappings $\Omega_1, \Omega_2 \subset \mathbf{M}$, a set of variables $v \subset \mathbf{V}$, a filter condition $R \in \mathcal{L}_{\mathbb{S}}$ and a function used for renaming of variables $\nabla : \mathbf{V} \to \mathbf{V}$ the following operations are defined:

operation	meaning
$\pi_v(\Omega_1)$	$\{\mu_1 \mid \exists \mu_2 : \mu_1 \cup \mu_2 \in \Omega_1 \wedge \mathrm{dom}(\mu_1) \subseteq v \wedge (\mathrm{dom}(\mu_2) \cap v) = \emptyset\}$
$\sigma_R(\Omega_1)$	$\{\mu \in \Omega_1 \mid (\mathbb{S}, \mu) \models R\}$
$\Omega_1 \cup \Omega_2$	$\{\mu \mid \mu \in \Omega_1 \vee \mu \in \Omega_2\}$
$\Omega_1 \bowtie \Omega_2$	$\{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 : \mu_1 \sim \mu_2\}$
$\Omega_1 \setminus \Omega_2$	$\{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2 : \neg(\mu_1 \sim \mu_2)\}$
$\rho(\nabla, \Omega_1)$	$\{\nabla \circ \mu \mid \mu \in \Omega_1\}$

Compared to Schmidt et al., ρ is added to the algebraic operations to allow for variable renamings in solution mappings. $\mathbb{S}$ is the structure $(\mathbf{B} \cup \mathbf{L} \cup \mathbf{I}; <; +, *; 0, 1)$

Before the notion of a graph pattern and its evaluation can be appropriately defined, there is the substitution of a solution mapping into the triple pattern left. A thing which was already implicitly done in example 2 to identify the red subgraph. To do that formally, μ is overloaded to be usable as a function $\mu : \mathbf{T} \to (\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{L} \cup \mathbf{B})$, such that $\mu(t)$ replaces all occurrences of $v \in \mathrm{dom}(\mu)$ in t with $\mu(v)$.
Now the graph pattern expression and their evaluation can be formally defined:

Definition 11: Graph Pattern Expression and their Evaluation (similar to [51, 62])
Let $t \in \mathbf{T}$ be a triple pattern as defined in definition 8. The following table gives the recursive definition of a graph pattern P, the SPARQL syntax and the evaluation of the graph pattern into the set of solution mappings. The infinite set of all graph patterns is denoted as $\mathbf{P}$. The evaluation of a graph pattern $P \in \mathbf{P}$ over the knowledge graph $G \in \mathbf{G}$ is written as a function $[[P]]_G : \mathbf{P} \times \mathbf{G} \to \mathcal{P}(\mathbf{M})$ and gives a set of SPARQL mappings as defined in definition 9. The recursive definition of a graph pattern assumes that P_1, P_2 are graph patterns, R is a filter condition, $v \subset \mathbf{V}$ and ∇ a renaming function as in definition 10.

graph pattern P	syntax	evaluation $[[P]]_G$
$t := (s, p, o)$	$s\ p\ o$	$\{\mu \mid \mathrm{dom}(\mu) = var(t) \wedge \mu(t) \in E_G\}$
$(P_1$ `AND` $P_2)$	$\{P_1\ .\ P_2\}$	$[[P_1]]_G \bowtie [[P_2]]_G$
$(P_1$ `UNION` $P_2)$	$\{\{P_1\}$ `UNION` $\{P_2\}\}$	$[[P_1]]_G \cup [[P_2]]_G$
$(P_1$ `OPT` $P_2)$	$\{\{P_1\}$ `OPTIONAL` $\{P_2\}\}$	$[[P_1]]_G \mathbin{⟕} [[P_2]]_G$
$(P_1$ `FILTER` $R)$	$\{\{P_1\}$ `FILTER` $R\}$	$\sigma_R([[P_1]]_G)$
`SELECT`(v, ∇, P_1)	$\{$`SELECT` v' `WHERE` $\{P_1\}\}$	$\rho(\nabla, \pi_v([[P_1]]_G)$

v' denotes v but written as a list separated with spaces. Additionally, there is the

option to implicitly define ∇ with the AS keyword used in-place between the renamed variable var $\in v$ and the new variable new $\in \mathbf{V}$. All other variables not included on the left-hand side of the AS keyword will be mapped to themself.

Turtle is used for serializing the graph patterns in $\mathbf{P}$. Therefore, serializing a graph pattern is just a matter of serializing the triple patterns with Turtle and writing down the other operators using the syntax as defined. For example, a conjunction ("AND") of triple patterns is denoted with a dot. The other short hands presented as Turtle's features to shorten the conjunctions of RDF triples can also be used. There are further rules of precedence, association, syntax, and operators (like the concept of grouping) defined in SPARQL, which abstract away from the logical core. These allow to relax the requirements for the brackets in definition 11 and can be found in [54].
Now that the basic syntax of the body of a SPARQL query is defined by serializing graph patterns, one has to define the head. In this work, only SELECT queries are used, such that it will be enough to define this query type.

Definition 12: SPARQL SELECT Query
A SPARQL SELECT query Q is a graph pattern P of the form SELECT$(var(Q), \nabla, P')$ where $P' \in \mathbf{P}$ and ∇ a renaming function as defined in 10. P uses the syntax as defined for the SELECT clause in definition 11 but without the outermost brackets. The infinite set of SPARQL SELECT queries is denoted with $\mathbf{Q}$.

Like Turtle, SPARQL allows the definition of prefixes but instead of "@prefix", „PREFIX" is used. To show the algebra and the SPARQL syntax in action, the following example continues example 2.

Example 3: SPARQL Algebra and syntax applied

Given the visualization, in example 2 the triple patterns lying at the heart of the graph pattern and needed to specify the query can be identified. All of the triple patterns should apply simultaneously, implying the need for the conjunction of the triple patterns. In the end, only the names of the persons should be retrieved, such that only ?name is projected. There is no renaming of variables needed here. Therefore, the identity function id mapping each variable to itself is used.

SELECT(?name,id,((((((?person, :gender, "female@en")
AND (?person, rdf:type, :Person))
AND (?person, :name, ?name))
AND (?person, :contact_with, ?person2))
AND (?person2, :allergic_to, "broccoli"@en))
AND (?person2, rdf:type, :Person)))

Using the left associative property of AND, the algebra can be transformed into the SPARQL syntax.

```
PREFIX : <http://example.org/>
SELECT ?name WHERE {
```

```
        ?person :gender ''female''@en .
        ?person a :Person
        ?person :name ?name .
        ?person :contact_with ?person2 .
        ?person2 :allergic_to ''broccoli''@en .
        ?person2 a :Person
    }
```

Listing 2.3: Algebra transformed into SPARQL syntax

Finally, given a SPARQL `SELECT` query the evaluation of the query is defined analogue to the evaluation of a graph pattern.

Definition 13: Execution of Q over G [14]
The execution of the SPARQL `SELECT` query $Q \in \mathbf{Q}$ over the knowledge graph $G \in \mathbf{G}$ is written as a function $[[Q]]_G : \mathbf{Q} \times \mathbf{G} \rightarrow \mathcal{P}(\mathbf{M})$ and gives a set of SPARQL mappings. Each SPARQL mapping assigns the set of variables $var(Q) \subset \mathbf{V}$ to RDF terms.

It is essential to note that a SPARQL endpoint typically performs the process of executing a query over a knowledge graph. The SPARQL endpoint exposes a knowledge graph by answering SPARQL queries using the knowledge graph. Most knowledge graphs are exclusively accessible using SPARQL queries, which consolidates the use of SPARQL as the primary medium to extract information from a knowledge graph.

2.2.4 Validating SHACL Constraints over an SPARQL Endpoint

At this point, a knowledge graph might be made available by a SPARQL endpoint and can then be queried using SPARQL queries. However, as mentioned in the introduction, the goal is to exploit the structure of the knowledge graph to get further insights into machine learning models by using semantic constraint validation. SHACL – short for Shapes Constraint Language – is the W3C recommended language to describe constraints and validate RDF graphs against them [38]. The approach is described in terms of the abstract syntax for the SHACL core constraint components, as introduced by Cormen et al. [15]. The notation has the advantage of being compact, allowing recursive schemas, and being translatable into the SHACL language.
Let us return to the example knowledge graph about the properties of people having contact with each other. The government might have passed a law that states that everyone who is not vaccinated can only have contact with up to five vaccinated strangers. A vaccinated stranger is defined as another person who is vaccinated and not pregnant. Now, the knowledge graph can be checked against those constraints to enforce the law. The example constraint makes the distinction between two groups of people. Here, these two groups are described by shapes. The first shape might be called, `:risk_group` and the second `:vaccinated_stranger`. A set of shapes is called a shape schema. A shape comes with a logical expression, which enforces constraints. Further components of the shape are the name, which identifies the shape, and the target definition, which defines the scope for the shape in the knowledge graph.
A constraint uses certain properties of the entity, which property paths can identify.

Definition 14: Property Path (general concept as in [38])
A property path p defines a path between two nodes in a knowledge graph. The evaluation of p over $G \in \mathbf{G}$ is done with a function $\texttt{path}(p, G)$ to give a SPARQL mapping μ with $\text{dom}(\mu) = \{?\text{s}_p, ?\text{e}_p\}$. The variable $?\text{s}_p$ represents the start node and the variable $?\text{e}_p$ the end node of the path. The infinite set of property paths is denoted with $\overrightarrow{\mathbf{P}}$. The recursive definition of a property path is given in the table below, which uses $I \in \mathbf{I}$, $p_1, ..., p_n \in \overrightarrow{\mathbf{P}}$ and the renaming functions

$$\begin{aligned}\nabla_{\text{seq}} =&\{(?\text{s}_{p_1} \mapsto ?\text{s}_p), (?\text{e}_{p_n} \mapsto ?\text{e}_p)\} \cup \\ &\{(?\text{e}_{p_i} \mapsto ?\text{s}_{p_{i+1}}) \mid \forall i \in [1, 2, ..., n-1]\} \cup \\ &\{(?\text{v} \mapsto ?\text{v}) \mid \forall ?\text{v} \in (\mathbf{V} \setminus (\{?\text{s}_{p_1}\} \cup \{?\text{e}_{p_i} \mid \forall i \in [1, 2, ..., n]\}))\}\end{aligned}$$

$$\nabla_{\text{inv}} =\{(?\text{s}_{p_1} \mapsto ?\text{e}_p), (?\text{e}_{p_1} \mapsto ?\text{s}_p)\} \cup \{(?\text{v} \mapsto ?\text{v}) \mid \forall ?\text{v} \in (\mathbf{V} \setminus \{?\text{s}_{p_1}, ?\text{e}_{p_1}\})\}$$

name	property path p	evaluation $\texttt{path}(p, G)$
Predicate paths	I	$[[(?\text{s}_p, I, ?\text{e}_p)]]_G$
Inverse paths	$\hat{}p_1$	$\rho(\nabla_{\text{inv}}, \texttt{path}(p_1, G))$
Sequence paths	$p_1/p_2/.../p_n$	$\pi_{?\text{s}_p, ?\text{e}_p}(\rho(\nabla_{\text{seq}}, \bowtie_{i \in [1,2,...,n]} \texttt{path}(p_i, G)))$

To abbreviate the notation, let $[[p]]_G$ be the shorthand for

$$\{(v_s, v_e) \mid \{(?\text{s}_p \mapsto v_s), (?\text{e}_p \mapsto v_e)\} \in \mathcal{G}_{path(p,G)}\}$$

Using the concept of property paths, constraints can be defined:

Definition 15: Property Path Constraint [15]
A property path constraint is a logical expression ϕ included in the language that contains the words of the following grammar defined in the Backus-Naur-Form [36]:

$$\begin{aligned}\text{c} &:= \top \mid s \mid I \mid \geq_n p_1.\phi \mid \text{EQ}(p_1, p_2) \\ \phi &:= \neg\phi \mid \phi \wedge \phi \mid \text{c}\end{aligned}$$

where s is a shape name, $I \in \mathbf{I}$, $p_1, p_2 \in \overrightarrow{\mathbf{P}}$ and $n \in \mathbb{N} \cup \{0\}$. The infinite set of all possible constraints is denoted with $\mathbf{C_P}$.

Therefore, a constraint is a logical expression ϕ, consisting of terms c. These terms can be split into two (possibly empty) groups. The first group is called inter-shape constraints and includes all terms referring to shape names denoted with s in definition 15. The second group, including the rest, is called intra-shape constraints. Hence, the intra-shape constraints describe the entity based on its properties, and the inter-constraints allow entities to depend on each other.
During the validation of a knowledge graph, each shape is checked against a possibly empty set of target nodes. A target query identifies these target nodes.

Definition 16: Target Query [15]
A target query Q_{T_s} is a SPARQL `SELECT` query with $|\text{dom}([[Q_{T_s}]]_G)| = 1$ for all $G \in \mathbf{G}$ defined for a shape named s. The infinite set of all possible target queries is $\mathbf{Q_T} \subset \mathbf{Q}$.

Without loss of generality, let $\Omega = [[Q_{T_s}]]_G$ be the set of solution mappings of a target query $Q_{T_s} \in \mathbf{Q_T}$ with $\forall \mu \in \Omega \ \text{dom}(\mu) = \{?x\}$ and $?x \in \mathbf{V}$ an arbitrary variable. In that case, the notation of the set of solution mappings can be shortened to a set of solution values $\{\mu(?x) \mid \mu \in \Omega\}$. Therefore, to check whether there is a solution mapping, $\{(?x \mapsto v_1)\} \in \mathcal{G}_\Omega$ the shorthand $v_1 \in \Omega$ can be used.
Given all the components, the notion of a shape schema can be defined.

Definition 17: Shape Schema [15]
A shape schema is a triple $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ consisting of a set of shape names S, a function returning a target query $\texttt{TARG} : S \to \mathbf{Q_T}$ and a function returning a property path constraint $\texttt{DEF} : S \to \mathbf{C_P}$ for each shape $s \in S$. The components of the shape schema $\mathcal{S}$ can be accessed by $\mathcal{S}.S$, $\mathcal{S}.\texttt{TARG}$ and $\mathcal{S}.\texttt{DEF}$. The infinite set of all possible shapes is denoted with $\mathbf{S}$ and the infinite set of all possible shape schemas with $\mathbf{SN}$

Example 4: Shape Schema with references to the SHACL language

This example continues the example in the introductory text of this section by defining the shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ according to the definition above. As mentioned, there will be two shapes. The first one called `:risk_group` and the second one called `:vaccinated_stranger`. Therefore, it is

$$S = \{\texttt{:risk\_group}, \texttt{:vaccinated\_stranger}\}$$

The attentive reader might notice the colons in front of the names, which typically indicate an IRI with an empty prefix in the Turtle syntax. That is because in the SHACL language (not the abstraction used here), a shape is identified by an IRI of the class `sh:NodeShape` and a shape schema is basically a knowledge graph serialized with Turtle containing a set of IRIs annotated with RDF terms defined in the SHACL namespace `http://www.w3.org/ns/shacl#` with the shorthand `sh` [38].
Next, the `TARG` function needs to be defined for S. As every person can be a `:vaccinated_stranger` the shape has as scope the nodes of the class person. A person is in the scope of `:risk_group` if the person is not vaccinated. That is because the law should be enforced, and the goal is to identify the persons in the scope of `:risk_group` with more than five contacts with persons of type `:vaccinated_stranger`.

```
PREFIX : <http://example.org/>
SELECT ?x WHERE {
    ?x a :Person .
    ?x :vaccinated False
}
```

Listing 2.4: `TARG(:risk_group)`

```
PREFIX : <http://example.org/>
SELECT ?x WHERE {
    ?x a :Person
}
```

Listing 2.5: `TARG(:vaccinated_stranger)`

In the SHACL language, there is no concept of a target query. Instead, there are RDF triples indicating the target nodes explicitly, by class or implicitly as subject or object of a specific predicate [38]. Therefore, the targets would be defined by two RDF triples $(\texttt{:risk\_group}, \texttt{sh:targetClass}, \texttt{:NotVaccinatedPerson})$ and $(\texttt{:vaccinated\_stranger}, \texttt{sh:targetClass}, \texttt{:Person})$.
Finally, the function DEF giving the property path constraints for each shape $s \in S$ needs to be defined. To conform with :risk_group, a person cannot have more than five contacts with :vaccinated_stranger.

$$\texttt{DEF}(\texttt{:risk\_group}) = \neg(\geq_6 \texttt{:contact\_with.:vaccinated\_stranger})$$

A :vaccinated_stranger has to be vaccinated and cannot be pregnant.

$$\texttt{DEF}(\texttt{:vaccinated\_stranger}) = (\geq_1 \texttt{:vaccinated.True}) \wedge (\geq_1 \texttt{:pregnant.False})$$

When using the SHACL language to express constraints, nodes of the class sh:PropertyShape are used. These can be characterized by various properties and have to be connected to a sh:NodeShape via sh:property. In this example, the full SHACL shape schema expressed with Turtle would be:

```
PREFIX : <http://example.org/>
PREFIX sh: <http://www.w3.org/ns/shacl\#>
:risk_group
    a sh:NodeShape ;
    sh:targetClass :NotVaccinatedPerson ;
    sh:property [
        sh:path :contact_with ;
        sh:qualifiedValueShape :vaccinated_stranger ;
        sh:qualifiedMaxCount 5 .
    ] .
:vaccinated_stranger
    a sh:NodeShape ;
    sh:targetClass :Person ;
    sh:property [
        sh:path :vaccinated ;
        sh:hasValue "True" ;
        sh:minCount 1 .
    ] ;
    sh:property [
        sh:path :pregnant ;
        sh:hasValue "False" ;
        sh:minCount 1 .
    ] .
```

Listing 2.6: The Example SHACL Shape Schema Serialized with Turtle

The task of validating a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ against a knowledge graph G is

to find a faithful assignment σ. An assignment σ is a set of atoms of the form $s(v)$, where $s \in S$ is a shape and v is a node $v \in V_G$. If $s(v) \in \sigma$ then v is called valid with respect to s according to σ and $\neg s(v) \in \sigma$ means that v violates s according to σ. A valid assignment can only contain $s(v)$ or $\neg s(v)$ and not both. However, it has to contain $s(v)$ or $\neg s(v)$ for each target $v \in \bigcup_{s \in S} [[\texttt{TARG}(s)]]_G$. Additionally, a valid assignment σ is called faithful if for all $s(v) \in \sigma$, v *indeed* satisfies s and for all $\neg s(v) \in \sigma$, v *indeed* does not satisfy s. The satisfaction is measured with respect to the assignment. Therefore, there can be a faithful assignment σ with $s(v) \in \sigma$ and another one σ_2 with $s(v) \notin \sigma_2$. [15]
What remains is to evaluate whether a node v *indeed* satisfies a shape $s \in S$ of a given knowledge graph G and an assignment σ. But that is something which can be reduced to the problem of evaluating a property path constraint c given by $\texttt{DEF}(s)$ as defined in definition 15. The evaluation is denoted with $[[c]]_{G,v,\sigma}$ and is done inductively over the components of c. Table 2.3 includes the non-trivial part of the evaluation. The rest is evaluated according to a normal logical expression.

Constraint C	Evaluation $[[C]]_{G,v,\sigma}$
s	$\top$ if $s(v) \in \sigma$ else $\bot$
I	$\top$ if $v = I$ else $\bot$
$\geq_n p_1.\phi$	$\top$ if $\lvert\{v' \mid (v,v') \in [[p]]_G \text{ and } [[\phi]]_{G,v',\sigma} = \top\}\rvert \geq n$ else $\bot$
$\text{EQ}(p_1, p_2)$	$\top$ if $\{v' \mid (v,v') \in [[p_1]]_G\} = \{v' \mid (v,v') \in [[p_2]]_G\}$ else $\bot$

Figure 2.3: Evaluation of a constraint C given a node v and a assignment σ [15]

Therefore, a node v in a knowledge graph G is *indeed* valid with respect to a shape s in a shape schema $\mathcal{S}$ if $[[\texttt{DEF}(s)]]_{G,v,\sigma} = \top$ is true.
An engine producing a faithful assignment σ for a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ given a knowledge graph G can only use SPARQL queries to get information about the knowledge graph. Such an engine is shown in Algorithm 1, which is now described. In a first step, for each shape, s the target nodes $[[\texttt{TARG}(s)]]_G$ of the shapes are retrieved. Given the instances to be checked against each shape s, the constraints encoded in $\texttt{DEF}(s)$ are converted into SPARQL queries. The solution mappings retrieved, yield a set of rules involving the atoms $s(v)$ with $v \in [[\texttt{TARG}(s)]]_G$ and atoms $s'(v')$ connected to $s(v)$ by inter-shape constraints. (line 2 - 5). This set of rules can then be solved by a SAT solver to get the faithful assignment σ (line 6). Finally, it is useful to convert the assignment produced into a partial function as defined below (line 7 - 13).

Definition 18: Entity Validation [58]
The result of a SHACL validation process for a node $v \in V_G$ and a shape $s \in S$ given a knowledge graph G and a SHACL shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ is represented as a partial function:

$$\text{validate}:(\mathbf{B} \cup \mathbf{I}) \times \mathbf{S} \times \mathbf{G} \rightarrow \{\top, \bot\}$$

2.3 Explainable AI

AI is "the study and design of intelligent agents, [which] [...] take actions that maximize [their] chances of success" [60]. When this definition of AI is extended to explainable

Algorithm 1 Pseudocode of a SHACL engine

function RUNSHACLENGINE(Shape Schema $\mathcal{S}$, Knowledge Graph G)
  $rules \leftarrow \emptyset$
  for each $s \in S$ **do**
    $rules \leftarrow rules \cup$ **materialize**$(S.\texttt{DEF}(s), G)$
  end for
  $\sigma \leftarrow$ **SAT_Solver**$(rules)$
  $validate \leftarrow \emptyset$
  for each $s(v) \in \sigma$ **do**
    $validate \leftarrow validate \cup \{(s, S, G) \mapsto \top\}$
  end for
  for each $\neg s(v) \in \sigma$ **do**
    $validate \leftarrow validate \cup \{(s, S, G) \mapsto \bot\}$
  end for
  return $validate$
end function

AI, the goal is to produce agents whose behavior can be understood by humans. "[It allows] users to comprehend and trust the results and outputs created by machine learning algorithms" [16]. "[That is] explainability [is the capability of the model to] summarize the reasons [for their] behavior, gain the trust of the users, or produce insights about the causes of their decisions" [27]. This is in contrast to the interpretability of AI, which only requires to "[...] describe the internals of a system in a way that is understandable to humans" [27]. To tackle explainable AI in the context of semantic constraint validation, first, the most needed machine learning concepts are introduced, followed by a more specific introduction to the covered models.

2.3.1 Basics of Machine Learning

Machine learning aims at providing algorithms that enable computers to learn from data. A good definition of learning in this context is given by Mitchell [46], who states that learning from experience E with respect to some class of tasks T and performance measure P is to perform better at T (as measured by P) with more experience E. As this definition of learning lies in the heart of machine learning, the short introduction is structured based on it. Usually, machine learning is applied for tasks where the direct design of an algorithm is too difficult or where it is not even known how the task can be solved. In analogy to a functional algorithm, which gets an input and produces an output, a machine learning algorithm produces a machine learning model, which can then transform an input into an output. Therefore, a task is to transform an input into an output according to a specific eventually unknown schema $ot(.)$. [28]

Definition 19: Task
A task is described in terms of what a machine learning model will get as input and which type of output the user is interested in. Therefore, it is a tuple $(\mathbb{I}, \mathbb{T}, ot(.))$, where $\mathbb{I}$ denotes the type of the input, $\mathbb{T}$ denotes the type of the output, and $ot : \mathbb{I} \rightarrow \mathbb{T}$ the schema (represented as a function) to be learned by the machine learning model. $ot(.)$ is usually referred to as objective truth or the function which returns the optimal prediction.

As there is an endless number of algorithms, there is also a variety of task types, restricting or making specific assumptions about the input and the output of the task. Two common tasks in machine learning are classification and regression. Classification restricts the output type to be categorical. Hence, the output is a set of labels, and the goal is to assign an input a label. Therefore, an example for such a task would be $(\mathbb{R}^n, \{\text{Setosa}, \text{Versicolour}, \text{Virginica}\}, ot(.))$, which refers to the classification performed on the basis of the popular iris dataset [23]. In that case, $ot(.)$ would be the function always estimating the correct iris given the sepal and petal, length, and width. Regression restricts the output to be continuous and is, therefore, a numerical value, e.g., $\mathbb{T} = \mathbb{R}$). A machine learning algorithm is then used to learn from a tuple of examples by estimating the parameters of a machine learning model.

Definition 20: Machine Learning Model
A machine learning model $M_\theta : \mathbb{I} \to \mathbb{T}$ is a function parameterized with learnable parameters θ solving a task $(\mathbb{I}, \mathbb{T}, ot(.))$, by mapping problem instances to targets.

Roughly, there are two types of machine learning algorithms: the supervised ones and the unsupervised ones. The algorithm type is chosen depending on the available experience to solve the task. The unsupervised ones are only given a tuple of $N \in \mathbb{N}$ problem instances D of the task $(\mathbb{I}, \mathbb{T}, ot(.))$ and, therefore, $D \in \mathbb{I}^N$. In contrast, the supervised algorithms have as experience D the problem instances, but now annotated with the matching targets and, therefore, $D \in (\mathbb{I} \times \mathbb{T})^N$. D here denotes a dataset and is a tuple, so an example can be identified by its position in the tuple. Therefore, a dataset is defined as follows:

Definition 21: Dataset
Given a task $(\mathbb{I}, \mathbb{T}, ot(.))$, a dataset D is a tuple of samples. The dataset is defined depending on the type of the algorithm. If it is a supervised algorithm, $D = ((\mathbf{x}_i, t_i) \mid i \in [1, ..., N])$ is a tuple of $N \in \mathbb{N}$ samples. Each sample consists of a problem instance $\mathbf{x}_i$ and a target $t_i \in \mathbb{T}$. If it is an unsupervised algorithm, $D = (\mathbf{x}_i \mid i \in [1, ..., N])$ is a tuple of $N \in \mathbb{N}$ samples. In both cases, the problem instance is a feature vector $\mathbf{x}_i = (x_{i,1}, x_{i,2}, ..., x_{i,K})^T \in \mathbb{I}$ with $K \in \mathbb{N}$ features.

The machine learning algorithm discussed above to produce a machine learning model M_θ given a dataset D is called the inducer of the model. The inducer is defined below.

Definition 22: Inducer
Given a task $(\mathbb{I}, \mathbb{T}, ot(.))$, a machine learning algorithm is a function $\mathcal{I}_\zeta : (\mathbb{I} \times \mathbb{T})^N \mapsto \mathbb{F}_{\mathcal{I}_\zeta}$. $\mathcal{I}_\zeta$ is called inducer with hyperparameters ζ and the hypothesis space, $\mathbb{F}_{\mathcal{I}_\zeta} \subset \mathcal{P}(\mathbb{I} \times \mathbb{T})$ which is the space of functions, which the produced machine learning model can represent.

Hyperparameters are the parameters used to configure the inducer. Typical examples of hyperparameters are the learning rate and the maximum depth of a decision tree.

2.3.2 Learning Algorithms and Performance Measures

The inducer is specific to the model. However, in common, an inducer tries to minimize the expected loss of the model it produces. The expected loss is a property of the model

solving a task and depends on the performance measure P defined for the task. As will be seen, the expected loss is difficult to determine directly but will help understand the metrics used in machine learning. [8]
This section is written based on [8] but emphasizes the difference between the objective truth $ot(.)$ and ground truth $gt(.)$ and uses the definitions provided in the last section. The exact formulas used can be found in [8].
Let us assume a task $(\mathbb{I}, \mathbb{T}, ot(.))$. The data generation process will be the first step in producing a trained machine learning model to tackle the task. The process generates the dataset by sampling the objective truth $ot(.)$ via experiments (there is a data-generation distribution [28]). Therefore the process abstracts from the objective truth $ot(.)$ to the ground truth $gt(.)$. As $gt(.)$ only refers to the samples taken it turns out to be a partial function $gt : \mathbb{I} \to \mathbb{T}$. In this work, the ground truth refers to the samples in the dataset and, therefore, it holds

$$(\mathbf{x}_i, t_i) \in D \iff (\mathbf{x}_i \mapsto t_i) \in \mathcal{G}_{gt}$$

As random fluctuations might be involved in the experiments, the dataset can already contain errors known as noise.
Usually, only a restricted number of samples are available when the inducer creates the trained model. Nevertheless, the model should be able to approximate the objective truth $ot(.)$ as well as possible for all possible inputs in $\mathbb{I}$. The capability of transferring the knowledge seen to new inputs is called generalization. The generalization capabilities of the model are measured using a so-called train-test split. The train-test split splits up the dataset into two disjoint subsets D_{train} and D_{test}. Then D_{train} should be used to train the model and D_{test} to approximate the capabilities of the model, to generalize using the performance measure P.
Generalization can be further understood by decomposing the expected loss into the bias, the variance, and the noise. The origin of the noise is the sampling process, so the bias and the variance still need to be explained.
Both terms are defined with respect to a model trained on different datasets $D_1, D_2, \ldots$ but for the same task. That is, an inducer $\mathcal{I}_\zeta$ creates the models via $M_{\theta_1} = \mathcal{I}_\zeta(D_1), M_{\theta_2} = \mathcal{I}_\zeta(D_2), \ldots$. The bias is the extent to which the average prediction of the models deviates from the objective truth, and variance is the extent to which the models' predictions vary around the average prediction. If the variance term dominates, that might imply that the model mainly learned the characteristics of the dataset used during training and, therefore, generalizes poorly to unseen data. On the other hand, if the bias term dominates, that might be a sign that the model is generally not capable of learning the needed structures or the inducer with the given hyperparameters can not learn the required model M as $M \notin \mathcal{F}_{\mathcal{I}_\zeta}$. Therefore, a well-trained model usually needs a balanced bias and variance: The model is biased enough to generalize well but sufficiently complex to adopt the dataset's structure.
The following example shows that in some cases approximating the expected loss of the model M_θ via P on D_{test} is enough. That is important as determining the concrete bias-variance decomposition often cannot be done as the required amount of data is unavailable and $ot(.)$ is not accessible. Furthermore, the example visualizes the concepts of bias, variance, noise, objective truth, and ground truth.

Example 5: Estimating the required depth of a decision tree to approximate the sigmoid function

In this example, the sigmoid function $ot(x) = \frac{1}{1+e^{-x}}$ should be approximated by regression. Here a decision tree for regression is used, but any other model used for regression could be used. Most important is that the inducer used to train the model needs to have a hyperparameter, which limits the complexity of the model. A study is done to find the maximum depth of the decision tree such that the model generalizes well. Following the procedure described to explain the bias and the variance for each possible depth, 100 decision trees are trained on a dataset D_j sampled from $ot(.)$. Each dataset has $N = 1000$ samples and is divided by a train-test split. During the process, for each depth, the average squared loss using a distinct test dataset is calculated:

$$\frac{1}{100 * N} \sum_{j=1}^{100} \sum_{i=1}^{N} (M_{\theta_j}(\mathbf{x}_i) - t_i)^2$$

This is done as the performance measure used here is the mean squared error. The decision trees are used to estimate the bias, the variance, and the noise as the components of the expected loss. This decomposition and the average test error per depth are used to create Figure 2.4.

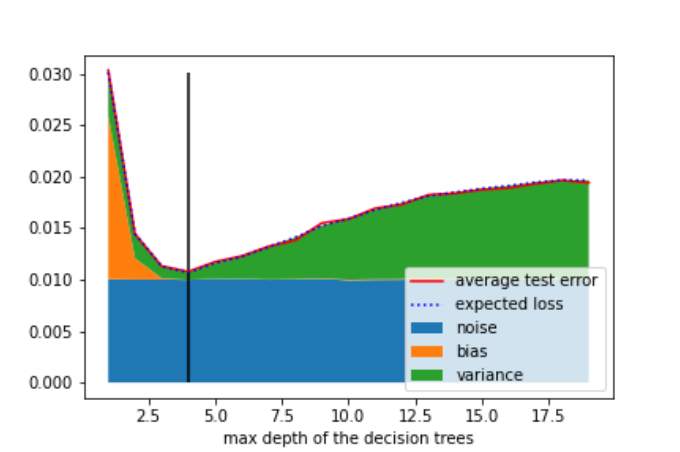
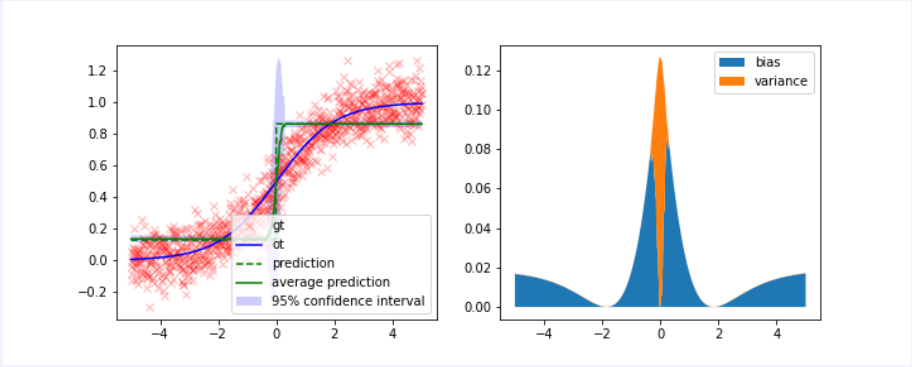


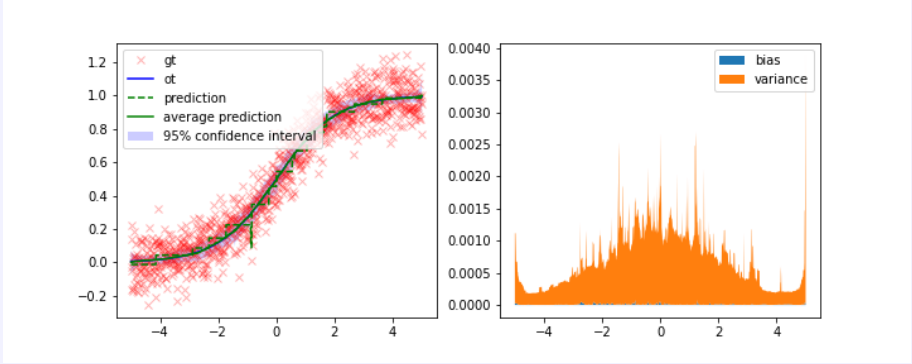
Figure 2.4: Trading of bias vs. variance to estimate the maximal depth of a decision tree. Estimation is based on 100 decision trees trained on new sampled datasets with 1000 samples.

First of all, one can see that the average test error strongly correlates with the expected loss. Although, the expected loss depends on $ot(.)$ and the average test error on $gt(.)$. To calculate the bias-variance decomposition from [8] $h(x)$, representing the optimal regression function is replaced by $ot(x)$.
The vertical line marks the depth of 4, corresponding to the lowest expected loss and, therefore, the best possible value for the max depth hyperparameter. Next, the bias term dominates in the region of a too simple model, and the variance term dominates in the regions of a too complex model. Figure 2.5 visualize the model predictions per max depth and the bias-variance decomposition per input value. For example, in the Figure for a maximum depth of 8, one can see that the model has learned unnecessarily complex structures that

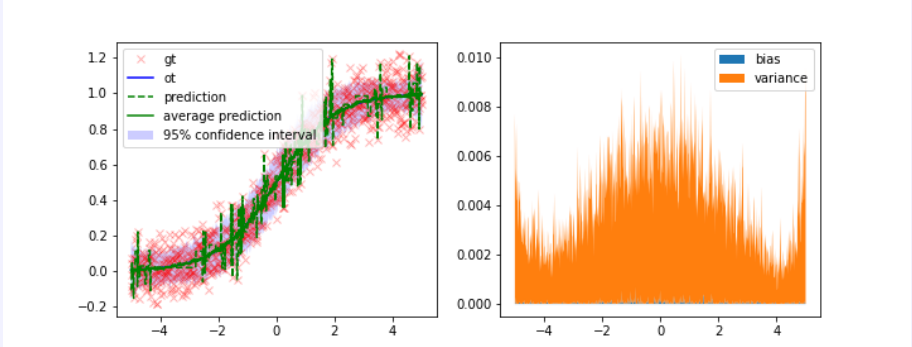
only fit a particular dataset and, therefore, do not generalize well because of the high variance all over the place. This is in contrast to a maximum depth of 1, where the decision tree is incapable of capturing the structure of the sigmoid function, leading to a high bias term in large parts of the model. Finally, the difference between the objective truth and the ground truth is visualized.



(a) max depth = 1



(b) max depth = 4



(c) max depth = 8

Figure 2.5: Models' predictions and Bias-Variance Tradeoff visualized for a maximal depth of 1, 4 and 8. Refers to the experiment in Figure 2.4

2.3.3 Tree-based Models

The topic of artificial intelligence is known as a fast-developing field, which can be assessed through the growth of the number of AI journal publications. For example, from 2019 to 2020 the number of AI journal publications has grown by a factor of 1.345; in comparison to the growth rate of 1.196 from 2018 to 2019 [67]. Still, the most commonly used algorithms are linear and logistic regression directly followed by decision trees or random forests [37]. While the first kind of model fails when features are correlated in a non-linear way, decision trees have the expressive power to solve such types of problems [47].
This basic type of algorithm remains popular, as they can produce models with a high bias, and this is also a property that makes them model-inherent interpretable. Furthermore, they are the basic building blocks for more complex models created through bagging or boosting. Two good examples for boosting are AdaBoost (i.e., with decision trees as estimators) and Gradient Boosting. Among other models, random forests are a well-known bagging model. All these learning methods make use of tree-based models and are supervised algorithms. Therefore, this section introduces tree-based models but uses a visual approach (like in [63]) referred to later in this thesis.

General Concept Regardless of what kind of tree-based model is used, the input space $\mathbb{I}$ is partitioned into distinct cuboids $R_1, ..., R_T \subset \mathbb{I}$ ($T \in \mathbb{N}$ the number of cuboids). Each of these cuboids R_t are assigned a model ($t \in [1, ..., T]$) [8]. For example, when using standard decision trees, constant models $M_{c_t} : R_t \to \mathbb{T}$ with $r \mapsto c_t$ are used, such that all inputs $r \in R_t$ are assigned the constant $c_t \in \mathbb{T}$. This kind of partitioning in combination with the assigned predictions c_t is visualized in Figure 2.6 using the model from the example 5.

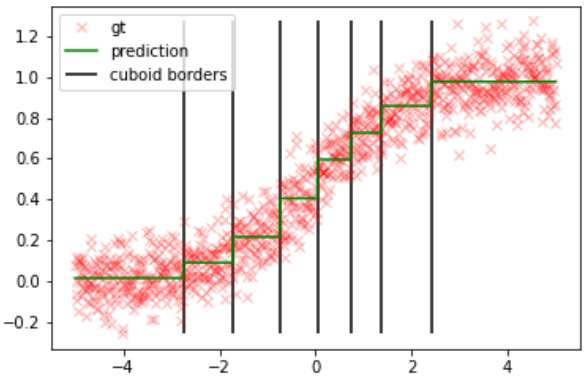


Figure 2.6: Partitioning of the one-dimensional input space given a decision tree trained to approximate the sigmoid function

Training a tree-based model is, therefore ,twofold: (i) Partitioning the input space, and (ii) assigning each cuboid a model. The input space is partitioned by learning a function $\mathcal{T} : \mathbb{I} \to [1, ..., T]$, which assigns each $x \in \mathbb{I}$ a cuboid R_t.
$\mathcal{T}$ is represented as a binary tree. Each leaf of the tree refers to a cuboid, R_t and all the other nodes are split nodes. A split node $n_{d,u}$ has two child nodes $n_{(d+1),l}$, $n_{(d+1),r}$ and is coupled with a split criterion $f_k \leq \theta_{d,u}$ ($k \in [1, ..., K]$, K is the number of available features, $n_{d,u}$ refers to the u'th node on the depth d and $\theta_{d,u}$ is a parameter comparable

with f_k). Given a sample $(\mathbf{x}_i, t_i)$, the split criterion is evaluated via

$$(\mathbf{x}_i \models f_k \ \leq \ \theta_{i,u}) \Longleftrightarrow (x_{i,k} \ \leq \ \theta_u) \tag{2.1}$$

If $\mathbf{x}_i$ is a model for the split criterion, then the procedure continues with $n_{(d+1),r}$ otherwise with $n_{(d+1),l}$. When a leaf referring to a cuboid R_t is reached by following this procedure from the root node $n_{1,1}$, the output of $\mathcal{T}(\mathbf{x}_i)$ is known to be t. Depending on the implementation, operators other than $\leq$ may be allowed.
In general, finding the optimal $\mathcal{T}$ is infeasible as the structure of the tree and the parameters of each node need to be found, which spans a solution space that grows exponentially with the number of nodes available for $\mathcal{T}$.
In the case of decision trees, a greedy approach is used to learn $\mathcal{T}$ and, therefore, the solution might not be optimal. This work is centered around explaining and interpreting models with respect to semantic constraint validation. Consequently, it is unavoidable to understand the basic approach used to train a decision tree.

Greedy Inducer Here, a decision tree used for regression with cuboid borders, as visualized in Figure 2.6, should be learned. The process is a recursive one, and in each step, there is a set of samples $R_{d,u} \subset D$ and one needs to decide whether the node $n_{d,u}$ corresponding to the step will be a leaf node or a split node. This criterion needs to be chosen wisely, as shown in example 5. In the case of a regression task, one might stop if all the samples $(\mathbf{x}_i, t_i) \in R_{d,u}$ have the same target value t_i or the target values have a limited standard deviation. The first criterion will lead to decision trees with high variance because the decision trees will be specialized to the dataset used for training. The second criterion requires further knowledge about the dataset, and the standard deviation might be different in various spaces of the dataset. In this example, the inducer decides that the node will be a leaf node if, $d = 4$ as example 5 has shown that a maximal depth of 4 is reasonable for the given task.
If $n_{d,u}$ will be a leaf node, a constant model $M_{c_{d,u}}$ has to be selected such that the performance of the model is as good as possible, given the available data $R_{d,u}$. As typical for regression tasks, the performance is measured by the mean squared error:

$$P(M_{c_{d,u}}, R_{d,u}) = \frac{1}{|R_{d,u}|} \sum_{(\mathbf{x}_i, t_i) \in R_{d,u}} (t_i - c_{d,u})^2 \tag{2.2}$$

minimizing the term with respect to $c_{d,u}$ leads to the optimal solution, which is the average:

$$c_{d,u} = \frac{1}{|R_{d,u}|} \sum_{(\mathbf{x}_i, t_i) \in R_{d,u}} t_i \tag{2.3}$$

As a leaf is reached, $R_{d,u}$ corresponds to a cuboid R_t, which is referred to by $\mathcal{T}$. The feature vectors corresponding to the samples in $R_{d,u}$ form a subset of R_t, which will be real in most cases. That is because R_t represents the cuboid of all possible feature vectors x_i that get the prediction c_t assigned due to $\mathcal{T}(\mathbf{x}_i) = t$ and $R_{d,u}$ is just the subset of the samples of D used to infer the leaf and c_t during training.
If $n_{d,u}$ will be a split node, the data $R_{d,u}$ has to be split into disjoint sets $R_{(d+1),l}$ and $R_{(d+1),r}$ given a criterion $f_k \leq \theta_{d,u}$ as described above. f_k and $\theta_{d,u}$ are chosen by trying all features for all possible splitting values and choosing the one maximizing the performance of the sum of the child nodes, measured by formula 2.2. The calculation is performed

with the optimal prediction $c_{(d+1),l}$ and $c_{(d+1),r}$ calculated with (2.3). Then the procedure is recursively called for $n_{(d+1),l}$ and $n_{(d+1),r}$.
Finally, the training of a decision tree is just a matter of starting the recursive procedure for the root node $n_{1,1}$ with the data $R_{1,1} := D$.

Example 6: Estimating the $R_{d,u}$ of the motivating example decision tree

Here, the recursive procedure described is applied to discover the sets $R_{d,u}$ given the dataset D shown in example 7 and the decision tree in Figure 1.3. Starting with the root node $n_{1,1}$ (Node 0 in Figure 1.3), for which $R_{1,1} = D$ is given, the examples are split according to the criterion: allergic_to ≤ 0 (using formula 2.1). To give $R_{2,1}$ and $R_{2,2}$. Next, the procedure would be called for $n_{2,1}$. Executing the procedure until the end, gives for each node $n_{d,u}$ the indices of D occurring in $R_{d,u}$. The result is shown in Figure 2.5; additionally referring to the persons and the enumeration of the nodes in Figure 1.3.

Node	$n_{d,u}$	Persons	$\{i \mid (x_i, t_i) \in R_{d,u}\}$
0	$n_{1,1}$	`:Max`, `:Maria`, `:Eva`, `:Laura`	$[1, ..., 9999]$
1	$n_{2,1}$	`:Maria`, `:Eva`	$[3334, ..., 9166]$
4	$n_{2,2}$	`:Max`, `:Laura`	$[1, ..., 3334] \cup [9167, ..., 9999]$
2	$n_{3,1}$	`:Eva`	$[4167, ..., 9166]$
3	$n_{3,2}$	`:Maria`	$[3334, ..., 4166]$

Table 2.5: For each node $n_{d,u}$ in Figure 1.3, the indices of the dataset D included in $R_{d,u}$ are shown

The model's training procedure stays the same when a decision tree for classification is trained. Besides that, the performance measure changes from the mean squared error to the *negative cross-entropy* or the *Gini index* and, therefore, the optimal prediction changes to be the target class referred to by the most examples in the node.

Visually interpreting the Decision Tree This work builds on the kind of visualizations shown in Figure 1.3 and 2.7 proposed by the `dtreeviz` library [63]. The visualizations are built to explain what has been learned by the decision tree and why the decision tree makes a certain decision given a problem instance. This is done with respect to a specific dataset D.
For each node $n_{d,u}$[1], the distribution of the ground truth values of the set of samples $R_{d,u}$ is visualized. As described above, $R_{d,u}$ refers to the samples used to decide whether the node $n_{d,u}$ will be a split or a leaf node given the dataset D. In the case that $n_{d,u}$ is a split node, the distribution of the ground truth is shown with respect to the split feature f_k. As the values of the other features are ignored, the distribution shows the marginal effect the feature has on the ground truth. Further, the size of distributions is scaled with the number of samples included in $R_{d,u}$.
Next, to explain why the decisions are made by the decision tree induced by the learning algorithm during training, Figure 2.7 is investigated further.

[1]The indexing is demonstrated for some nodes in the Figure 2.7

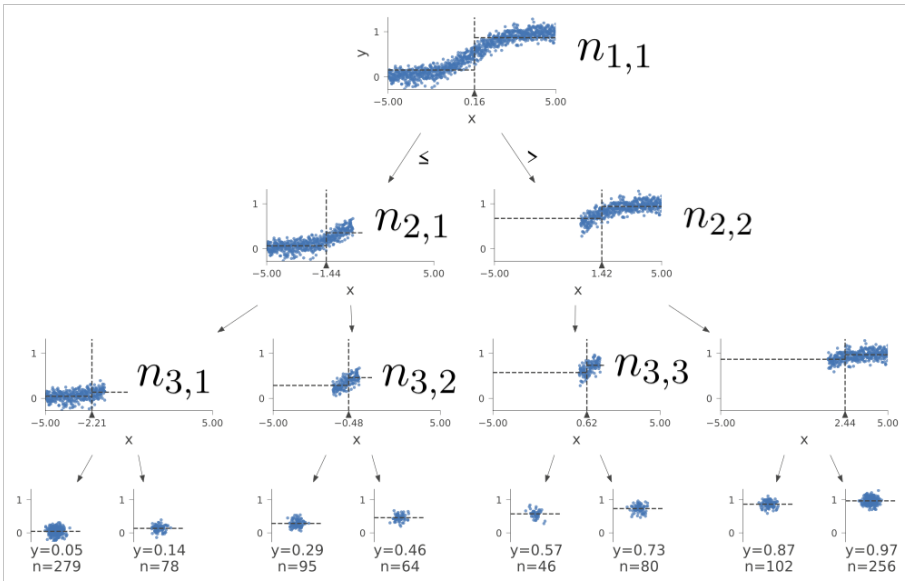


Figure 2.7: Visualization of a decision tree trained to approximate the sigmoid function with a maximal depth of four

Figure 2.7 shows the decision tree trained to approximate the sigmoid function from example 5, the explanation will be about the regression task but can be transferred analog to the classification task. In the case of regression, scatter plots are used to show the distribution of the points $\{(\mathbf{x}_{i,k}, t_i) \mid (\mathbf{x}_i, t_i) \in R_{d,u})\}$ per split node. Starting at the root node $n_{1,1}$ with $R_{1,1} = D$, the learning algorithm had to choose the split feature and the decision boundary for each split node. In example 5 only one feature x is used. x is the input to the sigmoid function, and will be the split feature for each split node. Hence, the split criterion for all nodes is $x \leq \theta_{d,u}$ and $\theta_{d,u}$ has the value shown right under the marker at the x-axis. Now it is visually verifiable that $\theta_{d,u}$ is chosen correctly and, thus, the performance

$$-(P(M_{c_l}, R_{d,l}) + P(M_{c_r}, R_{d,r}))$$

is maximized. Therefore, explaining the decision made at each split node is a matter of visually verifying that the new cuboid border shown by the vertical dotted line is chosen optimally. The horizontal dotted line corresponds to the optimal predictions, which would be given if the child nodes were leaves.
In the case of regression, the optimal predictions of the learning algorithm estimated for each node can be verified as the horizontal dotted lines visualize them. In the leaves, the horizontal dotted line directly corresponds to the predictions made.
As a final note: For these visualizations to work and explain why specific cuboid borders are chosen, and the decision tree makes particular predictions, the dataset used for visualization should be the one initially used to train the decision tree. If a different dataset is used, the visualization will show how the decision tree behaves with respect to the new data, but will not show how the cuboid boundaries are chosen during the training process.

2.4 Summary

This chapter introduced the main concepts used in this thesis. The first part featured the concepts necessary for semantic constraint validation. First, knowledge graphs are

introduced in the context of RDF graphs, which give entities in the graph a semantic context. Next, building on N-Triples, Turtle is presented as a serialization format for RDF graphs. As it turns out, SPARQL, the W3C recommended query language to query data from RDF graphs, uses a syntax similar to Turtle with the addition of query variables; extending RDF triples to triple patterns. Evaluating a triple pattern over a knowledge graph gives a set of SPARQL solution mappings. Using algebraic operations over these sets, manifested in the SPARQL language as graph patterns, allows for writing expressive SPARQL queries. In this work, the SPARQL evaluation of the query $Q \in \mathbf{Q}$ over the knowledge graph $G \in \mathbf{G}$ is denoted with $[[Q]]_G$. Among other use cases, in the abstract syntax for the SHACL core constraint components as introduced by Cormen et al. [15], SPARQL queries are used to define the target of SHACL shapes. Multiple interconnected SHACL shapes S form a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$, which represent constraints over a knowledge graph. The SHACL validation of $\mathcal{S}$ over a knowledge graph G is performed by a SHACL engine and gives the validation results in the form of the entity validation function.

In the second part, explainable AI is introduced as the extension of AI, allowing one to comprehend and trust an AI agent's results. A focus is on explainable machine learning and the main concepts of machine learning: An inducer solves a given task $(\mathbb{I}, \mathbb{T}, ot)$ by training a machine learning model on a given dataset sampled from ot. Whether the resulting model matches the user's expectations (e.g., generalizes well) is usually checked by measuring the model's performance with a performance measure (e.g., mean squared error, accuracy) on a test set sampled separately from the objective truth. Finally, tree-based models and, specifically, decision trees are introduced. A decision tree visualization is discussed, which explains a decision tree trained on the task of approximating the sigmoid function based on the data used to train the model.

Chapter 3

Related Work

In this chapter, state-of-the-art methods of topics related to this thesis are presented in short. At the end of each section, the approach followed in this thesis is positioned with respect to state-of-the-art methods.

3.1 Data Mining and Data Extraction from Knowledge Graphs

The semantic web provides large linked open data sources, which try to connect large amounts of data crawled from the web into a knowledge graph. For example there is *DBpedia* [45] and *Wikidata* [65]. *DBpedia* is a community-based project, that automatically extracts and combines multi-lingual data from Wikipedia into a single knowledge graph. Therefore Wikipedia info boxes are exploited as a source of structured data. In comparison, *Wikidata* is a knowledge graph directly associated with the Wikimedia project (i.e., including Wikipedia) aiming to provide "[...] universal identifier for all relevant named entities"[65] in an approach to get the central authority for this purpose.
It is necessary to extract the data first to make this kind of data usable with standard machine learning approaches. Narasimha et al. [48] proposed a data mining system for linked data called *LiDDM*. *LiDDM* is a model that offers a GUI conducted pipeline to extract linked data from different sources and make it usable for statistical analysis; with results visualized in the end. For data extraction, the user can choose between specifying the SPARQL query and an automatic query builder based on predicate recommendations. The user is only needed to specify the triples. Cheng et al. [11] provide a theoretical framework to automatically query semantic feature vectors from a knowledge graph based on SPARQL. Given a set of entities $\mathcal{E}$ of interest, they automatically propose a feature vector $v_e \in \{0, 1\}$ with semantic neighborhood information for all $e \in \mathcal{E}$. Moghaddam et al. [3] proposed a framework capable of automatic feature extraction from a knowledge graph by generating SPARQL queries based on traversing an RDF graph. Like Cheng et al., they start with a set of entities of interest but aim to retrieve literals instead of a lengthy binary vector automatically. This thesis uses SPARQL queries like the ones generated by the frameworks above. The user can use any of the tools to generate the query and extract the dataset automatically. The approach won't be dependent on how the SPARQL query to retrieve the dataset is designed. The concept of entities of interest will be formalized because, during the extraction of the dataset, these entities will be aligned with the samples in the extracted dataset. Therefore not only using the knowledge graph as a data source but also as a source for the semantic context of the samples in the

dataset, which stays usable after data extraction.

3.2 Integrity Constraint Validation

Cormen et al. [15] introduced the semantics for recursive SHACL and provided complexity bounds for the general problem of SHACL constraint validation over a knowledge graph. In general, the problem turns out to be not solvable in polynomial time, in the case of $P \neq NP$. However, they identified tractable fragments of the language [14] and proposed SHACL2SPARQL [13] an algorithm to tackle these fragments. Nevertheless, the execution of SHACL validation over large knowledge graphs stays expensive, which is why Figuera et al. [21] presented Trav-SHACL. Trav-SHACL is a SHACL engine built to detect invalid entities early in an approach to maximize the scalability of the engine. Both engines interleave the saturation performed by the SAT solver (see section 2.2.4) with the materialization and grounding of the rules. However, Trav-SHACL rewrites the queries sent to the SPARQL endpoint (e.g., pushing `FILTER` terms into the queries to make them more selective). Furthermore, Trav-SHACL chooses the shape validation, such that the shapes, that can invalidate the most number of entities, are validated first. This thesis depends on the performance of SHACL validators and proposes heuristics to minimize the time spent performing SHACL constraint validation. Compared to Trav-SHACL, the heuristics used for optimization will be agnostic to the SHACL engine, enabling the wider application.

3.3 Model-agnostic Interpretabilty Methods

Model-agnostic interpretability methods can be applied in an approach to make black-box models better interpretable. On the one hand, there are global methods to analyze how features influence the prediction of a machine learning model on average (e.g., Partial Dependence Plot [24], Accumulated Local Effects Plot [2]), to measure the feature importance (e.g., model reliance [22]) or to build another approximating model, which can be interpreted more easily.
For example, given a dataset D having the feature set $Z = \{f_1, f_2, ..., f_M\}$ the Partial Dependence Plot shows the marginal distribution of the predictions of a machine learning model M_θ given a subset of the features $Z_C \subset Z$. The marginalization is done by taking the average prediction (in the case of regression) and the most mentioned class (in the case of classification) per Z_C feature combination. This allows seeing the general influence the features in Z_C have on M_θ.
On the other hand, we have local methods to explain specific predictions of a model like LIME [56]. Again one might assume a trained machine learning model M_θ, but now one is just interested in the approximate linear effect the features have on the prediction at a given problem instance $\mathbf{x_i}$. Therefore a new dataset D_2 for supervised machine learning is generated by varying the feature values of $\mathbf{x_i}$ and for each problem instance generated ask for $M_\theta(\mathbf{x_i} + s_j)$. The s_j $(j \in [1, ..., |D_2|])$ might be estimated by sampling $|D_2|$ times from a multivariate normal distributed random variable with zero mean and a covariance matrix $\epsilon * \mathcal{I}$ where $\epsilon > 0$ and $\mathcal{I}$ the identity matrix with dimension $|\mathbf{x_i}| \times |\mathbf{x_i}|$. Now $|D_2|$ can be used to train an easily interpretable model, which approximates M_θ at $\mathbf{x_i}$ locally. For instance, in the case of continuous feature and target values, a linear regression $M_w(x) = w * x$ can be used. The weights w, as well as the standard deviation of the weights w_{std}, can be estimated an interpreted as a feature importance vector $\frac{w}{w_{std}}$.

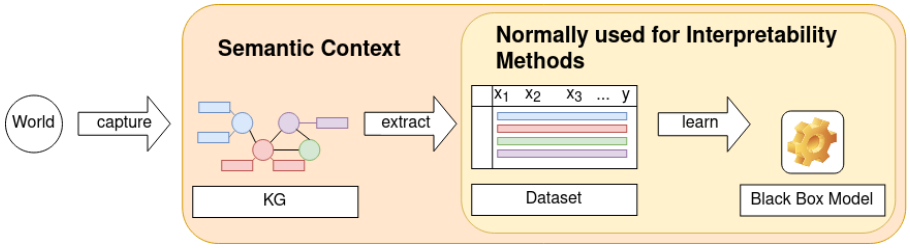


Figure 3.1: Layers to be considered by interpretability methods: The real world goes through many layers before the interpretability method is used to inform the human about the model. The semantic context is ignored by interpretability methods used normally. (The figure is inspired by [47])

Both methods help investigate the features' importance when making a prediction with M_θ for $\mathbf{x_i}$.
However, both examples and the other usual interpretable methods named above have one major drawback: They try to explain the model only with respect to the features, without considering the semantic context provided by a knowledge graph. Figure 3.1 illustrates this drawback. The entities of the real world can be captured together with their semantic context in a knowledge graph. The dataset is created based on the entities in the knowledge graph, but the extraction process discards the semantic context the knowledge graph provides. An inducer uses the dataset to train a machine learning model (i.e., a black box model), which is then analyzed by the interpretability methods (e.g., global or local) for better interpretability. However, the semantic context the knowledge graph provides is not considered. This thesis proposes a model-agnostic interpretable method. The semantic context is used in the process of the constraint validation to interpret machine learning models (i.e., especially decision trees) better with respect to patterns detected through the constraint validation.

3.4 Explainable Machine Learning over Knowledge Graphs

Machine learning over knowledge graphs is used for (i) link prediction to identify missing facts (i.e. triples) in the knowledge graph [59], (ii) entity clustering to detect repeating patterns in entities, or even to discover completely synonymous entities [61], (iii) Image Classification [43] and various further applications. All of these approaches somehow make use of the semantic context the knowledge graph provides: Embeddings (e.g., graph-, node-, link embeddings, or messages passed in graph neural networks) incorporate the semantic context and encode it in a latent vector representation. Rule-based approaches yield insights based on the semantic context [44]; for example, in the context of link prediction.
Embedding-based approaches usually scale well with large-size knowledge graphs, but predictions based on them are not inherently explainable. In contrast, when a rule-based approach makes a prediction, it is explainable, since the prediction was made based on a set of rules. Nevertheless, comprehensible rules are usually handcrafted, making these approaches scale worse.
For example, Halliwell et al. [30] propose using user-scored explanations for rating state-of-the-art model explanations for link predictions. They generate ground truth explanations for links to be rated based on horn clauses. Mohamed et al. [25] generate explainable labeled clusters based on rule mining. Nevertheless, using embeddings for the task of clus-

tering, the clusters' labels are explainable through they originate from rules.
In this thesis, the semantic context is exploited by using SHACL constraint validation to make machine learning models' predictions explainable. The explainability, however, originates from using handcrafted rules (i.e., constraints).

3.5 Visualization of SHACL constraints

In the semantic web, the communication and cooperation of machines with humans is a topic [7]. The visualization of SHACL constraints allows domain experts to understand SHACL constraints without needing them to know RDF Schema (i.e., an RDF vocabulary for data-modeling) and the exact SHACL specification.
Arndt et al. [49] propose a graphical RDF Schema editor and visual SHACL editor in a single tool. The tool allows domain experts to create RDF vocabularies and SHACL shape schemas (i.e., SHACL constraints) in a graphical toolbox. The tool is not based on a familiar visual notation. In contrast, Lieber et al. [41] present *unSHACLed* a tool, agnostic to the RDF vocabulary used for constraint modeling, that makes use of known visualizations like UML and VOWL. Supporting all common constraint types and editing operations, they were able to reuse familiar notations. In a study with users proficient with Linked Data and UML, 81% of the questions were answered correctly. However, they could not cover the visualization notation's scalability. A recent master thesis [1], making use of 3D visualization, experimentally studied the effect of the network size based on synthetic shape schemas. It turned out that the number of nodes and links to be rendered had the highest impact on rendering. Nevertheless, up to 5,000 nodes could be rendered in under ten seconds.
This thesis does not visualize SHACL shape schemas. Still, it aims to visualize constraint validation results in a model-coherent way, promoting the interpretability of machine learning models (e.g., decision trees). Therefore a known visualization library (i.e., dtreeviz [63]) is adapted, and synthetic shape schemas are used for the evaluation.

3.6 Summary

This chapter presents various topics related to the thesis, supported by current papers. Data extraction is a central topic and will be based on user-defined SPARQL queries. However, the user is free to use automatically generated queries by other frameworks. Constraints exploit the semantic context of entities for an interpretability method based on SHACL validation. The SHACL process will be speed-up by heuristics agnostic to the SHACL engine. When the constraint validation results are available they will be visualized in a familiar way (i.e., familiar to users of dtreeviz). Model predictions are made explainable based on handcrafted constraints.

Chapter 4

Approach

This chapter introduces the problem described in the motivating example more formally. Therefore, the concepts presented in the background section are used. Afterward, the approach to solving the problem is explained in detail, and heuristics are proposed to improve the performance of the approach. The motivating example continues to help the reader understand the approach.

4.1 Problem Definition

The problem approached in this thesis is twofold. The first part is to validate constraints over machine learning models while using the semantic context of the entities in an underlying knowledge graph. The second part is to use the validation results to make the machine learning model better interpretable or even explainable. The following two sections present these two problems formally using the motivating example.

4.1.1 Validating Constraints over Machine Learning Models

To be able to use the semantic context of the entities provided in a knowledge graph to validate a machine learning model against a set of constraints, a validation engine is needed. The validation engine needs to connect the information provided by the knowledge graph and the predictions a machine learning model makes based on samples. To do so, the problem of constraint validation for machine learning models is narrowed down to the case, in which a mapping between the samples in the dataset and the nodes in the knowledge graph exists. The mapping will be called sample-to-node mapping and is defined as follows:

Definition 23: Sample-to-node Mapping
Given a dataset D for a specific task with $N \in \mathbb{N}$ samples and a knowledge graph G, a sample-to-node mapping is a total function

$$\eta : [1, ..., N] \to (\mathbf{B} \cup \mathbf{I}) \cap V_G$$

mapping a sample to an IRI or a blank node in G. The infinite set of sample-to-node mappings is given by $\boldsymbol{\eta}$

At this point, it is important to recognize that η does not make any restrictions on the dataset and their assigned nodes in the knowledge graph. On the left-hand side of the

function, the indices of the dataset are used to allow duplicates in the dataset (e.g., required for oversampling [40]). This is in contrast to using $\mathbb{I} \times \mathbb{T}$ on the left, which would not allow duplicates to be used. In addition, η enables a node in the knowledge graph to be associated with several or no samples. The knowledge graph connected through η with the dataset is referred to by *underlying knowledge graph*.

Example 7: The Function η used in the Motivating Example

This example refers to the dataset in Figure 1.1 and the underlying knowledge graph in Figure 1.2. In the motivating example, the dataset consists of a set of person types (the person column). For each person type, various features (representing the problem instance) and the vaccination status (representing the target) are defined. Additionally, the overall ratio (the weight column) of this type of person in the whole dataset is given. Assuming a dataset of $N = 9999$ examples as in Figure 1.4, there will be $\frac{1}{12} * 9999 \approx 833$ examples of the type `:Maria`. Each example is then mapped to a node in the underlying knowledge graph. The same procedure is applied to the other types of persons.
That is what η does for each example in the dataset. Table 4.1 shows the complete sample-to-node mapping and the corresponding problem instances and targets.

i	$\mathbf{x}_i$				t_i	$\eta(i)$
	allergic_to	gender	pregnant	country	vaccinated	
1	PEG	male	$\bot$	Germany	$\bot$	`:Max_0`
⋮	⋮	⋮	⋮	⋮	⋮	⋮
3333	PEG	male	$\bot$	Germany	$\bot$	`:Max_3332`
3334		female	$\top$	Germany	$\bot$	`:Maria_0`
⋮		⋮	⋮	⋮	⋮	⋮
4166		female	$\top$	Germany	$\bot$	`:Maria_832`
4167		female	$\bot$	Germany	$\top$	`:Eva_0`
⋮		⋮	⋮	⋮	⋮	⋮
9166		female	$\bot$	Germany	$\top$	`:Eva_4999`
9167	PEG	female	$\bot$	Germany	$\bot$	`:Laura_0`
⋮	⋮	⋮	⋮	⋮	⋮	⋮
9999	PEG	female	$\bot$	Germany	$\bot$	`:Laura_832`

Table 4.1: The Dataset Extracted from Figure 1.1 and 1.2 Annotated with the Sample-to-node Mapping

Given the index $i \in [1, ..., N]$ of a sample $(\mathbf{x}_i, t_i)$ in a dataset D, η maps the index to a node $v \in \mathbf{B} \cup \mathbf{I}$ of a knowledge graph. During inference, a trained machine learning model M_θ transforms the problem instance $\mathbf{x}_i$ into a prediction $M_\theta(\mathbf{x}_i)$ for the target. These two steps connect the entities or nodes in the knowledge graph to the prediction a machine learning model makes based on them. The next step is to use the semantic context of the entities and the corresponding predictions in the form of constraints.
A constraint will be defined similarly to an implication used in Boolean formulas. In general, a Boolean implication (e.g., $A \rightarrow B$ where A and B are Boolean formulas) consists

of a condition (i.e., A) and a restriction (i.e., B), getting active when the condition applies. Regarding the validation engine, the left-hand side (i.e., A) of the constraint will be a shape schema with a target shape, describing a condition on the entities found in the knowledge graph. This is done according to the W3C recommendation in the SHACL language [38] and allows using the semantic context of the entities in the knowledge graph. The right-hand side of the implication (i.e., B) then allows for restrictions of the target prediction of the machine learning model depending on the condition. The structure of the constraint allows to generate explanations in specific cases: If the condition applies and the target prediction is according to the restrictions, then the restrictions can be used as an explanation for the target prediction (B might be true because of a A).

Definition 24: Constraint
A constraint C is composed of a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ as defined in definition 17 ($\mathcal{S} \in \mathbf{SN}$), a shape $ts \in S$ (called target shape) and a logical expression ϕ_T with only the T as a free variable. T is the placeholder for the predicted target of the machine learning model, to be validated. C is serialized as follows:

$$\mathcal{S}\big|_{ts} \rightsquigarrow \phi_T$$

where $\rightsquigarrow$ is a symbol for a binary operator[a]. The components of C can be accessed with $C.\mathcal{S}$, $C.ts$ and $C.\phi_T$. The infinite set of constraints is defined to be $\mathbf{C}$.

[a] The semantics of the constraint will be defined in section 4.2.2

Until the semantic of the $\rightsquigarrow$ is formally defined, the reader should translate it with "is a requirement for".

Example 8: An Example Constraint

In the introduction, the constraint „*Every pregnant person in Germany, which has more than 20 contacts to non-vaccinated persons should get vaccinated*“ was formulated and will now be transformed into the form defined above. The constraint can clearly be expressed in the form of an implication having a set of conditions on the node found in the knowledge graph on the left-hand side (e.g., the person lives in Germany, is pregnant, and has more than 20 contacts with non-vaccinated persons) and a Boolean expression about the target (e.g., the person has to be vaccinated) on the right-hand side.
First, the necessary condition on the node is defined. Therefore, the abstract notation from section 2.2.4 is used to define the shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$.

$$\begin{aligned} S &= \{\texttt{:PersonShape}, \texttt{:NotVaccinatedPersonShape}\} \\ \texttt{DEF}(\texttt{:PersonShape}) &= (\geq_1 \texttt{:pregnant.True}) \ \wedge \ (\geq_1 \texttt{:country.Germany}) \wedge \\ &\quad (\geq_{21} \texttt{:contact\_with.:NotVaccinatedPersonShape}) \\ \texttt{DEF}(\texttt{:NotVaccinatedPersonShape}) &= (\geq_1 \texttt{:vaccinated.False}) \end{aligned}$$

```
PREFIX : <http://example.org/>
SELECT ?x WHERE {
    ?x a :Person
```

```
}
```

Listing 4.1: `TARG(:PersonShape)` and `TARG(:NotVaccinatedPersonShape)`

When evaluating the shape schema against the knowledge graph to validate the constraint, `:PersonShape` defines the condition based on `:NotVaccinatedPersonShape`. Therefore, the target shape is `:PersonShape` and in case that a node is valid according to the target shape the corresponding target prediction of the machine learning model should be according to

$$\phi_{\text{vaccinated}} = (\text{vaccinated} = \top)$$

Hence, the final constraint is

$$\mathcal{S}|_{\texttt{:PersonShape}} \rightsquigarrow \phi_{\text{vaccinated}}$$

Given the definition of the sample-to-node mapping (definition 23) and the constraint (definition 24), to be validated over a machine learning model and the underlying knowledge graph, the problem can be formulated.

Lemma 2: The Limited Problem of Validating Constraints over Machine Learning Models
The input of the problem of validating constraints over machine learning models is a 5-tuple $(\mathcal{C}, M_\theta, D, G, \eta)$ where $\mathcal{C} \subset \mathbf{C}$ a set of constraints, M_θ a machine learning model with parameters θ, D a dataset according to definition 21, $G \in \mathbf{G}$ and $\eta \in \boldsymbol{\eta}$ a sample-to-node mapping. Given a sample-to-node mapping η, the problem is to efficiently evaluate each constraint $C \in \mathcal{C}$ for each sample indexed with $[1, ..., N]$ over the machine learning model M_θ and the knowledge graph G.

4.1.2 Constraint-based Explanations

The problem addressed in this thesis goes one step further than validating a set of constraints as described in lemma 2. Another goal is to use the knowledge gained through the validation to make the model behavior more evident to humans. This happens according to the definition of explainable resp. interpretable AI: "[...] produce agents, whose behavior can be understood by humans [...]" resp. "[...] describe the internals of a system in a way that is understandable to humans" and leads to a model, producing predictions that can be trusted by the user (see chapter 2.3).
When using the validation results of the user-defined constraints, the goal is to summarize the validation results so that they can be combined with the patterns used by the machine learning model. Therefore, when talking about explaining a model in the context of constraint validation in this thesis, it refers to the following definition of the problem.

Lemma 3: The Problem of Explaining Machine Learning Model Behavior using Constraint Validation Results
Given the validation results for each constraint $C \in \mathcal{C} \subset \mathbf{C}$ for each sample in the dataset D, indexed with $[1, ..., N]$, over the machine learning model M_θ, the problem

is to summarize the knowledge gained in such a way, that the behavior of M_θ gets more apparent to the user.

4.2 Validating Constraints over Machine Learning Models

As shown graphically in Figure 1.5, the approach described in this chapter can be represented sequentially. The first three steps are the ones that build the foundation and contribute to an approach to solve the problem described in lemma 2. This is also the problem tackled in this section. The initial situation assumed is that a user wants to solve a task (see definition 19) using a machine learning model based on data available in a knowledge graph (see definition 6), but the machine learning model should respect a given set of user-created constraints as defined in definition 24. In the first subsection, the first step called „Propositionalization“ is described in more detail and will provide a dataset (see definition 21) and a sample-to-node mapping (see definition 23). Building on that, a machine learning model is trained on the dataset, which completes the inputs of the problem tackled and leads to the third step called „Constraint Validation“ (see Figure 1.5). This step is described in detail in section 4.2.2 by presenting the semantics of the constraints. This leads to the validation engine solving the problem model-agnostic.

4.2.1 Prepositionalization

Most machine learning algorithms that train machine learning models demand a propositional form of the input data. This is the kind of data matching the definition 21. Each sample is associated with a number of features and a target, given that it is not an unsupervised algorithm. Hence, the goal is to generate a dataset given a knowledge graph and, in parallel, create the needed sample-to-node mapping by tracking the generation process. There is already some work tackling the first part of the goal by using user-defined SPARQL queries [11, 48]. A recent paper even proposed a generic distributed framework that can generate these kinds of queries automatically [3]. Since this procedure has proven itself and is also suitable for extracting the required mapping, SPARQL queries are used here for propositionalization. In this work, the user is required to provide the query. Nevertheless, a framework like the one in [3] can be used as long as it generates queries as specified below. To get started, the following provides the components used to specify the query. First, the user must decide on a corpus of entities, making predictions about [11].

Definition 25: Set of Seed Nodes
The set of seed nodes $\mathbf{s} \subset \mathbf{I} \cup \mathbf{B}$ represents the corpus of entities a user wishes to make predictions about.

Each seed node represents a starting point in the knowledge graph from which one or more samples will be extracted. Similar to the notion of a target query, which retrieves target instances to be checked against a shape, one can define a *seed query* Q_s, which gives, when executed over a knowledge graph, the set of seed nodes to be used to construct the dataset.

Definition 26: Seed Query
Given a set of seed nodes $\mathbf{s} \subset \mathbf{I} \cup \mathbf{B}$ and a knowledge graph $G \in \mathbf{G}$, a seed query $Q_s \in \mathbf{Q}$ is a SPARQL query with $[[Q_s]]_G = \mathbf{s}$ and $|\text{dom}([[Q_s]]_G)| = 1$.

The infinite set of seed queries equals the infinite set of target queries $\mathbf{Q_T}$.
To connect a seed node with a node representing a feature, property paths are needed. Therefore, one must choose K property paths $p_1, p_2, ..., p_K \in \overrightarrow{\mathbf{P}}$. Each property path p_j connects a seed node s_i to the corresponding feature value $x_{i,j}$ extracted from the knowledge graph. In the case of a supervised algorithm, an additional path $p_{t_i} \in \overrightarrow{\mathbf{P}}$ is needed to connect the seed node s_i with the corresponding target value t_i.
Finally, the concept of a seed query and the concept of property paths need to be combined to build a dataset constructing SPARQL query Q_D. In definition 11 the `SELECT` query as defined in definition 12 is listed as a graph pattern, which allows the usage of nested queries in SPARQL. Therefore, the seed query can be integrated as a nested query into Q_D. Furthermore, a property path can be seen as a shorthand for a more complicated graph pattern. This becomes evident in definition 14. Each recursive component of a property path can be evaluated using the algebraic operations over sets of solutions mappings. Here, a property path is used as an in-place operation between two variables. These variables replace the variables $?s_p$ and $?e_p$ in the definition 14. The variable on the left-hand side of the property path refers to, $?s_p$ and the variable on the right-hand side to $?e_p$.

Example 9: Using Property Paths and Nested Queries as Graph Patterns

In the motivating example, one might be interested in all the pairs of people, who are related to each other and the first person is constrained to be not vaccinated. Here two persons are related to each other, when there is a path in the knowledge graph connecting them and the path consists of a maximum of two edges labeled with `:contact_with`.
Let ∇ be the identity function; mapping each variable to itself. First, a query to extract non-vaccinated persons is defined

$$Q_{\text{not vaccinated}} = \texttt{SELECT}(?x,\nabla,((?x,\ \texttt{rdf:type},\ \texttt{:Person}) \\ \texttt{AND}\ (?x,\ \texttt{:vaccinated},\ \texttt{False}))$$

Next, two property paths are needed to express the concept of paths connecting two persons via edges labeled with `:contact_with`.

$$p_1 := \texttt{:contact\_with}$$
$$p_2 := p_1/p_1$$

p_1 is a predicate path and p_2 a sequence path consisting of two predicate paths. Now the full query can be specified:

$$\texttt{SELECT}(\{?x,?y\},\nabla,\ ((Q_{\text{not vaccinated}}\ \texttt{AND}\ (?x,\ p_1,\ ?y))\ \texttt{UNION} \\ (Q_{\text{not vaccinated}}\ \texttt{AND}\ (?x,\ p_2,\ ?y)))$$

There might be seed nodes s_i for which a property path p_j does not lead to a feature value $x_{i,j}$ in a knowledge graph G because $\forall f \in \mathbf{B} \cup \mathbf{I}\ (s_i, f) \notin [[p_j]]_G$. Therefore, the dataset may contain empty entries in the case of some samples, and Q_D has to use the `OPT` graph pattern for each property path referring to a feature. Hence, a sample including an empty entry will not be discarded. Therefore, the SPARQL query Q_D can be built as follows:

$$\texttt{SELECT}(\{?\text{x}, ?\text{f}_1, ..., ?\text{f}_K\}, \nabla, (Q_s\ \texttt{AND}\ \underset{j\in[1,...,K]}{\texttt{OPT}}\ (?\text{x}, p_j, ?\text{f}_j))) \tag{4.1}$$

An additional optional graph pattern using p_{t_i} must be used when building a dataset for a supervised algorithm.

Example 10: Query used to Generate the Motivating Example Dataset

There is already a dataset given in the motivating example (see Figure 1.1), which was extracted from a knowledge graph like the one in Figure 1.2. Here the goal is to specify how the dataset was extracted using a SPARQL query Q_D, which has the form shown as a graph pattern above. First, a seed query is needed. In the case of the motivating example, one is interested in the corpus of persons and the seed query. Therefore, one needs to extract all entities of type person.

$$Q_s = \texttt{SELECT}(?\text{x},\nabla, (?\text{x}, \texttt{rdf:type}, \texttt{:Person}))$$

As the knowledge graph is kept simple, the property paths leading to the features as well as the target are predicate paths:

$$\texttt{:allergic\_to}, \texttt{:gender}, \texttt{:pregnant}, \texttt{:country}, \texttt{:vaccinated} \in \overrightarrow{\mathbf{P}}$$

Finally, the query can be specified using the SPARQL syntax.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX : <http://example.org/>
SELECT ?x ?allergic_to ?gender ?pregnant ?country ?vaccinated
WHERE {
    ?x rdf:type :Person .
    OPTIONAL{ ?x :allergic_to ?allergic_to }
    OPTIONAL{ ?x :gender ?gender }
    OPTIONAL{ ?x :pregnant ?pregnant }
    OPTIONAL{ ?x :country ?country }
    OPTIONAL{ ?x :vaccinated ?vaccinated }
}
```

Listing 4.2: SPARQL query to generate the motivating example dataset

The kind of query Q_D presented here already comprises the seed query. Therefore, when evaluating Q_D against a knowledge graph, each solution mapping represents a sample annotated with the corresponding *seed node*. The transformation from a tuple of solution mappings to a dataset and a sample-to-node mapping is given below:

Definition 27: Transforming a Tuple of Solution Mappings into a Dataset and a Sample-to-node Mapping
Given a task $(\mathbb{I}, \mathbb{T}, ot(.))$, the function $\mho : \mathbf{M}^N \times \mathbf{V} \to (\mathbb{I} \times \mathbb{T})^N \times \boldsymbol{\eta}$ transforms a given tuple of solution mappings Ω of length $N \in \mathbf{N}$ and a variable ?t specifying the target (if there is one, „None“ otherwise) into a dataset and a sample-to-node mapping. This only requires that for each solution mapping $\mu \in \Omega$ it's ?x $\in \text{dom}(\mu)$ and $\mu(?x) \in \mathbf{I} \cup \mathbf{B}$ gives the seed node of the solution mapping.

$$\mathbf{x}_i := (\mu(?v) \mid \mu = \Omega[i] \wedge ?v \in \text{dom}(\mu) \setminus \{?x, ?t\})$$

$$\mho(\Omega, ?t)[1] \mapsto \begin{cases} ((\mathbf{x}_i, \underbrace{\Omega[i](?t)}_{t_i}) \mid i \in [1, ..., N]) & ?\text{t} \neq \text{None} \\ (\mathbf{x_i} \mid i \in [1, ..., N]) & \text{else} \end{cases}$$

$$\mho(\Omega, ?t)[2] \mapsto \{(i \mapsto \Omega[i](?x)) \mid i \in [1, ..., N]\}$$

If ?t is not „None“, then for all $\mu \in \Omega$ it's ?t $\in \text{dom}(\mu)$.

As one can observe, the dataset generating query Q_D can also be of other forms deviating from 4.1. For example, the query can use groupings and aggregations (see [53]) as long as the requirements of the previous definition are fulfilled.

Definition 28: A Dataset Generating Query
Given a dataset D and a knowledge graph G, a dataset generating query Q_D is the one, which evaluates to a set of solution mappings $\Omega := [[Q_D]]_G$, such that tuple(Ω) fulfills the requirements in definition 27 and $\mho(\Omega, ?t)[1] = D$.

The dataset and the sample-to-node mapping can now be constructed depending on the machine learning algorithm. If it is an unsupervised algorithm, and a knowledge graph G and a dataset generating query Q_D are given, then the dataset D and the sample-to-node mapping are constructed as follows:

$$(D, \eta) = \mho(\text{tuple}([[Q_D]]_G), \text{None})$$

If the algorithm is a supervised algorithm, the variable for target identification ?t is required additionally:

$$(D, \eta) = \mho(\text{tuple}([[Q_D]]_G), ?t)$$

In both cases, the sample-to-node mapping generated has as domain the original set of seed nodes chosen by the user as corpus of entities to make predictions about (e.g., given $G \in \mathbf{G}$, $Q_s \in \mathbf{Q_T}$, $\mathbf{s} := [[Q_s]]_G$ and the dataset extracting query $Q_D \in \mathbb{Q}$ comprises Q_s, the extracted sample-to-node mapping η it will full fill $\text{dom}(\eta) = \mathbf{s}$)

Example 11: Creating the Motivating Example dataset

As all the components to create the dataset from the motivating example are ready, that is, the knowledge graph G (see Figure 1.2) and the dataset generating query Q_D (see example 10), the dataset D (see Figure 1.1) and the matching sample-to-node mapping η (see example 7) can be created by executing:

$$(D, \eta) = \mho(\text{tuple}([[Q_D]]_G), ?vaccinated)$$

This concludes the section, as the goal of generating a dataset from a knowledge graph while creating the required sample-to-node mapping has been achieved.

4.2.2 Constraint Evaluation

At this point, all the inputs $(\mathcal{C}, M_\theta, D, G, \eta)$ of the problem defined in lemma 2 are available. Initially, a user defines a set of constraints $\mathcal{C}$ to be evaluated over a knowledge graph G and a machine learning model M_θ. Here the model is assumed to be already trained with parameters θ; for example, using the dataset D generated from the knowledge graph G. The generation process is tracked to create the sample-to-node mapping η. Here the semantics of the constraints are defined, such that approaches to a more efficient evaluation can be tackled in section 4.3.
As explained in section 4.1, the sample-to-node mapping allows connecting the prediction $M_\theta(\mathbf{x}_i)$ of a problem instance $\mathbf{x}_i$ with the validation result of a target shape ts in a shape schema $\mathcal{S}$ over the node $\eta(i)$ in a knowledge graph G. This connection was, therefore, incorporated into the definition of the constraint with the hint of a semantics similar to the implication used in Boolean formulas. Now, the semantics of a constraint is approached bottom-up and from the left-hand side to the right-hand side.
On the left-hand side of a constraint, there is the shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ annotated with a target shape $ts \in S$. The target shape denotes the shape, which is used during the validation (see definition 18) of the node $\eta(i)$. This is necessary because for each shape $s \in S$ there are validation results corresponding to $\eta(i) \in [[\texttt{TARG}(s)]]_G$. Evaluating the left-hand side of the constraint is, therefore, done by executing the entity validation function with the given parameters:

$$[[\mathcal{S}|_{ts}]]_{G,\eta(i)} = \begin{cases} \text{validate}(\eta(i), ts, G) & \eta(i) \in [[\texttt{TARG}(ts)]]_G \\ \top & \text{else} \end{cases} \tag{4.2}$$

As there might be instances retrieved by the seed query ($n \in [[Q_s]]_G$), which are not part of the target definition of the target shape defined for the constraint ($n \notin [[\texttt{TARG}(ts)]]_G$), the case distinction is necessary. These nodes n are handled as if they were valid according to the target shape.
On the right-hand side of the constraint, there is the logical expression ϕ_T involving one free variable T. To evaluate the formula, T needs to be substituted with the predicted result $M_\theta(\mathbf{x}_i)$ and then be evaluated. Formally, the evaluation is done over a structure $\mathbb{S}$ with $\mathbb{T}$ as a carrier, functions, relations and constants defined for $\mathbb{T}$ and an assignment β with $\mathcal{G}_\beta = \{(T \to M_\theta(\mathbf{x}_i))\}$. Therefore, it holds:

$$[[\phi_T]]_{M_\theta(\mathbf{x}_i)} = 1 \Leftrightarrow (\mathbb{S}, \beta) \models \phi_T \tag{4.3}$$

Next, the evaluation results need to be combined. The combination is done with the special symbol $\rightsquigarrow$ defined for the evaluation with a 3-valued logic designed for this application.

Definition 29: Interpreting formulas using the 3-valued logic
The 3-valued logic extends the Boolean logic by the introduction of a third symbol -1. The symbol is used to rate whether a formula σ is modeled by an interpretation $\mathcal{I} : var(\sigma) \to \{0, 1, -1\}$. Therefore it holds:

$$(\mathcal{I} \models \sigma) \in \{-1, 0, 1\}$$

-1 expresses that $\mathcal{I}$ does not apply to σ, 0 expresses that $\mathcal{I}$ is an invalid interpretation for σ and 1 expresses that $\mathcal{I}$ is an valid interpretation for σ.

To define the semantics of $\rightsquigarrow$, the Scott brackets are used for notation convenience.

Definition 30: The Semantics of $\rightsquigarrow$
Let A and B be formulas, which can be evaluated using the Scott brackets by $[[A]]_\alpha$ and $[[B]]_\beta$ and the necessary parameters α and β. Syntactically writing $A \rightsquigarrow B$ is allowed if $\forall \alpha, \beta \; [[A]]_\alpha, [[B]]_\beta \in \{-1, 0, 1\}$. Then the evaluation of $A \rightsquigarrow B$ denoted with $[[A \rightsquigarrow B]]_{\alpha,\beta}$ is defined as shown in the following truth table.

$[[A]]_\alpha$	$[[B]]_\beta$	$[[A \rightsquigarrow B]]_{\alpha,\beta}$
0	0	-1
0	1	-1
1	0	0
1	1	1

All further possible combinations of $[[A]]_\alpha$ and $[[B]]_\beta$ evaluated to -1.

The following example should make the evaluation of formulas using the $\rightsquigarrow$ of the 3-valued logic, and the interpretation coming with it, better understandable.

Example 12: Evaluating formulas of the 3-valued logic

This example is about a voting scenario. The candidates are *Armin Laschet* or *Olaf Scholz*. Each person permitted to vote is given a voting form with two boxes, one for each candidate. A vote is called compliant with the rules if exactly one box is checked. There are two formulas given.

Formula A evaluates to -1 if the vote was not done compliant with the rules, 0 if the vote was done compliantly for *Armin Laschet*, and 1 if the vote was done compliantly for *Olaf Scholz*.

Formula B evaluates to 1 if the person checked the box for *Armin Laschet* and otherwise evaluates to 0.

Let us assume person 1 only checks the box for *Olaf Scholz*, person 2 only checks the box for *Armin Laschet*, and person 3 checks both boxes. In this context, the interpretation $\mathcal{I}$ determines the boxes checked by providing the person.
Now we may be interested in evaluating $A \rightsquigarrow B$ ("Voting compliantly for *Olaf Scholz* is a requirement for checking the box for *Armin Laschet*") and $\neg B \rightsquigarrow A$ ("Not checking the box for *Armin Laschet* is a requirement to vote compliantly for *Olaf Scholz*") according to definition 30 for the different interpretations, given in

the form of persons. Table 4.2 and 4.3 shows the result of the evaluation.

p	$[[A]]_p$	$[[B]]_p$	$[[A \rightsquigarrow B]]_p$
Person 1	1	0	0
Person 2	0	1	-1
Person 3	-1	1	-1

Table 4.2: Evaluating $A \rightsquigarrow B$

p	$[[\neg B]]_p$	$[[A]]_p$	$[[\neg B \rightsquigarrow A]]_p$
Person 1	1	1	1
Person 2	0	0	-1
Person 3	0	-1	-1

Table 4.3: Evaluating $\neg B \rightsquigarrow A$

In the context of evaluating a constraint $C \in \mathbf{C}$, the evaluation is even simpler, as $\mathcal{S}|_{ts}$ and ϕ_T always evaluate to 0 or 1. Definition 31 concludes the process of evaluating a constraint C given a problem instance $\mathbf{x}_i$, a knowledge graph G, a machine learning model M_θ and a sample-to-node mapping η.

Definition 31: Constraint Evaluation given a Problem Instance
A constraint $C \in \mathbf{C}$ can be evaluated over a problem instance $\mathbf{x}_i$ given a trained machine learning model M_θ, a knowledge graph G and a sample-to-node mapping η with $[[C]]_{M_\theta(\mathbf{x}_i),G,\eta(i)}$.

As a remark, C is of form $\mathcal{S}|_{ts} \rightsquigarrow \phi_T$ and, therefore, evaluating C according to definition 31 means to apply formulas 4.2 and 4.3, and combine these results with definition 30. The whole evaluation process is demonstrated using the following example, building on the motivating example.

Example 13: Evaluating the Example Constraint

This example continues the motivating example and, therefore, makes use of the sample-to-node mapping η from example 7, created in example 11 from the knowledge graph G illustrated in Figure 1.2. The goal is to validate the constraint

$$\mathcal{S}|_{\texttt{:PersonShape}} \rightsquigarrow \phi_{\text{vaccinated}}$$

from example 8, given the already trained decision tree in Figure 1.3. Since only a part of the knowledge graph is given, the SHACL validation process will be done by reasoning using table 4.4:

person	#cw	:allergic_to	:gender	:pregnant	:country	:vaccinated
:Max	15	PEG	male	$\bot$	Germany	$\bot$
:Maria	25		female	$\top$	Germany	$\bot$
:Eva	30		female	$\bot$	Germany	$\top$
:Laura	10	PEG	female	$\bot$	Germany	$\bot$

Table 4.4: Dataset from 1.1 annotated with the number of contacts with non-vaccinated persons (#cw)

The evaluation of the left-hand side of the constraint is done by executing validate($\eta(i)$, :PersonShape, G) for all $i \in [1, ..., 9999]$. Therefore, $\mathcal{S}$ needs to be

validated against G by running a SHACL engine, like the one given in algorithm 1, to give validation results for all $i \in [1, ..., 9999]$. In this example, it is $[[Q_s]]_G \subseteq [[\texttt{TARG}(ts)]]$ as the seed query Q_s equals the target query of the shape ts (see example 8 and 10) and, hence, the else case in formula 4.2 does not occur. Inspection of the table above shows that only the nodes `:Maria_j` with $j \in [0, ..., 832]$ will be valid with respect to `:PersonShape`, because they are pregnant, live in Germany and have more than 20 contacts to non-vaccinated persons (e.g., fulfill `DEF(:PersonShape)`).
For the evaluation of the right-hand side, $\phi_{\text{vaccinated}}$ needs to be evaluated for all x_i with $i \in [1, ..., 9999]$. Hence, $[[\phi_T]]_{M_\theta(\mathbf{x}_i)} = 1$ iff $(M_\theta(\mathbf{x}_i) = \top)$. According to the dataset in example 7 and the decision tree in Figure 1.3, $\phi_{\text{vaccinated}}$ is true only for the instances $\mathbf{x}_{4167}, ..., \mathbf{x}_{9166}$, i.e., those that belong to `:Eva`. Table 4.5 includes the summarized evaluation results for each sample in the dataset.

i	$\eta(i)$	$[[\mathcal{S}\|_{\texttt{:PersonShape}}]]_{G,\eta(i)}$	$[[\phi_{vaccinated}]]_{M_\theta(\mathbf{x}_i)}$	$[[C]]_{M_\theta(\mathbf{x}_i),G,\eta(i)}$
1	`:Max_0`	0	0	-1
⋮	⋮	⋮	⋮	⋮
3333	`:Max_3332`	0	0	-1
3334	`:Maria_0`	1	0	0
⋮	⋮	⋮	⋮	⋮
4166	`:Maria_832`	1	0	0
4167	`:Eva_0`	0	1	-1
⋮	⋮	⋮	⋮	⋮
9166	`:Eva_4999`	0	1	-1
9167	`:Laura_0`	0	0	-1
⋮	⋮	⋮	⋮	⋮
9999	`:Laura_832`	0	0	-1

Table 4.5: Sample-to-node mapping annotated with the (intermediate results of the) constraint evaluation of the example constraint

Applying $[[C]]_{M_\theta(\mathbf{x}_i),G,\eta(i)}$ to all problem instances $\mathbf{x}_i$ in a dataset D and all constraints $C \in \mathcal{C}$ then yields a constraint validation result for each pair of sample and constraint. This result is recorded by the function defined in the following definition.

Definition 32: Model-Validation-Result Function
Given a set of constraints $\mathcal{C} \subset \mathbf{C}$ and a dataset D with N samples, the model-validation-result function

$$\Theta_{M_\theta,D,G,\eta} : \mathcal{C} \times [1, ..., N] \to \{1, 0, -1\}$$

maps a constraint C and an index i of a problem instance $\mathbf{x}_i$ in D to the validation result:

$$\Theta_{M_\theta,D,G,\eta}(C, i) \mapsto [[C]]_{M_\theta(\mathbf{x}_i),G,\eta(i)}$$

Θ assumes a trained machine learning model M_θ, a knowledge graph G, and a dataset D with the matching sample-to-node mapping $\eta \in \eta$. The set of all possible model-validation-result functions is defined as $\mathbf{\Theta}$. If the context is clear, Θ might be used instead of $\Theta_{M_\theta,D,G,\eta}$.

The model-validation-result function tells the user, whether the sample resp. the prediction made by the model is valid (1), or invalid (0) according to the constraint, or the constraint was not applicable (-1) to the sample resp. the prediction. In the context of a constraint as stated in definition 24, the semantics of $\rightsquigarrow$ allows for preserving the evaluation result of the left-hand side of the constraint. That is, a constraint does not apply to a sample, when ts is invalidated explicitly by the property path constraints defined in $\mathcal{S}.\texttt{DEF}(ts)$. In the remaining cases, the formula on the right-hand side of the implication determines the validation result. This is in contrast to using the standard Boolean implication:

Lemma 4: $\rightsquigarrow$ is closely related to $\rightarrow$
Reuse the notation from definition 30 and let $\rightarrow$ be the implication used in Boolean formulas, then for all possible A and B it holds $|[[A \rightsquigarrow B]]_{\alpha,\beta}| = [[A \rightarrow B]]_{\alpha,\beta}$.

The lemma can be used to convert Θ to use the standard Boolean implication and the 2-valued logic by taking the absolute value of each validation result. The 2-valued version of Θ is referred by $|\Theta|$ and defined analog to Θ but maps a constraint and the index of a sample in a dataset as follows:

$$\Theta_{M_\theta,D,G,\eta}(C,i) \mapsto \left|[[C]]_{M_\theta(\mathbf{x}_i),G,\eta(i)}\right| \tag{4.4}$$

4.2.3 The Validation Engine

This section serves as a summary of chapter 4.2 by providing pseudocode of the validation engine (algorithm 2), approaching the problem described in lemma 2, and showing how one might get the inputs of the problem. The algorithm assumes an inducer $\mathcal{I}$, a knowledge graph $G \subset \mathbf{G}$, the components (i.e., the dataset generating query Q_D and a target variable ?t) used to generate a dataset from G as well as the constraints to be checked $\mathcal{C} \subset \mathbf{C}$. These inputs are transformed into the input tuple of the problem described in lemma 2 and, afterward, the constraints are used to validate the so-created model, solving the problem described. Saying that, the user may skip the training of the machine learning model with the extracted dataset, by using an already trained model. However, this may lead to less useful visualizations based on the results of the engine, created in the coming sections. More details can be found as a final note in section 2.3.3.
The algorithm results in the model-validation-result function from definition 32. The algorithm is further improved in section 4.3. The following example demonstrates the algorithm line by line using the examples already provided.

Example 14: Applying Algorithm 2 to the Motivating Example

In a first step, the result of section 4.2.1, which is definition 27, is used to transform the dataset generating query Q_D and the variable $?t$ occurring in all solution mappings in $[[Q_D]]_G$ into the dataset D and the sample-to-node mapping η (line 2). The dataset generating query is the one provided in example 10, which is then used in example 11 to generate the dataset. By doing so, the sample-to-node mapping, as shown in example 7, is created on the fly. Afterwards, the learning algorithm $\mathcal{I}$ is used to train a machine learning model M with parameters θ using the generated dataset D (line 3). Here, it is assumed that the decision tree in Figure 1.3 is created, to solve the classification problem of "to get vaccinated or not to get vaccinated" given the dataset. Techniques to improve the quality of the model such as hyperparameter optimization, like finding the maximal depth of the decision tree as in example 5, or further steps to prepare the dataset are omitted. The division of the data set into a training set and a test set is also left out, but should be done as described in section 2.3.2 to be able to measure the generalization capabilities of the model.
At this point, the input tuple $(\mathcal{C}, M_\theta, D, G, \eta)$ to the limited problem of validating constraints over machine learning models are ready. As a first step to solve the problem, a SHACL engine is needed to evaluate the shape schemas $C.S$ given by the constraints $C \in \mathcal{C}$ (lines 9 - 13). The process of performing the SHACL validation for each constraint returns a dictionary of entity validation functions (see definition 18). Therefore, *validate* has one entry per distinct SHACL shape schema mentioned in the constraints (line 4). In this example, G needs only be checked against one shape schema, as only the constraint from example 8 is used.
The validation results for the `:PersonShape` are then used to evaluate the constraint for each sample (line 19). Therefore, the formula 4.2 is used in lines 20 - 24 and the formula 4.3 in line 25. Given definition 30, these results are combined (lines 26 - 32) to populate the result function Θ. This procedure is already demonstrated in example 13.

4.2.4 Complexity

Given the pseudocode of the validation engine, it is appropriate to analyze the time complexity of the approach. Inspection of the constraint validation engine yields three main parts to take into consideration:

1. Querying the dataset together with the sample-to-node mapping
2. Performing the SHACL validation
3. Evaluating the constraints

In the pseudocode, the first two parts are outsourced and hidden behind $[[Q_D]]_G$ and **runShaclEngine**. However, it should be noted, that even checking whether a given SPARQL mapping is contained in the set of answers of a query is PSPACE-complete, when not restricted to specific graph patterns [51]. The complexity of the problem can be reduced to be coNP-complete, by limiting the graph patterns to `OPTIONAL`, `FILTER`, and `AND`. Therefore, checking whether a SPARQL mapping will be retrieved by a SPARQL query

Algorithm 2 Pseudocode of the Validation Engine

function MAIN(Inducer $\mathcal{I}$, Query Q_D, Knowledge Graph G, Target Variable $?t$, Constraints $\mathcal{C}$)
    $(D, \eta) \leftarrow \mho(\text{tuple}([[Q_D]]_G), ?t)$
    $M_\theta \leftarrow \mathcal{I}(D)$ ▷ This is optional. A trained model M_θ might also be provided to the engine, instead of $\mathcal{I}$
    validate $\leftarrow$ **performShaclValidation**$(\mathcal{C}, G)$
    return evaluateConstraints$(\mathcal{C}, \textit{validate}, \mathcal{M}_\theta, \eta, G)$
end function
function PERFORMSHACLVALIDATION(Constraints $\mathcal{C}$, Knowledge Graph G)
    validate $\leftarrow \emptyset$
    for each $C \in \mathcal{C}$ **do**
        if $C.\mathcal{S} \notin$ *validate* **then**
            validate$[C.\mathcal{S}] \leftarrow$ **runShaclEngine**$(C.\mathcal{S}, G)$
        end if
    end for
    return *validate*
end function
function EVALUATECONSTRAINTS(Constraints $\mathcal{C}$, Entity Validation Function *validate*, Model $\mathcal{M}_\theta$, Sample-to-node Mapping η, Knowledge Graph G)
    $\mathcal{G}_\Theta \leftarrow \emptyset$
    for each $C \in \mathcal{C}$ **do**
        for each $i \in \{1, ..., \textbf{length}(D)\}$ **do**
            if $(\eta(i), C.ts, G) \in dom(\textit{validate}[\mathcal{C}.\mathcal{S}])$ **then**
                left $\leftarrow$ *validate*$[\mathcal{C}.\mathcal{S}](\eta(i), C.ts, G)$
            else
                left $\leftarrow \top$ ▷ Case: $\eta(i) \notin [[\texttt{TARG}(ts)]]_G$
            end if
            right $\leftarrow [[C.\phi_T]]_{M_\theta(D[i])}$
            if *left* $\wedge$ *right* **then**
                $\mathcal{G}_\Theta \leftarrow \mathcal{G}_\Theta \cup \{(C, i) \mapsto 1\}$
            else if *left* $\wedge \neg$*right* **then**
                $\mathcal{G}_\Theta \leftarrow \mathcal{G}_\Theta \cup \{(C, i) \mapsto 0\}$
            else
                $\mathcal{G}_\Theta \leftarrow \mathcal{G}_\Theta \cup \{(C, i) \mapsto -1\}$
            end if
        end for
    end for
    return $\mathcal{G}_\Theta$
end function

written according to formula 4.1 will be in coNP and the user may be advised to remove the `OPTIONAL` patterns when possible to make the problem solvable in polynomial time.
Also, checking a SHACL schema against a knowledge graph is NP-complete in general [15]. Therefore, the user of the engine should choose SHACL schemas non-recursively or without negations in recursive dependencies to reduce the complexity of the SHACL schema evaluation to polynomial time.
This result is not surprising, as in algorithm 1 a SAT solver is used. The main difference between the engine in algorithm 1 and the one implemented in SHACL2SPARQL [14]

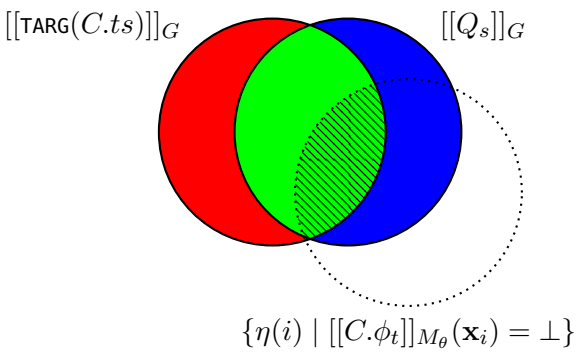


Figure 4.1: Venn-Diagram visualizing the Different Kinds of Nodes Occurring during the Validation

or Trav-SHACL [21] is that the saturation performed by the SAT solver is performed interleaved with the materialization and grounding of the rules.
This leaves the third part to be performed by the implementation of the constraint validation engine. Algorithm 2 proves that evaluating the constraints, given the necessary components, as in the **evaluateConstraints** function, can be done with a time complexity of $O(|\mathcal{C}| * |D|)$ assuming constant length constraints and a data structure like a hash table to access the SHACL validation result in constant time.
Overall, the complexity of the validation engine depends on the complexity arising through querying the knowledge graph and performing the SHACL validation. In a worst-case scenario, this results in the problem being PSPACE-complete. However, wisely choosing the SPARQL query to retrieve the data and the constraints yields a polynomial run time, on which further extensions and improvements can be built.

4.3 Improving and Extending the Approach

In this section, the aim is to improve the previously presented approach with heuristics to reduce the time needed by the validation engine. First, heuristics are presented to reduce the number of nodes in the knowledge graph, which need to be validated during the SHACL schema validation of the different constraints. In this first section, it is also presented how redundant validation runs for shape schemas used by multiple constraints are avoided. Next, heuristics are worked out to reduce the time spent on joining SHACL schema validation results with the samples in the dataset via the sample-to-node mapping. The third section investigates performing the SHACL constraint validation during SPARQL query processing, which uses the heuristics from the first section to improve the runtime. Finally, a new type of constraint is introduced and distinguished from the old one.

4.3.1 Reducing the SHACL Shape Schemas

An inspection of the validation engine given in algorithm 2 shows that the information needed by the validation engine with respect to the SHACL validation results is communicated too broad to the SHACL engine. In the context of evaluating constraints over a machine learning model, the calls to the entity validation function in line 21 specify the need of the algorithm for SHACL validation results.
The entity validation function is called for every combination of a constraint C and a

node $\eta(i)$ related to a sample in the dataset. That is, the evaluation only requires the validation results for the nodes retrieved, when executing the seed query Q_s, used during creation of the dataset, over G (in Figure 4.1 the right circle). Additionally, the algorithm only makes use of the entity validation function if the entity validation function contains the needed validation result. For a given constraint C with a target shape, $C.ts$ this is only the case if the node is included in the target definition of the shape (left circle in Figure 4.1). Finally, one can exploit the form of the constraint C in the case of the 2-valued logic. Taking definition 24 and lemma 4 brings C in the form

$$\mathcal{S}|_{ts} \rightarrow \phi_T$$

Because of the Boolean implication now used in C, if $[[C.\phi_t]]_{M_\theta}(\mathbf{x}_i)$ evaluates to true the whole constraint will also evaluate to true independent of the SHACL schema validation result generated for the evaluation of the left-hand side of the constraint (dotted circle in Figure 4.1).
As the entity validation function is created from a valid assignment, at most the validation results for the target nodes of the shapes in the shape schema are included, which are too many. This can be seen in Figure 4.1. The area in blue represents the set of nodes originally not included in the target definition of the target shape. The red area, as well as the not dotted green area (in the case of the 2-valued logic), represent the nodes validated by the SHACL engine, although, the results are not needed by the validation engine to validate the constraints.

Lemma 5: Only Perform the SHACL Validation for Target Nodes Needed by the Validation Engine
Given a constraint $C \in \mathbf{C}$ (as in definition 24), a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$, a seed query $Q_s \in \mathbf{Q_T}$ used to retrieve the dataset D and the sample-to-node mapping η with problem instances $\mathbf{x}_i$ from a knowledge graph $G \in \mathbf{G}$, the nodes needed to be validated by a SHACL engine to perform the constraint validation of C can be limited to

$$H_2 = [[\texttt{TARG}(C.ts)]]_G \cap [[Q_s]]_G \cap \{\eta(i) \mid [[C.\phi_t]]_{M_\theta(\mathbf{x}_i)} = \bot\} \tag{4.5}$$

in case of the 2-valued logic (the dotted area marked in green in Figure 4.1) or

$$H_3 = [[\texttt{TARG}(C.ts)]]_G \cap [[Q_s]]_G \tag{4.6}$$

in case of the 3-valued logic (the area marked in green in Figure 4.1).

Therefore, the goal is now to reduce the SHACL schema $C.\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ to a SHACL schema $\mathcal{S}' = (S', \texttt{TARG}', \texttt{DEF})$, in such a way, that the validation results for the instances in H_3 resp. H_2 does not change. This can be formally expressed with

$$\forall G \in \mathbf{G}\ \forall v \in H\ \ validate_{\mathcal{S}}(v, C.ts, G) = validate_{\mathcal{S}'}(v, C.ts, G) \tag{4.7}$$

where $validate_{\mathcal{S}}$ is the entity validation function for $\mathcal{S}$ and H is H_2 or, H_3 depending on the logic used.
To remove the kind of instances validated unnecessarily by the SHACL engine, the target definition of the target shape $C.ts$ can be limited to the intersection H. Therefore, the seed query Q_s is extracted from the dataset generating query Q_D by only projecting $?x$:

$$Q_s = \texttt{SELECT}(?x, id, Q_D) \tag{4.8}$$

Next, in case of the 2-valued logic the nodes in the set $\{\eta(i) \mid [[C.\phi_t]]_{M_\theta}(\mathbf{x}_i) = \bot\}$ can be estimated by early evaluating the right-hand side of the constraint and making use of the sample-to-node mapping η. Given the IRIs of the nodes, a filter condition R can be built to restrict the validation to these nodes. In the case of the 3-valued logic, it holds $R = \top$. Both Q_s and R can now be used in the target definition of the target shape $C.ts$ of the reduced shape schema:

$$\texttt{TARG}'(s) = \begin{cases} \texttt{TARG}(s) & s \neq C.ts \\ \texttt{SELECT}(?x, id, ((Q_s \texttt{ AND } \texttt{TARG}(C.ts)) \texttt{ FILTER } R)) & \text{else} \end{cases}$$

Clearly, the change of TARG(C.ts) does not affect formula 4.7.
The execution of the SHACL engine over a shape schema, as performed in line 11, produces *at most* the validation results for the target nodes of the shapes in the shape schema. However, the evaluation of a constraint only needs the SHACL validation results for the shape $C.ts$ and further results should only be produced because of inter-shape constraints. This is the second kind of instances validated unnecessarily by the SHACL engine. These shapes can be identified by taking the shape schema $\mathcal{S}$ as a directed graph, as defined below:

Definition 33: Dependency Graph of a Shape Schema [21]
Given a shape schema, $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ the directed dependency graph $\Phi_\mathcal{S}$ is a tuple $(V_{\Phi_\mathcal{S}}, E_{\Phi_\mathcal{S}})$ with

$$V_{\Phi_\mathcal{S}} = S$$

and

$$E_{\Phi_\mathcal{S}} = \{(s_i, s_j) \mid s_j \text{ appears in } \texttt{DEF}(s_i)\}$$

Given, $\Phi_\mathcal{S}$ one can identify the needed shapes S' to be the ones reachable from $C.ts$. This is valid as the criterion $\texttt{DEF}(C.ts)$ can still be evaluated as before because formula 4.7 stays unaffected.

Lemma 6: Remove unneeded Shapes from the Shape Schema
Given a constraint $C \in \mathbf{C}$ (as in definition 24) and a shape schema, $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ the shapes to evaluate C can be limited to the shapes, which are reachable from $C.ts$ in the directed dependency graph $\Phi_\mathcal{S}$.

Algorithm 3 makes use of the two lemmas and is a replacement for **performShaclValidation** (line 4 in algorithm 2). Additionally, the algorithm takes care of shape schemas used by multiple constraints. Taking multiple constraints into consideration is important as each shape schema should only be validated once over the knowledge graph, to avoid the redundant generation of SHACL schema validation results for shapes occurring in multiple reduced shape schemas.

Lemma 7: Simultaneous Generation of SHACL Schema Validation Results for Constraints using the Same Shape Schema
Given a set of constraints $\mathcal{C} \subset \mathbf{C}$ the SHACL schema validation results for all $\mathcal{C}_\mathcal{S} \in \mathcal{P}(\mathcal{C})$ for which all $C_i, C_j \in \mathcal{C}_\mathcal{S}$ imply $C_i.\mathcal{S} = C_j.\mathcal{S}$ should be generated simultaneously.

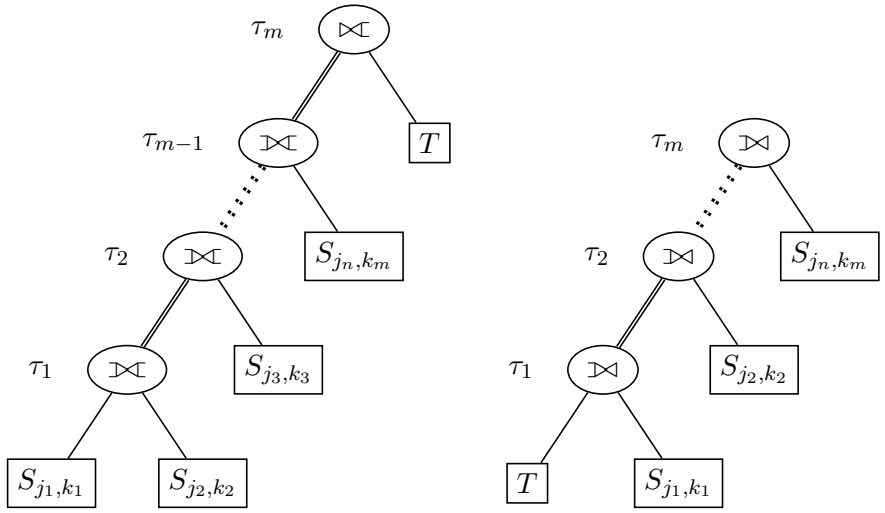
However, lemmas 5 and 6 can both be extended easily to multiple constraints. A group of constraints using the same shape schema $\mathcal{S}$, but different target shapes ts_i ($i \in [1, 2, ...]$), needs for evaluation all shapes reachable from the ts_i (lines 3 - 11). These are stored in *relevantShapes*. In the next steps, for each shape schema occurring in the constraints (line 13), a subset of the target shapes must be selected for which the target definition can be reduced (lines 15 - 20). That is the target definition of a target shape ts, involved in the evaluation of another target shape ts' may not be reduced, because the nodes $[[\texttt{TARG}(C.ts)]]_G \setminus [[Q_s]]_G$ may be needed to evaluate ts'. In case the target definition of a target shape can be reduced, the filter condition used to further reduce the nodes in the target definition is computed per target shape. To avoid the evaluation of the right-hand side of the constraints, the SHACL validation needs to be performed interleaved with the constraint evaluation. Here for reasons of simplicity, line 17 can be thought of an oracle giving the correct filter condition. Finally, the SHACL engine can be run on the reduced shape schema $\mathcal{S}'$ consisting of the updated target definitions TARG and the shapes S' relevant for evaluation of the target shapes ts_i (line 21).

Algorithm 3 Pseudocode of the Reduced SHACL Validation

```
1: function REDUCEDSHACLVALIDATION(Constraints C, Query Q_D, Knowledge Graph G)
2:     schemasForConstraints ← ∅ ▷ the list of constraints using the same shape schema
3:     relevantShapes ← ∅ ▷ the needed shapes per shape schema
4:     involvedShapes ← ∅ ▷ the shapes involved when evaluating a constraint
5:     Q_s = SELECT(?x, id, Q_D) ▷ id is the identity function
6:     for each C ∈ C do
7:         Φ_{C.S} ← createDependencyGraph(C.S)
8:         involvedShapes[C] ← DFS(Φ_{C.S}, C.ts)
9:         relevantShapes[C.S] ← relevantShapes[C.S] ∪ involvedShapes[C]
10:        schemasForConstraints[C.S] = schemasForConstraints[C.S].append(C)
11:    end for
12:    validate ← ∅
13:    for each S ∈ schemasForConstraints do
14:        TARG ← S.TARG
15:        for each C ∈ schemasForConstraints[S] do
16:            if C.ts ∉ ⋃_{c∈schemasForConstraints[S]\{C}} involvedShapes[c] then
17:                R ← getFilterConditionForConstraint(C)
18:                TARG(C.ts) ← SELECT(?x, id, ((Q_s AND TARG(C.ts)) FILTER R))
19:            end if
20:        end for
21:        validate[S] ← runShaclEngine((relevantShapes[S], TARG, S.DEF), G)
22:    end for
23:    return validate
24: end function
```

4.3.2 How to Join the SHACL Validation Results with the Dataset

In the complexity analysis of the **evaluateConstraints** function in algorithm 2, it is assumed that all the validation results fit into the main memory and, therefore, a hash table can be used to access the SHACL validation results in one step. However, this might not be the case, when a large dataset with many seed nodes combined with various



(a) Join T at the end (b) Join T at the beginning

Figure 4.2: Execution Trees with Intermediate Results τ and Double Lines Denoting Pipelining

constraints should be handled.
This problem can be tackled by taking the different components involved as relations in a relational schema, which in turn allows using appropriate physical operators known from relational databases. First, there is the sample-to-node mapping η and the dataset D retrieved from a knowledge graph G. These two can be handled as one relation $T(\underline{\text{idx}}$, node_id, x, y). Each entity of T represents a sample $(\mathbf{x_i}, y_i)$ in D with the corresponding node_id $\eta(i)$, identified by an index i ($i \in [1, ..., N]$ and N the number of samples in D). Next, each SHACL validation corresponding to a constraint $C_j \in \mathcal{C}$ of a schema $\mathcal{S}_j$ with target shape $ts_{j,k}$ yields a relation $S_{j,k}(\underline{\text{node\_id}}, \text{result}_{j,k})$, where $j \in [1, ..., m]$; $k \in [1, ..., |\mathcal{S}_j.S|])$ and $m = |\mathcal{C}|$ the number of constraints. The entities of $S_{j,k}$ represent the SHACL validation results *validate*(node_id, $ts_{j,k}$, G) for each seed node identified by a node_id $\in [[\mathcal{S}_j.\texttt{TARG}(ts_{j,k})]]_G$.
During the evaluation of the constraints for each sample in the dataset, the SHACL validation results are needed, which expresses the need for the relations joined with respect to the seed nodes respectively the node identifiers. While performing the join, special attention to the $S_{j,k}$ relations is required as there will be node identifiers without a SHACL validation result for some j, k as only nodes in the corresponding target definition of the target shape are handled. Therefore, outer-joins must be used to retain all dataset indices and validation results. Restricting the selection of query execution trees to left-deep trees yields two different join strategies, which are depicted in Figure 4.2.
In the present context, applying the reductions from section 4.3.1 when possible, it can be seen that the following equations hold:

$$|T| = |[[Q_D]]_G| = |D| \tag{4.9}$$

$$|S_{j,k}| = |[[\mathcal{S}_j.\texttt{TARG}(ts_{j,k})]]_G| \tag{4.10}$$

$$|\tau \ltimes S_{j,k}| = |\pi_{\text{node\_id}}(\tau) \cup \pi_{\text{node\_id}}(S_{j,k})| \geq |S_{j,k}|, |\tau| \tag{4.11}$$

$$|T \bowtie \tau| = |\tau \ltimes T| = |T| \geq |\tau| \tag{4.12}$$

where τ denotes an arbitrary intermediate result occurring in the execution trees.
Formula 4.12 is conceptually more involved, that is why it is proved below:

physical operator for $R \underset{A=B}{\bowtie} S$	requirements	read cost
Index join	Index on B in S and B is unique in S	$\|R\| + \|R\|(k + log(\#(B, S)))$ where k denotes the average number of B matches in S given the A values from R.[1]
Sort join	R and S are sorted w.r.t A resp. B	$\|R\| + \|S\|$
Hash join	Requires S to match into memory, in one or vertical partitioned	$\|R\| + \|S\|$

Table 4.6: Simplified Cost Model for Different Physical Join Operators

Proof. Each part of the equation can be proved on its own:
1.) $|T \rtimes \tau| = |\tau \ltimes T|$ is true for all possible relations T and τ because of symmetry.
2.) $|\tau \ltimes T| = |T|$ is proved by reasoning over τ_j ($l \in [1, ..., m]$) given the execution trees in Figure 4.2.
In case of Figure 4.2a, it is $\tau_1 = (S_{j_1,k_1} \ltimes S_{j_2,k_2})$ and $\tau_l = (\tau_{l-1} \ltimes S_{j_{(l+1)},k_{(l+1)}})$. We note that for each $S_{j,k}$ the node_id is unique. Therefore the node_id will be unique for all l in τ_l. T contains all *node_id*'s which can occure at least once (see lemma 5); implying $\forall l\ \pi_{\text{node\_id}}(\tau_l) \subseteq \pi_{\text{node\_id}}(T)$. This is sufficient for $\forall l\ |\tau_l \ltimes T| = |T|$ to be true in case of Figure 4.2a.
In case of Figure 4.2b, it is $\tau_1 = (T \rtimes \mathcal{S}_{j_1,k_1})$ and $\tau_l = (\tau_{l-1} \rtimes S_{j_l,k_l})$. With the argumentation as above we can see that $\forall l\ \pi_{\text{node\_id}}(\mathcal{S}_{j_l,k_l}) \subseteq \pi_{\text{node\_id}}(T)$. Clearly $\pi_{\text{node\_id}}(\tau_1) = \pi_{\text{node\_id}}(T)$ applies and $\forall l\ \pi_{\text{node\_id}}(\tau_l) = \pi_{\text{node\_id}}(T)$ follows by induction over l. Again, this is sufficient for $\forall l\ |\tau_l \ltimes T| = |T|$ to be true in case of Figure 4.2b.
3.) $|T| \geq |\tau|$ can be proved with the same arguments as **2.)**. □

Equations 4.9 and 4.10 put the cardinality of the relations in the context of the validation engine. That is, the cardinality of T is equal to the cardinality of the dataset retrieved from the given knowledge graph and the cardinality of $S_{j,k}$ is equal to the number of nodes included in the target definition of $ts_{j,k}$.
Next, both execution plans in Figure 4.2 are analyzed separately using the cost model in Table 4.6 with respect to the order in which the $S_{j,k}$ relations are best joined and which execution plan should be chosen.

a) Join T at the end With the equations 4.11 and 4.12 it follows that during the execution of the operation tree the intermediate results are increasing monotonously up to $|T|$.
In case of the index join, the cost can be approximated to be:

$$\underbrace{|\tau_{m-1}| + |\tau_{m-1}| * \left(\frac{|T|}{\#(\text{node\_id}, T)} + log(\#(\text{node\_id}, T))\right)}_{\text{read cost } \ltimes}$$

[1] $\#(B, S)$ denotes the number of different B values in S

$$+ \underbrace{\sum_{i=0}^{m-2} [|\tau_i| * (2 + log(\#(\text{node\_id}, S_{j_{i+2},k_{i+2}}))]}_{\text{read cost } \ltimes} + \underbrace{|\tau_m| - \sum_{i=1}^{m-1} [|\tau_i|]}_{\text{write cost - pipelining}}$$

$$= |\tau_{m-1}| * \left(\frac{|T|}{|[[Q_s]]_G|} + log(|[[Q_s]]_G|) \right) + \sum_{i=0}^{m-2} [|\tau_i| * (1 + log(|S_{j_{i+2},k_{i+2}}|))] + |T| + |\tau_0|$$

where $\tau_0 = S_{j_1,k_1}$, $|\tau_m| = |T|$, and the node_ids are unique for all $S_{j,k}$ and in each τ_i. Therefore, to simplify the calculation, the k in the formula for the read cost of the index join is set to 1, when joining with a S. In the join with, T it is used that the node_ids in T correspond to the seed nodes in the sample-to-node mapping, which can be retrieved with the Q_s generated with equation 4.8.
In case of the sort or the hash join, the cost can be approximated to be:

$$\underbrace{(|\tau_m - 1| + |T|) + \sum_{i=0}^{m-2} [|\tau_i| + |S_{j_{i+2},k_{i+2}}|]}_{\text{read cost}} + \underbrace{\sum_{i=1}^{m} [|\tau_i|]}_{\text{write cost}}$$

where $\tau_0 = S_{j_1,k_1}$ and $|\tau_m| = |T|$. However, these types of joins do not support pipelining as the final result is written multiple times until the final result is ready. The sort join comes with the additional requirement of sorted input relations, adding a cost in the order of $O(m * |T| * log(|T|))$.
Therefore, considering the cost complexity of the physical join operators, the following heuristic to minimize the cost arises:

Lemma 8: Join-Heuristic 1
When joining T at the end, the $j_1, ..., j_m$ and $k_1, ..., k_m$ in Figure 4.2a should be chosen such that the cardinality of the intermediate results $\tau_1, ..., \tau_{m-1}$ are as small as possible.

One might think about converting the left-deep execution tree into a right-deep execution tree to reduce the cost[2] to:

$$\underbrace{|T| * (2 + log(\#(\text{node\_id}, \tau_{m-1}))) + \sum_{i=0}^{m-2} [|S_{j_{i+2},k_{i+2}}| * (2 + log(\#(\text{node\_id}, \tau_i))]}_{\text{read cost}}$$
$$+ \underbrace{|\tau_m| - \sum_{i=1}^{m-1} [|\tau_i|]}_{\text{write cost - pipelining}}$$

where the k in the index join is again set to 1, assuming the maximal number of matches, when joining with an intermediate result τ_i. However, this involves maintaining the index structure of the intermediate results. Assuming a b-tree is used for indexing, this would yield an additional cost in the size of $O(|T| * log(|T|))$, which might lead to a worse execution time.

[2] Assuming $|\tau_i| > |S_{j_{i+2},k_{i+2}}|$ for enough i's

b) Join T at the beginning The execution plan leads to a constant cardinality of the intermediate results of $|T|$ right from the beginning (see 4.12). Therefore, the cost calculation will be analog to the last paragraph but with the difference that all $|\tau_i|$ are as worse as possible, i.e., $|T|$. Further, an index structure over the node_ids in T will be useful as it can be used in every step of the execution plan. On the good side, it allows joining the SHACL validation results incrementally with T and, therefore, interleaving the join and the SHACL validation will yield usable results right after the first SHACL schema is validated, and the results are joined.

Lemma 9: Join-Heuristic 2
T should be joined in the end or as early as intermediate SHACL validation results in combination with the dataset indices are needed.

4.3.3 Performing SHACL Constraint Validation during SPARQL Query Execution

Although validation of a knowledge graph builds on querying, when performed over a SPARQL endpoint, the sections 2.2.3 and 2.2.4 introduced querying and validating a knowledge graph as two different concepts. This section presents the concept of validating SHACL constraints during SPARQL query execution as proposed by Rohde et al. [58] as a possibility to improve the validation engine in algorithm 2. Originally, SHACL constraint validation is done during SPARQL query execution to "... [increase] the explainability of SPARQL query results by annotating them with information from the SHACL shape schema validation." [58]. Here the inspection of algorithm 2 shows that the SHACL schema is evaluated over the knowledge graph (line 11) and afterwards combined (lines 18 - 34) with the query results retrieved before (line 2). As shown by Rohde et al., performing both task simultaneously allows to repeatedly exploit the knowledge encoded in the SPARQL query like shown in section 4.3.1.
Next, the notation from section 2.2.3 is extended to perform the SHACL validation from section 2.2.4 during query execution. Afterward, the pseudocode used to repeatedly exploit the knowledge encoded in the seed query Q_s is presented.
In comparison to the notation introduced in [58], the concept of the explanation and the solution mapping is decoupled in a first step and instead there is an explanation function, which is able to map the needed validated RDF terms to a set of validation results.

Definition 34: Explanation Mapping
An explanation mapping E is a partial function, which maps an RDF term $t \in (\mathbf{B} \cup \mathbf{L} \cup \mathbf{I})$ to a set of validation results for that term. A validation result consist of the shape $s \in \mathbf{S}$ against which the node is validated and the validation result, which can be valid ($\top$) or invalid ($\bot$). Therefore it's

$$E : \mathbf{B} \cup \mathbf{L} \cup \mathbf{I} \to \mathcal{P}(\mathbf{S} \times \{\top, \bot\})$$

Two explanation mappings E_1 and E_2 can be united via $E_1 \cup E_2$ by merging the partial functions as follows:

$$E_1 \cup E_2 = \{(t \mapsto E_1(t) \cup E_2(t)) \mid t \in (\mathbf{B} \cup \mathbf{L} \cup \mathbf{I})\}$$

where all unspecified values t are assumed to have as default value the empty set. The infinite set of explanation mappings is denoted with $\mathbf{E}$.

To annotate the query results, it is necessary to extend the notion of a set of solution mappings as introduced in section 2.2.3 with the explanation defined above.

Definition 35: A modified valSPARQL Mapping Set
A modified valSPARQL mapping set is a triple (Ω, E, sn) where $\Omega \subset \mathbf{M}$, $E \in \mathbf{E}$ and $sn \in \mathbf{SN}$.

As in section 2.2.3 several algebraic operations over the solution sets need to be defined. The solution sets are now modified valSPARQL mapping sets instead of sets of SPARQL solution mappings.

Definition 36: Algebraic Operations over Modified valSPARQL Mapping Sets
Given two modified valSPARQL mapping sets $F_1 = (\Omega_1, E_1, sn_1)$ and $F_2 = (\Omega_2, E_2, sn_2)$, a set of variables $v \subset \mathbf{V}$, a filter condition R and a function used for renaming of variables $\nabla : \mathbf{V} \to \mathbf{V}$ the following operations are defined:

operation	Ω	$\mathcal{G}_E$	sn
$\pi_v(F_1)$	$\pi_v(\Omega_1)$	$\{(\mu(x) \mapsto E_1(\mu(x))) \mid x \in v \wedge \mu \in \Omega\}$	sn_1
$\sigma_R(F_1)$	$\sigma_R(\Omega_1)$	$\{(s \mapsto E_1(s)) \mid s \in \mathrm{dom}(\mu) \wedge \mu \in \Omega\}$	sn_1
$F_1 \cup F_2$	$\Omega_1 \cup \Omega_2$	$E_1 \cup E_2$	$sn_1 \cup sn_2$
$F_1 \bowtie F_2$	$\Omega_1 \bowtie \Omega_2$	$E_1 \cup E_2$	$sn_1 \cup sn_2$
$F_1 \setminus F_2$	$\Omega_1 \setminus \Omega_2$	$\{(s \mapsto E_1(s)) \mid s \in \mathrm{dom}(\mu) \wedge \mu \in \Omega\}$	sn_1
$\rho(\nabla, F_1)$	$\rho(\nabla, \Omega_1)$	E_1	sn_1

Given the operation in the left most column, the result is a new valSPARQL mapping set (Ω, E, sn).

The next step is to build the explanation mapping during SPARQL query evaluation. This is done inductively over the graph pattern expressions and analog to definition 11. Therefore, the induction starts with the evaluation of a triple pattern, which utilizes a modified version of the entity explanation function defined in [58].

Definition 37: Entity Explanation Function
The entity explanation function $\exp : \mathbf{M} \times \mathbf{T} \times \mathbf{SN} \times \mathbf{G} \to \mathbf{E}$ maps the substituted subject s of a triple pattern t, given a mapping μ to an explanation mapping with respect to the matching shapes $sh \in S$ from a shape schema $sn = (S, \texttt{TARG}, \texttt{DEF})$ over a knowledge graph G.

$$\exp(\mu(t), t, sn, G) = \{(s \mapsto \{(sh, \mathrm{validate}(s, sh, G)) \mid sh \in S \wedge s \in [[\texttt{TARG}(sh)]]_G\})\}$$

where $s = \mu(t)[0]$

Now the evaluation of a triple pattern with respect to a shape schema and a knowledge graph can be defined as

$$[[t]]_G^{sn} = ([[t]]_G, \bigcup_{\mu \in [[t]]_G} \exp(\mu(t), t, sn, G), sn)$$

The further steps of the induction stay the same as in definition 11 but now use the algebraic operations over the modified valSPARQL mapping sets. Therefore, all occurrences of $[[P]]_G$ in definition 11 need to be replaced with $[[P]]_G^{sn}$ where $sn \in \mathbf{SN}$, $P \in \mathbf{P}$ and $G \in \mathbf{G}$. This concludes the notation needed to perform SPARQL query execution during SHACL constraint validation.

Definition 38: Execution of Q over G while validating w.r.t. sn
The execution of the SPARQL `SELECT` query Q over the knowledge graph G while validating w.r.t. a shape schema sn is written as a function $[[Q]]_G^{sn} : \mathbf{Q} \times \mathbf{G} \times \mathbf{SN} \to \mathcal{P}(\mathbf{M}) \times \mathbf{E} \times \mathbf{SN}$ and gives a modified valSPARQL mapping set.

Finally, to make up the decoupling in the beginning, the query results Ω can be joined with the modified valSPARQL explanation E to get the annotated query results as in [58]:

$$\Omega \bowtie E = \{(\{(t, \mu(t)) \mid t \in \mathrm{dom}(\mu)\}, \{(t, sh, val) \mid (sh, val) \in \mathrm{E}(t) \wedge t \in \mathrm{dom}(\mu)\}) \mid \mu \in \Omega\} \tag{4.13}$$

Clearly, this step might also be pushed down into the evaluation of the query, which would result in a slight change of definition 36 to handle SPARQL mapping sets joined with the explanation mapping [58].
The following describes the pseudocode of the valSPARQL approach (algorithm 4). The approach decomposes the given query Q into so-called subject star-shaped queries (line 2), whose results are joined afterwards according to the algebraic operations (see definition 9 in [58]) (line 11). A subject star-shaped query is a non-nested query consisting of the conjunction of triple patterns with the same subject [66]. The pseudocode uses such a decomposition and additionally performs the join (equation 4.13) per star-shaped query result mapping set (line 9). During the decomposition, for each star-shaped query, a shape s is determined. This is done by associating each star-shaped query with a class of entities in the knowledge graph and inferring the shape in the shape schema associated with that class. Besides performing SHACL validation during query processing, also the SHACL schema is reduced as described in section 4.3.1 (lines 4 - 7).

Algorithm 4 Pseudocode of the valSPARQL Approach

```
1:  function valSPARQL(Query Q, Shape Schema S, Knowledge Graph G)
2:      plan ← generateBushyExecutionPlan(Q, S)        ▷ ssq is a star-shaped query
3:      for each (ssq, s) ∈ plan do                    ▷ s is the shape chosen for ssq
4:          sn_i ← S
5:          Φ_{sn_i} ← createDependencyGraph(sn_i)
6:          sn_i.S ← DFS(Φ_{sn_i}, s)
7:          sn_i.TARG(s) ← SELECT(?x, id, (ssq AND sn_i.TARG(s)))
8:          (Ω_i, E_i, sn_i) ← [[ssq]]_G^{sn_i}
9:          M_i ← Ω_i ⋈ E_i
10:     end for
11:     return applyOperators(plan, M_1, ...)
12: end function
```

The approach described in this section combines the process of the execution of a SPARQL query with the evaluation of a SHACL schema over a knowledge graph. In the validation engine (algorithm 2) the dataset constructing query Q_D is executed without evaluating a SHACL schema. However, the constraints $\mathcal{C}$ are already known and, therefore, one may decide to early evaluate a SHACL schema $\mathcal{S}$ coming from a set of constraints $C \subset \mathcal{C}$ with the target shapes ts_i ($i \in [1, 2, ...]$) matching the shapes s assigned to the star-shaped queries.

4.3.4 Different Types of Constraints

In definition 24 a constraint is defined to be of a specific form, which makes use of a SHACL shape schema associated with a target shape as a condition and a logical expression about the predicted target as a restriction. This kind of constraint can be used during inference to explain and check the model's predictions. When the condition of the constraint applies to the problem instance and its semantic context in the knowledge graph, the validation result grades the prediction made by the model to be correct or incorrect according to the constraint. If the validation result confirms the model's prediction, the constraint gives an explanation for the prediction. If the prediction turns out to be incorrect, the result might be used to identify the reason for that. However, in both cases, the explanation or the reasoning is only necessarily true if the underlying knowledge graph is correct. Given the application, this kind of constraint is now referred to as *prediction constraint*.

Definition 39: Prediction Constraint
The type of constraint as defined in definition 24, which, when evaluated over a machine learning model given the underlying knowledge graph and the sample-to-node mapping, makes a statement about the correctness of the predictions of the model, by assuming a correct data basis is called prediction constraint.

This is in contrast to a constraint, which does not take into consideration the predictions of the model. As no statement is made about the truth of the model's predictions, the constraint validation results are only about the integrity of the semantic data in the knowledge graph. It is the samples in the dataset, which are now validated or invalidated with respect to the semantic context given in the knowledge graph. Therefore, each constraint validation result measures the trustworthiness of the sample in the dataset.
Clearly, there are different possibilities to represent such a constraint. In some cases, it might be enough to mark the ith sample as valid/invalid if the associated node $\eta(i)$ is retrieved by a SPARQL query and else as invalid/valid. However, using a SHACL shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ to represent constraints over a knowledge graph is the W3C recommended approach and is at least as expressive as a SPARQL query (see below). The definition is done analog to a prediction constraint, but only uses the condition part of the implication.

Definition 40: Data Constraint
A constraint $C \in \mathbf{C}$ is called a data constraint if the constraint only measures the trustworthiness of the samples in the dataset, by looking at their feature values and their semantic context given in the knowledge graph.
Here every data constraint C is composed of a shape schema $\mathcal{S} = (S, \texttt{TARG}, \texttt{DEF})$ ($\mathcal{S} \in$

SN) and a target shape $ts \in S$. The constraint is serialized as follows:

$$\mathcal{S}\big|_{ts}$$

The components of C can be accessed with $C.\mathcal{S}$ and $C.ts$.

To stay with the 3-valued logic and the notation known from the prediction constraints, a data constraint C is evaluated with respect to a knowledge graph G and the node $\eta(i)$, that uses the sample-to-node mapping η applied to the i'th sample in a dataset, using the Scott-brackets:

$$[[C_d]]_{G,\eta(i)} = \begin{cases} \text{validate}(\eta(i), ts, G) & \eta(i) \in [[\texttt{TARG}(ts)]]_G \\ -1 & \text{else} \end{cases} \tag{4.14}$$

Using the 3-values logic comes with the benefits of more detailed validation results. Precisely, a data constraint only applies to a seed node and the associated sample, when the seed node is included in the target definition of the specified target shape of the constraint. If a data constraint labels a sample as valid or invalid, that is because the shape applies or does not apply to the related seed node.
For example, a data constraint C_Q can be used to mark the instances retrieved by a query Q as 'valid' (1) and the rest as 'not applicable' -1, by setting $C_Q.\mathcal{S} = (\{s_1\}, \{s_1 \mapsto Q\}, \{\})$. The example shows that using SHACL for data constraints is at most as expressive as a constraint, which would only be based on a SPARQL query.
Again, the 3-valued logic can be transformed into the 2-valued logic by taking the absolute value. However, the transformation may lead to misleading results as it might be the case that every sample seems to be valid, although, in fact, not a single sample was validated. Further, the transformation shows the similarity of the left-hand side $\mathcal{S}\big|_{ts}$ of a prediction constraint defined in definition 24 with a *data constraint* C:

$$\left|[[C]]_{G,\eta(i)}\right| = [[\mathcal{S}\big|_{ts}]]_{G,\eta(i)} \tag{4.15}$$

Although the validation results of a data constraint only make a statement about the validity of the samples in the dataset and not the predictions of the model, for notation convenience both will be saved by the validation engine in the model-validation-result function Θ as defined in definition 32.
Adopting the **evaluateConstraints** function in algorithm 2 to different types of constraints is a matter of differentiating between the types. When evaluating a data constraint line 23 needs to assign -1 and lines 25 - 32 can be replaced with

$$\mathcal{G}_\Theta \leftarrow \mathcal{G}_\Theta \cup \{(C, i) \mapsto \mathit{left}\}$$

This change does not change the complexity of the validation engine as evaluating the right-hand side of the prediction constraint in combining the results was assumed to take constant time.

4.4 Constraint-based Explanations and Interpretations

In this section, the focus is on using the validation results available as model-validation-result function Θ to get insights into the model behavior (see lemma 3) and explain

predictions made by the model, when possible. Θ was generated by validating the model M_θ and the samples extracted from a knowledge graph G with user-defined prediction and data constraints. Because of the different semantics as explained in section 4.3.4 different usage scenarios arise:

1. Training a model, which conforms to specific requirements
   (a) Understand the influence of bad data (w.r.t. data constraints) used to train the model
   (b) Analyze why the model may have made bad decisions w.r.t. prediction constraints
2. During the model inference explain a model prediction made based on a problem instance w.r.t. prediction constraints

These scenarios are relevant for different kind of users. The machine learning engineer is interested in creating models, which conform to requirements of the client and have as less errors as possible. In that case, the explanations can be used to gain insights into the model behavior (scenario 1), which then might be used to improve the model. For example, through changing the hyperparameters of the model, being able to perform further data cleansing or further feature engineering. The next group of people are the ones offering the model to the world. For them it is important to have a model, which makes predictions conforming to governmental regulations, or scientific facts (e.g., medical or physical) and social principles. As they are not interested in improving the model directly, but instead want to see that the model they are selling is correct (scenarios 1b and 2). Finally, there are the users of the model. As a user, one wants to be able to justify the decision made by the model and one might have certain constraints on the model, which should be respected and can be used to explain the predictions made by the model (scenario 2). In this thesis, a graphical approach is used for explanations, as this is meant to facilitate the understanding of the model by the different kinds of user groups and can be interpreted properly according to the given scenario.
The following sections build a framework for visualizing constraint validation results, while also applying the framework to confusion matrices and to decision trees.

4.4.1 Frequency Distribution Tables to Summarize and as a Basis for Visualizations

Frequencies or counts of specific observations build the foundation of the visualizations used. More specific frequency distributions are used, as they allow to organize the available model validation results in a meaningful way, which connects the knowledge gained through validation with the structure of the model to be explained. Basically, a frequency distribution table is used to show the different measurement categories and the number of observations per category.

> **Definition 41: Frequency Distribution Table [42]**
> A frequency distribution table $F_{C_1,C_2,...,C_n} \in \mathbb{N}^{|C_1|\times|C_2|\times...\times|C_n|}$ is an n-dimensional matrix of natural numbers, where $C_1, C_2, ..., C_n$ are sets of measurement categories.

Therefore, independent of the model, the easiest frequency distribution table would have the validation results as the single group of measurement categories and, hence, can be

visualized by a pie chart. This kind of visualization can be used as a very raw summary of the validation results.

Example 15: A First Visualization for the Example Constraint

First, it should be noted that the example constraint is a prediction constraint and, therefore, it is assumed that the underlying semantic context of the samples in the knowledge graph is correct. The validation results are already given in example 13, from which a frequency distribution table is build, having the validation results as categories for both kinds of semantics (see Figure 4.3 and 4.4). This kind of one dimensional data is easily visualized by a pie chart as shown in Figure 4.5 and 4.6. As a pie chart converts frequencies into fractions, the number n of samples in the dataset are given below the chart. Comparing the two visualizations, the 3-valued one gives more information, as it becomes clear, that there are not many pregnant persons in Germany, which have more than 20 contacts to not vaccinated persons in the dataset. Further, the pie chart shows that the few to which the constraint applies, the model recommends not getting vaccinated, which invalidates the predictions.

Groups	invalid	valid
Complete Dataset	833	9166

Figure 4.3: Frequency distribution table $F_{\{0,1\}}$ using the 2-valued logic for the motivating example constraint

Groups	not applicable	invalid	valid
Complete Dataset	9166	833	0

Figure 4.4: Frequency distribution table $F_{\{-1,0,1\}}$ using the 3-valued logic for the motivating example constraint

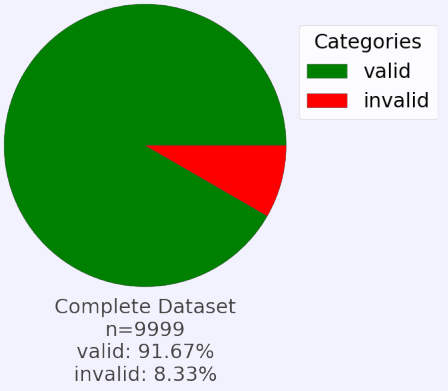


Figure 4.5: Pie Chart of the frequency distribution table in Figure 4.3

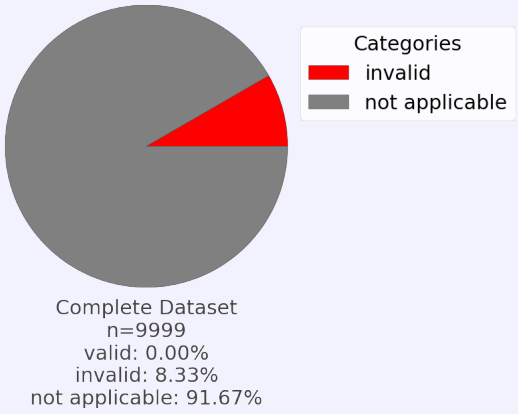


Figure 4.6: Pie Chart of the frequency distribution table in Figure 4.4

In a next step, this kind of frequency distributions are extended by splitting the dataset into different groups. Depending on the machine learning task, one might create a group

for each class (classification) or have a range of target values for each group (regression). In both cases the grouping can be made depending on the prediction of the model or based on the ground truth target value in the dataset. As the data is now two-dimensional, a histogram can be used for visualization.

Example 16: Group by Target for more Meaningful Visualisations

Here the 3-valued logic is used and the frequency distribution from example 15 is extended, such that each ground truth class builds a separate group. As the validation results from 13 are used, the model, which is validated, is the decision tree in Figure 1.3 and the predicted class equals the ground truth class in that case for all samples in the dataset. Besides this fact, the histogram shows that the invalidated predictions were made on the basis of falsely labeled data. The corresponding histogram is shown in Figure 4.8.

Groups	valid	invalid	not applicable
vaccinated	0	0	5000
not vaccinated	0	833	4166

Figure 4.7: Frequency distribution table $F_{\{\text{vaccinated,not vaccinated}\},\{-1,0,1\}}$ for the motivating example constraint using a grouping by the target class

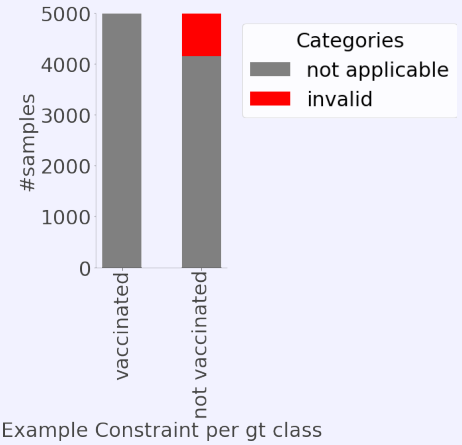


Figure 4.8: Histogram of the frequency distribution table in Figure 4.7

In the last two examples, the complete dataset is used to build the frequency distribution table. But clearly this does not need to be the case. It might be useful to summarize only a fraction of the constraint validation results belonging to specific samples in the dataset. As the table is created dependent on a constraint, which completes the components needed to define the frequency distribution tables used to summarize the validation results.

Definition 42: A Function to Create Frequency Distribution Tables to Summarize Model Validation Results given one Constraint
Let D be a dataset, G a set of arbitrary group identifiers, C a constraint, N the number of samples in D and $\mathbf{\Gamma}$ the endless space of grouping function $\Gamma : G \rightarrow$

$\mathcal{P}([1, ..., N])$, then the function

$$\mathcal{F}_{G,C} : \mathcal{P}([1, ..., N]) \times \mathbf{\Theta} \times \mathbf{\Gamma} \to \mathbb{N}^{|G| \times |\{-1,0,1\}|}$$

maps a subset of the indices D_{idx} of D, a model validation result function Θ and a grouping function Γ to a frequency distribution table:

$$\mathcal{F}_{G,C}(D_{idx}, \Theta, \Gamma) \mapsto F^{C,D_{idx}}_{G,\{-1,0,1\}}$$

where

$$F^{C,D_{idx}}_{G,\{-1,0,1\}} = \left(\left| f^{C,D_{idx}}_{g,v} \right| \right)_{\substack{g \in G \\ v \in \{-1,0,1\}}}$$

$$f^{C,D_{idx}}_{g,v} = \{i | i \in (D_{idx} \cap \Gamma(g)) \wedge \Theta(C, i) = v\}$$

Clearly Θ, Γ and D_{idx} have to refer to the same dataset D.

The definition concludes the creation of frequency distribution tables for one constraint given a dataset (a tuple of samples), the model validation result function and a function used for grouping. The following example shows the application of definition 42 using the previous examples.

Example 17: Creating Frequency Distribution Tables Formally

Example 15 and example 16 both created frequency distribution tables. Both of them use the whole dataset ($D_{full} = [1, ..., N]$) as shown in example 7, the model-validation-result Θ function as shown in example 13 and the constraint C defined in example 8.
In example 15, there is only a single group. Therefore, Γ_{all}, would be the function mapping all indices of instances in D to the same group called *Complete Dataset*. Therefore, to create the frequency distribution table in Figure 4.4 the following formula is evaluated.

$$\mathcal{F}_{\{\text{Complete Dataset}\},C}(D_{full}, \Theta, \Gamma_{\text{all}})$$

In the second example, two grouping functions are used such that instances with the same predicted target class or ground truth target class are grouped together. In the first case, it holds

$$\Gamma_{\text{predicted class}}(g) \mapsto \{i \mid M_\theta(i) = g\}$$

and in the second case

$$\Gamma_{\text{ground truth class}}(g) \mapsto \{i \mid t_i = g\}$$

Therefore, creating the frequency distribution table corresponding to Figure 4.7 is a matter of executing

$$\mathcal{F}_{\{\text{vaccinated,not vaccinated}\},C}(D_{full}, \Theta, \Gamma_{\text{predicted class}})$$

or

$$\mathcal{F}_{\{\text{vaccinated,not vaccinated}\},C}(D_{full}, \Theta, \Gamma_{\text{ground truth class}})$$

In definition 42 the domain of the grouping function $\mathrm{dom}(\Gamma)$ is one dimensional, but clearly multiple grouping functions $\Gamma_1, \Gamma_2, ..., \Gamma_M$ with domains $G_1, G_2, ..., G_M$ can be combined to give a function $\Gamma_{[1,...,M]} : G_1 \times G_2 \times ... \times G_M \to \mathcal{P}([1, ..., N])$ with

$$\Gamma_{[1,2,...,M]}(g_1, g_2, ..., g_M) \mapsto \bigcap_{i \in [1,...,M]} \Gamma_i(g_i) \tag{4.16}$$

where $(g_1, g_2, ..., g_M) \in G_1 \times G_2 \times ... \times G_M$. Setting $G = G_1 \times G_2 \times ... \times G_M$ allows to generalize the definition to multidimensional grouping functions.

4.4.2 Decomposing the Confusion Matrix

In the last section, frequency distribution tables were used to summarize the model validation results of a single constraint and it is shown how multiple grouping functions can be merged together. Here these concepts are used to decompose the confusion matrix w.r.t to the constraint validation results.
The confusion matrix is a tool often used when evaluating a machine learning model M_θ trained on a classification task and is basically a frequency distribution table [20]. Given a classification task, the confusion matrix counts the number of samples $(\mathbf{x}_i, t_i) \in D$ with $M_\theta(\mathbf{x}_i) = t_i$ (true positives), $M_\theta(\mathbf{x}_i) \neq t_i$ (false positives), $M_\theta(\mathbf{x}_i) = t_i$ (true negatives), and $M_\theta(\mathbf{x}_i) \neq t_i$ (false negatives).
Given the decision tree in Figure 1.3 and the dataset given in example 7, the confusion matrix can be inferred to be the one shown on the left in Figure 4.9. On the right side of the figure, the confusion matrix is decomposed, such that the upper matrix M_{valid} counts the instances, which are valid according to the example constraint and the lower matrix only the invalid instances M_{invalid}. The visualization is a result of the frequency distribution table created via

$$\mathcal{F}_{G_{gt} \times G_{pred}, C}(D_{full}, |\Theta|, \Gamma_{[\text{predicted class,ground truth class}]})$$

where $G_{gt} = G_{pred} = \{\text{vaccinated}, \text{not vaccinated}\}$ are the group names. Extending the confusion matrix decomposition to the 3-valued logic is just a matter of adding an additional matrix $M_{\text{non applicable}}$ to the right-hand side, which counts the samples marked as non applicable.
At this point, it is of interest to investigate the meaning of the different counts. Starting with the type of the example constraint (e.g., a prediction constraint). A sample occurring in M_{valid} in the upper left or lower right corner (marked with a yellow border) indicates that the constraint confirms the prediction made by the model and argues why the prediction is correct. Therefore, in the example, it can be explained why the predictions are correct: The persons are either not pregnant, live not in Germany, do not have more than 20 contacts with non-vaccinated persons, or are vaccinated themselves.
If a sample is counted in M_{valid} in the lower left or upper right corner (marked with a blue border), the constraint confirms the prediction, but now also the generalization made by the model, and explains why the prediction is correct. This case does not occur in the example as the model does make predictions deviating from the ground truth (because of the small number of different instances in the dataset).
The samples counted in M_{invalid} are invalidated by the constraint, which disproves the prediction made by the model. High numbers in the upper left and the lower right corner (marked with a green border) can indicate that the model failed to generalize well (e.g.,

overfitting). In the example, this is indeed the case: The model failed to generalize, and does not deviate from the ground truth values.
In contrast, high numbers in the lower left or upper right corner (marked with a purple border) can indicate underfitting, i.e., is a model which has not yet captured the structure of the data, will probably violate the constraint in a way in which predictions deviate from the ground truth values.
In the case of data constraints, these interpretations do not apply. But decomposition may be used to discover patterns in the semantic context of the data in the knowledge graph, which makes the model make valid or invalid predictions. Therefore, these constraints can give rise to improved interpretability of the model.

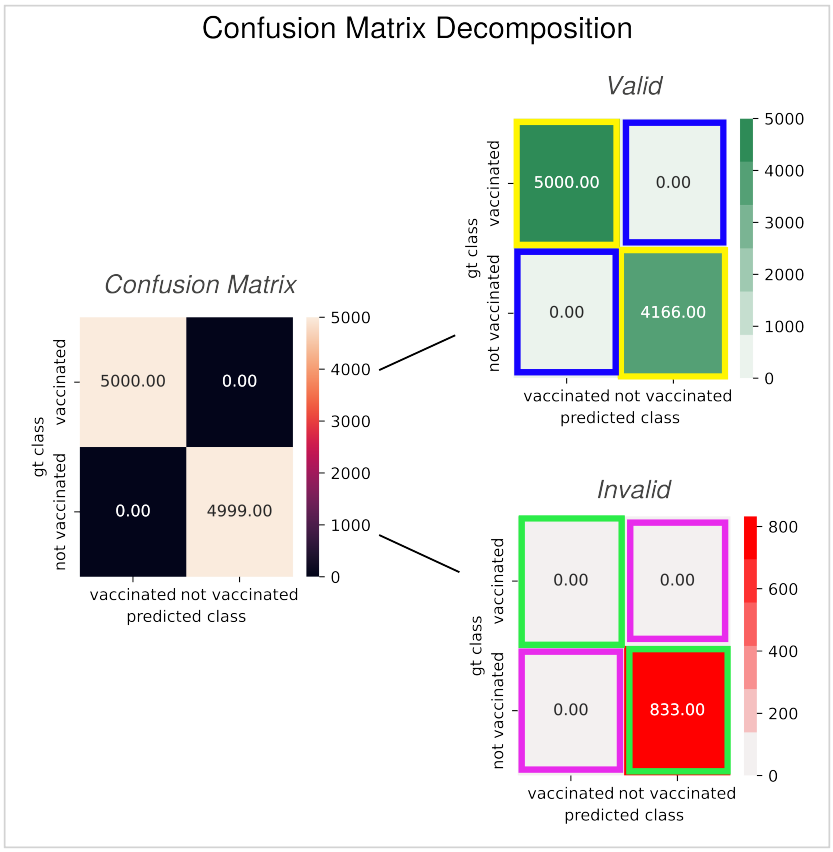


Figure 4.9: Decomposing the Confusion Matrix into Its Valid and Invalid Components

Besides the decomposition of the confusion matrix, this approach could also be used for other kind of frequency distribution tables using a two-dimensional grouping function $\Gamma_{[1,2]} : G_1 \times G_2 \mapsto \mathcal{P}([1, ..., N])$. In this case, the matrix on the left needs to use the entries of the frequency distribution that result when the validation results are summed per group $(g_1, g_2) \in G_1 \times G_2$.

4.4.3 Visualizing the Model Validation Results Given Multiple Constraints

In section 4.4.1, frequency distribution tables were introduced to summarize the model validation results given a single constraint. However, the model-validation-result function

might contain the validation result of multiple constraints. Hence, a way is needed to visualize the results in that case.
A natural extension of the last section is to create a frequency distribution table for each constraint using the same subset of indices D_{idx}, the same set of groups G, and the same set of possible validation results V. Given the constraints c_j ($j \in [1, ..., M]; M \in \mathbb{N}$), the frequency distribution tables $F_{G,V}^{c_j,D_{idx}}$ have the same dimensions and all sum up to $|D_{idx}|$. In most cases, the grouping will be defined without making use of the constraint validation results and, therefore, summing the validation results per group will be the same for all constraints. This motivates summarizing the model validation results in a histogram, having M bars per group and each bar visualizes the distribution of the validation result for the given group.

Example 18: Visualizing the Validation Results of Multiple Constraints Using a Histogram

This example extends example 16 by adding the constraint $C2$ expressing that males should not be vaccinated and the constraint $C3$ that females should be vaccinated in general. The following table gives the model-validation-result function Θ (short version of the table in example 13 showing the validation results but for multiple constraints).

dataset indices i	Person	$\Theta(C, i)$	$\Theta(C2, i)$	$\Theta(C3, i)$
1...3333	`:Max`	-1	1	-1
3334...4166	`:Maria`	0	-1	0
4167...9166	`:Eva`	-1	-1	1
9167...9999	`:Laura`	-1	-1	0

Table 4.7: The model-validation-result function for the motivating example given the constraints C,$C2$ and $C3$

As the frequency distribution tables are needed, $\mathcal{F}$ is executed as in in example 17 but for the constraint $C2$ and $C3$. The results of the executions are shown in the frequency distribution tables shown in Figure 4.10 and 4.11.
Afterwards, all three frequency distribution tables (Figure 4.7, 4.10 and 4.11) are visualized in Figure 4.12 using a histogram with a group of bars for each group (the predicted class) and each bar in the group for one of the constraints. Therefore, the validation results can be compared visually.
It turns out that all persons predicted to be vaccinated are females (valid according to $C3$). Further, approximately $\frac{2}{3}$ of the non-vaccinated persons are males (valid according to $C2$) and the rest of them are females (invalid according to C and $C3$). As the motivating example, the constraint was a prediction constraint; the predictions, confirmed by the model validation results, can now be explained. In the example, there are 5,000 predictions valid according to $C3$ and 3,333 predictions valid according to $C2$. These predictions can now be explained with the conditions defined for $C2$ and $C3$, respectively. These predictions are correct because males

should not be vaccinated, resp., females should be vaccinated.

Groups	valid	invalid	not applicable
vaccinated	0	0	5000
not vaccinated	3333	0	1666

Figure 4.10: Frequency distribution table $F_{\{\text{vaccinated,not vaccinated}\},\{-1,0,1\}}$ for the constraint $C2$ using a grouping by the target class

Groups	valid	invalid	not applicable
vaccinated	5,000	0	0
not vaccinated	0	1,666	3,333

Figure 4.11: Frequency distribution table $F_{\{\text{vaccinated,not vaccinated}\},\{-1,0,1\}}$ for the constraint $C3$ using a grouping by the target class

Figure 4.12: Histogram of the frequency distribution tables in Figure 4.7, 4.10 and 4.11

This kind of visualization is great to compare the validation results per constraint. However, it stays unclear how the validation results of the different constraints are correlated, i.e., in general knowing how the validation results of the different constraints overlap. For example, one cannot say in general whether the instances invalidated by constraint C are included in the ones invalidated by constraint $C3$ or whether there are predictions validated by $C2$ but invalidated by C or $C3$. Another interesting question, in the case of the 3-valued logic, is whether all the predictions made by the model on the basis of the

samples belonging to the subset of indices given are covered by at least one constraint. That is, there is a constraint C for each index $i \in D_{idx}$ such that $\Theta(C, i) \neq -1$. To answer these kinds of questions, another function COV is defined to create frequency distribution tables, which summarize the validation results of multiple constraints in a single 2-dimensional frequency distribution table.

Definition 43: A Function to Create Frequency Distribution Tables to Summarize the Model Validation Results of Multiple Constraints
Let D be a dataset, G a set of arbitrary group identifiers, $\mathcal{C}$ a tuple $(c_1, c_2, ..., c_M)$ of constraints with falling priority, N the number of samples in D and $\mathbf{\Gamma}$ the endless space of grouping function $\Gamma : G \to \mathcal{P}([1, ..., N])$, then the function

$$COV_{G,\mathcal{C}} : \mathcal{P}([1, ..., N]) \times \mathbf{\Theta} \times \mathbf{\Gamma} \to \mathbb{N}^{|G| \times |\mathcal{C}| * |\{-1,0,1\}|}$$

maps a subset of the indices D_{idx} of D, a model validation result function Θ and a grouping function Γ to a frequency distribution table:

$$COV_{G,C}(D_{idx}, \Theta, \Gamma) \mapsto \left(\left| c_{g,v}^{c_i, D_{idx}} \right| \right)_{\substack{g \in G \\ (v,c_i) \in (\{-1,0,1\} \times \mathcal{C})}}$$

where

$$c_{g,0}^{c_i, D_{idx}} = f_{g,0}^{c_i, D_{idx}} \setminus \left(\bigcup_{j<i} f_{g,0}^{c_j, D_{idx}} \right)$$

$$c_{g,1}^{c_i, D_{idx}} = f_{g,1}^{c_i, D_{idx}} \setminus \left(\bigcup_{j<i} f_{g,1}^{c_j, D_{idx}} \cup \bigcup_{i} f_{g,0}^{c_j, D_{idx}} \right)$$

$$c_{g,-1}^{c_i, D_{idx}} = f_{g,-1}^{c_i, D_{idx}} \setminus \left(\bigcup_{j<i} f_{g,-1}^{c_j, D_{idx}} \cup \bigcup_{i} f_{g,0}^{c_j, D_{idx}} \cup \bigcup_{i} f_{g,1}^{c_j, D_{idx}} \right)$$

and $f_{g,v}^{c_i, D_{idx}}$ is defined as in definition 42.

As in definition 42, the sum of the entries is $|D_{idx}|$ per table. However, definition 43 summarizes the validation results of $|\mathcal{C}| * |D_{idx}|$ validation results by only counting the ones with the highest priority. Therefore, a tuple of constraints $\mathcal{C}$ with falling priority is needed and, furthermore, it is assumed that an invalided prediction has a higher importance than a valid prediction, which in turn is more important than a non-applicable one. Given these priorities, it is possible to count exactly one validation result per sample.
When using the 3-valued logic, the resulting frequency distribution table shows the coverage of the validation results over the subset of samples (i.e., the number of validation results are not *not applicable*). Due to this, the summarization method from definition 43 will be referred to by *coverage*.

Example 19: Applying Definition 43 to Example 18

In this example, the constraint validation results from 18 are reused and definition 43 is applied as follows. First, the inputs are determined:

$$\begin{aligned} G =& \{\text{vaccinated}, \text{not vaccinated}\} \\ \mathcal{C} =& \text{tuple}(\{C, C1, C2\}) =: (c_1, c_2, c_3) \\ N = 9999 =& |D| \\ D_{full} =& [1, ..., N] \end{aligned}$$

where D is the usual motivating example dataset and $\Gamma_{\text{ground truth class}}$ the grouping function from example 17. To execute

$$COV_{G,\mathcal{C}}(D_{full}, \Theta, \Gamma_{\text{ground truth class}})$$

first, the sets $f_{g,v}^{c_i, D_{full}}$ need to be estimated for all $g \in G$, $v \in \{-1, 0, 1\}$ and $i \in \{1, 2, 3\}$. This is done in Figure 4.13 using the ground truth values from Figure 1.1.

(v, c_i)	$f_{\text{vaccinated},v}^{c_i, D_{full}}$	$f_{\text{not vaccinated},v}^{c_i, D_{full}}$
$(0, c_1)$	$\emptyset$	$[3334, ..., 4166]$
$(0, c_2)$	$\emptyset$	$\emptyset$
$(0, c_3)$	$\emptyset$	$[3334, ..., 4166] \cup [9167, ..., 9999]$
$(1, c_1)$	$\emptyset$	$\emptyset$
$(1, c_2)$	$\emptyset$	$[1, ..., 3333]$
$(1, c_3)$	$[4167, ..., 9166]$	$\emptyset$
$(-1, c_1)$	$[4167, ..., 9166]$	$[1, ..., 3333] \cup [9167, ..., 9999]$
$(-1, c_2)$	$[4167, ..., 9166]$	$[3334, ..., 4166] \cup [9167, ..., 9999]$
$(-1, c_3)$	$\emptyset$	$[1, ..., 3333]$

Figure 4.13: Table showing the $f_{g,v}^{c_i, D_{full}}$

Building on the sets $f_{g,v}^{c_i, D_{full}}$, the sets $c_{g,v}^{c_i, D_{idx}}$ can be estimated as in Figure 4.14.

(v, c_i)	$c_{\text{vaccinated},v}^{c_i, D_{full}}$	$c_{\text{not vaccinated},v}^{c_i, D_{full}}$
$(0, c_1)$	$\emptyset$	$[3334, ..., 4166]$
$(0, c_2)$	$\emptyset$	$\emptyset$
$(0, c_3)$	$\emptyset$	$[9167, ..., 9999]$
$(1, c_1)$	$\emptyset$	$\emptyset$
$(1, c_2)$	$\emptyset$	$[1, ..., 3333]$
$(1, c_3)$	$[4167, ..., 9166]$	$\emptyset$
$(-1, c_1)$	$\emptyset$	$\emptyset$
$(-1, c_2)$	$\emptyset$	$\emptyset$
$(-1, c_3)$	$\emptyset$	$\emptyset$

Figure 4.14: Table showing the $c_{g,v}^{c_i, D_{full}}$

Finally, taking the cardinality of the sets give the frequency distribution table.

Groups	$(c_1, 0)$	$(c_2, 0)$	$(c_2, 1)$	$(c_3, 1)$
vaccinated	0	0	5000	0
not vaccinated	833	833	0	3333

Figure 4.15: Frequency distribution table showing the coverage results for the constraints c_1, c_2, c_3 using a grouping by the ground truth class

The frequency distribution table can now be visualized as usual. This is done in Figure 4.16 using the grouping by the ground truth class on the left side and without a grouping on the right side. In comparison to Figure 4.12, it becomes clear that all the predictions made by the model are covered by at least one constraint. In the case of predictions not covered by any constraint, these will always fall in the $(-1, c_1)$ category, which will be called *not covered* in future visualizations.
Further, the plot allows making statements about the overlapping of validation results. A prediction marked as valid is not invalidated by any other constraint. A lesser important constraint validation result can only be overlapped by a more important one, given that the validation result is the same. Therefore, in the example approximately 83% of the samples in the dataset are predicted according to the constraints and approximately 8% is invalidated by the example constraint and another 8% is invalidated by the second constraint, but not by the example constraint.

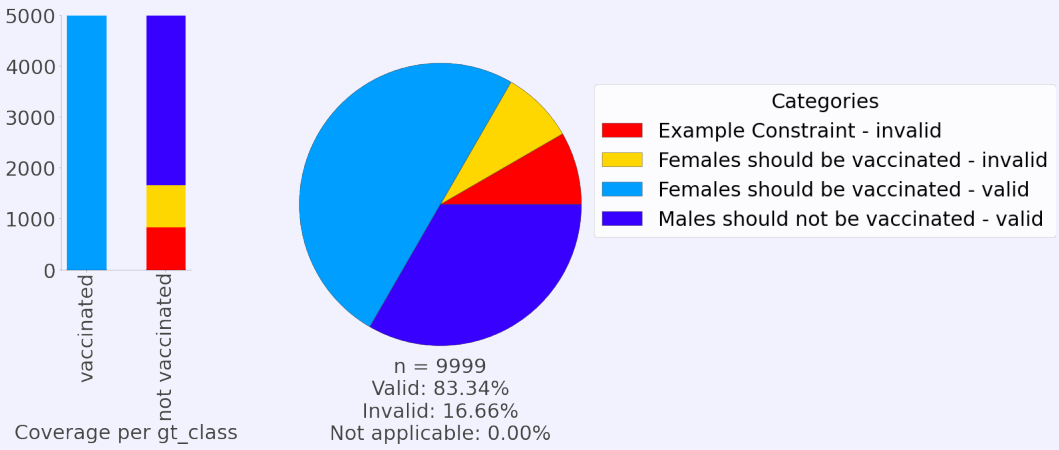


Figure 4.16: Validation Results of multiple constraints summarized by coverage. Left side shows the results grouped by the ground truth class and the right side shows the results without a grouping

4.4.4 Supporting the Explainability of Decision Trees

In section 2.3.3, it is explained how the learning algorithm of a decision tree works. Therefore, Figure 2.7 was used to visually interpret the predictions made by the model based on the dataset used to train the model and the internal structure of the model. At this point, it is a matter of keeping the gist of the visualization and additionally

make use of the constraint validation results to create a visualization, which allows to further interpret the decision tree with respect to the user-defined constraints and explains predictions of the decision tree in suitable cases as demonstrated in section 4.4.2 and example 18.

Frequency Distribution Tables to Summarize the Constraint Validation Results w.r.t. the Leaves of the Decision Tree The frequency distribution tables introduced before lay the foundation to visualize the validation results. But they do not make use of the internal structure of the decision tree. To reflect the internal structure of the decision tree, each frequency distribution table will correspond to a node $n_{d,u}$ and the set of samples used to create the frequency distribution table is limited to $R_{d,u}$. As defined in section 2.3.3, $R_{d,u}$ is the set of samples, which were used to decide, whether the node $n_{d,u}$ is going to be a split or a leaf node. Therefore, the resulting node visualization in Figure 2.7 (in the case of a regression task) and Figure 1.3 (in the case of a classification task) made use of this limitation to show the effect these samples had on the decisions made for that node (e.g., the node type and parameters). This is what the visualization of a frequency distribution table using the constraint validation results of the samples $R_{d,u}$ should also achieve: *Show the node relevant constraint validation results, to allow for interpretations of patterns in the data and predictions of the decision tree, leading to the internal structure of the decision tree.*
Formally, a grouping function is defined, which captures the mapping of indices of samples in the dataset to nodes in the decision tree according to the $R_{d,u}$ sets:

$$\Gamma_{\text{nodes}}(n_{d,u}) = \{i \mid (x_i, t_i) \in R_{d,u}\} \tag{4.17}$$

Using the new defined grouping function, the frequency distribution table to be used for the visualization of the leaf nodes, to summarize the constraint validation results, can be defined:

$$\mathcal{F}_{G,C}(\Gamma_{\text{nodes}}(n_{d,u}), \Theta, \Gamma_{\text{all}}) \tag{4.18}$$

$$COV_{G,\mathcal{C}}(\Gamma_{\text{nodes}}(n_{d,u}), \Theta, \Gamma_{\text{all}}) \tag{4.19}$$

where Γ_{all} corresponds to the function, which maps all samples to a single group. As discussed in section 4.4.3 for the visualization of the constraint validation results in case of multiple constraints, formula 4.18 has to be executed once per constraint or use the frequency distribution table showing the „coverage“ of the constraint validation results by prioritizing them.
The procedure leads to a pie chart as shown in Figure 4.6 (in the case of a single constraint) or Figure 4.16 (in the case of multiple constraints and using coverage), or to a histogram as shown in Figure 4.12. The latter one additionally uses $\Gamma_{\text{predicted class}}$, which does not makes sense for a leaf node, as the prediction will be constant for all samples visualized. However, $\Gamma_{\text{ground truth class}}$ can be used in case of classification. This way the plots in the leafs will show the distribution of the ground truth class as the leaves of Figure 1.3 did. This modification additionally allows for interpretations analogue to section 4.4.2 (see Figure 4.17).

Frequency Distribution Tables to Summarize the Constraint Validation Results w.r.t. the Split Nodes of the Decision Tree In a next step, frequency distribution tables should be created for the split-nodes, by extending the work done for

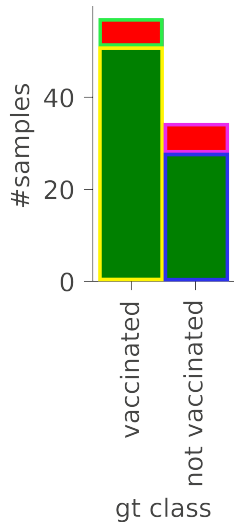


Figure 4.17: Decision Tree Node Visualization of the frequency distribution table using $\Gamma_{\text{ground truth class}}$. The border colors are chosen according to the interpretation from section 4.4.2. The visualized leaf predicts the persons to be *vaccinated*.

the leaf nodes. As discussed in section 2.3.3, figures 1.3 and 2.7 use the split nodes, to show the marginal effect of the split feature on the ground truth. With respect to the constraint validation results, the goal is to *show the marginal effect the split feature has on the constraint validation results, to be able to identify patterns in the data and predictions made by the decision tree.* To achieve the goal, a grouping function $\Gamma^{f_k}_{\text{bins}}$ is defined, which groups the samples to be visualized, according to their split feature value (e.g., the value of f_k):

$$\Gamma^{f_k}_{\text{bins}}(b) = \{i \mid \exists(\mathbf{x}_i, t_i) \in D \wedge (b-1) * s \leq (\mathbf{x}_{i,k} - \min) < b * s\} \tag{4.20}$$

where

$$\begin{aligned} b &\in [1, ..., B], \\ \max &= max(\{\mathbf{x}_{i,k} \mid \exists t_i \ (\mathbf{x}_i, t_i) \in D\}), \\ \min &= min(\{\mathbf{x}_{i,k} \mid \exists t_i \ (\mathbf{x}_i, t_i) \in D\}), \\ s &= \frac{\max - \min}{B} \end{aligned}$$

and B is the number of groups.
Finally, the formulas needed to create the frequency distribution table per split node $n_{d,u}$ can be given,

$$\mathcal{F}_{G,C}(\Gamma_{\text{nodes}}(n_{d,u}), \Theta, \Gamma^{f_k}_{\text{bins}}) \tag{4.21}$$

$$COV_{G,\mathcal{C}}(\Gamma_{\text{nodes}}(n_{d,u}), \Theta, \Gamma^{f_k}_{\text{bins}}) \tag{4.22}$$

where formula 4.21 is used in case of a single constraint C and formula 4.22 in case of a set of constraints $\mathcal{C}$.
Figures 1.3 and 2.7 both visualize the cuboid borders added by the corresponding split nodes to explain the set of samples used in the child nodes. The procedure leads to split nodes as used in Figure 1.4 (in the case of a single constraint) and 4.18 (in the case

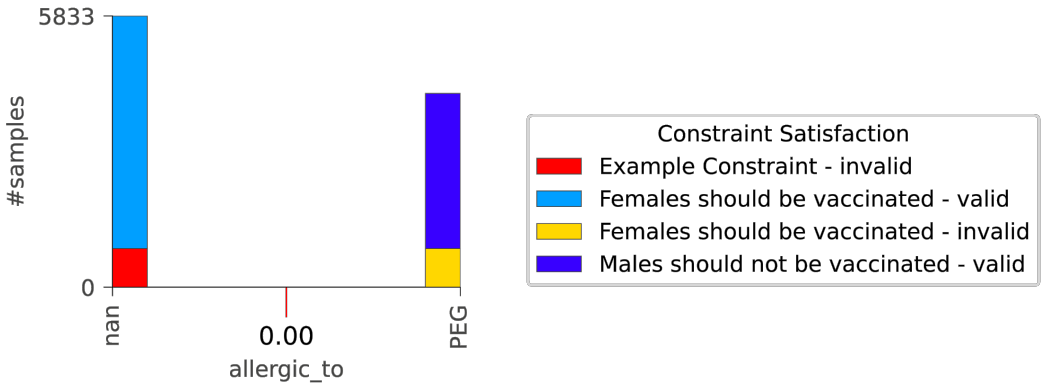


Figure 4.18: Summarizing validation results in bins given the allergic_to split feature and marking the cuboid border added by the split node. The numerical values on the x-axis have been replaced with their corresponding categorical label, the cuboid border, however, can only be given numerical.

of multiple constraint using coverage). The latter should emphasize the significance of summarizing the constraint validation results using coverage as the given procedure does not allow visualizing the constraint validation results of multiple constraints in a useful way, when showing the marginal effect of the split feature.

Composing the Visualizations Finally, all the visualizations of the different nodes can be composed analogue to figures 1.3 and 2.7. This now allows tracking the flow of the constraint validation results through the decision tree. To highlight the number of samples summarized in each visualization, the size of the visualizations per node are chosen proportional to the number of samples summarized.

Example 20: Visualizing the Validation Results Θ given a Decision Tree M_θ

Using the validation results of the example constraint given in Figure 4.7, the application of the formulas 4.21 and 4.18 can be seen in Figure 1.4. Borrowing the node enumeration from Figure 1.3, node 0 and node 1 are split nodes. Hence, formula 4.21 was applied, which allows to annotate the corresponding visualization with the cuboid borders associated with the „allergic_to“ and the „pregnant“ feature. As a consequence of setting D_{idx} to $\Gamma_{\text{nodes}}(n_{d,u})$ for all nodes $n_{d,u}$ and marking the cuboid borders, the flow of the validated and invalidated samples can be tracked from the root node into the leaves. For example, the invalidated samples can be tracked to occur only in the case of persons pregnant but not allergic to PEG. The leave nodes are based on the frequency distribution tables created with formula 4.18 and, therefore, do not show the distribution of the ground truth class.
Using $\Gamma_{\text{ground truth class}}$ instead and making use of formulas 4.22 and 4.19, enables to visualize multiple constraints and additionally shows the distribution of the ground truth class in the leaves. The result of this procedure is depicted in Figure 4.19. The interpretations given in section 4.4.2 allow making statements about the leaves of the decision tree:

1. Not allergic persons which are not pregnant are classified correctly according to the constraint and to the ground truth data.

2. Not allergic persons which are pregnant are classified correctly to the ground truth data. However, the classification violates the example constraint, which might imply overfitting, as will be seen later.

3. Men allergic to PEG are classified correctly according to the constraints and to the ground truth data.

4. Women allergic to PEG are classified incorrectly according to a constraint stating that females should be vaccinated. The decision was made on the basis of ground truth data.

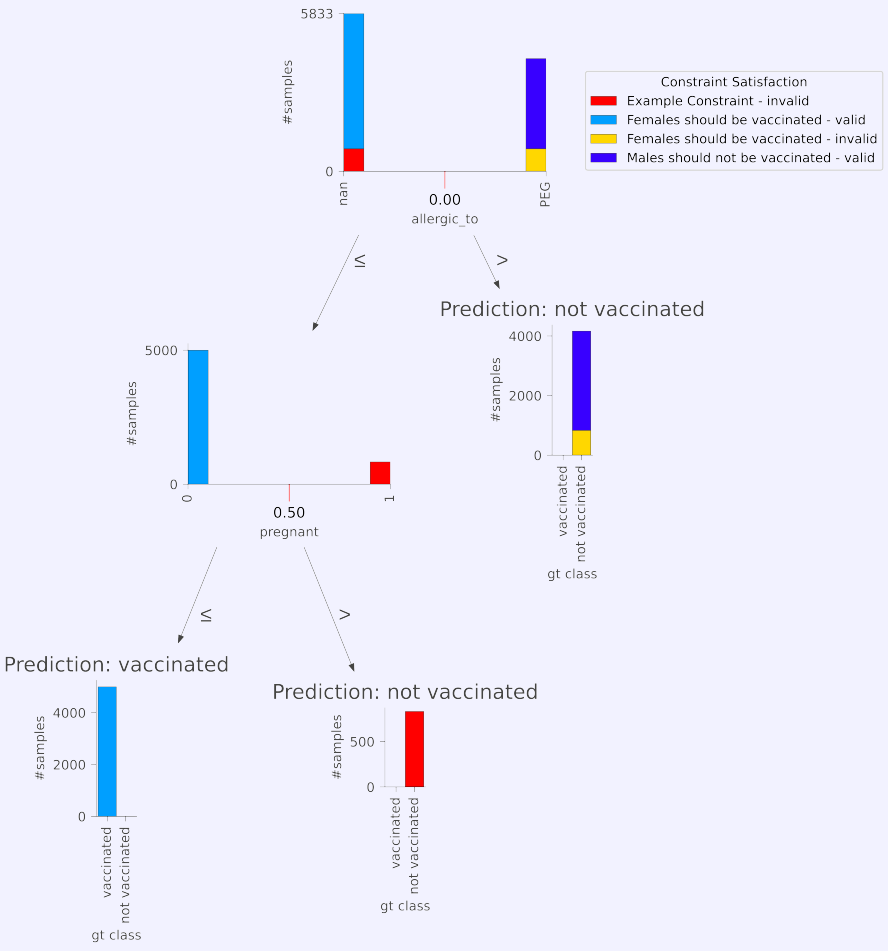


Figure 4.19: Decision Tree of Figure 1.3 annotated with the validation results from Figure 4.7 and showing the ground truth class distributions in the leaves.

Additionally, Visualizing the Ground Truth Values of the Samples in Case of a Decision Tree Used for Regression In the case of a regression task, the ground truth values will be continuous, which allows to explicitly show the marginal effect the

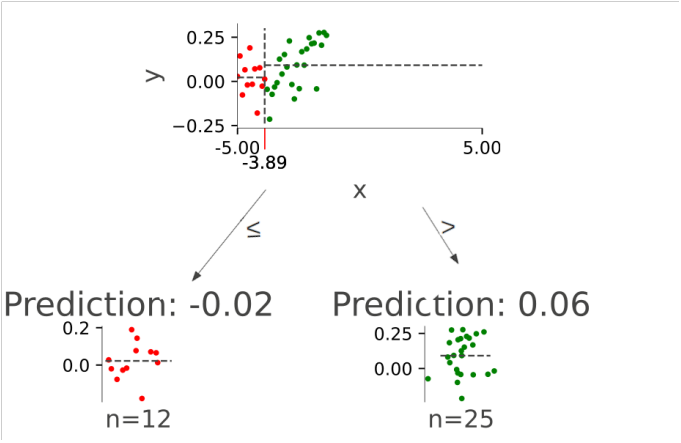


Figure 4.20: Showing the ground truth target values as in 2.7 while visualizing the constraint validation results. Red dots resp. green dots stand for invalidated resp. validated samples / predictions made on the basis of the samples

split features have on the precise ground truth value and the constraint validation result of a constraint. Therefore, the split node visualizations from Figure 2.7 can be reused and extended by choosing the colors of the samples in the scatter plot according to their constraint validation result (see Figure 4.20).
However, this procedure only works for a single constraint and can be quite messy for large datasets. This is why the procedure described before can also be applied to the regression case; except for the grouping by the ground truth value.

Evaluating the Constraints per Decision Tree Node Until this point, the model-validation-result function Θ was created by evaluating each constraint once for each sample $\mathbf{x}_i$, given the prediction $M_\theta(\mathbf{x}_i)$ of the decision tree. However, in the case of prediction constraints, it might be interesting to see the distribution of the validation results per node. For example, in section 2.3.2, these distributions might have helped to choose the maximal depth of the decision tree, as over- and underfitting can be detected per node $n_{d,u}$ (i.e., using the interpretations from section 4.4.2).
As each node in the decision tree is associated with a constant model $M_{c_{d,u}}$ (see section 2.3.3), approaching the problem in lemma 3 corresponds to solving multiple instances of the problem in lemma 2. Each instance is associated with a node $n_{d,u}$ in the decision tree. Hence, the formulas 4.21 and 4.22 need to use the model-validation-result function $\Theta_{M_{c_{d,u}},D,G,\eta}(C,i)$. This is in contrast to the formulas 4.18 and 4.19, which can stay the same as the predictions made in the leaves of the decision tree correspond to the overall prediction the decision tree M_θ would make. Luckily, the repeated evaluation of the same constraints over different models does not require the repeated evaluation of the SHACL shape schemas corresponding to the constraints, because the entity validation function *validate* stays unchanged. Further, it turns out that it is enough to execute line 5 (algorithm 2) once for every occurring constant model prediction. This is for a group of constant models $M_{c_1}, M_{c_2} \ldots$ with $c_1 = c_2 = \ldots = c$ only $\Theta_{M_c,D,G,\eta}$ is needed. In the case that these groups include all the constant models of the leave nodes, the evaluation of $\Theta_{M_\theta,D,G,\eta}$ can be skipped. This limits the maximal amount of model-validation-result functions to the number of distinct predictions the model makes on the basis of the dataset used for the evaluation.

Example 21: Performing Node-based Constraint Validation

The motivating example is a binary classification problem. Therefore, only a maximum of two different model-validation-result functions should be needed. In the case of the motivating example decision tree, there are two different options. First, one could use the model-validation-result function for the two possible classification results. The first group of constant models corresponding to the prediction *vaccinated* would be, $\{M_{c_{1,1}}, M_{c_{2,1}}, M_{c_{3,1}}\}$ and the second group corresponding to *not vaccinated* would be $\{M_{c_{2,2}}, M_{c_{3,2}}\}$. Clearly, the two groups cover each leaf and split node of the decision tree. The alternative used here is $\Theta_{M_\theta,D,G,\eta}$ and $\Theta_{M_{\text{'vaccinated'}},D,G,\eta}$. The first function is used for all the leaf nodes and is already given in Figure 4.7. The second one is given below in Figure 4.21 and is used for the split nodes.

dataset indices i	Person	$\Theta(C,i)$	$\Theta(C2,i)$	$\Theta(C3,i)$
1...3333	`:Max`	-1	0	-1
3334...4166	`:Maria`	1	-1	1
4167...9166	`:Eva`	-1	-1	1
9167...9999	`:Laura`	-1	-1	1

Figure 4.21: Table showing the model validation result function $\Theta_{M_{\prime vaccinated\prime},D,G,\eta}$ for the motivating example given the constraints C,$C2$ and $C3$

Using both model-validation-result functions (i.e., $\Theta_{M_\theta,D,G,\eta}$ and $\Theta_{M_{\text{'vaccinated'}},D,G,\eta}$) the decision tree can be annotated with the constraint validation results per node (see Figure 4.22)
The visualization is done based on the formulas 4.22 (with Θ replaced with $\Theta_{M_{\prime vaccinated\prime},D,G,\eta}$) and 4.19. Comparing the validation results of the newly created plot with Figure 4.19 one can validate that only the validation results of the split nodes have changed. Further, the constant predictions (corresponding to the constant models used during the validation process), which would be made at a split nodes, if they were leaf nodes, are given above the plots.
In the case of the motivating example, it seems that the model could be improved by limiting the decision tree to a maximal depth of, 2 as this is also according to the overfitting noted in example 20. Doing so would transform the node $n_{2,1}$ into a leave node and, therefore, all persons not allergic to „PEG“ would get the recommendation to get vaccinated. This is in the sense of the example constraint,

as it is evaluated to be valid for all samples in $R_{2,1}$ given.

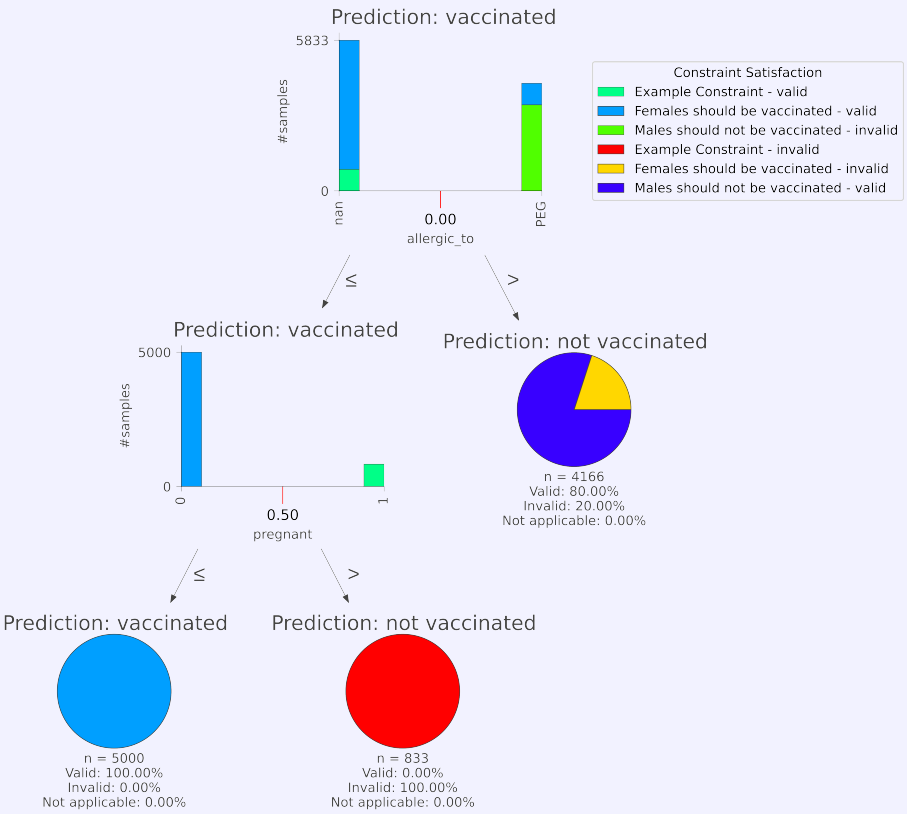


Figure 4.22: Decision Tree of Figure 1.3 annotated with the validation results calculated per node

4.5 Summary

The chapter tackled two problems: Validating constraints over machine learning models and using constraint validation results to interpret and, when possible, explain machine learning model predictions.
Therefore, a model-agnostic validation engine was proposed, which can validate prediction and data constraints. The evaluation of both types of constraints uses the SHACL constraint validation as a central component, which allows using the semantic context of the entities in the knowledge graph when defining the constraints. The sample-to-node mapping allows for an association of the seed nodes (i.e., the entities of interest) and their SHACL constraint validation results with the samples in the dataset. The mapping is retrieved simultaneously with the dataset from a knowledge graph using a dataset generating query. The dataset generating query can be used to create the seed query, which only retrieves the seed nodes. Further, both types of constraints are defined given a target shape, which is evaluated over the seed nodes associated with the samples in the dataset. The data constraint does not require any further information for evaluation, but the prediction constraints additionally consider the model's predictions. This is also why data constraints can only help interpret, while prediction constraints can also be used to

explain model predictions in some cases. The evaluation is done using a 3-valued logic, and unlike the 2-valued one, it allows the identification of samples that are only valid because the condition of the constraint does not apply.
Several heuristics are proposed to speed up the execution of the validation engine: Three heuristics are proposed to accelerate the SHACL constraint validation by minimizing the number of SHACL constraint validation results that need to be produced. The assignment of SHACL constraint validation results to samples in the dataset is analyzed in terms of the join strategy with physical join operators, resulting in two heuristics to speed up the assignment. Finally, the theory behind a SPARQL query engine is presented, which pushes down the SHACL constraint validation and the joining of the SHACL constraint validation results with the samples in the dataset into the SPARQL query execution.
Although the heuristics may help optimize the validation engine's execution time, time-complexity-wise, the theoretical approach is PSPACE-complete as querying the dataset is PSPACE-complete, and the SHACL constraint validation is NP-complete in general. However, reasonable decisions regarding the dataset generating query and the constraints make the problem solvable in polynomial time.
The results of the validation engine are stored in a model-validation-result function. Frequency distribution tables are consulted for the summarization of a single constraint. The constraint validation results of multiple constraints are further cumulated by a concept called coverage. Coverage reduces the constraint validation results to the number of samples in the dataset through prioritizing constraints and validation results (i.e., invalid over valid over not applicable results). As an application, confusion matrices and decision trees are annotated with the constraint validation results, and general interpretations based on prediction constraints are proposed. The decision tree visualization is inspired by the dtreeviz visualization, but extended to visualize the constraint validation results. Classification, as well as regression tasks, are supported. As the decision tree is composed of constant models, constraints can also be evaluated per node. Both data and prediction constraints can be used to detect patterns learned by the model.

Chapter 5

Implementation

Chapter 4 explained the approach by showing its logical components. These components were put together in pseudocode to allow for a theoretical understanding. The steps were supplemented with the practical application in case of the motivating example. This chapter enriches the theoretical components with a practical implementation written in Python [64]. The implementation is publicly available on GitHub[1].
Python is chosen for its simple syntax, readability, and extensive support for machine learning, as the most popular machine learning frameworks, are written in Python [37]. However, Python comes with the overhead of interpreting and manipulating Python objects, which is why performance-optimized libraries for Python as NumPy [31], pandas [55], and scikit-learn [10, 50] are based on pre-compiled functions written in the C language. Similar to the examples, the implementation will also be centered on decision trees, specifically the decision trees generated by the scikit-learn library.

5.1 Design, Structure and Dependencies

This section is meant to give an overview of the implementation. First, the most essential dependencies should be mentioned.
A combination of NumPy and pandas is used. These libraries provide the capabilities for vectorized code execution [31] and allow performing high-level data analysis in Python [55]. For the visualizations, matplotlib [33] is used and, in the case of tree-like visualizations, the logic provided by dtreeviz [63] is reused. The dtreeviz library also provides an interface to access trained decision trees of various libraries uniformly. Similar to dtreeviz, Graphviz [26] is used to compose graph-structured visualizations of the nodes visualized before. SPARQL queries are parsed with RDFlib [9] and send to a SPARQL endpoint via the SPARQLWrapper package [32].
Next, to give a high-level overview over the different modules, Figure 5.1 is explained.
On the left-hand side of Figure 5.1, backed in grey, the modules used to implement the constraint validation engine are depicted. These modules are capable of performing the steps described in algorithm 2.
The Dataset is a central module as it is responsible for storing and managing all the information collected from the endpoint and the SHACL validations, i.e., the dataset itself, the sample-to-node mapping and the SHACL validation results. Given a set of constraints, the Dataset initiates the process of the SHACL constraint validation.

[1] https://github.com/JulianLoewe/Validating_Models

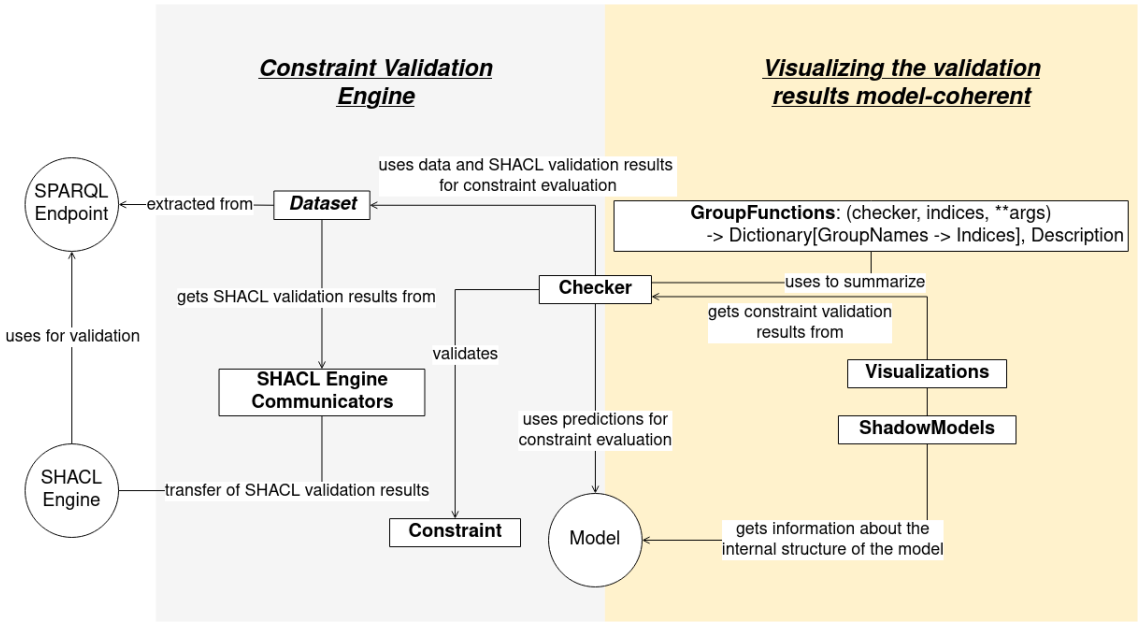


Figure 5.1: Overall Structure of the Implementation

Next is the Checker module, which connects the information stored in the Dataset with the model predictions by initiating the evaluation of the constraints. Furthermore, it stores the constraint validation results afterward.
The SHACL Engine Communicators and the Constraint module remain. The first one receives SHACL validation requests from the Dataset, forwards them to the SHACL engine of the user's choice, and while receiving the results transforms them to a representation independent of the chosen SHACL engine. The latter one implements the logic to evaluate a constraint given the necessary information from the Checker module.
On the right-hand side of the figure, backed in yellow, the components needed to summarize and visualize the constraint validation results are shown. The Checker and the model partially belong to this part. The model is needed to make predictions for evaluating the constraints, but also the internal structure of the model, represented model-implementation-agnostically by the ShadowModels module, is used to create model-coherent visualizations of the constraint validation results.
In addition to the previously mentioned functions of the Checker module, it is also able to summarize the validation results in frequency distribution tables, make use of coverage and, therefore, utilizes grouping functions. Applying these concepts to constraint validation results were introduced in definitions 42 and 43. The summarized or raw validation results and the information about the internal structure of the model are then combined to create the visualization.
The modules will be investigated further with respect to "Performance Efficiency" and "Portability and Maintainability". Both are software quality goals defined in [35].

5.2 Performance Efficiency

This section is meant to describe the performance-relevant implementation details. First, the Dataset module is described as the join described in section 4.3.2 needs to be performed efficiently using the heuristics proposed. Next, the building process of the grouping function Γ_{nodes} corresponding to the set $R_{d,u}$ from section 2.3.3 is found, while also de-

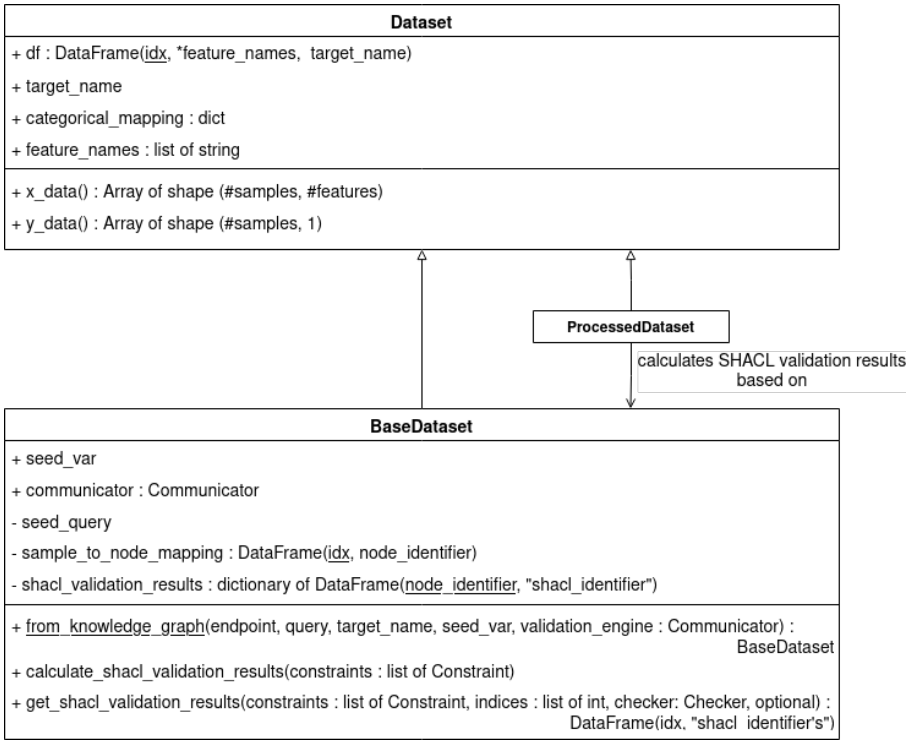


Figure 5.2: The DATASET Module

scribing the structure of the SHADOWMODELS module. Subsequently, the role of caching and lazy calculation of results is outlined. Finally, approaches to parallel computing using the visualization algorithm are touched upon.

5.2.1 The Dataset Module

As shown in Figure 5.2, the module contains three classes: the *Dataset*, the *BaseDataset*, and the *ProcessedDataset* class. All of them make extensive use of the pandas data frame, which allows storing two-dimensional tabular data with the capability to index the rows by a column. In this work, the index column in a data frame will be unique and, therefore, the data frame index implies a functional dependency, analog to a key in a relation of a relational database.

The *Dataset* class is a generic class holding the whole dataset in a pandas data frame called "df" with an index column idx, a column for each feature, and a column for the target. In analogy to a relation in a relational database, the type of the data frame will be denoted by *DataFrame*(idx, *feature_names, target_name)[2]. Therefore, the feature names and the target name need to be stored to identify the problem instances (the "x_data") and the target (the "y_data"). A dictionary called "categorical_mapping" is used to give the user the option to map each feature to a dictionary, which maps encoded feature values to meaningful terms. For example, in a classification task with a *Dataset* containing encoded target classes "categorical_mapping[target_name]" is a dictionary mapping the encoded target classes to their decoded class names.

The *BaseDataset* extends the *Dataset*, with the notion of the samples-to-node mapping, which is realized by a *DataFrame*(idx, node_identifier). Each entry in the data frame

[2]* here denotes the unpacking of the followed list

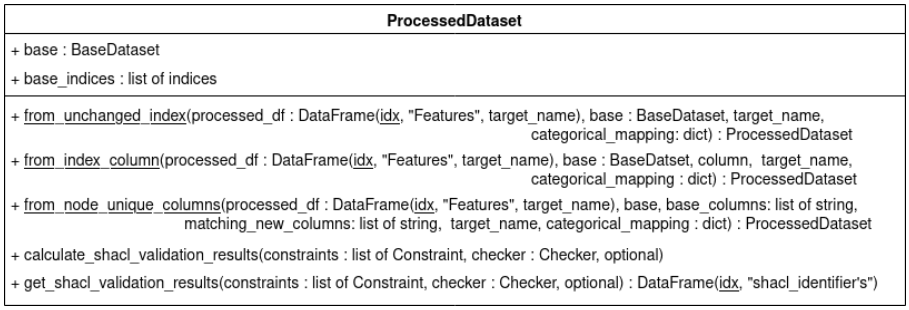


Figure 5.3: The *ProcessedDataset* Class

maps an index i of a sample in the dataset to the node $\eta(i)$. At this point, it should be mentioned that *DataFrame*(idx, *feature_names, target_name) and *DataFrame*(idx, node_identifier) can be concatenated to give the T relation mentioned in section 4.3.2. The concatenation does not require a join as both are only stored separate for implementation convenience.

The *BaseDataset* is assumed to be retrieved from a knowledge graph with a user defined query from a given endpoint using the underlined static method of the same name. In contrast to definition 27, the user is allowed to specify the seed variable and the target variable (target_name) to be different from ?x and ?t. A SHACL validation engine communicator is needed, which provides a method to perform the SHACL validation given a set of constraints. The *BaseDataset* starts the SHACL validation process and stores its results in a dictionary "shacl_validation_results" mapping (shape schema, target shape) tuples (here called "shacl_identifier") to *DataFrame*(node_identifier, shacl_identifier). Each entry assigns a SHACL validation result to a node in the knowledge graph. The construct thereby represents the *validate* function in algorithms 2 and 3. The SHACL validation is triggered by calling "calculate_shacl_validation_results". Each data frame in "shacl_validation_results" corresponds to a relation $S_{j,k}$ in section 4.3.2.

However, during the constraint evaluation, the SHACL validation results are needed per sample. Therefore, the method "get_shacl_validation_results" produces SHACL validation results joined as described in section 4.3.2. Implementation-wise, the procedure results in a *DataFrame*(idx,"shacl_identifier's") mapping each index of a sample to SHACL validation results. The implementation makes use of the join method provided by the pandas library and attempts to make use of the join heuristics from section 4.3.2. First, the heuristic from lemma 8 is implemented in a greedy manner: The SHACL validation results per constraint are grouped by their target definition and ordered according to their average size. Joining in the order estimated should keep the number of different entities with SHACL validation results in the intermediate results small, such that the size of intermediate results increases gradually. Secondly, the heuristic from lemma 9 is implemented by joining the SHACL validation results first by default, as there is no need for intermediate SHACL validation results in the current implementation.

The last class is the *ProcessedDataset* (as depicted in Figure 5.3), which allows the user of a *BaseDataset* to modify the data frame "df" (the original dataset) as needed. In contrast to the pseudocode provided in algorithm 2, one cannot expect that the original dataset extracted from the knowledge graph is the final dataset, which can be used to train, validate or test the model (see example 14). However, the sample-to-node mapping that was originally extracted when the *BaseDataset* was created must remain intact when the original dataset is modified.

This is effectively tackled by a mapping called "base_indices" from the new indices i to

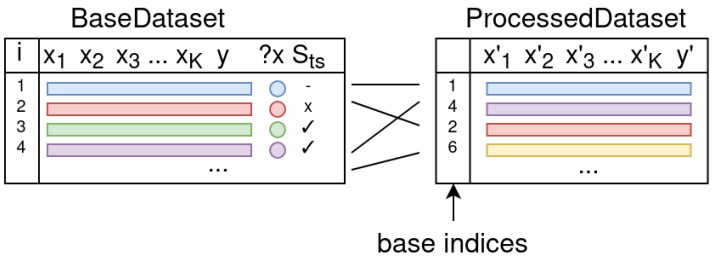


Figure 5.4: **Reconstructing the Sample-to-node Mapping of a Modified Dataset:** The left-hand side shows the *BaseDataset* extracted from the knowledge graph with the seed nodes ?x and the SHACL validation results S_{ts}. The right-hand side shows a processed version of the dataset called *ProcessedDataset* with reconstructed base indices such that the results associated with the original *Dataset* can be used

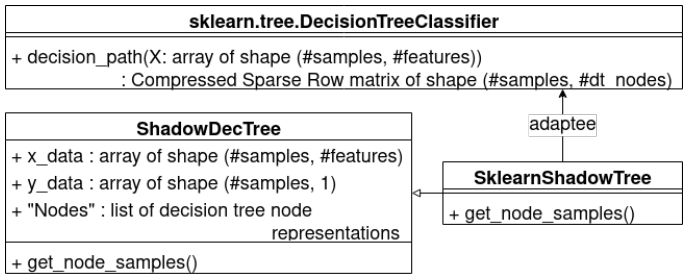


Figure 5.5: Usage of the Adapter Pattern in the ShadowModels Module (see [63])

the old ones j in the *BaseDataset* (see Figure 5.4). The mapping is represented as a list in which the i-th element contains the corresponding index j. The static methods are provided for the user to create the *ProcessedDataset* depending on the changes done to the original dataset, represented as *BaseDataset*. In the best case, the index column of the original dataset was kept or copied to a new column before the modifications were done. As modifications like dropping or duplicating samples will now affect the original index column, the mapping can be read from the index or copied column respectively. If that is not the case, the final dataset has to be joined with the original dataset on a column with values unique to the original index values. In both cases, the result is a *ProcessedDataset*, representing the final dataset, in which every sample can be mapped to the original seed node and the SHACL validation results associated with them.

5.2.2 Estimating the Decision-tree-node-to-samples Mapping

After the validation engine is done and the constraint validation results are available, the validation results are summarized with respect to the model provided. In the case of decision trees, a decision-tree-node-to-samples mapping is needed. That is, the assignment made in section 2.3.3 by the set $R_{d,u}$ for each node $n_{d,u}$ in the decision tree or the grouping function Γ_{nodes} from section 4.4.4. This is very relevant to performance, since the number of nodes in the decision tree increases exponentially with the depth of the decision tree. Implementation-wise, each node in the decision tree has a unique identifier and the goal is to retrieve a dictionary mapping each identifier to a list containing the indices in the dataset associated with the samples handled by that node.
To further understand the implementation, Figure 5.5 shows the structure of the SHAD-

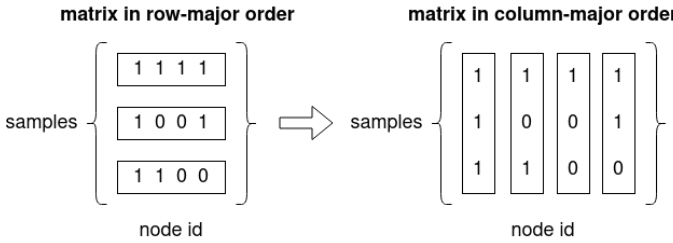


Figure 5.6: Converting a Row-major Matrix into a Column-major Matrix

owModels module. In general, the module is responsible for providing insights into the trained model needed for the visualizations. As this should be independent of the library used to train the model, the adapter pattern is used. The *ShadowDecTree* class provides the interface for the library-agnostic adapter. Multiple adapters are already implemented in the dtreeviz library [63]. As an example, Figure 5.5 shows the adapter class *SklearnShadowTree* which implements the „get_node_samples“ method using the „decision_path“ method from a given an instance of a trained *DecisionTreeClassifier* of the scikit-learn library.
The "decision_path" method gives a matrix $M \in \{0,1\}^{\#\text{samples} \times \#\text{decision\_tree\_nodes}}$ with a nonzero entry at position (i,j) indicating that the sample i is handled by the node j. As it is typical for Python, the matrix is saved in row-major order. That means, the rows of the matrix are physically stored next to each other. Therefore, it is recommended for the inner loop to address the column j when iterating over the entries of the matrix in a nested loop. The implementation of the "get_node_samples" method provided by the dtreeviz library does exactly that while adding i to the list referred to by the decision-tree-node-to-samples mapping with entry j if $M_{i,j}$ it is nonzero. Therefore, accessing the Python dictionary structure and adding an item to Python list $\#\text{samples} * \#\text{decision\_tree\_nodes}$ times.
However, as vectorized code execution using NumPy turns out to be faster in many cases [31], the following heuristic is proposed:

Lemma 10: Efficiently Estimating the Decision-tree-node-to-samples mapping
The decision-tree-node-to-samples mapping should be estimated using vectorized code execution.

Implementation-wise, M is converted into a column-major form (see Figure 5.6) and the nonzero row indices of each column j are written into the decision-tree-node-to-samples mapping with entry j. This results in accessing the Python dictionary only $\#samples$ times and avoids the usage of the Python lists.

5.2.3 Caching and Calculating the Needed Intermediate Results

During the constraint validation process and the following visualization process, specific intermediate results are needed multiple times. Therefore, the goal is to calculate them only once and cache them if needed.

SHACL Validation Results When validating multiple constraints that refer to the same SHACL shape schema, sequential validation would lead to the duplicate evaluation of

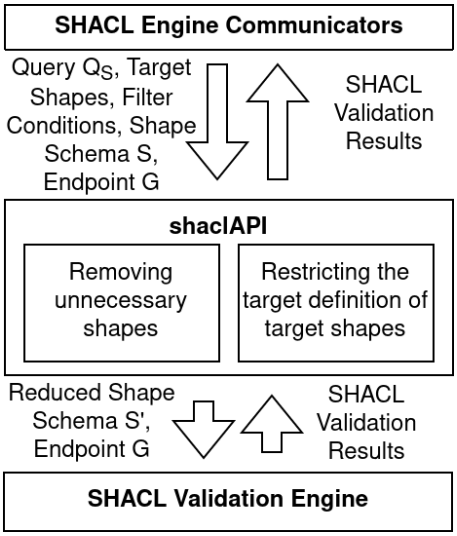


Figure 5.7: **shaclAPI:** Reducing Shape Schemas SHACL-engine-agnostically

SHACL shapes. To avoid this, the SHACL validation of multiple constraints is performed according to the heuristics described in section 4.3.1. These heuristics further minimize the load on the SHACL engine by simultaneously reducing the SHACL schema to the needed shapes and targets. The heuristics were implemented in an API because they work independently of the SHACL engine implementation. The API is called shaclAPI and is publicly available in GitHub[3].

Figure 5.7 illustrates the process. The SHACL Engine Communicators module contacts the shaclAPI per SHACL schema and set of constraints making use of the schema. Algorithm 3 shows the steps needed. However, the shaclAPI is built purpose-independent and, therefore, needs the seed query Q_s instead of the dataset generating query Q_D (see section 4.2.1 for definition and 4.3.1 regarding the conversion) for the reduction of the target definitions of the target shapes mentioned in the constraints. In section 4.3.1, the form of prediction constraints is exploited to further reduce the target definitions of the target shapes by pushing filter terms into the target queries of the shapes. These can be provided optionally per target shape and will be squashed together with the original target definition and the given query Q_S into new target definitions. After removing unnecessary shapes and restricting the target definition, when possible, as summarized above and shown in algorithm 3, the reduced shape schema is forwarded to the SHACL validation engine. The SHACL validation results are returned entity-wise[4] and incrementally to the user of the shaclAPI.

The SHACL validation process is originally triggered by the Dataset module, which stores and joins them according to the join strategy (see section 4.3.2). The raw validation results are stored in the "shacl_validation_results" dictionary and when joined the "sample_to_node_mapping" data frame is extended to additionally store the joined results. The Dataset module also takes care of identifying already validated SHACL schemas in new constraints by calculating MD5 checksums given the serialized representation of the SHACL schema. Therefore, identifying the subset of SHACL schemas given by the constraints.

Storing the results in the Dataset agnostic of the machine learning model allows using

[3]https://github.com/JulianLoewe/shaclAPI

[4]That is, separately for each node addressed by the target definitions of the shapes in the shape schema

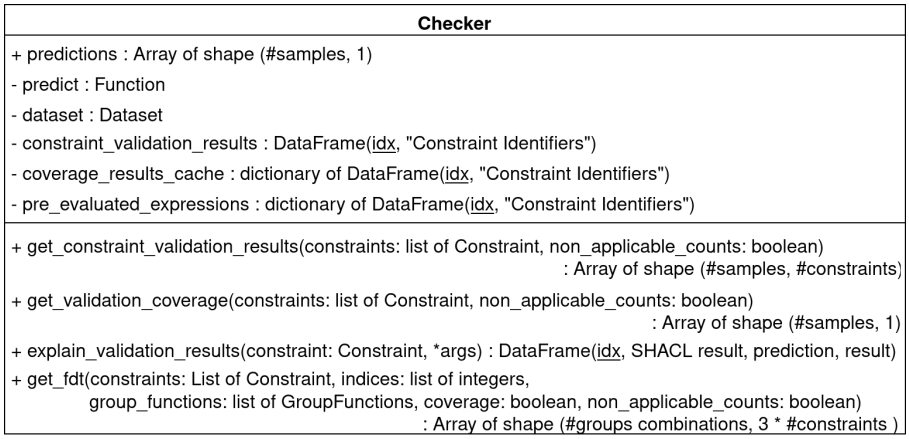


Figure 5.8: **The Checker Class**: The *Checker* stores prediction, constraint validation, coverage results and evaluated expression results, and is able to return summarized validation results.

the SHACL validation results for multiple machine learning models and training iterations without the need for re-validating the SHACL schemas over the knowledge graph. Further, the given implementation of the *PreprocessedDataset* does not push further `FILTER` terms based on the samples used, but just forwards the need for SHACL validation results to the *BaseDataset* instance. This enables the user of a *BaseDataset* `base` to create multiple instances of the ProcessedDataset given `base` without triggering redundant SHACL validations.

Constraint Validation Results and Model Predictions The CHECKER module triggers the validation of the constraints over a given machine learning model. An implementation of a class contained in the CHECKER module is depicted in Figure 5.8. A checker instance is initialized given a dataset and a predict function representing the machine learning model. Both cannot be changed after initialization, as the checker heavily relies on caching to minimize computation time. Here we assume prediction constraints, as data constraints can be evaluated given only the dataset. A prediction constraint requires, for the evaluation, the predictions of the machine learning model over the problem instances in the dataset. As making predictions can be costly, they are cached (in the "predictions" field) to be reused, when evaluating multiple constraints. There are various methods provided by the checker instance, yielding constraint validation results in different formats (see Figure 5.8). All of them start with requesting the SHACL validation results from the DATASET, calculate the constraint validation results per sample in the dataset, and provide a "non_applicable_counts" parameter, which let the user choose to use the 3-valued logic or not. Further, the SHACL validation can be interleaved with the constraint validation by evaluating the right-hand side expression of the prediction constraint first, to determine nodes in the knowledge graph, that do not require SHACL validation results (see section 4.3.1). The evaluation results of the right-hand side of the constraints are then stored in the "pre_evaluated_expressions" field to avoid the recomputation during the final evaluation of the constraint when the needed SHACL validation results are available.
As the validation results are needed in every method and in the case of the decision tree visualization for every node, the results are cached in the "constraint_validation_results" field. The model-coherent visualizations (see section 4.4) are based on frequency distribu-

tion tables. These can be calculated with the "get_fdt" method. The method unifies the frequency distribution table creation from definitions 42 and 43. The latter one is applied if the "coverage" parameter is set to true. In that case, the "get_validation_coverage" method is used to assign each sample in the dataset the constraint validation result with the highest priority (see example 19). The calculation involves combining multiple validation results, which is why the amount of data needed can make the operation costly, and the results are therefore cached in the „coverage_results_cache" field. The further parameters of the function correspond to the definitions provided and are generalized to multiple grouping function by formula 4.16. Finally, the *Checker* class provides a method to explain the constraint validation results by providing the user for each sample in the dataset with the SHACL validation result, the prediction made by the model, and the resulting constraint validation result.

5.2.4 Using Parallel Computation for the Visualization Process

This section is specific to the decision tree visualization (see section 4.4.4) and the algorithm used to compose the different node visualizations. The algorithm is based on the dtreeviz library [63] and is by default a sequential process (as shown in algorithm 5) with similar options for customization as the dtreeviz library provides.

Algorithm 5 Pseudocode of the Constraint Validation Results Annotated Decision Tree Visualization Algorithm

function DTREEVIZ(Decision Tree M_θ, Constraints $\mathcal{C}$, Checker H, boolean *non_applicable_counts*, boolean *coverage*, leaf grouping function Γ)
  $fdts \leftarrow \emptyset$
  $visualizations \leftarrow \emptyset$
  $\Gamma_{\text{nodes}} \leftarrow$ **getDecisionTreeNodeToSamplesMapping**(M_θ)
  for each $node \in$ **splitnodes**(M_θ) **do**
    $f_k \leftarrow$ **splitFeature**$(node)$
    $fdts \leftarrow fdts + \{H.\textbf{get\_fdt}(\mathcal{C}, \Gamma_{\text{nodes}}(node), \Gamma_{\text{bins}}^{f_k}, coverage, non\_applicable\_counts)\}$
  end for
  for each $leaf \in$ **leafnodes**(M_θ) **do**
    $fdts \leftarrow fdts + \{H.\textbf{get\_fdt}(\mathcal{C}, \Gamma_{\text{nodes}}(leaf), \Gamma, coverage, non\_applicable\_counts)\}$
  end for
  for each $fdt \in fdts$ **do**
    $visualizations \leftarrow visualizations + \{$**visualize**$(fdt)\}$
  end for
  return compose$(visualizations, M_\theta)$
end function

The algorithm takes as input the constraints $\mathcal{C}$; to be validated over the decision tree M_θ, a checker instance as described in the previous section, two boolean parameters; specifying whether the 3- or 2-valued logic and the coverage from section 4.4.3 should be used, and the grouping function, which should be used for the leaves (see section 4.4.4). Given the information, the decision-tree-node-to-samples mapping, as discussed in section 5.2.2, is calculated and used afterwards to calculate the frequency distribution table of the constraint validation results for each split- and leaf-node according to section 4.4.4. The frequency distribution tables can then be visualized, as shown in the examples in section 4.4, and need to be composed into the final visualization afterward.

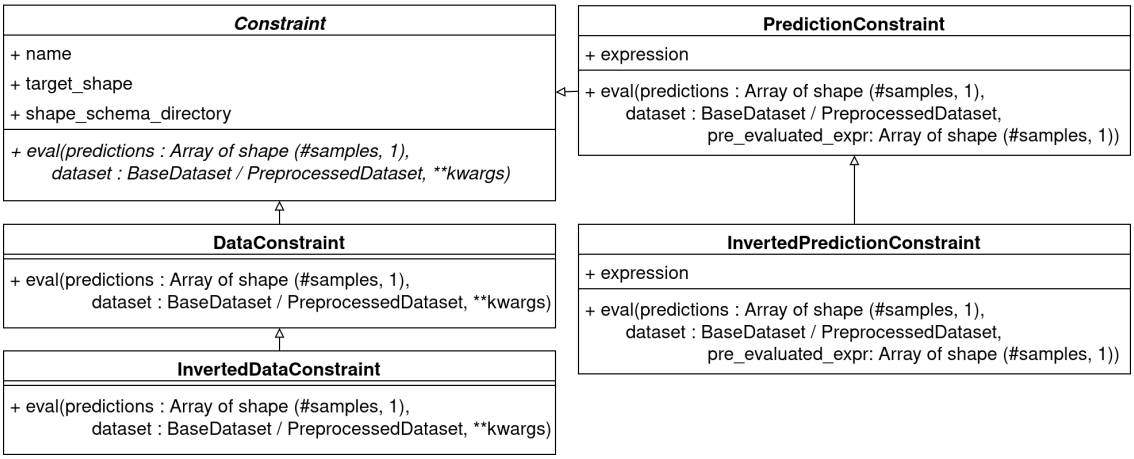


Figure 5.9: The CONSTRAINT Module

Clearly, the algorithm provides several possibilities for parallel computation:

(1) Validating SHACL schemas over the knowledge graph

(2) Computing the constraint validation results

(3) Summarizing the validation results into frequency distribution tables

(4) Visualizing the frequency distribution tables

Option (1) would heavily rely on the multiprocessing performance of the SPARQL endpoint or would require multiple instances of the SPARQL endpoint, which is usually not the case. As will become apparent, much of the computational time of the evaluation process (2) is spent on generating the SHACL results (2), and the parallelization of the join would require a library like dask [57] if one wants to stay with pandas data frame for data management. The implementation of (3) would come either with large main memory requirements or with the need for the validation results to be stored on process-shared read-only memory. Currently, the Python implementation supports (4) as this only requires passing the frequency distribution table to the process and allows outsourcing the time needed to create and write the visualizations onto the disk.

5.3 Portability and Maintainability

Various parts of the implementation were created with extensibility in mind and are structured in such a way that the constraint validation engine is decoupled from the visualization part (see Figure 5.1). This makes the individual components easily maintainable and extensions to the implementation can be made with ease. Further, it turns out that the constraint validation engine is agnostic to the SHACL engine and the machine learning model used.

Constraints The CONSTRAINT module (see Figure 5.9) provides implementations of the constraint types described in section 4.3.4. A central concept is the usage of SHACL constraints to make use of the semantic context of the entities in the knowledge graph which are connected to the samples in the dataset. Therefore, the abstract base class

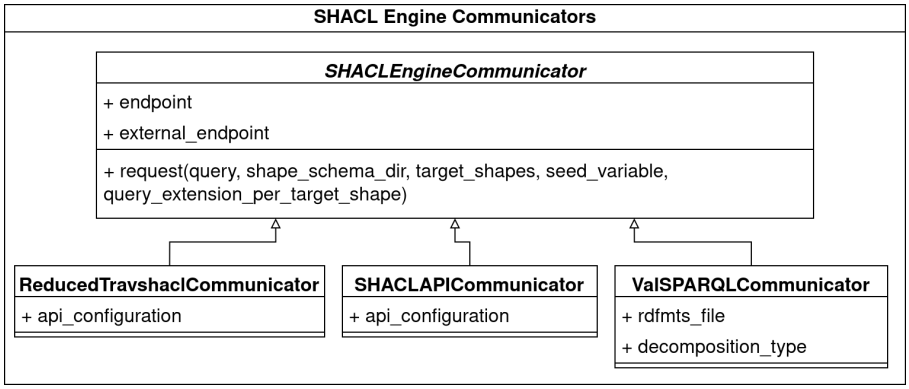


Figure 5.10: The SHACL Engines Communicator Module

Constraint provides the capabilities to integrate SHACL validation results into the constraint evaluation. All constraint types have to inherit from *Constraint* and implement the "eval" method, to be usable with the constraint validation engine. As an example, the *PredictionConstraint* class implements the type of constraint defined in definition 24 by extending *Constraint* with an expression representing the right-hand side of the prediction constraint. A user may change the semantics of the prediction or data constraint by inheriting from them. As an example, the implementation provides inverted constraints, which invert the constraint validation result.

SHACL Engines The SHACL Engines Communicator module (see Figure 5.10) makes the constraint validation engine agnostic to the SHACL engine used. A communicator to be used by the Dataset module has to inherit from the abstract base class *SHACLEngineCommunicator* and has two endpoints assigned. The first one corresponds to an internal endpoint like the shaclAPI (Figure 5.7) and the second one is an external link to the SPARQL endpoint used as the data source and for SHACL validation. Three different communicators are already implemented: The *ReducedTravshaclCommunicator* directly integrates the shaclAPI to reduce the SHACL shape schema for Trav-SHACL [21]. The second and the third ones implement the concept of performing the SHACL constraint validation during SPARQL query execution as described in section 4.3.3. The *SHACLAPICommunicator* makes use of extended capabilities of the shaclAPI to join SPARQL query results with SHACL validation results, while the *ValSPARQLCommunicator* uses the valSPARQL engine to push down the join into the SPARQL query evaluation.

Checker The Checker module includes the *Checker* class as depicted in Figure 5.8 and is model-implementation-agnostic as only one function is required to get predictions from the model. For instance, the implementation of example 21 does not require to make any changes to the *Checker* class. Although, the predictions are now based on the decision tree node and not the whole decision tree. Instead, a prediction method can be defined to return the constant predictions, which would be made by the node (see section 2.3.3).

Group Functions Group functions are a concept introduced in definition 42. Formally, they take a group name and return a set of indices of samples from the dataset belonging to that group. In the implementation, a group function in the Group Functions module

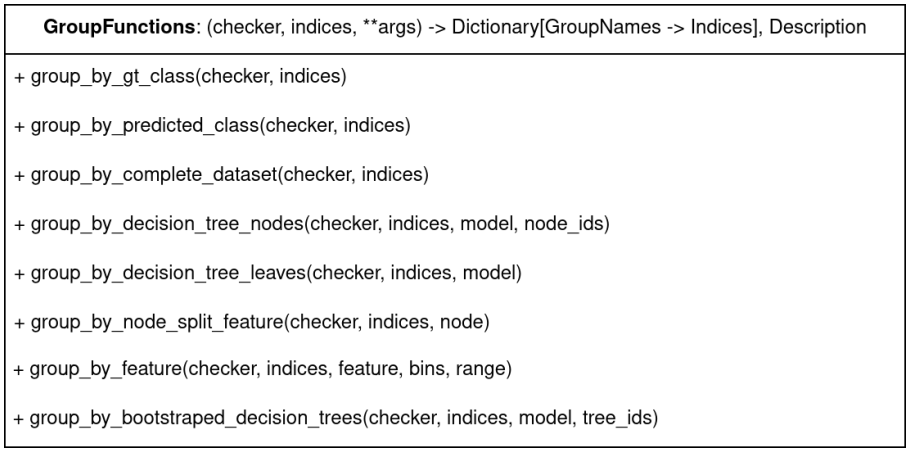


Figure 5.11: The Group Functions Module

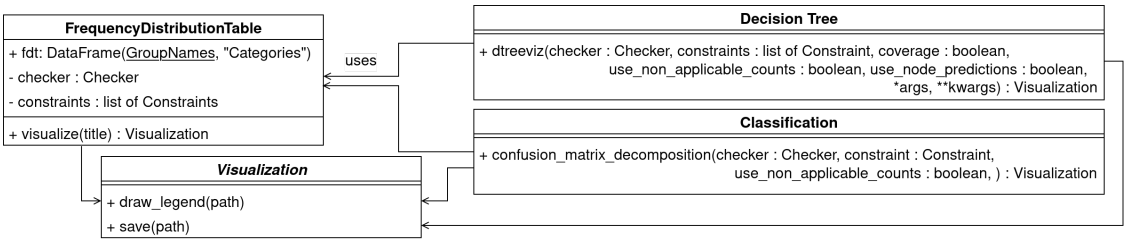


Figure 5.12: The Visualization Module

(see Figure 5.11) takes an instance of the *Checker* class and the indices of the dataset to be grouped and returns a dictionary mapping each of the group names to the subset of indices belonging to that group and a description string saying how the grouping is named. The checker instance provides all the components (e.g., the problem instances, the targets, the predictions, and the sample-to-node mapping) which might be needed to group the indices. However, further arguments might be provided to the group functions if needed. For example, the "group_by_feature" function; generating the theoretical $\Gamma^{f_k}_{\text{bins}}$ function, additional takes the feature, the number of bins, and the range for the binning as inputs. Therefore, new grouping functions can be added easily and used in existing visualizations. For instance, the visualization of the leaves in the decision tree makes use of a grouping function to determine the frequency distribution to show (e.g., Figure 4.19 makes use of group_by_gt (Γ_{gt}) and Figure 4.22 makes use of group_by_complete_dataset (Γ_{all})).

Visualizations In general, the visualizations generated are specific to the model. However, frequency distributions and the according visualizations can be used independently of the model. That is the concepts shown in sections 4.4.1 and 4.4.3 can be applied independent of the chosen model. The information content and the connection to the internal structure of the model then depend only on the grouping function and the fraction of the samples in the dataset selected to be visualized at once. Therefore, the work done in sections 4.4.4 and 4.4.2 is an example of a suitable composition of frequency distributions which can be created for other models as well. Figure 5.12 shows roughly the content of the Visualization module.

Shadow Model The visualization of the model w.r.t. the constraints is based on the internals of the model. However, the library can be chosen according to the user's preference as the visualization algorithm is independent of the concrete implementation of the model, but instead makes use of the adapter pattern (reused from the dtreeviz library [63]; see Figure 5.5).

5.4 Summary

The approach proposed in chapter 4 is realized in a library implemented using Python. The implementation is split into two parts the constraint validation engine and the visualization of the validation results in a model-coherent way.
The validation engine makes heavy use of pandas data frames and NumPy vectorization while implementing the heuristics proposed in section 4.3.1 (i.e., using the shaclAPI) and in section 4.3.2 (i.e., the implementation of the join strategies using the pandas join function). The DATASET module is split into the *BaseDataset* and the *ProcessDataset* to allow for dataset preparation (e.g., data preprocessing, sampling, feature engineering) while keeping the sample-to-node mapping intact.
The visualization part is implemented based on the logic provided by the dtreeviz library. Nevertheless, the logic is extended to support multiprocessing and annotation with multiple constraints. Frequency distribution tables are introduced to split the summarizing stage from the visualization stage and allow for the implemented parallelization of the plot generation. Finally, parts of the algorithm were improved through vectorization with NumPy (e.g., the decision-tree-node-to-sample mapping generation) and caching of intermediate results (i.e., SHACL validation results, constraint validation results, coverage results, and model predictions).
The whole implementation is built for extensibility with respect to new constraint types, the usage of other SHACL engines, the group functions used during summarization, visualizations for new model types, and the library used to train the machine learning model.

Chapter 6

Application: InterpretME

This section introduces a resource that utilizes specific aspects of this thesis and was produced as a collaborative project with four members of the Scientific Data Management Research Group. All authors contributed equally to the resulting resource paper. The resource is called InterpretME [12] and will show how parts of the concepts in this thesis can be applied.

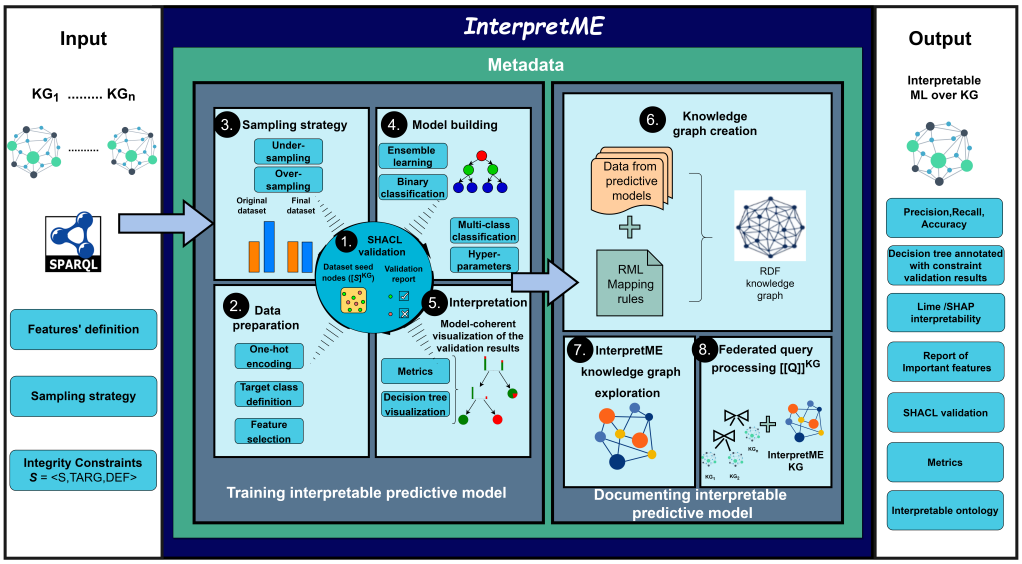


Figure 6.1: The InterpretME Architecture [12]

InterpretME aims at making supervised machine learning models (e.g., decision trees as in this thesis) better interpretable by tracking the different stages used to produce the trained model in an automated way. That is, the concept of a machine learning pipeline is extended to collect metadata about the characteristics of the model and the data used to train the model. Figure 6.1 shows the architecture of InterpretME proposed in the resource paper. In a first step, the user provides the inputs to InterpretME in the form of knowledge graphs, a sampling strategy, a set of integrity constraints and features' definitions. The features' definitions are SPARQL sub-queries, which are combined similar to the property path in section 4.2.1 to a dataset generating query. The query is then used to retrieve the dataset and the sample-to-node mapping; allowing to apply the SHACL validation to samples in the dataset.
In comparison to the sequential process proposed in Figure 1.5 and the general idea

of validating the trained machine learning model, InterpretME postpones the constraint validation step. The step is now called SHACL validation step as the integrity constraints mentioned are the data constraints from section 4.3.4. The training pipeline uses the constraint validation results as additional features. This allows the model to make use of the additional patterns encoded into the validation results and, therefore, in a way make use of the semantic context of the samples in the knowledge graph. Indeed, it turns out that the usage of the constraint validation results improves the performance of the model in terms of accuracy.
In the final step of the model training pipeline, the model is evaluated w.r.t. performance metrics (see section 2.3.2); interpretability methods (e.g., LIME [56] and dtreeviz [63]) are applied and the approach from section 4.4 is used. The latter one allows to spot patterns in the usage of the samples, annotated with the constraint validation results, by the model.
The data collected (e.g., the inputs, the sample-to-node mapping, the model evaluation results, and the results of the interpretability methods) is composed into a new knowledge graph. In terms of this work, the new knowledge graph simplifies and facilitates the interpretation of the model predictions w.r.t. the original entity and the constraint validation results. This is achieved by exploiting the sample-to-node mapping to align the entities in the new knowledge graph to the entities in the input knowledge graphs through `owl:sameAs` connections. For example, results of interpretability methods are usually only associated with the sample in the dataset, which makes it hard to understand the context of the sample under specific circumstances. However, the generated knowledge graph can be queried together with the input knowledge graphs in a federated manner to give the interpretability result for a specific sample together with its original semantic context and the matching constraint validation results. In this context, the constraint validation results can also be used as flags to mark certain entities of interest.
Evaluating InterpretME shows that the interpretability of machine learning models can be increased by, among other things, applying the methods proposed in this work.

Chapter 7

Experimental Evaluation

In this section, the approach is evaluated in combination with the proposed heuristics using the Python implementation and with respect to the following research questions:

Q1 What is the impact of the join heuristics proposed in section 4.3.2?

Q2 What is the impact of reducing the SHACL shape schemas and the simultaneous generation of SHACL schema validation results, depending on the schema topology and the knowledge graph used?

Q3 What is the impact of the parallel node plot generation as mentioned in section 5.2.4?

Q4 How does the decision tree visualization implementation perform in comparison to dtreeviz?

Q5 How much is the overhead added by the validation engine and the visualization of the annotated decision tree compared to the time needed to train the model and other interpretability methods (e.g., LIME [56])?

The experimental settings are as follows:

Benchmarks There are four different benchmarks used during the experimental evaluation. The first two use synthetic data to allow for load tests and the construction of benchmarks suitable to show the effect of the heuristics proposed. The latter two are benchmarks as performed in [12] to show two more realistic applications.

Synthetic Data A consists of a knowledge graph G with semantic data generated based on a distribution of entities to classes and a shape schema $\mathcal{S} = (\{0, 1, 2, ..\}, \texttt{TARG}, \texttt{DEF})$ such that $[[\texttt{TARG}(i)]]_G$ gives the entities of class i. Figure 7.1 depicts the two shape schema topologies which will be used: The first is a star-shaped one (short: star), and the second is a binary tree-shaped one (short: bt). For both shape schema topologies, three shape schemas of different sizes are generated, which are used in combination with the distribution of the entities to classes as schematically depicted in Figure 7.2. The different test beds of *Synthetic data A* can be identified with topology_K_distribution, where the topology is star or bt, distribution is in $\{o, d, n\}$, and K denotes the number of constraints (each of the target shapes colored in Figure 7.1 belongs to a data constraint). The knowledge graph is built such that for each $s \in S$, half of the entities in $[[\texttt{TARG}(s)]]_G$ will be valid according to s.

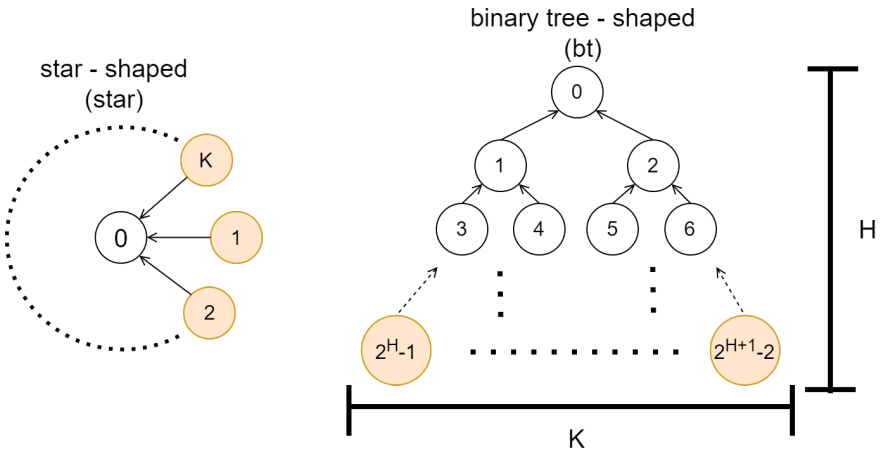


Figure 7.1: **Shape Schema Topologies of *Synthetic Data A***: K denotes the number of constraints, H the height of the binary tree. The nodes represent the shapes, and a directed edge connecting shape i with shape j represents an inter-shape constraint ($\geq_1$ `:link_ij`.j). There are no intra-shape constraints, except shape 0 enforces the intra-shape constraints ($\geq_l$ `:literal_k`.$\top$) $\wedge$ $\neg$($\geq_{(m+1)}$ `:literal_k`.$\top$) for all $(k, l, m) \in \{(1, 1, 2), (2, 2, 4), (3, 1, 8), (4, 2, 16)\}$. The K colored nodes will be used as target shapes for data constraints.

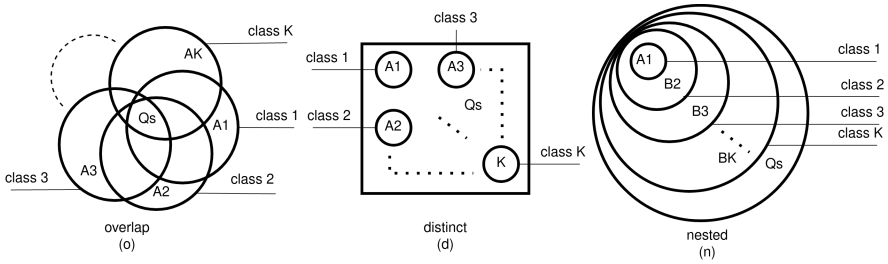


Figure 7.2: **Venn Diagrams of Classes in *Synthetic Data A***: Qs is the set of entities, which will be retrieved by the seed query in the benchmarks building on *Synthetic Data A*. K denotes the number of constraints. Each set starting with A contains $4,000$ entities, and each set starting with B contains $4,000$ + (#numberOfBSetsContained + 1) * $2,000$ entities. Each class not included in the figure contains $4,000$ entities. The diagram only shows the classes associated with a target definition of a target shape of a constraint.

The generated knowledge graph G only contains the entities and literals necessary to satisfy or violate the constraints, but does not include any further data which might be used to create the dataset and define a prediction task. Therefore, when running the validation engine over one of the test beds, the dataset generating query Q_D only retrieves the seed nodes (e.g., the entities in Qs) to build the sample-to-node mapping η. However, this is enough to evaluate the validation engine with the heuristics proposed in section 4.3 based on data constraints as they do not require a trained machine learning model to be evaluated.

Synthetic Data B complements *Synthetic Data A*, as the steps which could not be evaluated with *Synthetic Data A* can be evaluated with *Synthetic Data B*, i.e., the performance of the visualization algorithm. Therefore, the dataset D, the sample-to-node mapping η, as well as the entity validation function (see definition 18) are generated synthetically on the basis of the number of samples N (#samples) of the dataset D, the number of seed nodes M (#nodes), and the number of constraints $|\mathcal{C}|$ (#constraints). The algorithm used to generate D is the one originally used to generate the „Madelon“ dataset [29], which is implemented in the scikit-learn library [10]. It basically creates clusters of samples with the same label. However, the clusters can overlap. The predictive task is a classification task; aiming to label each problem instance with its original cluster in the dataset. The default parameters used to generate D can be found in table 7.1. The samples-to-node mapping η is generated by choosing randomly with uniform probability and replacement from a set of unique seed nodes with cardinality M. The validate function is populated as needed by the engine: Given a request $(v, s, G) \in (\mathbf{B} \cup \mathbf{I}) \times \mathbf{S} \times \mathbf{G}$ and a constraint $C \in \mathcal{C}$ the validation result is generated by a random process from $\{\top, \bot, \mathit{None}\}$, where *None* indicates that the entity validation function does not contain a validation result for the given request. The random process samples for each $C \in \mathcal{C}$ a triple $(p_1, p_2, p3)$ uniformly from $[0, 1]^3$. The triple is normalized to $(\hat{p_1}, \hat{p_2}, \hat{p_3})$ such that $\sum_{i \in \{1,2,3\}} \hat{p}_i = 1$. Finally, for each request: $\top$ will be chosen with probability $\hat{p_1}$, $\bot$ with $\hat{p_2}$ and *None* with $\hat{p_3}$. The data constraints can be generated by choosing a random shape schema directory and a target shape[1]. The decision tree used in the benchmark is trained on D with the default parameters provided by the scikit-learn library and with a maximal depth of 5 unless specified otherwise.

The French Royalty KG is an extended version of the fully curated one proposed in [30]. For each person in the knowledge graph, the class `dbo:Person` is added. In addition, the different numbers of children, predecessors, and other counts are materialized. Furthermore, the KG is extended to include the rule-based derived `dbo:hasSpouse` relationships from [30]. Overall, this results in a knowledge graph with $3,430$ entities; having an average of 9.2 triples with the entity as the subject. The prediction task is to classify whether a person in the knowledge graph has a spouse and is tackled by the InterpretME pipeline (section 6) to give a trained decision tree. A data constraint is created for each logical rule proposed in [30] to explain a `dbo:hasSpouse` link. This procedure results in ten constraints used to validate the samples based on entities of type `dbo:Person`. Each of the constraints is based on a separate shape schema with a single shape. Precisely, the constraints use different combinations of the `dbo:hasChild`, `dbo:hasParent`, and `dbo:hasSpouse` links. For example, one constraint describes that two different people who have the same child should both have a

[1]Both do not have to exist as they will be only used to query the entity validation function for SHACL validation results

Parameter	Value
number of clusters per class	1
number of informative features	2
number of redundant/repeated features	0
number of samples	4^{10}
number of seed nodes	4^{10}
number of constraints	5

Table 7.1: The default parameters used during the generation of test beds of *Synthetic Data B*

spouse and another one stating, that if person A has person B as a spouse then person B also has a spouse.

The Lung Cancer KG is a knowledge graph from the biomedical domain about lung cancer patients. For each lung cancer patient, the knowledge graph includes attributes of the individual such as gender, age, cancer stage, smoking habits, and biomarkers for lung cancer. There are a total of $21,340,353$ entities with each of them occurring in an average of 2.97 triples as the subject. In this case, the prediction task is binary classification to predict whether the patient has "ALK" or "other" as biomarker. Again, the task is tackled by the InterpretME pipeline to give a trained machine learning model. Data constraints are created on the basis of medical protocols. That is, special treatments should be prescribed in accordance with the patients' biomarker values. Precisely, four different data constraints are defined based on a medical protocol specifying that patients without the EGFR biomarker should not take the drugs Afatinib and Gefitinib. Each constraint is based on a target shape included in separate star-shaped SHACL schemas with three shapes (the topology is the star-shaped one in Figure 7.1 with $K = 2$).

Engines As mentioned in section 5.3 the validation engine is agnostic to the SHACL engine; during the experimental evaluation Trav-SHACL [21] is used. It is accessed through the „ReducedTravshaclCommunicator", which makes use of the shaclAPI (Figure 5.7); implementing the heuristics in 4.3.1.

Metrics The following metrics will be used during the evaluation of the experiments:

Number of SHACL validation results: Given a set of constraints $\mathcal{C} \subset \mathbf{C}$ and a knowledge graph $G \in \mathbf{G}$, the number of SHACL validation results is |dom(validate)|, where validate is the entity validation function generated by a SHACL engine during the evaluation of the constraints in $\mathcal{C}$.

Execution time: The elapsed time spent on executing an experiment or a part of it; measured using the Python library *time*[2]. Each experiment is executed five times and the average execution time, as well as the standard deviation of the execution times, are reported. All the experiments are executed on a machine equipped with an Intel® Core(TM) i5-6500 at 3.20 GHz and 8 GiB RAM.

[2]https://docs.python.org/3/library/time.html

bt				star				
7_o	15_o	31_o	15_d	5_o	15_o	25_o	15_d	5_n
34.6	117.0	416.5	79.9	29.3	231.0	633.6	142.4	46.1

Table 7.2: Execution time [s] observed during the execution of the validation engine on performing the SHACL validation of all constraints in 9 test beds of *Synthetic Data A* without using any heuristic.

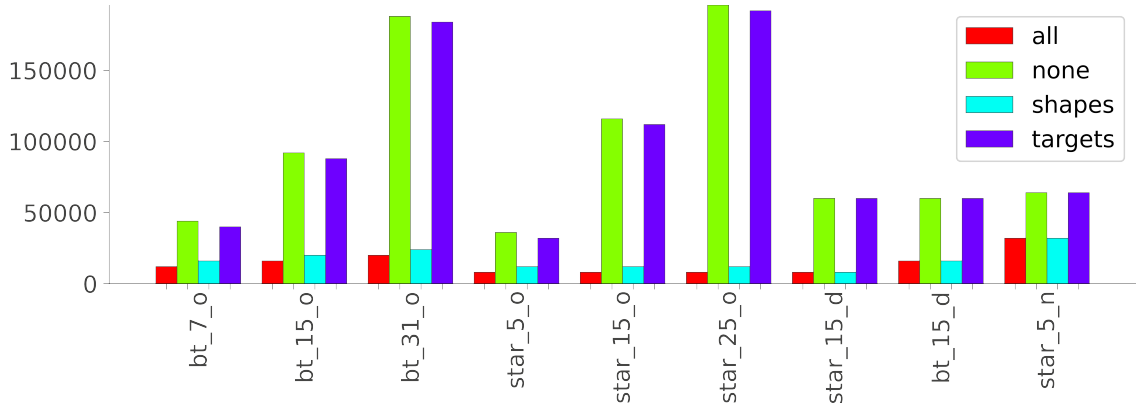
7.1 Validation Engine

This section aims to experimentally evaluate the validation engine by using the synthetically generated benchmark setups in *Synthetic Data A* and *Synthetic Data B*. *Synthetic Data A* provides knowledge graphs and different shape schemas, which will allow answering the research question **Q2**. *Synthetic Data B* provides a dataset, the sample-to-node mapping as well as the entity validation function, which will be used to evaluate the join strategy (see section 4.3.2). Both benchmarks will lead to an answer to the research question **Q1**. Evaluating the validation engine in two parts by answering **Q1** and **Q2** is natural, as can be seen in algorithm 2; the SHACL constraint validation is performed first (line 4) and the constraint evaluation (line 5), which in case of data constraints reduces to the execution of the join, follows.

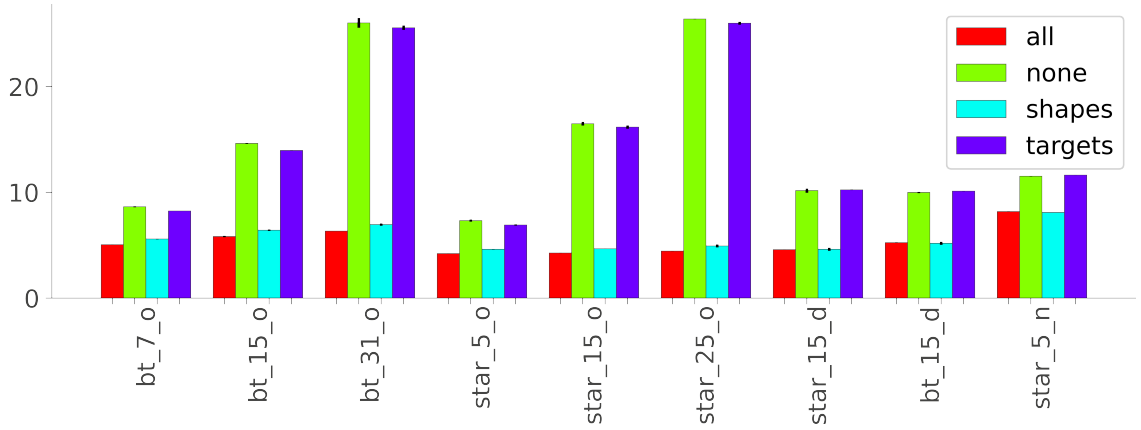
7.1.1 SHACL Constraint Validation

An ablation study based on nine different test beds of *Synthetic Data A* is performed to show the effects of the heuristics proposed in section 4.3.1. Therefore, the execution time of the SHACL constraint validation (e.g., applying the heuristics, running the SHACL engine, collecting the SHACL validation results) and the number of SHACL validation results generated in the process are measured.
The study is split into two parts: In the first part, only a single constraint of each test bed is evaluated over the dataset extracted from the knowledge graph. In the second part, all constraints as shown in Figure 7.2 are evaluated. Figure 7.3 presents the results of the first part of the ablation study (with none as the baseline). The results of the second part of the ablation study are shown in Figure 7.4 with the baseline separately shown in table 7.2. Based on the results, the impact of the heuristics is discussed.

Prune Shape Network refers to the heuristic proposed in lemma 6 and excels in reducing the number of SHACL validation results needed to be generated in the first part of the ablation study. That is because in both shape schema topologies used, the number of shapes can be reduced, when only validating a single constraint. In the star-shaped case, the reduction is theoretically able to remove $(K-1)$ shapes leaving 2 shapes in the network, and in the binary-tree-shaped case even $2*(K-1)-\log_2(K)$ shapes leaving $\log_2(K)+1$ shapes in the network. In the „overlap“-case, the number of SHACL validation results can be estimated to be $(\#\text{shapes}*4,000+K*4,000)$. Therefore, the total number of SHACL validation results in the „overlap“ case is reduced from $(K+1)*4,000+K*4,000$ resp. $(2*K-1)*4,000+K*4,000$ in the baseline case (e.g., not using any heuristic) to $2*4,000+4,000$ resp. $(\log_2(K)+1)*4,000+K*4,000$ when only pruning the shape network. The proactive reader might calculate the theoretical reduction of the number of SHACL validation results with the formula for the „distinct“-case $\#\text{shapes}*4,000$ and

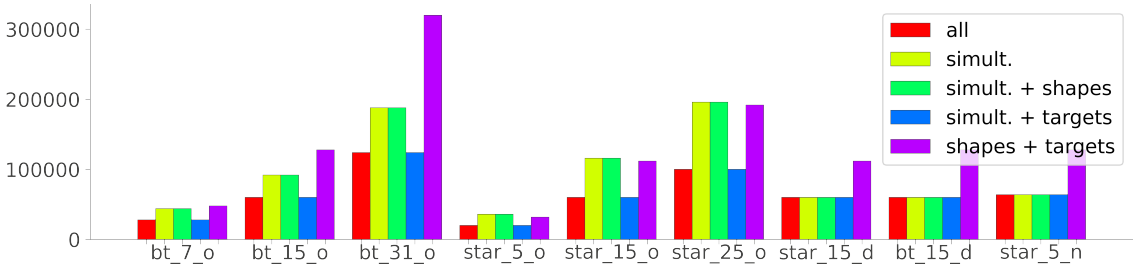


(a) Number of SHACL validation results

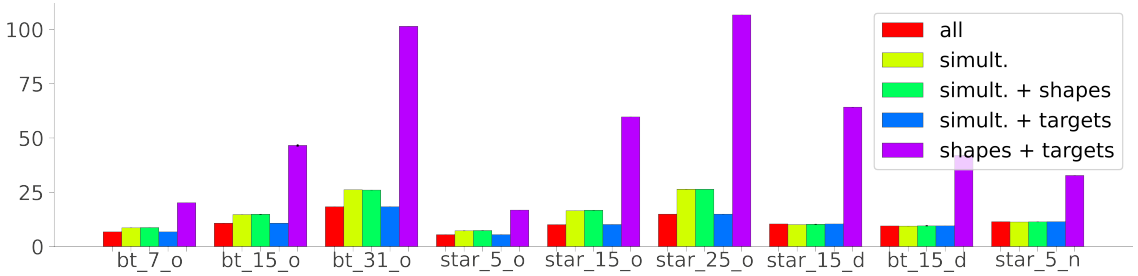


(b) Execution time [s]

Figure 7.3: Metrics observed during the execution of the validation engine on performing the SHACL validation of a single constraint in nine test beds of *Synthetic Data A*. none - the baseline does not make use of any heuristics, shapes - the shape network is pruned, targets - the target reduction is applied, all - combination of shapes and targets



(a) Number of SHACL validation results



(b) Execution time [s]

Figure 7.4: Metrics observed during the execution of the validation engine on performing the SHACL validation of all constraints in nine test beds of *Synthetic Data A*. simult. - constraints are validated simultaneously, shapes - the shape network is pruned, targets - the target reduction is applied, all - combination of simult., shapes and targets

the „nested"-case $(\#\text{shapes} - K) * 4,000 + \sum_i^K (4,000 + (i-1) * 2,000)$. As it turns out, the theoretical numbers exactly match the estimated numbers in Figure 7.3a, which proves the functionality of the heuristic for the given test beds. Comparing the results in figures 7.4 and 7.3 shows a strong correlation between the number of SHACL validation results and the execution time; emphasizing the importance of the shape network pruning when possible. In the second part of the ablation study, the shape network pruning does not impact as none of the shape schemas can be reduced.

Target Reduction refers to the heuristic proposed in lemma 5. Both, in the first part and in the second part of the ablation study, the target reduction impacts and reduces the number of targets of the target shapes given by the constraints. Precisely, in the first part of the ablation study, in the „overlap"-case, the target reduction exactly removes the $4,000$ additional entities included in the target definition of the shape but not included in Qs. This scales in the second part of the ablation study to $K * 4,000$ as for each constraint $4,000$ entities were validated without being included in the seed nodes. In the „distinct"- and „nested"-case, the set of entities Qs includes all the entities in the target definitions of the shapes, which is why the target reduction does not have a further impact and the number SHACL validation results stay the same in all cases. Even though the target reduction was not able to reduce the number of SHACL validation results as much as the shape network pruning, it still impacts the average execution time as can be seen in figures 7.3b and 7.4b.

Simultaneous Constraint Validation refers to the heuristic proposed in lemma 7 and, therefore, only applies to cases in which multiple constraints are given, which is not the case in the first part of the ablation study. However, it excels in the second part

by preventing the SHACL engine to produce redundant validation results. Even if the heuristics above are able to reduce the number of SHACL validation results needed, still shapes are processed multiple times when validating each constraint on its own. All setups in Figure 7.4 make use of the heuristics except for the „shapes + targets“-configuration. Therefore, Figure 7.4a shows the impact of the simultaneous constraint validation on the number of SHACL validation results. In theory, the impact can be calculated as follows: In the case of the star-shaped shape schemas, shape 0 is validated K times over the knowledge graph, and in the case of the binary tree, a shape at depth i is validated $2^{(h-i)}$ times. This is the reason for the additional SHACL validation results, which the „shapes + targets“-configuration generates in comparison to the „all“-configuration. Although the two heuristics perform well compared to the baseline, still the shaclAPI needs to apply the heuristics above unnecessarily often, the SHACL validation is performed with smaller and more queries and the validation engine needs to manage unnecessary validation results. All these components contribute to a comparatively high execution time of the „shapes + targets“-configuration in Figure 7.4b.

Conclusion The preceding discussion leads to an answer to the research question **Q2**. Given the nine test beds, pruning the shape network alone reduced the execution time by 65% on average, while reducing the target definition of the target shapes only reduced the execution time by 3% on average. The simultaneous constraint validation was able to improve the average execution time by 90%. Applying all heuristics reduced the execution time by 94% on average. Further, the number of SHACL validation results strongly correlates with the execution time and whether the heuristic was able to reduce the number of actions (e.g., queries to the endpoint) of the SHACL engine. All heuristics proposed in section 4.3.1 at least impact on the execution time of the SHACL validation in the same order of magnitude they were able to shrink the number of SHACL validation results. However, this depends on the knowledge graph and the SHACL shape schema. Additionally, figures 7.4 and 7.3 show that in cases the heuristics do not apply, it also does not impact negatively in the given test beds.

7.1.2 The Join Strategy

In section 4.3.2 two different join strategies were presented:

A) *Join T at the end.* The SHACL validation results are first joined together before being linked to the dataset via the sample-to-node mapping. The two join heuristics (see lemma 8 and 9) propose to use this approach to keep intermediate results as small as possible and, in turn, minimize the execution time of the join operations in Figure 4.2a. However, full-outer-joins are required to keep all SHACL validation results. The implementation of lemma 8 was described in section 5.2.

B) *Join T at the beginning.* The SHACL validation results are directly joined with the dataset via the sample-to-node mapping. The intermediate results are always as large as possible, but only left-outer joins are required to keep all samples in the dataset. Furthermore, this strategy can be used when joined SHACL results are needed early.

In the first step, the *Synthetic Data B* benchmark is used for a load test of the implementation. Therefore, the number of samples, nodes, or constraints are varied, to see the effect of each of the parameters of the test bed on the execution time of the join strategy. Figure 7.5 presents the results in terms of execution time.

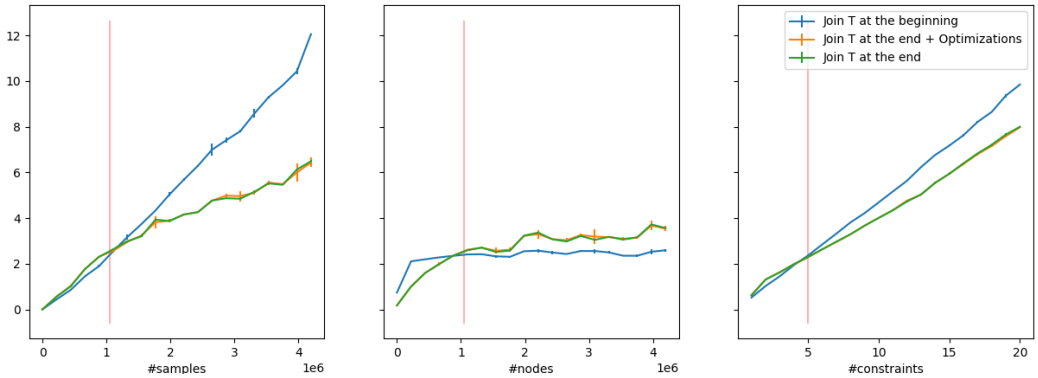
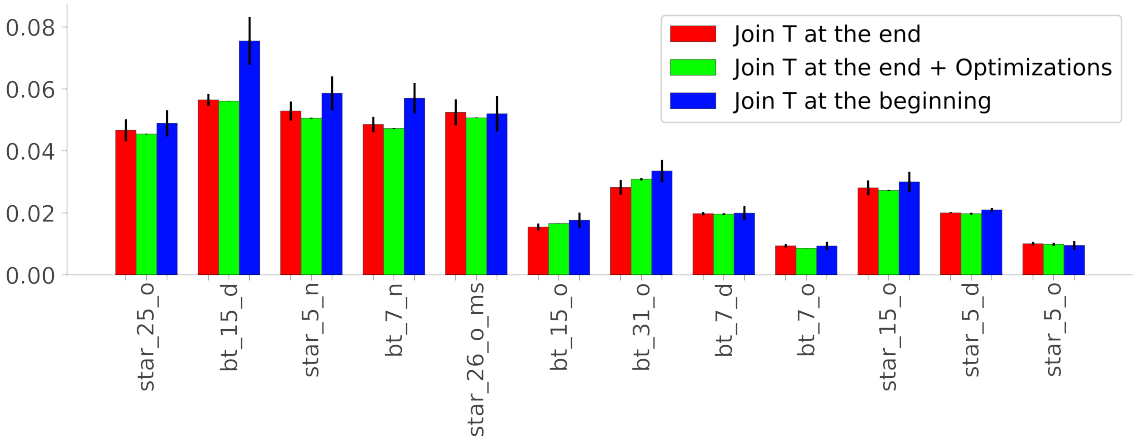
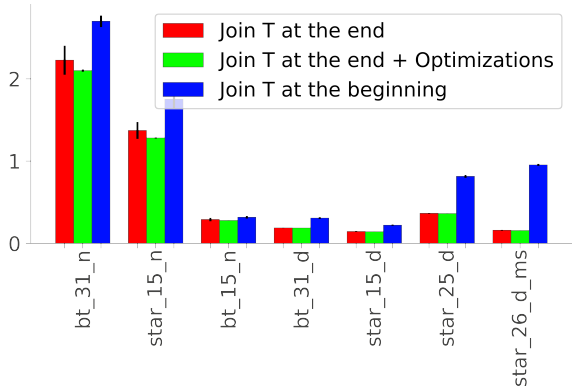


Figure 7.5: Execution time [s] of the different join strategies observed during the execution of the validation engine on performing different join strategies for test beds of *Synthetic Data B* with a varied number of samples (#samples), number of seed nodes (#nodes) or number of constraints (#constraints). The vertical red line marks the default value of the varied parameter as shown in table 7.1

Varying the number of samples increases the time spent on joining with strategy **A** and **B** in a roughly linear fashion. In the case of strategy **A** resp. **B** the execution time raises approximately by 1.38 resp. 2.75 seconds per million samples. However, a small number of samples results in a larger portion of unique nodes in the sample-to-node mapping in relation to the number of samples, and strategy **B** turns out to be faster. Similar behavior can be seen when varying the number of seed nodes: Raising the number of seed nodes and, therefore, increasing the portion of unique nodes in the sample-to-node mapping in relation to the number of samples, strategy **B** is faster. In the opposite case, strategy **A** is faster. Varying the number of constraints scales the execution time in a linear fashion. Finally, one can observe that the further optimizations to decrease the cardinality of the intermediate results do not impact in case of the given benchmark.
Both observations may be due to the random nature of *Synthetic Data B*: A large portion of unique nodes makes overlapping of entities of the SHACL validation results unlikely. On the other hand, a small to moderate portion of unique nodes makes overlapping more probable. However, the random nature of the experiment is not natural. In the real world, the entities often match or overlap (e.g., multiple constraints about different person types) or are nested into each other (e.g., constraints becoming more specific about a protocol). This motivates the *Synthetic Data A* benchmark, in which entities are distributed to classes and the SHACL validation results are not generated by a random process and assigned to a random subset of the seed nodes. This is the second step of the evaluation of the join strategy to validate the behavior observed in the preceding benchmark. The results are depicted in Figure 7.6.
First of all, one can observe that strategy **B** performs worse or as good as strategy **A** without optimizations. A reason for that might be the nested or distinct class distributions of the entities to classes, which encourage constantly growing intermediate results. However, that is even true for the „overlap“-case in which the number of SHACL validation results is equal for all constraints. The latter can be due to the implementation of strategy **B** having a larger overhead in comparison to strategy **A**. That is because **B** has to use multiple pandas join calls where, on the other hand, **A** only requires a maximum of two calls.
Next, one can observe that the optimizations impact differently depending on the distri-



(a) Cheap Joins



(b) Expensive Joins

Figure 7.6: Execution time [s] observed during the execution of the validation engine on performing different join strategies for 19 different test beds of *Synthetic Data A*. ms - marks a variation of the test bed, in which the target definition of five shapes each refer to the same class, Optimizations - The optimizations to reduce the number of intermediate results (see section 5.2)

bution of entities to classes. However, the execution times generally seem to vary less in the optimized case.
In the „nested“-case, it is necessary to order the SHACL validation results of the different target shapes according to their size to get constantly growing intermediate results during the execution of the join strategy. Therefore, the optimizations impact and the average execution time decreases by approximately 6%.
The test beds ending with “_ms” are specially designed only to produce constantly growing intermediate results during the join when the SHACL validation results of the constraints with a target shape having the same class in their target definition are joined next to each other. However, the experiments do not show a difference in the execution time of strategy **A** when using the optimizations or not.
In the case of a distinct distribution, the intermediate results will constantly grow independent of the order in which the SHACL validation results are joined. Therefore, it is logical that the optimizations do not impact, and the average execution time of the join strategy is approximately the same in all of these test beds.
In the „overlap“-case, the order does not matter at all, and, as already noted, the intermediate results are of constant size independent of the join order. In this case, the optimization adds an unnecessary overhead (e.g., sorting by the number of SHACL validation results and grouping per target definition). Therefore, it can even lead to worse execution times, as observed in some cases.
In summary, research question **Q1** can be answered: Join strategy **A** is indeed faster in cases where the distribution of entities to classes is chosen in a way that encourages the production of constantly growing intermediate results. The second part of the experiments even shows that in the case of a relatively small number of seed nodes, strategy **B** is slower in general. Further, optimizing the order of the join operations in an approach to keep the intermediate results small stabilizes the execution time of the join strategy.

7.2 Visualization Algorithm

This section aims at answering the research question **Q3**, while also investigating the execution time of the different stages of the visualization algorithm. In addition, results will be collected on research question **Q4**. Again, benchmark *Synthetic Data B* is used. However, the validation engine is run before the experiment such that the constraint validation results are already available and linked to the samples of the dataset.
First, a load test is conducted over the visualization algorithm running in serial and in parallel mode. In parallel mode, the plots of the nodes of the decision tree are created in parallel. The experiment is executed analog to the load test of the join experiment: On top of the number of nodes (#nodes), constraints (#constraints) and samples (#samples), the maximum depth of the decision tree is varied, to see the effect of the height of the decision tree on the execution time. The visualization algorithm can be divided into different stages, whose execution times are measured individually.

summarizing. The stage summarizing the constraint validation results into frequency distribution tables. Corresponds to lines 5 - 11 in algorithm 5.

histogram creation. The stage visualizing the given frequency distribution tables corresponding to the split nodes of the decision tree with histograms. Corresponds to line 13 in algorithm 5.

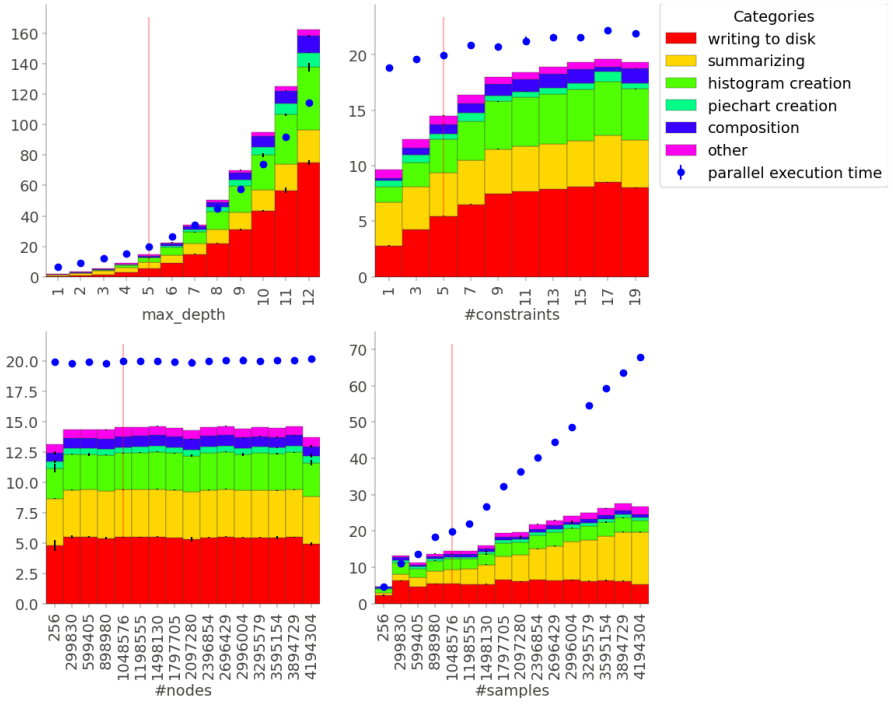


Figure 7.7: Execution time [s] observed during the execution of the visualization algorithm for test beds of *Synthetic Data B* with a varied number of samples, nodes, constraints and maximal depth of the trained decision tree. The bars show the execution time of the different components of the visualization algorithm running in serial mode. The dots show the overall execution time of the visualization algorithm running in parallel mode (see section 5.2.4). The vertical red line marks the default value of the varied parameter as shown in table 7.1.

pie chart creation. The stage visualizing the given frequency distribution tables corresponding to the leaves of the decision tree with pie charts. Corresponds to line 13 in algorithm 5.

writing to disk. The stage storing the created visualizations (e.g., the pie charts and the histograms) to disk.

composition. The stage composing the stored visualizations into decision trees. Corresponds to line 15 in algorithm 5.

other. Intermediate steps (e.g., intermediate result transformations, estimation of the decision-tree-node-to-samples mapping (see section 5.2.2))

The visualization algorithm makes use of the coverage concept introduced in section 4.4.3. The results of the experiment are depicted in Figure 7.7 and are the basis to answer the research question **Q3**.
Varying the maximal depth of the decision tree, to be visualized with the annotations, corresponding to the constraint validation results, raises the execution time exponentially due to the exponentially rising number of split and leaf nodes of the decision tree. Further, the parallel node plot generation is able to improve the execution time in cases of high decision trees. For decision stumps (e.g., decision trees of depth 1) and shallow decision trees, the parallel execution time is higher than the serial execution time due to the

overhead coming with multiprocessing. The overhead of the multiprocessing even raises linear with the number of samples in the dataset, although process spawning is used, which should not copy the dataset, but only the validation results summarized into frequency distribution tables to the assigned processes. In the serial execution, the execution time for summarizing the validation results increases proportional to the number of samples, while keeping the execution times of the other stages approximately constant. Therefore, the overhead added by the multiprocessing should be constant as the summarizing step is not executed in parallel; pointing to an error in the implementation of the parallel node plot generation. Varying the number of seed nodes (#nodes) to be considered during the generation of the test bed does not impact on the execution time of the visualization algorithm. Conversely, varying the number of constraints impacts the time spend on writing to disk and creating the histograms up to a specific point, in which the complexity of the histograms stops increasing as the concept of coverage suppresses the visualization of additional less important validation results. This answers the research question **Q3**: Parallel node plot generation can impact positively in the case of high decision trees. However, with the current implementation, multiprocessing comes with a large overhead; scaling is proportional to the number of samples in the dataset. This currently limits the application since the overhead will surpass the time savings in most cases.

As mentioned in section 5, the visualization algorithm uses the logic provided by the dtreeviz library. However, the implementation had to be extended to support the multiprocessing and the visualization of multiple constraints. Further, the concept of frequency distribution tables as the basis of each node visualization was added for implementation convenience. It basically allowed to split the summarizing stage from the visualization stage. Finally, parts of the algorithm were improved through vectorization with NumPy (e.g., the decision-tree-node-to-sample mapping generation (see section 5.2.2)). The next experiment investigates the performance of the visualization algorithm in comparison to its original implementation in the dtreeviz library and, therefore, aims at answering the research question **Q4**.

Again, the experiment is conducted using the test beds provided by *Synthetic Data B*. For comparability reasons, the visualization algorithm only has to visualize the validation results of a single constraint since the original implementation only showed the distribution of the ground truth values of the samples per node of the decision tree. In Figure 7.7 it can be seen that the visualization of the split nodes is more costly compared to the visualization of the leaves of the decision tree. To further investigate this behavior, all test beds are used to measure the execution time of the visualization algorithm and the dtreeviz implementation; once for the visualization of the leaves only (e.g., the fancy option of the dtreeviz library turned off) and once for the entire decision tree. This time only the maximal depth of the trained decision tree and the number of samples in the dataset are varied since the number of seed nodes has shown to not impact the performance of the visualization algorithm. The results of the experiment are depicted in Figure 7.8.

At first glance, the visualization algorithm clearly outperforms the dtreeviz implementation: Varying the maximal depth or the number of samples in the dataset in all cases, the visualization algorithm is faster than the dtreeviz implementation.

For both implementations, keeping the maximal depth of the decision tree constant, the execution time scales proportionally to the number of samples. The execution time of the dtreeviz implementation and the visualization algorithm implementation increases by about 85 and 0.9 seconds per million samples respectively given a maximal depth of 5 of the decision tree.

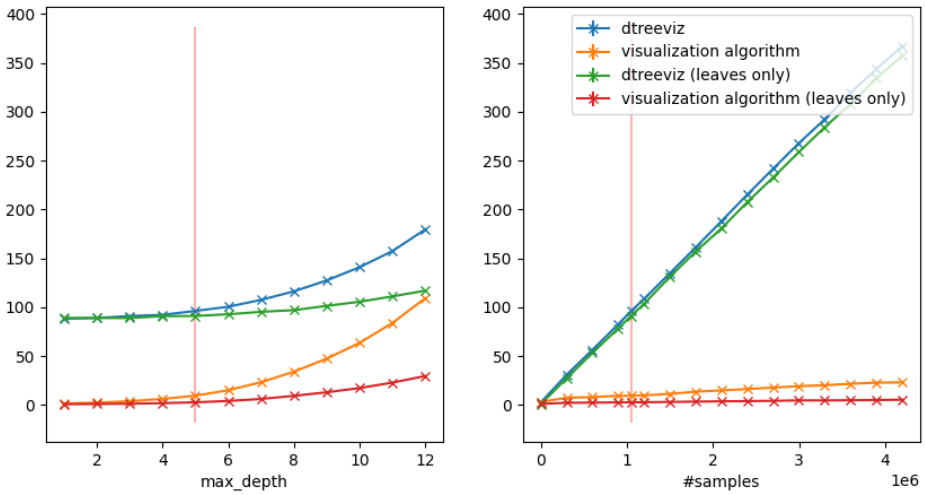


Figure 7.8: Execution time [s] observed during the execution of the visualization algorithm and it's corresponding implementation of the dtreeviz library for test beds of *Synthetic Data B* with a varied number of samples in the dataset (right figure) and trained decision trees of a varied maximal depth (left figure). The vertical red line marks the default value of the varied parameter as shown in table 7.1.

However, the visualization algorithm seems to scale slightly worse with a raising maximal depth of the decision tree. Limiting the visualization to the leaves of the decision tree reduces the execution time to the execution time needed to visualize an entire decision tree, but only of approximately half the depth. This is the expected behavior as a decision tree of height h has $2^h - 1$ split nodes and 2^h leaf nodes.
Nevertheless, depending on the number of samples, the visualization algorithm starts with a time advantage which can be partially derived from the improvement of the decision-tree-nodes-to-samples mapping creation. To experimentally verify this theory, a final experiment is conducted with the same setup as before, but now only measuring the execution time needed by the visualization algorithm and by the dtreeviz implementation for generating the decision-tree-node-to-samples mapping (Γ_{nodes}). Figure 7.5 presents the results in terms of execution time.
Indeed, the experiment shows that the improved decision-tree-nodes-to-samples mapping creation, which makes use of the vectorized code execution, is faster for the observed settings. However, the conversion shown in Figure 5.6 seems to make the creation dependent on the maximum depth of the decision tree (with the original version such a dependency was not recognized). The improvement is exponential for a relatively small number of samples but turns out to be linear for a larger number of samples. In the test beds using a million samples (e.g., the ones on the left of Figure 7.8) approximately 70 seconds of the initial time advantage are because of the worse decision-tree-nodes-to-samples mapping creation implementation.
Therefore, the research question **Q4** can now be answered: The visualization algorithm scales considerably better w.r.t. the number of samples in the dataset. This gives the visualization algorithm an initial time advantage when testing the scalability of the visualization algorithm with respect to the maximum depth of the decision tree. However, the visualization algorithm seems to scale slightly worse with a growing height of the decision

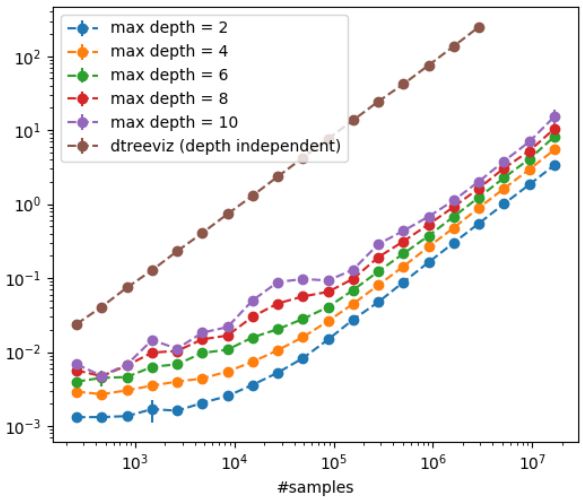


Figure 7.9: Execution time [s] of the generation of the decision-tree-node-to-samples mapping, observed during the execution of the visualization algorithm and its corresponding implementation of the dtreeviz library for test beds of *Synthetic Data B* with a varied number of samples in the dataset and trained decision trees of a varied maximal depth.

benchmark	#samples		#nodes	
	before prep.	after prep.	before prep.	after prep.
LC-KG	2,102	428	1,083	295
FR-KG	2,212	1,988	2,212	1,988

Table 7.3: Number of samples and seed nodes of the extracted dataset for the non-synthetic benchmarks before preparation through the InterpretME pipeline and afterwards.

tree.

7.3 Summary: Real Data Application

This section is meant to summarize the previous results by applying them to the non-synthetic benchmarks (*The French Royalty KG* (FR-KG) and *The Lung Cancer KG* (LC-KG)). Furthermore, research question **Q5** is answered based on these benchmarks. The InterpretME pipeline from section 6 is applied to the benchmarks while measuring the time spent on the different stages of the validation engine and the visualization algorithm as in the previous sections. After InterpretME extracts the dataset from the endpoint, the dataset is prepared for training of the model (e.g., removing duplicates, under-sampling, encoding categorical features, and feature selection). Table 7.3 shows the number of samples and the number of unique seed nodes in the datasets before and after the preparation. In each run of the benchmark the decision tree, trained with the prepared dataset, is visualized; annotated once for each constraint with the constraint validation results of the constraint, and once annotated with all constraint validation results using the concept of coverage (see section 4.4.3).

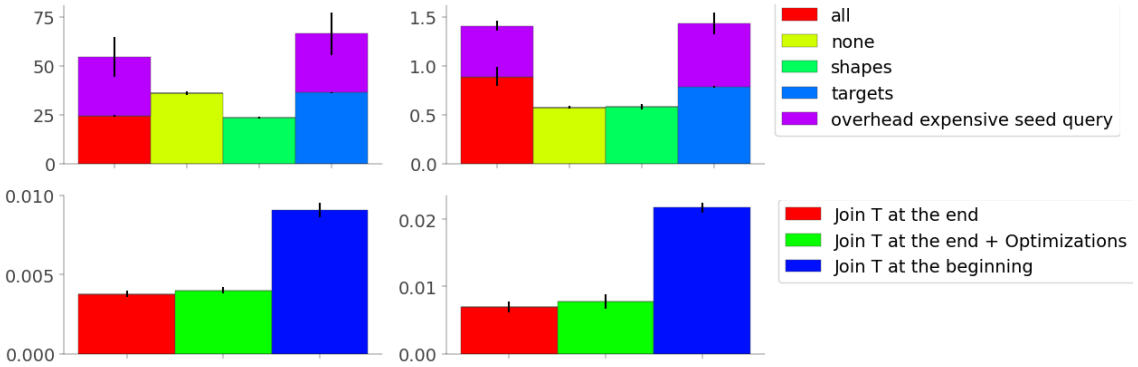


Figure 7.10: Execution time [s] observed during the execution of the validation engine on performing different join strategies (at the bottom) and the SHACL validation using the different heuristics (at the top) for the non-synthetic benchmarks (FR-KG on the right and LC-KG on the left).

The Validation Engine needs to be executed once for each benchmark. Figure 7.10 presents the execution times for the different heuristics and strategies evaluated in section 7.1 for the two non-synthetic benchmarks. The execution time spent on executing the join is lower when joining T at the end. This is the expected outcome since the strategy has already shown to excel in cases the distribution of entities to classes is chosen in a way that encourages the production of constantly growing intermediate results or the number of samples is relatively low (the answer to **Q1**; see section 7.1.2). The latter criterion applies to both benchmarks (see Figure 7.3) and in the case of *The French Royalty KG* all target shapes of the constraints are targeting partially overlapping subsets of entities of the class `dbo:Person`. Optimizing the order of the join operations does not impact positively but only adds a marginal overhead.
The heuristics to reduce the number of shapes can be applied to the SHACL shape schemas of the constraints in *The Lung Cancer KG*. Each shape schema contains three shapes of which only two are required and indeed the execution time decreases when applying the heuristic. However, the other available heuristics cannot impact positively: In both cases, the target definitions of the target shapes of the constraints are already subsets of the set of seed nodes and simultaneous constraint validation is not applicable as none of the constraints share the SHACL shape schema. Instead, it can be seen that generating the seed query from the dataset generating query, as shown in section 4.3.1, adds an overhead to the SHACL validation. This is due to the execution of expensive queries generated by the SHACL engine while performing the SHACL validation. Specifying the seed query manually (e.g., using the Q_s from section 4.2.1) leads to the results matching the answer to the research question **Q2**.
Finally, the non-synthetic benchmarks show that optimizing the SHACL constraint validation impacts more than optimizing the join strategy. Further, it should be emphasized that the positive impact of the heuristic depends strongly on the shape schema and the KG (the answer to **Q2**; see section 7.1.1).

The Visualization Algorithm is applied multiple times during the execution of the InterpretME pipeline. Figure 7.11 shows the fractions of the execution time of the visualization algorithm spent for the two non-synthetic benchmarks. These portions are nearly independent of the size of the decision tree which is why the figure does not differentiate between the two benchmarks. However, the number of constraints that are visualized

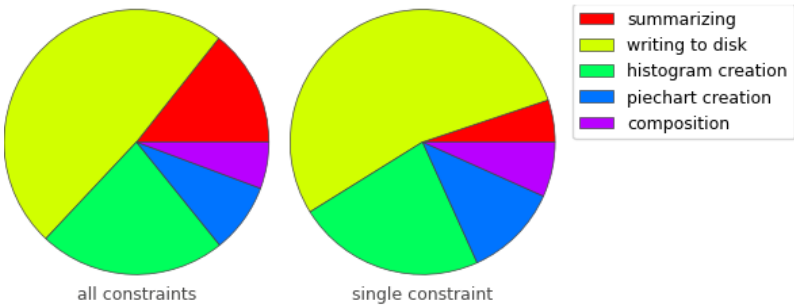


Figure 7.11: Average portion of the execution time spent on different stages of the visualization algorithm observed for *The Lung Cancer KG* and *The French Royalty KG*

impacts the time needed to summarize the constraint validation results (cf. Figure 7.7); which can also be observed in Figure 7.11. Visualizing the decision tree with a single constraint took 3.62 seconds on average for *The Lung Cancer KG* and 2.51 seconds on average for *The French Royalty KG* (with standard deviations of under 0.05 seconds). This is as fast as the dtreeviz implementation or slightly faster and matches the answer to the research question **Q4** from section 7.2: The visualization algorithm excels at a larger number of samples, but scales slightly worse than dtreeviz with the size of the decision tree. When the decision tree should be annotated with the constraint validation results of multiple constraints using the coverage concept, it took 4.45 seconds on average for *The Lung Cancer KG* and 3.80 seconds on average for *The French Royalty KG* (with standard deviations of under 0.16 seconds). Again, it becomes obvious that the time needed to annotate the decision tree with the constraint validation results of multiple constraints grows slower than linearly with the number of constraints.
Generating the decision tree node plot visualizations in parallel raises the execution time of the visualization algorithm when annotating the decision tree with the constraint validation results of a single constraint to 4.75 seconds on average for *The Lung Cancer KG* and 4.15 seconds on average for *The French Royalty KG* (with standard deviations of about 0.5 seconds). The decision tree has to be of a specific size, which grows linear with the number of samples in the dataset, to benefit from the multiprocessing implemented (the answer to **Q3**; see section 7.2).

Summary. Finally, research question **Q5** asking about the overall overhead added by the validation engine and the visualization algorithm compared to the normal model training and other kinds of interpretability methods (e.g., dtreeviz and LIME) can be answered. As a reference, measurements of the execution times of the InterpretME pipeline are used; they are listed in table 7.4. Here the overall execution time of the approach is made up of the dataset extraction, the model training, the constraint evaluation per sample in the dataset, and the visualization of the annotated decision tree. It is assumed that all constraints are used to annotate a single decision tree using the coverage concept. In the average worst-case scenario, the approach took up to 73.33 seconds (using the generated seed query; only applying the target reduction; joining T at the beginning) in the case of *The Lung Cancer KG* and up to 6.65 seconds in the case of *The French Royalty KG* (with standard deviations of 10.86 and 1.76 seconds resp.). However, in the average best-case scenario, the approach took 30.39 and 5.8 seconds resp. (using the manually specified seed query; only pruning the shape network; joining T at the end; with standard deviations of 1.18 and 0.28 seconds resp.). Regarding **Q5** that makes the execution time

benchmark	dataset extraction		model training		dtreeviz		LIME	
	avg.	std.	avg.	std.	avg.	std.	avg.	std.
LC-KG	0.77	0.45	1.68	0.07	4.05	0.70	1894.67	19.14
FR-KG	0.20	0.06	1.19	0.04	2.51	0.09	424.12	3.02

Table 7.4: Reference execution times [s] measured for *The French Royalty KG* (FR-KG) and *The Lung Cancer KG* (LC-KG). dataset extraction – time spent executing the dataset generating query; model training – time spent on finding an appropriate model for the predictive task of the benchmark (includes preprocessing, sampling, feature selection, hyper parameter grid search and decision tree training); dtreeviz – time spent to visualize the trained decision tree with dtreeviz; LIME – time spent on calculating LIME interpretability results for a quarter of the samples in the dataset

of using the approach up to 44 times resp. 6 times slower with respect to the model training but still up to 62 resp. 73 times faster than using LIME. Due to the large differences in execution time depending on the benchmark, no clear answer can be given. However, due to the complexity of the approach (see section 4.2.4) expensive scenarios will lead to a large overhead of the approach.

Chapter 8

Conclusions and Future Work

8.1 Lessons Learned

This work makes use of semantic constraint validation to support explainable AI. Semantic constraint validation excels in checking structured data against constraints. Machine learning models in the context of explainable AI are often model-inherent explainable. Nevertheless, their predictions lack rationality for human constraints or scientific facts. Combining explainable AI with semantic constraint validation allows for interpretability and explainability combing with more trust in predictions.
Technically, the concepts were combined by aligning the entities in the knowledge graph (i.e., the set of seed nodes) with the samples in the dataset by the sample-to-node mapping. Two types of constraints are proposed: *Prediction constraints* can be used to explain and check the model's predictions based on a correct data basis. The semantic context of the problem instance (represented through an entity in the knowledge graph) is used to grade the prediction to be correct or incorrect. If the prediction is correct, an explanation is found for the prediction. On the contrary, the constraints' statement reasons why the prediction is wrong and may be used to improve the model. *Data constraints* measure the trustworthiness of the samples in the dataset. They are able to do so since they measure the integrity of the semantic context of the entities to be valid or invalid[1].
The first problem formulated is the efficient validation of the constraints over the machine learning model given the semantic context of the entities in the dataset. It is tackled by a validation engine integrating the dataset extraction, the model training, the SHACL validation, and the constraint validation. Heuristics are proposed to speed up the execution of the engine, exploiting the actual need for SHACL validation results, the join strategy, caching of intermediate results, and vectorization techniques. A Python implementation for the engine is provided. As well as the approach, the implemented engine is agnostic to the machine learning model and the SHACL engine. Based on the implementation, the approach is evaluated with respect to the engine's performance. The evaluation is conducted using two synthetic and two non-synthetic benchmarks. Minimizing the number of SHACL validation results by generating only the required SHACL validation results for each constraint at a time and avoiding re-evaluation of shapes, the execution time of the SHACL validation could be reduced to 6% of a naive approach not using the heuristics. Choosing a good join strategy does impact, but the total time savings are less than the total time savings from optimizing the SHACL validation, as joining is a less complex operation.

[1] or even not applicable in the case of the 3-valued logic

The second problem formulated is the summarization of the knowledge gained through validating the constraints to make the behavior of the machine learning model clearer to the user. Frequency distribution tables are consulted for the summarization of a single constraint. A concept, referred to by coverage, allowed to reduce the constraint validation results of multiple constraints by prioritizing. Based on that, model-coherent summaries are created for decision trees in the form of annotated decision trees. The visualizations created this way are based on the dtreeviz library, but implemented to support the parallel node plot generation and the annotation with the validation results of multiple constraints. The experimental evaluation has shown that the parallelization approach is not yet mature, but the coverage allows visualizing multiple constraints with a less than linear increasing execution time. A comparison of the visualization algorithms' execution time with the execution time of the dtreeviz library showed that it scales considerably better w.r.t. the number of samples in the dataset, but slightly worse w.r.t. the depth of the decision tree visualized.
The overall approach is integrated into the InterpretME pipeline, and an evaluation of the pipeline has shown that the methods in this work can provide improved interpretability. Visualization-specific interpretations w.r.t. over- and underfitting are given as a suggestion for further constraint-depending interpretations.

8.2 Limitations

The approach proposed in this work comes with some limitations. First, it is assumed that the dataset is extractable from a knowledge graph. Although the semantic web continuously grows, most of the time, the data will not be instantly available as a knowledge graph. In that case, the available (un)structured data needs to be transformed into RDF knowledge graphs (e.g., using mapping rules [34]). However, the transformation alone will most likely not be enough when the entities in the data do not come with a sufficient semantic context, which goes beyond the features to be extracted for each entity. This implies that the knowledge graph may have to be extended or connected with further data. As turned out in section 7.3, the heuristics require a simple seed query to be used efficiently. Therefore the set of seed nodes may need to be marked with a class. So in this case it is also necessary to have the SPARQL endpoint either locally available or the permissions for modifications.
Next, extracting the dataset has to be possible in an efficient way (i.e., such that the time to execute the SPARQL query is reasonable), this may require further modifications to the knowledge graph. As the constraints are defined based on SHACL, the constraints can only be as expressive as the SHACL specification allows, given the knowledge graph. Specific constraints involving negations in recursive dependencies may not be evaluated in polynomial time by a SHACL engine (see section 4.2.4).
Further, as usual for approaches making use of rule-based explanations, domain knowledge is needed to define the constraints as well as to interpret the results. Most likely the constraints need to be user-specified to be well understood. User-defined constraints also come with the short-coming of only detecting patterns in the data, which are also suspected (e.g., one can only detect persons with an invalid birth date, if one creates a constraint that checks for it).
Finally, *Data Constraints* can be used to check for the integrity (i.e., trustworthiness) of the data. This is even necessary as explanations are only available for *Prediction Constraints*, which do require a valid data basis to be applicable. Moreover, to check a model *indirectly* against facts encoded as constraints, the data basis has to be large and

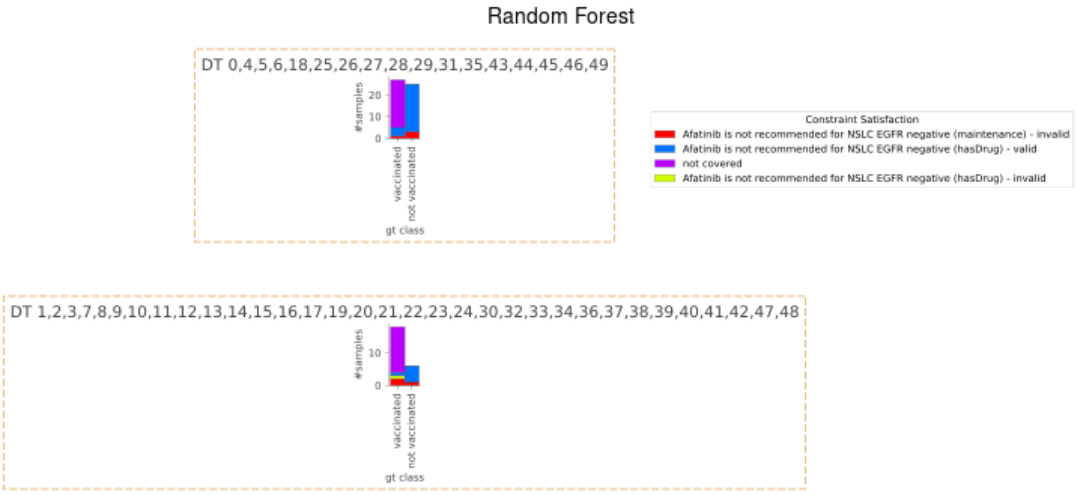


Figure 8.1: Draft for the visualization showing the representative leaves in a random forest corresponding to a specific prediction given a problem instance. The bootstrapping process is tracked, to identify the samples on which the prediction is based for each decision tree. Similar constraint validation result distributions are clustered using KMeans and only a representative per cluster is shown. The orange border indicates that the majority of decision trees in the cluster voted for the prediction. *The Lung Cancer KG* provides the data on which the random forest is trained.

diverse enough. This is especially an obstacle if a small subset of the available data should be used for validation (see section 2.3.2), which will not contain a problem instance for every harmful case one might want to detect. The latter is usually referred to by the *curse of dimensionality* [8].

8.3 Future Work

In future work, the visualization approach may be extended to further machine learning models (i.e., based on decision trees: Random Forests, AdaBoost Gradient Boosting). Figure 8.1 shows a draft for the visualization of the prediction a random forest makes based on a problem instance. The dtreeviz library is continuously extended, e.g., by adding a visualization of the classification boundaries or the feature-target space. Both kinds of visualizations could be used to further show the constraint validation results in different contexts. Section 4.3.3 presented the theory of performing SHACL constraint validation during the SPARQL query execution and promises further execution time improvements, when applied in a way that avoids the duplicate validation of shapes.

List of Figures

List of Tables

Bibliography

[1] Haney Alom. 2022. *A Library for Visualizing SHACL over Knowledge Graphs.* Master's thesis. Gottfried Wilhelm Leibniz Universität Hannover - Fakultät Für Elektrotechnik und Informatik.

[2] Daniel W. Apley and Jingyu Zhu. 2020. Visualizing the effects of predictor variables in black box supervised learning models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 82, 4 (2020), 1059–1086. https://doi.org/10.1111/rssb.12377

[3] Farshad Bakhshandegan Moghaddam, Carsten Draschner, Jens Lehmann, and Hajira Jabeen. 2021. Literal2Feature: An Automatic Scalable RDF Graph Feature Extractor. In *Further with Knowledge Graphs*, Mehwish Alam, Paul Groth, Victor de Boer, Tassilo Pellegrini, Harshvardhan J. Pandit, Elena Montiel, Víctor Rodríguez Doncel, Barbara McGillivray, and Albert Meroño-Peñuela (Eds.). IOS Press. https://doi.org/10.3233/SSW210036

[4] David Beckett. 2014. RDF 1.1 N-Triples. W3C Recommendation. https://www.w3.org/TR/n-triples

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. 2014. RDF 1.1 Turtle. W3C Recommendation. https://www.w3.org/TR/turtle/

[6] Tim Berners-Lee, Roy Fielding, and Larry Masinter. 2005. *RFC3986: Uniform Resource Identifiers (URI): Generic Syntax.* RFC 3986. RFC Editor. 1–61 pages. https://www.rfc-editor.org/rfc/rfc3986.txt

[7] Tim Berners-Lee, James Hendler, and Ora Lassila. 2001. The Semantic Web : a new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American* 284, 5 (2001), 34–43. https://www.jstor.org/stable/26059207

[8] Christopher M. Bishop. 2006. *Pattern recognition and machine learning.* Springer, New York, NY. http://www.loc.gov/catdir/enhancements/fy0818/2006922522-d.html

[9] Carl Boettiger. 2018. rdflib: A high level wrapper around the redland package for common rdf applications. https://doi.org/10.5281/zenodo.1098478

[10] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning.* 108–122.

[11] Weiwei Cheng, Gjergji Kasneci, Thore Graepel, David Stern, and Ralf Herbrich. 2011. Automated feature generation from structured knowledge. In *Proceedings of the 20th ACM international conference on Information and knowledge management - CIKM '11*, Iadh Ounis, Ian Ruthven, and Craig Macdonald (Eds.). ACM Press, New York, New York, USA, 1395–1404. https://doi.org/10.1145/2063576.2063779

[12] Yashrajsinh Chudasama, Julian Gercke, Disha Purohit, Philipp D. Rohde, and Maria Esther Vidal. 2022. Generating Interpretable Machine Learning Models over Knowledge Graphs. under review. https://doi.org/10.5281/zenodo.6523741

[13] Julien Corman, Fernando Florenzano, Juan L. Reutter, and Ognjen Savković. 2019. SHACL2SPARQL: Validating a SPARQL Endpoint against Recursive SHACL Constraints. In *Proceedings of the ISWC 2019 Satellite Tracks*. CEUR-WS.org, Aachen, Germany, 165–168.

[14] Julien Corman, Fernando Florenzano, Juan L. Reutter, and Ognjen Savković. 2019. Validating Shacl Constraints over a Sparql Endpoint. In *The Semantic Web – ISWC 2019*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtěch Svátek, Isabel Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.). Lecture Notes in Computer Science, Vol. 11778. Springer International Publishing, Cham, 145–163. https://doi.org/10.1007/978-3-030-30793-6{_}9

[15] Julien Corman, Juan L. Reutter, and Ognjen Savković. 2018. Semantics and Validation of Recursive SHACL. In *The Semantic Web – ISWC 2018*, Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl (Eds.). Lecture Notes in Computer Science, Vol. 11136. Springer International Publishing, Cham, 318–336. https://doi.org/10.1007/978-3-030-00671-6{_}19

[16] IBM Corp. [n. d.]. Explainable AI. https://www.ibm.com/watson/explainable-ai.

[17] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll, and Brian McBride. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/

[18] Xiangfeng Dai, Marwan Bikdash, and Bradley Meyer. 2017. From social media to public health surveillance: Word embedding based clustering method for twitter classification. In *SoutheastCon 2017*. IEEE, 1–7. https://doi.org/10.1109/SECON.2017.7925400

[19] Anna Essén, Isabella Scandurra, Reinie Gerrits, Gayl Humphrey, Monika Alise Johansen, Patrick Kierkegaard, Jani Koskinen, Siaw-Teng Liaw, Souad Odeh, Peeter Ross, and Jessica S. Ancker. 2018. Patient access to electronic health records: Differences across ten countries. *Health Policy and Technology* 7, 1 (2018), 44–56. https://doi.org/10.1016/j.hlpt.2017.11.003

[20] Tom Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (2006), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010

[21] Mónica Figuera, Philipp D. Rohde, and Maria-Esther Vidal. 2021. Trav-SHACL: Efficiently Validating Networks of SHACL Constraints. In *Proceedings of the Web Conference 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM, New York, NY, USA, 3337–3348. https://doi.org/10.1145/3442381.3449877

[22] Aaron Fisher, Cynthia Rudin, and Francesca Dominici. 2019. All Models are Wrong, but Many are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously. *Journal of Machine Learning Research* 20, 177 (2019). http://jmlr.org/papers/v20/18-760.html

[23] Ronald A. Fisher. 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics* 7, 2 (1936), 179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

[24] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics* (Oct. 2001), 1189–1232. https://doi.org/10.1214/aos/1013203451

[25] Mohamed H. Gad-Elrab, Daria Stepanova, Trung-Kien Tran, Heike Adel, and Gerhard Weikum. 2020. ExCut: Explainable Embedding-Based Clustering over Knowledge Graphs. Springer, Cham, 218–237. https://doi.org/10.1007/978-3-030-62419-4{_}13

[26] Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.

[27] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining Explanations: An Overview of Interpretability of Machine Learning. In *2018 IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 80–89. https://doi.org/10.1109/DSAA.2018.00018

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. The MIT Press, Cambridge, Massachusetts and London, England.

[29] Isabelle Guyon, Steve Gunn, Asa Ben-Hur, and Gideon Dror. 2004. Result Analysis of the NIPS 2003 Feature Selection Challenge. *Advances in neural information processing systems* 17 (2004).

[30] Nicholas Halliwell, Fabien Gandon, and Freddy Lecue. 2021. User Scored Evaluation of Non-Unique Explanations for Relational Graph Convolutional Network Link Prediction on Knowledge Graphs. In *Proceedings of the 11th on Knowledge Capture Conference*, Anna Lisa Gentile and Rafael Gonçalves (Eds.). ACM, New York, NY, USA, 57–64. https://doi.org/10.1145/3460210.3493557

[31] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[32] Ivan Herman, Sergio Fernández, Carlos Tejo Alonso, and Alexey Zakhlestin. 2022. SPARQLWrapper. https://github.com/RDFLib/sparqlwrapper

[33] John D. Hunter. 2007. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[34] Enrique Iglesias, Samaneh Jozashoori, David Chaves-Fraga, Diego Collarana, and Maria-Esther Vidal. 2020. SDM-RDFizer. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudre Mauroux (Eds.). ACM, New York, NY, USA, 3039–3046. https://doi.org/10.1145/3340531.3412881

[35] ISO/IEC 25010. 2011. ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. (2011).

[36] J. Backus. 1959. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. (1959). https://www.semanticscholar.org/paper/The-syntax-and-semantics-of-the-proposed-algebraic-Backus/d075466245c0a58a6c2c98198ae2c6d937b0af11

[37] Kaggle. [n. d.]. State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021

[38] Holger Knublauch and Dimitris Kontokostas. 2017. Shapes Constraint Language (SHACL). W3C Recommendation. (july 2017). https://www.w3.org/TR/shacl/

[39] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. 2014. Test-driven evaluation of linked data quality. In *Proceedings of the 23rd international conference on World wide web - WWW '14*, Chin-Wan Chung, Andrei Broder, Kyuseok Shim, and Torsten Suel (Eds.). ACM Press, New York, New York, USA, 747–758. https://doi.org/10.1145/2566486.2568002

[40] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. 2017. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research* 18, 17 (2017), 1–5. http://jmlr.org/papers/v18/16-365.html

[41] Sven Lieber, Ben De Meester, Pieter Heyvaert, Femke Brückmann, Ruben Wambacq, Erik Mannens, Ruben Verborgh, and Anastasia Dimou. 2021. Visual notations for viewing RDF constraints with UnSHACLed. *Semantic Web* (2021), 1–36. https://doi.org/10.3233/SW-210450

[42] S. Manikandan. 2011. Frequency distribution. *Journal of Pharmacology & Pharmacotherapeutics* 2, 1 (2011), 54–56. https://doi.org/10.4103/0976-500X.77120

[43] Kenneth Marino, Ruslan Salakhutdinov, and Abhinav Gupta. [n. d.]. The More You Know: Using Knowledge Graphs for Image Classification. https://doi.org/CVPR

[44] Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. 2019. Anytime Bottom-Up Rule Learning for Knowledge Graph Completion. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, Thomas Eiter and Sarit Kraus (Eds.). International Joint Conferences on Artificial Intelligence Organization, California, 3137–3143. https://doi.org/10.24963/ijcai.2019/435

[45] Pablo N. Mendes, Max Jakob, and Christian Bizer. 2012. DBpedia: A Multilingual Cross-Domain Knowledge Base. (2012).

[46] Tom M. Mitchell. 2013. *Machine learning* ([nachdr.] ed.). McGraw-Hill, New York.

[47] Christoph Molnar. 2022. *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable* (second edition ed.). Christoph Molnar, Munich.

[48] Venkata Narasimha, Pavan Kappara, Ryutaro Ichise, and OP Vyas. 2011. LiDDM: A Data Mining System for Linked Data. In *Workshop on Linked Data on the Web. CEUR Workshop Proceedings*, Vol. 813. 108.

[49] Natanael Arndt, André Valdestilhas, G. Publio, Andrea Cimmino Arriaga, Konrad Höffner, and Thomas Riechert. 2021. A Visual SHACL Shapes Editor Based On OntoPad. *SEMANTiCS Posters&Demos* (2021). https://www.semanticscholar.org/paper/A-Visual-SHACL-Shapes-Editor-Based-On-OntoPad-Arndt-Valdestilhas/21785c2ecacfd60ecf5c74703ed74a14a2d55106

[50] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[51] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems* 34, 3 (2009), 1–45. https://doi.org/10.1145/1567274.1567278

[52] Luka Petravi'c, Rok Arh, Tina Gabrovec, Lucija Jazbec, Nika Rupči'c, Nina Starešinič, Lea Zorman, Ajda Pretnar, Andrej Srakar, Matjaž Zwitter, and Ana Slavec. 2021. Factors Affecting Attitudes towards COVID-19 Vaccination: An Online Survey in Slovenia. *Vaccines* 9, 3 (2021). https://doi.org/10.3390/vaccines9030247

[53] Eric Prudhommeaux, Steve Harris, and Andy Seaborne. 2013. SPARQL 1.1 Query Language. W3C Recommendation. https://www.w3.org/TR/sparql11-query/

[54] Eric Prud'hommeaux and Andy Seaborne. 2008. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/

[55] Jeff Reback, jbrockmendel, Wes McKinney, Joris Van den Bossche, Tom Augspurger, Phillip Cloud, Simon Hawkins, Matthew Roeschke, gfyoung, Sinhrks, Adam Klein, Patrick Hoefler, Terji Petersen, Jeff Tratner, Chang She, William Ayd, Shahar Naveh, JHM Darbyshire, Marc Garcia, Richard Shadrach, Jeremy Schendel, Andy Hayden, Daniel Saxton, Marco Edward Gorelli, Fangchen Li, Matthew Zeitlin, Vytautas Jancauskas, Ali McMaster, Pietro Battiston, and Skipper Seabold. 2022. pandas-dev/pandas: Pandas 1.4.1. https://doi.org/10.5281/zenodo.6053272

[56] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?". In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Balaji Krishnapuram, Mohak Shah, Alex Smola, Charu Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, New York, NY, USA, 1135–1144. https://doi.org/10.1145/2939672.2939778

[57] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*. 130–136.

[58] Philipp D. Rohde, Julian Gercke, and Maria Esther Vidal. 2022. valSPARQL: Do not Ignore SHACL Constraints, Validate them during SPARQL Query Processing. under review.

[59] Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. 2021. Knowledge Graph Embedding for Link Prediction. *ACM Transactions on Knowledge Discovery from Data* 15, 2 (2021), 1–49. https://doi.org/10.1145/3424672

[60] Stuart J. Russell and Peter Norvig. 2016. *Artificial intelligence: A modern approach* (third edition, global edition ed.). Pearson, Boston and Columbus and Indianapolis. https://doi.org/Stuart

[61] Alieh Saeedi, Eric Peukert, and Erhard Rahm. 2018. Using Link Features for Entity Clustering in Knowledge Graphs. Springer, Cham, 576–592. https://doi.org/10.1007/978-3-319-93417-4{_}37

[62] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory - ICDT '10*, Luc Segoufin (Ed.). ACM Press, New York, New York, USA, 4–33. https://doi.org/10.1145/1804669.1804675

[63] Tudor Lapusan Terence Parr and Prince Grover. [n. d.]. dtreeviz: Decision Tree Visualization. https://github.com/parrt/dtreeviz.

[64] Guido van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[65] Theo van Veen. 2019. Wikidata. *Information Technology and Libraries* 38, 2 (2019), 72–81. https://doi.org/10.6017/ital.v38i2.10886

[66] María-Esther Vidal, Edna Ruckhaus, Tomas Lampo, Amadís Martínez, Javier Sierra, and Axel Polleres. 2010. Efficiently Joining Group Patterns in SPARQL Queries. In *The Semantic Web: Research and Applications*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache (Eds.). Lecture Notes in Computer Science, Vol. 6088. Springer Berlin Heidelberg, Berlin, Heidelberg, 228–242. https://doi.org/10.1007/978-3-642-13486-9{_}16

[67] Daniel Zhang, Saurabh Mishra, Erik Brynjolfsson, John Etchemendy, Deep Ganguli, Barbara Grosz, Terah Lyons, James Manyika, Juan Carlos Niebles, Michael Sellitto, Yoav Shoham, Jack Clark, and Raymond Perrault. [n. d.]. The AI Index 2021 Annual Report. https://doi.org/10.48550/arXiv.2103.06312