Abstract. Over the last years, knowledge discovery for Big Data has grown immensely and the requirement for automated user assistant techniques for the knowledge discovery have attracted more focus. Machine-processable metadata is needed in order to support the automation so that the automated systems are able to gain access to relevant information essential for the knowledge discovery processes. The automated systems apply to numerous mining algorithm and diverse data sources. None of the existing approaches proposes a flexible and extensible model of the knowledge discovery metadata artifacts that is reusable in various systems and for varied purposes as the metadata is typically stored in ad-hoc manners. Therefore, in this thesis, we propose a comprehensive, generic and extensible metamodel to enable automated intelligent discovery assistant. Aiming at semantic-awareness and the incorporation of external resources, we present the metamodel as an RDF formalization. Moreover, we provide a metadata repository where the user can access, manipulate and explore the metadata. Finally, we discuss the benefits of the approach, and present directions for the future work.

Keywords: Metadata; Metamodel; Knowledge Discovery; Data Mining; RDF; Triple Store; Metadata Management Tool

1 Introduction

Nowadays, the emergence of Big Data has evolved from the fast development of networking, data storage and the data collection capacity. More data enables more extensive analysis. More extensive analysis leads to more confident decision making, and better decisions bring greater profit. Big Data concerns the large volume, variety and velocity of the data from heterogeneous and autonomous sources so that it becomes difficult to process these data using traditional data processing applications. Data mining is one of the most efficient approaches to analyze Big Data. Data mining is an analytical process designed to explore data, usually large volumes of data, in the search of consistent patterns or systematic relationships between variables. Its findings are then validated by applying the detected patterns to new subsets of data. Data mining process is a part of the knowledge discovery which also involves the data selection, preprocessing and evaluation processes. The need for automated knowledge discovery assistant has emerged recently. The idea is to build a system that advises users in all stages of a data analysis process. In order to perform data mining over Big Data, the main challenges are the feature extraction and feature selection from the raw data, which can be structured, semi-structured and unstructured, as well as to decide which mining techniques can and should be used in which contexts.

Conceptually, the metadata can be used to improve the performance of the data mining algorithm. Metadata is the data that describes other data. It summarizes basic information about the data and its main purpose is to facilitate the discovery of relevant information. Metadata can be used to guide the automated knowledge discovery assistant system. Dataset characteristics can be used to determine the appropriate data mining algorithm to be applied. Metadata about data mining operators is essential for knowing when to apply an operator. For instance, certain techniques can only handle continuous values. Data profiling information can enable user support during the preprocessing steps. Historical data is also intensely interesting to the data miner. It is historical data that provides the basis discovery of patterns, trends, relationships, associations, etc. The historical mining metadata can provide recommendations that are based on the similarities with previous successful cases. To enable the automation of the metadata processing, the
metadata needs to be managed and persistently stored so that it can be easily accessed. It should be available under a public domain license to enable its reuse. Moreover, it needs to be represented in a machine-readable format so that the system is able to process it. The metadata quality also needs to be managed.

The motivation of the thesis is the creation and exploitation of a metadata repository to support the automated advising systems which now mainly focuses on the knowledge discovery, and later on will expand to the query recommendation and other business intelligence areas. To automate the metadata processing, the reasoner mechanism must know all the relationships occur in the metadata model. Moreover, there are many existing proposals for metadata repositories (i.e., metadata models) and we would like to provide means to cross them to gain benefits in combining the assorted metadata models and accessing the metadata in those repositories. Thus, to overcome the diversity of data sources and specifications of various systems, an approach for correlating similar concepts is strongly needed. The concepts have to be represented at a high-enough abstraction level so that the various systems can use them for their own needs according to their specifications. In addition, to achieve semantic-awareness, Resource Description Framework (RDF) is considered as a based formalization of the metadata repository. RDF is widely applied in the Linked Data initiative. It can be used to infer the new knowledge that has not been explicitly stated. The main goal of this thesis is the research about the metadata management and the knowledge discovery related metadata artifacts, design of a generic and extensible metadata model as an RDF formalization, and the implementing of a metadata management tool for the knowledge discovery processes. The RDF representation provides semantics that enables machine processing of this metadata. The main focus is on how to store and manage the metadata in an efficient and flexible way. We follow the Semantic Metamodel for Analytical Metadata (SM4AM) [22] as a model and extend its metamodel to support the knowledge discovery metadata artifacts.

Contributions. We propose a generic and extensible metamodel to support automated knowledge discovery activities and a prototype of the metadata management tool for the exploration of the metadata repository. Specifically, we first annotate all elements involved in the knowledge discovery process and semantically model them to enable the automation of such tasks by extending the SM4AM metamodel. Second, we deploy this semantic metamodel in the triple store repository where reasoning can be performed. Third, we provide the web service API as a set of basic functionalities to access and manipulate the metadata and explore it in a simple way. In addition, we implement a basic application for the visualization purpose, and it will be used to demonstrate the utilization. Note that the automation of the knowledge discovery processes including automated metadata extraction and metadata exploitation are out of the scope of this paper, and we introduce them only as examples of the potential usage for the further development.

The rest of the paper is organized as follows. We present the related work in Section 2. Next, we propose our solution including the needs of the universal metadata repository and clarifying the metadata artifacts for knowledge discovery in Section 3. Section 4 presents the metamodel formalisation, and the tool architecture design. Moreover, Section 5 explains the implementation details. Finally, Section 6 discusses the benefits of the approach, while Section 7 concludes the paper.

2 Related Work

This section summarizes the existing concepts and technologies that are typically used as basis for the metadata modeling and management in terms of knowledge discovery.

2.1 Metadata in Knowledge Discovery Context

As discussed in [21], metadata plays an important role in data mining. Metadata can be used to guide the data mining process. The data mining tool can consult the metadata repository and determine the types of queries to pose to the DBMS. Another approach of using metadata is to mine the metadata itself. The metadata might have more meaningful information than the data itself, and we can mine the metadata repository instead to discover patterns.

In order to annotate all elements involved in the knowledge discovery process, we first look at the metadata in data warehousing which is the most related usage in our case. The data warehouse
metadata management was first focused in the context of Foundations of Data Warehouse Quality (DWQ). As classified in [24], the concept of the DWQ approach are fundamentally based on the division of data and processes classified in a grid which is organized in three location levels and three perspectives. The location levels are divided into the source, data warehouse and client levels. The three perspectives are specifically the physical indicating the physical properties of the data, the logical (e.g., the database schema) and the conceptual which acts as a centralized, reference model of all submodels of the involved information systems and explains the role of each module. The metadata classification in this paper can be reused in the knowledge discovery context, especially the metadata concerning the source and the data warehouse location level, whereas the client level mostly aligns with the OLAP analysis which is out of the scope of the knowledge discovery aspect.

In addition, as proposed in [11], the metadata is organized into three major categories: business metadata, technical metadata and process execution metadata. There are several metadata aligned in this paper that can be useful in our project. The business metadata is used to represent the data sources information and the data terminology. The technical metadata represents the schema and preprocessing information, and the process metadata describes the preprocessing and the mining process performance and statistic.

According to the survey in [20], it has been stated that in an Intelligent Discovery Assistant (IDA), which has been developed to advise users in all stages of a data analysis process, the metadata is a significant source of prior knowledge supporting the expertized system. This metadata should be stored in a machine-readable format so that it can be extracted automatically and applied on new problems. Thus, deciding which prior knowledge to save and how to store it is the main challenge in designing an IDA. The prior knowledge types are categorized as follows:

1. Available Operators: the different existing operators that can be applied in the data mining process should be registered.
2. Metadata on the Input Dataset: the information about the input data is necessary in order to know which mining techniques can be applied. In Meta-Learning Systems, which are machine learning systems that discovers the relationship between the datasets characteristics and the algorithm performance, the relationship could be learned. The best model is selected to perform based on the characteristics of a new dataset. Different characteristics of the dataset have been defined in [3].
3. Metadata on Operators: the operator inputs, outputs, preconditions and effects (i.e., the operator properties) are essential for knowing when to apply an operator, for examples, some modeling techniques take tabular data as an input and generate a predictive model as an output. Certain operators can only handle categorical attributes (i.e., precondition), or some operators perform long execution time (i.e. effect). This metadata can also be used for checking the validity of a workflow and supporting automatic workflow generation via AI planning [10].
4. Predictive Models: the rules that suggest users what to do (e.g., handle missing value first). Expert Systems are systems centered around a knowledge base of expert rules defined by human experts to suggest useful techniques. The system asks the user questions for assessing which rules to apply.
5. Case Base: a set of successful prior data analysis workflows that is maintained so that they can be reused in similar cases. Case-Based Reasoning Systems use this prior data to provide advice based on previous successful cases according to their similarity to the new problem.

Furthermore, considering the dataset characteristics and quality, data profiling is another aspect of metadata that should be stored in order to reveal data patterns, errors and similarity. As explained in [14], data profiling is the process of analyzing the data validity in an existing data source and collecting statistics and information about that data. In the knowledge discovery perspective, it helps understand the data at hand and appropriately configure tools. Moreover, during the preprocessing, profiling results can be used to measure and monitor the general quality of a dataset, and conduct the transformation or mark as a violation.

Finally, we also follow the knowledge in [23] which proposes a comprehensive metadata framework to aid the user assistance activities. The framework is mainly focused on query recommendation assistance. Analytical Metadata consists of (i) schema representing the data model by means
of the multidimensional (MD) model, (ii) vocabulary defining business terms and their relationships which could be represented with an ontology, (iii) user characteristics capturing the explicitly stated information about the users, (iv) dataset characteristics, (v) queries, (vi) query logs, (vii) query sessions keeping evidence about how the data has been explored, (viii) traceability metadata capturing the origin of data and explanations about how the data has been transformed, (ix) user preferences, and finally (x) data usage statistics indicating how many times the data has been used.

2.2 Metadata Management and Tools

The main challenges of metadata management are highlighted in [26] which outlines the essentials of the metadata management for the metadata technology. It has been stated that we need to ensure that metadata is persistently kept over time, stored where it can be easily accessed and indexed to provide availability. Furthermore, the quality of the metadata directly affects the trustability and searchability. The metadata should be accurate and all relevant information about the resource should be captured. It should conform to a specific metadata standard and should not contain contradictions. Finally, it should be made available in a properly machine-readable format and up-to-date. Some metadata can be captured by automatic processes, while other characteristics require a human contribution. However, this metadata can be automatically linked to reference data published by trustworthy sources.

The authors of [24] discuss three layers of data warehouse metadata structure designed to represent levels of abstraction. The lowest layer presents the real life data warehouse environment processes (e.g., data stores, ETL, OLAP). The middle layer is an abstraction of the way the data warehouse environment is structured (i.e., model/metadata level). Lastly, the upper layer represents the formalism for expressing the contents of the repository (i.e., metamodel level). This concept of metadata repository structure layer motivated our research for metadata modelling to overcome the heterogeneity of various systems.

Moreover, there are several metadata repository tools which can be used as guidelines to implement a metadata management tool. Pentaho Metadata Editor [18] is a tool that builds Pentaho metadata domain and models. The Pentaho metadata model maps the physical structure of the concrete data source into a logical business model. The metadata are stored in a centralized metadata repository. The tools allow the users to enrich their data by applying metadata properties and concepts (i.e., a collection of metadata properties). The ability to let the users create new metadata concepts by their own and add them to the repository is the main purpose of our project.

Oracle Metadata Management (OMM) [16] is a metadata repository which allows the users to interactively search and browse the metadata, and provides data lineage, impact analysis, semantic definition and semantic usage examination for any metadata asset within the catalog. OMM provides capabilities including data governance, metadata annotation (i.e., data glossary management), data lifecycle and administration. To analyze metadata, the users can visualize the diagram of a model, trace and analyze the data flow (i.e., connection definitions to data stores and physical transformation rules) and semantic lineage (i.e., detailing the relationships between conceptual, logical and physical model) of a metadata element. Overall, the OMM [16] tool provides capabilities that need to be considered for a metadata management tool.

Finally, Adaptive Metadata Manager [2] is a web-based solution that offers enhanced capabilities in the areas of data governance and metadata management. It helps capturing data definitions and lineage from diverse sources including data modeling tools, relational database, XML schemas, programming languages, business intelligence and Big Data framework. It provides the abilities to discover and manage inconsistencies in metadata from various data sources by using business terminology management and maps business concept to logical and physical schema for traceability. The users can analyze the impact of changing a data element on other data element. The tool is able to extract metadata from various areas using industry standards for metadata exchange and tool specific APIs as needed.

However, none of these tools allow the user to elevate the data domain semantics by linking with external ontologies, and since their repositories are not built following the RDF formalization, the metadata repositories provide limited exploitation possibilities due to the lack of semantic-awareness.
2.3 Metamodel Standard

In order to build a comprehensive, generic and extensible metadata model, one can follow the metamodel standard. Common Warehouse Metamodel [15] is a specification for metadata modeling and defines a standard for interchange of data warehouse and business intelligence metadata between various systems in distributed heterogeneous environments. Nevertheless, CWM is mostly focused on the metadata relating to the data warehouse. Since we focus on the metadata for the knowledge discovery purposes, we discover that there are some more metadata that need to be stored in order to support automated knowledge discovery system that CWM does not cover, for example, mining algorithm specification, dataset characteristics and data dictionary. CWM also cannot be easily extended by linking with external concepts. Moreover, our main goal is to design the metamodel as generic as possible in order to make it usable with any concrete system environment.

Another approach is to follow the idea from SM4AM [22] proposing a generic and extensible approach for defining and modeling metadata artifacts supporting the user assistance. To achieve semantic-awareness, SM4AM is created as an RDF formalization enabling machine-processing of these metadata. The metamodel level is used for the unified formalization of the metadata from different and heterogeneous systems which can use the metamodel to instantiate specific models for their needs. While the metamodel is common for different systems, the model and instance levels can vary depending on the particular system. It is interesting to point out that SM4AM uses the concepts of dictionary and type. In order to capture the possible methods or actions which are not explicitly listed, dictionary and type can be used to give the user additional detail about what kind of metadata they might need. Different systems may have different specifications and purposes. A dictionary defines the set of potential methods, algorithms or metadata types depending on the concrete system. Each method corresponds to its type and each type belongs to the dictionary. Then the users can specify the dictionaries needed for a specific system in the model level.

3 Our Approach

In this section, we subsequently discuss the approaches providing metadata management system (typically, for knowledge discovery), and primarily focusing on the knowledge discovery metadata artifacts used and their exploitation for supporting the automated knowledge discovery assistant.

3.1 Generic Metadata Repository for Knowledge Discovery

The main idea of the project is to build a metadata management system that is able to store the metadata in a generic way due to the fact that the repository will be used in various systems. Each system has its own characteristics and operates for a specific purpose, and high heterogeneity and diversity of data sources can be used in its knowledge discovery processes. Moreover, there are many existing mining algorithms which also have their own specification. The metadata model schema should be extensible in order to capture all of these specifications. Therefore, the idea of the modeling abstraction levels, where three levels of modeling are defined (i.e., metamodel, model and instance) as discussed in [22], is proposed. Designing the generic metamodel is the first priority of all the means. It needs to capture the generality that various systems specifications can be instantiated at the model level. The metamodel should be reasonably assured that it is complete, especially in terms of relationships.

3.2 Metadata Artifacts for Knowledge Discovery

In this section, we concisely annotate all metadata artifacts related to the knowledge discovery process. To clarify these artifacts we extend the Analytical Metadata taxonomy from [23] as illustrated in Figure 1 by following the fundamental metadata mentioned in section 2.1.

The definitional category keeps information about the meaning of data and activities involved in the system. Data characteristic provides information that gives knowledge relating to
The particular task underlying the dataset. Therefore, dataset characteristics can jointly affect the performance of a learning algorithm when it is applied to the particular task represented by the dataset [4]. The dataset characteristics can be used to determine the suitable mining algorithm to exploit the given dataset. Vocabulary could be provided to be a reference terminology where to map all gathered metadata leading to the data governance. It defines business terms and their relationships, and can be efficiently represented with an ontology. The vocabulary concept extensively covers user-defined glossary, external domain ontology, external mining ontology and the dictionary defined the set of potential methods and operations mentioned in Section 2.3. Algorithm specification is related to the semantics of the mining operators and refers to the metadata on the operator mentioned in [20]. Planning-Based Data Analysis Systems use AI planners to generate workflows which required additional metadata on operators. The Algorithm specification can be represented with an ontology as well. Lastly, storing dataset related schema can be helpful for the preprocessing, and it can be used to determine the relationship between variables correlated to the feature selection purpose.

The data quality category captures the currency, accuracy, validity or completeness of the data. One must provide adequate profiling metadata to help guide the preprocessing process. To tackle data quality from a technical point of view, we propose metadata profiling processes as explained in [14].

The navigational category keeps evidence about the knowledge discovery activities, how they explore data, and the data used and produced by the processes. The artifacts are classified as passive and active. Active metadata represents the knowledge discovery processes which use different inputs and produce one or several outputs (i.e., passive metadata). Passive metadata is the metadata about dataset, mining model and report, while active metadata captures the preprocessing, mining process and process workflow metadata. Most commonly a dataset corresponds to the contents of a single database table, or a single statistical data matrix, where every column of the table represents a particular variable, and each row corresponds to a given member of the data set in question. Typically, mining datasets contain nominal or ordinal features. The mining model is created by applying an algorithm to the data. It is a set of data, statistics, and patterns that can be applied to new data to generate predictions and make inferences about relationships. The report represents the result of the evaluation or the application, for example, performance vector, accuracy, lift chart and mean square error. Preprocessing is an important step in the knowledge discovery process that evaluate the input data to produce output that is used as an input to the data mining process. In general, data preprocessing includes data cleaning (missing value, noise and outliers and duplicated data), data transformation (normalization, aggregation, generalization and discretization) and data reduction (sampling, remove redundant attributes). Mining processes consist of modeling, model evaluation or model application steps [7]. Lastly, the knowledge dis-

---

**Fig. 1.** Knowledge Discovery Metadata Taxonomy
covery activities are composed of preprocessing tasks and mining processes coordinated as process workflow which can be used in Case-Based Reasoning Systems [20].

The **lineage** category covers metadata artifacts about the original source of data in the system and describes what actions have been performed over data. **Traceability metadata** tracks the information about data sources, targets and transformation operation. This information allows the data lineage to be preserved, by recording when and how it was derived, and where it came from. Keeping the **data source information** can be helpful for preprocessing and data traceability purposes as well. The data source metadata (e.g., creator, created date, physical location) can also be used to determine the validity and trustworthiness of the data provided.

Finally, the **rating** category captures the **performance** and the **statistics** of the processes which can be kept and used to enhance the mining algorithm. The **performance** artifact relates to the metadata about process execution time and throughput which can be used to capture which part of the system is causing a bottleneck. The **statistics** artifact presents the statistics on the results of running process itself, including measures such as total number of processed rows and rows rejected, raised error, exception handling and severity. Details of the most recent executions of processes can be recorded, identifying when they ran and whether they completed successfully. Detailed metadata information of prior operator executions is stored, used to estimate the operators performance based on similar cases [20].

### 3.3 Usage Flow

As illustrating in Figure 2, the metadata is being extracted from knowledge discovery processes and stored in the repository in the metamodel conformance format so that it can be processed and exploited to help enhance the subsequent mining tasks. When a new mining task needs to be performed on a given dataset, one can post a query such as giving the dataset characteristics, finding the suitable mining algorithm based on the algorithm specification, or finding the best algorithm that gives the best performance based on the previous mining case that performed on the similar dataset characteristics, or giving the dataset profiling, constructing the preprocessing workflow based on the previous similar case.

![Fig. 2. Metadata Usage Flow](image)

### 4 Knowledge Discovery Metadata Management Tool Design

This section presents the formalization of the artifact design used as a base for the implementation. The first subsection outlines the metamodeling capturing the knowledge discovery artifacts. The second subsection introduces the need for semantic-aware formalization. The last subsection presents the design of a system architecture to access and manage the metadata stored in the repository.
4.1 Metamodel

The metamodel is designed to capture all the knowledge discovery metadata artifacts mentioned in Section 3.2 either directly mapping an artifact to the metamodel element one-to-one, or indirectly retrieving an artifact information from more than one metamodel element. Even so it must allow the users to add any particular components that conform to the conceptualization as they prefer. The metamodel has been built by extending the SM4AM metamodel [22] to support the query recommendation and the knowledge discovery system, and can be extended to support other business intelligence areas afterwards. The ObjectModel Core metamodel, the Transformation package and Data Mining package concepts in CWM Specification [15], as well as the data mining ontology in [10], have been used as a guideline. As mention in [22], the central focus of the metamodel is the Evidence class that refers to pieces of evidence that are collected for the query recommendation and we further extend it to include the artifacts for the knowledge discovery also. The original Evidence is subcategorized to DataProperty and UserAction. We extend the SM4AM metamodel by adding Passive as another subclass of the Evidence to keep the information about the data used and produced by the knowledge discovery processes (i.e., datasets, features, mining models and reports), as well as Operator to keep the evidence about all the operator metadata including the mining activities. The metamodel classes and their relationships are presented in Figure 3. The complete metamodel elements explanation can be seen in Appendix A.

Fig. 3. A Metamodel for Knowledge Discovery

In order to demonstrate how the metamodel can be used, the concordant part of the metamodel along with examples of possible instantiations of model and instance levels is depicted with figures (see Figures 4 to 7). A model element can be instantiated as an instance of the metamodel elements, and an instance element can be directly instantiated the concrete model. We denote the class by putting in brackets behind the instance (i.e., Location (SourceAttribute) means that Location is an instance of SourceAttribute, and in case of the property, it indicates the super property). It is also important to point out that every metamodel element is related to its corresponding type that belongs to the dictionary, but these elements do not appear in the metamodel illustrated here for better comprehension. Every dictionary, including DomainOntology and MiningOntology, are subclasses of Dictionary. Moreover, every subclass of Evidence is related to DomainConcept served the semantics of the data, as well as Schema corresponds to a defined structure representing the schema.
4.1.1 DataSet Elements

**DataSet** is the representative of the input and output of the knowledge discovery processes relying on the **Passive** category. The dataset can be used to serve any data that has the structure in tabular format whose columns represent particular variables. Every **Passive** subcategory associates with the corresponding element **DataProperty** and **DataSource** which are linked by **hasProperty** and **hasSource** property respectively, and also inherits **hasDomainConcept** and **usesSchema** properties from the parent concept (i.e., Evidence). The **DataProperty** captures the information related to the particular characteristic of the dataset, and the **DataSource** capturing the information of a certain data source where all the data come from. Typically, all mining datasets contain **Feature** representing a logical attribute describing a domain of data to be used as input to data mining operations. Figure 4 illustrates the corresponding piece of the metamodel, and its potential model and instance level examples that we explain next. The other aspects of the passive metadata (i.e., mining model and report) will be explained in 4.1.3

**Example.** The model level box represents the potential extension of the metadata. **DBPediaOntology** and **CityConcept** are instances of the **DomainOntology** and **DomainConcept** respectively. It shows the idea of linking the data with the external ontology which in this case is with DBPedia Ontology\(^1\). **ExcelFile** is an instance of **DataSource**, and it has **hasLocation** and **hasOwner** properties, which both are instantiated properties of **hasSourceAttribute**, consecutively associated with **Location** and **Owner** instantiated from **SourceAttribute**. **AttrValueDataTable** represents a kind of dataset. **Feature** can be categorical (i.e., **Categorical**), numerical (i.e., **Numerical**), or both, depending on its usage. Furthermore, **Mean**, **NumRow**, **Category** and **LowerBound** exemplify instances of **DataProperty**. The instance level illustrates how concrete dataset instance (**DataSetInstance**) containing **City** and **Salary** features would be stored. **ParisResource** retrieved from DBPedia linked data is attached with **Paris** feature category for semantics purpose.

\(^1\) [http://mappings.dbpedia.org/server/ontology/classes/]
4.1.2 Preprocessing Elements

After introducing one kind of passive metadata, we now define the classes related to one kind of active area of the knowledge discovery metadata artifacts. The first piece of evidence that belongs to the Operator category is Preprocessing. It represents any operators that transform the raw data into a format that will be more effectively processed for the purpose of the user and prepare it for another operators procedure. Commonly it is used as a preliminary data mining practice. performAfter property defines the preceding operator composing the actions into an ordered list that represent the process workflow. The operator input data is associated with uses property, and the operator output data is associated with produces property. Every Operator can have OperatorAttribute connected with them to captures characteristics specific for the mining operation, as well as PerformUser capturing the user who performs the operator, and the performance and statistical evidences are recorded by TimeAttr and Statistical. Preprocessing related classes are correlated with the properties illustrated in Figure 5

**Example.** On the model level, MissingValueHandling is an example of possible Preprocessing extension. MissingValueHandling is characterized with Script and SourceReference attributes, and has TimeExecution and Success to measure the performance. The figure shows the instances of the real missing value handling process. PlaceMeanValue operation uses Salary1 as an input column and places the outcome in Salary2 target column. It is preceded by Normalization operator.

4.1.3 Mining Operator Elements

Another piece of evidence of Operator is Modeling, ModelEvaluation and ModelApplication. The Modeling captures processes through which a model is created to predict an outcome by applying data mining techniques on the data to discover the interesting patterns, whereas the ModelEvaluation apprehends a mining task that is used to check some aspect of the quality of a classification or approximation model attained from the modeling process, and finally the ModelApplication captures a task that computes the result of a mining model for prediction to new data. Each operator attaches with Parameter used to tune algorithms or to set mandatory values, and specifies how a mining feature is to be used or manipulated for a mining operation by setting with AttributeUsage. Figure 6 depicts related part of the metamodel with potential model and instance levels for this context.
Example. The model level defines a classification modeling operation (i.e., ClassificationModeling) that corresponds to parameters MinimalLeafSize and MinimalSplitSize used to control the behaviour of the operator, and each feature correlated to the operator can be assigned the weight for tuning the performance purpose. Special mining model classes can be created specific for the operators which in this case we exemplify with DecisionTreeModel representing any classification model created as a tree. The DecisionTreeModel is characterized by TreeHeight which is an instance of DataProperty. Likewise, the Report can be instantiated as LiftAnalysis, ConfusionMatrix and Accuracy used for specific purposes. The instances level represents the example of the metadata captured during the running mining operator starting from the DecisionTreeLearning which is a kind of the classification modeling operator. DecisionTreeLearning uses DataSet1 as an input with the weight assigned for the Salary feature, and sets the MinimalLeafSize and the MinimalSplitSize parameters as “21” and “8” respectively. Then it produces Model1 as an output which the model is continuously consumed by DecisionTreeTesting evaluation process and DecisionTreeApplying application process. The DecisionTreeTesting carries out CMInstance representing the confusion matrix, and LiftAnalysis representing the lift chart. Lastly, the DecisionTreeApplying accomplishes the result stored in ResultClass response feature, and computes the accuracy to measure the performance of the model.

4.1.4 Algorithm Class Elements While the class discussed by now clarify evidence about the mining activities and the data used and produced by the processes, several classes are used concerning the semantics of the mining operators. AlgorithmClass is the main class representing the algorithm specification. Algorithm specifications are specified using the AlgorithmAttribute. The AlgorithmClass can be defined by linking with the external mining ontology (i.e., MiningConcept and MiningOntology) to find a setting and semantic of a specific mining algorithm. It is important
to outline that Operator and AlgorithmClass serve different purposes. The Operator means to store the metadata collected from the physical running processes, while the AlgorithmClass denotes the specification of each algorithm which can be thought as the mining ontology. The AlgorithmClass related classes are correlated by the properties illustrated in Figure 7.

**Example.** As a model example we present the concept of metadata on mining operators (IOPE) specific to the algorithm. The operator metadata properties (i.e., the operators inputs, outputs, preconditions and effects) are captured by AlgoInput, AlgoOutput, AlgoPrecondition and AlgoEffect. The model can also capture the external mining ontology retrieved from Data Mining Work Flow (DMWF) ² (i.e., eProplan-ontology and ClassificationLearner). The instance level shows two approaches to define the specification of the algorithm. The first approach is to explicitly explore the instantiation of the ontology retrieved from the external linked data which in this case is RMDecisionTree from RapidMiner³ instantiating the ClassificationLearner. Alternatively the specification can be defined by directly assigning AlgoInput, AlgoOutput, AlgoPrecondition and AlgoEffect. AlgoPrecondition and AlgoEffect can be described as rules expressed in the Semantic Web Rule Language (SWRL)⁴. The main challenge here is to specify the type of input or output for the algorithms. Expressing a sentence like “the algorithm ClassificationAlgorithm specifies input class AttributeValueDataTable” became problematic. It would mean that a particular algorithm (ClassificationAlgorithm, which is in the instance level, specifies a particular type of input (AttributeValueDataTable), which is in the model level, but classes cannot be assigned as property values to instances in OWL. To tackle this problem, the special instance of AttributeValueDataTable must be created (i.e., AttributeValueDataTableInstance). In this way, we associated ClassificationAlgorithm with AttributeValueDataTableInstance [8]. The same solution can be applied with AlgoOutput. Please also note that AttributeValueDataTableInstance instantiates from two classes, AlgoInput and AttributeValueDataTable.

⁴ [http://www.w3.org/Submission/SWRL/](http://www.w3.org/Submission/SWRL/)
4.2 Towards Semantic-Aware Formalization

Since the main goal of the project is to be able to express the metamodel in a generic and flexible way in order to add new concepts easily, relational data structure is not an option as it is strict enforcement of data conformity and referential integrity work against the need to adapt the schema evolution. Furthermore, predefining all mining algorithms specification is extremely cumbersome task. Instead, the solution should provide a capability to link with an external ontology since the mining algorithm ontology might have already been published, likewise the vocabulary which can be linked to the existing domain ontology from the Linked Open Data. When focusing on the system aiming at flexibility and extensiveness, RDF arises as a good option. As discussed in [22] that RDF represents data as machine-readable so that the system becomes semantic-aware and able to automate the metadata processing, and it brings a good ratio between expressiveness and computational complexity. In addition, in RDF, schema and instances are kept together and evolve together. The semantics presented in the RDF can be exploited for providing advanced mining support functionality based on the automatic reasoning over RDF schema and data structures.

Thus, we decided to build the metadata repository based on the RDF formalization.

As explained in [1], the basic RDF block is the triple consisting of a subject, a predicate and an object. A subject represents a resource using an IRI (Internationalized Resource Identifier), while a predicate represents a binary relationship between subject and object, and an object can be a resource or a literal. An ontology is logical semantics that enables reasoning which could help to answer queries, discover relevant source, relate objects for data integration, map business and technical term, and detect inconsistencies or redundancies. Ontology consists of schema part and instance part. RDFS represents the RDF schema providing a data modeling vocabulary for RDF data. RDF and RDFS statements can be translated to First Order Logic (FOL) to enable the semantic reasoning possibilities and be used to infer non-explicit knowledge. While RDFS offers taxonomic relations, object relations and properties, OWL has a richer vocabulary and gives more facilities to define objects and their semantic relationships. With OWL it is possible to specify the cardinality of object relations, and also use logical operators in definitions (e.g., use union of classes as a range of relation). However, the cost of this additional expressiveness is significantly higher computational complexity.

![Fig. 8. A Metamodel for Knowledge Discovery as RDF Formalization](image-url)
From the implementation point of view, the direction of generating the model using RDF formalization is to instantiate the model element as an instance of the metamodel elements using rdf:type, and an instance element can directly instantiate the concrete model using rdf:type as well. rdf:type is used to state that a resource is an instance of a class and definitely split the conceptual layers between model and instance. The metamodel classes as RDF formalization and their relationships are presented in Figure 8. The metamodel elements for knowledge discovery belong to dm namespace. We use QB4OLAP\(^5\) to represent the schema. qb:DataStructure corresponds to a defined structure representing the multidimensional schema. If other data models need to be captured, QB4OLAP vocabulary should be exchanged with an alternative data model such as relational data.

4.3 Architecture Design

The overall architecture of the knowledge discovery metadata management tool is described in the diagram in Figure 9. Three modules are proposed. The repository is where we store the metadata, as well as the metamodel. According to the reasoning explained in Section 4.2, building the metadata repository based on RDF technology is the best fit with our requirements. A triple store is a framework used for storing and querying RDF data. It affords a mechanism for persistent storage and access of RDF graphs. Generally, RDF data is a directed labeled graph, thus it can be stored in the native graph database such as Neo4j\(^6\) as well. However, there is a main difference between the triple store and the graph database. Typically, triple store can infer new triples according to the formal RDF inference rules (e.g., Man is a subclass of Human. All Man instances have Male gender. If John has Male gender, then the inference mechanism can infer that John is Man). Such inference will not be available in a general graph database since the graph edges do not have any special semantics there. The most important feature is that the triple store supports SPARQL query language [25]. SPARQL is a semantic query language being able to retrieve and manipulate data stored in RDF format. It can be used to express queries across assorted data sources and allows for a query containing of triple patterns, conjunctions, disjunctions and optional patterns. The choice of the technology is explained in Section 5.3.

![Fig. 9. Architecture Design](image)

Then, the web service layer is provided as a metadata management tool where the metadata is manipulated and explored for the knowledge discovery purpose. It is implemented following the Model-View-Controller (MVC)\(^7\) software architectural pattern. The MVC pattern separates the different aspects of an application: model logic, business logic and user interface logic, and provides loose coupling between these elements. We can reuse the common business logic and other modules in different services without code duplication since the domain logic resides outside of the web service. The Model connects with the repository to manipulate the metadata inside.

\(^5\) [http://www.w3.org/TR/vocab-data-cube/](http://www.w3.org/TR/vocab-data-cube/)
\(^6\) [http://neo4j.com/](http://neo4j.com/)
\(^7\) [http://c2.com/cgi/wiki?ModelViewController](http://c2.com/cgi/wiki?ModelViewController)
the repository. The Controller processes user requests sent via RESTful APIs, builds appropriate models and sends them back to the view for rendering, and the View populates the model data and returns a response directly via RESTful API.

Lastly, the basic application is used for the demonstration and visualization purpose. The application will be implemented as a basic graphical user interface. In addition, the metadata is available through a SPARQL endpoint to enable its reuse and the external services can query the metadata repository directly via SPARQL endpoint protocol shipped to the triple store. SPARQL endpoint facilitates users to query a knowledge base via the SPARQL language. The result is typically returned in machine-processable formats. Therefore, the SPARQL endpoint is mostly considered as a machine-friendly interface towards a knowledge base.

5 Implementation

This section describes the implementation of the system architecture mentioned in the previous section. We introduce the useful functionalities of the tool and present the proper technologies chosen for each module. In addition, some challenges in the implementation are discussed along with the solution proposed. The implementation challenges can be seen in Appendix D.

5.1 User Roles

Authentication is the main restriction in our tool due to the fact that the tool will be provided to support many areas, and the repositories will be created specific to the needs. Therefore, the repositories have to manage a set of users who can access and manipulate the data resided. In OMM [16], each user of the system is assigned a role and access permissions granted to repositories. OMM has two basic configuration roles as Administrator which has all permissions, and Guest which by default has view permission to all Contents in the Repository. The access permission is divided to View, Update and Administer. The View has only read access, while the Update has the ability to set object properties, and the Administer can add and delete the object and perform other metadata management actions on model and instance object type. Accordingly, we will mimic the OMM by defining the default roles which are adjusted to be concordant with the RDF scheme as explained in table 2 in Appendix C. The authorization user is split into four roles: Viewer, Contributor, Admin and Super Admin. Only the Super Admin can create, access and manage every repository and assign the role for each user in each repository. The Admin can view and execute queries, as well as insert/delete triples in both model and instance level in the authorized repositories, while the Contributor can insert/delete triples in the instance level only. The Viewer can only view and execute query on the authorized repositories. Each repository will be assigned the users who can access it, and the authorized user will be granted the access permissions as View or Manipulate. Please note that assigning the user the higher permission than his/her role is not permitted. For example, User A has a role as Viewer. It is forbidden to assign to User A the ability to manipulate any repository.

5.2 Functionalities

5.2.1 Web Service APIs Functionalities The basic APIs are provided as follows: importing/exporting RDF file, adding/deleting triples, basic query, and repository and user management. Regarding OMM [16], we can harvest and import the model and metadata from external metadata repositories. Nevertheless, since the imported model and metadata must conform to our metamodel, we will not allow any none conformable model to be imported. The import function allows users to load a huge chunk of RDF triples at once and the model level can be created by this facility. A triple or a list of triples should be possibly added or deleted. The inserted triples must be validated and conformed to the model schema as well. Inserting instance elements into the model level is not allowed, and vice versa. Inserting an instance property that does not conform to the property’s domain and range specification is also forbidden. Basic queries allow users to search for a specific metadata or view a list of a specific elements, as well as to obtain the elements belonging to the specific namespace owing to the fact that the tool will support further range of
the BI system. More sophisticated queries can be executed via SPARQL endpoint provided by the triple store. Furthermore, since there is a need to build the metadata management system that is extensible and able to cover possibly every model expressing the specific cases where some cases are not explicitly known, the external ontology and resource from the Open Linked Data can be retrieved. The details of the web service API are shown in Table 3 in Appendix C. Besides, we provide the notification functionality to notify the subscriber of the changes made to the system since it is essential to accommodate the metadata exploitation mechanism if the metadata repository have been changed. There are many levels of subscription. Repository subscriber will be notified when any changes made to the repository. Namespace subscriber will be notified when changes are made to the subscribed namespace, and Instance subscriber will be notified only when changes are made to the instance level. Users are able to redefine SPARQL queries and save them in the system. Then, the user can execute the predefined query by exclusively passing the parameters associated with the query without having to rewrite the query repeatedly. The documentation of the APIs is published and accessible through APIs portal.

5.2.2 Application Functionalities

The basic application will offer functions supporting the demonstration, for instance, browsing metadata, exploring metadata information, adding/deleting metadata, visualizing data lineage, semantic definition and semantic usage, and terminology management. We can browse a list of metadata and can search for a particular one, as well as the ability to explore and modify the model. The tool also provides the terminology management to capture, define, maintain and implement the data glossary of terminology, data definitions, code sets, domains and validation rules. The data glossary extensively cover the data dictionary artifacts mentioned in Section 3.2. The glossary can be imported from external ontology. Data lineage helps to track information about data from source to destination along with the various processes and rules involved showing how the data is used along its journey. This knowledge about data can be helpful for a better data governance process, data quality, master data management and overall metadata management. Moreover, tracking the data lineage of the data source can lead to identifying systemic issues in data capture or errors in data transformation, which may lead to missing value handling. Knowing the source of a missing value will often guide what alleviation technique to use [12]. In the semantic definition scenario, one may wish to simply discover what are the meanings of these mining attributes, and one may wish to see the usage of the terminology in the architecture by viewing the semantic usage. The details of the application functionalities are shown in Table 4 in Appendix C.

5.3 Choice of Technology

Regarding the RDF storage comparison and selection, seven candidates were selected and compared in order to find the best solution suitable for our needs. The key point is that it must support the RDF semantic nature and have a high query performance in terms of fast retrieval and inference ability so that it can support the large amount of the triples since the repository might combine the knowledge from Linked Open Data in the future and expand to support the other business intelligence areas. For more details of selection and comparison see Appendix B. Concisely, OpenLink Virtuoso\(^9\) is selected as the best qualification among the other alternatives based on the essential criteria properly chosen to satisfy the requirements. According to Berlin SPARQL Benchmark (BSBM) [5] and Triple Store Evaluation Analysis Report [19], Virtuoso performs the best overall in the performance studies. It is generally fairly usable and has good functional capabilities. Moreover, Virtuoso also provides the Open Source Edition with unlimited storage size and a cloud-based license available using Amazon’s EC2 service. The Virtuoso server can be accessed through a number of client APIs. These include a native (JDBC) interface, a SPARQL endpoint interface and a Jena-based interface.

The tool is developed using Java as a core since it has been a very well-designed, mature platform and widely used. Besides, the Virtuoso Drivers for JDBC (i.e., Java database connectivity

\(^8\) https://anypoint.mulesoft.com/apiplatform/upc-3/#/portals/organizations/d19d2186-ca19-40b9-9b50-f3dd466a1a798/apis/22828/versions/24274

\(^9\) http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/
technology) are also available providing methods for querying and updating data in a database. Hence, the choice of the web service API framework was aimed at finding the one that was widely used for developing RESTful APIs in Java. The research showed that Spring Framework\(^\text{10}\), an open source application framework, is one of best in the market today. Spring Framework is very well documented, provides various tutorials and a strong community support through forums. Spring Framework provides many functionalities that make implementation easier, such as the HTTP message converter support that is used for automatic conversion of instances to JSON though the Jackson library. Finally, JavaServer Pages (JSP)\(^\text{11}\) and Asynchronous JavaScript And XML (AJAX) are used to implement the application and the graphical user interface.

5.4 RDF Validation

Virtuoso, as well as other triple stores, does not support the validation and this kind of validation is not part of the RDF model, and does not validate inserts against the schema. The solution is to implement our own validation by implementing a sort of parser listener that checks incoming triples whether the subject and the object are in the domain and range of the defined properties, and that they are not elements from different levels. Strictly speaking, a model (i.e., can be viewed as an ontology) should not contain any instances, because it is supposed to be a conceptualization of the domain. More detail of the validation can be seen in Appendix E. Note that not all possible invalid triples will be caught since the open world constraints placed on RDF languages make validation difficult and incomplete. In the open world, there could always be more information supplying properties or referents.

5.5 External Ontology and Resource Challenge

Most of the Linked Data provide the SPARQL endpoint through which we can query the data and the ontology such as DBPedia SPARQL endpoint\(^\text{12}\). The DBpedia ontology is based on OWL and forms the structural backbone of DBpedia describing classes and properties. We will not load the external triples into our repository. We rather link with the external ontology using owl:sameAs. An owl:sameAs statement indicates that two URI references refer to the same thing. It is often used in defining mappings between ontologies. Then we can construct the SPARQL query to retrieve the external ontology using SPARQL SERVICE statement as shown in Figure 10. Likewise, the local instances can be linked to any external resources.

```sql
SELECT * WHERE {
  ex:Plant owl:sameAs ?externalOnt .
  SERVICE <http://dbpedia.org/sparql> {
    ?externalOnt ?predicate ?object
  }
}
```

![Fig. 10. Example of SPARQL SERVICE statement](http://dbpedia.org/sparql)

6 Discussion

This section presents a use of the metadata management tool in the knowledge discovery context to help support the automation. Please note that the automation techniques are out of the scope of this paper. We introduce the example of the knowledge discovery metadata that can be stored in the repository in conjunction with the proof that all the metadata artifacts mentioned in Section 3.2 are perfectly kept and can be retrieved to provide the useful knowledge supporting the knowledge discovery process. The demonstration is presented in Appendix F. The metamodel

\(^{10}\) [http://projects.spring.io/spring-hateoas/](http://projects.spring.io/spring-hateoas/)


\(^{12}\) [http://dbpedia.org/sparql](http://dbpedia.org/sparql)
captures pieces of evidence for the representation of the metadata artifacts for knowledge discovery described in Section 3.2. In conclusion, all the passive metadata artifacts are defined by the passive related classes, whereas preprocessing, mining process, Process workflow and traceability metadata artifacts are captured and tracked by the operator related classes. For representation of the schema artifact we use QB4OLAP similar to [22]. Data characteristics and profiling metadata are covered in concrete data properties. Furthermore, all the data dictionaries and the domain ontologies represent the vocabulary, and the algorithm related class and the mining ontologies represent the algorithm specification. Data source information is captured by the data source classes. Finally, the operation properties record the performance and statistic artifacts.

Regarding the potential knowledge discovery process in the context of the actual running environment, the metadata is captured when the dataset is inserted into the system. These metadata can be used to determined the suitable mining algorithm to process on, or to keep the records of successful prior data analysis so they can be adapted and reused in situations when the dataset that has similar characteristics arrives. Furthermore, both preprocessing and mining operator metadata is captured when the process is performed. These metadata can be used for the automated IDA such as generating preprocessing workflow for the similar profiling of the derived dataset comparing with the previous cases or to generate the mining workflow based on previous successful cases according to their similarity to the new problem. The information retrieved from the mining ontology is very useful. It can be used to indicate which operator should be used to apply with a given dataset and help advising systems to build a system that advises users in all stages of a data analysis process, that is, essentially the provision of an automated intelligent discovery assistant. Furthermore, it helps to support the process of checking the precision of workflows, understanding the goals behind given workflows, inventory of AI planner generated workflow completions, storage, retrieval, adaptation and repair of previous workflows.

The examples of possible queries supporting the automation are as follows:

Finding the datasets that the missing value handling preprocessing process needs to perform on.

```
SELECT distinct ?dataset WHERE {
    ?submeta rdfs:subClassOf* dm:DataSet .
    ?model a ?submeta .
    ?submodel rdfs:subClassOf* ?model .
    ?dataset a ?submodel .
    ?dataset ex:hasNumMissingValue ?property .
    FILTER(xsd:integer(?property) > 0)
}
```

Given the type of the dataset, finding the suitable mining algorithm based on the algorithm input specification.

```
SELECT distinct ?algorithm WHERE {
    ex:dataset a ?datasetType .
    ?algoModel a dm:AlgorithmClass .
    ?algoSubModel rdfs:subClassOf* ?algoModel .
    ?algorithm a ?algoSubModel .
    ?algorithm ex:hasInput ?input .
    ?input a ?datasetType
}
```

Retrieving all the operators and its properties performed by the specific user.

```
SELECT distinct ?operator ?property ?value WHERE {
    values (?v) { (dm:hasOpAttribute)(dm:hasTimeAttr)(dm:hasStat) }
    ?by rdf:type dm:performBy .
}
```
There are some limitations currently carried by the RDF usage. In RDFS, rdfs:Class is an instance of itself. Thus, it presents the possibly infinite reference of classes. rdfs:Resource, which from the initial purpose is the superclass of all classes, plays role of a superclass and an instance of rdfs:Class at the same time as stated in [17]. Moreover, when it comes to the modeling, RDF brings confusion. RDF classes are neither instance creators since there are not constructor methods, nor templates since the instances do not automatically inherit the properties of their instantiated classes. Therefore, it can not guarantee that the users will follow the model schema and constraint. We are only able to provide the guidelines to try to maximize the implemented properties.

Even if the metamodeling brings great benefits, after the tool demonstration has conducted, we acknowledged that there are certain challenges and complications in the usage. The complexity occurs on the user side when they create the model and instance levels. We have to define more dissipate elements to make a distinction between the metamodel and model level, for example, we cannot use dm:uses property in the model and instance level since we would like to completely differentiate the levels. Thus, this leads to the creation of more properties (i.e., cx:input). More examples of the complication can be seen in Appendix G. Moreover, RDF does not have different abstraction levels since both instances and classes are actually stored together, and it does not have the ability to enforce the distinction between these levels. Thus, another solution has been discussed as an alternative modeling using RDF formalization. However, the latter approach does not conform to the metamodeling specification but conceptually the idea is similar. We have some fixed classes (i.e., the metamodel) and some classes that can be defined by the user (i.e., the model) which are the extension mechanism. Thus in this approach, the metamodel and the model level are merged. The classes defined by the users are extended as a subclass of the fixed classes using rdfs:subClassOf. It is possible that a fixed class can act as an extended class and an element in the instance level can directly instantiate the fixed classes without having to defined any additional extended classes. The details of the alternative approaches is explained in Appendix G. However, we do not continue the latter approach as it does not qualify the requirement since we would like to build a metadata repository that can be used as a standard for any business intelligence systems. The metamodel is to be used to enable better integration possibilities for heterogeneous systems. We would like to maximize the benefit in combining other metamodels of the metadata in other domains related the business intelligence areas. Moreover, a emphatic difference between the model and a metamodel is the goal of metamodeling to derive and manifest the underlying logical and semantically rich relationships. In the alternative approach, we have to define all possible elements and relationships covering all of the specifications from the heterogeneous systems so that the metadata can be exploited by the reasoner mechanism since the reasoner mechanism must know all the relationships occur in the metadata model. Besides, the model from the alternative approach can not be reused in other systems as the model is intensely specialized to the current system.

7 Conclusions and Future Work

The goal of the research described in this paper is to present an approach for organizing and implementing a metadata management tool for the knowledge discovery systems using the extension modeling mechanism based on the RDF formalization. The motivation for this research is to use the metadata technique to support the automated systems in the business intelligence related area. This paper proposes a generic and extensible knowledge discovery metamodel being able to capture all knowledge discovery related metadata in the single consolidated model in a semantic-aware format, and thereby can enable automation and allow various systems to gain access to relevant shared metadata. In addition, the metadata management tool is developed providing the ability to access, manipulate and explore the metadata.

As part of the future work in this research, we plan to extend incrementally the repository to support other business intelligence areas. The tool functionalities will be improved to support additional requirements such as the automated metadata extraction, and the reasoner mechanism will be built to explore novel knowledge possibilities. The automated systems will be developed using the metadata repository as one of the core mechanisms.
References

17. Pan, J.Z., Horrocks, I.: Metamodelling architecture of web ontology languages. In: The Emerging Semantic Web, Selected papers from the first Semantic web working symposium, Stanford University, California, USA, July 30 - August 1, 2001 (2001)
A Appendix : Metamodel Specification

### A.1 Classifiers

1. **Class AttributeUsage**
   - **Description**
     An AttributeUsage instance specifies how a mining Feature is to be used or manipulated for a mining operation [15].
   - **Superclasses**
     rdfs:Resource
   - **Possible Instantiated Model Element Example**
     - Weight: indicates the weight the algorithm should assign to an attribute.
   - **Associations**
     - relatedTo : Feature

2. **Class AlgorithmClass**
   - **Description**
     An AlgorithmClass instance represents a concept of metadata on mining operators (IOPE) specific to the algorithm to indicate which operator should be used to apply with a given dataset, and to support Planning-Based Data Analysis Systems [20]. It can be linked with the external mining ontology to find a setting and semantic of a specific mining algorithm.
   - **Superclasses**
     rdfs:Resource
   - **Possible Instantiated Model Element Example**
     - ClassificationLearning: describes the specifications of the algorithm. The classification learning algorithm can only handle categorical feature with missing value free and produces exactly one model and one categorical result. It must have minimal size for split, minimal leaf size and minimal gain as parameters [9].
   - **Associations**
     - hasAlgoAttribute : AlgorithmAttribute
     - hasMiningConcept : MiningConcept

3. **Class AlgorithmAttribute**
   - **Description**
     An AlgorithmAttribute instance defines an attribute specific to the algorithm.
   - **Superclasses**
     rdfs:Resource
   - **Possible Instantiated Model Element Example**
     - AlgorithmInput: defines an input restriction used by the algorithm.
     - AlgorithmOutput: defines an output restriction produces by the algorithm.
     - AlgorithmPrecondition: specifies restrictions on the algorithm input. For example, most data mining algorithms will only operate on data sets satisfying particular properties (e.g., having numerical or categorical attributes, or not containing any missing values). The algorithm preconditions are described as rules expressed in the Semantic Web Rule Language (SWRL) [9].

---

13 [http://www.w3.org/Submission/SWRL/](http://www.w3.org/Submission/SWRL/)
– AlgorithmEffect: defines the effect of the algorithm depending on parameter values and the algorithms input. For example, a discretization algorithms output is equal to its input with the exception of a set of columns, which are specified by a parameter. In the output data set, these columns value types will be changed to categorical. The algorithm effects are described as rules expressed in the Semantic Web Rule Language (SWRL\(^{14}\)) [9].

– AlgorithmParameter: defines a parameter restriction to the algorithm.

4. Class Continuous
   (a) Description
   A Continuous instance represents a feature that has a real number as an attribute value. Continuous attributes are typically represented as floating-point variables.

   (b) Superclasses
   Feature

   (c) Possible Instantiated Model Element Example
   – Interval: describes properties of the interval feature measured on a scale in which each position is equidistant from one another. This property allows for the distance between two pairs to be compared. Interval data cannot be multiplied or divided.

   – Ratio: describes properties of the ratio feature which can be compared as multiples of one another.

   – NormalizedColumn: describes feature column whose all values has been normalized.

5. Class DataProperty
   (a) Description
   A DataProperty instance represents the characteristic of Meta-features. The dataset provides information that could be able to give knowledge or strongly relate to the particular task underlying the dataset. Therefore, meta-features can jointly affect the performance of a learning algorithm when it is applied to the particular task represented by the dataset [20]. DataProperty can also be used to represent the data profiling guiding the data cleaning process. It reveals the data errors, measure and monitor the general quality of a dataset, and guide the transformation [14].

   (b) Superclasses
   Evidence

6. Class DataSet
   (a) Description
   A Dataset instance represents logical data which shows how physical data should be interpreted logically by the mining algorithm [15]. Typically, it corresponds to the data structure where every column of the table represents a particular variable, and each row corresponds to a given member of the data set.

   (b) Superclasses
   Passive

   (c) Possible Instantiated Model Element Example
   – MissingValueFreeDataTable: represents a dataset whose all columns are missing value free.

   – NominalDataTable: represents a dataset whose all columns are nominal.

\(^{14}\) http://www.w3.org/Submission/SWRL/
- TimeSeries: represents a sequence of data points, typically consisting of successive measurements made over a time interval.

(d) Associations
   - hasFeature : Feature

7. Class DataSource
   (a) Description
      A DataSource instance keeps the information of the physical data used in the mining process. Sources can be databases, files, open data sources, etc. It contains source attributes which define specific attributes of the data source. Keeping dataset's sources is significant for the validity of sources and data exploration [22]. For Big Data which data come from many external data sources, this information is essential for choosing the appropriate features to be mined. This information can help aiding in preprocessing as well.

(b) Superclasses
    rdfs:Resource

(c) Associations
    - hasSourceAttribute : SourceAttribute

8. Class Discrete
   (a) Description
      A Discrete instance describes a finite or countably infinite set of values

(b) Superclasses
    Feature

(c) Possible Instantiated Model Element Example
   - Categorical: describes properties of a categorical feature and list the specific categories that are recognized in the feature. A taxonomy can be specified to organize the categories into hierarchy [15].
   - Ordinal: describes a feature that has ordered category values.
   - Binary: describes a feature whose unit can take on only two possible states, traditionally termed 0 and 1 in accordance with the Boolean algebra.

9. Class DomainConcept
   (a) Description
      A concept, which is the main component of the ontology, represents a set or class of entities or things within a domain. A DomainConcept instance serves the semantics of the data. The concepts can be organized into taxonomies or associative relationships [6].

(b) Superclasses
    rdfs:Resource

(c) Possible Instantiated Model Element Example
   - Protein: defines a concept within the domain of molecular biology ontology
   - City: defines a concept within the domain of place ontology in DBPedia\(^{15}\)

10. Class DomainOntology
    (a) Description
        A DomainOntology instance represents the specification of conceptualisations which include a vocabulary of terms and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretations of terms [6]. The domain ontology can be retrieved from the external linked data ontology.

\(^{15}\)http://mappings.dbpedia.org/server/ontology/classes/
11. Class DSWithResponse
   (a) **Description**  
      DSWithResponse extends the DataSet concept meant to represent the dataset containing the response features from the mining process.

   (b) **Superclasses**  
      DataSet

   (c) **Associations**  
      – hasResponse : Response

12. Class Feature
   (a) **Description**  
      The logical attribute describes a domain of data to be used as input to data mining operations [15].

   (b) **Superclasses**  
      Passive

13. Class MiningConcept
   (a) **Description**  
      A MiningConcept instance represents a mining algorithm concept within a mining ontology.

   (b) **Superclasses**  
      rdfs:Resource

14. Class MiningOntology
   (a) **Description**  
      A MiningOntology instance represents the algorithm semantic and specification of the mining conceptualisations. The mining ontology can be retrieved from the external linked data ontology.

   (b) **Superclasses**  
      rdfs:Resource

   (c) **Associations**  
      – containMiningConcept : MiningConcept

15. Class MiningModel
   (a) **Description**  
      A MiningModel instance holds a result of the mining task. The mining model is created by applying an algorithm to the data. It is a set of data, statistics and patterns that can be applied to new data to generate predictions and make inferences about relationships [7].

   (b) **Superclasses**  
      Passive

   (c) **Possible Instantiated Model Element Example**  
      – DecisionTreeModel: predicts an outcome, and describes how different criteria affect that outcome.
16. Class ModelApplication
   (a) **Description**
   A ModelApplication instance describes a task that computes the result of a mining model to new data. The concept of the model application in predictive data mining refers to the application of a model for prediction to new data. The application phase can be as simple as generating a report or as complex as implementing a repeatable data mining process depending on the requirements [7].

   (b) **Superclasses**
   Operator

17. Class ModelEvaluation
   (a) **Description**
   A ModelEvaluation instance represents a mining task that is used to check some aspect of the quality of a classification or approximation model. The model evaluation aids to find the best model that represents the data and how properly the chosen model will work in the future [7].

   (b) **Superclasses**
   Operator

18. Class Modeling
   (a) **Description**
   Modeling is the process through which a model is created to predict an outcome by applying data mining techniques on the data to discover the patterns [7].

   (b) **Superclasses**
   Operator

   (c) **Possible Instantiated Model Element Example**
   - ClassificationModeling: describes the task of generalizing known structure to apply to new data.
   - RegressionModeling: expresses the task that attempts to find a function which models the data with the least error.
   - ClusteringModeling: defines the task of discovering groups and structures whose observations in the same cluster are similar.
   - AssociationRuleModeling: represents the task searching for relationships between variables and finding interesting associations amongst observations.

19. Class Schema
   (a) **Description**
   Schema expresses the data structure and relationship between its elements of the data source which is essential for the preprocessing process. It enables us to quickly identify and confirm relationships at a glance [11]. The multidimensional data model schema can be expressed using QB4OLAP 16.

   (b) **Superclasses**
   rdfs:Resource

20. Class Operator

16 http://www.w3.org/TR/vocab-data-cube/
(a) **Description**
An Operator instance represents a mining operation to be performed on a given data set [15]. Operators use different inputs (i.e., DataSet or MiningModel) and produce one or several outputs (i.e., DataSet, MiningModel or Report) [10].

(b) **Superclasses**
Evidence

(c) **Associations**
- hasParameter : Parameter
- uses : Passive
- produces : Passive
- setting : AttributeUsage
- hasOpAttribute : OperatorAttribute
- implement : AlgorithmClass
- performAfter : Operator
- hasStat : Statistical
- hasTimeAttr : TimeAttr
- performBy : PerformUser

21. Class OperatorAttribute
(a) **Description**
An OperatorAttribute instance means to keep attribute of the operator.

(b) **Superclasses**
rdfs:Resource

(c) **Possible Instantiated Model Element Example**
- Script: keeps the implementation of the operation.

22. Class Parameter
(a) **Description**
Parameters are used to tune the algorithms or to set compulsory values. The behaviour of each data mining operator is controlled by a set of parameters usually specified as key value-pairs for an operator. Such parameters can be quite simple (e.g., k-fold cross validation), quite complex (e.g., an SQL statement as a target expression for attribute construction), or sometimes even structured (e.g., a list of parameters to be optimized by an optimization operator) [9].

(b) **Superclasses**
rdfs:Resource

(c) **Possible Instantiated Model Element Example**
- MinimalSplitSize: sets the minimal size of a node in order to allow a split for the decision tree operator.
- LearningRate: sets the learning rate determining by how much we change the weights at each step for the neural network operator.

23. Class Passive
(a) **Description**  
This is an abstract class representing the input and output objects of the mining operators.

(b) **Superclasses**  
Evidence

(c) **Associations**  
- hasProperty : DataProperty
- hasSource : DataSource

24. Class PerformUser  
(a) **Description**  
A PerformUser instance captures evidence about the user who perform the operator used for traceability purpose.

(b) **Superclasses**  
rdfs:Resource

25. Class Preprocessing  
(a) **Description**  
Data preprocessing describes any type of processing performed on raw data to prepare it for another processing procedure. Generally, it is used as a preparatory data mining practice. Data preprocessing transforms the data into a format that will be more easily and effectively processed for the purpose of the user. Each preprocessing task can define preceding and succeeding tasks to coordinate the flow of control between tasks [15].

(b) **Superclasses**  
Operator

(c) **Possible Instantiated Model Element Example**  
- MissingValueHandling: describes the data cleaning operator dealing with missing values.
- Normalization: describes the data transformation operator dealing with parameters of different units and scales.

26. Class Report  
(a) **Description**  
The report represents the result of the evaluation or the application [9].

(b) **Superclasses**  
Passive

(c) **Possible Instantiated Model Element Example**  
- ConfusionMatrix: expresses a specific table layout that allows visualization of the performance of an algorithm.
- Accuracy: presents the proportion of the time that the predicted class equals the actual class, usually expressed as a percentage.
- LiftChart: represents the improvement that a mining model affords when compared against a random guess, and measures the variation in terms of a lift score.
- MeanSquareError: computed the average of the squares of the differences between the predicted and actual values.

27. Class Response
(a) **Description**
Response extends the Feature concept meant to represent the feature containing response values from the mining process.

(b) **Superclasses**
Feature

28. Class SourceAttribute
(a) **Description**
In [11], it has suggested that the Source Descriptive Information metadata should keep the information about the ownership description, business descriptions, update frequencies of original sources, legal restrictions on the use of each source, access methods and access rights privileges.

(b) **Superclasses**
rdfs:Resource

(c) **Possible Instantiated Model Element Example**
- Creator: records the creator of the data source.
- CreationDate: records the date that the data source was created.

29. Class Statistical
(a) **Description**
Details of the most recent executions of processes can be recorded, identifying when they ran and whether they completed successfully. These classes allow the lineage of data to be preserved, by recording when and how it was derived, and where it came from [15]. Detailed metadata information of prior operator executions is stored, used to estimate the operators performance based on similar cases [20].

(b) **Superclasses**
rdfs:Resource

30. Class TimeAttr
(a) **Description**
A TimeAttribute instance defines the time parameter related to the operator (e.g., execution time). Process-execution metadata should be retained so trend analysis can be performed. This information can reveal bottlenecks in the process, and trending can expose portions of the dataset that lack the required scalability. Measures of data quality should also be trended [11].

(b) **Superclasses**
rdfs:Resource
Appendix : RDF Storage Comparison and Selection

B.1 Criteria Selection

B.1.1 Native RDF store  This criterion is the most important key features since the metadata, as well as the metamodel, is going to be stored in RDF format. We do not accept non-native RDF store since they require additional tasks to transform and store the data in RDF construction and do not inherit the RDF characteristics.

B.1.2 SPARQL support  Supporting SPARQL query language and SPARQL endpoint is also an essential feature due to the fact that the metadata repository need to be retrieved and explored in a semantically expressive way. Likewise, we do not accept the systems that do not support SPARQL.

B.1.3 License  The economical aspect of the solution is always an important factor if we consider as a part of an academic project. This criterion is expressed in terms of the software license. In case of open source solutions, the price for most services is free which is obviously preferred.

B.1.4 Storage size in Free Version  This term refers to the maximum number of RDF triples that can be stored in the repository. Since the requirement of the project is to be able to store the metadata to support the automated system which now mainly focus on the knowledge discovery, and later on expanding to the other business intelligence areas, the storage could grow extensively bigger in the future. Unlimited storage size is obviously preferred.

B.1.5 Query Performance  One of the most important requirement is to be able to explore the metadata repository efficiently. Thus, it is necessary to consider this criterion as the priority. In order to measure how efficient each system is in query performance, the values of low, medium and high are introduced. High performance is certainly preferred.

B.1.6 Throughput  Throughput is measured in term of query performance in the concurrency environment where multiple clients access to the repository simultaneously. Even though it is not considerably important in this stage, determining this factor when deploying the service in the cloud in the future can become valuable. Likewise, the criterion is divided into three categories: low, medium, high. High throughput is certainly preferred.

B.1.7 Migration  Migration is determined in the sense of transferring data or application from the current system to the new system. Providing an efficient way of migration is useful when we consider the project growth and would like to provide the service in the cloud, or we might find a better solution afterward. A ranking of easy, medium and difficult is introduced. Easy migration is desired, but the medium is also accepted. Solutions classified in this category as difficult is undesirable.

B.1.8 Usability and Support  Usability and support are considered in terms of ease of installation, ease of development, ease of administration, documentation and professional technical support. This criterion is helpful when deploying, learning and adjusting the solution to fit the needs and later it makes maintenance of the system easier. Similarly, three categories are introduced: easy, medium and difficult. The system that has difficult manageability and administration are not taken into account since we do not want to spend too much time on learning and configuration.
B.1.9 Cloud Service This criterion describes if product comes together with a build-in cloud service. Nevertheless, even if the solution does not provide this additional service, we always deploy or migrate the system to the external cloud server. Considered systems are divided into two categories: those that provide build-in cloud server (Yes) and those that do not (No). This criterion is not of high priority, and both Yes and No are accepted, however, Yes is obviously preferred.

B.1.10 Inferencing Support Inference ability is one of the most beneficial factors in RDF and OWL technology. This criterion describes if the system comes together with an embedded expert inferencing or rules engine support such as OWL-Lite\(^\text{17}\) and OWL2 RL\(^\text{18}\). However, this is an additional feature since simple RDF and RDFS rules can also provide basic inference. Considered systems are divided into three categories: those that provide fully inferencing support engine (Fully), those that provide partially support (Partially), and those that dont (None). Fully is certainly attractive, but it is also computationally expensive.

B.1.11 Load Performance Load performance is important in the case of importing a large amount of data at once. Although, this ability is not very necessary in the first stage since there is not much data to load, considering deploying in the real environment where the users can migrate their repository into our system could be extremely challenging. Candidate systems are divided, in terms of performance, into three categories: low, medium, high. Low performance is accepted, but with undesired.

B.1.12 Visualizer This criterion is measured if the system also provides the visualizer facilitating the project since the project is going to have an ability to visualize the data lineage and exploring the metadata and model. The systems that come with the visualizer are highly preferred.

B.1.13 Backup and Recovery This criterion is measured in terms of what the system provides as its backup mechanism. To classify candidate systems, three groups are defined: None, Manual and Automatic. None means the system does not support the backup and recovery. Manual means that the system requires custom scripting for backing up and Automatic means that the system provides a fully managed backup solution automatically. This criterion is desirable in the project since the stored data is essential. Thus we should have a strategy for capturing and restoring backups in the case of data loss events. It is not acceptable if the system does not provide the backup and recovery mechanism.

B.2 Candidates Selection

B.2.1 OpenLink Virtuoso At the core, Virtuoso\(^\text{19}\) is an object-relational SQL database handling RDF and other graph data, relational data, XML data and objects. OpenLink Virtuoso supports SPARQL embedded into SQL for querying RDF data stored in Virtuoso’s database. Virtuoso combines the flexibility of relational access through inheritance, run time data typing, late binding and identity-based access. It supports the key SOA concepts including creating composite applications from loosely coupled data, construction of web services from heterogeneous data sources, as well as integration, management and synchronization of business processes. Virtuoso can function as a web application server which leverages a messaging architecture using the HTTP protocol. It works equally well for REST applications. Virtuoso is also able to operate as a Linked Data Server, allowing data access to be abstracted from a conventional REST scenario into an abstract collection of data resources identified by a URI. Virtuoso comes with a web-based application called Virtuoso Conductor that provides an interface for the database management functionality typically performed by an administrator. It supervises the administration of both

\(^{17}\) http://www.w3.org/TR/owl-features/
\(^{18}\) http://www.w3.org/TR/rif-owl-rl/
\(^{19}\) http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/
internal databases and those accessed through the platform’s connection layer. Role security is also maintained from this interface.


B.2.2 Ontotext GraphDB GraphDB\(^\text{20}\) (former name 'OWLIM') is a software component for storing and manipulating huge volumes of RDF data. GraphDB is packaged as a Storage and Inference Layer (SAIL) for the Sesame OpenRDF framework\(^\text{21}\). It is made up of three distinct components: an RDF database, an inference engine that uses rules to infer new knowledge from explicit statements, and a query engine for accessing the explicit and implicit knowledge.

GraphDB has a focus on inferencing. They support many of the different types of inferencing such as owl-horst, owl-max (most of OWL-Lite with RDFS), owl2-rl-conf (OWL2 RL, does not support D-Entailment) and owl2-rl-reduced. Controlling access to a GraphDB repository and assigning user accounts to roles with specified access permissions is achieved with a combination of HTTP authentication and the Sesame server’s deployment descriptor.

GraphDB Lite is free, whereas GraphDB Standard and GraphDB Enterprise are commercially licensed. GraphDB Lite is an ”in memory” engine. GraphDB Standard is a robust standalone database engine. GraphDB Enterprise is a clustered version which offers horizontal scalability and failover support and other enterprise features.

B.2.3 Jena TDB Jena is an open source Semantic Web framework for Java. It offers an API to extract data from and write to RDF graphs. TDB\(^\text{22}\) is a component of Jena for RDF storage and query. It supports the full range of Jena APIs. TDB can be used as a high-performance RDF store on a single machine.

A TDB store can be accessed and manipulated with the provided command line scripts and via the Jena API. When accessed using transactions, a TDB dataset is shielded against corruption, unexpected process terminations and system crashes. Jena-security transparently prevents requests to the Graph or Model interface, estimates access restrictions and either allows or rejects the requests. The system is authentication agnostic and works with most authentication schemes.

B.2.4 Oracle Spatial and Graph Oracle Spatial and Graph\(^\text{23}\) supports a full range of geospatial data and analytics for land management and GIS, sales territory management, mobile location services, transportation, LiDAR analysis and location-enabled Business Intelligence. As part of Oracle Spatial and Graph, Oracle delivers advanced RDF Semantic Graph data management, querying and inferencing that are generally used in many applications ranging from linked open data platforms to semantic data integration and social network analysis.

Oracle integrates with Jena via an implementation of the Jena Model and Graph interfaces. High performance bulk loading with Oracle Database parallel and direct path loading and loading through Jena. It is proven scalability to over 54 billion triples (LUBM 200K benchmark) with scalability to the 8 petabyte limit of Oracle Database. It provides SPARQL and SQL parallel querying and updating of RDF graphs with SPARQL 1.1, SPARQL endpoint web services, Java APIs with open source Apache Jena and Sesame, SQL queries with embedded SPARQL graph patterns, SQL insert/update and triple-level security that meets the most powerful security conditions with Oracle Label Security.

Oracle Spatial and Graph is not included in Oracle Standard Edition. It is an option for Oracle Enterprise Edition, and must be licensed separately.

\(^{20}\)http://www.ontotext.com/products/ontotext-graphdb/
\(^{21}\)http://www.aduna-software.com/technology/sesame
\(^{22}\)https://jena.apache.org/documentation/tdb/
\(^{23}\)http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html
B.2.5 AllegroGraph

AllegroGraph\footnote{http://franz.com/agraph/allegrograph/} is a persistent graph database designing to store RDF triples. AllegroGraph enables to scale up to billions of quads by using efficient memory utilization in combination with disk-based storage while maintaining superior performance. AllegroGraph is designed for supreme loading speed and query speed.

Although AllegroGraph is a graph database, it was developed to meet W3C standards for the RDF, so it is properly considered an RDF Database. AllegroGraph provides a REST protocol architecture, essentially a superset of the Sesame HTTP Client. The ontological entailments are dynamically maintained by AllegroGraphs RDFS++ engine required for reasoning. The materialization of the graph is the pre-computation and storage of inferred triples so that future queries run more efficiently.

AllegroGraph provides a free version as well as Developer and Enterprise version. The difference between these versions is the limited number of triples that can be stored in the system.

B.2.6 Neo4j

Neo4j\footnote{http://neo4j.com/product/} is a highly scalable open source graph database implemented in Java, and comes with a web based administration tool that includes full transaction support and visual node-link graph explorer.

Neo4j provides high availability, high speed querying through traversals, declarative graph query language, and it can scale to billions of nodes and relationships. Neo4j achieves the Property Graph Model efficiently down to the storage level. Neo4j, comparing with other graph processing or in-memory libraries, provides full database characteristics including cluster support, ACID transaction compliance, runtime failover and making it suitable to use graph data in production scenarios.

Neo4j Community Edition is a free version ideal for smaller projects that do not require high levels of scaling or professional services and support, while Enterprise Edition comes with enterprise-grade availability, management, and scale-up and scale-out capabilities.

B.2.7 Apache Stanbol

Apache Stanbol\footnote{https://stanbol.apache.org/} is an open source modular software stack intended to extend traditional content management systems with semantic services. It provides an ability to extract semantics from the contents and store semantic information in an Apache Solr based document repository for faster document retrieval. Functionalities are provided as RESTful services returning results as RDF and JSON. It is implemented using a framework such as Apache Solr for semantic search, Apache Tika for plain text and metadata extraction, Apache Clerezza and Apache Jena as RDF and storage frameworks.

The content enhancement engines are Natural Language Processing (NLP) processing textual content, and Linking engines suggesting Entities by links to several Linked Data Sources or links to Entities managed by the Entity hubs. The ClerezzaYard and the SesameYard store the managed Entities within a TripleStore. Both are not very efficient for label based lookups as required by the Entity Linking engines of the Stanbol Enhancer.

The value in the Table 1 is based on the results from Berlin SPARQL Benchmark (BSBM) \cite{5} and some existing works which can be seen in \cite{13}, \cite{19}. Please note that even if Apache Stanbol uses Apache Jena TDB as the RDF storage, but natively it is a content-based storage which the ontologies is kept apart in Apache Jena TDB to support the semantic functionality. Since Apache Stanbol uses Apache Jena TDB as the RDF storage, the performance of the Apache Stanbol is considered the same as Jena TDB.

B.3 Choice Justification

In order to narrow down the selection between six provider systems, we choose the most important criteria considering the RDF specification requirements. In our project, we should pay special attention to native RDF storage, SPARQL support, and query performance criteria as mention in Section B.1. As one can see from Table 1, Virtuoso and GraphDB give the best answer in all those...
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Virtuoso</th>
<th>GraphDB</th>
<th>Jena TDB</th>
<th>Oracle Spatial and Graph</th>
<th>AllergroGraph</th>
<th>Neo4j</th>
<th>Apache Stanbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native RDF store</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SPARQL support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Query performance</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>License</td>
<td>Open source</td>
<td>Limited edition free</td>
<td>Open Source Licensed Software</td>
<td>Limited edition free</td>
<td>Open Source</td>
<td>Open source</td>
<td></td>
</tr>
<tr>
<td>Free version storage size</td>
<td>Unlimited</td>
<td>100 million</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>Unlimited</td>
<td>5 Million</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Throughput</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Usability &amp; Support</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
<td>Difficult</td>
<td>Easy</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Migration</td>
<td>Medium</td>
<td>Medium</td>
<td>Hard</td>
<td>Medium</td>
<td>Easy</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Cloud service</td>
<td>Yes (pay-by-the-hour)</td>
<td>No</td>
<td>Yes (pay-by-the-hour)</td>
<td>Yes (12 months trial)</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Inferencing support</td>
<td>Partially</td>
<td>Fully</td>
<td>Fully</td>
<td>Partially</td>
<td>Partially</td>
<td>None</td>
<td>Fully</td>
</tr>
<tr>
<td>Load performance visualizer</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

**Table 1. RDF Storage Comparison Grid**

criteria. Even though Neo4j has very high performance in almost every criterion as well, it does not support RDF nature and SPARQL query language which are considered as the first prior essential for the project. However, when considering the inferior essential factor such as the licensing and the free version storage size, Virtuoso is the best one. Moreover, Virtuoso also provides free cloud service and its performance based on [5] is slightly greater than GraphDB. It is also possible to upgrade the product edition easily, and possible to migrate the service into the cloud.
C Appendix : Knowledge Discovery Metadata Management Tool Functionalities

C.1 User Roles

Table 2 below shows different roles of the users. The authorization user is split into four roles: Viewer, Contributor, Admin and Super Admin. Only the Super Admin can create, access and manage every repository and assign the role for each user in each repository. The Admin can view and execute query, as well as insert/delete triples in both model and instance level in the authorized repositories, while the Contributor can insert/delete triples in the instance level only. The Viewer can only view and execute query on the authorized repositories.

<table>
<thead>
<tr>
<th>Role</th>
<th>View/Query</th>
<th>Edit Instance Level</th>
<th>Edit Model Level</th>
<th>Manage Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viewer</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Contributor</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Admin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
</tr>
<tr>
<td>Super Admin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2. User Roles

C.2 Web Service APIs Functionalities

Table 3 shows the basic APIs provided by the tool along with the authorized user roles who have permission to execute each function. The import function allows users to load a huge chunk of RDF triples at once. A triple or a list of triples should be possibly added or deleted. The inserted triples must be validated and conformed to the model schema. Basic queries allow users to search for a specific metadata or view a list of a specific elements, as well as obtain the elements belonging to the specific namespace. More sophisticated queries can be executed via SPARQL endpoint provided by the triple store. Furthermore, the external ontology and resource from the Open Linked Data can be retrieved. In addition, the notification functionality is also provided to notify the subscriber of the change made to the system. The users are able to predefine their own SPARQL queries and save in the system. Then, the user can execute the predefined query by exclusively passing the parameters associated with the query without having to rewrite the query repeatedly.
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Description</th>
<th>Authorized Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create new repository</td>
<td>Add another metadata repository separately from the existing one</td>
<td>Super Admin</td>
</tr>
<tr>
<td>Delete existing repository</td>
<td>Delete the repository and all of its content triples</td>
<td>Super Admin</td>
</tr>
<tr>
<td>Create user</td>
<td>Add new user to the system and assign his/her role</td>
<td>Super Admin</td>
</tr>
<tr>
<td>Delete user</td>
<td>Remove existing user from the system</td>
<td>Super Admin</td>
</tr>
<tr>
<td>Assign repository permission</td>
<td>Assign the user who can view or manipulate the specific repository.</td>
<td>Super Admin / Repository Admin</td>
</tr>
<tr>
<td>View users list</td>
<td>List all the users, their roles and information</td>
<td>Super Admin / Repository Admin</td>
</tr>
<tr>
<td>List all triples</td>
<td>List all triples in a specific repository.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Search triples</td>
<td>Search triples in a specific repository that match with the provided keyword.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Add triples</td>
<td>Insert a triple or a list of triples into the repository.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Delete triples</td>
<td>Remove a triple or a list of triples from the repository.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>View model hierarchy</td>
<td>Retrieve all model elements and their hierarchy in a specific repository.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>List all instances</td>
<td>List all instances of a specific model element in a specific repository.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>List all model element's predicates</td>
<td>Retrieve all hierarchical predicates of a specific model element in the repository.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>List all instance properties</td>
<td>Retrieve all properties of a specific instance in a specific repository.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Import RDF file</td>
<td>Import RDF file into a specific repository.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Export RDF file</td>
<td>Export triples of a specific repository into a file.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Retrieve external ontology</td>
<td>Retrieve the whole ontology from the external linked data endpoint.</td>
<td>Any</td>
</tr>
<tr>
<td>Retrieve external resource</td>
<td>Retrieve the specific resource and its properties from the external linked data.</td>
<td>Any</td>
</tr>
<tr>
<td>Subscribe</td>
<td>Subscribe to the repository to be notified when any change has been made.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Predefine SPARQL</td>
<td>Predefine SPARQL queries and store them in the system.</td>
<td>Super Admin / Admin</td>
</tr>
<tr>
<td>Execute predefined SPARQL</td>
<td>Execute the predefined query by exclusively passing the parameters associated with the query.</td>
<td>Authenticated Users</td>
</tr>
</tbody>
</table>

Table 3: Web Service APIs Triples Management Functionalities
## C.3 Application Functionalities

Table 4 shows the basic application functionalities. The users can browse a list of metadata and can search for a particular one, as well as the ability to explore and modify the model. The tool also provides the terminology management to capture, define, maintain and implement the data glossary of terminology, data definitions, code sets, domains and validation rules. The glossary can be imported from external ontology. Data lineage helps to track information about data from source to destination along with the various processes and rules involved showing how the data is used along its journey. In the semantic definition scenario, one may wish to simply discover what are the meanings of these mining attributes, and one may wish to see the usage of the terminology in the architecture by viewing the semantic usage.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Description</th>
<th>Authorized Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browse metadata as a tree</td>
<td>List all metadata model elements and their hierarchy in the repository as a tree. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>View metamodel as a graph</td>
<td>Generate a graph representing the whole metamodel and its relationship. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>View metadata as a graph</td>
<td>Generate a graph representing the specific metadata and its properties. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>View metadata information</td>
<td>See the specific metadata information and its properties. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Create metamodel subclass</td>
<td>Add new subclass of a specific metadata model element into a specific repository. This function allows the user to define his own extension. Only the Repository Admin who has permission to manipulate this repository is allowed.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Create metadata model instance</td>
<td>Add new instance of a specific metadata model element into a specific repository. Only the users who have permission to manipulate this repository are allowed.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Define new predicate</td>
<td>Add new predicate for a specific metadata model element into a specific repository. Only the users who have permission to manipulate this repository are allowed.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Add/edit/delete metadata property</td>
<td>Manage metadata triples. Only the users who have permission to manipulate this repository are allowed. Please note that the Contributor can modify triples in the instance level only.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Load external ontology</td>
<td>Retrieve the whole ontology from the external linked data endpoint and load it into the repository. Only the users who have permission to manipulate this repository are allowed.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Load external resource</td>
<td>Retrieve the specific resource and its properties from the external linked data endpoint and load it into the repository. Only the users who have permission to manipulate this repository are allowed.</td>
<td>Super Admin / Repository Admin / Contributor</td>
</tr>
<tr>
<td>Visualize data lineage</td>
<td>Navigates the source and transformation where the information comes from. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
<tr>
<td>Visualize semantic definition and usage</td>
<td>Discover what are the meanings of these mining attributes, and see the usage of the terminology in the architecture by viewing the semantic usage. Only the users who have permission to access this repository are allowed.</td>
<td>Authenticated Users</td>
</tr>
</tbody>
</table>

*Table 4. Application Functionalities*
D Appendix : Implementation Challenge

D.1 Metamodel, Model and Instance Level Separation Issue

On behalf of the RDF schema, the model and instance level are kept together. Hence, in order to store and retrieve each level separately, we introduce three graphs per repository to keep the metamodel, model, and the instance elements independently. Note that these graphs are not decisively separated. Bear in mind that the instances are declared themselves as types of the model elements and the SPARQL query is able to retrieve two joint graphs together as shown in Figure 11. It also greatly gives the benefit of the authorization since we can manage the authorization of these graphs independently.

\[
\text{SELECT } ?\text{model} ?\text{metamodel} \\
\text{from } <\text{metamodel graph}> \text{ from } <\text{model graph}> \text{ from } <\text{instance graph}> \\
\text{WHERE } \{ \\
\text{ex:dataset a } ?\text{model} . \\
?\text{model a } ?\text{metamodel} \\
\}\n\]

Fig. 11. Example of three joint graphs query

D.2 Data Lineage Challenge

We provided data lineage visualization by tracing back recursively, following the instantiated properties of `dm:uses` and `dm:produces` metamodel properties. To view a lineage of a specific data element, we query which operation produced this element, and retrieve the input of the operation. Then we recursively query the previous operations and their related data until the input of the operation is not an output of any operation. In addition, since the dataset consists of one or more features, and the features may come from different transformations, we also trace back every possible feature lineage.

D.3 Authentication Procedure

We implemented the HTTP Basic Authentication, where the authentication information is passed through the HTTP header. This technique is the simplest for enforcing access control because it does not use cookies, session identifier and login pages. Unfortunately, the basic authentication method does not provide protection for the credentials that are being transmitted. Therefore, it must be used over HTTPS with SSL/TLS protocol. Virtuoso has provided a set of security functions which can be easily executed as an SQL command. In order to set the user’s primary role, the command `USER_SET_OPTION (‘<username>', ‘PRIMARY_GROUP', '<role>')` is conducted. Typically, the default roles in Virtuoso are `SPARQL_SELECT` being capable to run only SPARQL SELECT command, `SPARQL_UPDATE` being capable to run any kind of SPARQL UPDATE command, and `dba` who has all permissions. The new role can be created via Virtuoso Conductor interface. A user assigned to the Contributor role is granted permission to write access to the instance level graph and read access to the model level graph.

D.4 Notification Mechanism

Java Message Service (JMS) technology has been offered to publish the notification. JMS allows the application to send and receive messages between two or more senders and receivers. On the receiver side, the listener method is implemented to intercept asynchronous incoming messages. Then when a change occurs, the system publishes a message to a topic, and it will automatically trigger the receiver’s listener method. It can be used with other programming language like Python, Javascript and Ruby.
E Appendix : RDF Validation

1. Subjects and objects of the inserted triples are not the elements from different levels. Strictly speaking, a model (i.e., can be viewed as an ontology) should not contain any instances, because it is supposed to be a conceptualization of the domain, and vice versa, except if the predicate is rdf:type.

   # Existing triples in the model level
   ex:Country rdf:type dm:DomainConcept.
   ex:City rdf:type dm:DomainConcept

   # Existing triples in the instance Level
   ex:Spain rdf:type ex:Country.
   ex:Barcelona rdf:type ex:City

   # Inserted Triples into the model level
   ex:Country ex:hasCity ex:Spain -- forbidden
   ex:Country ex:hasCity ex:City -- permitted
   ex:Barcelona ex:isCityOf ex:Spain -- forbidden

   # Inserted Triples into the instance level
   ex:Barcelona ex:isCityOf ex:Country -- forbidden
   ex:Barcelona ex:isCityOf dm:DomainConcept -- forbidden
   ex:Barcelona ex:isCityOf ex:Spain -- permitted
   ex:Country ex:hasCity ex:City -- forbidden

2. The properties of the metamodel can not be used in the model and instance level.

   # Metamodel Level
   dm:uses rdf:type rdf:Property, rdf:Class;
   rdfs:domain dm:Operator;
   rdfs:range dm:Passive

   # Inserted Triples into the instance level
   ex:PlaceMean dm:uses ex:Dataset1 -- forbidden

3. In the model level, the domain and range of new properties can not be the metamodel classes.

   # Inserted Triples into model level
   ex:hasOwner rdf:type rdf:Property, dm:hasSourceAttribute;
   rdfs:domain dm:DataSource; -- forbidden
   rdfs:range ex:Owner

4. In the model level, as a property is to be an instance of a meta property, its domain and range classes must be instances of the domain and range meta classes of the meta property.

   # Metamodel Level
   dm:hasSourceAttribute rdf:type rdf:Property, rdf:Class
   rdfs:domain dm:DataSource;
   rdfs:range dm:SourceAttribute

   # Existing triples in the model level
   ex:ExcelFile rdf:type dm:DataSource.
   ex:Categorical rdf:type dm:Discrete

   # Inserted Triples into model level
5. In the instance level, subjects and objects of the inserted triples are in the domain and range of the defined properties.

```nblast
ex:hasOwner rdf:type rdf:Property, dm:hasSourceAttribute ;
    rdfs:domain ex:ExcelFile ; -- permitted
    rdfs:range ex:Categorical -- forbidden
```

```nblast
# Existing triples in the model level
ex:input rdf:type dm:uses, rdf:Class ;
    rdfs:domain ex:MissingValueHandling ;
    rdfs:range ex:Numerical

# Existing triples in the instance level
ex:PlaceMean rdf:type ex:MissingValueHandling .
ex:DataSet1 rdf:type ex:DataSet .
ex:Salary rdf:type ex:Numerical

# Inserted Triples into the instance level
ex:PlaceMean ex:input ex:Dataset1 -- forbidden
ex:PlaceMean ex:input ex:Salary -- permitted
```
Appendix : Demonstration

First of all, we import the metamodel produced in N3\textsuperscript{27} format into the repository. The full content of the metamodel can be seen in Appendix H. We conceptually reproduce knowledge discovery process starting from capturing dataset metadata, performing preprocessing on the dataset, applying data mining techniques on the dataset to discover the interesting patterns, evaluating the generated patterns and make use of the knowledge obtained to take better decisions. The dataset used for exemplification is the Iris Data Set\textsuperscript{28} introduced by Sir Ronald Fisher in 1988. The dataset consists of 50 samples from each of three species of Iris plant (i.e., Iris Setosa, Iris Virginica and Iris Versicolor). Based on the combination of four features (i.e., the length and the width of the sepals and petals), a linear discriminant model was developed to distinguish the species from each other. In fact, the dataset is already cleaned and there is no need for additional transformations, but we assume that it has missing value and outlier and demands for normalization. The metamodel is capable to keep every metadata artifact related to the knowledge discovery without having to define any additional relationships. We are able to store the Iris dataset metadata using \textit{dm:DataSet} instance along with its features using \textit{dm:Feature} instance and data source information such as creator and creation date using \textit{dm:DataSource} instance. Furthermore, the dataset characteristics and the data profiling are also able to be expressed using \textit{sm4am:DataProperty} instance, for example, minimal value, maximal value, mean value, standard deviation, number of observations, class correlation, number of missing values and number of duplicated values. The physical dataset and its schema are kept in the QB4OLAP related classes. We manage to retrieve plant ontology from DBPedia and load it as an instance of \textit{dm:DomainOntology}, and then the Iris plant resource are retrieved and attached to the dataset using \textit{dm:hasDomainConcept} property to provide supplementary semantics of the dataset. Neural network algorithm is chosen to be applied on the dataset as a mining modeling operator. We extend the parameter element to support the specific parameters of the neural network operator, for example, number of hidden layers, hidden layer size, learning rate and momentum, and it is possible to link with the external mining ontology. We download the eProPlan Base Ontology\textsuperscript{29}, the eProPlan DMWF Ontology\textsuperscript{30} and the neural network part of the Rapid Miner Ontology\textsuperscript{31}, and import them into the repository. Afterward, we link our neural network algorithm with the neural network concept. Next, a missing value handling, an outlier handling and a normalization preprocessing operator are performed on the dataset consecutively. After that, the neural network modeling operator, modeling evaluation

\textsuperscript{27} \url{http://www.w3.org/TeamSubmission/n3/}
\textsuperscript{28} \url{https://archive.ics.uci.edu/ml/datasets/Iris}
\textsuperscript{29} \url{http://www.e-lico.org/ontologies/dmo/e-Lico-eProPlan-Core.owl}
\textsuperscript{30} \url{http://www.e-lico.org/ontologies/dmo/e-Lico-eProPlan-DMWF.owl}
\textsuperscript{31} \url{http://www.e-lico.org/public/dmo/e-Lico-eProPlan-Operators-RapidMiner.owl}
operator and model application operator are executed. These operator metadata are captured by \textit{dm:Preprocessing}, \textit{dm:Modeling}, \textit{dm:ModelEvaluation} and \textit{dm:ModelApplication} respectively. The operators are linked together via \textit{dm:performAfter} property formed the process workflow. Each operator instance has its own execution script, execution duration, throughput, number of rows rejected and other properties associated by \textit{dm:hasOpAttribute}, \textit{dm:hasStat} and \textit{dm:hasTimeAttr} properties, as well as \textit{dm:performBy} and \textit{ex:performedWhen} keeping the information about the user who performs the operation and the time the operation is operated for traceability purpose.

The neural network modeling operator uses the Iris dataset as an input and produces a predictive model covered by \textit{dm:MiningModel}. The modeling evaluation operator uses the Iris dataset and the predictive model as input and produces a lift chart and a confusion matrix covered by \textit{dm:Report}. Lastly, the model application operator also consumes the Iris dataset and the predictive model and produces a prediction result captured by \textit{dm:Response} along with the accuracy measure recorded by \textit{dm:Report}. The data lineage of the result (\textit{ex:class_result}) is visualized in Figure 12.
G Appendix: Alternative Modeling Approaches

In this approach, the metamodel and the model level are merged. We cannot call this approach as metamodeling, and hence we cannot extend the SM4AM metamodel as it is in the different levels and concepts. The Evidence and the DataProperty elements must belong to dm namespace, and we cannot include \texttt{sm4am:UserAction} in this approach. However, the idea is conceptually similar. We have some fixed classes (i.e., the metamodel) and some classes that can be defined by the user (i.e., the model). The fixed classes and their relationships are presented in Figure 13. The classes defined by the users are extended as a subclass of the fixed classes using \texttt{rdfs:subClassOf}. \texttt{rdfs:subClassOf} is used to declare that all the instances of one class are instances of another. With using \texttt{rdfs:subClassOf} we stay at the same abstraction level since a subclass is at the same level as its superclass.

Figure 14 illustrates the corresponding piece of fixed classes concerning the dataset, and its potential extended classes and instance level examples. \texttt{dbpedia-owl:Place} and \texttt{dbpedia-owl:City} are subclasses of the \texttt{dm:DomainOntology} and \texttt{dm:DomainConcept} respectively. It shows the idea of linking the data with the external ontology which in this case is with DBPedia Ontology\textsuperscript{32}. \texttt{ex:ExcelFile} is a subclass of \texttt{dm:DataSource}, and it has \texttt{ex:hasLocation} and \texttt{ex:hasOwner} properties, which both are subproperties of \texttt{dm:hasSourceAttribute}, consecutively associated with \texttt{ex:Location} and \texttt{ex:Owner} inheriting from \texttt{dm:SourceAttribute}. Furthermore, \texttt{ex:MeanSD}, \texttt{ex:StandardDeviation}, \texttt{ex:Category}, \texttt{ex:Taxonomy}, \texttt{ex:OrderType} and \texttt{ex:LowerBound} exemplify the extension of \texttt{dm:DataProperty}. Notice that the \texttt{ex:DataSetInstance} directly instantiates from the \texttt{dm:DataSet} since there is no separation between metamodel and model in the alternative approach, as well as \texttt{dm:hasSource} property which can be used in the instance level directly without having to define new property for the separation purpose.

\textsuperscript{32} [http://mappings.dbpedia.org/server/ontology/classes/]
Fig. 14. DataSet Elements

The exemplification of the preprocessing and mining operator are depicted with figure 15 and 16.

G.1 Complication of the Usage in the Metamodeling Approach

G.1.1 Number of Model Elements Creation  In the metamodeling approach, more dissipate elements need to be defined to make a distinction between the metamodel and model level. We need also a model element so that we can create instances. In contrast, it is possible that a fixed class in the alternative approach can act the same as a model element. Certain fixed classes already have their own meaning and an element in the instance level can directly instantiate the fixed class without having to define additional classes.

# Metamodel Level (Fixed Classes)
dm:DataSet rdf:type rdfs:Class

# Metamodeling Approach
# Model Level
ex:DataSetModel rdf:type dm:DataSet

# Instance Level
ex:DataSetInstance rdf:type ex:DataSetModel

# The Alternative Approach
# Instance Level
ex:DataSetInstance rdf:type dm:DataSet
G.1.2 Number of Property Elements Creation In metamodeling approach, since all elements including the properties in the metamodel level can not be used in the model and instance level, each property in the metamodel needs to be instantiated from both rdf:Property and rdfs:Class. This way, the property in the metamodel level are able to be instantiated as a class at the model level. Conversely, the other approach does not strict to this rule. The properties of the fixed classes can be used in any levels, and new properties can be defined by just using rdfs:subPropertyOf.
# Metamodel Level (Fixed Classes)
dm:uses rdf:type rdfs:Class, rdf:Property .
dm:hasSourceAttribute rdf:type rdfs:Class, rdf:Property

# Metamodeling Approach
# Model Level
ex:input rdf:type dm:uses, rdf:Property .
ex:hasOwner rdf:type dm:hasSourceAttribute, rdf:Property

# Instance Level
ex:DecisionTreeLearning ex:input ex:DataSetInstance .
ex:ExcelInstance ex:hasOwner ex:Varunya

# The Alternative Approach
# Extended Classes
ex:hasOwner rdfs:subPropertyOf dm:hasSourceAttribute

# Instance Level
ex:DecisionTreeLearning dm:uses ex:DataSetInstance .
ex:ExcelInstance ex:hasOwner ex:Varunya

G.1.3 Domain and Range of the Property  Owing to the fact that we can not use any metamodel elements in the model level in metamodeling approach, additional model elements must be created. Especially the case that we would like to express more than one domain elements of some properties, the superclass ought to be defined in the model level. In the other hand, we can just simply place the fixed classes as the domain directly in the the alternative approach.

# Metamodel Level (Fixed Classes)
dm:hasSourceAttribute rdf:type rdfs:Class, rdf:Property ;
rdfs:domain dm:DataSource ;
rdfs:range dm:SourceAttribute

# Metamodeling Approach
# Model Level
ex:Owner rdf:type dm:SourceAttribute .
ex:DataSourceModel rdf:type dm:DataSource .
ex:ExcelFile rdfs:subClassOf ex:DataSourceModel .
ex:RDFFile rdfs:subClassOf ex:DataSourceModel .
ex:hasOwner rdf:type rdfs:Class, rdf:Property ;
rdfs:domain ex:DataSourceModel ;
rdfs:range ex:Owner

# Instance Level
ex:ExcelInstance ex:hasOwner ex:Varunya

# The Alternative Approach
# Extended Classes
ex:Owner rdf:type dm:SourceAttribute .
ex:hasOwner rdfs:subPropertyOf dm:hasSourceAttribute ;
rdfs:domain dm:DataSource ;
rdfs:range ex:Owner

# Instance Level
ex:ExcelInstance ex:hasOwner ex:Varunya
Appendix: Imported Metamodel in N3 format

@prefix dm: <http://dm.com/>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns/>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema/>.
@prefix sm4am: <http://sm4am.com/>.
@prefix xml: <http://www.w3.org/XML/1998/namespace>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema/>.
@prefix qb: <http://purl.org/linked-data/cube/>.

dm:AttributeUsage dm:relatedTo dm:Feature ;
  rdf:type rdfs:Class .
dm:relatedTo rdfs:domain dm:AttributeUsage ;
  rdfs:range dm:Feature ;
  rdf:type rdf:Property ;
  rdf:type rdfs:Class .
dm:AlgorithmClass dm:hasAlgoAttribute dm:AlgorithmAttribute ;
  dm:hasMiningConcept dm:MiningConcept ;
  rdf:type rdfs:Class .
dm:hasAlgoAttribute rdfs:domain dm:AlgorithmClass ;
  rdfs:range dm:AlgorithmAttribute ;
  rdf:type rdf:Property ;
  rdf:type rdfs:Class .
dm:hasMiningConcept rdfs:domain dm:AlgorithmClass ;
  rdfs:range dm:MiningConcept ;
  rdfs:subPropertyOf dm:hasConcept ;
  rdf:type rdfs:Class .
dm:AlgorithmAttribute rdf:type rdfs:Class .
dm:Concept rdf:type rdfs:Class .
dm:Continuous rdfs:subClassOf dm:Feature .
sm4am:DataProperty rdfs:subClassOf sm4am:Evidence .
dm:DataSet dm:hasFeature dm:Feature ;
  rdfs:subClassOf dm:Passive .
dm:hasFeature rdfs:domain dm:DataSet ;
  rdfs:range dm:Feature ;
  rdf:type rdf:Property ;
  rdf:type rdfs:Class .
dm:DataSource dm:hasSourceAttribute dm:SourceAttribute ;
  rdf:type rdfs:Class .
dm:hasSourceAttribute rdfs:domain dm:DataSource ;
  rdfs:range dm:SourceAttribute ;
  rdf:type rdf:Property ;
  rdf:type rdfs:Class .
dm:Dictionary dm:containConcept dm:Concept ;
  rdf:type rdfs:Class .
dm:Discrete rdfs:subClassOf dm:Feature .
dm:DomainConcept rdfs:subClassOf dm:Concept .
dm:DomainOntology dm:containDomainConcept dm:DomainConcept ;
  rdfs:subClassOf dm:Dictionary .
dm:containDomainConcept rdfs:domain dm:DomainOntology ;
  rdfs:range dm:DomainConcept ;
  rdfs:subPropertyOf dm:containConcept ;
  rdf:type rdfs:Class .
dm:DSWithResponse dm:hasResponse dm:Response ;
  rdfs:subClassOf dm:DataSet .
dm:hasResponse rdfs:domain dm:DSWithResponse ;
  rdfs:range dm:Response ;
dm:uses rdfs:domain dm:Operator ;
   rdfs:range dm:Passive ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:performAfter rdfs:domain dm:Operator ;
   rdfs:range dm:TimeAttr ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:hasStat rdfs:domain dm:Operator ;
   rdfs:range dm:Statistical ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:hasTimeAttr rdfs:domain dm:Operator ;
   rdfs:range dm:TimeAttr ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:performBy rdfs:domain dm:Operator ;
   rdfs:range dm:PerformUser ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:OperatorAttribute rdf:type rdfs:Class .
dm:Parameter rdf:type rdfs:Class .
dm:Passive dm:hasProperty sm4am:DataProperty ;
   dm:hasSource dm:DataSource ;
   rdfs:subClassOf sm4am:Evidence .
dm:hasProperty rdfs:domain dm:Passive ;
   rdfs:range sm4am:DataProperty ;
   rdf:type rdfs:Property .
dm:hasSource rdfs:domain dm:Passive ;
   rdfs:range dm:DataSource ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:PerformUser rdf:type rdfs:Class .
dm:Preprocessing rdfs:subClassOf dm:Operator .
dm:Response rdfs:subClassOf dm:Feature .
dm:SourceAttribute rdf:type rdfs:Class .
dm:Statistical rdf:type rdfs:Class .
dm:TimeAttr rdf:type rdfs:Class .
dm:containConcept rdfs:domain dm:Dictionary ;
   rdfs:range dm:Concept ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .
dm:hasConcept rdfs:domain rdfs:Resource ;
   rdfs:range dm:Concept ;
   rdf:type rdfs:Property ;
   rdf:type rdfs:Class .