University of Padova

Department of Mathematics

Master Thesis in Big Data Management and Analytics

A General Framework of Automated User Intent Mapping to Complex Analytical Workflows

Local Supervisor
Massimiliano de Leoni
University of Padova

Co-supervisors
Sergi Nadal
Petar Jovanovic
Universitat Politècnica de Catalunya

Master Candidate
Zyad Abduljabbar Moqbel Al-Azazi

Academic Year

2023-2024

To my beloved parents, whose unwavering support and unconditional love have been my guiding light and the cornerstone of all my achievements.

To my wife, my steadfast companion, whose encouragement and resilience carried me through every challenge, never letting me falter in pursuit of my dreams.

To my brother, whose presence inspires me to strive to be the best version of myself.

To my extended family, whose prayers and support have been a source of strength during this journey.

To my dear friends, whose unwavering kindness and support, even through the simplest gestures, have made this milestone possible.

Abstract

The emergence of numerous AutoML (Automated Machine Learning) tools, such as Auto-sklearn and TPOT, as solutions to the challenges surrounding the utilization of Machine Learning and Data Science by non-technical users to solve different problems related to data has been driven by prior efforts dedicated towards the automatic creation of data pipelines. Furthermore, the creation of these ML-specialized pipelines is only the tip of the iceberg in terms of the challenges on the road to finding the optimal set of pipelines. Thus, existing solutions specialize in the process of selecting the most effective candidate pipelines (or more accurately limiting the search space for these pipelines), whether through the optimization of data preprocessors, the optimization of model choice, or the optimization of the hyperparameter tuning process. However, there are other challenges that are not addressed enough within these efforts. The first challenge is the generation of complex analytical workflows that satisfy the needs of users aside from their differences; the AutoML tools specialize in ML-focused tasks, such as classification and regression, without considering other data-centered tasks, such as descriptive analytics or data visualization. Additionally, the workflows generated from these tools are designed to run on specific execution engines regardless of the preferences of users and their limited scope of expertise.

This work focuses on generalizing the framework for generating engine-agnostic complex analytical workflows using Knowledge Graphs (KGs) as a method to automate the workflow creation process, as well as encode the metadata and real-world knowledge relevant to it. Thus, it addresses the previously mentioned challenges by developing a new generalized and extensible ontology that represents the entire process of generating analytical workflows from user intents. In addition, a more generalized workflow generation algorithm is adopted to ensure the generation of workflows that satisfy other user intents beyond classification, in this case data visualization. Finally, a rule-based optimization technique is incorporated within the workflow generation framework to enable the selection of data preprocessing operators instead of using all possible operator combinations.

The experimental setting consists of comparisons between the previous workflow generation proposal and the work proposed in this thesis, in addition to comparisons between the workflows selected by the rule-based selective generation and all the other possible workflows for each intent, separately.

Contents

Listing of figures

Listing of tables

Listing of acronyms

ML	Machine Learning
AutoML	Automated Machine Learning
SOTA	State-of-the-Art
RDF	Resource Description Framework
TBOX	Terminological Box
CBOX	Constrained Vocabulary Box
ABOX	Assertional Box
DM	Data Mining
KG	Knowledge Graphs
SHACL	Shapes Constraint Language

1 Introduction

1.1 Context and Topic

The importance of Data Science at the current times, given the vast amounts of data generated everyday, has led research efforts toward mobilizing data science into tools that people within different domains can harness to solve diverse data-related problems. A significant outcome of this research is the creation of AutoML, a set of tools that enable non-ML-savvy users of harnessing the power of ML methods to solve ML-related problems. The main advantage of using these tools is that non-technical users do not need to possess advanced knowledge levels to be able to utilize them. AutoML tasks can be divided into four main categories: data preparation, feature engineering, model selection, and model evaluation [2]. A great deal of the existing AutoML tools are specialized in ML-related tasks, namely classification and regression; however, ML-related problems are only a subset of the data-focused problems. Popular open-source AutoML tools, such as Autogluon[3], H2O AutoML[4] and TPOT[5], only take into account the extension of the set of algorithms solving certain ML problems without taking into account the expansion on the level of tasks solved. For example, Autogluon supports integration with Pytorch[*] to utilize its built-in classifiers to perform classification tasks; however, there is no focus on extending the tool to perform other ML tasks (such as clustering) or other data tasks (such as visualization). Additionally, the ML workflows generated by those tools are only

[*]https://pytorch.org/

executable in their specific engines, i.e. the workflow construction is tightly related to engine-specific execution. More importantly, none of the current SOTA tools supports the mapping (in some cases dissection) of an input user intent to a set of tasks. In this context, an intent is defined as the set of analytical tasks that a user wants to perform on a dataset, possibly with specific requirements related to the specific algorithms or the parameter values to be used. A user intent can be as simple as "perform a classification task on the penguins dataset" or as specific as "perform a classification task on the penguins dataset using the Decision-Trees algorithm with the learning rate set to 0.01."

The first intent example above may reflect a lack of technical knowledge from the user side; thus, it is very important for the intent-to-workflow mapping approach to incorporate the vast domain knowledge relevant to the set of potential data tasks to be solved, algorithms used, relevant parameters, as well as the principles of constructing the workflows qualified to solve those tasks. Generally, an intent could result in multiple possible workflows due to factors such as the number of algorithms capable of solving a task, the number of variations an algorithm may have, or the number of possible parameter values for an algorithm parameter. For example, a classification task could be solved by a Decision Tree algorithm or an SVM algorithm; moreover, an SVM algorithm has three kernel variations: hypertangent, polynomial, and RBF, with the latter having a set of possible values for the "gamma" parameter (0.1, 1, 10). Additionally, the workflow generation process should not be tightly coupled with a specific engine. Thus, the utilization of Knowledge Graphs for this problem is most suitable due to their ability to encode the needed domain knowledge, in addition to the advantages related to capturing the metadata relevant to the process and transformations inflicted on the data. The use of KGs to automatically compose data mining workflows has been explored in [6] and [7] leveraging KGs to create workflows to solve specific data use cases.

The ExtremeXP[†] project offers a data-driven approach that aims to provide users with relevant information to facilitate the decision making process, protecting users from the technical complexities surrounding the process of obtaining these insights. One mechanism is the mapping of user intents into complex analytical workflows. As can be seen in Fig.1.1, the process relies on a workflow generator backed by a knowledge base and guided by the intent of the user. The output of the whole process is tailored to the user on the basis of the user's profile, expressed preferences, and feedback.

The work of this thesis serves within the ExtremeXP project and is concerned with generalizing the framework of the intent-to-workflow mapping to cover additional data tasks beyond

[†]https://extremexp.eu/

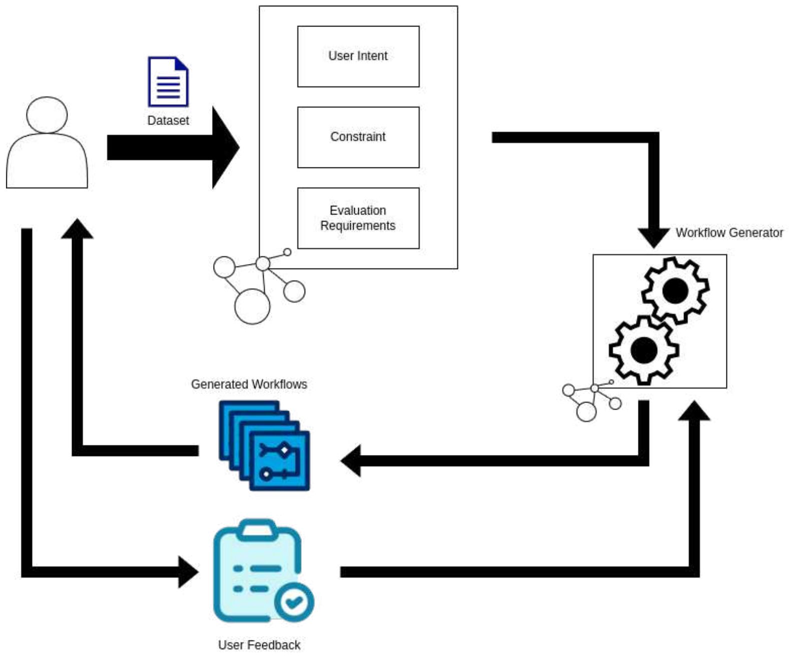


Figure 1.1: A diagram illustrating a simplified view of the overall ExtremeXP project

the tasks covered by existing AutoML tools, such as Data Visualization. The framework is powered by a Knowledge Graph ontology that represents concepts related to the generation process. Furthermore, the work also takes into account the extensibility of the framework by providing a set of guidelines allowing the inclusion of more tasks, while maintaining the engine-agnostic property of the workflow generation mechanism. This allows the generated workflows to be executed in any execution environment, as long as the proper translation method is available. Additionally, decreasing the number of possible workflows for each task by supporting the selection from operators performing the same data pre-processing tasks.

1.2 Objectives and Contributions

The main motivation behind this master thesis is to develop an optimized and generalized method that enables the successful mapping of analytical user intents to executable analytical workflows. The term "generalized" in this context is used to refer to the inclusion of different data analytics tasks besides ML classification tasks, such as data visualization and descriptive analytics. As for the optimization aspect of this approach, it is aimed towards the generation of a minimal number of workflows by limiting the number of logical plans generated from an abstract plan. Hence, the main objectives of this work are as follows:

- Creating a Knowledge Graph ontology that represents the semantics of the workflow generation process starting from a user intent resulting in intent-satisfying workflows.
- Developing a generalized, Knowledge-Graph-backed, engine-agnostic workflow generator that is capable of generating valid and functional analytical workflows from the input of a user intent.
- Optimizing the workflow generator to result in a minimal number of workflows for each task by selecting the appropriate operators for each preprocessing step.

These objectives essentially dictate the contributions this work brings, which are the following:

- The introduction of a generalized and extensible ontology that effectively represents the process of mapping user intents to valid workflows.
- The development of a generalized, multipurpose, engine-agnostic workflow generator. In addition to composing a set of guidelines to facilitate the successful extension of the workflow generator to integrate other analytical problems without changing the core generation algorithm.
- Incorporating a rule-based component selection technique within the general overall framework to enable the generation of a minimal number of workflows.

It should be clearly noted that an existing engine-specific workflow translator is used to ensure the successful conversion of generated workflows to engine-executable workflows.

1.3 Thesis Outline

The structure of this thesis is as follows. Chapter 2 provides an overview of the work directly related to the topics covered by this thesis; AutoML tools, Data Mining Ontologies and Workflow Optimization. Next, Chapter 3 presents the overall framework of the approach divided into steps that demonstrate the different levels of the presented work, in addition to a set of formal definitions of the core concepts within the work. After that comes Chapter 4 where the ontology that represents the whole process of capturing user intent and generating workflows satisfying the user intent is thoroughly explained, concluding with a comparison between the proposed ontology and other DM ontologies. Then, Chapter 5 provides a comprehensive explanation of the workflow generation process, a description of the generalization of the workflow generation process, and introduces the integrated rule-based selection mechanism. Next, Chapter 6 introduces a set of guidelines that will be used in the future to extend the framework to cover more user intents. As for Chapter 7, it delves deeper into the experimental aspect of the work by comparing the proposed work with a similar previous work in terms of behavior under the growth of factors directly affecting the complexity of the generation mechanism. Finally, Chapter 8 states the conclusion of the work, in addition to proposing potential future directions.

2
Related Work

The work developed and discussed in this thesis is focused on generalizing the framework of generating data analytical workflows using knowledge graphs. In this chapter, an overview of the work related to the problems addressed in this thesis is provided. In Section 2.1, the existing open-source AutoML tools are reviewed in terms of problems addressed along with their support for data preprocessing operations. In Section 2.2, the existing data mining ontologies are reviewed to understand the existing representations of the data and ML processes. Finally, in Section 2.3, existing workflow optimization techniques are explored to aid in the design of the optimization approach to be adopted by the framework.

2.1 AutoML Tools

Over the last few years, there has been an immense focus on AutoML tools as an important component that targets those with low to no technical expertise in machine learning to harness the power of ML to solve different problems. Open-source AutoML tools have taken care of the accessibility aspect, making this technology available to everyone regardless of their technical background. Nevertheless, users have found themselves overwhelmed with the number of AutoML solutions to choose from; thus, there was a need to be able to compare these solutions against each other. A challenge that [8], [9], [10] attempted to solve by benchmarking various tools using different approaches. In this section, some of the most prominent open source AutoML tools and frameworks mentioned in the aforementioned benchmarks will be discussed, specifically these tools covering the preliminary and necessary data management steps.

Among the most prominent of these tools is Autogluon-Tabular [3]; a tool that offers an "ensembling and stacking models" approach rather than the traditional model and hyperparameter selection approach. Despite the fact that Autogluon offers a two-phase preprocessing approach; model-agnostic and model-specific, some of these techniques offer quite naive solutions. For example, the model-agnostic approach includes an elimination of uncategorized features that consist of non-numeric and non-repeating fields, as well as the absence of an imputation technique for missing discrete variables; instead, a label of "Unknown" is assigned to missing discrete fields, and they are handled at test time, which could significantly affect the results desired by the user. There are also other tools that are built on existing platforms or libraries, such as H2O AutoML [4] built on the H2O.ai [*] engine along with TPOT [5] and auto-sklearn[11, 12] both built on the well-known scikit-learn library [†]. With auto-sklearn in both versions incorporating an inclusive set of data processors that includes one-hot encoding, missing data imputation, target class balancing, and rescaling inputs along with a more extensive set of 14 different feature processors that include feature selection and kernel approximation, among others[11]. On the other hand, [5] does not provide any kind of data pre-processing, but it incorporates many feature processors, including different scaling techniques, along with feature selection processors, such as Recursive Feature Elimination and Variance Threshold. As for [4], it also has the unique feature of handling categorical data natively by supporting group-splits on categorical variables, along with the availability of data processors and other feature and feature selection processors. Table 2.1 contains a summary of the preprocessing capabilities of the discussed tools.

Other tools usually associated with AutoML include FLAML[13] and HyperOpt[14]; however, these are not considered in this overview as they are not standalone tools. Both [13] and [14] are specialized in hyperparameter optimization and model selection (in the case of [13]); thus, they are usually incorporated into bigger AutoML frameworks, so that other tasks can be handled by other tools.

It is important to note that all the previously mentioned tools generate engine-specific workflows that are run on specific engines, which is a problem that [7] attempted to solve by adopting an engine-agnostic workflow generation paradigm. Although this approach works well, introduces a novel approach in encoding input and output constraints within the KG-backed generator, and incorporates the necessary data preprocessing steps, it is focused mainly on gen-

[*]https://docs.h2o.ai/

[†]https://scikit-learn.org/stable/

erating classification workflows.

2.2 Data Mining Ontologies

One of the various reasons behind the existence of many ontologies attempting to represent the processes composing data mining is the purpose driving the creation of these ontologies. For instance, OntoDM[15] was motivated by the need for a general and formalized framework for the rapidly developing data mining field. At a later time but with a similar motivation, ML-schema[16] was then introduced as an attempt to create a unified representation that aims to facilitate the representation and communication of the ML experiments together with their associated datasets. Evidently, a close-up inspection of these ontologies will guide us to understand the characteristics of generalized ontologies of data mining. For example, in OntoDM, the main concepts introduced are: Dataset, Data Mining Task, Algorithm, Generalization (representing outputs of data mining algorithms), and Constraint. On the other hand, [16] also introduces equivalent classes, but with more focus on ML-centric data mining activities. This can be clearly seen in how ML-schema has one type of output, Model, in comparison with OntoDM which has the concept Generalization covering different types of output, such as models, patterns, clusters and probability distributions.

The automatic creation of Data Mining workflows using ontologies has also gained significant attention. With DMWF[17] being one of the earlier attempts to represent data mining workflows in a way that users and planners comprehend while also depicting intermediary stages of workflow planning. DMWF introduced concepts such as MetaData, Goal, Task, and Method to clearly distinguish between different levels of workflow planning. It also introduced operator conditions to ensure the construction of semantically correct workflows. As for DMOP[18], the objective was to utilize semantic meta-mining to provide support to decision-making steps that define the result of a data mining process; it defined the concept of DM-Hypothesis, which can be DM-Model or DM-PatternSet resulting from a DM-Task. DMOP contains thorough descriptions of tasks, algorithms, data, hypotheses, and workflows. Similarly, BIGOWL[6] was introduced to support knowledge management in Big Data Analytics while enabling the creation of workflows that satisfy users' requirements. BIGOWL does not differ much from DMOP in terms of the concepts and relations defined, as well as each class having its own set of conditions and allowing instances to be members of the class (corresponding to the description in [18]). In fact, BIGOWL uses the equivalence relation with DM-Algorithm in DMOP to define its DataMiningAlgorithm subclass [6].

AutoML Tool	Available Preprocessing	Additional Notes
AutoGluon	• Categorizing the datatypes of the columns. • Model-agnostic preprocessing: uncategorized data, non-numeric, non-repeating fields are discarded. • Model-specific preprocessing: missing data is only labeled as "Unknown"; unseen data is handled at test time.	Two different tools available for tabular data and time series data.
Auto-sklearn (1 & 2)	• Preprocessing is comprised of data preprocessing and feature preprocessing. • Data preprocessing is minimal and only includes: one-hot encoding, imputation, balancing and rescaling.	Feature preprocessing is more extensive.
H2O AutoML	• General preprocessing: data imputation, data normalization and one-hot encoding. • Specific preprocessing for tree-based models: group-splits on categorical variables.	Relies on the H2O library.
TPOT	Incorporates only feature scaling and feature processors.	

Table 2.1: Comparison of some AutoML tools preprocessing capabilities

2.3 Workflow Optimization

The term "Workflow Optimization" is a very broad term that entails several problems on several distinct levels. The word workflow in this context means data-centric workflows; workflows that focus on the manipulation and transformation of data through many steps into other forms of data or data products.

The problem of workflow optimization can be tackled from different aspects; one of the most researched and well-developed aspects borrows directly some of its principles from database management systems, specifically workflow optimization on the execution engine level. Workflow generation tools such as Helix [19] and KeystoneML [20] are good examples. Helix, for example, optimizes the time needed to generate intermediate workflow results by materializing common intermediate results and intelligently reusing them in other workflows instead of executing all the workflows from scratch. For example, intermediate data results coming from a min-max normalization step or a mean data imputation step will be the same across all workflows regardless of the final algorithm feeding on the data. Meanwhile, KeystoneML, a workflow generator backed by Apache Spark [‡] and specialized in developing end-to-end ML pipelines, incorporates two different levels of optimization: an operator-level optimization and a pipeline-level optimization. The operator level optimization is related to the choice of an exact physical plan to perform a certain step (for example, a normalization step could be done using z-score scaling, min-max scaling, ...etc); the optimizer on this level is cost-based and is also further divided into two levels: operator-specific and cluster-specific. The costs considered for this level are related to the computation and communication costs given certain parameters, such as the size of the input data and the number of workers. On the other hand, the optimization on the level of the whole pipeline targets the composition of a "pipeline profile" that dictates the materialization of certain intermediate results, similarly to Helix.

However, such low-level optimizations may not be within the scope of the interest of this work since the approach under development requires techniques that are compliant with the generalized, engine-agnostic paradigm that the proposed approach strives to achieve. Thus, the optimization techniques concerned at this stage are relevant to minimizing the number of generated workflows, which translates to the optimization on the operator level; operator-level optimization that is less dependent on cost optimization and more reliant on general rules for operator selection.

There has been a great deal of focus on automating the data pre-processing steps required for ML workflows. The efforts of automating the data preprocessing steps include not only the definition of the steps needed, but it also extends to the automatic selection of operators to accomplish these steps. For example, Auto-Prep[21] is one of the attempts focused on automating the following data preprocessing steps for classification and regression workflows: missing data imputation, categorical features encoding, feature reduction, feature scaling, in addition to automatic detection of duplicate rows and feature data types. The approach relies

[‡]https://spark.apache.org/

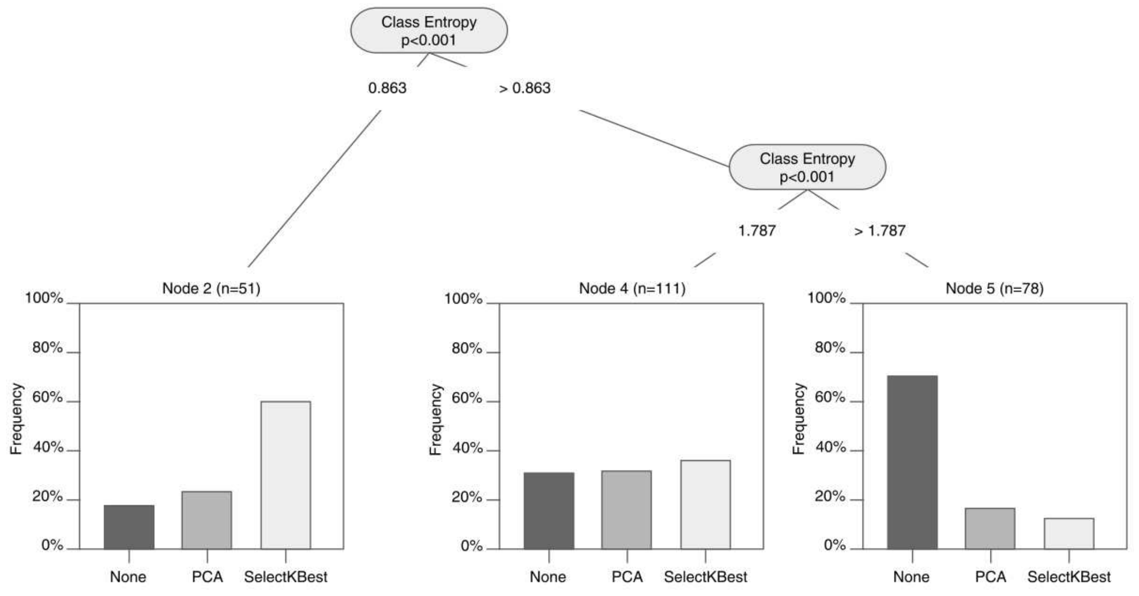


Figure 2.1: A diagram illustrating how decision tree model learned the relation between the entropy of a feature and the appropriate feature engineering operators in [1]

on predefined rules that are taken from an extensive review of the literature. Examples of these rules include how automatic data imputation operators are selected depending on how the behavior of the missing data is categorized: MAR (missing at random), MNAR (missing not at random), or MCAR (missing completely at random); however, despite the work determining appropriate imputation techniques for each case, it has been acknowledged that MNAR is a difficult case to discover and is better handled manually as it requires human intervention. Despite Auto-Prep claiming to achieve good results, the tool does not seem to be open for public access.

Nevertheless, a more modern approach that has been gaining momentum due to its ability to learn hidden patterns between different preprocessing steps/ operators and data characteristics is meta-learning. PERSISTANT[22] is an assistant for non-experts that utilizes the meta-learning technique to help select the best data preprocessing operators based on their effect on the final result of the workflow. PERSISTANT is specialized in classification workflows that employ the following algorithms: Decision Trees, Naive Bayes, PART, Logistic Regression and Nearest Neighbor. While the experimental results of PERSISTANT did not include a set of general operator-specific set of rules, it resulted in enforcing some heuristics such as concluding that Decision Trees and Logistic Regression are not affected by any data scaling operations, as well as some other WEKA[§]-specific heuristics related to the Nearest Neighbor implementation in WEKA embedding a scaling step; hence, not being affected by any scaling operators. However, a more recent work in [1] utilized meta-learning to compose a set of rules that dictate the order of pre-processing steps in classification workflows, in addition to employing decision tree models in meat-learning to determine the relations between the characteristics of the data set and the operators for data rebalancing and feature engineering (as illustrated in Fig.2.1).

[§]https://www.weka.io/

3
General Overview

This chapter is divided into two sections where Section 3.1 provides an overview of the approach presented in this work, while Section 3.2 formally defines important concepts that will be seen throughout the work.

3.1 Approach Overview

As stated previously, one of the main purposes of this work is to generalize the framework of mapping user intents to complex analytical workflows; thus, it is important to understand the approach of this work and distinguish it from the earlier solutions proposed.

Before looking at the approach, it is crucial to understand the design of the ontology adopted by this work, specifically the layers of the ontology. The traditional structure of an ontology is comprised of two layers: the Terminological Box (TBOX) and the Assertional Box (ABOX). The TBOX represents the schema that hosts the main concepts of the domain, the relationships between the concepts, and the properties. On the other hand, the ABOX hosts the instances belonging to the classes and having the relationships and properties defined in the TBOX. In certain cases, there is a need for more layers that host concepts that cannot be members of the TBOX or the ABOX. This is when the addition of the Constrained Vocabulary Box (CBOX) is essential to represent the various taxonomies or categories. The importance of adding this layer is related to improving the governance of the data and avoiding complexity on the TBOX level, especially when taking scale into consideration. Figure 3.1 shows how the scale could greatly

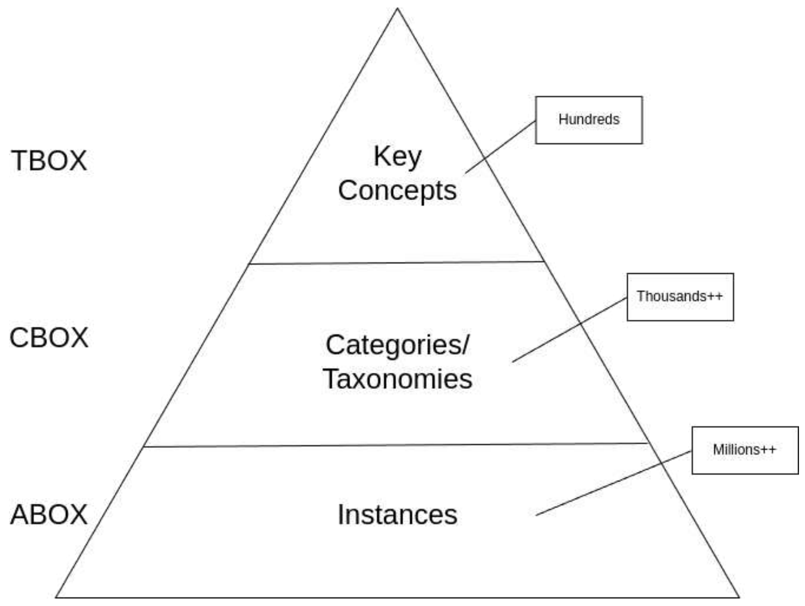


Figure 3.1: Scale across ontology layers. Diagram inspired from a presentation by the Gist Council (https://www.youtube.com/watch?v=0-j9nWFVoYc)

vary across layers in ontology-backed systems. The concepts hosted by the CBOX are simply categories (subclasses) of the concepts in the TBOX and more descriptive of the instances in the ABOX. For example, if the TBOX had the concept Algorithm, the CBOX would have different types of algorithms, such as Linear Regression, SVM, KNN and so on. The ABOX will contain multiple instances of each of these models. Thus, this approach allows us to introduce more precise concepts without making the TBOX complex and prone to exploding.

The overall view of the approach is shown in Figure 3.2. The first step of this approach is the creation of a new, improved, and generalized ontology. The main purpose of the ontology is to represent the entire process of generating complex analytical workflows, starting from the capture of the user intents and ending with RDF-formatted workflows that fulfill the user intent. Moreover, the ontology not only depicts the generation process, but also captures the characteristics of all the inputs and outputs resulting from the different steps of the workflow, along with reflecting any changes to the data on its annotation. Furthermore, the new ontology takes into account the incorporation of concepts related to user preferences and constraints introduced by the ontology in [23] that serves within the same scope of the ExtremeXP project. Although the new ontology borrows some of its design principles from some of the existing ontologies, such as DMOP and the ontology in [7], its novelty lies in its general approach to creating data mining workflows that extend beyond ML workflows, while maintaining the distinction between engine-agnostic and engine-specific levels. In this work, the case of workflows that solve the problem of data visualization is achieved as a proof-of-concept. Additionally, the new ontology introduces a method to encode selection rules to aid in preprocessing component selection, as explained later in Section 5.3.

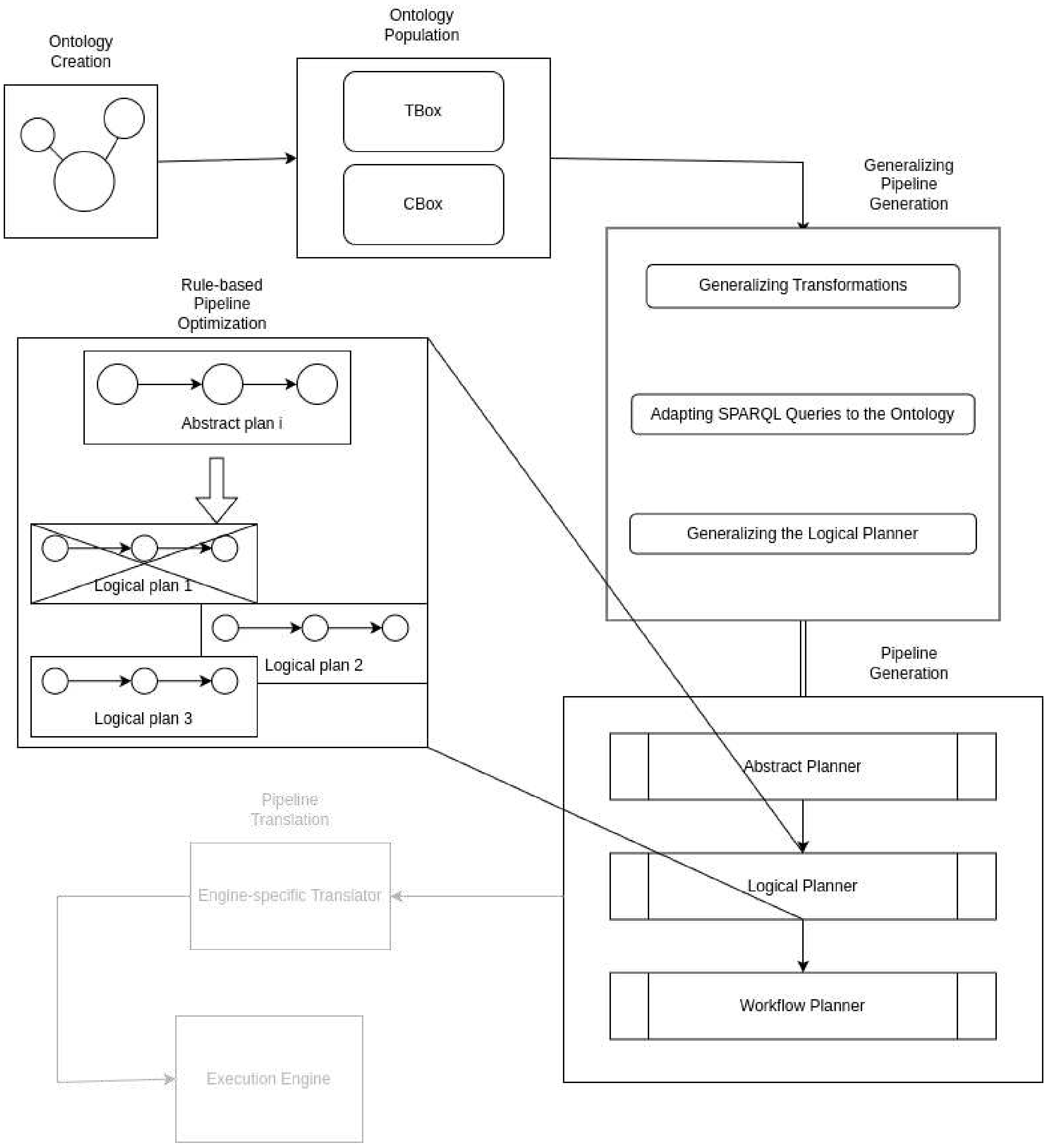


Figure 3.2: A diagram illustrating the overall approach of the proposed work

The second step is related to the population of the TBOX and the CBOX of the ontology.

For the TBOX population, it is populated by creating the triplets of the ontology, along with the corresponding data properties using the rdflib[*] in Python. Similarly, for the CBOX, it is populated using the same method to encode taxonomies that were natively created or directly borrowed from the DMOP ontology with the additional use of SHACL[†] shapes to represent constraints related to implementations' specifications of inputs and outputs and the tabular data characteristics for the selection rules. More details on population techniques are discussed in Chapter 4 and some other in Chapter 6 as part of the extension guidelines. The next step after the creation of the ontology and the population of its TBOX and CBOX was to adapt the entire generation algorithm to the newly created ontology. The first step was to change the SPARQL queries to become consistent with the new ontology. Then, some operators were generalized as part of the transformation generalization step, namely the data partitioning operators. Finally, the last step of generalization was to modify the logical planner to create workflows that not only serve ML workflows reliant on data partitioning, but also more workflows achieving solutions to other data mining problems.

After the generalization of the workflow generator, the generated workflows are tested to determine their validity and functionality; validity is determined by visually inspecting the RDF-formatted workflows to check whether there are mistakes in the construction of the steps of the workflows, while the functionality of the workflows is determined after applying engine-specific translation to multiple generated workflows and running them on the KNIME execution engine without any failure.

Since one of the main assumptions of the project is that the system users are possessive of little to no technical knowledge, some technical decisions related to the Logical planner need to be taken for the purpose of reducing the search space, in addition to the important factor that users may not be capable of making such decisions. Thus, a rule-based selection technique is incorporated into the Logical Planner as shown in Figure 3.2. This technique aims at mainly decreasing the number of possible data pre-processing operators to be used when constructing the possible logical workflows, instead of the brute-force method. Components are provided with a set of rules in the form of SHACL shapes that reflect the data characteristics the dataset should possess for a component to be deemed the best choice for the dataset. The set of preferences or rules are provided during the creation of the components based on general rules. Although the definition of these rules is outside the scope of this work, there is plenty of work that focuses on this specific area, some of which utilizes meta-learning to compose these rules,

[*]https://rdflib.readthedocs.io/en/stable/

[†]https://www.w3.org/TR/shacl/

as can be seen in [22] and [1]. It should be noted that not all steps are invoked every time the framework is used; the ontology population needs to be invoked whenever the framework is extended (more on extension guidelines in Section6.1). As for the generalization of the pipeline generation, this was in the scope of the work and is not invoked. Finally, pipeline generation is the part that is invoked the most by the user (more details on this in Chapter5).

3.2 Formal Definitions

The process of mapping a user intent to an analytical workflow is an extensive process with some important concepts to note as part of intent representation and generalization of the whole process. Hence, the following concepts and their formal definitions are presented in this section.

Task

A term that is used to describe an analytical task to be performed. This definition includes ML tasks such as classification, regression, and clustering, as well as other tasks related to data transformation, such as data processing and data visualization. The symbol T is used to represent the set of all tasks, with each task represented as t. In some cases, t implicitly includes a set of subtasks ST_t. For example, the task of data processing may include feature scaling, feature transformation, and data imputation.

Algorithm

A method that can be used to achieve a specific task. Algorithms such as SVM and Decision Trees are used to perform classification tasks. The set of all algorithms is denoted as A, accordingly:

$$\{a_t : a_t \in A_t \mid a_t \text{ solves } t, \text{ where } t \in T\}$$

where A_t is the set of algorithms that can solve t and A represents the set of all algorithms.

Data

A term that describes the set of all data instances generated and utilized by workflows. The set of all data instances used in a workflow is denoted as D, where d_0 is the input dataset provided by the user. D includes tabular data annotations along with models, visualizations, and any other types of inputs and outputs for an algorithm.

Implementation

The executable form for an algorithm a where M is the set of all implementations and m_a is the implementation implemening a. It is possible for an algorithm to have multiple implementations at the same time.

Parameter

A single parameter is a term used to refer to a certain factor configuration within an algorithm implementation, which can be related to the algorithm itself or the execution engine. A parameter is associated with a single value or a set of possible values. The set of all parameters is denoted by Par, while the set of parameters for a specific implementation is denoted by Par_m and a specific parameter of a specific implementation is par_m.

Component

A more abstract form of an implementation, where an implementation has at least one component. The set of all components is denoted by C. The components of the same implementation C_m have the same functionality with semantic differences caused by each component overriding a set of implementation parameters OP_c. A single overridden parameter op_c is a tuple of (par_m, v), where v is a parameter value. A component also exposes some of the other parameters that require user input EP_c where $EP_c \subseteq Par_i \setminus OP_c$.

Intent

A user intent I is defined by a tuple in multiple forms (d_0, t) or (d_0, t, a) or (d_0, t, a, EP_c) where d_0 is the input dataset, t is the task to complete, a is the algorithm's specific choice and EP_c is a set of tuples (ep_c, v) where each tuple contains an exposed parameter and the value chosen by the user.

Transformation Components

A transformation component tc_i is a component capable of transforming a specific input into a specific output and can be represented in a tuple $(t_a, \mathcal{I}_m, \mathcal{O}_m)$ where t_a is the task to be performed by tc_i, $\mathcal{I}_m$ is the set of input specifications and $\mathcal{O}_m$ is the set of output specifications that the transformation component will fulfill. Thus, given $d_{\text{in}}, d_{\text{out}} \in D$, it is necessary for each d^j_{in} to conform to $\mathcal{I}^j_i$, where $1 \leq j \leq |\mathcal{I}_i|$ and for each d^k_{out} to conform to $\mathcal{O}^k_i$, where $1 \leq k \leq |\mathcal{O}_i|$.

Rule

A rule r is defined by a tuple (tc_i, t, dc) where tc_i is a transformation component, t is the main task included within the user intent I and dc is a specific data characteristic that d_0 must meet for tc_i to be favored over other transformation components, where $\in \mathrm{TC}_i \setminus tc_i$

Abstract Plan

The set of abstract plans AP is represented by a tuple (I, A_{ap}), where I is the user intent motivating the creation of abstract plans, and A_{ap} is the set of algorithms to be applied in response to I. $A_{\text{ap}} = A_t$ if I defines an algorithm a; otherwise, $A_{\text{ap}} = A_t$ if I_ap defines only the task t.

Logical Plan

The set of logical plans for an abstract plan LP_{ap} is represented by a tuple (ap, TC_{ap}, α), where ap is the abstract plan and TC_{ap} is the set of transformation components needed to construct the logical plans and α is a function that is capable of selecting the most appropriate transformation components to be applied to the data set d_0 based on its characteristics.

Workflow Plan

For each logical plan, a workflow plan wp is created and represented by a tuple $(lp, D_{\text{lp}}, Par_{\text{lp}})$, where lp is the logical plan and D_{lp} is the set of all data instances utilized and produced by the components in lp and Par_{lp} is the set of component parameters that each s^j_{wp} will use and $0 \leq j < |S_{\text{WP}}|$, where S_{WP} is the set of steps to construct lp.

4 Ontology

In this chapter, the proposed ontology will be discussed in detail. The chapter is organized as follows: Section 4.1 will delve into the TBOX of the ontology, Section 4.2 will tackle the two levels of the taxonomies in the ontology's CBOX, Section 4.3 will be about the ontology's ABOX and finally in Section 4.4 the proposed ontology will be compared and aligned with other existing ontologies discussed in Section 2.2.

4.1 TBOX

Figure 4.1 shows the created ontology schema, which in addition to fulfilling the contributions of this work, also constitutes an effort to align the ontologies proposed in [7] and [23] as they all serve within the ExtremeXP project. For the purpose of this work, the main classes of this ontology that will be discussed in detail along with their object properties are the following: Intent, Data, Workflow, Task, Algorithm, Implementation, Component, Parameter, Parameter Specification, Transformation, Data Specification, Data Tag and Rule. It should be noted that there are more concepts in the ontology; however, they are outside the scope of this work.

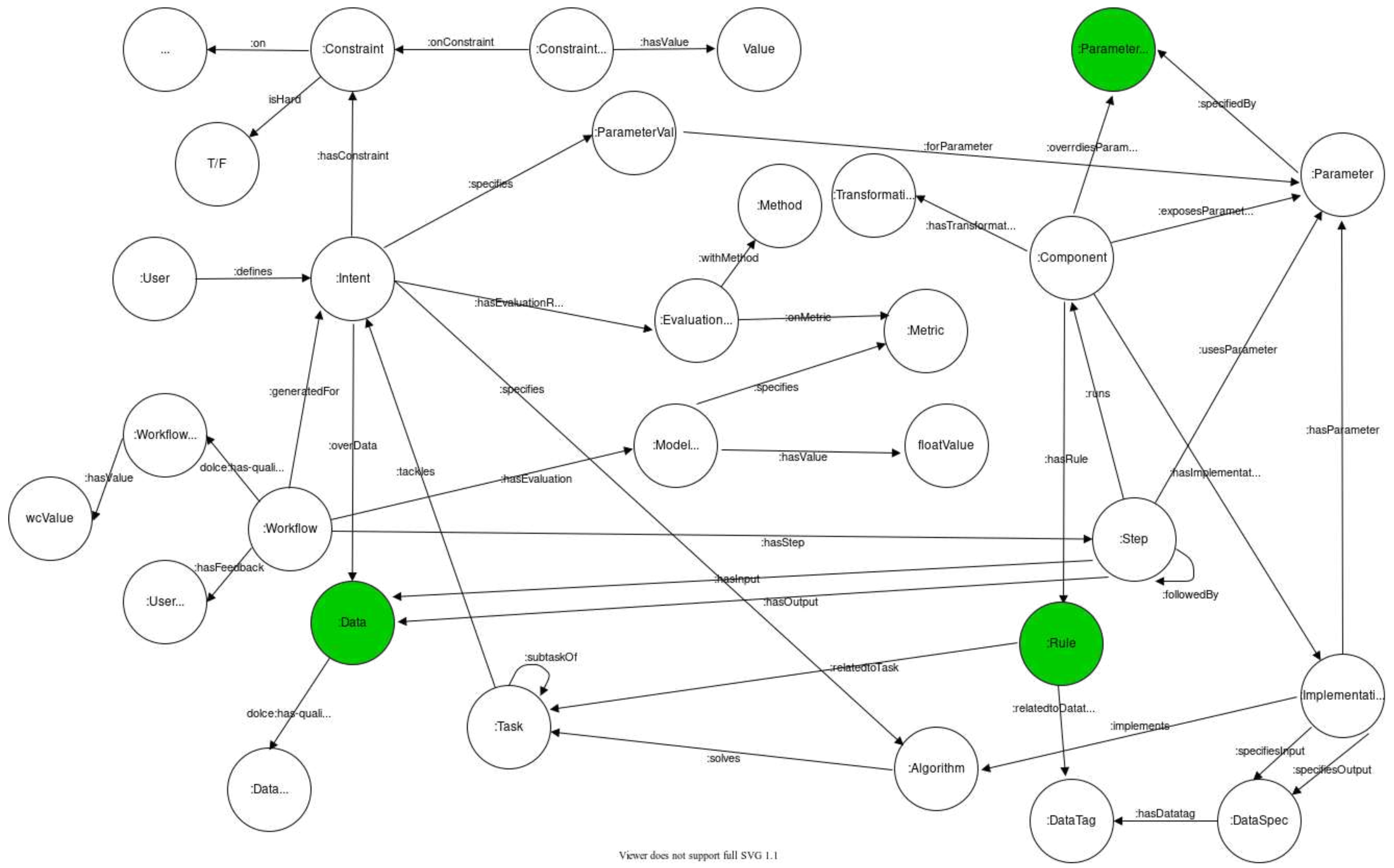


Figure 4.1: A diagram illustrating the proposed ontology with the concepts in green highlighting the novelty of the ontology in comparison to previous ontologies.

The proposed ontology introduces a novel concept, tb:Rule, used to encode general domain knowledge that assists in operator selection (details in Section 5.3). Moreover, the proposed ontology represents all the data instances using the concept tb:Data (including the input dataset), which none of the previous ontologies does, as this ontology distinguishes among the data instances directly using their annotations. The concept tb:ParameterSpecification, although inspired from other ontologies, is utilized to imply components overriding parameters as well as the workflow planning phase. Additionally, the concept tb:Parameter is connected to the concepts: tb:Implementation, tb:Component, tb:Step representing the different relations between a parameter and these concepts across the different workflow building stages.

Intent

This concept represents the intention of the system user, i.e. task a user wants to perform, the algorithm specified (if any), the values specified for certain parameters (if any) and any other requirements. Additionally it includes dataset the user inputs. The object properties *overData*, *specifies*, and *specifiesValue* connect the user intent to the dataset provided by the user, the algorithm specified and the parameter value specified, respectively (if any).

Object Properties	**DL**
overData	$\exists$ overData.Thing $\sqsubseteq$ Intent $\sqsubseteq$ $\forall$ overData.Data
specifies	$\exists$ specifies.Thing $\sqsubseteq$ Intent $\sqsubseteq$ $\forall$ specifies.Algorithm
specifiesValue	$\exists$ specifiesValue.Thing $\sqsubseteq$ Intent $\sqsubseteq$ $\forall$ specifiesValue.ParameterVal

Table 4.1: Object Properties of Intent as a Domain

Data

The concept of Data in this ontology, unlike other ontologies, is a more generic term; it does not only represent the dataset provided by the user, but it is also extended to include any inputs or outputs that are generated by the different steps within a workflow. In other words, Data includes the initial dataset provided by the user, the intermediary tabular datasets resulting from the different transformative steps of a workflow, and any other forms of data, including models, visualizations, and so on. This definition requires data characteristics to be incorporated to be able to distinguish between possible inputs and outputs. It should be noted that the subclass of TabularDataset of Data uses the definition of the DMOP[18] ontology and its defined data properites. The object property *has-quality* is directly taken from the DMOP[18] ontology, which in turn was incorporated from the DOLCE[24] ontology.

Object Properties	DL
has-quality	$\exists$ has-quality.Thing $\sqsubseteq$ Data $\sqsubseteq$ $\forall$ has-quality.DataCharacteristic

Table 4.2: Object Properties of Data as a Domain

Task

This concept represents the task or set of tasks inferred from the intent of the user; the concept Task includes all possible data mining processes that can be performed on the data. A task can consist of one simple task (such as Data Visualization or Data Clustering) or a task that could include other tasks, such as Descriptive Analysis including Data Cleaning and Data Visualization. The object property *subtaskOf* connects the concept Task to itself to represent the relation between a set of subtasks achieving a bigger task, while the object property *tackles* connects a task to the user intent it fulfills.

Object Properties	DL
subtaskOf	$\exists$ subtaskOf.Thing $\sqsubseteq$ Task $\sqsubseteq$ $\forall$ subtaskOf.Task
tackles	$\exists$ tackles.Thing $\sqsubseteq$ Intent $\sqsubseteq$ $\forall$ tackles.Task

Table 4.3: Object Properties of Task as a Domain

Algorithm

This concept represents all solutions suitable for solving a certain task. For example, the KNN algorithm is used to solve a clustering task, while a line plot is also an algorithm to solve a visualization task, and so on. *solves* is object property that connects an algorithm to the task it solves.

Object Properties	DL
solves	$\exists$ solves.Thing $\sqsubseteq$ Algorithm $\sqsubseteq$ $\forall$ solves.Task

Table 4.4: Object Properties of Algorithm as a Domain

Implementation

A concept that represents the executable code of an algorithm detailing all of its technical specifications. An implementation is connected to its corresponding algorithm through the object property *implements*. An implementation is also connected to its input and output specifications (data specification) through the object properties *specifiesInput* and *specifiesOutput*. Additionally, it is also connected to its parameters through the *hasParameter* object property.

Object Properties	**DL**
implements	$\exists$ implements.Thing $\sqsubseteq$ Implementation $\sqsubseteq$ $\forall$ implements.Algorithm
specifiesInput	$\exists$ specifiesInput.Thing $\sqsubseteq$ Implementation $\sqsubseteq$ $\forall$ specifiesInput.DataSepc
specifiesOutput	$\exists$ specifiesOutput.Thing $\sqsubseteq$ Implementation $\sqsubseteq$ $\forall$ specifiesInput.DataSepc
hasParameter	$\exists$ hasParameter.Thing $\sqsubseteq$ Implementation $\sqsubseteq$ $\forall$ hasParameter.Parameter

Table 4.5: Object Properties of Implementation as a Domain

DataSpec

This concept is used to represent all the possible specific types of input and output produced by the different implementations. As explained above, Data refers to the data instances resulting from the components, such as: tabular dataset, model, and visualization. DataSpec is used as a concept that covers all of the subclasses of the main Data subclasses. For example, there are many subclasses to Model – Decision Tree model, KNN model, normalizer model, and so on. DataSpec is only connected to DataTag using *hasDatatag*.

Object Properties	**DL**
hasDatatag	$\exists$ hasDatatag.Thing $\sqsubseteq$ DataSpec $\sqsubseteq$ $\forall$ hasDatatag.DataTag

Table 4.6: Object Properties of DataSpec as a Domain

DataTag

This concept is used to represent tags on DataSpec instances based on their characteristics or the change in characteristics due to various workflow steps. They are expressed as SHACL shapes. For example, a splitting component would take as an input a tabular dataset and will output two datasets with labels: TrainTabularDataset and TestTabularDataset, or in the case of normalization, the resulting dataset would have the tag NormalizedTabularDataset.

Component

Represents an abstraction on top of the implementation, where each implementation is connected to at least one component. A component usually overrides the value of one or more parameters from the implementation parameters, which allows an implementation to have multiple components performing the same task but resulting in different results due to different semantics. For instance, an implementation for a bar chart visualization may have three different components: a sum bar chart, an average bar chart, and a count bar chart. All these components share the same implementation parameters along with the same input and output specifications; nevertheless, each component performs the aggregation differently based on the aggregation type parameter, producing different visualizations. A component is linked to its corresponding implementation through the *hasImplementation* object property. A component is connected to the parameters it overrides through its parameter specification using the property *overridesParameter* and it is also connected to its exposed parameters with the property *exposesParameter*. A component also uses the property *hasTransformation* to connect to its corresponding transformation (if it exists). A component is also connected to a rule for the purpose of selecting the best component of the available components for a certain preprocessing task through *hasRule*.

Object Properties	**DL**
hasImplementation	$\exists$ implements.Thing $\sqsubseteq$ Implementation $\sqsubseteq \forall$ implements.Component
overridesParameter	$\exists$ implements.Thing $\sqsubseteq$ Component $\sqsubseteq \forall$ implements.ParameterSpecification
exposesParameter	$\exists$ exposesParameter.Thing $\sqsubseteq$ Component $\sqsubseteq \forall$ exposesParameter.Parameter
hasTransformation	$\exists$ hasTransformation.Thing $\sqsubseteq$ Component $\sqsubseteq \forall$ hasTransformation.Transformation
hasRule	$\exists$ hasRule.Thing $\sqsubseteq$ Component $\sqsubseteq \forall$ hasPreference.Rule

Table 4.7: Object Properties of Component as a Domain

Rule

Represents the heuristic rules used to select the best component for a pre-processing task based on the input dataset characteristics and the main task of the workflow. A rule is connected to a DataTag through the object property *relatedtoDatatag*, as well as a task through the object property *relatedtoTask*.

Transformation

A concept that is used to represent the transformations incurred by a component. Since compo-

Object Properties	**DL**
relatedtoDatatag	$\exists$ relatedtoDatatag.Thing $\sqsubseteq$ Rule $\sqsubseteq \forall$ relatedtoDatatag.DataTag
relatedtoTask	$\exists$ relatedtoTask.Thing $\sqsubseteq$ Rule $\sqsubseteq \forall$ relatedtoTask.Task

Table 4.8: Object Properties of Rule as a Domain

nents (not all) perform operations on the input dataset that result in a transformed dataset as an output, these changes need to be recorded and reflected on the characteristics of the data annotation propagating between the steps. These transformations, expressed in terms of SPARQL queries, are of three main types:

- Loader Transformation: this transformation is specific to the data loader at the beginning of any workflow where the data annotations of the original dataset are outputted from the loader component.
- Copy Transformation: a transformation that indicates the direct copy of the contents of the input directly into the output with minimal to no changes.
- SPARQL Transformation: a customized transformation that is defined depending on the functionality of the component; it mainly consists of INSERT and DELETE statements.

Parameter

A concept that represents the configuration parameters of an implementation; implementation parameters include algorithm parameters and engine-specific parameters. An example of an algorithm parameter would be the name of the categorical column to plot in a bar chart algorithm; whereas, the inclusion of N/A values is an implementation-specific parameter (a parameter in KNIME, in this specific case).

Object Properties	**DL**
specifiedBy	$\exists$ specifiedBy.Thing $\sqsubseteq$ Parameter $\sqsubseteq \forall$ specifiedBy.ParameterSpecification

Table 4.9: Object Properties of Parameter as a Domain

ParameterSpecification

It represents the assignment of a parameter to a value. Initially, not all the parameters of an implementation are assigned to values (except for the overridden parameters of a component). The concept is similar to an existing concept in the [16] ontology; however, it was adapted to the workflow generation case of this work. This is explained in detail in chapter 5.

Step

A concept that represents the building blocks of a workflow. It is connected to the following steps within the same workflow through *followedBy*. It is also connected to the input data and output data of each step using *hasInput* and *hasOutput*. It is also connected to the component it runs and all the parameters used through *runs* and *usesParameter*, respectively.

Object Properties	**DL**
followedBy	$\exists$ followedBy.Thing $\sqsubseteq$ Step $\sqsubseteq$ $\forall$ followedBy.Step
hasInput	$\exists$ hasInput.Thing $\sqsubseteq$ Step $\sqsubseteq$ $\forall$ hasInput.Data
hasOutput	$\exists$ hasOutput.Thing $\sqsubseteq$ Step $\sqsubseteq$ $\forall$ hasOutput.Data
runs	$\exists$ runs.Thing $\sqsubseteq$ Step $\sqsubseteq$ $\forall$ runs.Component
usesParameter	$\exists$ usesParameter.Thing $\sqsubseteq$ Step $\sqsubseteq$ $\forall$ usesParameter.Parameter

Table 4.10: Object Properties of Step as a Domain

Workflow

A concept that represents the process in which a user's intent is accomplished. It is connected to its corresponding intent using *generatedFor*. It is also connected to the steps forming it using the property *hasStep*, as well as, to its characteristics, user feedback and workflow evaluation through *has-quality*, *hasFeedback*, and *hasEvaluation*.

4.2 CBOX

Since this ontology was created for the purpose of workflow generation, a set of taxonomies that complement the existing terminology in the TBOX with human-level knowledge and constraints is needed. In this section, an explanation of the necessary concepts in this ontology level is given along with the corresponding population strategies. Moreover, with the important goal of extensibility and maintainability of this ontology, the vocabulary will be divided within two classes at this level: engine-agnostic concepts and engine-specific concepts. Understanding the details of this section is quite crucial for the upcoming Section 6.1

Object Properties	DL
generatedFor	$\exists$ generatedFor.Thing $\sqsubseteq$ Workflow $\sqsubseteq \forall$ generatedFor.Intent
hasStep	$\exists$ hasStep.Thing $\sqsubseteq$ Workflow $\sqsubseteq \forall$ hasStep.Step
hasFeedback	$\exists$ hasFeedback.Thing $\sqsubseteq$ Workflow $\sqsubseteq \forall$ hasFeedback.UserFeedback
hasEvaluation	$\exists$ hasEvaluation.Thing $\sqsubseteq$ Workflow $\sqsubseteq \forall$ hasEvaluation.ModelEvaluation
has-quality	$\exists$ has-quality.Thing $\sqsubseteq$ Workflow $\sqsubseteq \forall$ has-quality.WorkflowCharacteristics

Table 4.11: Object Properties of Workflow as a Domain

4.2.1 Engine-agnostic CBOX

This class includes all the taxonomies related to the process of generating workflows and the domain knowledge necessary to complete the process of workflow creation independently of the technical details related to the execution engine. The concepts included in this class are: tb:Task, tb:Algorithm, tb:Data, tb:DataSpec, tb:DataTag and tb:Rule. Any extensions in this class includes the addition of new tasks, algorithms, different types of data, or data tags; there is no change as it is very rare for the requirements of an algorithm, for example, to change.

Task

Tasks are added to the CBOX taking into account existing hierarchies in the categories of data tasks. In this ontology, the dmop:DM-Task hierarchy is borrowed. The choice of this hierarchy is due to its comprehensive categorization of data tasks. As shown in Figure 4.2, a task is divided into an induction task and data processing and each has its own subclasses until the leaves are reached where a task such as Classification would be categorized under Predictive Modeling Task.

Algorithm

Figure 4.3 shows the tb:Algorithm hierarchy, which is also very similar to tb:Task in terms of the hierarchical representation of its instances; hence, the DMOP hierarchy of dmop:DM-Algorithm is also borrowed for classification and other tasks. However, a similar structure was adapted to algorithms for other tasks (such as visualization).

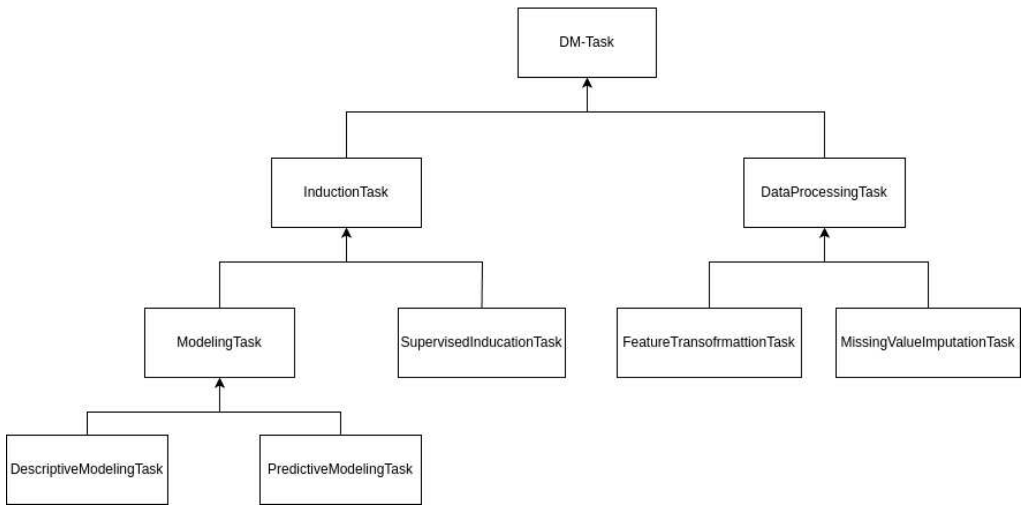


Figure 4.2: A diagram illustrating the DM-Task hierarchy in the DMOP ontology with the arrows indicating a subclassOf relationship

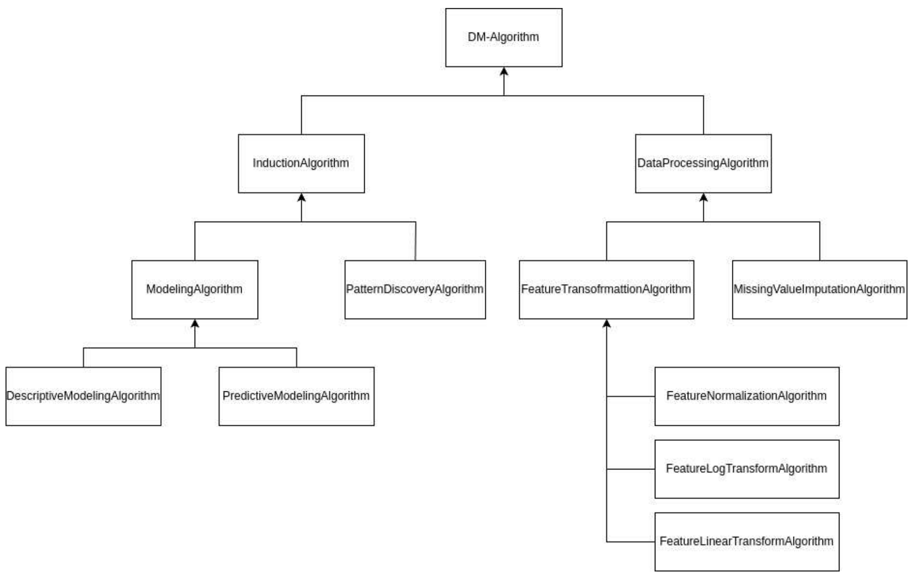

DM-Algorithm
InductionAlgorithm
DataProcessingAlgorithm
ModelingAlgorithm
PatternDiscoveryAlgorithm
FeatureTransofrmattionAlgorithm
MissingValueImputationAlgorithm
DescriptiveModelingAlgorithm
PredictiveModelingAlgorithm
FeatureNormalizationAlgorithm
FeatureLogTransformAlgorithm
FeatureLinearTransformAlgorithm

Figure 4.3: A diagram illustrating the DM-Algorithm hierarchy in the DMOP ontology with the arrows indicating a subclassOf relationship

Data

As explained in Section 4.1, tb:Data includes all possible abstract types of inputs and outputs of different steps. Thus, these types are manually added in this case after anticipating the possible outcomes of the different algorithms added previously. In this case, tb:Data has three differ-

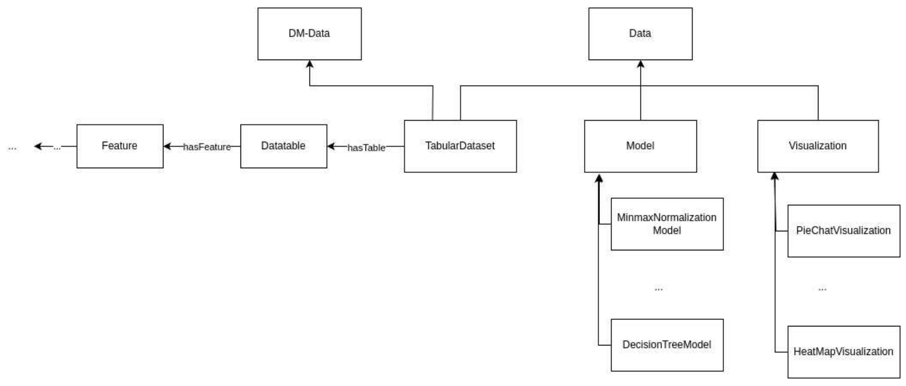


Figure 4.4: A diagram illustrating the Data classs in the ontology and how it is related to the hierarchy in the DMOP ontology with the empty arrows indicating a subclassOf relationship

ent subclasses: dmop:TabularDataset, cb:Model and cb:Visualization. Within each of these subclasses, there are further specific subclasses, as illustrated in Figure 4.4.

DataSpec

tb:DataSpec builds on the tb:Data specific leaves defined; a SHACL shape is created for each of the leaves shown in Figure 4.4.

DataTag

The instances of tb:DataTag were created manually as SHACL shapes manually. The tb:DataTag instances are not expected to grow largely as other previous concepts since they are limited by the data characteristics of the original dataset and the changes implied by the different transformative steps within the workflow.

Rule

The rule instances are associated with tasks, components and, data tags. Thus, the heuristic instructions they represent do not change across the different engines as they are easily transferrable but they rely on the aforementioned concepts, so they are not likely to change but they are very likely to increase as those specific concepts increase.

4.2.2 Engine-specific CBOX

This class includes vocabularies with definitions that are closely related to the execution engine and its configuration. This class includes: tb:Implementation, tb:Component, tb:Parameter and tb:Transformation. While the extension in this class directly builds on the concepts from the previous class, it is possible to have two implementations for the same algorithm for two different engines; thus, the creation of instances for this class of concepts is directly related to the execution requirements. Additionally, unlike previous class concepts, changes are likely to occur in response to changes in engine requirements or upgrades.

Although the previous iteration of this work [7] proposed an automated approach to generate the code for instances of this class for the KNIME engine (specifically, tb:Implementation and tb:Parameter instances), it also cited the need for manual revisions. Moreover, the proposed automated approach was able to create conflicts within the ontology due to some of the generated parameters having the same parameter keys and functionalities in different implementations. Hence, the instances for tb:Implementation, tb:Parameter and tb:Component were inserted manually to avoid any potential issues or conflicts. As for tb:Transformation instances, they were created manually since they rely on the components definitions.

4.3 ABOX

For the last level of the ontology, it consists of instances of the following classes: tb:Workflow, tb:Step, tb:ParameterSpec, and tb:Data. All instances belonging to these concepts are generated mainly during the workflow generation process with few exceptions. For tb:ParameterSpec instances, few of them are manually created by specifying the overridden parameters when manually creating tb:Component instances, while the majority of them are created during the workflow generation process.

Additionally, the main exception is the first tb:Data instance – the original dataset provided by the user; the dataset is imported as an RDF-formatted annotation. This annotation captures the characteristics of the dataset as a whole (number of columns, number of entries, encoding, and so on), along with detailed characteristics of each column (name of the column, data type of the column, whether a column is unique, or categorical, and so on). Other data instances are generated during the pipeline generation process either by transforming the original dataset and other transformed iterations of it, or by generating instances of other types as outputs from learner (producing model instances) or visualizer (producing visualization instances) implementations. It should be noted that while the development of the data annotator is outside the scope of this work, there was a need to slightly tweak the existing data annotator to correspond to the specifications of the KNIME execution engine. For instance, KNIME's definition of categorical columns does not include numerical columns, i.e. any numerical column is not considered categorical. On the other hand, the data annotator provides a more comprehensive definition of categorical columns that include numerical columns.

4.4 Comparison with Other Ontologies

As part of this work, it is very important to understand where the proposed ontology stands with respect to other ontologies that serve similar purposes. As discussed in section 2.2, there are many ontologies targeting the representation of machine learning or, more broadly, data mining processes whether for the purposes of standardizing and representation or automatic workflow generation. In table 4.12, it can be observed that the proposed ontology is more comprehensive in terms of concept coverage due to the nature of the ontology's purpose of generating workflows from user intents taking into consideration different variables, such as user-specified constraints and user feedback.

Additionally, in the proposed ontology, there are several levels of abstraction that are not covered by other ontologies. The first evidence on that is the separation among the concepts: tb: Algorithm, tb:Implementation, and tb:Component, with only BIGOWL[6] adopting a similar approach. However, none of the ontologies encapsulates the inputs and outputs of different algorithms in one concept with other ontologies, except for DMWF[17], with other ontologies opting to separate the main dataset from the products of the different algorithms. Moreover, the novelty of the proposed ontology is also apparent with the introduction of the concept tb:Rule used to encode some of the domain knowledge serving the main purpose of the ontology.

While the main purpose of Table 4.12 is to compare the proposed ontology to other existing ontologies, it can be utilized in the future as a reference not only to extend the ontology, but also to enrich it using existing taxonomies from the addressed ontologies.

Proposed Ontology	**DMOP**	**BIGOWL**	**ML-Schema**	**OntoDM**	**DMWF**
Intent	N/A	N/A	N/A	N/A	N/A
Task	DM-Task	N/A	Task	Data Mining Task	Goal
Algorithm	DM-Algorithm	Algorithm	Algorithm	Data Mining Algorithm	N/A
Implementation	DM-Operator	Component	Implementation	Data Mining Algorithm	Task
Component	N/A	Task	N/A	Data Mining Component	Method
Parameter	Parameter	Parameter	HyperParameter	Parameter	MetaData
Parameter Specification	OpParameterSetting	N/A	HyperParameterSetting	Parameter Setting	N/A
Data	DM-Data, DM-Hypothesis	N/A	Data, Model	Dataset, Generalization	IOObject
Data Characteristics	DataCharacteristic	N/A	DataCharacteristic	Data specification	MetaData
DataSpec	IO-Class	Data	N/A	N/A	N/A
DataTag	N/A	N/A	N/A	N/A	N/A
Workflow	DM-Workflow	Workflow	N/A	N/A	Workflow
Workflow Characteristics	N/A	N/A	N/A	N/A	N/A
Step	N/A	N/A	N/A	N/A	N/A
Rule	N/A	N/A	N/A	N/A	N/A
Model Evaluation	ModelPerformance	N/A	ModelEvaluation	Generalization evaluation	N/A
Requirement	N/A	N/A	N/A	N/A	N/A
Constraint	N/A	N/A	N/A	Constraint	N/A
User Feedback	N/A	N/A	N/A	N/A	N/A

Table 4.12: Mapping between the ontology terms between the proposed ontology and the different ML/DM ontologies and vocabularies

5
Workflow Generation

This chapter is focused on explaining all the details relevant to the workflow generation process, including the steps taken to generalize the workflow generation process, along with the intuition and implementation of selective workflow generation. Section 5.1 delves into the details of the workflow generation process given a user intent and producing a set of valid workflow plans. Section 5.2 tackles the steps taken to generalize certain aspects of the workflow generator and other necessary elements. Section 5.3 explains the basis of the selective workflow generator; rule-based preprocessing component selection based on the meta-characteristics of the dataset.

5.1 Overview of Workflow Generation Process

The first step in the workflow generation process is the ingestion of a single user intent (represented as a node of type tb:Intent); a user intent defines (at least) the main pillars of the generation process. The requirements needed for each intent depend on the type of task the user desires. Figure 5.1 shows some of the different possible intent forms. As the diagram shows, the validity of the intent depends on the task needed. For example, in the intent graph for the classification intent, the intent's requirements include defining the task and the dataset, namely the classification task and the Titanic dataset. Meanwhile, in the graph capturing the visualization intent, the definition of an exact visualization algorithm along with the categorical column to be visualized is needed due to the nature of the task. Moreover, each visualization algorithm may require a different set of values for a different set of parameters. Table 5.1 shows the differ-

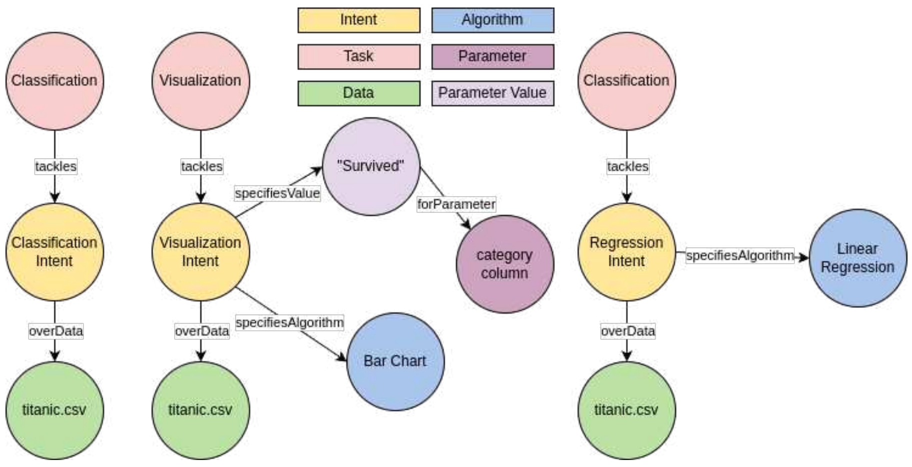


Figure 5.1: Examples illustrating different forms of user intent.

ent parameters needed for each of the visualization algorithms and whether they are optional or not. Thus, different intents dictate different intent graphs. On the other hand, a regression intent, similar to a classification intent, may not require the specification of an algorithm; nevertheless, a user intent could capture certain user preferences in terms of algorithms or even parameter values that go beyond the minimum validity requirements.

Visualization Algorithm	**Parameter**	**Optional**
Pie Chart	Categorical Column	No
	Frequency Column	Yes
Bar Chart	Categorical Column	No
	Frequency Columns	Yes
Histogram	Numerical Column	No
	Frequency Columns	Yes
Heatmap	Y-Axis Column	No
	X-Axis Columns	No
Scatter Plot	X-Axis Column	No
	Y-Axis Column	No
Line Plot	X-Axis Column	No
	Y-Axis Columns	No

Table 5.1: Different parameter requirements for the visualization algorithms

After capturing the user intent, the next step would be to generate a set of quite simple plans that illustrate the main steps needed to fulfill the intent of the user; abstract plans are very simplistic plans that demonstrate the steps needed to solve the task expressed by the user in their

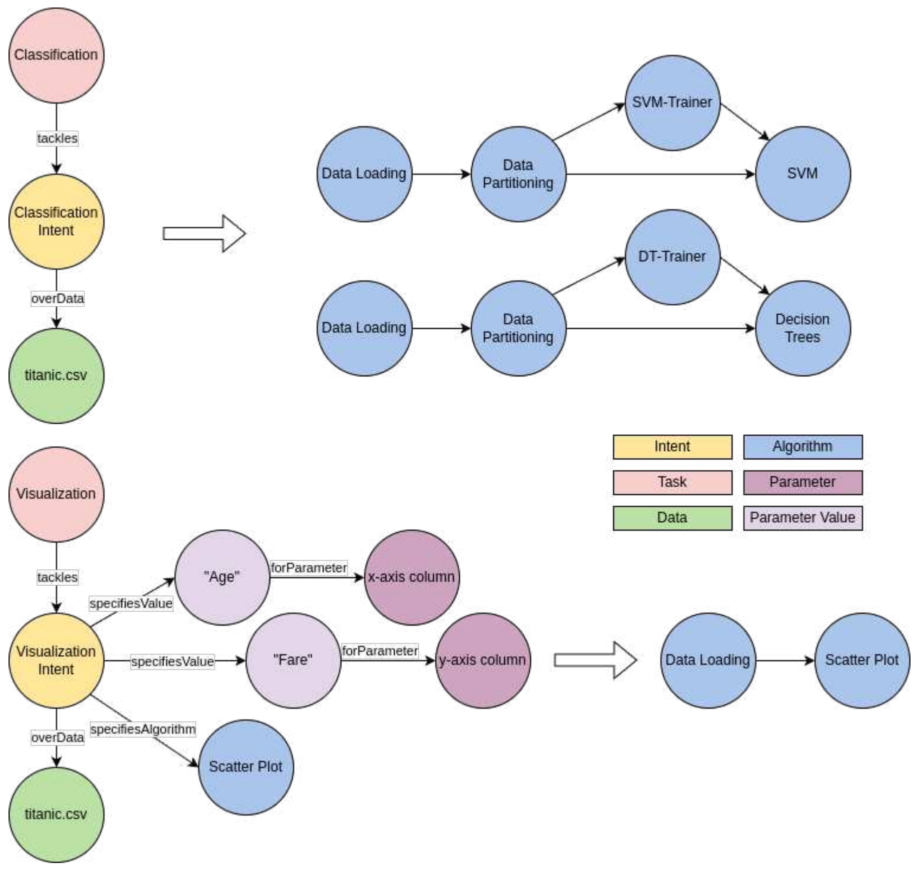


Figure 5.2: Abstract plans generated from the user intent.

intent. Naturally, the first step would be to load the data and then perform the necessary steps to solve the problem defined by the user. In Figure 5.2, different forms of intents are shown to affect the number and forms of possible abstract plans. For example, in the graph illustrating the classification intent where no specific algorithms are defined to solve the task, the set of abstract plans would demonstrate the necessary steps of loading the dataset, partitioning the dataset (as part of the requirements for the classification algorithms) and then training the classification model and, consequently, applying the model to the test dataset with different abstract plans utilizing different algorithms capable of performing the task. However, in the case of specifying an algorithm, the set of generated abstract plans is more precise; additionally, in the case of visualization tasks, it can be noticed how the process is straightforward with no need for intermediary steps due to the lack of any specific requirements by the scatter plot visualization algorithm.

Since an abstract plan represents a very broad engine-agnostic conceptualization of the task

Algorithm 5.1 Generalized Logical Planner Algorithm

```
1: a: Main algorithm solving the task
2: d: Dataset of the intent
3: procedure LogicalPlanner($a, d$)
4:     $\mathcal{C} \leftarrow \{c \mid \forall c = \langle a_c, I_c, O_c \rangle \in \mathcal{C} : a_c = a\}$    ▷ Operations applying algorithm
5:     $W \leftarrow \{\}$
6:     for each $c_i$ in $\mathcal{C}$
7:         $R \leftarrow \{r_i \mid \forall p \in I_{c_i} : \neg\rho(p)\}$    ▷ Unsatisfied operation preconditions, including partitioning
8:         $T \leftarrow \{\}$    ▷ $T$ is a matrix
9:         for each $r_i$ in $R$
10:            $t_i \leftarrow \{c \mid \forall c = \langle a_c, I_c, O_c \rangle \in \mathcal{C} : r_i \in O_c\}$    ▷ Operations related to $r_i$
11:            insert $t_i$ into $T$
12:        end for
13:        $V \leftarrow$ cross product of $T$    ▷ Generate every possible combination of operations
14:        for each $v_i$ in $V$
15:            $w_i \leftarrow$ BuildPlan($v_i, c_i$)
16:            append $w_i$ into $W$
17:        end for
18:    end for
19:    return $W$
20: end procedure
```

to be achieved by the workflow, a more extended, engine-specific plan is aimed for at this stage. The main idea in this phase is to boil down the abstraction of tb:Algorithm nodes in each of the abstract plans into the corresponding executable tb:Component instances to obtain what can be described as logical plans. Moreover, logical plans take into account the metadata of the input dataset in addition to the preconditions of each tb:Algorithm node specified in the abstract plan. Those preconditions are encoded as input requirements in the tb:Implementation instances corresponding to the tb:Algorithm. The preconditions represent certain dataset characteristics that should be present for the algorithm to be applied on the dataset.

The procedure of generating a set of logical plans from an abstract plan is captured in the pseudocode provided in Algorithm 5.1. Given a, the main algorithm solving an analytical task in an abstract plan (SVM, Decision Tree, and Scatter plot in the abstract plans illustrated in our examples in Figure 5.2), and d, the dataset provided by the user as part of their intent (the Titanic dataset in our example), a set of logical plans is generated for an abstract plan through the following steps:

1. For each of the components corresponding to the algorithms in the abstract plan, a list of input preconditions (SHACL shapes labeled as tb:DataTag nodes in the ontology discussed in Section 4.1) is obtained (lines 4 to 7).

2. For each of the input preconditions that are not satisfied by the dataset, a list of components capable of transforming the dataset so that it satisfies those preconditions is obtained (lines 9 to 12).

3. Then, using the different lists of transformative components for each of the unsatisfied preconditions, a set of logical plans is built using all the possible combinations of transformative components (lines 13 to 16).

In Figure 5.3, examples of the possible logical plans generated from the logical plans can be observed. It can be deduced from the logical plans that in the SVM classification case, the data needed to be partitioned, normalized as well as not containing any null values; whereas, the preconditions in the case of scatter plot are not as strict. As explained, each logical plan captures a different combination from all possible transformative combinations. Some of the differences to be highlighted are the number of components required to apply each transformation; in the case of simple workflows (scatter-plot workflow), one component is needed per workflow. However, when partitioning the data (classification workflow), the number of components per workflow is two, as it corresponds to the number of partitions. Another important aspect to note is the number of components corresponding to an algorithm as the relation between these two concepts is not one-to-one as explained previously, which can be clearly seen in the case of SVM posessing three different components.

The final step in generating the analytical pipelines is the creation of workflow plans. For each logical plan created exactly one workflow plan is created by applying the following steps for each component (the pseudocode for the algorithm is shown in Algorithm 5.2):

1. A node of type tb:Step is created and connected to the corresponding component using the tb:runs relation.

2. The specifications of inputs and outputs for each component are identified by retrieving them from the corresponding tb:Implementation node; in the case of simple workflows, if there is an output from the previous step that satisfies the conditions, it is plugged in as an input (the case for second step taking as an input the imputed dataset and not the imputation model). For workflows incorporating data partitioning, the output from

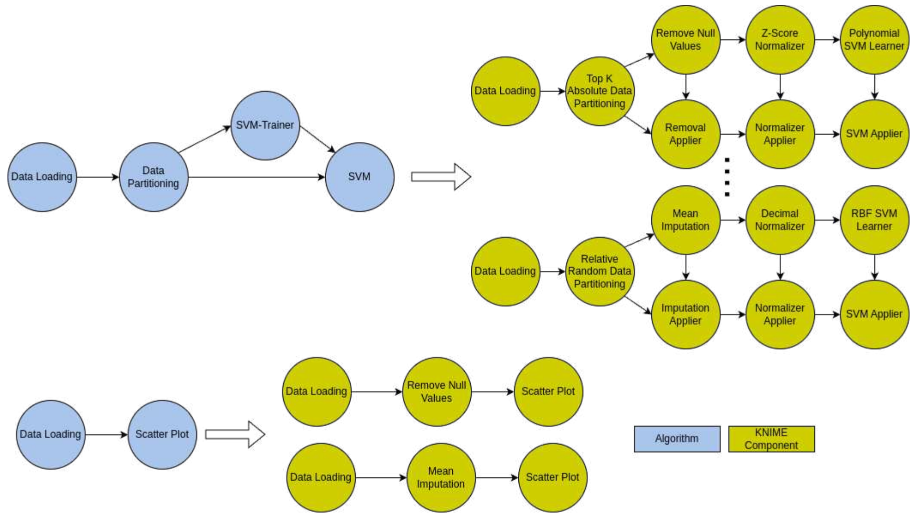


Figure 5.3: Logical plans expanded from abstract plans.

previous steps is assigned as input for the following steps based on the preconditions of each step.

3. Assigning values for all component parameters (other than the component overridden parameters) where the exposed parameters are assigned values expressed in the intent; for other parameters or if the intent does not include values for the exposed parameters, they are assigned to the predefined default values. In the case of parameters that require user input, manual review will be needed. The process of assigning values for parameters is represented by assigning a tb:ParameterSpecification node to the parameter through a tb:specifiedBy connection.

4. As for the component overridden parameters, the corresponding tb:ParameterSpecification nodes are retrieved and are then joined with the tb:ParameterSpecification nodes from the previous step. All the tb:Parameter nodes connected to tb:ParameterSpecification nodes are then linked to the tb:Step node running the tb:Component node.

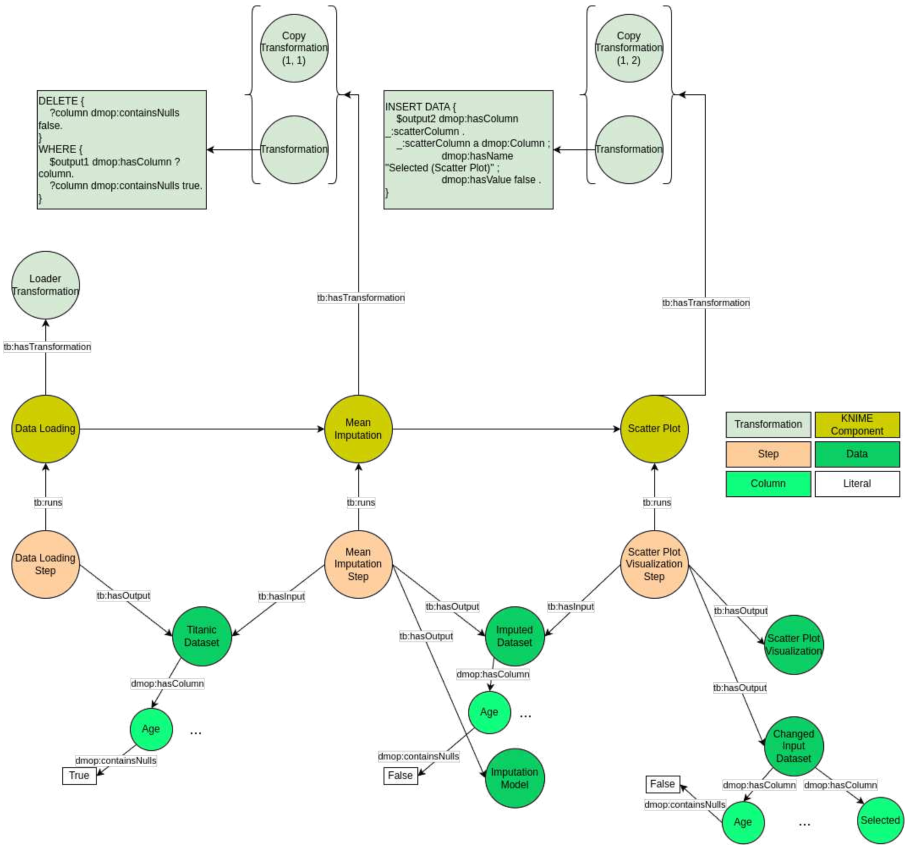
5. Finally, the transformation queries are executed to reflect the changes in the annotation of the dataset and propagate the changed annotation.

In Figure 5.4, the logical plan for visualizing the dataset is subjected to the previous steps to produce the final workflow plan. There is a one-to-one relation between the components of the logical plan and the steps of the workflow plan. As can be seen, each of the components in

Algorithm 5.2 Workflow Plan Builder

```
1: V: combination of operations to be applied
2: c: main component in the abstract plan
3: procedure BuildPlan($V, c$)
4:     $dataset\_node \leftarrow original\_dataset$
5:     $loader\_step$                                ▷ Initialize Data Loader step
6:     for each $v_i$ in $[V, c]$
7:         $test\_comp = get\_test\_component(v_i)$  ▷ Get the test component for each of the components
8:         if $test\_comp$ = None
9:             create_step()     ▷ Create a step for the component and update the workflow
10:            if component_output = $trainData, testData$     ▷ If the component outputs split tabular data
11:                $train\_node \leftarrow trainData$
12:                $test\_node \leftarrow testData$
13:            end if
14:        else
15:            create_train_step()     ▷ Create a step for the training component with $train\_node$ as an input
16:            if $test\_node != None$
17:                create_test_step()     ▷ Create a step for the testing component with $test\_node$ as an input
18:            end if
19:        end if
20:    end for
21:    return $W$
22: end procedure
```

this case has at least one transformation (not always the case) with the data loading component having a very special transformation specialized in providing the original dataset annotation. As for the other components each contain a copy transformation that directly copies the input with the specified index and outputs it to a specified index as well. For example, the imputation step (second step) takes the dataset as an input and outputs it, as it is, as the first output (of two outputs); however, the main reason for the copy transformations in these components is to facilitate reflecting the changes on the dataset annotation instead of deleting the annotation and creating it again. The third type of transformation seen is how the actual changes to the annotation are reflected with one example changing the values of the column properties (second transformation in the second step) and the other example adding a new column to the


```
DELETE {
    ?column dmop:containsNulls
false.
}
WHERE {
    $output1 dmop:hasColumn ?
column.
    ?column dmop:containsNulls true.
}
```

```
INSERT DATA {
    $output2 dmop:hasColumn
_:scatterColumn .
    _:scatterColumn a dmop:Column ;
                   dmop:hasName
"Selected (Scatter Plot)" ;
                   dmop:hasValue false .
}
```

Figure 5.4: Workflow plan created from logical plan.

existing dataset annotation through a number of DELETE and INSERT statements.

In Figure 5.5, a clear presentation of the different relations tb:Parameter nodes have across the different stages. Initially in the CBOX, the tb:Parameter nodes are connected to the corresponding tb:Implementation node using the relation tb:hasParameter; at this stage, there are no differences between the nodes. However, the first layer of distinction results from the abstraction level tb:Component represents with respect to tb:Implementation with some specific parameters being overridden by a component (essentially the main difference between the different components of a single implementation). The indirect connection between the component and a parameter through a tb:ParameterSpecification node is how an overridden parameter is represented, while the exposed parameter is directly connected to the component. There are also other tb:Parameter nodes that are exposed by a component; they remain unassigned to specific values waiting for a user input. After all the parameters of a component are assigned to tb:ParameterSpec nodes during the workflow planning stage, all of the parameters, including the parameters inherited from the implementation other than the exposed and overridden ones, will be connected to the corresponding tb:Step node, as can be seen in the lower diagram of the figure.

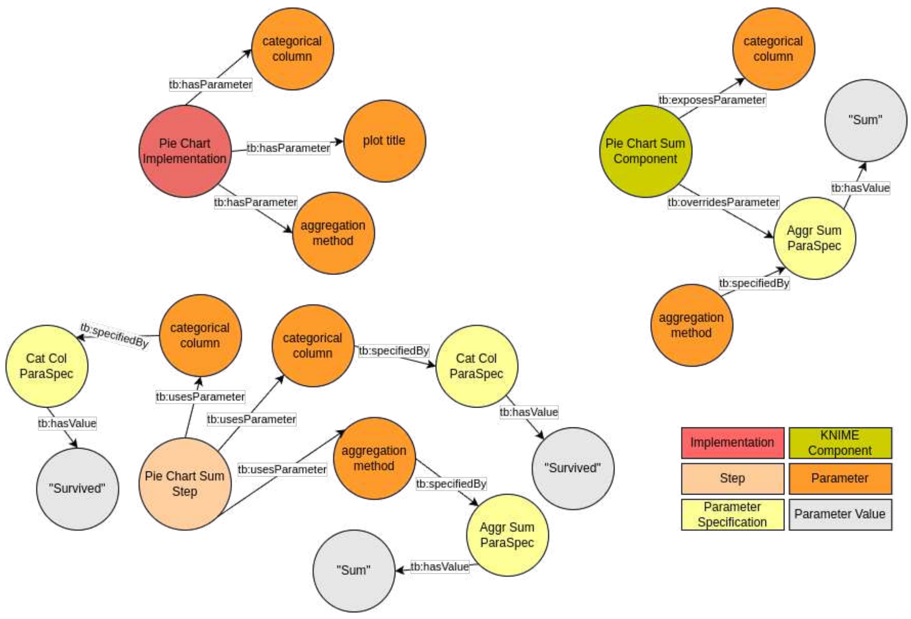


Figure 5.5: Parameter Different Relations in Across the Stages.

5.2 Generalizing the Workflow Generation Process

As mentioned in Section 2.2, the proposal in [7] is the only work that tackled the workflow generation process from an engine-agnostic perspective; the workflow generation process is independent of the execution engine that runs the final workflows. However, as can be seen from the pseudocode presented in Algorithm 5.3, the logical planner differentiated between the construction of train-test workflow plans and simple workflow plans based on a Boolean variable (line 2). Moreover, the inclusion of data partitioning components was not included within the general framework of data transformation components, rather treated as an exception (as can be seen in lines 13 to 14).

The inclusion of data partitioning components as part of data partitioning components is a very important step in the generalization of the generation process, as it allows for a better utilization of the knowledge base at hand. Thus, the first step is to distinguish between the data partitioning components outputs. The partitioning components take as an input a single

Algorithm 5.3 Previous Logical Planner Introduced in [7]

```
1: : a: main algorithm used in abstract plan
2: : d: Dataset to be used in the workflow
3: procedure ExpandPlan($a, d$)
4:     $train \leftarrow$ whether $a$ requires training
5:     $\mathcal{C} \leftarrow \{c \mid \forall c = \langle a_c, I_c, O_c, \tau_c \rangle \in \mathcal{C} : a_c = a\}$    ▷ Operations applying algorithm
6:     $W \leftarrow \{\}$
7:     for each $c_i$ in $\mathcal{C}$
8:         $R \leftarrow \{r_i \mid \forall p \in I_{c_i} : \neg\rho(p)\}$    ▷ Unsatisfied operation preconditions
9:         $T \leftarrow \{\}$    ▷ $T$ is a matrix
10:        for each $r_i$ in $R$
11:            $t_i \leftarrow \{c \mid \forall c = \langle a_c, I_c, O_c, \tau_c \rangle \in \mathcal{C} : r_i \in O_c\}$    ▷ Operations related to $r_i$
12:            insert $t_i$ into $T$
13:        end for
14:        if $train$
15:            $p \leftarrow \{c \mid \forall c = \langle a_c, I_c, O_c, \tau_c \rangle \in \mathcal{C} : a_c = Partitioning\}$    ▷ Partitioning Operations
16:            insert $p$ into $T$
17:        end if
18:        $V \leftarrow$ cross product of $T$    ▷ Generate every possible combination of operations
19:        for each $v_i$ in $V$
20:            if $train$
21:                $w_i \leftarrow$ BuildTrainTestPlan($v_i, c_i$)
22:            else
23:                $w_i \leftarrow$ BuildPlan($v_i, c_i$)
24:            end if
25:            append $w_i$ into $W$
26:        end for
27:    end for
28:    return $W$
29: end procedure
```

tabular dataset and output two datasets with different dimensions depending on the specific parameters of the partitioning component and the component itself. In the Python code shown in Figure 5.6, the code represents an example of a data partitioning component possessing a set of transformations (expressesd in SPARQL) with the last one adding the dmop:isTrainDataset and dmop:isTestDataset to each of the resulting data partitions annotations to distinguish between the data partitions.

```
    random_relative_train_test_split_component = Component(
    name='Random Relative Train-Test Split',
         .....
    transformations=[
        CopyTransformation(1, 1),
        CopyTransformation(1, 2),
        Transformation(
            query='''
DELETE {
    $output1 dmop:numberOfRows ?rows1.
    $output2 dmop:numberOfRows ?rows1.
}
INSERT {
    $output1 dmop:numberOfRows ?newRows1 .
    $output1 dmop:isTrainDataset True . # a property for the training dataset
    $output2 dmop:numberOfRows ?newRows2 .
    $output2 dmop:isTestDataset True . # a property for the testing dataset
}
WHERE {
    $output1 dmop:numberOfRows ?rows1.
    BIND(ROUND(?rows1 * (1 - $parameter3)) AS ?newRows1)
    BIND(?rows1 - ?newRows1 AS ?newRows2)
}
''',
        ),
    ],
)
    })
```

Figure 5.6: Example of Python code snippet showing a data partitioning component with the data transformations expressed in SPARQL.

Consequently, given these added properties to the annotation of intermediate datasets, specific data tags can be assigned to the output datasets as shown by the specific SHACL shape example for the training dataset in Listing 5.7.

These data tags play a very significant role in defining the input and output specifications of tb:Implementation nodes and their corresponding tb:Component nodes. For example, in the case of classification workflow employing a decision tree model, the decision tree learner component will have an input specification of cb:TrainTabularDatasetShape, while the decision tree applier component will have an input specification of cb:TestTabularDatasetShape. This

```
# TrainTabularDatasetShape
cbox.add((cb.isTrainConstraint, RDF.type, SH.PropertyConstraintComponent)
    )
cbox.add((cb.isTrainConstraint, SH.path, dmop.isTrainDataset))
cbox.add((cb.isTrainConstraint, SH.datatype, XSD.boolean))
cbox.add((cb.isTrainConstraint, SH.hasValue, Literal(True)))

cbox.add((cb.TrainTabularDatasetShape, RDF.type, SH.NodeShape))
cbox.add((cb.TrainTabularDatasetShape, RDF.type, tb.DataTag))
cbox.add((cb.TrainTabularDatasetShape, SH.property, cb.isTrainConstraint)
    )
cbox.add((cb.TrainTabularDatasetShape, SH.targetClass, dmop.
    TabularDataset))
```

Figure 5.7: Python code snippet showing data tag example expressed in the form of a SHACL shape for training data sets

approach eliminates the use of literal assignments when linking between the inputs and outputs during logical planning and contributes to the generalization of adding more partitioning components that could split the data into more than two partitions in the future (as will be seen in Section6.1).

The next step in generalizing the workflow generator is the generalization of the workflow builder. As discussed above, in the previous approach, there was a distinction between the construction of a train-test workflow and a simple workflow. However, in the generalized logical planner (shown in Algorithm 5.1), the workflow builder is more generalized as can be shown in the pseudocode in Algorithm 5.2. The generalized workflow builder works the following way:

- For each logical plan, the set of components constructing the plan is retrieved.
- Each component is checked for the existence of an equivalent testing component; if there is not, a simple workflow is constructed.
- If there is an equivalent testing component, the output specifications of the component are checked to see if they imply the partitioning of the data.
- If the data is partitioned, the following steps consider the existence of the partitioned data.
- If the data is not partitioned, the following steps assume a simple workflow structure.

```
    train_query = f'''
                    PREFIX sh: <{SH}>
                    PREFIX rdfs: <{RDFS}>
                    PREFIX cb: <{cb}>
                    PREFIX dmop: <{dmop}>

                    ASK {{
                        {{
                        {io_shape.n3()} a sh:NodeShape ;
                                        sh:targetClass dmop:TabularDataset ;
                                        sh:property [
                                            sh:path dmop:isTrainDataset ;
                                            sh:hasValue true
                                        ] .
                        }}
                    }}
                    '''
```

Figure 5.8: SPARQL query used to retrieve training data sets from the output of a data partitioning component

Finally, among the most important steps in generalizing the workflow generator is the creation of the SPARQL queries necessary to retrieve the needed information from the generalized ontology described in detail in Chapter 2. A relevant example of the queries used within the generation process is the query used to identify the training dataset from the outputs of the data partitioning component shown in Figure 5.8.

5.3 Rule-based Selective Workflow Generation

Upon closer inspection of the algorithm described in Algorithm 5.1, it can be observed that the algorithm is easily prone to rapidly grow in terms of complexity due to the following factors:

- **Number of Components** (C) capable of solving a certain task (as observed in line 4).
- **Number of Component Requirements** (R) needed to be satisfied in the dataset before a component can be used within a workflow (as seen in line 7).
- **Number of Requirement-satisfying Components** (T) available to fulfill a certain component requirement (as seen in line 10).

Thus, an approximation of the time complexity of Algorithm 5.1, assuming that all the components in C have the same number of requirements in R and each r_i has the same number of fulfilling components T, would be:

$$\mathcal{O}\left(\sum_{c_i \in C} \prod_{r_i \in R_i} |t_i|\right) = \mathcal{O}(|C|T^R) \tag{5.1}$$

However, it should be taken into account that this approximate complexity does not take into account the actual workflow building process (line 15), in addition to the complexity of all SPARQL-querying-reliant operations; SPARQL queries complexity is considered PSPACE-Complete according to [25].

The first intuitive thought that would come after looking at Equation 5.1 is whether there is any approach to control any of the factors that influence the complexity. As highlighted in Section 2.3, the focus of this work will be on operator choice, that is, the selection of requirement-satisfying component. As can be seen in Figure 5.9, some component requirements or data transformations can be accomplished using a number of semantically different components. For example, there are three different scaling methods to scale the features of a dataset, each having advantages and disadvantages according to the characteristics of the input dataset. Similarly, the choice between mean imputation and row removal in the case of empty value presence depends on the use case and, most importantly, the characteristics of the dataset. Therefore, the main motive of this section of the work is to introduce an approach to avoid the generation of all the possible workflows from all the preprocessing component combinations by selecting the most effective preprocessing components for each step, which could significantly decrease the number of generated workflows and the generation time needed. However, the important question that arises is regarding the component selection mechanism.

As shown in Figure 5.9, preprocessing component selection is performed considering the characteristics of the input dataset. For example, the decision on whether to remove null values or use mean imputation can be made by looking at the percentage of missing values in the dataset. Thus, it is very important for the dataset to be profiled; extract metadata characteristics of the input dataset, including statistics and the results of some test on dataset features (such as the normality of features). The profiling of the data set will result in the dataset being assigned to tb:DataTag nodes similar to those specifying the inputs and outputs in the case of tb: Implementation nodes; an example of this is the shape shown in Figure 5.10, which is assigned to a dataset based on the percentage of missing values in a data set.

In this case, there will be a tb:Rule node that connects the preprocessing node to a tb:Problem

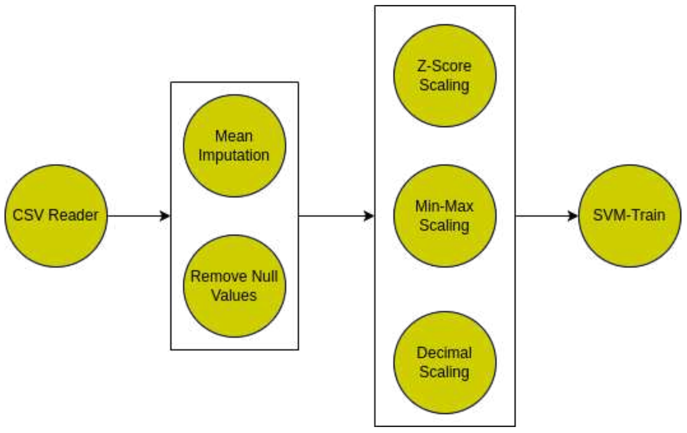


Figure 5.9: Different components can achieve the same requirement but in semantically different manners.

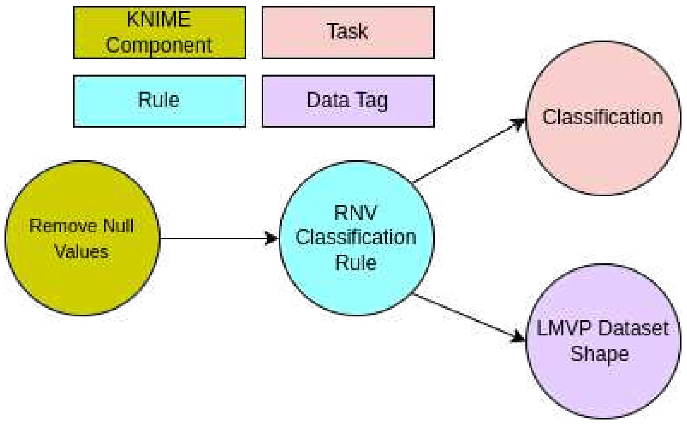


Figure 5.10: An example of how the component removing null values is connected to the the data tag related to the dataset possessing low percentage of missing values and Classification being the task when the rule can be applied.

node and a tb:DataTag node representing that the selection process of the preprocessing component takes into consideration the task at hand, in addition to the meta-data characteristics a dataset must possess.

The process by which the preprocessing components are chosen is described in the pseudocode in Algorithm 5.4. For a given task T, a list of components C to select from, and an input dataset d, the following steps are performed at the logical planner level:

1. For each component in C, the rules to be checked are retrieved from the ontology.

2. Then, each component is assigned to a score that relies on the number of rules it satisfies with respect to the input dataset, in addition to the weight of the rule satisfied. In the case of a component not satisfying a certain rule, the component's score is penalized.

3. The final score of the component is a combination of the actual component score and the component's rank within the set of all the components (a rank is a numeric value that is assigned to each of the components depending on the general preference of the components).

4. Finally, the components with the highest final scores are selected; in the case of two or more components having the same score, the rank is used to favor between them.

5. In the case of preprocessing components that do not possess a certain set of rules to be evaluated against or a predetermined ranking, the user will have the option to input the percentage of preprocessing components to be used instead of the whole set of components. The percentage of components to be selected are chosen randomly.

The method of applying the rules and incorporating them into the components is discussed in details in Section 6.1 but an example to illustrate the main ideas discussed in the steps above can be seen in the code in Figure 5.11, where the python code shows the different scaling components and how the rules are encoded. It should be noted that for each task, there are different sets of rules; in the case of classification tasks, the choice of the scaling component relies mainly on the distribution of the data, as well as the existence of outliers. The rules have different weights in the case of each component depending on the cruciality of the rule. For example, to select the Z-Score scaling component, it is very important that the data follows a normal distribution more than whether there exist outliers or not; the exact opposite for Min-Max scaling is the case with the existence of outliers being the most crucial rule regardless of the distribution of the data.

The details of the methods used to profile the data corresponding to the data tags in the rules highlighted in Figure 5.11 are discussed in Section 7.1.2.

Algorithm 5.4 Get Best Components Algorithm

```
1:  G: Ontology
2:  t: Task
3:  C: List of Components
4:  d: Dataset
5:  p: Percentage
6:  procedure GetBestComponents($G, t, C, d, p$)
7:      for each $c_i$ in $C$
8:          $c_i\_rules = get\_component\_rules(G, t, c_i)$    ▷ Get the rules for each component corresponding to a task
9:          $score$=0
10:         $components[c_i] = score$
11:         for each $cr_i$ in $c_i\_rules$
12:             if satisfies_shape($cir_i$, $d$)
13:                 $score+ = rule\_weight$    ▷ Assign a score to each component based on the weights of the satisfied rules
14:             else
15:                 $score- = rule\_weight$    ▷ Penalize the component for each rule it violates
16:             end if
17:             $components[c_i] = (score, rank)$    ▷ final score of the component relies on rule score and assigned rank
18:         end for
19:         sort($components[c_i]$)    ▷ Sort the components based on their final score
20:     end for
21:     $best\_components$ = highest_scores($components$)    ▷ return the components with the best score
22:     OR
23:     $best\_components$ = pick_randomly($components, p$)    ▷ when all the components have equal final scores, pick $p$ of them randomly
24:     return $best\_components$
25: end procedure
```

```
min_max_scaling_component = Component(
name='Min-Max Scaling',
rules={
    (cb.Classification, 2):[
        {'rule': cb.NotOutlieredDatasetShape, 'weight': 2},
        {'rule': cb.NotNormalDistributionDatasetShape, 'weight': 1}
    ],
    (cb.DataVisualization, 1): [
        {'rule': cb.TabularDataset, 'weight': 1}
    ]
})

z_score_scaling_component = Component(
name='Z-Score Scaling',
rules={
    (cb.Classification, 3):[
        {'rule': cb.NormalDistributionDatasetShape, 'weight': 2},
        {'rule': cb.OutlieredDatasetShape, 'weight': 1}
    ],
    (cb.DataVisualization, 1): [
        {'rule': cb.TabularDataset, 'weight': 1}
    ]
})

decimal_scaling_component = Component(
name='Decimal Scaling',
rules={
    (cb.Classification, 1):[
        {'rule': cb.NotNormalDistributionDatasetShape, 'weight': 1},
        {'rule': cb.OutlieredDatasetShape, 'weight': 1}
    ],
    (cb.DataVisualization, 2): [
        {'rule': cb.TabularDataset, 'weight': 1}
    ]
})
```

Figure 5.11: Python code snippet showing the different scaling components and how the rules are encoded within each component taking into account the main task and dataset characteristics essential for the rule to be deemed valid; each rule has a weigh and each component has an overall rank within each task.

6 Extending the Workflow Generation Framework

As a result of the proposed ontology and the generalized workflow generator, this chapter focuses on providing clear guidelines on the extension of the workflow generation framework to cover more intents without adjusting the workflow generation algorithm.

6.1 General Guidelines on Extending the Workflow Generator

Among the very important results that stem from the generalization efforts of this work is the composition of a complete set of guidelines that enable the extension of the workflow generation framework to widen the scope of the possible user intents by adding to the existing set of tasks, algorithms, and non engine-specific concepts. Furthermore, a subset of these guidelines focus on the extension of workflow generation to specific execution engines.

6.1.1 Engine-Agnostic Extension

The following guidelines should be used for the purpose of extending the workflow generator on an engine-agnostic level:

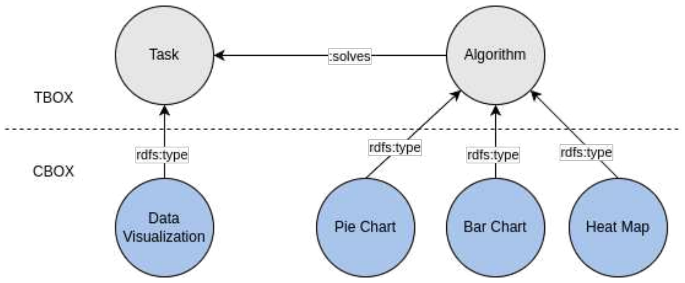


Figure 6.1: An example of adding a task along with the algorithms capable of solving it to the CBOX level.

- Given a specific task t and a set of algorithms $A_t = a_t$ where each a_t solves t, both t and A_t must be added to the CBOX of the ontology (example shown in Figure 6.1). Furthermore, in the case of an added task including a set of subtasks $ST_t = st_t$, it is also important to add them within the CBOX (Figure 6.1 shows the example of the visualization task).

- For $a_t \in A_t$, define the set of implementations $M_a = m_a$. Part of the definition of each m_a should be assigned to one of the tb:Implementation subclasses. The current existing subclasses are: tb:LearnerImplementation, tb:ApplierImplementation and tb:VisualizerImplementation. For example, in the case of the SVM algorithm, it has two possible implementations: an SVM learner implementation (tb:LearnerImplementation) and an svm applier implementation (tb:ApplierImplementatino); the same case would be for normalization or imputation algorithms. As for tb:VisualizerImplementation, it is very specific to visualization algorithms, such as heat map. In the event that the new implementations are not categorized under any of the existing subclasses, a new subclass could be added within the TBOX of the ontology, or the new implementations will be of type tb:Implementation by default (see Figure 6.2).

- For each m_a, the list of input and output specifications ($\mathcal{I}_m$, $\mathcal{O}_m$) must be defined. If the SHACL shape corresponding to any of the specifications does not exist, it must be added to the CBOX of the ontology according to the way shown in Figure 6.3.

- If there is a need to define new SHACL shapes related to the requirements or transformations applied by an implementation, a new SHACL shape must be defined in the CBOX (example in Figure 5.7 for one of the partition outputs).

- For each implementation m_a, a set of components $C_m = \{c_m\}$ must be defined, where $|C_m| \geq 1$. As mentioned in the formal definition of the term Component3.2, the set of

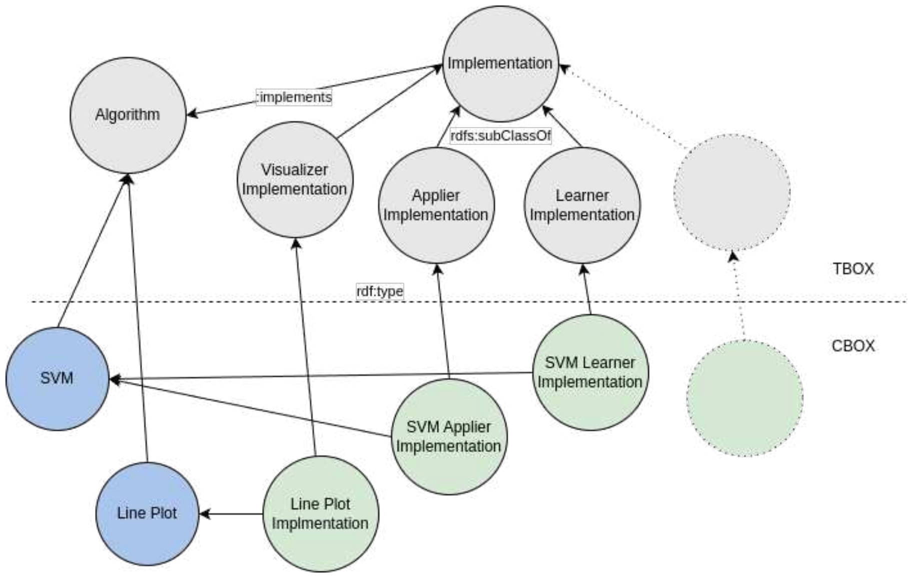


Figure 6.2: An example of adding a new implementation subclass along existing subclasses and vocabulary.

exposed and overridden parameters must be chosen for each of the created components from the set of all implementation parameters. In Figure 6.4, an example of a heatmap visualizer component is defined where the exposed parameters are "label column" and "chart Title".

- Another aspect of a component's definition is the creation of the set of transformation queries; the changes to the input dataset annotations (example in Figure 6.4).

- The final part of the engine-agnostic extension is the addition of the selection rules for transformation components. As shown in the example showing the selection rules for the scaling components in Figure 5.11. Each component c_m has as set of rules R_c for each $t \in T$; each rule has a weight depending on the importance of the rule for the component. Also the components have ranks within each task that are ordered in a descending manner (the greater the rank, the more favorable the component is for a specific task).

6.1.2 Engine-Specific Extension

In the case of engine-specific extensibility, the guidelines are mainly based on determining the execution engine and identifying the important configurations needed. The following guide-

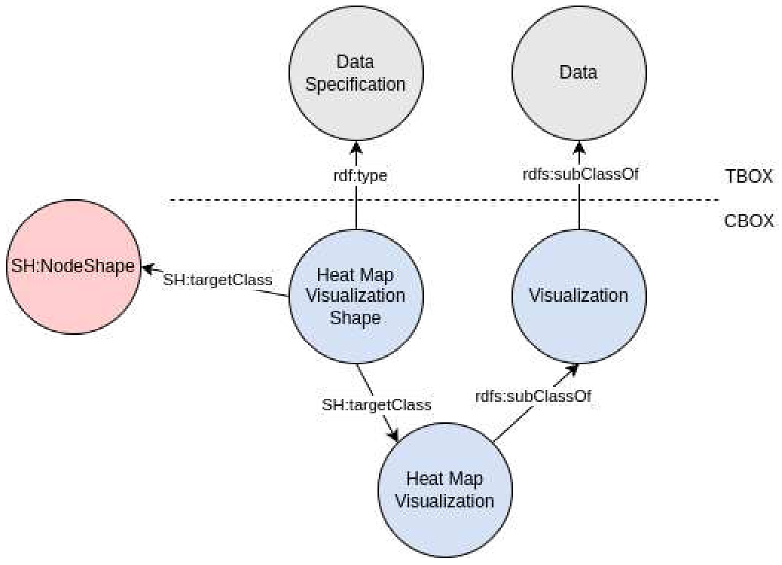


Figure 6.3: An example of adding a new implementation subclass along existing subclasses and vocabulary.

lines will be as general as possible; however, the examples will be mainly from the KNIME engine use case:

- For each execution engine to be included, there will be customized subclasses for the Implementation and Parameter classes that take into consideration specific configurations related to the nodes or components of the execution engine. Through Figure 6.5, it can be seen that the engine-specific subclasses are KnimeImplementation and KnimeParameter; they both have engine-specific arguments, such as knime node factory. Moreover, the input specification has two data tags related to the data not containing null values and the data being scaled. More importantly, the output specifications correspond to two outputs: a visualization and a tabular dataset.

- The set of engine-specific parameters need to be added to the definition of the engine-specific implementation. Although one can argue that parameters are unified across the algorithms since the definition of the algorithm, in nature, is engine-agnostic, the issue is with the values accepted by these parameters along with, most importantly, the existence of other parameters related to the engine and node configuration that cannot be disregarded. An example can be found in Figure 6.5.

- There may be a need to deal with engine-specific data specifications that may need to be defined; however, this is totally dependent on the engine definitions.

```
exposed_params = [
    'labelColumn', ### label column
    # Creating displayed cols depending on the input, ### numeric column
    'chartTitle', ### Title of the scatter plot
]

heatmap_visualizer_component = Component(
    name = "Heatmap Visualizer",
    implementation = heatmap_visualizer_implementation,
    exposed_parameters=[
        param for param in heatmap_visualizer_implementation.parameters.keys
            () if param.knime_key in exposed_params
    ],
    transformations = [
        CopyTransformation(1, 2),
        Transformation(
            query = '''
INSERT DATA {
    $output2 dmop:hasColumn _:heatmapColumn .
    _:heatmapColumn a dmop:Column ;
                    dmop:hasName "Selected (Heatmap)" ;
                    dmop:hasValue false .
}
'''
        )
    ]
)
```

Figure 6.4: Python code snippet showing an example of the heatmap visualizer component with the set of exposed parameters being "labelColumn" and "chartTitle"

- Finally, although the development of the dataset annotator and the engine-specific workflow translator was not within the scope of the work presented. It should be noted that the KNIME workflow translator introduced by [7] was functional with the need to tweak some aspects to consider a wider scope of nodes. As for the data annotator, there was a slight need to change the definition of some data types to conform to KNIME's data type definitions.

```
heatmap_visualizer_implementation = KnimeImplementation(
    name = "Heatmap Visualizer",
    algorithm = cb.HeatMap,
    parameters = [
        KnimeParameter("Label Column", XSD.string, "$$CATEGORICAL$$", '
            labelColumn', path="model"),
        KnimeParameter("Colunms Included Names Array Size", RDF.List, "
            $$NUMERIC_COLUMNS$$", 'included_names', condition="$$INCLUDED$$",
             path="model/columns"),
        KnimeParameter("Colunms Excluded Names Array Size", RDF.List, "
            $$NUMERIC_COLUMNS$$", 'excluded_names', condition="$$EXCLUDED$$",
             path="model/columns"),
        KnimeParameter("Image Width", XSD.int, 800, 'imageWidth', path="model
            "),
        KnimeParameter("Image Height", XSD.int, 600, 'imageHeight', path="
            model"),
        KnimeParameter("Resize to Full Window", XSD.boolean, True, '
            resizeToWindow', path="model"),
        ## Definitions for the rest of KNIME parameters
    ],
    input = [
        [cb.NonNullTabularDatasetShape, cb.NormalizedTabularDatasetShape, cb.
            TabularDataset]
    ],
    output = [
        cb.HeatMapVisualizationShape,
        cb.TabularDataset
    ],
    implementation_type = tb.VisualizerImplementation,
    knime_node_factory = 'org.knime.js.base.node.viz.heatmap.
        HeatMapNodeFactory',
    knime_bundle = KnimeJSBundle,
    knime_feature = KnimeJSViewsFeature
)
```

Figure 6.5: Python code snippet showing an example of an engine-specific implementaiton for the heatmap visualizer KNIME implementation.

7 Experimentation

In this chapter, two different experimental settings are presented, with Section 7.1 delving into the settings of each experiment. The first experiment setting features synthetic scenarios that aim to understand the performance of the generation algorithm and compare it to the previous approach. The second setting focuses more on testing the effectiveness of the rule-based pruning approach in terms of the selected workflows achieving near-best results to the global optimum achieved by the workflows generated in a brute force manner. Section 7.2 discusses the results of each experimental setting. It should be noted that the experimental runs mentioned in this section were performed on a Linux Ubuntu 22.04 machine with an AMD Ryzen 7 5800H 4.4GHz CPU and 16GB RAM.

7.1 Experimental Setting

7.1.1 Performance Benchmarking

As concluded previously by the analysis performed in Section 5.3, some of the main factors affecting the complexity of the generation algorithm are the following:

- **Number of components solving a task**: examples of this would be the different SVM components along with Decision Trees component that can be used to solve a classification task.

- **Number of preconditions for each component**: an SVM component has two preconditions that ensure the features of a dataset do not contain any null values and are scaled.
- **Number of components satisfying a precondition**: the precondition of a dataset to be scaled can be achieved using three possible components: Z-Score scaling, Decimal scaling, or Min-Max scaling.

The experimental setting relies on the creation of synthetic CBOXs taking into account the variables mentioned according to Table 7.1. A single user intent with a single problem and a single dataset is taken as an input. Then, utilizing one CBOX at a time, the problem is solved using a single algorithm with the number of components featured in the synthetic CBox; hence, generating the possible workflows.

# Components	# Preconditions	# Components per Condition
5	1	1
	2	2
10	3	3
	4	4
100	5	5

Table 7.1: Different Parameters for Synthetic CBox Creation

The generation experiments take into account four different scenarios:

- **Previous Brute-Force Approach**: the generation algorithm proposed in [7] taking into account all the possible combinations of components satisfying each precondition.

- **Proposed Brute-Force Approach**: the generation algorithm proposed in this work, which also considers all the possible combinations of components to achieve each precondition.
- **Proposed Selective Approach with 50% of the Components**: the proposed generation algorithm along with the rule-based pruning technique resulting in the selection of 50% of the components achieving a precondition instead of all of them.
- **Proposed Selective Approach with 1 Component**: the proposed generation algorithm with the rule-based pruning technique being most discriminant and selecting one component from all the components available to fulfill a precondition.

In this experiment, for each scenario, the different parameter values are collected along with the number of generated workflows and the generation time.

7.1.2 Evaluating the Rule-based Optimization

In this experimental setting, the main goal was to evaluate the performance of the workflows selected by the rule-based selection technique for a certain problem with regard to the performance of all the possible workflows generated. For this purpose, it was necessary to populate the CBOX of the ontology with components that correspond to KNIME components to enable the execution and evaluation of the generated workflows. Table 7.2 shows a list of the components created to populate the ontology along with the corresponding number of KNIME node parameters, overridden parameters, exposed parameters and transformations.

As for the rules incorporated for the purpose of selective generation process, some meta-characteristics of the dataset were extracted using the available data annotator, then SHACL shapes corresponding to these meta-characteristics were created. These meta-characteristics are:

- **Feature Normality**: each feature in the input dataset was tested to determine whether it had a normal distribution or not. The main test used is the Shapiro-Wilk test; if the $p\text{-value} < 0.05$, the feature is considered normally distributed. If $p\text{-value} > 0.05$, the skewness and kurtosis tests are used to determine whether the distribution is close to a normal distribution; if $|\text{skewness}| < 1$ and $|\text{kurtosis}| < 2$, the feature is considered normally distributed; otherwise, the feature is deemed not normally distributed. If the dataset contains at least one feature with a normal distribution, the whole dataset is considered normally distributed.

KNIME Node	**#P**	**Component**	**#OP**	**#EP**	**#T**
CSV Reader	65	Local CSV Reader	2	1	1
Partitioning	6	Random Relative	2	2	3
		Random Absolute	2	2	3
		Top K Relative	2	1	3
		Top K Absolute	2	1	3
Missing Value	7	Mean Imputation	3	0	2
		Drop rows	3	0	4
Missing Value (Applier)	0	Missing Value Applier	0	0	2
Normalizer (PMML)	4	MinMax Scaling	1	2	4
		Z-Score Scaling	1	0	4
		Decimal Scaling	1	0	4
Normalizer Apply (PMML)	0	Normalizer Applier	0	0	6
Decision Tree Learner	16	Decision Tree Learner	0	16	0
Decision Tree Predictor	5	Decision Tree Predictor	0	5	2
SVM Learner	9	Polynomial SVM Learner	1	5	1
		HyperTangent SVM Learner	1	4	1
		RBF SVM Learner	1	3	1
SVM Predictor	4	SVM Predictor	0	4	2
Pie Chart	106	Pie Chart Sum Visualizer	1	2	0
		Pie Chart Average Visualizer	1	2	0
		Pie Chart Count Visualizer	1	2	0
Bar Chart	122	Bar Chart Sum Visualizer	1	2	0
		Bar Chart Average Visualizer	1	2	0
		Bar Chart Count Visualizer	1	2	0
Histogram	140	Histogram Sum Visualizer	1	2	0
		Histogram Average Visualizer	1	2	0
		Histogram Count Visualizer	1	2	0
Scatter Plot	75	Scatter Plot Visualizer	0	2	2
Line Plot	85	Line Plot Visualizer	0	2	1
Heatmap	68	Heatmap Visualizer	0	2	2

Table 7.2: KNIME nodes integrated into the ontology as components within the CBOX to perform the experiments

- **Outlier Detection**: each feature in the dataset is tested; if there is one feature that contains an outlier, the whole dataset is deemed to contain outliers. Modified Z-Score was used for this purpose with points having an absolute score > 3, regarded as outliers. If the percentage of outlier points is above 25%, the feature is considered to contain outliers.

- **Missing Value Percentage**: a direct metric containing the percentage of rows containing at least one null value.

Moreover, three different datasets were used to create the intents driving the creation of the workflows: Titanic[*] dataset, Penguins[†] dataset, and Horses[‡] dataset. Finally, all generated workflows were translated into functional KNIME workflows using the workflow translator proposed by [7].

7.2 Results

7.2.1 Results for Performance Benchmarking

In Figure 7.1, we can see some interesting insights about the relations between the number of workflows generated and the time to generate these workflows under different parameters. Each point in the graph represents a tuple of three parameters (x, y, z), where x is the number of components performing a task, y is the number of preconditions for each component and z is the number of components satisfying a precondition. The scales of both axes are logarithmically scaled to facilitate the display of the data points. The general behavior of the relationship between the number of generated workflows and execution time across all generation techniques is linear. However, some important distinctions can be noted among generation approaches. Firstly, there is a slightly better performance shown by the current brute-force generator over the previous brute-force generator, indicated by the lower slope of the data in the linearly scaled data. Secondly, in the case of the 50% selective generator, the main effect is the backward displacement of the curve compared to the brute force curves. Additionally, in the case of the very discriminant selective generator, the number of workflows generated seems to stagnate at certain values despite the increase in generation time.

Looking deeper into the relation between the number of workflows generated with respect to the number of components per condition in Figure 7.2, we can clearly see the exponential

[*]https://www.kaggle.com/datasets/yasserh/titanic-dataset
[†]https://archive.ics.uci.edu/dataset/690/palmer+penguins-3
[‡]https://archive.ics.uci.edu/dataset/47/horse+colic

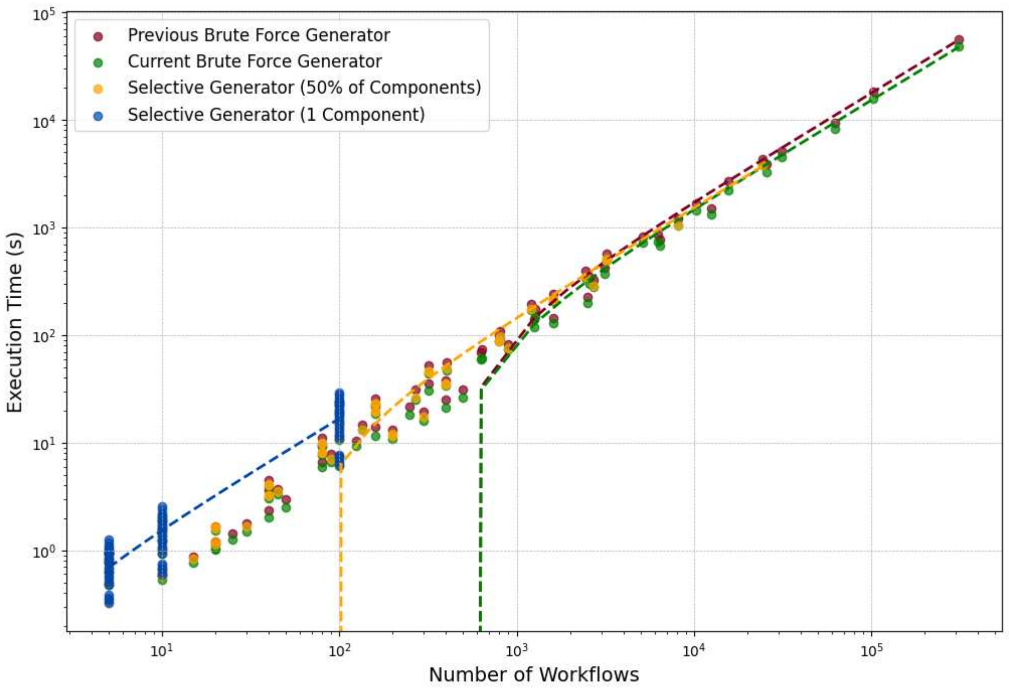


Figure 7.1: Relationship between the number of workflows and generation time in the case of the four generation scenarios

relationship between these two variables, especially in the case of brute-force generators. However, in the case of the 50% selective generator, the exponential growth is halted and the curve becomes composed of linearly increasing and constant lines. More interestingly, in the case of discriminant genertor, we can understand the reason for the stagnation seen previously in Figure 7.1; the number of generated workflows becomes equal to the number of components capable of solving a certain task.

On the other hand, in Figure 7.3, where the relation between the number of workflows generated and the number of preconditions per component, we can see that all generators display a similar exponential relationship (except in the case of the most discriminant generator). Nevertheless, although both factors have an exponential effect on the number of workflows, the number of preconditions per component shows a stronger influence compared to the number of components per condition, evident by the higher number of generated workflows for the same values of both variables.

As for Figure 7.4, the relationship between the generation time and the number of components per condition is demonstrated. As noted previously, the proposed brute-force generator shows a better performance than the brute-force generator proposed in [7]. The same halting

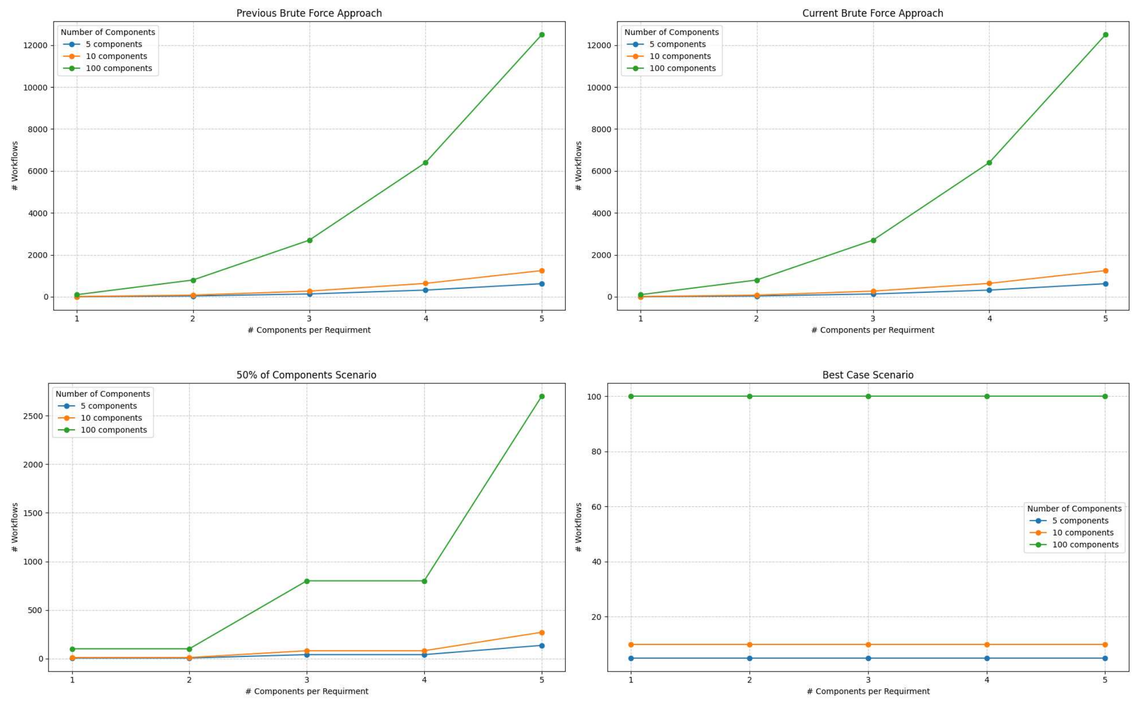


Figure 7.2: The relation between the number of generated workflows and components per precondition (preconditions per component = 3)

phenomena previously discussed can also be seen with respect to the generation time. However, the generation time does not show a constant behavior in the case of the very discriminant generator (shown in Figure 7.6, second graph), instead it is behaving linearly. This clearly shows that the execution time for generating the same number of workflows differs and that the main factors that affect it are the complexity parameters.

As for the relationship between the generation time and the number of conditions per component (shown in Figure 7.5), the exponential trend continues to prevail in all the generators, except for the most discriminant generator scenario with the relationship leaning towards linear-like curves as shown in Figure 7.7. As noted in the case of the number of generated workflows, the number of requirements per component seems to have the upper hand in terms of the greater influence over the generation time. In fact, Figure 7.6 shows how the change in the number of conditions per component contributes to the increase of both slope and displacement of the curves unlike how the change in the number of components per condition seems to only change the displacement of the curves in Figure 7.7.

In this section, we were able to see the linear relationship between the number of generated workflows and the generation time across different generators under different generation pa-

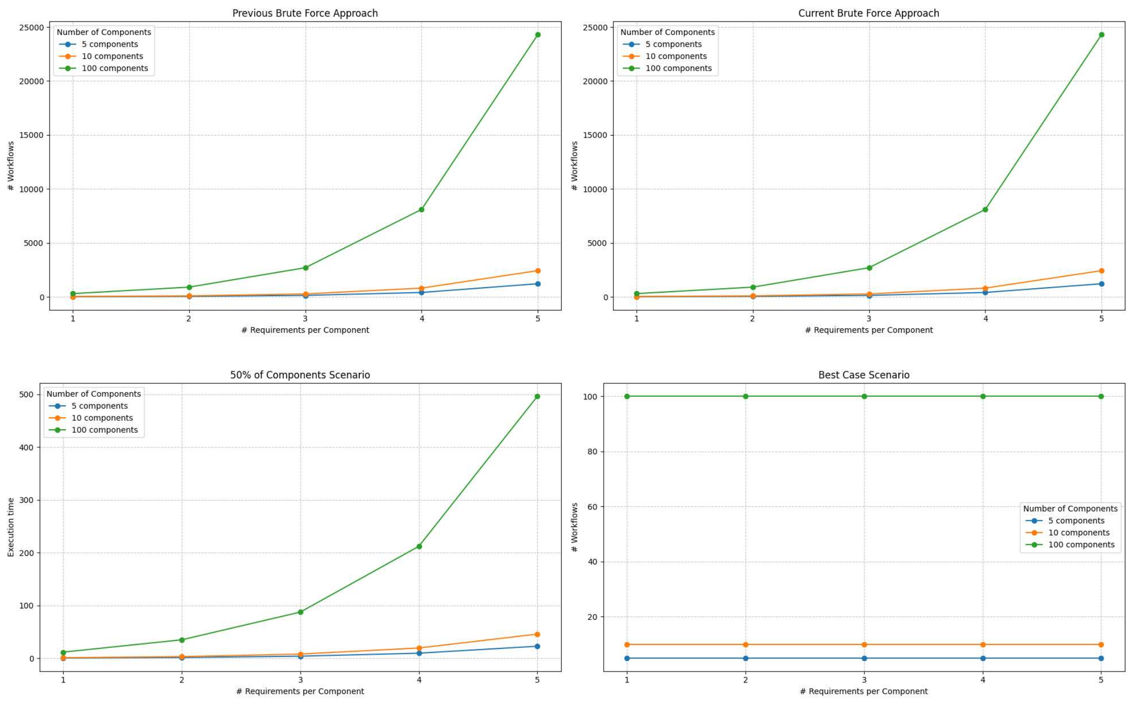


Figure 7.3: The relation between the number of generated workflows and preconditions per component (components per precondition = 3)

rameters. Moreover, it was evident that the preconditions per component variable seem to have a stronger influence over both the number of generated workflows and the generation time, compared to the components per condition variable. The selective generators are able to halt the rapid growth of the number of workflows by decreasing the number of components per requirement used in generating the workflows, specifically in the case of the most discriminant generator resulting in the number of workflows being equal to the number of components. However, the execution time is not tightly related to the number of generated workflows as it was evident that the same number of workflows does not imply the same generation time as the generation time is influenced by the number of requirements per condition.

7.2.2 Results for Rule-based Optimization

In Table 7.3, the significant differences in the number of generated workflows and consequently the generation time can be clearly seen between the brute-force generation algorithm and the rule-based selection. The difference is approximately 4.5 times and almost 80% fewer workflows across the three datasets. Another evident note is how the slight differences in generation

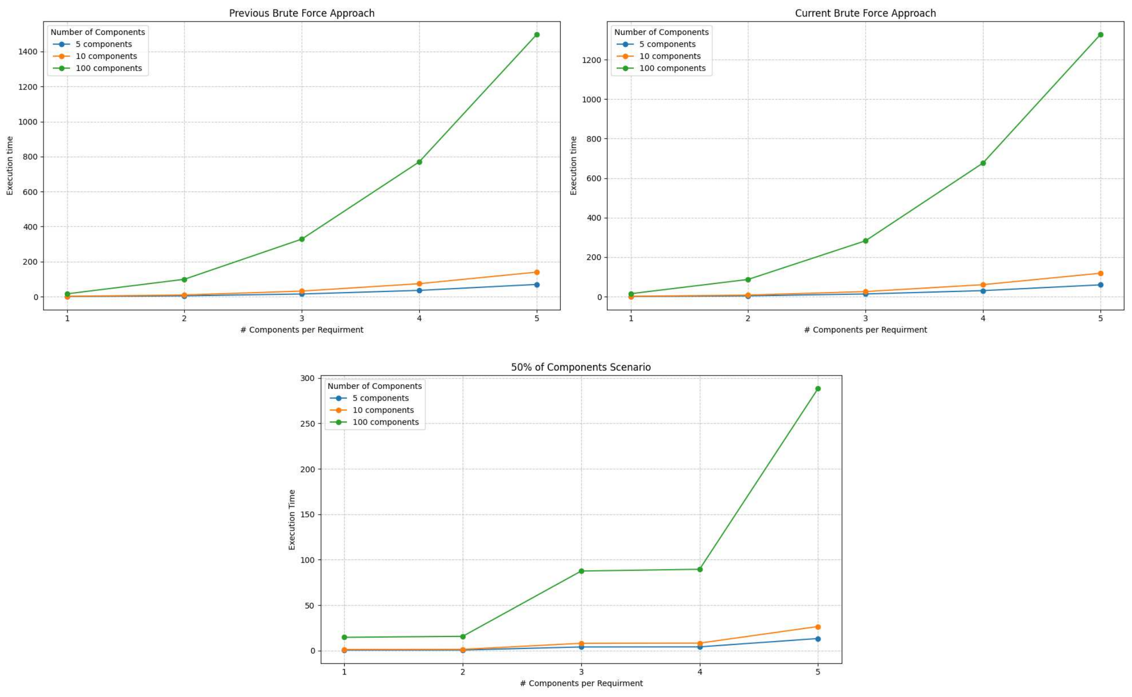


Figure 7.4: The relation between the generation time and components per precondition (preconditions per component = 3)

times could be affected by the number of features each dataset contains, with the Horses dataset having the highest generation time in both cases. The generation times seen in the table are the mean of three generation runs.

Dataset	Generation Time (s)		# Workflows		# Features (KNIME)		# Instances
	Brute Force	Selective	Brute Force	Selective	Num.	Cat.	
Titanic	32.05	6.85	76	16	6	6	891
Penguins	28.34	6.30			4	3	344
Horses	36.40	7.65			11	17	299

Table 7.3: Statistics related to the datasets and generated classification workflows in brute-force generator and selective generator

In the evaluation of the classification workflows, two important metrics were taken into account: balanced accuracy and f1-score for each class. The reason being the normal accuracy metric being easily influenced by imbalanced testing datasets which causes, in most cases, the performance of a model to be overestimated. However, the combination of the f1-score and balanced accuracy gives a more realistic evaluation of the performance of the model by measur-

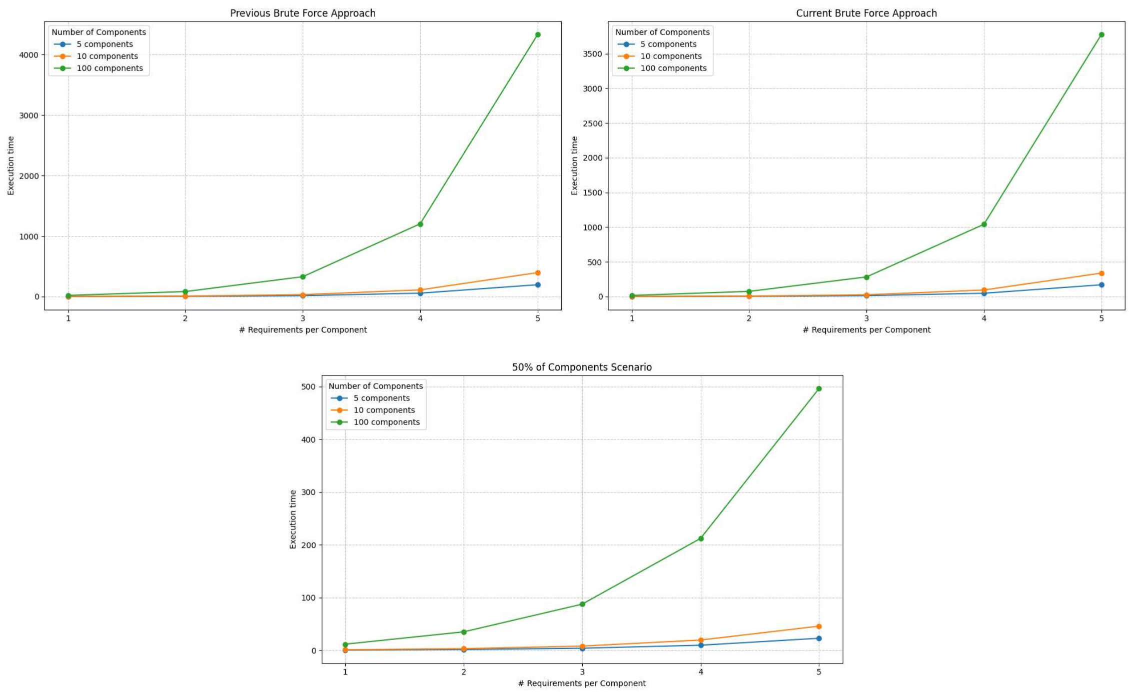


Figure 7.5: The relation between the number of generation time and preconditions per component (components per precondition = 3)

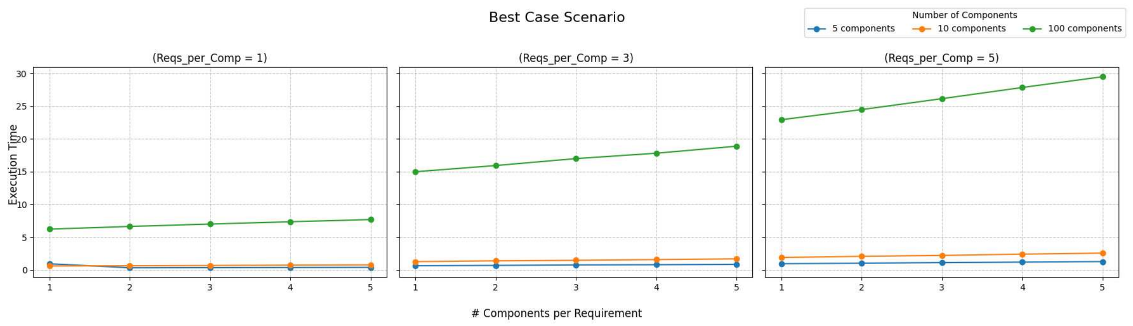


Figure 7.6: The relation between the generation time and components per precondition across different values for preconditions per component in the best case scenario (selection of 1 component)

ing the accuracy of each class individually and then averaging all of them, in addition to the f1-score of each class being an important indicator of the model's ability to generalize over all the classes.

An important distinction to be made has to do with the number of all generated workflows and the number of valid generated workflows. In the case of a workflow resulting in a model with a class f1-score that cannot be calculated, model is deemed invalid since it fails to generalize

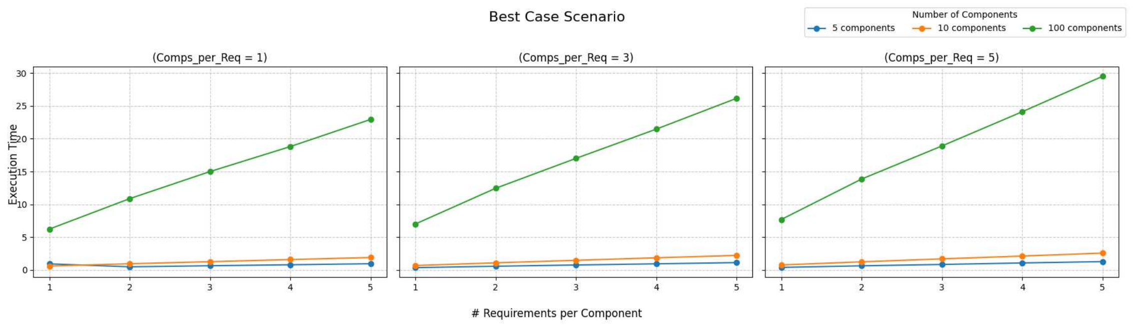


Figure 7.7: The relation between the generation time and preconditions per component across different values for components per precondition in the best case scenario (selection of 1 component)

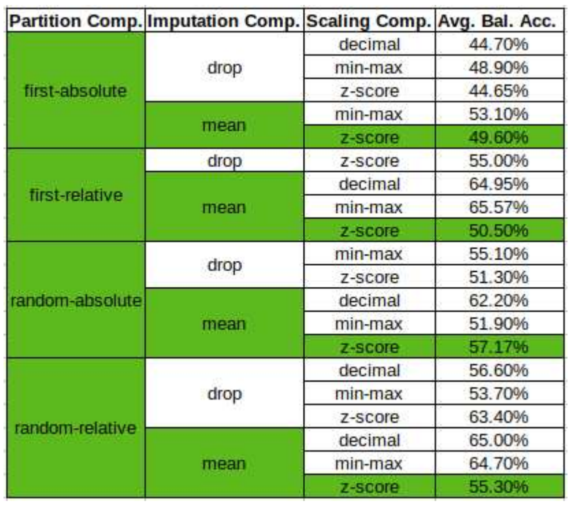
over all the classes and failing to produce an accurately representative balanced accuracy score. Hence, the workflow is deemed to be invalid.

As can be seen in Table 7.4, the percentage of valid workflows with respect to all possible workflows is approximately 51%; while the number of invalid selected workflows is one out of all the selectively generated workflows. In the case of the Titanic dataset, Decision Tree models seem to be the best option yielding the best performance; however, due to decision trees not needing any preprocessing steps, we end up with all the decision tree workflows in both cases of brute force and selective generation attempts. Regarding SVM workflows, we can see that the difference between the best selected SVM model and the best global SVM model does not exceed 2%.

Workflows	All				Valid			
Categories	**#**	**Mean**	**Min.**	**Max.**	**#**	**Mean**	**Min.**	**Max.**
All Workflows	76	54.63%	32.6%	82.68%	39	58.67%	32.6%	**82.68%**
Selected Workflows	16	59.21%	32.6%	82.68%	15	59.82%	32.6%	**82.68%**
DT Workflows	4	77.3%	72.4%	82.68%	4	77.3%	72.4%	82.68%
SVM (All Workflows)	72	53.42%	32.6%	70.4%	35	56.55%	32.6%	**70.4%**
SVM (Selected Workflows)	12	53.17%	32.6%	68.6%	11	53.46%	32.6%	**68.6%**

Table 7.4: Table illustrating the Balanced Accuracy statistics among all the generated and selected workflows for solving the classification task in the Titanic dataset

In Figure 7.8, we can get a sense of the average performance of SVM models produced by valid workflows adopting different combinations of preprocessing components. A very important observation is how the least number of valid workflows feature the first-relative partitioning and dropping rows, which could give insight into a combination that is the least fruitful in terms of producing valid workflows.



Partition Comp.	Imputation Comp.	Scaling Comp.	Avg. Bal. Acc.
first-absolute	drop	decimal	44.70%
		min-max	48.90%
		z-score	44.65%
	mean	min-max	53.10%
		z-score	49.60%
first-relative	drop	z-score	55.00%
	mean	decimal	64.95%
		min-max	65.57%
		z-score	50.50%
random-absolute	drop	min-max	55.10%
		z-score	51.30%
	mean	decimal	62.20%
		min-max	51.90%
		z-score	57.17%
random-relative	drop	decimal	56.60%
		min-max	53.70%
		z-score	63.40%
	mean	decimal	65.00%
		min-max	64.70%
		z-score	55.30%

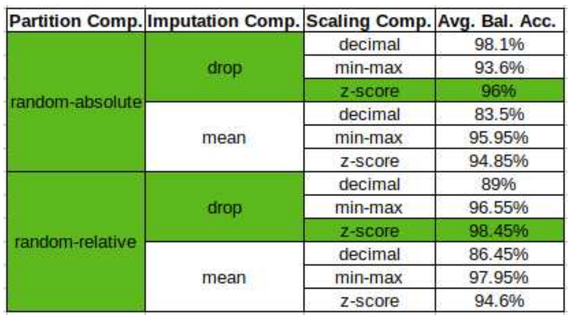
Figure 7.8: The set of pre-processing combinations producing valid workflows for the classification task in the Titanic dataset (selected workflows highlighted in green)

As for the Penguins dataset, as can be seen in Table 7.5, SVM models seemed to be the better choice over Decision Trees, with the best selected SVM model yielding a very close result to the best global SVM model. However, the overall number of functional generated workflows is 58 instead of 76 with 18 workflows not being capable of producing a model due the adopted partitioning technique resulting in a training dataset that contains only one class, which returns a fatal error in KNIME. This is mainly due to the way the dataset is organized (sorted by the class column). Additionally, we can see a very wide gap between the minimum and mean statistics between all and valid workflows across all the workflow categories. This could be mainly influenced by the percentage of valid workflows – approximately 44.8%.

Workflows	All				Valid			
Categories	#	Mean	Min.	Max.	#	Mean	Min.	Max.
All Workflows	58	54.47%	0.0%	100%	26	93.15%	75.2%	**100%**
Selected Workflows	13	52.43%	0.0%	99.2%	6	95.92%	93.03%	**99.2%**
DT Workflows	4	46.78%	0.0%	93.6%	2	93.32%	93.03%	93.6%
SVM (All Workflows)	54	55.04%	0.0%	100%	24	93.14%	75.2%	**100%**
SVM (Selected Workflows)	9	54.94%	0.0%	99.2%	4	97.23%	94.4%	**99.2%**

Table 7.5: Table illustrating the Balanced Accuracy statistics among all the generated and selected workflows for solving the classification task in the Penguins dataset

If we were to look at the valid combinations in Figure 7.9, it can be clearly noticed how the organization of the dataset influenced the validity of certain partitioning techniques. In fact, all partitioning techniques that feature random sampling were valid, while those that feature first (or top k) sampling were not valid. In addition, the best mean was achieved by a selected combination.



Partition Comp.	Imputation Comp.	Scaling Comp.	Avg. Bal. Acc.
random-absolute	drop	decimal	98.1%
		min-max	93.6%
		z-score	96%
	mean	decimal	83.5%
		min-max	95.95%
		z-score	94.85%
random-relative	drop	decimal	89%
		min-max	96.55%
		z-score	98.45%
	mean	decimal	86.45%
		min-max	97.95%
		z-score	94.6%

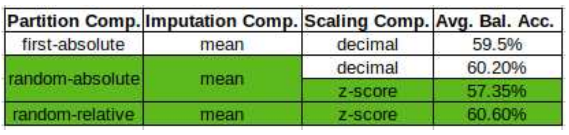
Figure 7.9: The set of pre-processing combinations producing valid workflows for the classification task in the Penguins dataset (selected workflows highlighted in green)

For the dataset with the highest number of dimensions, the overall performance of the models is quite poor (check Table 7.6). In a repetition of a familiar case from the Penguins dataset, 15 workflows were not functional for the following reasons: 3 workflows featuring preprocessing combinations resulting in one-class model training, 11 workflows produced an empty testing dataset and 1 workflow produced an empty training dataset. In general, the percentage of valid workflows to all the functional workflows is very low. As for the performance of the selected workflows, we can see that the selected workflows were able to achieve global optimum performance despite the overall underwhelming performance of the models.

Workflows	All				Valid			
Categories	#	Mean	Min.	Max.	#	Mean	Min.	Max.
All Workflows	61	47.68%	0.0%	66.6%	6	58.6%	53.3%	**61.4%**
Selected Workflows	16	56.83%	50%	66.6%	4	57.98%	53.3%	**61.4%**
DT Workflows	4	56.33%	51.4%	59%	1	56.6%	56.6%	56.6%
SVM (All Workflows)	57	47.08%	0.0%	66.6%	5	59%	53.3%	61.4%
SVM (Selected Workflows)	12	57.01%	50%	66.6%	3	58.43%	53.3%	**61.4%**

Table 7.6: Table illustrating the Balanced Accuracy statistics among all the generated and selected workflows for solving the classification task in the Horses dataset

Looking at the number of valid combinations in 7.10, we can notice the very low number of valid combinations. This can be attributed to multiple factors. The first being the high rate of missing values with the combinations featuring first-relative partitioning and dropping empty rows resulting in solely invalid models or broken workflows. A second possible reason is the nature of the dataset and the classification models at hand, in addition to the high dimensionality of the data, which begs the need for feature engineering techniques as part of the preprocessing components.



Partition Comp.	Imputation Comp.	Scaling Comp.	Avg. Bal. Acc.
first-absolute	mean	decimal	59.5%
random-absolute	mean	decimal	60.20%
		z-score	57.35%
random-relative	mean	z-score	60.60%

Figure 7.10: The set of pre-processing combinations producing valid workflows for the classification task in the Horses dataset (selected workflows highlighted in green)

The following Figures (7.11, 7.12 and 7.13) express the results in tables 7.4, 7.5 and 7.6, respectively. We can notice in the case of Figure 7.11 that the elimination of outliers seen on the left is equivalent to the elimination of all the invalid workflows. However, in the case of 7.12, the outliers appear after eliminating the invalid workflows, but the distribution of the data does not reach very low values; in fact, all balanced accuracy scores are approximately 75% and higher. Finally, for Figure 7.13, the distribution is condensed due to the elimination of many invalid data points that are approximately 0.0% with the new distributions having a lower bound over 50.0%.

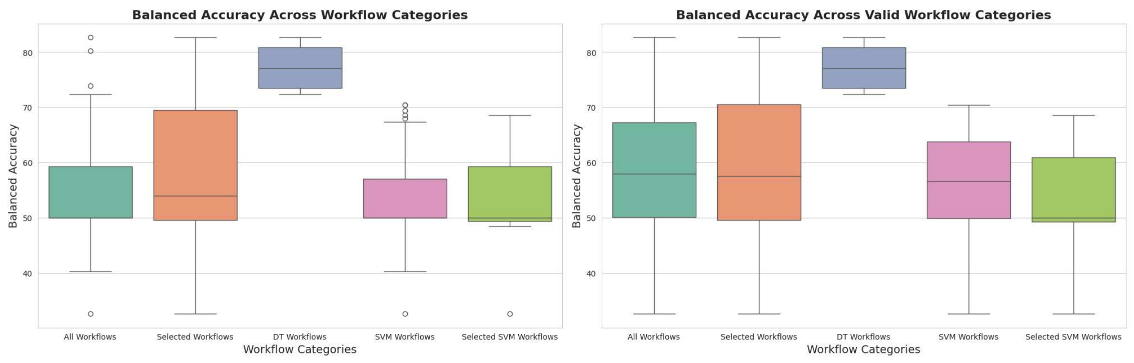


Figure 7.11: Box plots representing the differences in balanced accuracy statistics across the different categories for all functional workflows (on the left) and the valid functional workflows (on the right) for the classification task of the Titanic dataset

As for the visualization workflows, the most important part in evaluating the workflows was missing, the existence of evaluation metrics for produced visualizations. Although there are certain standards related to the evaluation of the understandability of different visualizations from a user point of view (such as the ones introduced in [26]), there is no research or literature

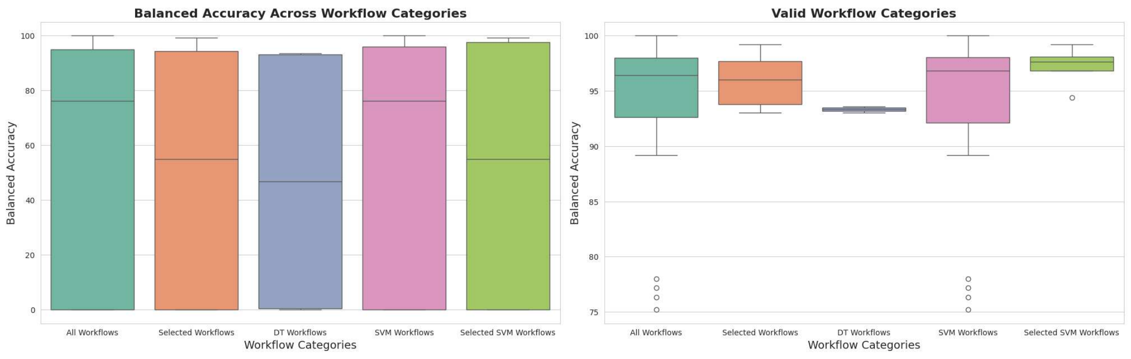


Figure 7.12: Box plots representing the differences in balanced accuracy statistics across the different categories for all functional workflows (on the left) and the valid functional workflows (on the right) for the classification task of the Penguins dataset

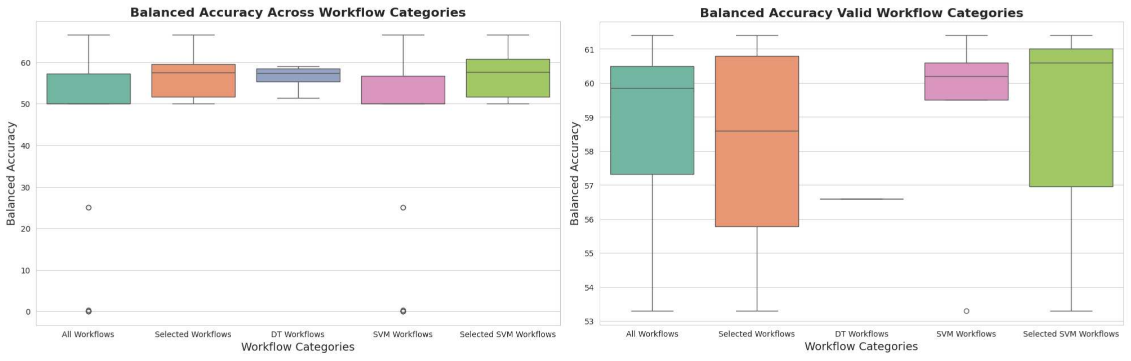


Figure 7.13: Box plots representing the differences in balanced accuracy statistics across the different categories for all functional workflows (on the left) and the valid functional workflows (on the right) for the classification task of the Horses dataset

investigating the effects of different preprocessing methods on different types of visualization. One potential reason could be the role of visualization as an analytical technique that steers the wheel of data understanding as part of the efforts guiding the selection of appropriate preprocessing components or even algorithms to solve a problem, i.e., means within the bigger solution to a problem rather than a stand-alone problem to be solved on its own. Despite all of this, in this part we will look at some interesting observations on the effects of some preprocessing techniques on some visualizations in an effort to detect any patterns.

The heat map plots shown in Figure 7.14 are produced using different scaling and imputation techniques using the Titanic dataset; for example, the heat map on the upper left side is produced after applying mean imputation and z-score scaling. Aside from the differences in scales, the heat map plots produced by the Min-Max scaling seem to show colors from the extreme ends of the scale; meanwhile, Z-Score scaling displays the majority of points around a certain value (due to the assumption of data normality). As for decimal scaling, the colors seem to be more balanced and variable but not with the same extreme variability as with the case in the heat map featuring min-max scaling. On the other end, the use of row removal for null values seems to also increase the variability in each column, unlike the quite similar color patterns produced in the heat maps featuring mean imputation.

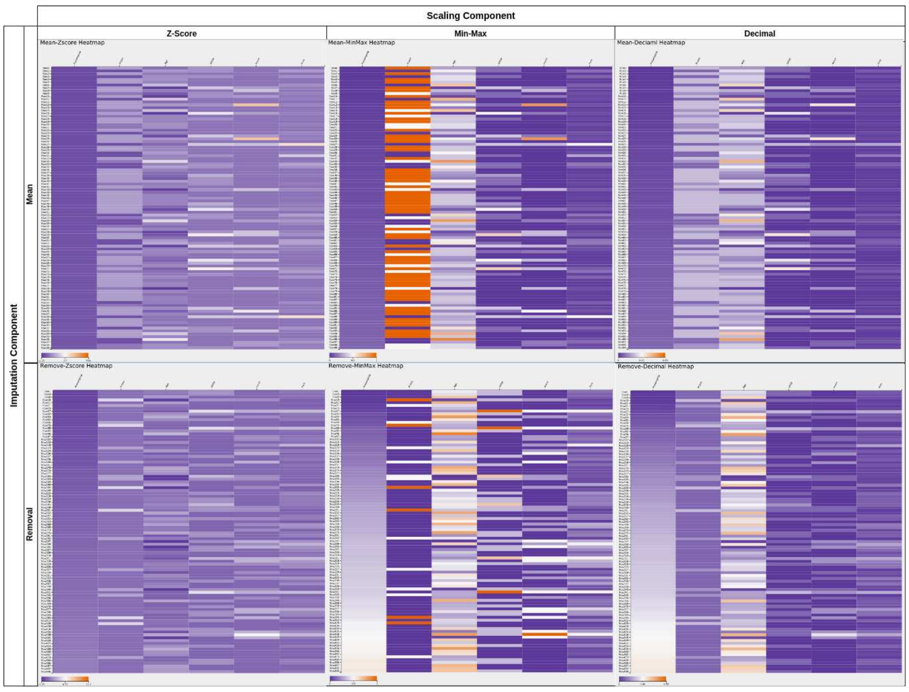


Figure 7.14: Possible heat maps generated from different scaling and imputation methods

Shifting the focus to line plots produced using the Diabetes dataset in Figure 7.15, we can see very different visualizations resulting from the adoption of different scaling techniques. The first interesting observation is the similarity between the line plot produced using decimal scaling and the line plot with no scaling, apart from the different scales, in a clear signal to the decimal scaling ability to preserve the magnitudes across values. In the cases of Z-Score and Min-Max scaling, the produced plots are quite different from the initial plot; Z-score disregards the magnitudes and focuses the data around the means, while Min-Max scaling seems to amplify the variability of the data possibly due to the existence of extreme outliers.

In the case of scatter plots, scaling the data is not of any influence on the overall plot; however, the adopted imputation method makes a difference in how the patterns of the data will be displayed. On the right of Figure 7.16, we can see how the mean imputation creates a set of

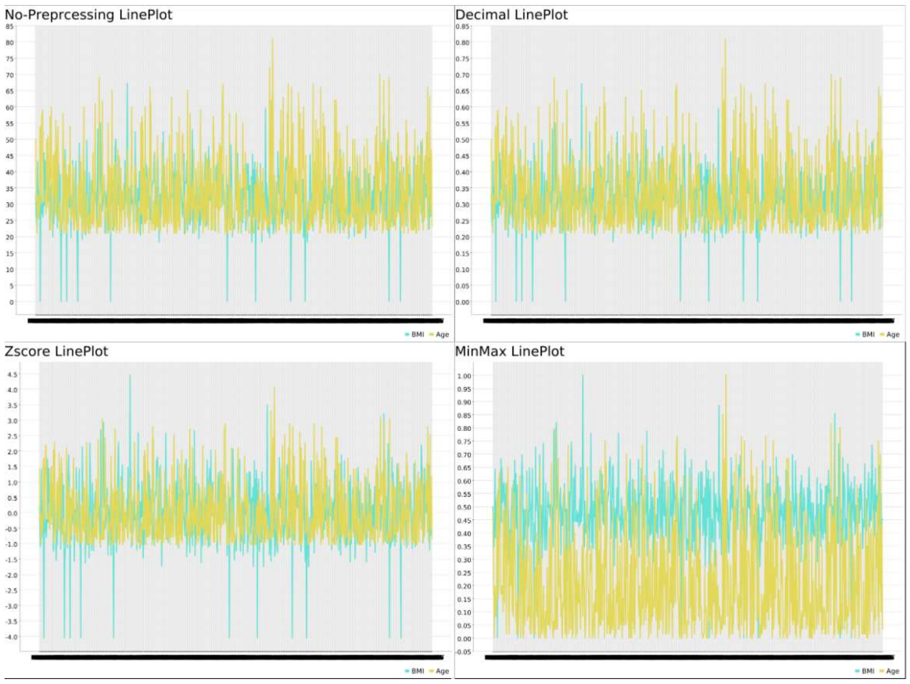


Figure 7.15: Possible line plots generated from different scaling methods

points clustered on the mean value. The question here becomes whether this is useful or not for the analysis of the user or whether it serves as a mere display of data points that serves no actual purpose.

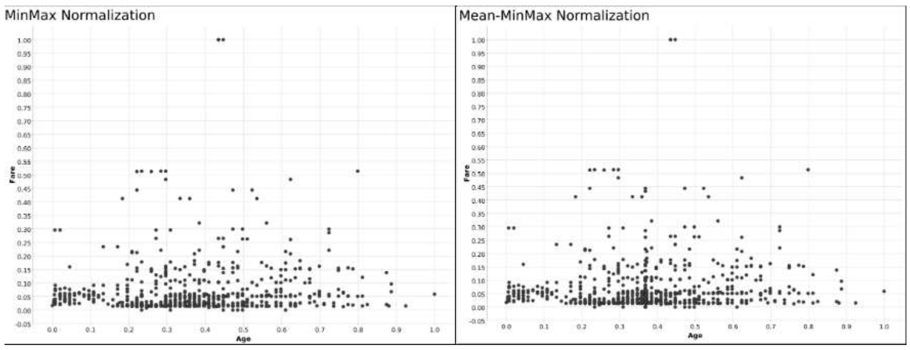


Figure 7.16: A comparison between two scatter plots using two different imputation methods: null values removal and mean imputation

The previous observations seem to show that the adoption of preprocessing techniques in producing visualization could be a result of certain user questions to be answered through these plots. Thus, it seems that the choice of preprocessing techniques for visualization tasks could be entirely reliant on user intents rather than any other factors.

On the other hand, the case for classification workflows shows some interesting relations. It seems that there is a direct trade-off between the number of components performing a task and the ability to reach a global optimum in terms of classification workflow performance, i.e., the more components available to solve a task, the more time it will take to generate the workflows. Although the selective generation shows great promise in solving this problem, we were able to see in the results showcased in Section 7.1.1 that even when we are able to produce a significantly less number of workflows, the generation time still increases due to other more influencing factors, namely the preconditions per component. However, it is possible that future component expansions should focus on the quality rather than the quantity of components available to solve a problem. This was clearly seen in the cases of the Penguins and Horses datasets where a high percentage of workflows produced were invalid or broken due to specific preprocessing components or component combinations not producing reliable output. Such a situation could have been mitigated by having partitioning components that feature stratified sampling, instead of first sampling partitioning. Nevertheless, the rule-based selective generation mechanism could be used to specify such details given the promising results it yielded in these experiments.

8 Conclusion

This work focused on generalizing the framework of mapping user intents to complex analytical workflows taking into account that a user intent is the set of tasks a user wants to perform on a dataset, with or without specifying certain settings such as the algorithm used or specific parameter values. In this work, the details of the ontology used to represent the main concepts and the entire workflow generation process are discussed in Chapter4. In Chapter5, the process of generalized workflow generation along with the rule-based selective generation were introduced. As for Chapter 6, the focus was on the set of guidelines to be used to extend the framework to cover more intents. Finally, in 7, the experimental settings were introduced and we were able to see that the workflow generation time was greatly influenced by the complexity of the generation algorithm rather than the number of generated workflows. Moreover, we were able to see that the selective workflow generator was able to produce classification results that are near the global optimum in less time and generating fewer workflows.

This work was motivated by the need to generalize the framework of mapping user-defined intents to analytical workflows and presents the following contributions:

- Creating a generalized extensible Knowledge Graph representation of the entire process of generating analytical workflows from a user intent, in addition to populating it.
- Developing a generalized, engine-agnostic workflow backed by the populated Knowledge Graph to perform the intent-to-workflow translation, producing analytical workflows covering analytical tasks beyond Machine Learning tasks.

- Integrating a rule-based selective generation mechanism within the generalized workflow generator to ensure the appropriate selection of pre-processing components optimizing both generation time and the number of generated workflows.

In addition to these objectives, a set of extension guidelines was compiled to constitute the first stepping stone for this framework. Moreover, through the results offered by the experiments, it was concluded that the rule-based selective generation shows great promise in reaching a near-optimal solution in less time and less number of generated workflows when compared to the ordinary brute-force generator.

Future work to further extend the potential of this framework can focus on multiple aspects. The first aspect would be related to further understand the importance of pre-processing component choice over the number of components available to perform a task (the trade-off between having numerous options or fewer more effective options), as this could lead to further optimization in generation time. Another important aspect is developing the data annotation transformation within a workflow to attempt to eliminate potentially broken workflows by analyzing possible invalid component combinations or pointless tabular outputs. A very influential future direction could be related to the creation of a data profiler that is capable of extracting useful meta-characteristics that could open the door for further optimization practices or, at the least, enhance the rule-based selective generator. Finally, the integration of an execution engine within the framework could open the door to adopting more optimization practices, as well as facilitating the workflow testing and evaluation processes.

References

[1] J. Giovanelli, B. Bilalli, and A. Abelló, "Data pre-processing pipeline generation for autoetl," *Information Systems*, vol. 108, p. 101957, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437921001514

[2] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705120307516

[3] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," 2020. [Online]. Available: https://arxiv.org/abs/2003.06505

[4] E. LeDell and S. Poirier, "H2o automl: Scalable automatic machine learning," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221338558

[5] R. Olson and J. Moore, *TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning*. Springer, 05 2019, pp. 151–160.

[6] C. Barba-González, J. García-Nieto, M. del Mar Roldán-García, I. Navas-Delgado, A. J. Nebro, and J. F. Aldana-Montes, "Bigowl: Knowledge centered big data analytics," *Expert Systems with Applications*, vol. 115, pp. 543–556, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417418305347

[7] V. D. i Cuesta, "Automated mapping of user intents to complex analytical workflows," Master's Thesis, University of Padova, Padova, Italy, 2023.

[8] A. Balaji and A. Allen, "Benchmarking automatic machine learning frameworks," *CoRR*, vol. abs/1808.06492, 2018. [Online]. Available: http://arxiv.org/abs/1808.06492

[9] M. Zöller and M. F. Huber, "Survey on automated machine learning," *CoRR*, vol. abs/1904.12054, 2019. [Online]. Available: http://arxiv.org/abs/1904.12054

[10] P. Gijsbers, M. L. P. Bueno, S. Coors, E. LeDell, S. Poirier, J. Thomas, B. Bischl, and J. Vanschoren, "Amlb: an automl benchmark," 2023. [Online]. Available: https://arxiv.org/abs/2207.12560

[11] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, *Auto-sklearn: Efficient and Robust Automated Machine Learning*, 05 2019, pp. 113–134.

[12] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter, "Auto-sklearn 2.0: Hands-free automl via meta-learning," 2022. [Online]. Available: https://arxiv.org/abs/2007.04074

[13] C. Wang, Q. Wu, M. Weimer, and E. Zhu, "Flaml: A fast and lightweight automl library," 2021. [Online]. Available: https://arxiv.org/abs/1911.04706

[14] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. Cox, "Hyperopt: A python library for model selection and hyperparameter optimization," *Computational Science Discovery*, vol. 8, p. 014008, 07 2015.

[15] P. Panov, S. Džeroski, and L. Soldatova, "Ontodm: An ontology of data mining," in *2008 IEEE International Conference on Data Mining Workshops*, 2008, pp. 752–760.

[16] G. C. Publio, D. Esteves, A. Ławrynowicz, P. Panov, L. Soldatova, T. Soru, J. Vanschoren, and H. Zafar, "Ml-schema: Exposing the semantics of machine learning with schemas and ontologies," 2018. [Online]. Available: https://arxiv.org/abs/1807.05351

[17] J. Kietz, F. Serban, A. Bernstein, and S. Fischer, "Data mining workflow templates for intelligent discovery assistance in rapidminer," 09 2010.

[18] C. M. Keet, A. Ławrynowicz, C. d'Amato, A. Kalousis, P. Nguyen, R. Palma, R. Stevens, and M. Hilario, "The data mining optimization ontology," *Journal of Web Semantics*, vol. 32, pp. 43–53, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570826815000025

[19] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. Parameswaran, "H <scp>elix</scp>: accelerating human-in-the-loop machine learning," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, p. 1958–1961, Aug. 2018. [Online]. Available: http://dx.doi.org/10.14778/3229863.3236234

[20] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht, "Keystoneml: Optimizing pipelines for large-scale advanced analytics," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, 2017, pp. 535–546.

[21] M. Bilal, G. Ali, M. W. Iqbal, M. Anwar, M. S. A. Malik, and R. A. Kadir, "Auto-prep: Efficient and automated data preprocessing pipeline," *IEEE Access*, vol. 10, pp. 107 764–107 784, 2022.

[22] B. Bilalli, A. Abelló, T. Aluja-Banet, and R. Wrembel, "PRESISTANT: learning based assistant for data pre-processing," *CoRR*, vol. abs/1803.01024, 2018. [Online]. Available: http://arxiv.org/abs/1803.01024

[23] G. Pons, "Tfm: Third draft," Master's Thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2023.

[24] S. Borgo, R. Ferrario, A. Gangemi, N. Guarino, C. Masolo, D. Porello, E. M. Sanfilippo, and L. Vieu, "Dolce: A descriptive ontology for linguistic and cognitive engineering1," *Applied Ontology*, vol. 17, no. 1, p. 45–69, Mar. 2022. [Online]. Available: http://dx.doi.org/10.3233/AO-210259

[25] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of sparql," *ACM Trans. Database Syst.*, vol. 34, no. 3, Sep. 2009. [Online]. Available: https://doi.org/10.1145/1567274.1567278

[26] A. Burns, C. Xiong, S. Franconeri, A. Cairo, and N. Mahyar, "How to evaluate data visualizations across different levels of understanding," 2020. [Online]. Available: https://arxiv.org/abs/2009.01747

Acknowledgments

I would like to express my heartfelt gratitude to DTIM at UPC for giving me the opportunity to develop my master's thesis under their guidance. I want to specifically thank Dr. Sergi Nadal and Dr. Petar Jovanovic for providing me with their exceptional guidance and continuous support throughout the whole thesis period. I also want to thank Dr. Massimilano De Leoni from UniPD for his valuable advice and feedback as it has positively impacted the quality of this thesis.

I would like to also thank the BDMA program for giving me the opportunity to embark on this challenging, yet rewarding academic journey. I want to thank all the people I had the chance to meet and learn from during my entire BDMA journey.

A

Workflow Generation Extension Guidelines

Table A.1: Table illustrating the engine-agnostic extension guidelines and references

Concept	**Description**	**Reference**
Task	Represents the data tasks possible to perform on a dataset. All of the tasks are defined in the CBOX of the ontology, so any extension of these tasks will be in the CBOX.	Refer to the CBOX generator here.
Algorithm	Represents the conceptual definition of possible solutions for each task and are directly connected to the Task instances they solve. They are also in the CBOX.	Refer to the CBOX generator here.

Concept	Description	Reference
Implementation	Represents the executable form of an algorithm. Each algorithm has at least on implementation instance that references it. The class Implementation has specialized subclasses such as Learner Implementation and Visualizer Implementation. The definition of subclasses is based on possible tasks, and it is performed in the TBOX. However, this is only an engine-agnostic extension of this concept.	Refer to the Implementation class definition here the TBOX generator here.
Input & Output Specifications	These two classes are part of the Data Specification concept and are responsible for defining the constrains of an implementation's inputs and outputs. These constraints are encoded as SHACL shapes and are defined in the CBOX and are referenced in an implementation definition.	Refer to the shapes defined in the CBOX here.
Component	An abstraction level of the implementations concept. Each implementation has at least one component; in the case of multiple components, each component performs the same task but using different methods. The concept Component also has similar subclasses to the Implementation subclasses. The definition of new Component subclasses is performed in the TBOX.	Refer to the definition of the implementation class here the TBOX generator here.
Transformation	A set of data annotation transformations expressed in SPARQL queries that are specific for each component. These are defined manually within each component instance definition. There are also the Copy Transformation and Load Transformation classes but for the specific tasks of obtaining the annotation of the input dataset and copying the input of a component as an output.	Refer to the definition of the Transformation classes here

Concept	**Description**	**Reference**
Rule	Represents domain knowledge used to select the components for preprocessing tasks. These are defined within the preprocessing component definitions.	Refer to the examples here.

Table A.2: Table illustrating the engine-specific extension guidelines and references

Concept	Description	Reference
Parameter	Represents the algorithm and execution engine configurations. An engine-specific subclass is created for this class to accommodate specific engine parameters that are vital for a successful engine translation process.	Refer to the definition of the Parameter class here and an example of an engine-specific subclass here.
Implementation	An engine-specific subclass is also required to include the specific implementation configurations. Each set of engine-specific implementations related to a certain algorithm is defined separately.	Refer to the definition of the Implementation class here and an example of an engine-specific subclass here. An example of engine-specific implementations for the SVM algorithm can be found here
Data Annotator	It is very crucial to verify that the datatype definitions are consistent between the data annotator used in the workflow generator and the specific execution engine targeted.	Refer to the following engine-specific data annotator here.
Additional Classes	More classes may need to be defined depending on the requirements of the specific execution engine targeted to ensure the successful engine-specific translation process.	Examples of engine-specific classes can be found here.