# SEMANTIC ANNOTATION OF MACHINE LEARNING AND DATA MINING ALGORITHMS

Lidija Jovanovska

**Master Thesis**
**Jožef Stefan International Postgraduate School**
**Ljubljana, Slovenia**

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL

Lidija Jovanovska

# SEMANTIC ANNOTATION OF MACHINE LEARNING AND DATA MINING ALGORITHMS

**Master Thesis**

SEMANTIČNO OZNAČEVANJE ALGORITMOV STROJNEGA UČENJA IN PODATKOVNEGA RUDARJENJA

**Magistrsko delo**

**Supervisor:** Asst. Prof. Dr. Pančе Panov

Ljubljana, Slovenia, October 2022

# Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. Dr. Panče Panov, for providing the much-needed support, knowledge, and mentorship throughout these past several years. You introduced me to the world of ontologies and meta-thinking about the ever-developing and complex fields of data mining and machine learning. Through working with you I learned a lot and grew not only as a researcher but as a person as well.

I want to thank the evaluation board of this thesis, Prof. Dr. Petra Kralj Novak and Dr. Dragi Kocev, for their invaluable feedback. Thank you for the thorough discussions which helped improve this thesis even further. What is more, I want to thank all the colleagues at the Department of Knowledge Technologies, who even though I had not had a chance to meet very personally due to the Covid-19 pandemic, seemed very helpful and proactive in the brief several months when we were actually able to sit down and exchange ideas.

Special gratitude goes to the Slovenian Research Agency (ARRS) for supporting the research presented in this thesis via the project Imperatrix (J2-9230) - Improving Reproducibility of Experiments and Reusability of Research Outputs in Complex Data Analysis.

A big thank you to my student colleagues with whom we shared many fun moments, in and out of academia. I want to thank my office-mate Stefan Popov with whom we shared our struggles and celebrated our achievements. A special shout-out goes to Bojan Evkoski, you are an infinite source of inspiration, and the best research partner I ever had.

I also want to thank my parents, Dragana and Veljo, and my brother Jovan for your selfless love and support. I would not be here today if it were not for everything you have done for me.

Last, but not least, I want to thank my close friends who helped me accomplish this. They are the ones who motivated me when everything seemed to make no sense. They are the reason I am now standing at the finish line of this chapter, ready to take on new challenges in my life.

# Abstract

As the amount of data that is collected and processed increases, the use of algorithms that are used to generate knowledge from the data, as well as to provide predictions, increases as well. The potential and aspirations of our data-driven world rest on the capabilities and the power of algorithms. Yet, even the most seasoned practitioners of machine learning and data science (ML and DM) have a hard time choosing which algorithm to use in a given research or application scenario. This task is even further hard to achieve because there are no public and comprehensive repositories of ML/DM algorithms, with most practitioners relying on multiple sources of information to comprehend a specific algorithm: i.e., scientific literature, web articles, or the documentation of an ML/DM software library. The lack of a centralized repository that contains comprehensive ML/DM algorithm information motivated the work described in this thesis.

In this thesis, we address the tasks of representation, semantic annotation, storage, and querying of algorithms in the domain of ML/DM. For improving the knowledge representation of ML/DM algorithms, we propose OntoDM-algorithms, an ontology extension of the OntoDM ontology of data mining. The extension includes several novel entities which serve to explicitly exhibit the complex structure of ML/DM algorithms. The developed ontology resource is further utilized to create an ontology-based annotation schema which is in turn used to annotate a selection of algorithms and populate the algorithm knowledge base.

Furthermore, we develop a web-based application to support semantic annotation, storage, and querying of ML/DM algorithms. The application offers two different user scenarios. The users can either use the annotation tool to manually annotate an ML/DM algorithm, or they can query the repository of previously annotated ML/DM algorithms. The query results can be exported in several formats and they can be further used as a regular tabular dataset. In addition, apart from following a manual approach to semantic annotation, we explored a methodology for semi-automatic annotation of ML/DM algorithms. The complete methodology and the findings are described and used to open new research questions in this domain.

The created repositories and the developed application are evaluated based on the FAIR guiding principles for data stewardship, the OBO Foundry principles, as well as in multiple annotation scenarios. The evaluation shows that the developed resources provide the needed infrastructure for annotating, storing, and querying ML/DM algorithms, while potential venues for improving the work are also presented.

# Povzetek

Ker se količina podatkov, ki se zbirajo in obdelujejo, povečuje, se povečuje tudi uporaba algoritmov, ki se uporabljajo za ustvarjanje znanja iz podatkov, pa tudi za zagotavljanje napovedi. Potencial in želje našega podatkovno vodenega sveta temeljijo na zmožnostih in moči algoritmov. Kljub temu imajo tudi najbolj izkušeni izvajalci strojnega učenja in podatkovnega rudarjenja (SU in PR) težave pri izbiri algoritma, ki ga bodo uporabili v danem raziskovalnem ali aplikacijskem scenariju. To nalogo je še dodatno težko doseči, ker ni javnih in celovitih repozitorijev algoritmov SU/PR, pri čemer se večina izvajalcev zanaša na več virov informacij, da bi razumeli določen algoritem: npr. znanstveno literaturo, spletne članke ali dokumentacijo o Knjižnici programske opreme SU/PR. Pomanjkanje centraliziranega repozitorija, ki bi vseboval izčrpne informacije o algoritmu SU/PR, je motiviralo delo, opisano v tem magistrskem delu.

V magistrski nalogi obravnavamo naloge zastopanja, semantične anotacije, shranjevanja in poizvedovanja algoritmov v domeni SU/PR. Za izboljšanje predstavitve znanja algoritmov SU/PR predlagamo OntoDM-algorithms, ontološko razširitev OntoDM ontologije podatkovnega rudarjenja. Razširitev vključuje več novih entitet, ki eksplicitno prikazujejo kompleksno strukturo algoritmov SU/PR. Razviti ontološki vir se nadalje uporabi za ustvarjanje sheme opomb, ki temelji na ontologiji, ki se nato uporablja za označevanje izbora algoritmov in polnjenje baze znanja o algoritmih.

Poleg tega razvijamo spletno aplikacijo za podporo semantičnega označevanja, shranjevanja in poizvedovanja algoritmov SU/PR. Aplikacija ponuja dva različna uporabniška scenarija. Uporabniki lahko uporabijo orodje za opombe za ročno komentiranje algoritma SU/PR ali pa poizvedujejo po repozitoriju predhodno označenih algoritmov SU/PR. Rezultate poizvedbe je mogoče izvoziti v več formatih in jih je mogoče nadalje uporabljati kot običajni tabelarični nabor podatkov. Poleg ročnega pristopa k semantičnemu označevanju smo raziskali tudi metodologijo za polavtomatsko označevanje algoritmov SU/PR. Celotna metodologija in ugotovitve so opisane in uporabljene za odpiranje novih raziskovalnih vprašanj na tem področju.

Ustvarjeni repozitoriji in razvita aplikacija so ovrednoteni na podlagi vodilnih načel FAIR za upravljanje podatkov, načel OBO Foundry, kot tudi v več scenarijih opomb. Ocena kaže, da razviti viri zagotavljajo potrebno infrastrukturo za označevanje, shranjevanje in poizvedovanje algoritmov ML/DM, predstavljena pa so tudi možna mesta za izboljšanje dela.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| JSI | Jožef Stefan Institute |
| IPS | International Postgraduate School |
| ML | Machine Learning |
| DM | Data Mining |
| IE | Information Extraction |
| NLP | Natural Language Processing |
| NER | Named Entity Recognition |
| GUI | Graphical User Interface |
| UI | User Interface |
| API | Application Programming Interface |
| RDF | Resource Description Framework |
| RDFS | Resource Description Framework Schema |
| OWL | Ontology Web Language |
| SPARQL | SPARQL Protocol and RDF Query Language |
| FAIR | Findable Accessible Interoperable Reproducible |

# Chapter 1

# Introduction

The volume of research publications on artificial intelligence (AI) has been growing steadily for the past 20 years. The AI Index, an organization that measures trends in AI annually produces the "AI Index Report", according to which, in the period from 2010 to 2021, the total number of AI publications doubled, growing from 162,444 in 2010 to 334,497 in 2021 (D. Zhang et al., 2022). A large part of these publications describes novel algorithms developed for different sub-domains of AI, such as machine learning (ML), computer vision (CV), and others. As a result, there is an increasing need for both better knowledge organization and knowledge representation of algorithms in the domain of AI.

From a meta-perspective, we can treat information about algorithms like any other data and apply well-defined practices in knowledge representation and engineering to develop a logical formalism of the domain of interest (Markman, 2013). The formalism, usually in the form of an ontology schema, would in turn allow us to semantically annotate any set of data. In computer and information science, ontology is a technical term denoting an artifact that is designed to enable the modeling of knowledge about some domain (Gruber, 1993). Ontologies provide a detailed description of a domain, first by organizing the domain classes in a taxonomy, and further by defining relations between the classes. Furthermore, the annotation process can be performed manually — through user input, or semi-automatically using natural language processing (NLP) models (Chowdhary, 2020) for tasks such as named entity recognition (NER) and relation extraction. Both approaches lead to the creation of a repository of algorithms that can be further used in various scenarios. Some of them include determining the choice of an algorithm, based on user-specific constraints (i.e., computational resources, data format, task), as well as semantic analysis of the repository data, and training AI models using the data.

In this thesis, we focus on a beginning-to-end approach to manual semantic annotation of algorithms in the domain of ML/DM by designing and implementing a system for semantic annotation, storage, and querying of machine learning/data mining (ML/DM) algorithms. To provide semantics to the data, we use controlled vocabularies in the form of ontologies, which formalize domain knowledge. While there exist several ontologies which we successfully reuse in our work, the need for more explicit knowledge representation led us to design an ontology extension named OntoDM-algorithms which extends the previous line of work (Panov et al., 2008), with a focus on the inner-workings of ML/DM algorithms. By annotating data with ontology-defined terms, we add machine processable meaning to the data. Based on the annotations, we can query the data, as well as infer new facts based on the knowledge that we have asserted.

It is important to note that this thesis has a "horizontal" structure in the sense that it explores both bottom-up and top-down approaches to semantic annotation, with an emphasis on the top-down approach described above. Hence, it leaves many open questions and much room for further work in the quest for knowledge representation and downstream ML tasks using ML/DM data.

## 1.1 Problem Description

This thesis is set to contribute to the field of knowledge representation of algorithms in the domain of ML/DM, with a focus on improving the reusability and accessibility of semantic annotations of algorithms. The aim is to formalize the knowledge about the algorithms in an ontology extension, following ontology design principles and best practices, and further develop an ontology-based annotation schema that will sufficiently encapsulate key ML/DM algorithm information. The next step is to develop a web-based system for manual semantic annotation, storage, and querying of ML/DM algorithms through which users, most commonly ML/DM practitioners and algorithm developers, can create annotations for any ML/DM algorithm, as well as query the algorithms repository. Additionally, the system ensures appropriate storage and validation of the data, which can be accessed publicly using a querying interface. Here, we will briefly present the domains of ML and DM and the main contributions of the thesis.

### 1.1.1 Knowledge representation of domain knowledge

Data mining, also called knowledge discovery in databases, in computer science, is the process of discovering interesting and useful patterns and relationships in large volumes of data. The field combines tools from statistics and AI with database management to analyze large digital collections, known as datasets (Clifton, 2022). On the other hand, machine learning is a field devoted to understanding and building methods that 'learn', i.e., methods that leverage data to improve performance on some set of tasks.

ML algorithms build a model on sample data, known as training data to make predictions or decisions without being explicitly programmed to do so (Mitchell et al., 1997). The abundance of research in these two domains has led to the development of many new algorithms. This in turn has driven the need for more explicit data description and organization, so that domain experts could more efficiently choose which algorithm to use in a given scenario. This choice is conditioned upon the task, the type of data available, as well as the computing resources at disposal. Having all these aspects in mind, we developed an extension of the OntoDM ontology, named OntoDM-algorithms, a resource focused on semantic representation of ML/DM algorithms, along with related representations closely tied to what an algorithm is.

### 1.1.2 Semantic annotation, storage, and querying

Once we have a controlled vocabulary that describes different aspects of ML/DM algorithms, then we can utilize it for semantically annotating algorithms. In this context, one of the key contributions of this thesis is a web-based application we designed and implemented that enables the semantic annotation, storage, and querying of ML/DM algorithms. The application infrastructure is divided into two functional units - namely, the semantic annotation tool and the semantic querying tool.

The semantic annotation tool enables users (i.e., researchers or domain practitioners), to input algorithm information which through the use of an ontology-based annotation schema is stored in a repository, in the form of a graph database ("NEO4J Graph Data Platform," 2022), and can be easily exported into a semantic format (i.e., RDF "RDF — Semantic Web Standards," 2022). The validated user annotations as well as a set of proof-of-concept annotations provided by us are then available to be retrieved by the user through the semantic querying tool.

There are multiple benefits from the creation of such a system. Firstly, it facilitates users' contribution to the population of a semantic repository through the annotation tool. Next, users can query and search the repository, enabling them to find the appropriate algorithm for their specific requirements, based on the task that is to be addressed, or the available computational resources. Additionally, data can be exported from the repository on demand using several common formats (CSV and PDF), as well as a standard model for data interchange on the web - RDF. Finally, through the logic endowed in the ontology schema, we can derive new information which is implicit and can be further inferred through the use of a reasoner.

### 1.1.3 Exploratory analysis of automatic annotation

The promising capabilities of large language models in various tasks, including named entity recognition (NER), inspired us to explore the capabilities of automatically populating a database of ML/DM algorithms, i.e., explore the capabilities of a bottom-up approach (Nadeau & Sekine, 2007). To do so we selected several text corpora describing ML/DM algorithms and used pre-trained language models from the Spacy and Scispacy libraries to automatically extract named entities (Neumann et al., 2019; Srinivasa-Desikan, 2018). Additionally, we trained a custom model from scratch, which proved the assumption that the task of NER in the domain of ML/DM is inherently hard due to the nature of the data. We point to findings during the annotation process of such data, as well as to potential improvements that can be applied to each step of the bottom-up approach.

## 1.2 Purpose, Goals, and Hypotheses

In this section, we present the purpose, hypotheses, goals, and contributions of the thesis.

### 1.2.1 Purpose

The purpose of this thesis is to develop a formal representation of ML/DM algorithms by designing and implementing an ontology extension and a knowledge base of ML/DM algorithms, as well as to provide services for populating and exploring the knowledge base.

### 1.2.2 Goals

Based on the main purpose of the thesis, we define the following operational goals:

- **G1.** Formalize knowledge about algorithms in the domain of ML/DM in an ontology resource.

- **G2.** Design and implement a web-based system for annotating, storing the annotations, and querying the stored annotated algorithms, where the developed system is used to populate and query the knowledge base.

- **G3.** Explore the possibilities of semi-automatic algorithm annotation by creating and populating a domain knowledge base from textual data.

### 1.2.3 Hypotheses

In this thesis, we are interested in the following research questions, formulated as hypotheses:

- **H1.** Knowledge about domain algorithms can be formalized in an ontology resource.

- **H2.** Semantic annotations of domain algorithms can be created using a manual annotation system and stored in a knowledge base.

- **H3.** A domain knowledge base can be automatically created and populated by using pre-trained natural language processing models to extract entities from structured text data.

### 1.2.4 Contributions of the thesis

This thesis is set to contribute to knowledge representation in the domains of ML/DM. This is done by providing an ontology schema and a system that allows users to semantically annotate and query domain algorithms. Additionally, a semi-automatic approach to semantic annotation is explored using natural language processing techniques. Hence, the contributions of the thesis are as follows:

- **SR1:** An ontology extension of the OntoDM-core ontology, named OntoDM-algorithms, for improved knowledge representation with respect to ML/DM algorithms;

- **SR2:** A web-based tool for manual semantic annotation, storage, and, querying of ML/DM algorithms;

- **SR3:** A populated domain knowledge base of algorithms; and

- **SR4:** An experimental study that demonstrates the use of a semi-automatic approach to create and populate a domain knowledge base;

## 1.3 Methodology

First, we will formalize the knowledge related to ML/DM algorithms in the form of an ontology. To do so, we follow the best practices in ontology design postulated by the OBO Foundry principles and the FAIR guidelines (Smith et al., 2007; Wilkinson et al., 2016). Initially, we will examine related state-of-the-art ontologies, such as OntoDM (Panov et al., 2008; Panov et al., 2013, 2014; Panov et al., 2016) and DMOP (Keet et al., 2015), which can be reused in the ontology extension, or equally importantly, serve as a reference point in the development process. The developed ontology extension will be publicly available with a permanent identifier and indexed on BioPortal. For developing the ontology extension, we use the Protégé open-source ontology editor (Musen, 2015).

We will further make use of the ontology extension by creating an ontology-based annotation schema. This annotation schema will be a key part of the web-based system for semantic annotation, storage, and querying of ML/DM algorithms. The system will enable users to annotate ML/DM algorithms using a user-friendly user interface, as well as query the repository using custom filters. For developing this system we will mainly rely on free and open-source technologies such as React ("React – A JavaScript library for building user interfaces," 2022) and related front-end technologies (used for building the user interface), Django (Forcier et al., 2008) (used for building the back-end services), and Neo4j ("NEO4J Graph Data Platform," 2022) (used for data storage and manipulation) and neosemantics (used for adding semantics to the data).

Then, we will use the developed tool to populate the knowledge base and provide example annotations using different types of ML/DM algorithms. The populated KB will enable the use of the querying tool without the presupposition of user-generated annotations, provide a guideline as to what an annotation is envisioned to contain, as well as validate the quality of the developed resources, i.e., the capability of the ontology to capture necessary information related to ML/DM algorithms and the capability of the system to successfully store such data. Ultimately, the result of this segment of the thesis will result in a populated KB as well as a discussion on the findings of the annotation process.

In the final section of the thesis, we will explore an automated approach to database population using language models. For this purpose, we select two text corpora in the domain of ML/DM: the SCIERC corpus of scientific paper abstracts (Luan et al., 2018) and a corpus of scientific paper abstracts published in the Arxiv repository (Sayak, 2021). We use the Arxiv abstracts to qualitatively evaluate the capabilities of large pre-trained language models included in the Spacy and Scispacy libraries. Since the results are not promising, we turn to train a custom model using the SCIERC dataset (Luan et al., 2018). The results of this approach will be presented accordingly. Additionally, we will manually annotate a small corpus of Arxiv abstracts to demonstrate the common annotation issues which make training language models extremely difficult.

## 1.4 Thesis Structure

In this chapter, we introduced the domain of the thesis, and we defined the purpose, goals, and hypotheses, as well as the contributions. Finally, we presented the proposed methodology with which we aim to achieve the goals and produce the specified scientific contributions.

In Chapter 2, we present the work relevant to the research presented in the thesis. We first present several notable algorithm conceptualizations, as per influential theoretical computer scientists, which served as reference points when developing the ontology. Next, we describe what an ontology is as well as common ontology engineering principles, and the two main approaches to developing an ontology. We also present several related ontologies in the domain of ML/DM and the common framework for representing algorithms in the ontologies. Finally, we describe the resources and the methodology used to carry out the semi-automatic KB population study.

Chapter 3 covers the design and implementation of an ontology extension for annotation ML/DM algorithms named OntoDM-algorithms. We first describe the relevant entities which have been reused from existing ontologies. Then, we delve deep into each of the novel entities, explaining the reasoning behind their introduction to the ontology extension and providing relevant examples. At the end of the chapter, we describe the operational aspects of the ontology development process.

In Chapter 4, we describe the design and implementation of a web-based application for annotating, storing, and querying ML/DM algorithms. We thoroughly describe the system architecture and the implementation of the user interface as well as the services for annotation and querying. The use of semantics in the application is also discussed, accompanied by guidelines on how it can be improved. In the final section of this chapter, we present the annotations which we have created using the developed resources as well as a discussion of the lessons learned in the manual annotation process.

Chapter 5 covers two use case scenarios — the annotation use case and the querying use case, accompanied by detailed depictions of the user interface. Additionally, we present annotations of two algorithm types, a single generalization and an ensemble algorithm, through which we illustrate some of the design features of the system. With these use cases, we showcase the capabilities of the system, from inputting and storing the data to querying the system for necessary information.

In Chapter 6, we evaluate the developed resources using the FAIR Guiding Principles for scientific data management and stewardship, the OBO Foundry Principles for ontology design, as well as present statistical ontology metrics of OntoDM-algorithms (Smith et al., 2007; Wilkinson et al., 2016).

Moving on to Chapter 7, we present the findings of the semi-automatic approach to database population using language models. We describe the data selection and annotation process in detail, as well as provide a discussion on the findings from the use of pre-trained and custom language models to automatically annotate paper abstracts on the topic of ML/DM algorithms.

Finally, in Chapter 8, we present our concluding remarks. We summarize the work, discuss the contributions of the thesis, as well as present venues for future contributions and improvement.

# Chapter 2

# Background and Related Work

The endeavor of semantic annotation of ML/DM algorithms is a task that relies heavily on extensive background research. This research draws upon decades of work done by theoretical computer scientists, linguists, and mathematicians, who were all trying to figure out what are the essential features of an algorithm. Using this background knowledge enables us to turn to established tools for knowledge organization in the form of an ontology. The field of ontology development will be described in detail, covering definitions, its philosophical origins, common approaches to development, as well as general principles of ontology design. Next, we put forward the most notable ontology resources developed in the fields of ML/DM and describe the framework which underlies their design. Finally, we describe the resources and methodology for an alternative approach for semi-automatic annotation.

## 2.1 Algorithm Conceptualizations

One of the goals of this thesis is to formalize knowledge about algorithms in the domains of ML and DM, hence we first have to be well acquainted with the way the term algorithm was conceptualized in previous work. One definition of an algorithm is: "algorithm is a clerical procedure which can be applied to any of a certain class of symbolic inputs and which will eventually yield, for each such input, a corresponding symbolic output" (Rogers Jr, 1987). Historically, there have been multiple efforts to formalize the term algorithm, which goes beyond the level of providing a linguistic definition, but rather precisely defines which entities are algorithms and which are not. These algorithm characterizations were made by scientists from various disciplines, not limited to mathematicians, linguists, and philosophers. Here, we describe three such characterizations made by the established theoretical computer scientists Hartley Rogers, Donald Knuth, and Michael Sipser.

### 2.1.1 Rogers' characterization

Professor Hartley Rogers in his 1967 Theory of Recursive Functions and Effective Computability characterizes an "algorithm" as a "clerical (i.e., deterministic) procedure applied to symbolic inputs and which will eventually yield, for each such input, a corresponding symbolic output" (Rogers Jr, 1987). He then describes the notion in approximate and intuitive terms as having ten features, five of which he asserts that most mathematicians would agree to, while the remaining five would be less obvious and consequently, less agreed upon. Rogers' features of an algorithm are presented in Table 2.1.

The last feature is somewhat controversial when bringing machine learning algorithms into the picture since most of them include procedures based on randomness, such as

Table 2.1: Rogers' ten features of an algorithm.

| # | Algorithm Feature |
|---|---|
| 1. | An algorithm is a set of instructions of finite size. |
| 2. | There is a capable computing agent. |
| 3. | There are facilities for making, storing, and retrieving steps in a computation. |
| 4. | Given 1. and 2. the agent computes in "discrete stepwise fashion" without use of continuous methods or analogue devices. |
| 5. | The computing agent carries the computation forward "without resorting to random methods or devices, like dice. (in a footnote Rogers wonders if 4. and 5. are the same). |
| 6. | No fixed bound on the size of the inputs. |
| 7. | No fixed bound on the size of the set of instructions. |
| 8. | No fixed bound on the amount of memory storage available. |
| 9. | A fixed finite bound on the capacity or ability of the computing agent. |
| 10. | A bound on the length of the computation. The only requirement is that a computation will terminate after some finite number of steps; not insisting on an a priori ability to estimate this number. |

random sampling procedures. However, this does not clash with the fourth feature, since a stochastic procedure is nonetheless carried out in a "step-wise" fashion.

### 2.1.2 Knuth's characterization

One of the most cited and widely accepted algorithm characterizations was provided by Donald Knuth, building on top of Hartley Rogers' list of algorithm features. Knuth stated five features that are widely accepted as requirements for an entity to be an algorithm (Donald et al., 1999), which can be seen in Table 2.2.

Table 2.2: Knuth's characterization of an algorithm.

| # | Algorithm Feature |
|---|---|
| 1. | **Finiteness.** An algorithm must always terminate after a finite number of steps. |
| 2. | **Definiteness.** Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. |
| 3. | **Input.** An algorithm has zero or more inputs: quantities that are given to it initially before the algorithm begins, or dynamically as the algorithm runs. |
| 4. | **Output.** An algorithm has one or more outputs: quantities that have a specified relation to the inputs. |
| 5. | **Effectiveness.** An algorithm is also generally expected to be effective, in the sense that its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper. |

While these requirements, echoed in the definition we presented at the beginning of this section, make intuitive sense, they include many vague terms, such as "precisely defined" or "sufficiently basic". Similarly, Stone defines an algorithm as: "A set of rules that precisely defines a sequence of operations such that each rule is effective and definite and such that the sequence terminates in a finite time (Stone, 1971)."

### 2.1.3 Sipser's three levels of description

Sipser's three levels of description of Turing machine algorithms interestingly correspond heavily to the way algorithms are represented in OntoDM-core, the ontology which we are to upgrade with this work and will be presented shortly afterward (Sipser, 1996). We will expand on this later, but for now, it is convenient to see that the high-level description corresponds to the specification layer, the implementation description corresponds to the implementation layer and the formal description corresponds to the execution layer.

- High-level description: "Wherein we use prose to describe an algorithm, ignoring the implementation details. At this level, we do not need to mention how the machine manages its tape or head."
- Implementation description: "In which we use prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level, we do not give details of states or transition function."
- Formal description: "The lowest, most detailed level of description that spells out in full the Turing machine's states, transition function, and so on."

## 2.2 Ontologies and Ontology Engineering Principles

The promise of computers to store, manage and integrate large amounts of data and information has led to an increasing need for knowledge representation and organization in a computer-readable format. This has led to the development of many knowledge organization systems, such as terminologies, taxonomies, and ontologies. Ultimately, ontologies have turned out to be an increasingly dominant strategy for the organization of scientific information in a computer-friendly form. Hence, they will be the focus of this section.

### 2.2.1 Ontology as a representational artifact

An *ontology* is a representational artifact, comprising a taxonomy as a proper part, whose representations are intended to designate some combination of universals, defined classes, and certain relations between them (Arp et al., 2015). This definition consists of multiple terms that need to be further described. A *taxonomy* is a hierarchy consisting of terms denoting entities linked by subtype relations. Here, a hierarchy resembles a graph-theoretic structure consisting of nodes and edges with a single top-most node (the "root") connected to all other nodes through unique branches. While hierarchies in general allow multiple inheritance, a taxonomy adheres to the concept of single inheritance, i.e., all nodes beneath the root have exactly one parent node. An *entity* can be anything that exists, including objects, processes, and qualities (Arp et al., 2015). As such, it can be anything that is sufficiently necessary to be included in the ontology, as determined by the ontology developer.

Entities in reality that are responsible for the structure, order, and regularity are designated as *universals*. Universals are repeatable in the sense that they can be instantiated by more than one object and at more than one time, whereas particulars are nonrepeatable: they can exist only in one place at any given time. As such, *particulars* are individual entities in reality restricted to particular times and places. Particulars instantiate universals, but they cannot themselves be instantiated. In virtue of instantiating the same universal, two particulars will be similar in certain corresponding respects. In our domain of interest, an example of a universal would be the ML/DM Algorithm entity, while a particular would be GaussianNB or DecisionTreeClassifier.

A *representation* is an entity (for example, a term, an idea, an image, a label, a description, an essay) that refers to some other entity or entities. This is an acknowledgment of the fact that an ontology is itself a mere representation, a potential entity in yet another representation. The perception of an entity is different from the entity itself, and hence we must make a distinction between an entity in an ontology and an entity that exists, separated from the subjective perception of the beholder. Note that a representation may be vague or ambiguous, and it may rest on error. An example here would be a memory of an algorithm formulation described in a paper, as opposed to the actual description of the algorithm in the paper.

An *artifact* is something that is deliberately designed by human beings to address a particular purpose, while a *representational artifact* is an artifact whose purpose is one of representation. Thus, a representational artifact is an artifact that has been designed and made to be about something and uses some public form or format. Examples include signs, books, drawings, maps, and databases. A simple representational artifact would be the drawing of the first version of the ontology described in this thesis. Before the ontology was drawn what existed were only cognitive representations of things I observed through my senses (i.e., reading papers, discussing). Upon the creation of the drawing, I created a representational artifact that exists independently of such cognitive representations and transforms them into something that is publicly observable.

Entities in an ontology are connected via relations, whose names are supposed to carry inherent semantics. Some examples of relations include the *is_ a* relation and the *has_ part* relation. Additionally, the relations can be further defined by adding cardinality restrictions as well as function characteristics (transitivity, reflexivity, symmetry, etc.). At the fundamental level, we distinguish three types of relations, based on the types of entities it connects:

1. **Universal-universal relations.** The paradigm example of a relation that holds between universals is the *is_ a* relation, as in protein molecule *is_ a* molecule, or `DecisionTreeClassifier` *is_ a* `flat classification algorithm`. The *is_ a* relation represents hierarchies of generality, i.e., more specific child universals stand in *is_ a* relation to more general, or parent universals.

2. **Universal-particular relations.** A paradigm example of a relation between a particular and a universal is the instantiates relation, as in Nikola Tesla *instantiates* human being, or CART *instantiates* ML/DM algorithm. All particulars stand in the instantiation relation to some universal, typically to several universals at different levels of generality – but universals themselves do not instantiate anything.

3. **Particular-particular relations.** A paradigm example of a relation holding between particulars is the *part_ of* relation. For example, John's left leg *part_ of* John, or texttttDecisionTreeClassifier *has_ part* `max_depth`.

### 2.2.2 Ontological realism

It is a widely accepted notion that we cannot know reality directly or know the things in reality as they are in themselves. But rather, we have access to reality only as it is mediated by our thoughts or concepts. Hence, our perceptions, thoughts, beliefs, and theories are most properly conceived as being about our constructions or projections, and only indirectly (if at all) about mind-independent entities in some external reality. This philosophical stance was perpetuated by Descartes, Locke, Berkley, and Kant, amongst others, as presented in Figure 2.1 (Berkeley, 1881; Descartes, 1999; Kant, 1908; Locke, 1847). It is important to note that the goal of ontology for the realist is not to describe the concepts in people's heads. Rather, ontology is an instrument of science, and the ontologist, like the scientist, is interested in terms or labels only insofar as they represent entities in reality. So, the goal of ontology is to describe and adequately represent those structures of reality that correspond to the general terms used by scientists.

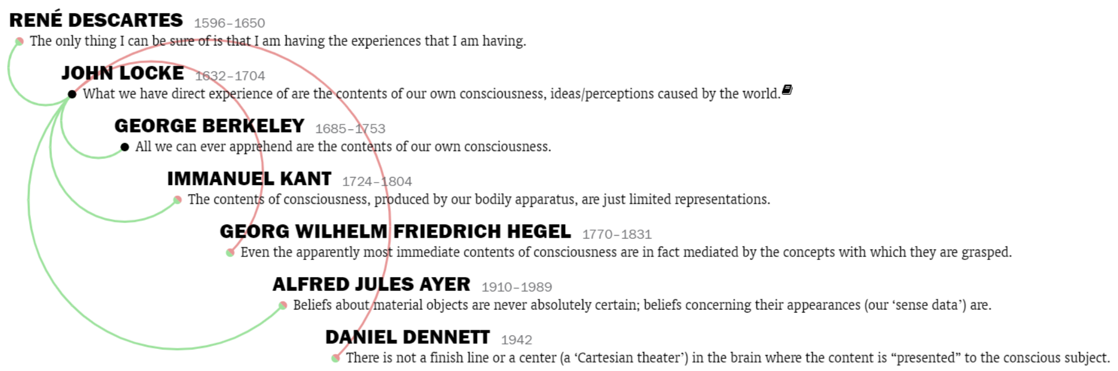

RENÉ DESCARTES 1596–1650
The only thing I can be sure of is that I am having the experiences that I am having.
JOHN LOCKE 1632–1704
What we have direct experience of are the contents of our own consciousness, ideas/perceptions caused by the world.
GEORGE BERKELEY 1685–1753
All we can ever apprehend are the contents of our own consciousness.
IMMANUEL KANT 1724–1804
The contents of consciousness, produced by our bodily apparatus, are just limited representations.
GEORG WILHELM FRIEDRICH HEGEL 1770–1831
Even the apparently most immediate contents of consciousness are in fact mediated by the concepts with which they are grasped.
ALFRED JULES AYER 1910–1989
Beliefs about material objects are never absolutely certain; beliefs concerning their appearances (our 'sense data') are.
DANIEL DENNETT 1942
There is not a finish line or a center (a 'Cartesian theater') in the brain where the content is "presented" to the conscious subject.

Figure 2.1: A trace of the idea of subjective realism, via expansions and refutations by philosophers. Source: https://www.denizcemonduygu.com/philo/browse/.

The most widely accepted view among philosophers today regarding the existence of an external world is the one of nonskeptical realism. Philosophical skepticism claims that we do not know propositions that we ordinarily think we do know (Comesaña & Klein, 2019). Nonskeptical, in this regard, means that we believe in the propositions, i.e., assume that they are true. In a survey of 1785 philosophy faculty members, 79.5% leaned towards nonskeptical realism, while very few opted for skepticism, idealism, and other (Bourget & Chalmers, 2020). Realism is the thesis that entities (which can be both physical things and concepts) have mind-independent existence, i.e. that they are not just a mere appearance in the eye of the beholder. Adapting this stance allows us to acknowledge that although our cognitive faculties do not deliver the entire truth about reality, this does not mean that the information that they do deliver should be viewed as nonrepresentative of how reality is.

Our faculties – our senses and cognition mechanisms – much like spectacles, microscopes, and telescopes – do indeed provide us with information about reality. They do this discretely, at different levels of granularity, and with the occasional need for correction. This correction can be made with the scientific method, which is itself an ongoing process of data collection and theorizing, using human perceptions supported by scientific experiments, and yielding results which still may be fallible but are also self-correcting over time.

It is a basic assumption of scientific inquiry that nature is at least to some degree structured, ordered, and regular. Scientific experimentation involves in every case observations of particular instances of more general types, i.e., this eukaryote cell under this microscope. The ultimate goal of science is to use observations and manipulations of such particulars to construct, validate, or falsify general statements and laws. Ontology is concerned with encoding information about the general features of things in reality, rather than information about particular individuals, times, or places.

### 2.2.3 General principles of ontology design

Arp et al., specify the following principles for designing an ontology that optimizes the utility in support of scientific research presented in Table 2.3 (Arp et al., 2015). They

Table 2.3: General principles of ontology design (Arp et al., 2015).

| # | Principle of Best Practice |
|---|---|
| 1. | **Realism.** The goal of an ontology is to describe reality. |
| 2. | **Perspectivalism.** There are multiple accurate descriptions of reality. |
| 3. | **Fallibilism.** Ontologies, like scientific theories, are revisable in light of new discoveries. |
| 4. | **Adequatism.** The entities in a given domain should be taken seriously on their own terms, not viewed as reducible to other kinds of entities. |
| 5. | **The Principle of Reuse.** Existing ontologies should be treated as benchmarks and reused whenever possible in building ontologies for new domains. |
| 6. | **The Ontology Design Process Should Balance Utility and Realism.** Sacrificing realism to address considerations of short-term utility when building an ontology may be at the detriment of the ontology's longer-term utility. |
| 7. | **The Ontology Design Process Is Open-Ended.** Scientific ontologies will always be subject to the need for an update in light of advances in knowledge; ontology design, maintenance, and updating is an ongoing process. |
| 8. | **The Principle of Low-Hanging Fruit.** In ontology design, begin with the features of the relevant domain that are easiest to understand and define, then work outward to more complex and controversial features. |

highlight the theoretical considerations the ontologist must have in mind, such as accepting a realist approach, iterative development, and accepting multiple accurate descriptions of the domain of interest. Additionally, other relevant principles which will be respected when designing the ontology extension are the OBO Foundry principles and the FAIR guiding principles which will be described in more detail in Chapter 6 (Smith et al., 2007; Wilkinson et al., 2016). All of the aforementioned principles serve to enhance the reusability and invaluableness of the developed resources.

### 2.2.4 Bottom-up and top-down ontology development approaches

An ontology can be created by following two generally different approaches: the top-down and the bottom-up development processes. A top-down development process starts with the definition of the most general concepts in the domain and subsequent specialization of the concepts, while the bottom-up development process starts with identifying the instances and moves up towards defining the more general concepts. Similarly, a data corpus can be annotated following either approach, the top-down approach being more manual, whereas the bottom-up approach is more automatized. In this thesis, we will mainly follow the top-down approach to design an ontology (see Chapter 3) and use the ontology to manually annotate a data corpus (see Section 4.4) while in Chapter 7 we will present a bottom-up approach for automatic data annotation. A qualitative comparison of the two approaches can be seen in Table 2.4.

#### 2.2.4.1 Bottom-up (Data-driven)

The bottom-up approach is based on raw data, most commonly in the form of text. This approach relies heavily on automated semantic analysis and consequently, the methods used come from the field of NLP. The amount of data required to automatically build an ontology or more broadly, a knowledge organization system depends on the methodology. For example, language models, like transformer-based neural nets, require a large corpus of text data, while techniques like latent semantic analysis require much fewer data. The main advantage of the bottom-up approach is that since it is automatic, it is easily scalable, i.e., increasing the data input would require no extensive changes to the methodology. While the representations produced by these models can account for the inter-dependencies between the entities, as Chomsky and his acolytes have pointed out for decades, language is just not an unambiguous vehicle for clear communication (Reboul, 2015). Without the human input of a nonlinguistic understanding of the context, the quality of these systems will always be limited (Browning, 2022).

#### 2.2.4.2 Top-down (Knowledge-driven)

The top-down approach is based on the knowledge that is obtained through human abstraction of raw data. As we discussed in Section 2.2.2, this approach can produce an ontology that is an insufficiently realistic representation of the real world, contingent on biased human perception. However, it is essential to adhere to best-practice design principles and engage in regular communication with community experts, to sieve out potential bias-induced inconsistencies. Here, the ability of the ontologist to deeply understand context is an advantage, at the expense of low scalability. The main benefit of this approach is that it is lightweight, both concerning data and computational resources, however, it requires a significant amount of human effort, as the ontology developer must go through heavy research to acquire the knowledge necessary to design an ontology.

#### 2.2.4.3 A comparison of both approaches

In Table 2.4, a qualitative comparison of both approaches is presented based on several criteria. When considering scalability, the top-down approach is much more difficult to scale, since it requires more human effort to gain knowledge to further expand the ontology. On the other hand, the bottom-up approach would not usually require modifications to the methodology, i.e., human effort, while it would potentially require more data (obtaining the data could however imply more human effort). Hence, the bottom-up approach is considered to be more scalable than the top-down approach. The expressiveness criterium is related to the contextual understanding of text data, which is fundamentally tied to the nature of language, as described earlier (Browning, 2022).

The development complexity of both approaches is high due to the required knowledge of the data that is required, as well as the methodologies used to utilize the data. As for the data requirements, bottom-up approaches require much more data to develop representations and contextual knowledge of sufficient quality. While the top-down approach also requires extensive research, the data requirements are much less due to full-bodied human thinking, not exclusively based on language. The top-down ontology developer has to understand the domain to design an ontology that encapsulates domain knowledge, while in the bottom-up approach, the focus is based more on the methods used, instead of the input data.

Table 2.4: Bottom-up and top-down ontology development. A qualitative comparison of bottom-up and top-down ontology development approaches.

| Approach | Scalability | Expressiveness | Development Complexity | Data Requirements | Domain Expertise |
|---|---|---|---|---|---|
| Top-down | Low | Low | High | Low | High |
| Bottom-up | High | Low | High | High | Low |

### 2.2.5 Formal and domain ontologies

Ontologies can differ mainly based on the generality of their content. So, we can distinguish between formal and material ontologies. A *formal ontology*, or top-level / upper-level ontology is domain neutral. It contains just those most general terms – such as "object" and "process" – which apply in all scientific disciplines. In a sense, it is much more related to philosophers. A material or *domain ontology* is domain-specific. It contains terms – such as "cell", "algorithm" or "time complexity" – which apply only in a subset of disciplines.

The use of domain ontologies is very common in many branches of science, starting from medicine, physics, and computer science. However, while multiple groups of researchers are creating incompatible domain ontologies focused on their specific local needs, we are led to the problem of inaccessible, non-sharable data, and nonoptimal use of resources. This is precisely why formal ontologies exist. They serve the purpose of providing a common backbone for domain ontologies, where the most general terms (universals) are defined. These terms are to be used as a starting point for developing definitions of terms representing the various sorts of lower-level universals and defined classes needed for different application purposes. The use of a top-level ontology ensures effective sharing of annotated information and it allows more effective governance and quality assurance of ontology development. Notable examples of formal ontologies include the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE), Standard Upper Merged Ontology (SUMO), and Basic Formal Ontology (BFO) (Arp et al., 2015; Borgo et al., 2006; Pease et al., 2002).

A domain is a delineated portion of reality corresponding to a scientific discipline such as cell biology or machine learning. Each domain ontology consists of a taxonomy together with other relations along with definitions and axioms governing how its terms and relations are to be understood. A domain ontology provides a controlled, structured representation of the entities within the relevant domain, one that can be used, for example, to annotate data pertaining to entities in that domain to make the data more easily accessible and shareable by human beings and processable by computers. Notable examples of domain ontologies mainly come from the domain of biology and include the Gene Ontology (GO), Foundational Model of Anatomy (FMA), and Chemical Entities of Biological Interest (ChEBI) (Consortium, 2004; Degtyarenko et al., 2007; Rosse & Mejino Jr, 2003).

**Basic Formal Ontology (BFO)** is an upper-level ontology developed to support integration of data obtained through scientific research (Arp et al., 2015). It was deliberately designed to be very small, so that it should be able to represent consistently those upper-level categories common to domain ontologies developed by scientists in different fields. It is small, also, to allow the exercise of the benefits of modularity and the division of expertise. BFO assists domain ontologists by providing a common top-level structure to support the interoperability of the multiple domain ontologies created in its terms. In this way, it helps to bring about a situation in which information compiled in separate repositories can form part of a common framework for the categorization of, and reasoning about, the entities in the corresponding domains. Entities are divided into two categories – continuants (entities that continue or persist through time, i.e., objects) and occurents (entities that occur or happen, i.e., events). This ontology was used as a top-level ontology in OntoDM and consequently, it is used by the ontology extension described in this thesis.

## 2.3 Overview of Relevant Ontologies

### 2.3.1 Ontologies for machine learning and data mining

In terms of formalizing domain knowledge, there are several notable efforts in the domain of data mining and machine learning. In this section, we will describe the OntoDM suite of ontologies (Panov et al., 2008), based on which an ontology extension was designed, presented in Section 3, as well as the Data Mining Optimization Ontology (DMOP) (Keet et al., 2015), the Expose ontology (Vanschoren & Soldatova, 2010) and the MEX vocabulary (Esteves et al., 2015).

**The OntoDM ontology** (Panov et al., 2008) consists of three modular ontologies: OntoDM-core (Panov et al., 2014) which represents core data mining entities such as datasets, data mining tasks, algorithms, models and patterns, OntoDT (Panov et al., 2016) — a generic ontology of data types, and OntoDM-KDD (Panov et al., 2013), which describes the knowledge discovery process. The ontology defines top-level concepts in data mining and machine learning, such as data mining task, algorithm, and their generalizations, which denote the results of applying an implementation of an algorithm to a given data set. Based on these general concepts, OntoDM also defines the components of algorithms, such as distance and kernel features, and other features they may contain. From the perspective of input and output data, there is a hierarchical representation of data in this ontology, from general concepts such as dataset to more specific concepts regarding their structure, such as the number of features, their role in a particular task, and finally the data type of each attribute. These features of OntoDM provide a complete formal representation of the data mining process from start to finish.

**The Data Mining OPtimization Ontology (DMOP)** was developed to support automation at various selection points in the DM process, including a deeper view of algorithms (mainly for classification and regression) (Keet et al., 2015). While other domain ontologies treat domain algorithms as black boxes and focus mainly on the inputs and outputs that specify the algorithms, DMOP provides a conceptualization of the internals of algorithms. The core concepts of DMOP are the different ingredients that go into the data mining process (DM-Process): The input of the process is composed of a task specification (DM-Task) and training/test data (DM-Data) provided by the user. Its output is a hypothesis (DM-Hypothesis), which can take the form of a global model (DM-Model) or a set of local patterns (DM-PatternSet). Tasks and algorithms are not processes that directly manipulate data or models, rather they are specifications of them: A DM-Task specifies a DM process (or any part thereof) in terms of the input it requires and the output it is expected to produce. A DM-Algorithm is the specification of a procedure that addresses a given DM-Task, while a DM-Operator is a program that implements a given DM-Algorithm and is executed by a DM-Operation. Instances of DM-Task and DM-Algorithm do no more than specify their input/output types (only processes have actual inputs and outputs). Each of these high-level entities is extended by corresponding hierarchies which are explained in more detail in a separate paper (Keet et al., 2013).

**Expose** is an ontology for data mining experiments (Vanschoren & Soldatova, 2010). It is complementary to OntoDM and DMOP and covers data mining experiments in detail, including experiment context, evaluation metrics, datasets, and algorithms. The authors provide ways to describe algorithm implementations using metadata such as name, version, URL, and the library to which they belong. The ontology also facilitates algorithm compositions and a look at the internal learning mechanisms of algorithms, i.e., how machine learning models are built and optimized. The ontology is used in designing the databases of OpenML, an online repository for storing and accessing data related to the execution of ML algorithms, containing information on Tasks, Runs, Flows, etc. (Vanschoren et al., 2014).

**The MEX Vocabulary** was introduced as a lightweight interchange vocabulary for machine learning experiments (Esteves et al., 2015). It is based on the PROV Ontology (PROV-O) to provide a light and simple representation of provenance information. The PROV Ontology ("Prov-O: The prov ontology," 2013) defines classes, relations, and restrictions to represent provenance information generated in different systems and different contexts.

MEX consists of three sub-vocabularies, each representing one big part of the data mining process. MEX-Core contains both the definitions of key data mining entities for representing the basic steps of executing a machine learning experiment and the provenance information for linking the published results with the produced metadata. The MEX-Algorithm sub-vocabulary represents the whole concept of machine learning algorithms. It contains definitions for the characteristics associated with every algorithm, such as the class of the algorithm, learning problem, and learning method. MEX-Performance defines the entities for representing the findings from a finished machine learning experiment.

### 2.3.2 A common framework for representing a data mining algorithm

A key ontology design pattern for the representation of algorithms is the Algorithm-Implementation-Execution design pattern (Lawrynowicz et al., 2017). This pattern has originated from independent research of several groups on modeling the domain of ML/DM. Ontologies which are based on this design pattern include the MEX vocabulary (Esteves et al., 2015), the DMOP ontology (Keet et al., 2015), and the OntoDM ontology (Panov et al., 2008). An advantage of adhering to this pattern is to conceptually align the ontologies which facilitate the metadata interchange process and reproducible research.

A key element in the pattern is the *Algorithm* class which represents any algorithm regardless of its software implementation. *Implementation* is an executable implementation of an algorithm (a script or a workflow). The *Execution* class is an execution of an implementation on a given machine. *Task* is a formal description of a process that needs to be completed, based on inputs and outputs, or defined conceptually. *Input* and *Output* are information entities, where the former is an input to an *Execution*, i.e., some data, and the latter is an output of the *Execution*, i.e., some transformed data, the result of some computations, etc. *Parameter* is a parameter of an implementation which is set before its execution. It is different from *Input* in that it is a variable. *ParameterSetting* is an entity that connects a parameter and its value that is being set before an implementation execution.

Since the aim of the MEX vocabulary was not to describe the complete data-mining process, unlike Onto-DM and DMOP, it does not include entities like Task, Input, Output, etc. Instead, MEX was designed to provide a simple and lightweight vocabulary for exchanging machine learning metadata and supporting data management in ML scenarios. It reuses terms from the PROV-O ontology such as Algorithm, Execution, and ExperimentConfiguration, modeled in a way that closely matches the design pattern.

DMOP was developed with a primary use case in meta-mining, that is meta-learning extended to an analysis of full DM processes. The design pattern is closely matched via the following entity chain: DM-Operation executes DM-Operator implements DM-Algorithm.

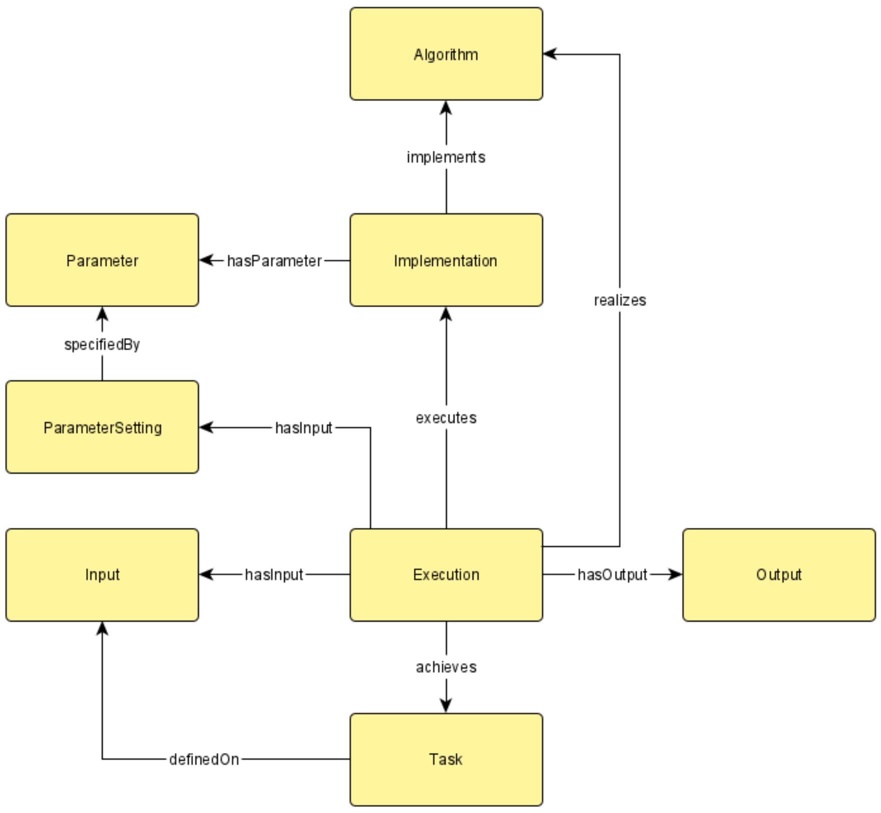


Figure 2.2: The Algorithm-Implementation-Execution ontology design pattern. Reproduced from (Lawrynowicz et al., 2017).

In OntoDM, the description structure includes three layers: a specification layer, an implementation layer, and an application layer. This structure is based on the upper-level ontologies, i.e., the more general ontologies which are reused, such as BFO, OBI, and IAO. Intuitively, it is clear that the specification layer covers the specification of an algorithm (i.e., expressed in some form of text), the implementation layer covers the runnable version of an algorithm (i.e., code or pseudocode), while the execution layer covers an actual execution of an algorithm (a workflow describing the exact inputs, outputs, as well as the execution environment).

## 2.4 Resources for Semi-automatic Annotation of ML/DM Text Data

In this section, we will present resources that are used for the semi-automatic annotation of ML/DM paper abstracts. The main ingredient of the process is the data, which naturally comes in the form of text, as many ML/DM algorithm developers describe the algorithms in publications, such as conference papers and journal papers. The other key factor besides the data are the methods used to annotate the data. We will present the methods that can be used to automatically annotate paper abstracts, mainly consisting of pre-trained language models, as well as a custom neural net model for named-entity recognition (NER).

### 2.4.1 Information extraction and named entity recognition

Information extraction (IE) is the task of automatically extracting structured information from unstructured and/or semi-structured machine-readable documents and other electronically represented sources. In most cases, this activity concerns processing human language texts employing NLP (Freitag, 2000). NER is a subtask of IE that seeks to locate and classify named entities mentioned in unstructured text into pre-defined categories such as person names, organizations, locations, medical codes, time expressions, etc. (Nadeau & Sekine, 2007).

NER systems have been created that use linguistic grammar-based techniques as well as statistical ML models. Hand-crafted grammar-based systems typically obtain better precision, but at the cost of lower recall and months of work by experienced computational linguists. ML-based NER systems typically require a large amount of manually annotated training data. Semi-supervised approaches have been suggested to avoid part of the annotation effort, where a combination of a small amount of labeled data with a large amount of unlabeled data is used for training (Nadeau & Sekine, 2007).

### 2.4.2 Data resources

Here, we will describe several compiled text corpora which cover paper abstracts. ML/DM practitioners usually describe novel algorithms in scientific publications. These publications contain information related to how the algorithm works, accompanied by experiments, findings, and potential applications. While papers contain different modalities, including text, images, tables, formulas, and pseudocode, the abstracts briefly describe the key aspects of an algorithm in a textual form. Hence, they are well-suited as a resource for NLP tasks, such as NER, relation extraction, etc. Apart from using already developed corpora, there are possibilities for compiling custom corpora, for example by collecting data from the OpenReview paper repository. Some of the resources already come packed with named entity annotations, while some require manual annotation.

#### 2.4.2.1 The SCIERC corpus of paper abstracts

The SCIERC corpus (Luan et al., 2018) includes annotations for scientific entities, their relations, and coreference clusters for 500 scientific abstracts. The abstracts are taken from 12 AI conference/workshop proceedings in four AI communities, from the Semantic Scholar Corpus. SCIERC extends previous datasets in scientific articles SemEval 2017 Task 10 and SemEval 2018 Task 7 by extending entity types, relation types, relation coverage, and adding cross-sentence relations using coreference links. The annotation schema consists of six types for annotating scientific entities (Task, Method, Metric, Material, Other-ScientificTerm, and Generic) and seven relation types (Compare, Part-of, Conjunction, Evaluate-for, Feature-of, Used-for, HyponymOf). Directionality is taken into account except for the two symmetric relation types (Conjunction and Compare). Coreference links are annotated between identical scientific entities. A Generic entity is annotated only when the entity is involved in a relation or is referred to with another entity.

#### 2.4.2.2 The Arxiv corpus of paper abstracts

The corpus of scientific paper abstracts published in the Arxiv repository contains 38972 paper abstracts, along with paper titles and subject categories. It is available to download from the Kaggle repository of datasets and models (Sayak, 2021). The categories are labeled according to the Arxiv category taxonomy which at the most general level distinguishes between fields like Computer Science, Economics, Physics, Quantitative Biology, etc. Most of the papers in this corpus belong to the Computer Science category along with smaller segments belonging to similar fields like Mathematics, Statistics, Electrical Engineering, and Systems Science.

### 2.4.3 Pre-trained language models

spaCy is an open-source software library for advanced natural language processing (Srinivasa-Desikan, 2018). spaCy also supports deep learning workflows that allow connecting statistical models trained by popular machine learning libraries like TensorFlow and PyTorch (Abadi et al., 2016; Paszke et al., 2019). Using Thinc as its backend, spaCy features convolutional neural network models for part-of-speech tagging, dependency parsing, text categorization, and NER. Prebuilt statistical neural network models to perform these tasks are available for 17 languages. Their models have emerged as the de facto standard for practical NLP due to their speed, robustness, and close-to-state-of-the-art performance.

scispaCy was developed as a robust, efficient, and performant NLP library to satisfy the primary text processing needs in the biomedical domain (Neumann et al., 2019). For this purpose, spaCy3 models were retrained for POS tagging, dependency parsing, and NER using corpora relevant to the biomedical text, while also the tokenization module was enhanced with additional rules.

# Chapter 3

# OntoDM-algorithms: An Ontology Extension for Annotating ML/DM Algorithms

In this chapter, we describe OntoDM-algorithms, an extension of the OntoDM-core ontology, with a focus on the representation of algorithms for predictive modeling. First, we present the relevant entities from which we extend the OntoDM-core ontology. Following this, we describe the newly added entities and the related modeling decisions. Finally, we delve deeper into how the ontology extension is brought together using all the relevant and available resources.

## 3.1 Relevant Entities from the OntoDM-core Ontology

Following the Principle of Reuse described in Section 2.2.3, when developing an ontology, it is essential to reuse as much as possible relevant ontological content that has already been created, so as to not spend time and effort reinventing the wheel. However, since not every ontology follows adequate design principles, the developed ontologies can be poor in quality and subsequently not directly utilizable. Still, they should not be disregarded altogether, but rather they should be examined to provide a reference point for the newly created content. In domains of ML/DM, several ontologies exemplify quality modeling, which we presented in Section 2.3.1. Specifically, we will use the OntoDM-core ontology as a backbone, which we will extend with new entities. In this thesis, we will mainly focus on the specification and implementation levels of the ontology, which generally contain information entities included in the ontology design pattern like data mining task, data mining algorithm, data mining dataset, etc.

The central entity in the OntoDM ontology, and consequently in the extension developed, is the Data Mining Algorithm. A *data mining algorithm* is an algorithm, designed to solve a data mining task. It takes as input a dataset of examples of a given datatype and produces as output a generalization (from a given class) on the given datatype. A data mining algorithm as a specification is a subclass of the IAO class plan specification having as parts a data mining task, an action specification (reused from IAO), a generalization specification, and a document (reused from IAO). The data mining task defines the objective that the realized plan should fulfill at the end giving as output a generalization, while the action specification describes the actions of the data mining algorithm realized in the process of execution. The generalization specification denotes the type of generalization produced by executing the algorithm. Finally, having a document class as a part allows us to connect the algorithm to the annotations of documents (journal articles, workshop articles, technical reports) that publish knowledge about the algorithm. In analogy with the taxonomy of datasets, data mining tasks, and generalizations, the ontology includes a taxonomy of data mining algorithms. The taxonomy design was based on the data mining task and the generalization produced as the output of the execution of the algorithm. Generally, a data mining algorithm can either be a single generalization algorithm (i.e., producing a single generalization as output), or an ensemble algorithm (i.e., producing two or more generalizations as output). The structure of the key classes in OntoDM-core can be seen in Figure 3.1.

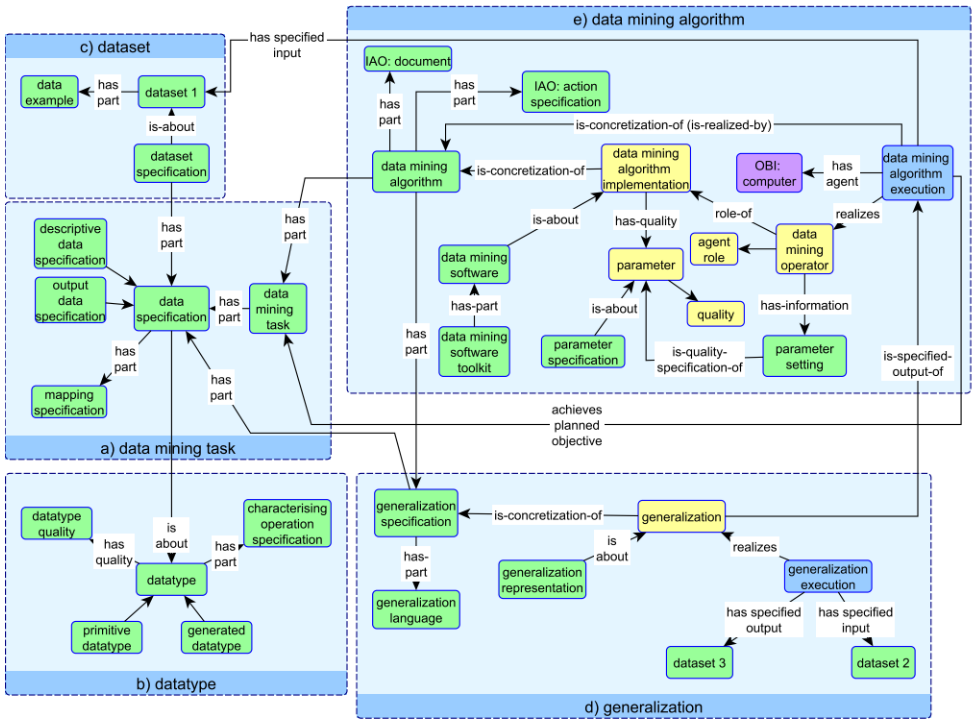


Figure 3.1: Key classes in the OntoDM-core ontology. The boxes in green represent classes at the specification layer, the yellow boxes denote classes at the implementation layer, and the blue boxes denote classes at the application layer, while the violet boxes represent material entities. The figure is reused from (Panov et al., 2014).

The task of data mining is to produce a generalization from given data. In OntoDM-core, the term generalization is used to denote the outcome of a data mining task. A data mining task is defined as a sub-class of the IAO class objective specification. It specifies the objective that a data mining algorithm execution process needs to achieve when executed on a dataset to produce as output a generalization. The definition of a data mining task depends directly on the data specification and indirectly on the datatype of the data at hand. This allows us to form a taxonomy of data mining tasks based on the type of data. The four basic classes of data mining tasks based on the generalizations that are produced as output used are clustering, pattern discovery, probability distribution estimation, and predictive modeling (Džeroski, 2006). These classes of tasks are included as a first level of the OntoDM-core data mining task taxonomy.

OntoDM imports the IAO class dataset (defined as "a data item that is an aggregate of other data items of the same type that have something in common") and extends it by further specifying that a DM dataset has part data examples. OntoDM-core also defines the class dataset specification to enable reasoning about data and datasets. It specifies the type of the dataset based on the type of data it contains. Using data specifications and the taxonomy of datatypes from the OntoDT ontology, a taxonomy of datasets is defined.

A generalization denotes the outcome of a data mining task. Many different types of generalizations have been considered in the data mining literature. The most fundamental types of generalizations are in line with the data mining tasks. These include clusterings, patterns, probability distributions, and predictive models. In OntoDM-core, the generalization specification class is a subclass of the OBI class data representational model. It specifies the type of the generalization and includes as part the data specification for the data used to produce the generalization, and the generalization language specification, for the language in which the generalization is expressed. Examples of generalization language formalisms for the case of a predictive model include the languages of trees, rules, Bayesian networks, graphical models, neural networks, etc. As in the case of datasets and data mining tasks, a taxonomy of generalizations is constructed. In OntoDM-core, at the first level, we distinguish between a single generalization specification and an ensemble specification. Ensembles of generalizations have as parts single generalizations. We can further extend this taxonomy by taking into account the data mining task and the generalization language.

The Parameter entity is modeled as a quality of the data mining algorithm implementation entity, which in turn is a concretization of the data mining algorithm. The parameters of the algorithm affect its behavior when the algorithm implementation is used as an operator. A parameter itself is specified by a parameter specification that includes its name and description.

The OntoDT taxonomy of datatypes consists of primitive datatypes, generated datatypes, subtypes, and defined datatypes (Panov et al., 2016). The primitive datatypes are pre-defined axiomatically and include the classes of discrete, enumerated, character, date and time, scaled, real and complex datatypes, while pre-defined instances of primitive datatypes include boolean, void, ordinal, rational, and integer datatypes. Generated datatypes are defined by a datatype generator and component datatypes. For example, the aggregated datatype is a generated datatype whose values are made up of values of other datatypes joined by an aggregate (tuple, class, set, bag, sequence, array, and table). Subtypes are derived datatypes obtained by restricting the value space of an existing datatype with the use of a subtype generator (range, selection, exclusion, size, extension, explicit enumeration). Defined datatypes are user-defined datatypes with a type specification.

## 3.2 Knowledge Modelling in the OntoDM-algorithms Extension

In this section, we introduce the OntoDM-algorithms extension. We will present its core entities, the modeling decisions as well as the implementation of the ontology. The ontology was designed following the OBO Foundry principles, which note that ontologies should be open, orthogonal, instantiated in a well-specified syntax, and designed to share a common space of identifiers (Smith et al., 2007). Additionally, the general principles for ontology design described in Section 2.2.3 were followed. Being an extension of OntoDM, OntoDM-algorithms reuses external entities from resources such as the Relations Ontology (RO) (Mungall, 2015), Information Artifact Ontology (IAO) (Ceusters, 2012), and the Basic Formal Ontology (BFO) which is used as an upper level ontology (Arp et al., 2015). The use of an upper-level ontology makes the ontology easily interoperable with other ontologies.

The ontology backbone was provided by OntoDM-core and it consists of the following main entities: data mining algorithm, data mining task, dataset specification, generalization specification, and parameter specification. Apart from these more general entities, we introduce new entities and taxonomies, such as the computational problem taxonomy, the model and algorithm parameter entities, the computational complexity entity, the assumption specification, the ensemble algorithm taxonomy, and the sampling entity. An overview of OntoDM-algorithms can be seen in Figure 3.2.

### 3.2.1 Computational problem

Generally, algorithms are designed to solve a specific problem or perform some sort of computation. In theoretical computer science, a computational problem is defined as a problem that may be solved by an algorithm. A computational problem can be further divided into five types: Decision Problem, Search Problem, Counting Problem, Optimization Problem, and Function Problem. In DM and ML, the most frequent problem types are the Search Problem (i.e., in Decision Trees), and the Optimization Problem (i.e., in Neural Networks, Support Vector Machines, etc.). More concrete examples of computational problems using mathematical notation will be given in Section 3.2.2.

**Search problem.** A search problem consists of a search space, start state, and goal state. Search algorithms provide search solutions through a sequence of actions that transform the initial state into the goal state. To formulate the search problem, its constituent factors need to be defined. The *initial state* is the state in which the search starts. The *state space* consists of all the possible states that can be attained from the initial state through a series of actions. *Actions* are steps, or operations undertaken by the search algorithm in a particulate state, while the *goal state* is the endpoint or the desired state. When there are multiple paths to the goal state, the *path cost* is used to choose the optimal path from the *initial state* to the *goal state* (Russell, 2010).

Search algorithms are usually divided into uninformed and informed search algorithms. *Uninformed search algorithms* do not have supplementary information (i.e., a heuristic function) that can assist them to attain the end goal other than the information given in the problem definition. Search algorithms usually operate on graph or tree data structures. Examples include breadth-first search, depth-first search, and uniform cost search. *Informed search algorithms* are heuristic algorithms that apart from the problem definition have additional information which helps in more efficient searching. This information is obtained by a heuristic function that estimates how close a state is to the goal state. Examples include greedy search algorithms, A search, Hill Climbing algorithm, etc. (Russell, 2010).

Decision Tree learning is one segment where informed search algorithms are heavily utilized (Breiman et al., 2017). The problem of learning an optimal decision tree is known to be NP-complete. As a consequence, practical decision tree learning algorithms are based on heuristics such as the greedy algorithm where locally optimal decisions are made at each node. In deep learning, uninformed search algorithms are heavily employed in more traditional approaches to hyperparameter tuning. Grid search is one of the most commonly-used methods to explore hyperparameter configuration space. It can be considered an exhaustive search or a brute-force method that evaluates all the hyperparameter combinations given to the grid of configurations. Random search works similarly, but instead of testing all values in the search space, it randomly selects a pre-defined number of samples between the upper and lower bounds as candidate hyperparameter values and then trains these candidates until the defined budget is exhausted. Because of the limitations of uninformed search algorithms, informed search algorithms are more reliable and hence used more frequently. Examples include Gradient-based optimization, Bayesian optimization, etc. (Yang & Shami, 2020).

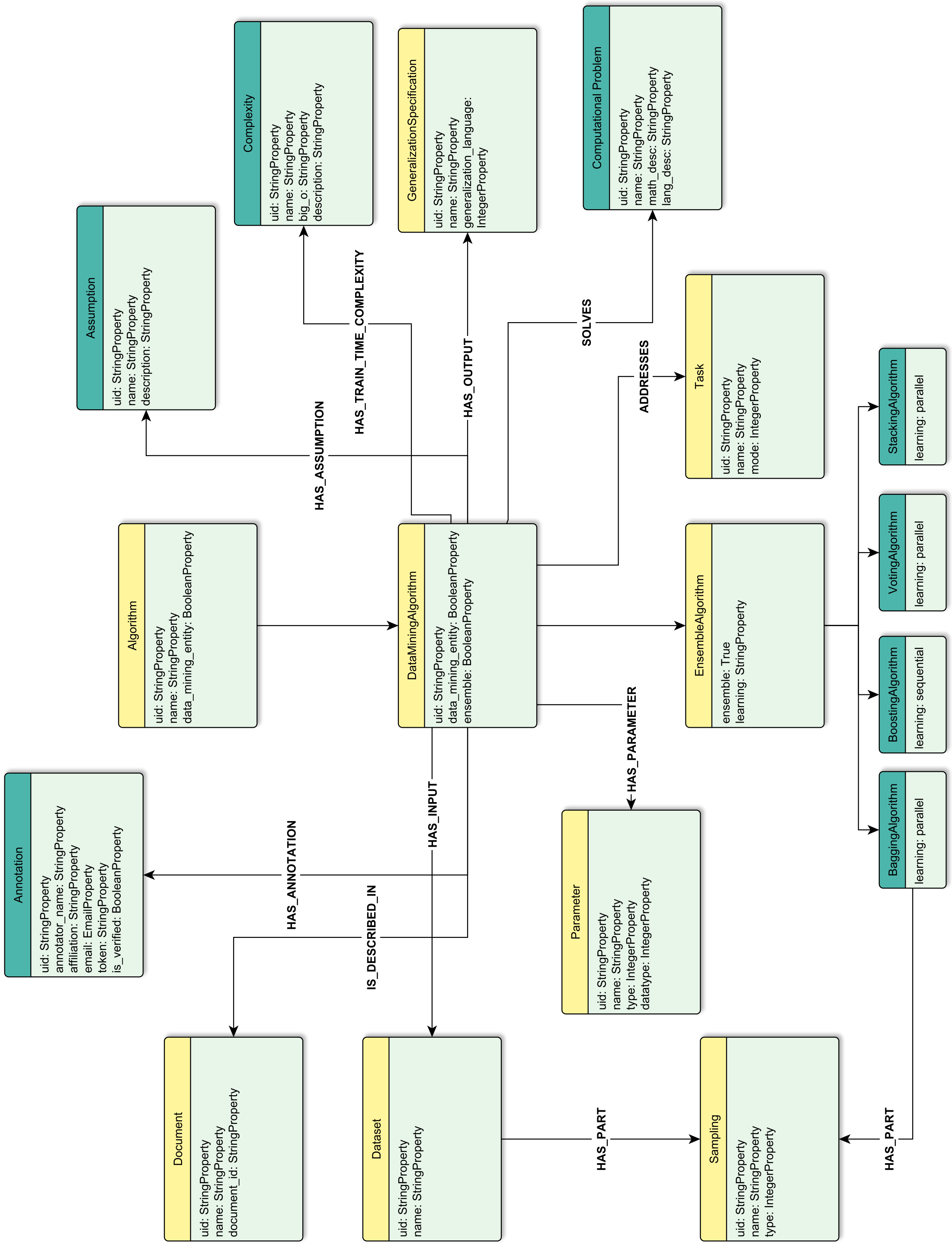


Figure 3.2: A diagram of top-level entities in OntoDM-algorithms. Each box represents a class, for which the full properties are provided, along with the corresponding datatypes. Unlabeled relationships represent the is_a relationship, and for most of the relationships there exist inverse relationships that are not shown for visualization purposes. The yellow boxes denote reused entities, while the green boxes denote newly added entities.

**Optimization Problem.** The key process of ML is to solve optimization problems. In mathematical terms, an optimization problem is the problem of finding the best solution from among the set of all feasible solutions to maximize or minimize the objective function (Sun et al., 2019). An optimization problem consists of three major components: a set of decision variables $x$, an objective function $f(x)$ to be either minimized or maximized, and a set of constraints that allow the variables to take on values in certain ranges (if it is a constrained optimization problem). Therefore, the goal of optimization tasks is to obtain the set of variable values that minimize or maximize the objective function while satisfying any applicable constraints. Many traditional methods can be used to solve optimization problems, including gradient descent, Newton's method, conjugate gradient, and heuristic optimization methods. The NEOS guide taxonomy of optimization problems is a well-structured taxonomy that focuses mainly on the sub-fields of deterministic optimization with a single objective function. The taxonomy is presented in Figure 3.3.

In ML model training, the model parameters are initialized and optimized by an optimization method until the objective function approaches a minimum value or the accuracy approaches a maximum value. Similarly, hyperparameter optimization methods aim to optimize the architecture of an ML model by detecting the optimal hyperparameter configurations. In OntoDM-algorithms we add the Computational Problem entity and the corresponding children as a descendant of the information processing objective entity, at the same level as the data mining task in OntoDM-core. We allow the optimization problem to be expressed with natural language and mathematical notation, stored in the latex format. The optimization problem entity can further be divided along several dimensions, such as, whether one or multiple objectives exist, whether there are constraints on the variables, and based on the variables' datatypes.

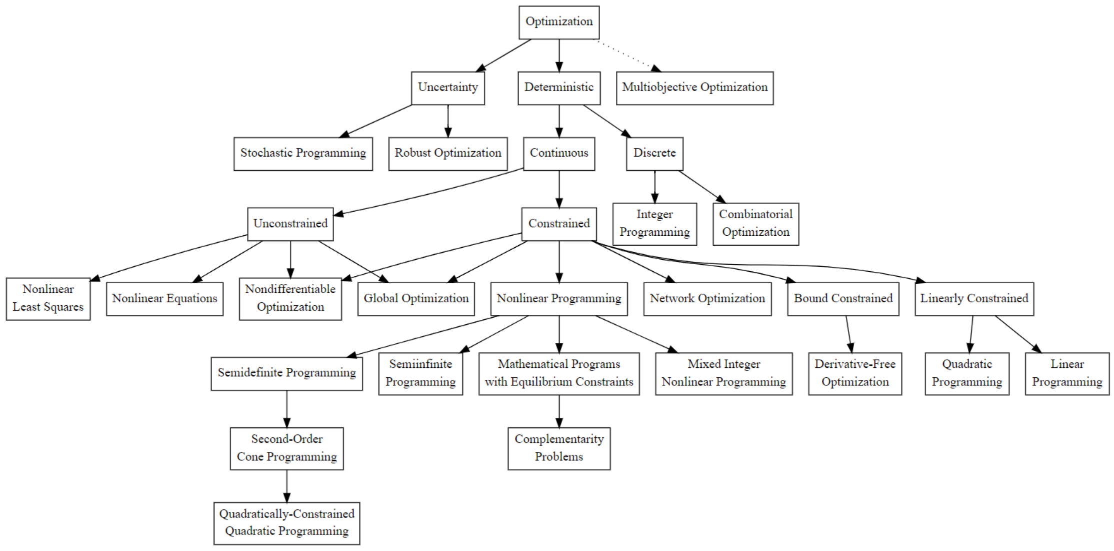


Figure 3.3: The optimization problem taxonomy. Source: https://neos-guide.org/optimization-tree.

**Continuous Optimization versus Discrete Optimization.** Some models only make sense if the variables take on values from a discrete set, often a subset of integers, whereas other models contain variables that can take on any real value. Models with discrete variables are discrete optimization problems; models with continuous variables are continuous optimization problems. Continuous optimization problems tend to be easier to solve than discrete optimization problems; the smoothness of the functions means that the objective function and constraint function values at point x can be used to deduce information about points in a neighborhood of $x$.

**Unconstrained Optimization versus Constrained Optimization.** Another important distinction is between problems in which there are no constraints on the variables and problems in which there are constraints on the variables. Unconstrained optimization problems arise directly in many practical applications; they also arise in the reformulation of constrained optimization problems in which the constraints are replaced by a penalty term in the objective function. Constrained optimization problems arise from applications in which there are explicit constraints on the variables. The constraints on the variables can vary widely from simple bounds to systems of equalities and inequalities that model complex relationships among the variables. Constrained optimization problems can be further classified according to the nature of the constraints (e.g., linear, nonlinear, convex) and the smoothness of the functions (e.g., differentiable or nondifferentiable).

**None, One or Many Objectives.** Most optimization problems have a single objective function, however, there are interesting cases when optimization problems have no objective function or multiple objective functions. Feasibility problems are problems in which the goal is to find values for the variables that satisfy the constraints of a model with no objective to optimize. Complementarity problems, common in engineering and economics, address the goal to find a solution that satisfies the complementarity conditions. Multi-objective optimization problems are used when optimal decisions need to be taken in the presence of trade-offs between two or more conflicting objectives. For example, choosing a portfolio might involve maximizing the expected return while minimizing the risk. However, in practice, problems with multiple objectives are often reformulated as single objective problems by either forming a weighted combination of the different objectives or by replacing some of the objectives with constraints.

**Deterministic Optimization versus Stochastic Optimization.** In deterministic optimization, it is assumed that the data for the given problem are known accurately. However, for many actual problems, the data cannot be known accurately for a variety of reasons. The first reason is due to simple measurement error. The second and more fundamental reason is that some data represent information about the future (e.g., product demand or price for a future time) and simply cannot be known with certainty. In stochastic optimization, the uncertainty is incorporated into the model. Robust optimization techniques can be used when the parameters are known only within certain bounds; the goal is to find a solution that is feasible for all data and optimal in some sense. Stochastic optimization models take advantage of the fact that probability distributions governing the data are known or can be estimated; the goal is to find some policy that is feasible for all the possible data instances and optimizes the expected performance of the model.

### 3.2.2 Algorithm parameters

Two types of parameters exist in ML/DM algorithms; ones that can be initialized and updated through the data learning process (e.g., the weights of neurons in neural networks), named model parameters, while the other, named algorithm (hyper)parameters, cannot be directly estimated from data learning and must be set before training an ML model because they define the ML model. Algorithm parameters are used to either configure an ML model (e.g., the penalty parameter C in a support vector machine, and the learning rate to train a neural network) or to specify the algorithm used to minimize the loss function (e.g., the activation function and optimizer types in a neural network, and the kernel type in a support vector machine). Because of this, they can also be referred to as hyperparameters, since the word 'hyper' (meaning over, above, beyond in Greek) semantically describes the relationship between the model parameters and the algorithm parameters, i.e., the algorithm parameters govern the model, or are 'above' the model parameters. Therefore, model parameters are parameters that change, i.e., are being updated during the training (learning) process, while algorithm (hyper)parameters remain static during the training process. The schema depicting the modeling of these entities in the ontology is presented in Figure 3.4.

To build an optimal ML model, a range of possibilities must be explored. The process of designing the ideal model architecture with an optimal hyperparameter configuration is named hyperparameter tuning. Tuning hyperparameters is considered a key component of building an effective ML model, especially for tree-based ML models and deep neural networks, which have many hyperparameters. The hyperparameter tuning process is different among different ML algorithms due to their different types of hyperparameters, including categorical, discrete, and continuous hyperparameters.

In general, ML/DM algorithms can be classified as supervised and unsupervised learning algorithms, based on whether they are built to model labeled or unlabeled datasets. Supervised learning algorithms are a set of ML algorithms that map input features to a target by training on labeled data, and mainly include linear models, k-nearest neighbors (KNN), support vector machines (SVM), naive Bayes (NB), decision-tree-based models, and deep learning (DL) algorithms. Furthermore, supervised learning methods can be classified as classification or regression methods, depending on whether the target variables are discrete or continuous. Moreover, several ensemble learning methods combine different single generalization algorithms to further improve model performance, like bagging, boosting, voting, and stacking. In this section, we study the important hyperparameters of common ML algorithms based on their names in the Python library scikit-learn (sklearn). We will focus only on supervised single generalization algorithms and ensemble algorithms.

**Linear regression.** Linear regression is a typical regression model that predicts a target $y$ with the following equation:

$$\hat{y}(w, x) = w_0 + w_1 x_1 + ... + w_p x_p, \tag{3.1}$$

where the target variable $y$ is expected to be a linear combination of $p$ input features $x = (x_1, \cdots, x_p)$, and $\hat{y}$ is the predicted value. The weight vector $w = (w_1, \cdots, w_p)$ is designated as an attribute 'coef_' and $w_0$ is defined as another attribute 'intercept_' of the linear model in sklearn. Usually, no hyperparameter needs to be tuned in linear regression. A linear model's performance mainly depends on how well the problem or data follows a linear distribution.

To improve the original linear regression models, ridge regression was proposed in (Hoerl & Kennard, 1970). Ridge regression imposes a penalty on the coefficients, and aims to minimize the objective function:

$$\min_{w} ||Xw - y||_2^2 + \alpha ||w||_2^2 \tag{3.2}$$

The complexity parameter $\alpha$ controls the amount of shrinkage: the larger the value of $\alpha$, the greater the amount of shrinkage, and thus the coefficients $w$ become more robust to collinearity. Therefore, the regularization strength parameter $\alpha$ represents a hyperparameter in the ridge regression model.

**KNN.** K-nearest neighbor (KNN) is a simple ML algorithm that is used to classify data points by calculating the distances between different data points. In KNN, the predicted class of each test sample is set to the class to which most of its k-nearest neighbors in the training set belong. Assuming the training set $T = (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n), x_i$ is the feature vector of an instance, and $y_i \in c_1, c_2, \cdots, c_m$ is the class of the instance, $i = (1, 2, \cdots, n)$, for a test instance $x$, its class $y$ can be denoted by (Keller et al., 1985):

$$y = \underset{c_j}{\operatorname{argmax}} \sum_{x_i \in N_k(x)} I(y_i = c_j), i = 1, 2, \cdots, n; j = 1, 2, \cdots, m, \tag{3.3}$$

where $I(x)$ is an indicator function, $I = 1$ when $y_i = c_j$, otherwise $I = 0$; $N_k(x)$ is the field involving the k-nearest neighbors of $x$. In KNN, the number of considered nearest neighbors, $k$, is the most crucial hyperparameter. If $k$ is too small, the model will be under-fitting; if $k$ is too large, the model will be over-fitting and require high computational time. In addition, the weighted function used in the prediction can also be chosen from 'uniform' (points are weighted equally) or 'distance' (points are weighted by the inverse of their distance), depending on specific problems. The distance metric and the power parameter of the Minkowski metric can also be tuned as they can result in minor improvements. Lastly, the 'algorithm' used to compute the nearest neighbors can also be chosen from a ball tree, a k-dimensional (KD) tree, or a brute force search. Typically, the model can determine the most appropriate algorithm itself by setting the 'algorithm' to 'auto' in sklearn.

**SVM.** A support vector machine (SVM) is a supervised learning algorithm that can be used for both classification and regression problems (Cortes & Vapnik, 1995). SVM algorithms are based on the concept of mapping data points from low-dimensional into high-dimensional space to make them linearly separable; a hyperplane is then generated as the classification boundary to partition data points. Assuming there are $n$ data points, the objective function of SVM is:

$$\underset{w}{\operatorname{argmin}} \Big\{ \frac{1}{n} \sum_{i=1}^{n} max \big\{ 0, 1 - y_i f(x_i) \big\} + C w^T w \Big\}, \tag{3.4}$$

where $w$ is a normalization vector; $C$ is the penalty parameter of the error term, which is an important hyperparameter of all SVM models. The kernel function $f(x)$, which is used to measure the similarity between two data points $x_i$ and $x_j$, can be chosen from multiple types of kernels in SVM models. Therefore, the kernel type would be a vital hyperparameter to be tuned. Common kernel types in SVM include linear kernels, radial basis function (RBF), polynomial kernels, and sigmoid kernels. Depending on the choice of kernel type, additional hyperparameters can be tuned. For example, the polynomial kernel has an additional conditional hyperparameter $d$ representing the 'degree' of the polynomial kernel function.

**Naive Bayes.** Naive Bayes (NB) algorithms are supervised learning algorithms based on the Bayes' theorem (H. Zhang, 2004). Assuming there are n dependent features $x1, \cdots xn$ and a target variable $y$, the objective function of NB can be denoted by:

$$\hat{y} = \underset{y}{\operatorname{argmax}} P(y) \prod_{i=1}^{n} P(x_i|y), \tag{3.5}$$

where $P(y)$ is the probability of a value $y$, and $P(xi|y)$ is the posterior probabilities of $x_i$ given the values of $y$. Regarding the different assumptions of the distribution of $P(x_i|y)$, there are different types of NB classifiers. The four main types of NB models are: Bernoulli

NB, Gaussian NB, multinomial NB, and complement NB. For the Gaussian NB, there is not usually any hyperparameter that needs to be tuned; the performance of a Gaussian NB model mainly depends on how well the dataset follows Gaussian distribution. The other variants have the additive (Laplace/Lidstone) smoothing parameter, $\alpha$, as the main hyperparameter that needs tuning.

**Decision Tree.** Decision tree (DT) is a common classification method that uses a tree structure to model decisions and possible consequences by summarizing a set of classification rules from the data (Breiman et al., 2017). A DT has three main components: a root node representing the entire data; multiple decision nodes indicating decision tests and sub-node splits over each feature; and several leaf nodes representing the result classes. DT algorithms recursively split the training set with better feature values to achieve good decisions on each subset. Pruning, which means removing some of the sub-nodes of decision nodes, is used in DT to avoid over-fitting. Since a deeper tree has more sub-trees to make more accurate decisions, the maximum tree depth, 'max depth', is an essential hyper-parameter that controls the complexity of DT algorithms.

There are many other important HPs to be tuned to build effective DT models. Firstly, the quality of splits can be measured by setting a measuring function (heuristic), denoted by 'criterion' in sklearn. The choice of the heuristic depends on the task being solved (classification or regression). Gini impurity and information gain are the two main types of measuring functions in the classification scenario, while the mean squared error and the mean absolute error are the most common options in the regression scenario.

The split selection method, 'splitter', can also be set to 'best' to choose the best split, or 'random' to select a random split. The number of considered features to generate the best split, 'max features', can also be tuned as a feature selection process. Moreover, there are several discrete hyper-parameters related to the splitting process: the minimum number of data points to split a decision node or to obtain a leaf node, denoted by 'min samples split' and 'min samples leaf', respectively; the 'max leaf nodes', indicating the maximum number of leaf nodes, and the 'min weight fraction leaf' that means the minimum weighted fraction of the total weights, can also be tuned to improve model performance.

**Deep Learning Models.** Deep learning (DL) algorithms are widely applied to various areas like computer vision, natural language processing, and machine translation since they have had great success solving many types of problems. DL models are based on the theory of artificial neural networks (ANNs). Common types of DL architectures include feedforward neural networks (FFNNs), deep belief networks (DBNs) (Hinton, 2009), convolutional neural networks (CNNs) (LeCun, Bengio, et al., 1995), recurrent neural networks (RNNs) (Mikolov et al., 2010) and many more. All these DL models have similar hyperparameters since they have a similar underlying neural network architecture. Compared with other ML models, DL models benefit more from hyperparameter optimization since they often have many hyperparameters that require tuning.

One set of hyperparameters is mainly related to the neural network model architecture. Here, the two main hyperparameters relate to the size of the network, measured with the number of layers (excluding the input and output layer), and the number of neurons in each layer. These two hyperparameters are tuned with respect to the complexity of the dataset, or the complexity of the problem; generally, a larger complexity requires a larger model size. Next, the activation function at each layer can be configured and tuned. The main choices are 'softmax', 'sigmoid', 'tanh', etc. The loss function is usually not tuned, since it depends directly on the problem type, i.e., binary cross-entropy is used for binary classification tasks, while root-mean-squared error (RMSE) is used for regression tasks.

Another set of hyperparameters relates to the general training procedure. Mini-batch size and the number of epochs represent the number of processed samples before updating the model, and the number of complete passes through the entire training set, respectively. Mini-batch size is affected by the resource requirements of the training process and the number of iterations. The number of epochs depends on the size of the training set and should be tuned by slowly increasing its value until validation accuracy starts to decrease, which indicates over-fitting. On the other hand, DL models often converge within a few epochs, and the following epochs may lead to unnecessary additional execution time and over-fitting, which can be avoided by the early stopping method. Early stopping is a form of regularization whereby model training stops in advance when validation accuracy does not increase after a certain number of consecutive epochs. The number of waiting epochs, called early stop patience, can also be tuned to reduce model training time.

Related to the optimization procedure we have another set of hyperparameters. The most obvious tuneable hyperparameter here is the type of optimization algorithm; common choices include stochastic gradient descent (SGD), adaptive moment estimation (Adam), etc. Here, the most important hyperparameter is the learning rate which determines the step size at each iteration, which enables the objective function to converge. A large learning rate speeds up the learning process, but the gradient may oscillate around a local minimum value or even cannot converge. On the other hand, a small learning rate converges smoothly, but will largely increase model training time by requiring more training epochs. An appropriate learning rate should enable the objective function to converge to a global minimum in a reasonable amount of time. Another common hyper-parameter is the drop-out rate. Drop-out is a standard regularization method for DL models proposed to reduce overfitting. In drop-out, a proportion of neurons are randomly removed, and the percentage of neurons to be removed should be tuned.

**Ensemble Algorithms.** In sklearn, the voting method can be set to be 'hard' or 'soft', indicating whether to use majority voting or averaged predicted probabilities to determine the classification result. The list of selected single ML estimators and their weights can also be tuned in certain cases. For instance, a higher weight can be assigned to a better-performing singular ML model in a voting model. When using bagging methods, the first consideration should be the type and number of base estimators in the ensemble, denoted by 'base estimator' and 'n estimators', respectively. Then, the 'max samples' and 'max features', indicating the sample size and feature size to generate different subsets, can also be tuned. In AdaBoost, the type of base estimator, 'base estimator', can be set to a decision tree or other methods. In addition, the maximum number of estimators at which boosting is terminated, 'n estimators', and the learning rate that shrinks the contribution of each classifier should also be tuned to achieve a trade-off between these two hyperparameters.

### 3.2.3 Complexity specification

An important aspect of any algorithm, and consequently any algorithm in the domains of ML and DM, is the computational complexity, which specifies the amount of resources required to run the given algorithm. Its importance stems from the fact that time and space are two of the most important considerations when seeking a practical solution to many computational problems, and so optimal resource usage is preferred.

The time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm, while the memory complexity describes the amount of memory space required to solve an instance of the computational problem. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount

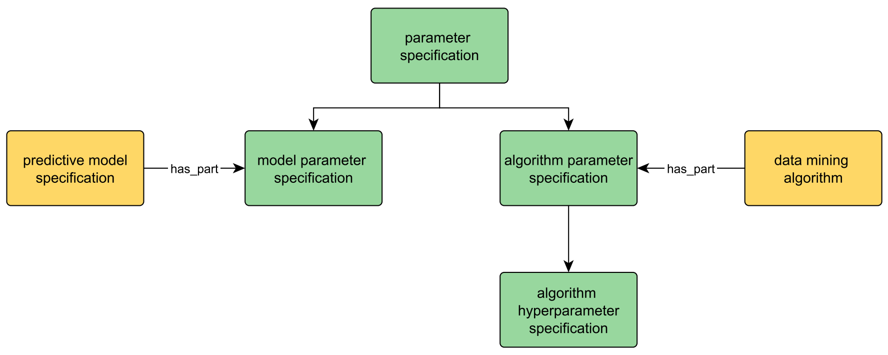

parameter specification
predictive model specification
has_part
model parameter specification
algorithm parameter specification
has_part
data mining algorithm
algorithm hyperparameter specification

Figure 3.4: Modelling of parameters in OntoDM-algorithms. The algorithm parameter/hyperparameter and model parameter entities in OntoDM-algorithms.

of time to perform. Thus, the amount of time taken, and the number of elementary operations performed by the algorithm is taken to be related by a constant factor.

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. Less commonly used is the average-case complexity, which is the average of the time taken on inputs of a given size (this makes sense because there are only a finite number of possible inputs of a given size).

In both cases, the time complexity is generally expressed as a function of the size of the input. Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases — that is, the asymptotic behavior of the complexity. Therefore, the time complexity is commonly expressed using big O notation. Algorithmic complexities are classified according to the type of function appearing in the big O notation. Space complexity shares many of the features of time complexity and serves as a further way of classifying problems according to their computational difficulty. The space complexity is typically estimated using asymptotic notation (Sipser, 1996).

The computational complexity is modeled in OntoDM-algorithms using the complexity specification entity which is connected to the data mining algorithm entity via the *has_part* relation. Following the theoretical basis, we split the complexity specification with respect to time and space, as well as with respect to whether the complexity is measured during training or run-time. As the computational complexity is expressed using a complexity function, we add a *has_part* relation between these two entities. The complexity function specification entity is further divided into sub-classes consisting of the most commonly encountered time complexities. The entities involved in this ontology schema fragment can be seen in Figure 3.5. To allow greater expressiveness, we add the annotation property "big O notation", using a maths notation written in latex. Each complexity function is accompanied by problems that can be solved using an algorithm with the given time complexity. The complexity functions added in the ontology can be seen in Table 3.1.

Consequently, we add both the time and memory complexity to the ontology. Apart from enabling an algorithm to have a specific complexity class, we also allow the complexity function to be expressed more explicitly, either by using a mathematical representation or a natural language representation. What this means is that algorithms that have the same complexity class can have a different mathematical representation of that class. Additionally, we divide the complexity specification into model training complexity and model prediction complexity. Model training complexity measures the time necessary for the al-

Table 3.1: Summarization of the time complexities included in OntoDM-algorithms.The time complexities are represented along with their running times (expressed with the big O notation) and example algorithms. Source: https://en.wikipedia.org/wiki/Time_complexity.

| Name | Running Time (T(n)) | Example Algorithms |
|---|---|---|
| constant time | $O(1)$ | Finding the median value in a sorted array of numbers |
| inverse Ackermann time | $O\big(\alpha(n)\big)$ | Amortized time per operation using a disjoint set |
| iterated logarithmic time | $O(\log^* n)$ | Distributed coloring of cycles |
| log-logarithmic | $O(\log \log n)$ | Amortized time per operation using a bounded priority queue |
| logarithmic time | $O(\log n)$ | Binary search |
| polylogarithmic time | $\text{poly}(\log n)$ | |
| fractional power | $O(n^c)$, where $0 < c < 1$ | Searching in a KD-tree |
| linear time | $O(n)$ | Finding the smallest or largest item in an unsorted array |
| "n log-star n" time | $O(n \log^* n)$ | Seidel's polygon triangulation algorithm |
| linearithmic time | $O(n \log n)$ | Fastest possible comparison sort |
| quasilinear time | $n\text{poly}(\log n)$ | |
| quadratic time | $O(n^2)$ | Bubble sort; Insertion sort; Direct convolution |
| cubic time | $O(n^3)$ | Naive multiplication of two n x n matrices. Calculating partial correlation |
| polynomial time | $2^{O(\log n)} = \text{poly}(n)$ | Karmakar's algorithm for linear programming |
| quasi-polynomial time | $2^{\text{poly}(\log n)}$ | Best-known O(log2n)-approximation algorithm for the directed Steiner tree problem sub-exponential time |
| (first definition) | $O(2^{n^\epsilon})$ for all $\epsilon > 0$ | Contains BPP unless EXPTIME (see below) equals MA |
| sub-exponential time | | |
| (second definition) | $2^{o(n)}$ | Best-known algorithm for integer factorization |
| exponential time | | |
| (with linear exponent) | $2^{O(n)}$ | Solving the travelling salesman problem using dynamic programming |
| exponential time | $2^{poly(n)}$ | Solving matrix chain multiplication via brute-force search |
| factorial time | $O(n!)$ | Solving the traveling salesman problem via brute-force search |
| double exponential time | $2^{2^{poly(n)}}$ | Deciding the truth of a given statement in Presburger arithmetic |

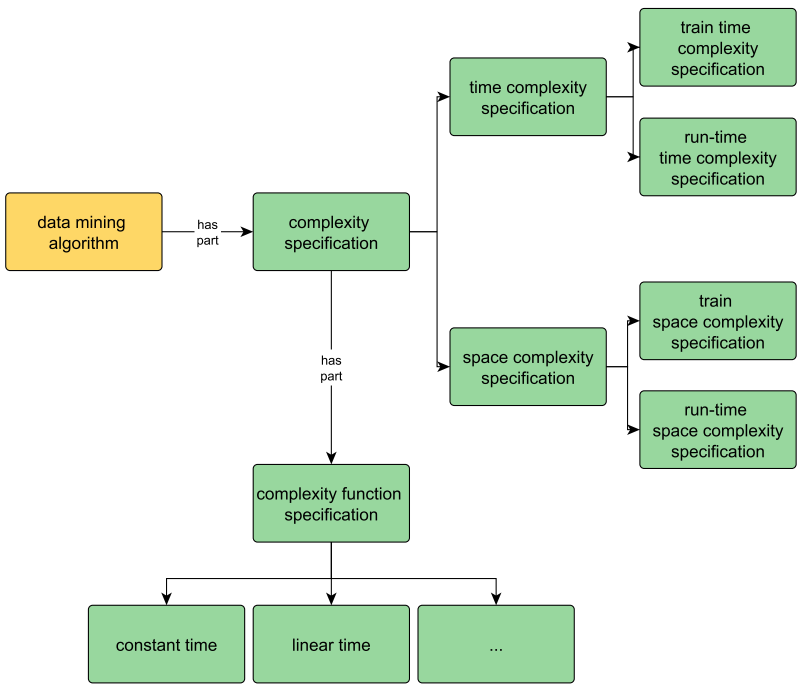

data mining algorithm
has part
complexity specification
time complexity specification
train time complexity specification
run-time time complexity specification
space complexity specification
train space complexity specification
run-time space complexity specification
has part
complexity function specification
constant time
linear time
...

Figure 3.5: Representation of computational complexity in OntoDM-algorithms. The core entities for representing the computational complexity of an algorithm. Entities with a yellow color represent imported entities, while the ones with green color represent newly added entities. Empty arrows represent is_a relations.

gorithm to train a specific predictive model, while model prediction complexity measures the time necessary for the trained model to make a single prediction. These measures can have the same values, but often that is not the case. The addition of the complexity function entities is an important improvement to the ontology since this information can have a crucial role in the use-case of a user choosing which algorithm to use given a specific set of available resources.

### 3.2.4 Algorithm assumption

An algorithm assumption is a hypothesis based on which an algorithm has been developed, and which should be true if the algorithm is to achieve the task it was designed to address (Keet et al., 2015). This means that the satisfaction of an assumption is a guarantee of the expected quality of the results produced by the ML/DM algorithm. In the case where the assumptions are violated, the algorithms would still produce results, which would have an inherently poor predictive capability. These assumptions are usually related to the data, e.g., the feature distribution, the dependency between the instances, etc. As ML is an empirical discipline, specific guidelines exist, which represent a less strict assumption. An example here would be the guideline that the hyperparameter $k$ in the Nearest Neighbors Classifier should be an odd number to avoid a tie. Another example is a neural network, which imposes no underlying assumptions on the data, while there are some general guidelines for more efficient learning, e.g., data standardization. In OntoDM-algorithms we include the algorithm assumption specification entity which is directly connected to the data mining algorithm via the has-part relationship. In Table 3.2 below, we present several common algorithm assumptions derived from the study of several supervised learning

Table 3.2: Examples of assumptions in ML/DM algorithms.

| Algorithm | Assumption |
|---|---|
| SVM | The examples are independent and identically distributed (IID), according to an unknown probability distribution. |
| Linear Regression | Linear relationship (the relationship between the independent and dependent variables must be linear) |
| Linear Regression | Normality of residuals (the residuals should follow a normal distribution) |
| Naive Bayes | Naive conditional independence assumption (between every pair of features given the value of the class variable) |
| Gaussian Naive Bayes | Each target class follows a Gaussian distribution |

algorithms, such as Ordinary Least Squares, SVM, Tree-based Models, KNN, etc.

### 3.2.5 Ensemble algorithms

As noted, in OntoDM-core, an ensemble algorithm is defined as a data mining algorithm that produces two or more generalizations as output. This entity is further divided into four ensemble algorithm types, based on the type of generalization that is produced as output. In our research, we define four types of ensemble algorithms, which are consistent with the OntoDM-core backbone to a certain extent. Specifically, we distinguish between bagging, boosting, voting, and stacking ensemble algorithms.

Bagging methods form a class of algorithms that build several instances of a black-box estimator on random subsets of the original training set and then aggregate their predictions to form a final prediction. Since bagging methods essentially constitute a sampling process, the Sampling entity is added to the ontology to better represent this algorithm type. This will be described in more detail in the following sections. Notable examples of bagging algorithms include: Pasting (Breiman, 1999), Bagging (Breiman, 1996), Random Subspaces (Ho, 1998) and Random Patches (Louppe & Geurts, 2012).

In boosting methods, base estimators are built sequentially and one tries to reduce the bias of the combined estimator. The motivation is to combine several weak models to produce a powerful ensemble. Notable examples of boosting algorithms include: AdaBoost (Freund & Schapire, 1997) and Gradient Tree Boosting (Friedman, 2001).

In voting methods, the goal is to combine conceptually different machine learning classifiers and utilize a voting procedure (hard/soft) to predict the class labels. Such a classifier can be useful for a set of equally well-performing models to balance out their individual weaknesses.

Finally, stacking is a method for combining estimators to reduce their biases (Wolpert, 1992). More precisely, the predictions of each estimator are stacked together and used as input to a final estimator to compute the prediction.

In the ontology, we model the four described ensemble algorithm types as children of the entity predictive modeling ensemble algorithm. This is true for most bagging and boosting algorithm instances since most of them rely on decision trees as base estimators (which are predictive modeling entities). However, in the voting and stacking scenarios, this could be debatable, since some model-free algorithms like Naive Bayes and Nearest Neighbors can also be constituent parts of these algorithm types. In practice, the voting and stacking scenarios usually include at least one predictive modeling algorithm, and so the pragmatic decision was made to model in this specific way. Alternatively, we could model

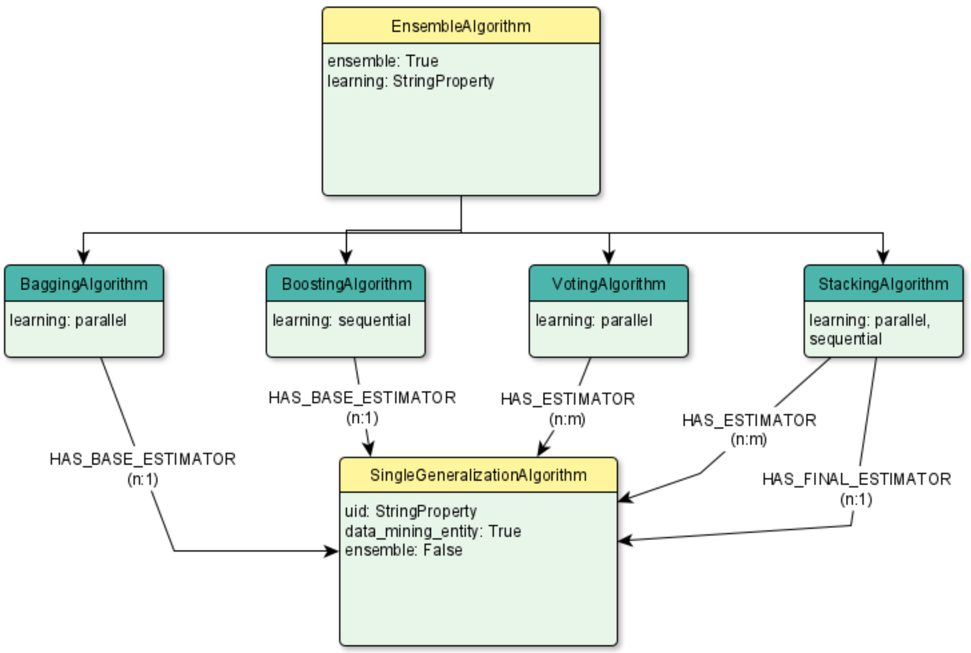


Figure 3.6: Modelling of ensemble algorithms in OntoDM-algorithms.

the four described ensemble algorithm types as direct children of the ensemble algorithm, however, this would mean digressing from the OntoDM-core modeling practices. The different ensemble algorithms and the connection with the single generalization algorithms can be seen in Figure 3.6.

### 3.2.6 Sampling

Sampling is a process used in statistical analysis in which a predetermined number of observations are taken from a larger population. The sampling process can be used in different use cases to achieve different objectives. In a model validation scenario, a dataset can be sampled in training and testing splits to validate the performance of an ML/DM model. In an ensemble algorithm, multiple models are trained on subsets of the dataset, obtained through a sampling process. This is especially present in bagging and boosting algorithms, such as Bagging (Breiman, 1996) and Stochastic Gradient Boosting (Friedman, 2002).

In this thesis, we will explore the sampling process from a perspective covering the second use case. The sampling entity is connected to the already existing data mining algorithm and dataset entities via the has-part relationship. We distinguish between four sampling types, distinguished based on three dimensions: whether the sampling is made on the features (columns in a dataset), or examples (rows in a dataset), whether replacement is used or not, and the sampling distribution. Notable instances of the sampling entity include:

- Sampling random subsets of the samples without replacement (Pasting),
- Sampling random subsets of the samples with replacement (Bagging, Stochastic Gradient Boosting),
- Sampling random subsets of the features without replacement (Random Subspaces),

- Sampling random subsets of the features without replacement (Random Patches).

## 3.3 Implementation of the OntoDM-algorithms Extension

In the development process, we relied on several tools. For automating ontology development tasks we used the ROBOT tool (a recursive acronym for "Robot is an OBO Tool"). It provides ontology processing commands for a variety of tasks, including commands for converting formats, running a reasoner, creating import modules, running reports, and various other tasks (Jackson et al., 2019). Specifically, we used ROBOT for extracting subsets of terms that were necessary for representing an ML/DM algorithm. These terms provided a backbone of the OntoDM-algorithms ontology extension and ensured that it could be easily plugged into the OntoDM ontology, which covered the data mining process from a broader perspective.

The ROBOT tool offers several extraction methods, depending on which classes concerning the terms in question are needed (i.e., super-classes, sub-classes, etc.). Upon experimenting with the settings of this method, we discovered that the method yielding the optimal results was the Minimum Information to Reference an External Ontology Term method (MIREOT) (Courtot et al., 2011). The MIREOT method preserves the hierarchy of the input ontology (subclass and subproperty relationships) but does not try to preserve the full set of logical entailments. Hence, even though the relationship entities were exported properly, the axioms comprised of class entities and relationship entities were not exported. We curbed this limitation by manually adding the axioms, which was not exceedingly time-consuming because of the size of the extracted ontology. Once the ontology extraction was completed and validated, the extracted terms were imported into the Protégé ontology editor, where the rest of the ontology development process was carried out (Musen, 2015).

OntoDM-algorithms is expressed in OWL-DL, a de facto standard for representing ontologies. It has 361 classes, 300 of which are external, i.e., reused from the referenced ontologies, while 61 are newly added. The ontology is regularly maintained and publicly accessible via the following Git repository: https://github.com/lidija-jovanovska/ontologies. It is also a part of the BioPortal repository of biomedical ontologies, at https://bioportal.bioontology.org/ontologies/ONTODM-ALGORITHM. Since BioPortal does not support integration with Git, there is a possibility of these two versions diverging if they are not updated properly. Hence, we have to maintain version consistency by manually updating both repositories using the BioPortal REST API. An alternative option for ensuring consistency is by using the Persistent uniform resource locator (PURL) that is used to redirect to the location of the requested web resource (the latest stable version of the ontology).

# Chapter 4

# Web-Based Application for Annotating, Storing and Querying ML/DM Algorithms

In this chapter, we present a web-based application we designed and implemented that enables the semantic annotation of ML/DM algorithms, storage of the annotated metadata, and querying of the repository. The infrastructure of the application is divided into two functional units — the semantic annotation tool and the semantic querying tool. Here, we first describe the system architecture by focusing on the system requirements and the technologies used to build the system. Next, we focus on the implementation details where we describe the developed services and the user interface (UI). Finally, we discuss the population of the graph database with semantic annotations of a selection of algorithms.

## 4.1 System Architecture of the Web-based Application

In this section, we will describe the main technologies and software used to build the web application. The key parts of the system are the database where we store the annotations created by the users, the back-end framework which handles how data is accessed and written in the database as well as the front-end framework, which is used for developing the UI, through which the user interacts with the system. The proposed architecture of the system is presented in Figure 4.1, where one can see the technologies that are used at each level; from the graph database (Neo4j), the back-end technologies (Django and Django REST Framework), the front-end technologies (React, MaterialUI, etc.), as well as the technologies which enable the communication between the system layers (Axios and Neomodel).

### 4.1.1 Software requirements

Before we move to the implementation, we must first define the software requirements which define the functional units of the system. The first requirement is that the system should include an annotation tool used to semantically annotate ML/DM algorithms. The tool should contain input fields defined using an annotation schema based on the OntoDM-algorithms ontology extension. Using this annotation tool the user is supposed to input information related to an ML/DM algorithm and be able to store the algorithm metadata in the repository. The second requirement of the system is a querying tool, which should contain input fields that serve as constraints to the repository search. Using the querying tool, the user is supposed to retrieve relevant information regarding ML/DM algorithms

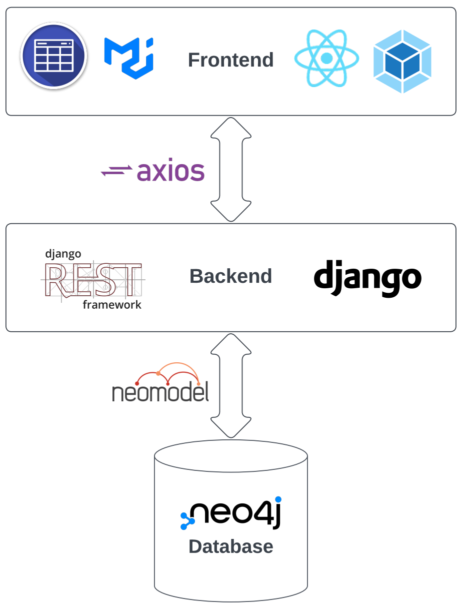


Figure 4.1: The architecture of the system for semantic annotation and querying.

that are stored in the repository. After the development process, the application will be evaluated through several annotation scenarios and querying use cases described in Section 4.4 and Chapter 5.

### 4.1.2 Database management system: Neo4j and Cypher

Neo4j (“NEO4J Graph Data Platform,” 2022) is a graph database management system, implemented in Java and accessible from software written in other languages using the Cypher query language through a traditional HTTP endpoint, or through the binary "Bolt" protocol (“Bolt protocol,” 2015).

The Neo4j architecture is designed for optimal management, storage, and traversal of nodes and relationships. The graph database takes a property graph approach, which is beneficial for both traversal performance and operations runtime. The main elements in a property graph database model are *nodes* which describe entities of a domain of interest and *relationships* which describes a connection between a source node and a target node. What is more, nodes and relationships can have properties (key-value pairs), which further describe them. An example Neo4j schema consisting of several node types and relationships can be seen in Figure 4.2.

Cypher (“Cypher query language,” 2022) is a declarative query language for property graphs. It was created for Neo4j to allow users to store and retrieve data from the graph database. It is a declarative, SQL-inspired language for describing visual patterns in graphs using ASCII-art syntax. The syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. An example Cypher query is shown in Figure 4.3. The query is used to retrieve all the nodes that are connected to the Person node with the name property set to ’Dan’, via the LOVES relationship.

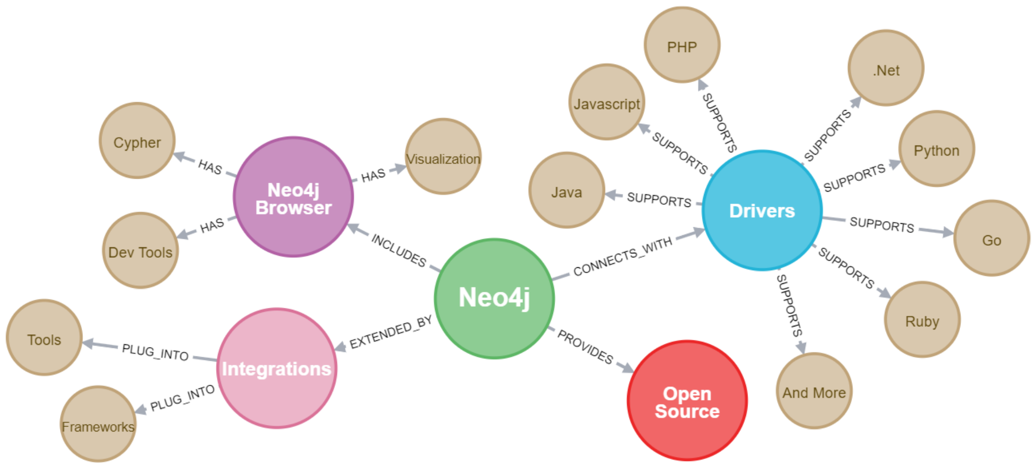


Figure 4.2: An example Neo4j graph schema.

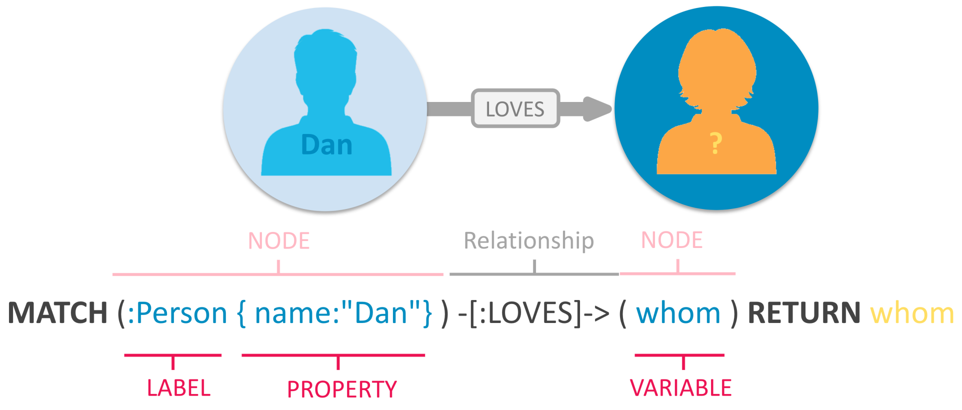

Dan
LOVES
?
NODE
Relationship
NODE
MATCH (:Person { name:"Dan"} ) -[:LOVES]-> ( whom ) RETURN whom
LABEL
PROPERTY
VARIABLE

Figure 4.3: An example Cypher query.

### 4.1.3 Back-end technologies: Django and Neomodel

In essence, the developed application is a Django application. Django is a free and open-source web development framework written in Python. The framework follows the Model-View-Controller (MVC) architecture. It consists of an object-relational mapper (ORM) that mediates between data models (defined as Python classes) and a relational database (Model), a system for processing HTTP requests with a web templating system (View), and a regular-expression-based URL dispatcher (Controller). The native format for the Model aspect of a Django application is a relational database, which although well-suited for transactional data, is not the chosen data model in our scenario.

Hence, since we store the data in a graph database, we need an external library to handle the communication between the back-end and the graph database. For this purpose, we utilize Neomodel, a library that serves as an Object Graph Mapper (OGM) for the Neo4j graph database ("NEO4J Graph Data Platform," 2022), built on the Neo4j driver. It uses Django model style definitions and boasts a powerful query API, schema expressivity with cardinality restrictions, full transaction support, hooks, and most importantly integration with the Django application.

### 4.1.4 Front-end technologies: React and MaterialUI

While Django templates can be useful in cases where data is merely presented to the user, the large degree of interactivity in our application made us opt for a more powerful tool in the form of a front-end framework instead. We chose to use React ("React – A JavaScript library for building user interfaces," 2022), a free and open-source library written in JavaScript, which is mainly used for building UIs based on UI components. It follows the declarative programming paradigm — where the structure and elements of computer programs are written so that the logic of computation is expressed, without describing its control flow. So, developers design views for each state of an application, and React updates and renders components when data changes. Its main feature is that its code is made of entities called components, which can be rendered to a particular element in the Document Object Model (DOM) using the React DOM library.

The two primary ways of declaring components in React are via function components and class-based components. Function components are declared with a function that then returns some JavaScript XML (JSX). Class-based components are declared using ES6 classes and unlike function components, they have an internal state. An interesting feature of React is hooks – functions that let developers "hook into" React state and life cycle features from function components. React components can communicate with the back-end through an HTTP client, such as Axios ("Axios — Promise based HTTP client for the browser and node.js," 2020).

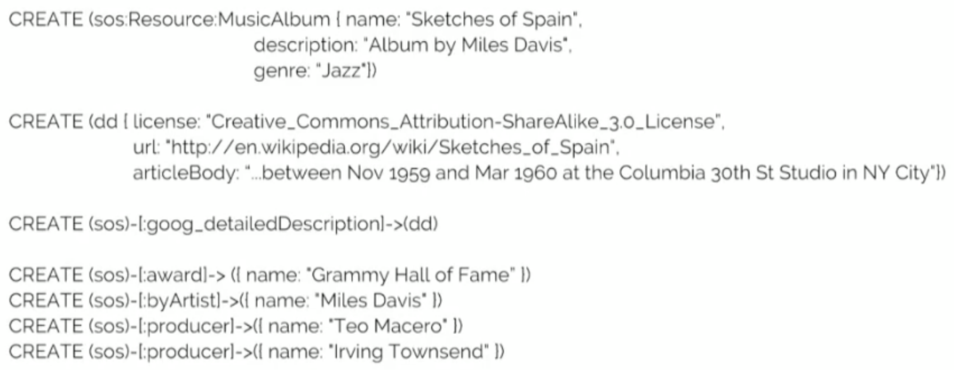
The design of the UI was created using Material UI — a library of React components that implements Material Design principles ("Material UI," 2022). Material Design is a design language that was first introduced by Google in 2014 ("Material design," 2022). It is a visual language that makes use of grid-based layouts, responsive animations and transitions, padding, and depth effects such as lighting and shadows. These components work in isolation, which means they are self-supporting and will inject only the styles they need to display.

### 4.1.5 Graph-like data models: property graphs and the RDF data model

While the relational model can handle simple cases of many-to-many relationships, as the connections in the data grow more complex, it becomes more natural to model the data as a graph. Graph-based schemas are also good for extension: as more features are added to the application, a graph schema can easily be extended to accommodate changes in the application's data structure.

A graph consists of two kinds of objects: vertices (also known as nodes or entities) and edges (also known as relationships or arcs). Many kinds of data can be modeled as a graph. Typical examples include social graphs (vertices are people, and edges indicate which people know each other), the web graph (vertices are web pages, and edges indicate HTML links to other pages), road or rail networks (vertices are junctions, and edges represent the roads or railway lines between them). While all the provided examples represent homogeneous graphs, the modeling domain usually consists of different kinds of objects, which are better modeled using heterogeneous graphs. There exist several models which structure data in graphs, the most prominent of them being property graphs and the RDF data model. Examples of both models and the syntax used to describe the same graph in both models can be seen in Figure 4.4.

In the property graph model, each vertex consists of a unique identifier, a set of outgoing edges, a set of incoming edges, and a collection of properties (key-value pairs). Each edge consists of a unique identifier, the vertex at which the edge starts (the tail vertex), the vertex at which the edge ends (the head vertex), a label to describe the kind of relationship



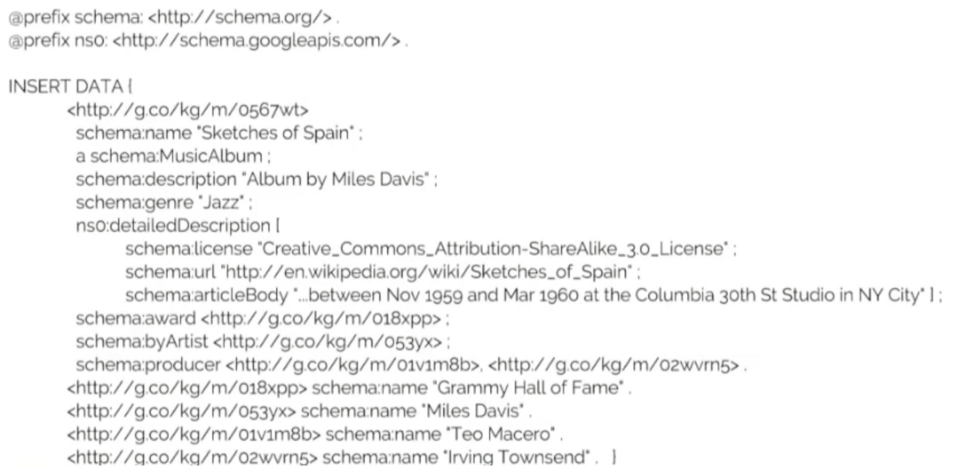
```
CREATE (sos:Resource:MusicAlbum { name: "Sketches of Spain",
                                  description: "Album by Miles Davis",
                                  genre: "Jazz"})

CREATE (dd { license: "Creative_Commons_Attribution-ShareAlike_3.0_License",
             url: "http://en.wikipedia.org/wiki/Sketches_of_Spain",
             articleBody: "...between Nov 1959 and Mar 1960 at the Columbia 30th St Studio in NY City"})

CREATE (sos)-[:goog_detailedDescription]->(dd)

CREATE (sos)-[:award]-> ({ name: "Grammy Hall of Fame" })
CREATE (sos)-[:byArtist]->({ name: "Miles Davis" })
CREATE (sos)-[:producer]->({ name: "Teo Macero" })
CREATE (sos)-[:producer]->({ name: "Irving Townsend" })
```

(a) An example of a Property Graph described using the Cypher syntax.



```
@prefix schema: <http://schema.org/> .
@prefix ns0: <http://schema.googleapis.com/> .

INSERT DATA {
        <http://g.co/kg/m/0567wt>
         schema:name "Sketches of Spain" ;
         a schema:MusicAlbum ;
         schema:description "Album by Miles Davis" ;
         schema:genre "Jazz" ;
         ns0:detailedDescription [
                schema:license "Creative_Commons_Attribution-ShareAlike_3.0_License" ;
                schema:url "http://en.wikipedia.org/wiki/Sketches_of_Spain" ;
                schema:articleBody "...between Nov 1959 and Mar 1960 at the Columbia 30th St Studio in NY City" ] ;
         schema:award <http://g.co/kg/m/018xpp> ;
         schema:byArtist <http://g.co/kg/m/053yx> ;
         schema:producer <http://g.co/kg/m/01v1m8b>, <http://g.co/kg/m/02wvrn5> .
        <http://g.co/kg/m/018xpp> schema:name "Grammy Hall of Fame" .
        <http://g.co/kg/m/053yx> schema:name "Miles Davis" .
        <http://g.co/kg/m/01v1m8b> schema:name "Teo Macero" .
        <http://g.co/kg/m/02wvrn5> schema:name "Irving Townsend" .  }
```

(b) An example of RDF data serialized using the Turtle syntax.

Figure 4.4: The data is described using the two data models: property graphs (Cypher syntax) and RDF (Turtle syntax). Adapted from (Howard, 2017).

between the vertices, and a collection of properties (key-value pairs). Some important aspects of this model are: Given any vertex, you can efficiently find both its incoming and its outgoing edges, and thus traverse the graph both forward and backward; By using different labels for different kinds of relationships, you can store several different kinds of information in a single graph, while still maintaining a clean data model.

RDF is closely tied to the vision of the semantic web, which is fundamentally a simple and reasonable idea: websites already publish information as text and pictures for humans to read, so why do they not also publish information as machine-readable data for computers to read (Matthews, 2005)? The Resource Description Framework (RDF) ("RDF — Semantic Web Standards," 2022) was intended as a mechanism to address this challenge, allowing data from different websites to be automatically combined into a web of data – a form of internet-wide "database of everything". Triples can serve as a good internal data model for applications, even when there is no interest in publishing RDF data on the semantic web.

RDF statements can be written in a Turtle language syntax, which is more human-readable, or in an XML format, which does the same thing much more verbosely. RDF has a few quirks because it is designed for internet-wide data exchange. The subject, predicate, and object of a triple are often Universal Resource Identifiers (URIs). For example, a predicate might be a URI such as http://myapp.com/namespace#has_part or http://myapp.com/namespace#solves, rather than just HAS_PART or SOLVES. The reasoning behind this design is that you should be able to combine your data with someone else's data, and if they attach a different meaning to the word has_part or solves, you will not get a conflict because their predicates are stored as part of a different namespace, for example, http://other.org/foo#has_part and http://other.org/foo#solves.

The RDF data model is identical to the property graph model. Some of the main differences are that the RDF data model does not uniquely identify instances of relationships of the same type, i.e., it does not support the addition of attributes to the relationships. These shortcomings are commonly handled via modeling workarounds. In RDF, all information is stored in the form of very simple three-part statements named triples: (subject, predicate, object). For example, in the triple (Jim, likes, bananas), Jim is the subject, likes is the predicate (verb), and bananas is the object. The subject of a triple is equivalent to a vertex in a graph. The object is one of two things:

1. A value in a primitive datatype, such as a string or a number. In that case, the predicate and object of the triple are equivalent to the key and value of a property on the subject vertex. For example (Lucy, age, 33) is identical to having a vertex Lucy with properties "age": 33.

2. Another vertex in the graph. In that case, the predicate is an edge in the graph, the subject is the tail vertex, and the object is the head vertex. For example, in (Lucy, marriedTo, Alan) the subject and object Lucy and Alan are both vertices, and the predicate marriedTo is the label of the edge that connects them.

## 4.2 Implementation of the Services for Annotation and Querying

### 4.2.1 Service setup

First, using the Neo4j Desktop Application we created a new project instance and a Database Management System (DBMS), where we created our graph database. After setting the required database access credentials (username and password), the IP address and the corresponding ports are provided (i.e., Bolt, HTTP, and HTTPS ports). By default, Neo4j uses the Bolt application protocol, which is generally carried over a regular TCP or WebSocket connection (Forouzan, 2002). The database URL through which we can set up the connection between the application and the database has the following format: 'bolt://username:password@localhost:7687'. This URL is added to the settings file in the Django application using the neomodel config utility.

Additionally, in the settings file, we make use of the neomodel install_labels script, which installs the indexes and constraints for the entire schema at compile time. This should increase the memory overhead, however, it would reduce the time overhead to query the database. We use the default settings for the remaining Django configuration. For rendering the views, we rely on the Django REST Framework class APIView which, like Django's native View class, handles Requests from the front-end and returns adequate Responses.

### 4.2.2 Database conceptual model design

The database conceptual model design was based on the ontology annotation schema. The design followed an object-oriented approach, where we modeled ontology entities as classes by instantiating the neomodel StructuredNode class (i.e., nodes in the graph database), while relationships were created by instantiating the neomodel Relationship class. Note that we model the entities explicitly by setting node-level properties, while for the relationships we only define the head and tail nodes, although the property graph model allows for the definition of relationship-level properties. It is important to note that Neo4j reuses its internal ids when nodes and relationships are deleted. This means that applications using, and relying on internal Neo4j ids, are brittle or at risk of making mistakes. Because of this, we use application-generated ids by using neomodel utilities (UniqueIdProperty).

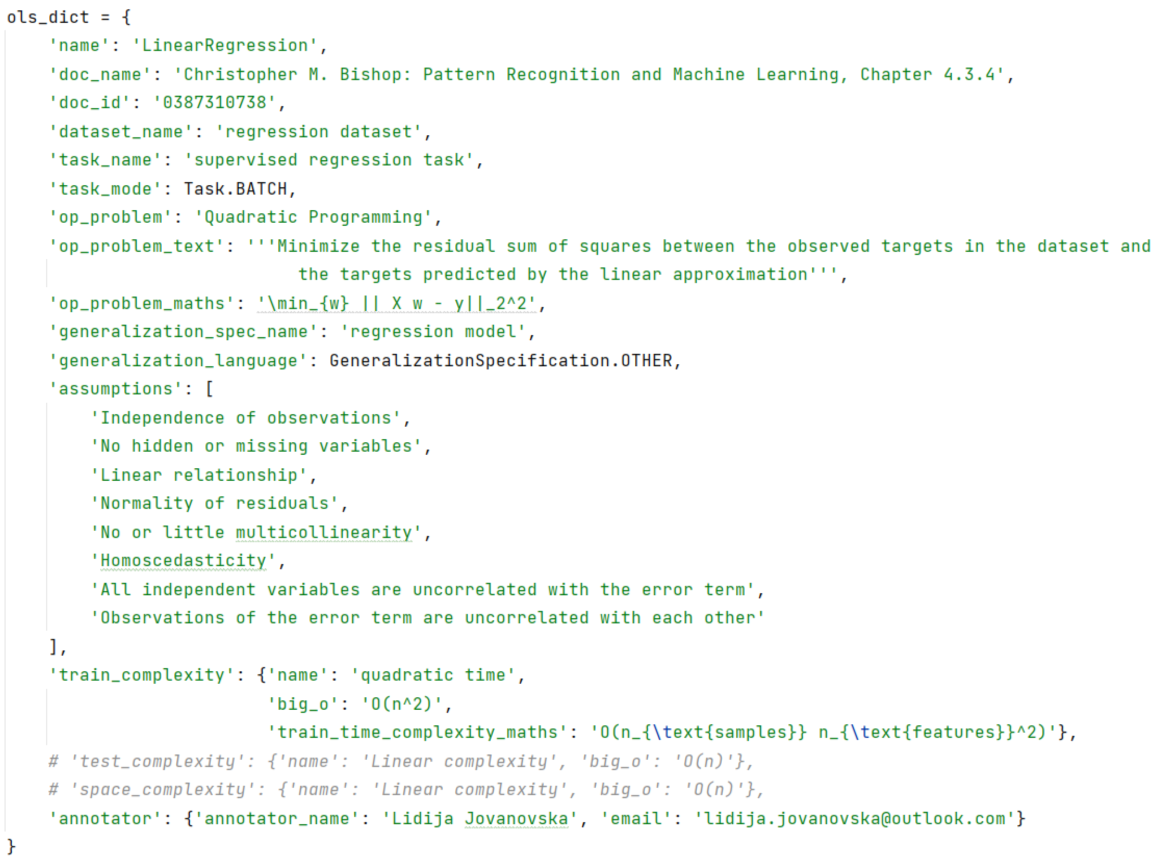
Since the web application was not designed to have a user administration system, we represent each annotation session with the Annotation class. Some of the properties which are defined and explicitly input by the user are the Annotator name and surname, affiliation, and email. Each annotation object has a connection to the algorithm which was annotated in the session. An inverse relationship is also added in the DataMiningAlgorithm class, allowing for bi-directional traversal.

### 4.2.3 Email validation

The Annotation class also includes internal properties used by the application to validate the annotation. This validation step was essential to provide accountability for the quality of the annotation and to ensure that the personal information input by the user is accurate and verifiable. Once the e-mail is entered by the user, it is passed through a validation scheme using Django's built-in Email Validator. The validation operates by matching the email string to several regular expression patterns, such as the domain, the existence, and position of the "@" character, etc. If the e-mail is not valid, the annotation is not stored in the database and a HTTP_412_PRECONDITION_FAILED response is returned. If the e-mail is valid, and all annotation quality validation criteria are satisfied (i.e., algorithm name is input), the annotation is saved, however, an Annotation object is instantiated with the is_verified property set to False. Meanwhile, a unique token is generated, along with a link, which is sent to the user via the provided e-mail. Once the user opens the link, they are redirected to the main page, and the is_verified property is set to True. The e-mails which are not validated, along with the corresponding annotations, can then be removed collectively using a scheduled script on a specific time frame.

### 4.2.4 Annotation service

We essentially have two main services in the application, implemented through two APIView objects — the Annotation service (which validates and stores the annotation), and the Querying service (which retrieves the queried data). Several composite services retrieve data from the database and display the data as selection options in the Annotation service. These composite services are mainly used when annotating ensemble algorithms. The Annotation service receives a request from the front-end, through which the annotation data is passed as a payload. An example of the annotation data can be seen in Figure 4.5.



```
ols_dict = {
    'name': 'LinearRegression',
    'doc_name': 'Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4',
    'doc_id': '0387310738',
    'dataset_name': 'regression dataset',
    'task_name': 'supervised regression task',
    'task_mode': Task.BATCH,
    'op_problem': 'Quadratic Programming',
    'op_problem_text': '''Minimize the residual sum of squares between the observed targets in the dataset and
                          the targets predicted by the linear approximation''',
    'op_problem_maths': '\min_{w} || X w - y||_2^2',
    'generalization_spec_name': 'regression model',
    'generalization_language': GeneralizationSpecification.OTHER,
    'assumptions': [
        'Independence of observations',
        'No hidden or missing variables',
        'Linear relationship',
        'Normality of residuals',
        'No or little multicollinearity',
        'Homoscedasticity',
        'All independent variables are uncorrelated with the error term',
        'Observations of the error term are uncorrelated with each other'
    ],
    'train_complexity': {'name': 'quadratic time',
                         'big_o': 'O(n^2)',
                         'train_time_complexity_maths': 'O(n_{\text{samples}} n_{\text{features}}^2)'},
    # 'test_complexity': {'name': 'Linear complexity', 'big_o': 'O(n)'},
    # 'space_complexity': {'name': 'Linear complexity', 'big_o': 'O(n)'},
    'annotator': {'annotator_name': 'Lidija Jovanovska', 'email': 'lidija.jovanovska@outlook.com'}
}
```

Figure 4.5: The annotation metadata for the LinearRegression algorithm.

To ensure a baseline for sufficient annotation information, we define a set of required fields; i.e., fields that must have user input, so that the annotation can be processed and stored. These fields include key aspects of an algorithm — the name, the dataset type which can be processed by the algorithm, the task it addresses, as well as the generalization it produces. Furthermore, the email of the annotator has to be input, for the e-mail verification procedure to be run in the next step. Since the annotator is not obliged to fill out every field in the annotation form, some of the information may be missing. This is verified for every node to avoid adding blank nodes in the database. Another verification procedure covers duplicate handling.

For some entities, like the Document entity, we check if an instance with the same information (document name and document id) already exists in the database. If not, we add a new Document instance, along with a connection between the DataMiningAlgorithm instance and the Document instance, while if it exists, we only add the connection. The duplicate handling procedure is not run for other entities, like the Parameter entity. This decision seems intuitive since Parameter names are descriptive and often serve different purposes for different algorithms.

An important design pattern is that not all nodes are instantiated in the same way. For example, the Dataset taxonomy is defined in the ontology-based annotation schema, and the user can only specify the type of dataset that the annotated algorithm can take as input, and not specify a dataset instance of the type which is selected. In other words, in this scenario we do not instantiate the Dataset class, but rather just add a connection between the algorithm and the selected Dataset type. Specifying a Dataset instance can be easily implemented, however, it is a matter beyond the specification level of algorithm design we are covering. In the end, once all the information from the Request is processed and validated, the annotation is stored in the database.

### 4.2.5 Querying service

Unlike the Annotation service, the Querying service does not cover any validation procedures, because the user does not input new data, but merely specifies several filters based on several entities. In this case, we implemented the options to query with respect to the type of task that the algorithm addresses, the type of optimization problem it solves, as well as the training time complexity.

The queries can be done either through the Neomodel querying API, which ultimately gets compiled into a Cypher query or by simply writing Cypher queries as strings and executing them using the Neomodel Cypher utilities. The latter option is in theory much faster, since it skips the compilation step, and despite the database being too small for computation time to be seriously considered, it is wise to have the scaling factor in mind. The three available filters relate to the task, the optimization problem, and the training time complexity. The user can specify any combination of the filters (i.e., one, none, all), and each scenario is handled by the service accordingly.

For each filter, we run one query based on the condition if the filter is specified or not. So, for example, if the Task type is specified, we filter all the algorithms that are connected to the corresponding Task type. If it is not specified, we simply retrieve all the algorithms in the database which have a connection to any Task type. For example, the Cypher query that retrieves the DataMiningAlgorithm instances that address a specific Task (as set by the user) is presented in Figure 4.6.

```
MATCH (n:DataMiningAlgorithm)-[:ADDRESSES]->(t:Task)
            WHERE t.name="{task}"
                              RETURN n
```

Figure 4.6: A Cypher query used in the application. The query is used to retrieve all the DataMiningAlgorithm instances that address a Task type specified by the user.

The process is repeated for each filter. The results from all the queries are aggregated via an intersect operation, i.e., we retrieve only the algorithms that satisfy all of the filters. We define which properties are passed back to the front-end in a function defined for the DataMiningAlgorithm class tagged with a property decorator which returns a dictionary of the necessary properties and the corresponding values.

### 4.2.6 Semantics

In Chapter 3, we described how the ontology was built, and in Section 4.2.2, we described the graph database conceptual model. Here, we explain how these two could be bridged to allow the storage of the annotated data in a semantic format, as well as utilizing the logic defined in the ontology to infer new knowledge. For this we turn to Neosemantics, a plugin that enables the use of RDF and its associated languages and schemas (e.g., OWL (“OWL — Semantic Web Standards,” 2012) and RDFS (“RDFS — Semantic Web Standards,” 2004)) in Neo4j. Its functionalities include the import and export of RDF in multiple formats, model mapping on import/export of ontologies and taxonomies, graph validation based on SHACL constraints, and inferencing utilities (“SHACL — Semantic Web Standards,” 2017). Here, we overview each of the functionalities we utilized, presented textually and using the Cypher syntax in Table 4.1.

**Importing the OntoDM-algorithms ontology.** Before the ontology is imported, we must set up some configuration parameters. First, we have to create a constraint that the Unique Resource Identifier (URI) property of each entity in the ontology is unique. Next, we have to initialize the graph configuration. An example of one parameter which can be set covers defining how to handle the vocabulary URIs (i.e., if we select shorten, '<http://www.ontodm.com/OntoDM-core/algorithms>' is mapped to 'algorithms', which makes the imported data more readable). An extended description of all the configuration parameters is presented at the following link https://neo4j.com/labs/neosemantics/4.0/reference/.

In our case, we use the default parameter configuration. After the configuration is done, we import the ontology as an RDF graph. This fetches the ontology from the provided link, which requires an internet connection, but is also handy since it is publicly available, and consistency is ensured. The ontology is imported in the Turtle format which represents information using semantic triples that comprise a subject, predicate, and object. Upon inspection, the ontology is imported as expected, with one remark. Namely, when developing an ontology, it is common practice to include a small set of instances, which provide an annotation guide to anyone who would be using the ontology. However, when we instantiated individuals externally (in Protégé), they were correctly imported as nodes in Neo4j, whereas the instantiation relationship (rdfs_type) was not present between the ontology entities and the instances.

**Exporting the graph database conceptual model as an ontology.** It is possible to export the Graph database conceptual model in the form of an OWL Ontology. The same output produced by the db.schema() procedure can be generated as RDF/OWL through the /onto method. The /onto method will run db.schema() on your Neo4j graph and will generate owl:Class definitions for each label found, and owl:ObjectProperty definitions for each relationship along with rdfs:domain and rdfs:range based on the labels of their start and end nodes. It is possible to set the serialization format by setting corresponding parameters.

**Exporting the populated graph database in RDF.** Besides exporting the graph database conceptual model, we can also export the populated database in the RDF format. Neosemantics implements different ways to specify what we want to export, i.e., by using a node id or URI, or by label and property value, etc. Finally, the most powerful way of selecting the portion of the graph that we want to serialize as RDF would be to use Cypher. In this case, it is a POST request that takes as payload a JSON map with at least one cypher key having as its value the query returning the graph objects (nodes with their properties and relationships) to be serialized. We can also specify the format by specifying the "format" key in the JSON map.

Table 4.1: The main commands from the neosemantics module in Neo4j.

| Command | Use-case |
|---|---|
| CALL n10s.graphconfig.init(); | Initialize graph configuration parameters |
| CREATE CONSTRAINT n10s_unique_uri<br>ON (r:Resource)<br>ASSERT r.uri IS UNIQUE; | Create uniqueness constraint on each resource |
| CALL n10s.rdf.import.fetch(<ontology_uri>, 'Turtle') | Import ontology from URI as RDF using Turtle format |
| :GET http://localhost:7474/rdf/neo4j/onto | Export graph database conceptual model as OWL ontology |
| :POST http://localhost:7474/rdf/neo4j/cypher<br>{"cypher":<br>"MATCH (n) OPTIONAL MATCH (n)-[r]-()<br>RETURN n, r", "format": "Turtle"} | Export graph database as RDF using Turtle format |
| MATCH<br>(doc:Document),<br>(class:owl__Class{uri: "http://purl.obolibrary.org/obo/IAO_0000310"})<br>MERGE (doc)-[r:rdfs_type]->(class)<br>RETURN * | Add the rdf:type relationship between instances of the Document class and the OWL class from the imported ontology |

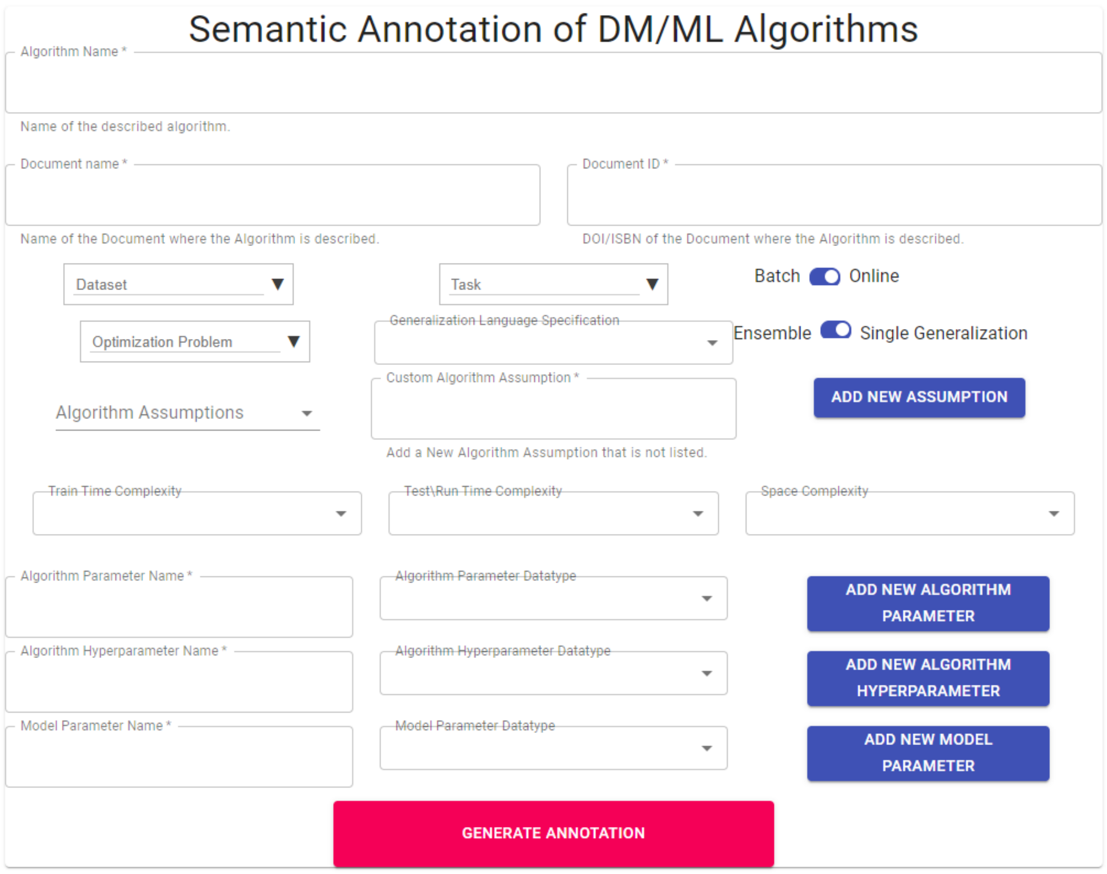
**Setting the annotations as instances of the ontology.** To harvest the possibilities of inference using the imported ontology, we have to connect the annotated data to the corresponding entities in the ontology. In RDF, this is done by using the 'rdf:type' (www.w3.org/1999/02/22-rdf-syntax-ns#type) property which ties an individual to a class of which it is a member. It is an instance of the 'rdf:Property' and it represents a triple of the form 'R rdf:type C', i.e., C is an instance of rdfs:Class, and R is an instance of C. The 'rdfs:domain' of 'rdf:type' is 'rdfs:Resource', while the 'rdfs:range' of 'rdf:type' is 'rdfs:Class'. We manually define which entity from the graph database conceptual model gets mapped to which ontology entity by writing a Cypher query for each entity type. This mapping is performed after an annotation is added to the graph database.

## 4.3 Implementation of the Graphical User Interface for ML/DM Algorithm Annotation and Search

The initial Graphical User Interface (GUI) was implemented as a one-page annotation form that contained the selected set of entities from the ontology-based annotation schema. The GUI design was presented in our paper published at the MIPRO conference in 2021 (Jovanovska & Panov, 2021). The Annotation tool was initially implemented as a single class component that contained multiple MaterialUI-native components, such as Grid, TextField, Select, Button, etc. For the case when there was a hierarchy of options, and selecting an option at one level, meant also including all the descendants of the selected option, coupled with the possibility for multiple selections, we made use of an external component. The first version of the Annotation GUI can be seen in Figure 4.7.

While the initial GUI design served its purpose of providing a proof-of-concept design, it was lacking interactivity, and some other design patterns, such as a more semantically organized interface, delineated sections, and navigation. These aspects were key to improving the user experience. In the following iteration, we opted for a more modular design, where similar-themed inputs would be grouped into sections and visually separated from each other, progressing from more general (high-level) information to a more in-depth description. For this purpose, we settled on a horizontal tab navigation design, located in the top part of the screen, as can be seen in Figure 4.8. Each tab consisted of information that referred to a specific aspect of an algorithm description. For example, the Complexity tab contained input fields related to the time and space complexity, while the Parameters tab comprised of fields related to the type, name, and datatype of the parameters related to the algorithm. Another notable addition in this version was the MaterialTable component which allowed us to better structure and manipulate the Documents and Parameters aspects. With the use of this component, multiple attributes could be specified for the given entity, each entry being simultaneously added to a table presented in the same tab. The table component came packed with navigation features such as a search bar, a page view, multiple selections, and removal functionality, as well as exporting into several supported formats.

In this iteration, we switched from using mostly class components to using only functional components. Class components are "stateful" components as they tend to implement logic and state. React lifecycle methods (i.e., componentDidMount) can be used inside class components. A Class-based development is considered outdated in the React developers community because they become unreadable if the logic is stored in several places and scaling is not trivial, as testing becomes harder. Today, the mainstream development practice relies on functional components, which are basic JavaScript functions. They are sometimes referred to as "stateless" components as they simply accept data and



Name of the described algorithm.

Name of the Document where the Algorithm is described.

DOI/ISBN of the Document where the Algorithm is described.

Add a New Algorithm Assumption that is not listed.

Figure 4.7: The first version of the GUI for the annotation tool.

display them in some form; that is, they are mainly responsible for rendering UI and are typically presentational only. Additionally, the introduction of hooks in React 16.8 made it significantly easier to manage variable and component states, like with for example the useState hook. Hooks let you split one component into smaller functions based on what pieces are related (such as setting up a subscription or fetching data), rather than forcing a split based on lifecycle methods. They essentially let you use more of React's features without classes while allowing for code reuse and better code organization.

In the third and final iteration of the GUI design, we included a vertical side menu, an improved single-page layout, and card sections, as can be seen in Figure 4.9. In this phase, we also implemented the Querying tool which allowed the user to retrieve information from the database, based on specific filters. The addition of the querying tool required a reorganization of the GUI.

First, the user navigates to the main page where there is a brief introduction to the tools and their usage. From there the user could navigate to the Annotation page or the Querying page via the 'Annotate' and 'Search' buttons, respectively. While the tabs navigation was a convenient pattern for smooth navigation, there was room for improvement. And so, instead of a horizontal menu positioned at the top of the screen, we implemented a vertical side menu using the Stepper component from the MaterialUI library. This component clearly showed the sections, and by clicking on a specific section name it immediately repositioned the user to the desired section. This way, the user could navigate from the end of the annotation form to its beginning instantaneously. Whereas the vertical side menu accounted for the navigation aspect, the one-page design left the content unstructured. To

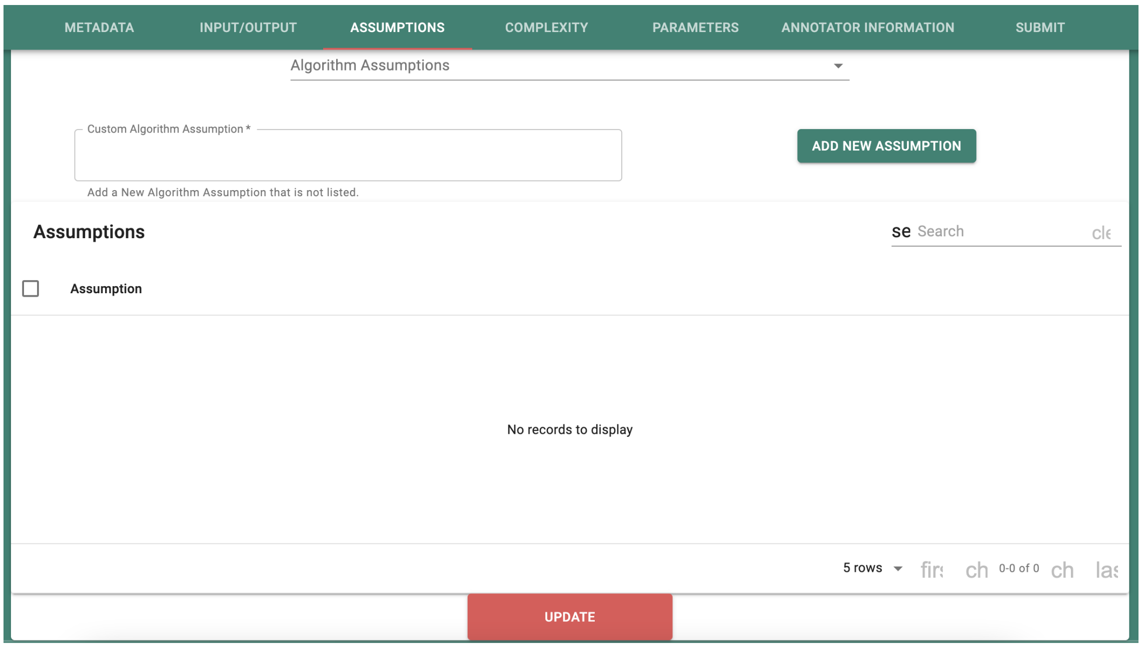


Figure 4.8: A snippet of the second version of the GUI for the annotation tool.

retain the semantic segmentation of the content, we made use of the Card component.

Each section was implemented as a functional component, where the corresponding variables and methods were defined. The sections were implemented as children functional components of the Annotation page. The connection between the Annotation page and the section components was made through multiple useState hooks which were passed as parameters to the child components, where the state was updated. In this way, the design was modular in the sense that each functional component served a single function — to store and present data related to a given aspect of an algorithm. The control was delegated to the parent component which had the role to pass the input data to its children components. The communication from the front-end to the back-end is done using the Axios HTTP client.

In this version, we also added the support for ensemble algorithms, which typically make use of single generalization algorithms in specific ways. In the UI, this is handled by a selection field that lets the user specify if the algorithm is a single generalization algorithm or an ensemble algorithm. We further distinguish between four types of ensemble algorithms, namely bagging, boosting, voting, and stacking ensembles. Based on the user selection, for each ensemble type, a corresponding view is rendered that contains all the necessary input fields.

Bi-directional communication between the user and the system allows for a richer user experience and utilization of the services. To address this, a Querying GUI (shown in Figure 4.10) was designed and implemented in the application. In the interface, the user can specify three filters (described in Section 4.2.5), which constrain the algorithm annotations that are retrieved. These filters were selected based on discussions with domain experts, where it was concluded that from a user perspective these filters are intuitive and useful. Sorted in a decreasing level of generality, the filters are designed with flexibility in mind, allowing the user to specify one, multiple, or none for a given query. For example, if a user only wishes to constrain the task that the algorithm addresses, one can set that filter, leaving the rest empty. If one wishes to retrieve all the algorithms in the database, it is also possible to leave all the filters empty. At the moment, all the filters support only

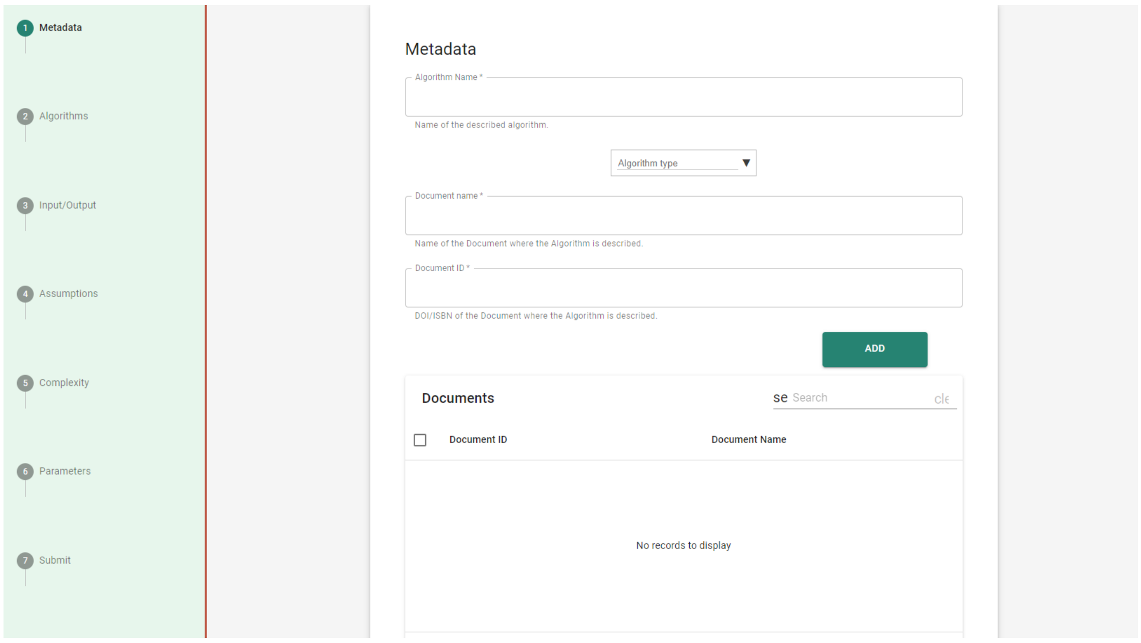
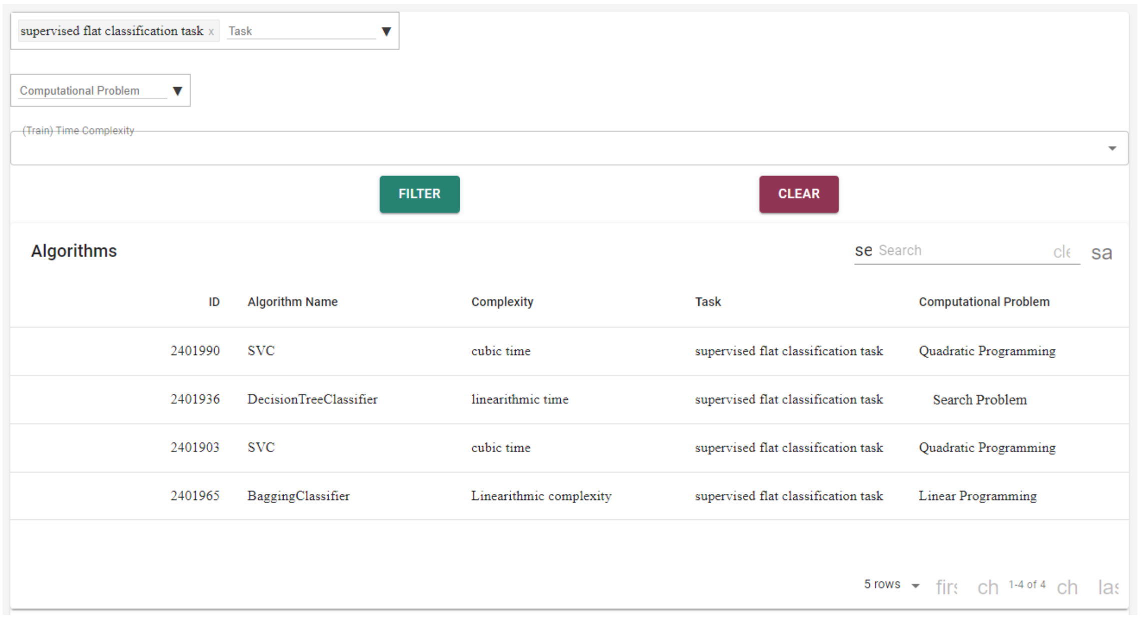


Figure 4.9: The final GUI for the annotation tool.

single-selection constraints, which means that the user cannot retrieve algorithms that address two or more different tasks in a single query. After specifying the filters, using identical UI elements used in the annotation tool, the user submits the query by clicking on the Filter button. This sets the values for each of the filters using the React useState hook and triggers a request to the back-end via Axios. After the request is handled in the back-end and the requested data is retrieved, the state of the results table is updated using the useState hook and the requested data is presented to the user. We also implemented a Clear Data button which resets the state of the results table, enabling the user to execute a different query without having to remove the results manually.

## 4.4 Population of the Graph Database

The most effective way to assess the quality of the developed resources is to see how well they achieve their purposes. This can be assessed through the process of annotating an ML/DM algorithm using the web application. On one hand, the ontology's capacity to encapsulate relevant information will be assessed, whereas its complexity will be measured by looking at the amount of effort required to produce one complete annotation. On the other hand, the quality of the annotation resource will be assessed by running the annotation process through the developed web application. Ultimately, this would result in not only the quality verification of the developed resources but also a populated graph database which would in turn represent a meaningful contribution to this work. A populated graph database would mean that the querying component of the web application could be used immediately upon deployment so that users can retrieve information even before annotating some ML/DM algorithm. Additionally, considering the complexity of the ML/DM domain, it would be a welcome guide to how the annotation process was conceptualized by the developers.



Algorithms

| ID | Algorithm Name | Complexity | Task | Computational Problem |
|---|---|---|---|---|
| 2401990 | SVC | cubic time | supervised flat classification task | Quadratic Programming |
| 2401936 | DecisionTreeClassifier | linearithmic time | supervised flat classification task | Search Problem |
| 2401903 | SVC | cubic time | supervised flat classification task | Quadratic Programming |
| 2401965 | BaggingClassifier | Linearithmic complexity | supervised flat classification task | Linear Programming |

Figure 4.10: The Querying GUI where the task is specified; the retrieved results are shown in the table.

### 4.4.1 The Scikit-learn library

Scikit-learn (sklearn) is a free software machine learning library for Python. It features various classification, regression and clustering algorithms including support-vector machines, random forests, k-means, etc (Pedregosa et al., 2011). It is one of the most popular machine learning libraries on GitHub due to its utility and robustness. Its documentation is comprehensible (although not comprehensive), and it features diverse information related to the ML/DM algorithm. It provides documentation in the form of a user guide, where a short description of the algorithm is presented, along with optionally the optimization problem it solves, useful tips about the parameters, and practical examples of how to use the algorithm. Besides having more user-friendly documentation, more technical details on the implementation of the algorithm are also supplied in the API section. Here, one can inspect each of the parameters (information that is passed to the algorithm), and the attributes (parameters that represent the state of the algorithm) related to the algorithm. Unlike the user guide documentation, here we have a more structured and comprehensive approach, where for each parameter the name, accepted datatypes, optional values (or a range of values), default value, as well as a brief description of the parameter function are listed. Additional information related to the methods that can be called on the algorithm object is supplied (i.e., fit, predict, score, etc.), as well as usage notes and examples.

To populate the database we needed a fairly large source where ML/DM algorithm information is publicly available and accessible. For this purpose we found sklearn to be an adequate source. One of the drawbacks of sklearn is that the documentation is not comprehensive, and structured enough, i.e., there is no format for specifying ML/DM algorithm information. For example, it is often the case that the time complexity for one algorithm is specified, while in another case it is not mentioned at all. In such cases, we turned to related sources to find the required information. If the documentation referred to the paper where the algorithm was first presented or some related paper, this was the first place to look for the required information. In a sense, the sklearn documentation served as a higher level of abstraction for ML/DM algorithm information, in comparison to the original papers where the algorithms are described. It would require an extensive amount of effort to go through each original paper, as this is not a straightforward task. An added benefit of using sklearn was that the documentation was tied to a specific implementation, allowing us to extract additional information related not only to the specification of an algorithm but also to its implementation, including information related to its parameters, their datatypes, etc.

### 4.4.2 Setting up the population engine

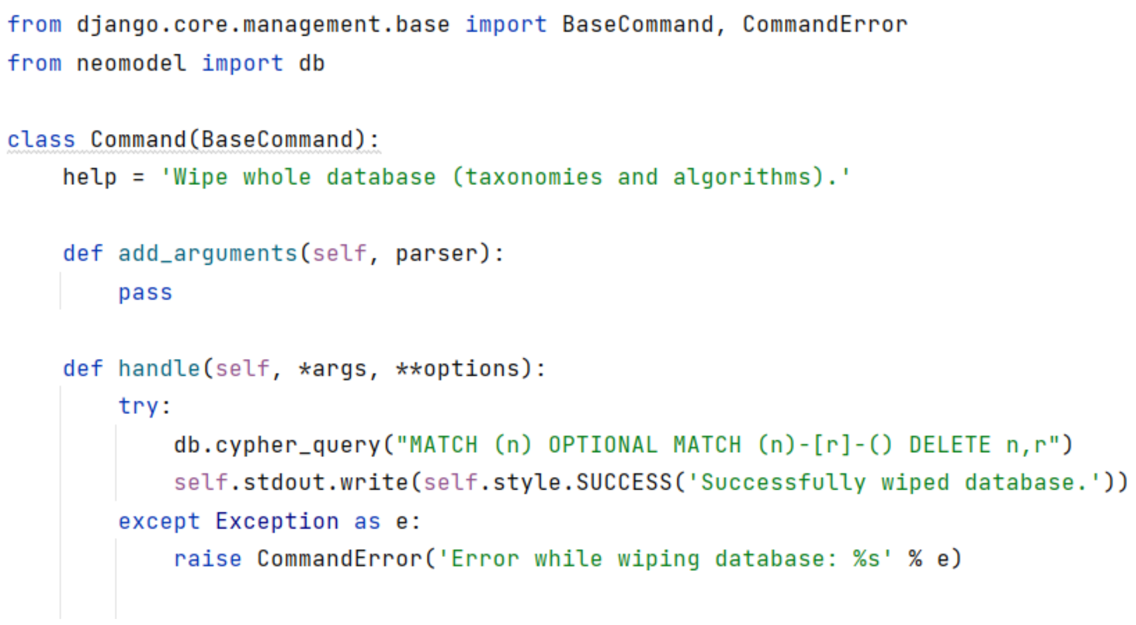
The best way to test the database conceptual model's ability to describe our domain of interest is to annotate a representative data sample. First, we need to populate the database with ontology instances which are offered as selection options in the annotation tool. For example, our schema defines common algorithm assumptions, optimization problems, and complexity instances. We can populate the database either via the Neo4j Desktop Application, by manually typing Cypher queries and executing them through Neomodel or by using Neomodel database commands. While using the Neo4j GUI is the easiest and most interactive way, it does not scale well, as we must manually populate the database every time we make changes in the schema, or to the instances. Since our schema is experimental and iteratively modified, it is essential to populate the database in an automated and efficient way. Using Cypher queries would be more efficient, however their execution would have to be done via the Neomodel library so that we can execute them on demand. A nifty way to handle these operations is by writing custom Django management commands. Namely, our actions can be registered with the manage.py script, which links our settings file where we define, among other things, our database settings. In this way, we can create several management commands which would perform our CRUD operations in the Neo4j database when the application is starting. Initially, we created three separate management commands:

- populate_db_taxonomies: Populates the database with the necessary taxonomies (optimization problems, complexity specifications, and algorithm assumptions).
- populate_db_algorithms: Populates the database with algorithm annotations for the selected representative algorithms (i.e., Decision Tree Classifier).
- delete_db: Deletes the entire populated database.

The first two commands allow us to create the schema and the annotations separately. So, for example, when we want to add an algorithm annotation, we can just run the second command, while the schema remains intact. The third script enables us to delete the whole database with a single Cypher query, which is very useful when we make changes to the schema since changes in the schema mean that the annotations need to be adjusted appropriately. The management script for deleting the database can be seen in Figure 4.11.

### 4.4.3 Annotation of specific ML/DM algorithms

We annotated both single generalization and ensemble algorithms which address different tasks and operate differently internally. To ensure consistency we will refer to them using their names as presented in the sklearn documentation, while the name in the brackets represents their original naming. These algorithms are: LinearRegression (Ordinary Least Squares), SVC (Support Vector Machine) (Cortes & Vapnik, 1995), KNeighborsClassifier (K Nearest Neighbors) (Keller et al., 1985), DecisionTreeClassifier (CART) (Breiman et al.,



```python
from django.core.management.base import BaseCommand, CommandError
from neomodel import db

class Command(BaseCommand):
    help = 'Wipe whole database (taxonomies and algorithms).'

    def add_arguments(self, parser):
        pass

    def handle(self, *args, **options):
        try:
            db.cypher_query("MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r")
            self.stdout.write(self.style.SUCCESS('Successfully wiped database.'))
        except Exception as e:
            raise CommandError('Error while wiping database: %s' % e)
```

Figure 4.11: The Django management script for deleting the graph database.

2017), GaussianNB (Gaussian Naive Bayes) (H. Zhang, 2004), BaggingClassifier (Bagging) (Breiman, 1996), GradientBoostingRegressor (Gradient Tree Boosting) (Friedman, 2001), VotingClassifier, StackingClassifier and MLPClassifier (Multi-layer Perceptron) (Hinton, 2009). The annotated algorithms can be seen in Table 4.2, where almost full annotation information is presented. For presentational purposes, the problem text description is not included, as it can be lengthy, and some of the assumptions are not included, as well as the types and the datatypes of the parameters.

### 4.4.4 Lessons learned

First, it is obvious to say that the annotation process was extremely difficult due to several reasons: the complexity of the ML/DM domain, the lack of a well-structured information source, and the potential lack of expressiveness in the annotation schema. The first reason cannot be argued since the field of ML/DM is relatively young in comparison with other sciences and is still evolving at a fast pace. Next, as we mentioned that sklearn is not comprehensive, many times we had to turn to different information sources to gather some information. Sometimes not even the authors specify this information in a structured way and so there is the potential of having information related to different implementations of the same algorithm.

Notable examples of entities that were hard to annotate were the time complexity entity, as well as the computational problem entity. Additionally, the computational problem for some algorithms could not be easily described. While it is clear that SVC and MLPClassifier are designed to solve an optimization problem, for KNeighborsClassifier and GaussianNB the choice was not straightforward. Perhaps, practitioners with more expertise would be able to provide complete annotations, or the annotation schema needs to be adjusted in the following iteration to address these situations.

Another notable finding is that since not all algorithms produce a model as output, they do not have a 'training time complexity, but rather compute the predictions directly, and so the time complexity refers to the prediction (or test) time complexity. An example of this is the vanilla KNeighborsClassifier algorithm. In ensemble algorithms, much of the information is inherited from the base estimator, i.e., the single generalization algorithm which is used in the ensemble. Information like the dataset type, the task type, the assumptions, and the computational problem are the same for the DecisionTreeClassifier and the BaggingClassifier. Although there can be more assumptions for the ensemble algorithm (when compared to the base estimator) and the computational problem can be different for the remaining ensemble algorithms, the dataset type and the task type rarely change, since they are defined solely based on the input and output type.

Finally, the stacking and voting ensemble algorithms were unique in the sense that they are not described in some document, i.e., a paper, because they are more conceptual — operating by combining the outputs of different algorithms to produce a single, less biased output. The information related to the documents, optimization problems, and time complexities can be represented with a dictionary where each of the constituent algorithm information will be added. However, this can also be inferred since the information is already described for the single generalization algorithms.

Table 4.2: Complete annotations for a selection of ML/DM algorithms.

| **Algorithm:** | **LinearRegression** |
|---|---|
| *Document:* | Christopher M. Bishop: Pattern Recognition and Machine Learning, Chapter 4.3.4, id: 0387310738 |
| *Dataset:* | regression dataset |
| *Task:* | supervised regression task |
| *Computational* | Optimization Problem (Quadratic Programming) |
| *Problem:* | $\min_w \|\|Xw - y\|\|_2^2$ |
| *Generalization:* | regression model |
| | Independence of observations |
| *Assumption:* | No hidden or missing variables |
| | Homoscedasticity |
| *Time complexity:* | quadratic time |
| | $O(n_{\text{samples}} n_{\text{features}}^2)$ |
| | fit_intercept |
| | normalize |
| *Parameters:* | copy_X |
| | n_jobs |
| | positive |
| *Estimators:* | none |

| **Algorithm:** | **SVC** |
|---|---|
| *Document:* | Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods (1999) id: / |
| *Dataset:* | flat classification dataset |
| *Task:* | supervised flat classification task |
| | Optimization Problem (Quadratic Programming) |
| *Computational* | $\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$ |
| *Problem:* | subject to $y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i$, |
| | $\zeta_i \geq 0, i = 1, ..., n$ |

Continued on next page

Continued from previous page

| | |
|---|---|
| *Generalization:* | classification model |
| *Assumption:* | none |
| *Time complexity:* | cubic time<br>$O(n_{features} \times n^3_{samples})$ |
| *Parameters:* | C<br>kernel<br>degree<br>gamma<br>coef0<br>shrinking<br>probability<br>tol<br>cache_size<br>class_weight<br>verbose<br>max_iter<br>decision_function_shape<br>break_ties<br>random_state |
| *Estimators:* | none |
| **Algorithm:** | **KNeighborsClassifier** |
| *Document:* | Discriminatory Analysis — Nonparametric Discrimination: Small Sample Performance, id: ADA800391 |
| *Dataset:* | flat classification dataset |
| *Task:* | supervised flat classification task |
| *Computational Problem:* | Optimization Problem (Integer Programming)<br>$y = \text{argmax}_{c_j} \sum_{x_i \in N_k(x)} I(y_i = c_j)$,<br>$i = 1, 2, \cdots, n; j = 1, 2, \cdots, m$ |
| *Generalization:* | classification model |
| *Assumption:* | Similar things exist in close proximity |
| *Time complexity:* | constant time<br>$O(1)$ |
| *Parameters:* | C n_neighbors<br>algorithm<br>weights<br>leaf_size<br>p<br>metric<br>metric_params<br>n_jobs |
| *Estimators:* | none |
| **Algorithm:** | **DecisionTreeClassifier** |
| *Document:* | L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984, id: 9781315139470 |
| *Dataset:* | flat classification dataset |
| *Task:* | supervised flat classification task |

Continued on next page

Continued from previous page

| | |
|---|---|
| *Computational Problem:* | Search Problem<br>$\theta^* = \operatorname{argmin}_\theta G(Q_m, \theta)$<br>$G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta)) + \frac{n_m^{right}}{n_m} H(Q_m^{right}(\theta))$ |
| *Generalization:* | classification model |
| *Assumption:* | none |
| *Time complexity:* | linearithmic time<br>$O(n_{samples} n_{features} \log(n_{samples}))$ |
| *Parameters:* | criterion<br>splitter<br>max_depth<br>min_samples_split<br>min_samples_leaf<br>min_weight_fraction_leaf<br>max_features<br>random_state<br>min_impurity_decrease<br>class_weight<br>ccp_alpha |
| *Estimators:* | none |
| **Algorithm:** | **GaussianNB** |
| *Document:* | Updating Formulae and a Pairwise Algorithm for Computing Sample Variances, id: STAN-CS-79-773 |
| *Dataset:* | multi-class classification dataset |
| *Task:* | supervised multi-class classification task |
| *Computational Problem:* | Optimization Problem (Maximum Likelihood Estimation)<br>$\hat{y} = \operatorname{argmax}_y P(y) \prod_{i=1}^{n} P(x_i \| y)$ |
| *Generalization:* | probability distribution specification |
| *Assumption:* | Conditional independence between every pair of features given the value of the class variable<br>The likelihood of the features is assumed to be Gaussian |
| *Time complexity:* | linear time<br>$O(n_{samples} n_{features})$ |
| *Parameters:* | priors<br>var_smoothing |
| *Estimators:* | none |
| **Algorithm:** | **BaggingClassifier** |
| *Document:* | Bagging predictors, Machine Learning, 24(2), 123-140, 1996, id: https://doi.org/10.1007/BF00058655 |
| *Dataset:* | Same as base estimator's |
| *Task:* | Same as base estimator's |
| *Computational Problem:* | Same as base estimator's |
| *Generalization:* | predictive models ensemble specification |
| *Assumption:* | none |
| *Time complexity:* | $O(n\_estimators * complexity(base\_estimator))$ |

Continued on next page

Continued from previous page

| | |
|---|---|
| | n_estimators |
| | max_samples |
| | max_features |
| | bootstrap |
| | bootstrap_features |
| *Parameters:* | oob_score |
| | warm_start |
| | n_jobs |
| | random_state |
| | verbose |
| *Estimators:* | DecisionTreeClassifier |
| **Algorithm:** | **GradientBoostingRegressor** |
| | Friedman, J.H. (2001). Greedy function approximation: |
| *Document:* | A gradient boosting machine. Annals of Statistics, |
| | 29, 1189-1232, id: 10.1214/aos/1013203451 |
| *Dataset:* | regression dataset |
| *Task:* | supervised regression task |
| | |
| *Computational* | Search Problem |
| *Problem:* | $h_m = \arg\min_h L_m = \arg\min_h \sum_{i=1}^{n} l(y_i, F_{m-1}(x_i) + h(x_i))$ |
| *Generalization:* | predictive models ensemble specification |
| *Assumption:* | none |
| *Time complexity:* | $O(n\_trees * n\_samples * log(n\_samples * n\_features))$ |
| | loss |
| | learning_rate |
| | n_estimators |
| | subsample |
| | criterion |
| | min_samples_split |
| | min_samples_leaf |
| | min_weight_fraction_leaf |
| | max_depth |
| | min_impurity_decrease |
| *Parameters:* | init |
| | random_state |
| | max_features |
| | alpha |
| | verbose |
| | max_leaf_nodes |
| | warm_start |
| | validation_fraction |
| | n_iter_no_change |
| | tol |
| | ccp_alpha |
| *Estimators:* | DecisionTreeRegressor |
| **Algorithm:** | **VotingClassifier** |
| *Document:* | / |
| *Dataset:* | flat classification dataset |

Continued on next page

Continued from previous page

| | |
|---|---|
| *Task:* | supervised flat classification task |
| *Computational Problem:* | / |
| *Generalization:* | ensemble specification |
| *Assumption:* | / |
| *Time complexity:* | / |
| *Parameters:* | voting<br>weights<br>n_jobs<br>flatten_transform<br>verbose |
| *Estimators:* | DecisionTreeClassifier<br>KNeighborsClassifier<br>SVC |
| **Algorithm:** | **StackingClassifier** |
| *Document:* | / |
| *Dataset:* | binary classification dataset |
| *Task:* | supervised binary classification task |
| *Computational Problem:* | / |
| *Generalization:* | ensemble specification |
| *Assumption:* | / |
| *Time complexity:* | / |
| *Parameters:* | stack_method<br>n_jobs<br>passthrough<br>verbose |
| *Estimators:* | BaggingClassifier<br>KNeighborsClassifier<br>GaussianNB<br>final_estimator: LogisticRegression |
| **Algorithm:** | **MLPClassifier** |
| *Document:* | Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors, id: https://doi.org/10.1038/323533a0 |
| *Dataset:* | flat classification dataset |
| *Task:* | supervised flat classification task |
| *Computational Problem:* | Optimization Problem<br>argmin $Loss(\hat{y}, y, W)$<br>$Loss(\hat{y}, y, W) = -\frac{1}{n}\sum_{i=0}^{n}(y_i \ln \hat{y_i} + (1-y_i)\ln(1-\hat{y_i})) + \frac{\alpha}{2n}\|\|W\|\|_2^2$<br>$W^{i+1} = W^i - \epsilon \nabla Loss_W^i$ |
| *Generalization:* | classification model |
| *Assumption:* | / |
| *Time complexity:* | $O(n\_epochs * n\_samples * n\_features * n\_neurons)$ |

Continued on next page

Continued from previous page

| | |
|---|---|
| | hidden_layer_sizes |
| | activation |
| | solver |
| | alpha |
| | batch_size |
| | learning_rate |
| | learning_rate_init |
| | power_t |
| | max_iter |
| | shuffle |
| | random_state |
| *Parameters:* | tol |
| | verbose |
| | warm_start |
| | momentum |
| | nesterovs_momentum |
| | early_stopping |
| | validation_fraction |
| | beta_1 |
| | beta_2 |
| | epsilon |
| | n_iter_no_change |
| | max_fun |
| *Estimators:* | none |

# Chapter 5

# Use Cases

In this chapter, we present two use cases that exemplify the functionalities of the developed web-based system. Upon launching the web application, the user is presented with the main page where a short description of the functionalities of the application is provided. The user can then either navigate to the annotation tool — to annotate an ML/DM algorithm, or the querying tool — to query the populated repository, i.e., to obtain information related to the ML/DM algorithms that are stored in the repository.

## 5.1 Annotation Use Case

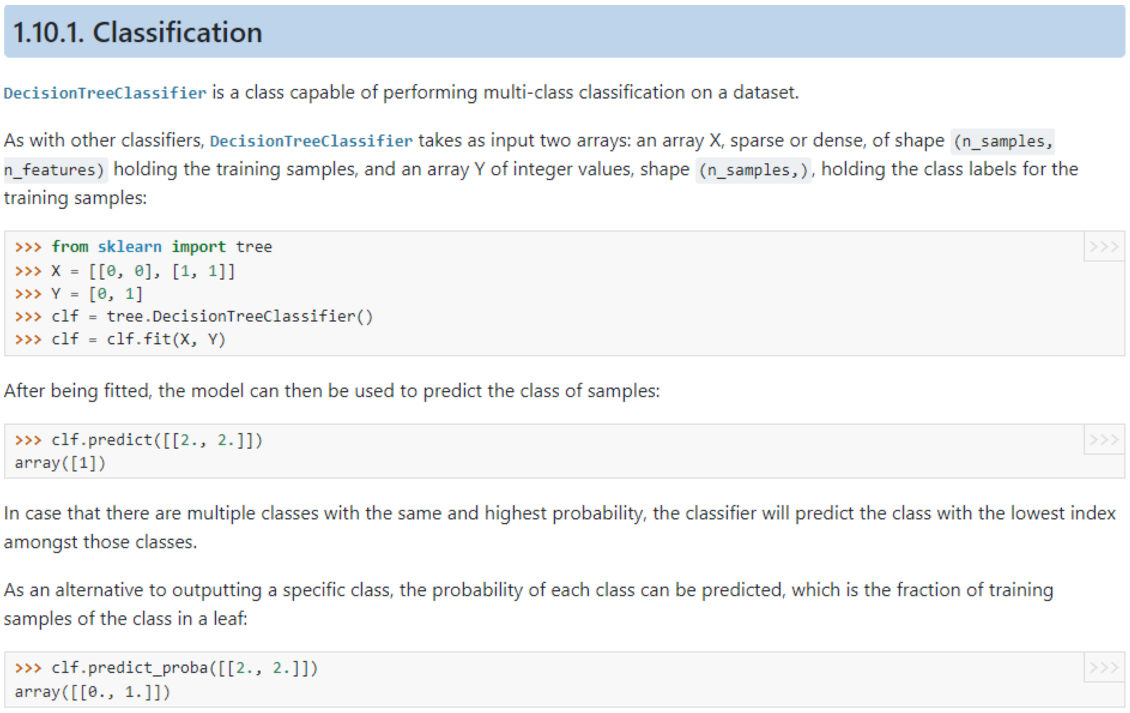
In this section, we present the procedure for annotating ML/DM algorithms. To illustrate the capabilities of the annotation tool we annotate two ML/DM algorithms: the DecisionTreeClassifier (single generalization algorithm), and the BaggingClassifier (ensemble algorithm). To do so, we refer to the sklearn documentation, described in Section 4.4.1, as we focus on annotating algorithms present in that toolkit. The information is gathered from the sklearn documentation, which covers specification information, and the API section, which covers implementation information. The documentation snippets of the sources are presented in Figure 5.1.

### 5.1.1 Annotation of a single generalization algorithm: DecisionTreeClassifier

The annotation interface consists of several sections which cover different aspects of information related to the algorithm. Most of the sections apply to both single generalization and ensemble algorithms. Hence, in this section, by following an example algorithm — the DecisionTreeClassifier, we will cover information that is mutual for both algorithm types, while in the next section, we will describe the information that is exclusively used for the annotation of ensemble algorithms.

The first section in the annotation tool is the Metadata section, seen in Figure 5.2a. Here, the ML/DM algorithm name and type are input, as well as the documents in which the algorithm is described. Each document has a name and an identifier, and the user can input multiple documents, as well as remove them in the case of typing mistakes.

The next section is the Input/Output section (see Figure 5.2b). Here, the user can input the type of task, the type of dataset that can be used as input, and the type of output the algorithm produces through the generalization specification type and generalization language fields. Next, the user can specify the assumptions of the algorithm in the Assumptions section, presented in Figure 5.2c. Here, the user can select from assumptions, added as instances in the ontology, or specify custom assumptions using a text field.



1.10.1. Classification

DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset.

As with other classifiers, DecisionTreeClassifier takes as input two arrays: an array X, sparse or dense, of shape (n_samples, n_features) holding the training samples, and an array Y of integer values, shape (n_samples,), holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

After being fitted, the model can then be used to predict the class of samples:

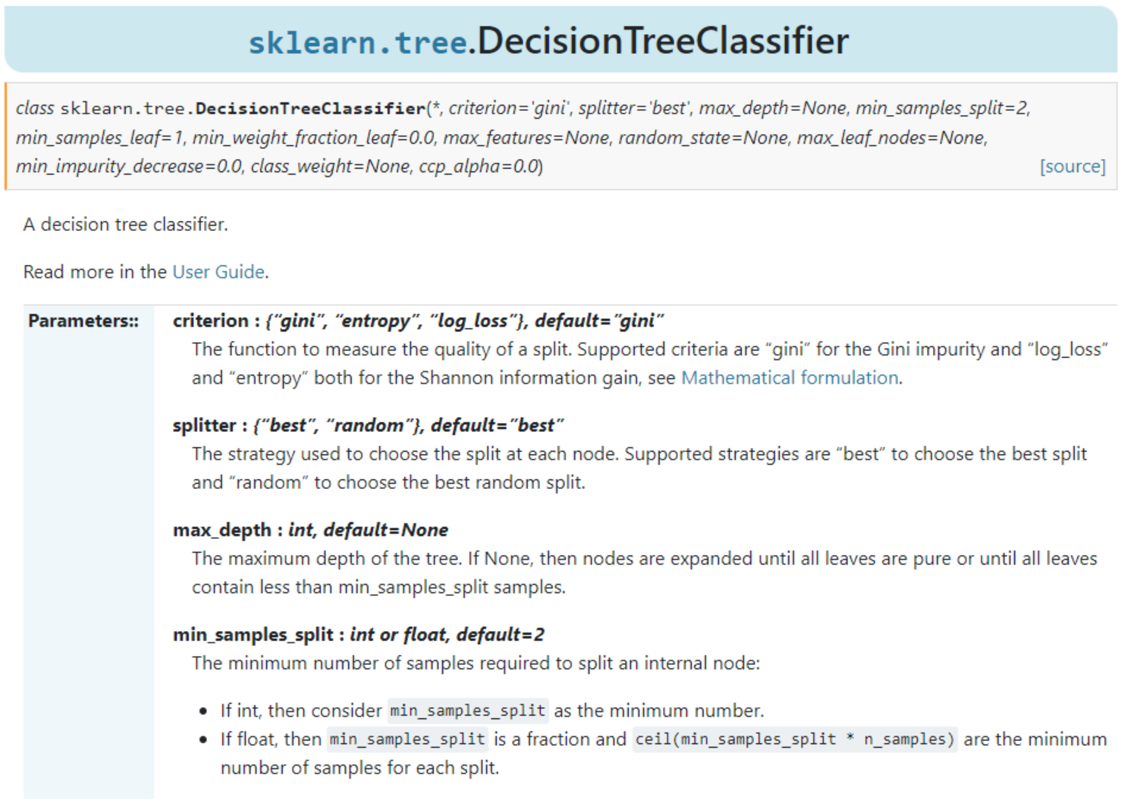
```
>>> clf.predict([[2., 2.]])
array([1])
```

In case that there are multiple classes with the same and highest probability, the classifier will predict the class with the lowest index amongst those classes.

As an alternative to outputting a specific class, the probability of each class can be predicted, which is the fraction of training samples of the class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

(a) Sklearn user guide documentation



sklearn.tree.DecisionTreeClassifier

*class* sklearn.tree.**DecisionTreeClassifier**(**, criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None, ccp_alpha=0.0*) [source]

A decision tree classifier.

Read more in the User Guide.

**Parameters::**

**criterion : *{"gini", "entropy", "log_loss"}, default="gini"***
The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "log_loss" and "entropy" both for the Shannon information gain, see Mathematical formulation.

**splitter : *{"best", "random"}, default="best"***
The strategy used to choose the split at each node. Supported strategies are "best" to choose the best split and "random" to choose the best random split.

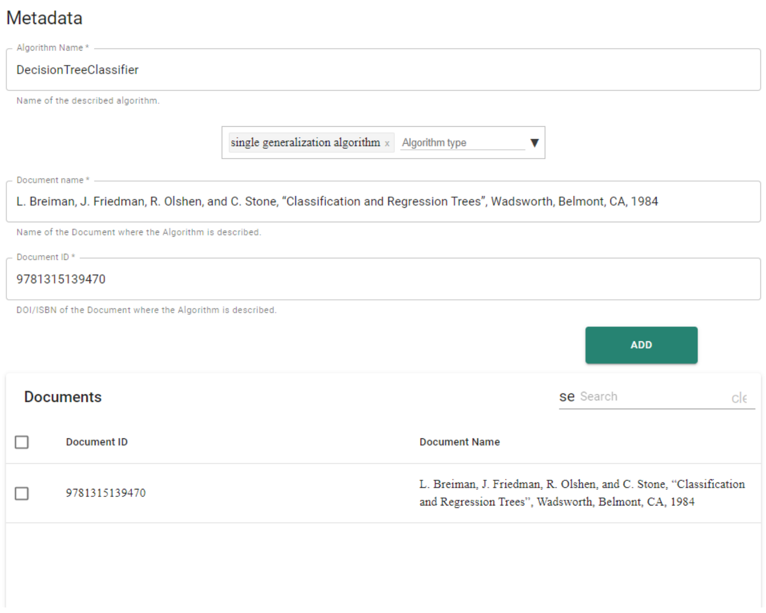
**max_depth : *int, default=None***
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

**min_samples_split : *int or float, default=2***
The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

(b) Sklearn API guide documentation

Figure 5.1: Snippets from the sklearn documentation for the DecisionTreeClassifier algorithm.


Metadata

Algorithm Name *
DecisionTreeClassifier
Name of the described algorithm.

single generalization algorithm x Algorithm type

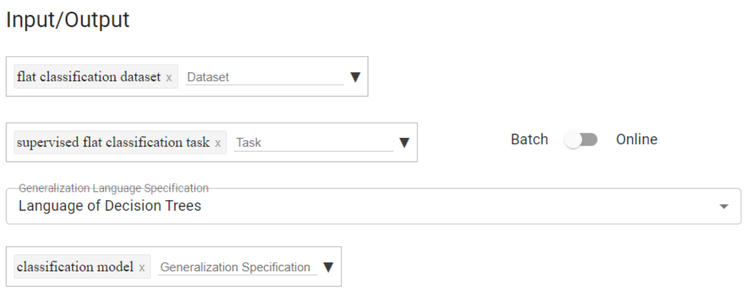
Document name *
L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984
Name of the Document where the Algorithm is described.

Document ID *
9781315139470
DOI/ISBN of the Document where the Algorithm is described.

Documents

| | Document ID | Document Name |
|---|---|---|
| ☐ | 9781315139470 | L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees", Wadsworth, Belmont, CA, 1984 |

(a) Metadata section


Input/Output

flat classification dataset x Dataset

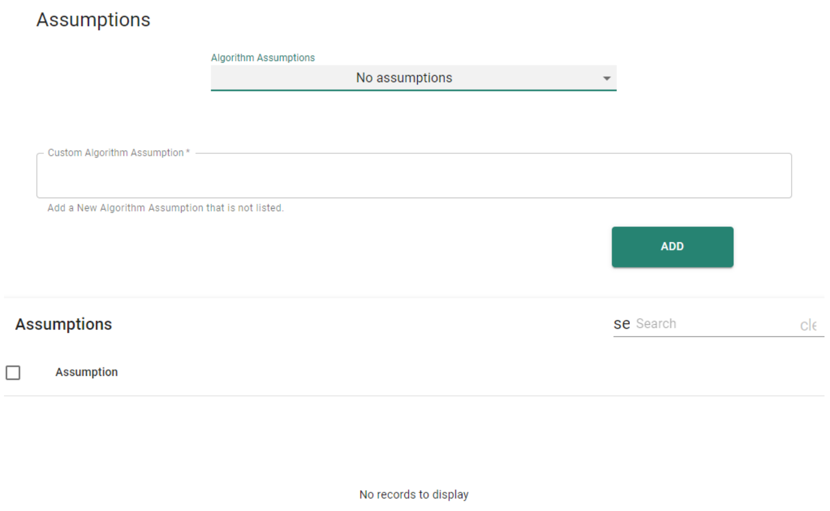
supervised flat classification task x Task

Batch Online

Generalization Language Specification
Language of Decision Trees

classification model x Generalization Specification

(b) Input/Output section


Assumptions

Algorithm Assumptions
No assumptions

Custom Algorithm Assumption *
Add a New Algorithm Assumption that is not listed.

Assumptions

| | Assumption |
|---|---|

No records to display

(c) Assumptions section

Figure 5.2: The Metadata, Input/Output, and Assumptions sections of the user interface.

Following this, we have the Complexity section (see Figure 5.3a), where information related to the type of computational problem the ML/DM algorithm solves and its computational complexity is to be input. In this section, the type of the computational problem is selected, and additional information about the problem using both the textual description and a mathematical notation (latex notation). Additionally, the (train) time complexity of the algorithm can be specified by selecting from the time complexities described earlier. The user can also represent the time complexity in more detail with mathematical notation (latex notation), to specify the parameters that determine the computational time complexity. Afterwards comes the Parameters section shown in Figure 5.3b. Using the already familiar table functionality, the user can input the name and datatype of each parameter that is a part of the annotated ML/DM algorithm. Finally, the annotation is concluded in the Annotator Information section (Figure 5.3c) by adding personal information related to the annotator, such as the name, affiliation, and email.

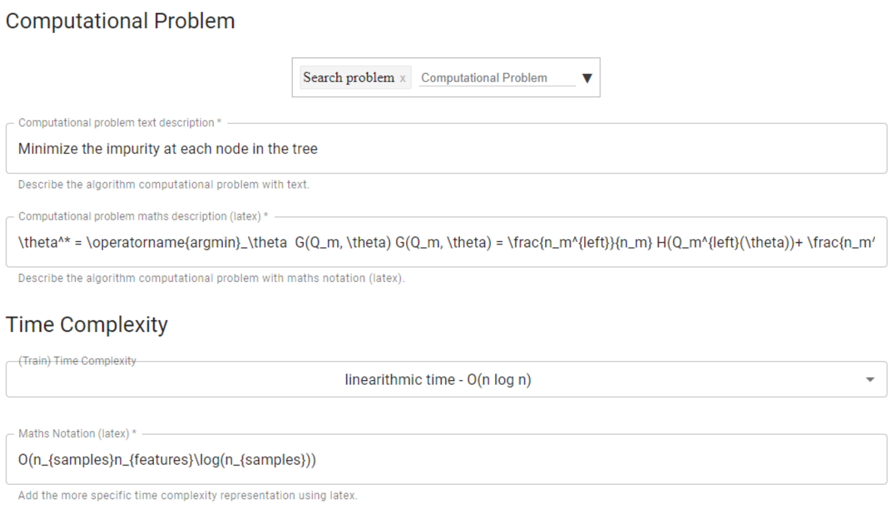
### 5.1.2 Annotation of an ensemble algorithm: BaggingClassifier

As we mentioned in the previous section, the annotation tool follows the same layout for both single generalization and ensemble algorithms, barring one difference. In fact, after inputting the name of the ML/DM algorithm in the annotation form, the user can specify if the algorithm is a single generalization or an ensemble algorithm. If the annotated algorithm is an ensemble algorithm, the user can further specify the type. Namely, the user is presented with four options based on our differentiation of ensemble algorithms into four groups: Bagging, Boosting, Voting, and Stacking (described in Chapter 3).

If the user is unsure of the specific ensemble type, they can just select the ensemble algorithm type and no additional section will be rendered. However, if the type of the ensemble algorithm is specified, an additional section corresponding to each of the types is rendered (see Figure 5.4). The selection options that include ML/DM algorithms (such as the Base Estimator, the Estimators, and the Final Estimator fields) are fetched from the populated database. This allows for the connection between ML/DM algorithms (for which annotation data is already available and validated), and ensemble algorithms, leading to greater data inter-linking.

In the case of the annotation of a Bagging ensemble algorithm, additional information such as the sampling type and the base estimator can be input (see Figure 5.4a). If the user selects the Boosting type, they can input the base estimator of the ensemble algorithm (see Figure 5.4b). In the Voting type case (see Figure 5.4c, the user can select multiple ML/DM algorithms that are constituent parts of the Voting ensemble algorithm. Finally, if the Stacking type is chosen (Figure 5.4d), the user can select multiple ML/DM algorithms and the final algorithm that produces the final output.

Here, we present an end-to-end annotation scenario for an ensemble ML/DM algorithm, the BaggingClassifier. We must note that some of the information related to an ensemble algorithm, such as the dataset type, the task type, as well as potentially the optimization problem and the assumptions can be inherited from the base ML/DM algorithm. However, for the scope of this research, we did not manage to verify if this applies to all ensemble algorithms, and consequently, we decided this information should be specified explicitly by the user. The complete annotation for the BaggingClassifier is shown in Figures 5.5 and 5.6.



Computational Problem

Search problem x Computational Problem ▼

Computational problem text description *

Minimize the impurity at each node in the tree

Describe the algorithm computational problem with text.

Computational problem maths description (latex) *

\theta^* = \operatorname{argmin}_\theta  G(Q_m, \theta) G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\theta))+ \frac{n_m'

Describe the algorithm computational problem with maths notation (latex).

Time Complexity

(Train) Time Complexity

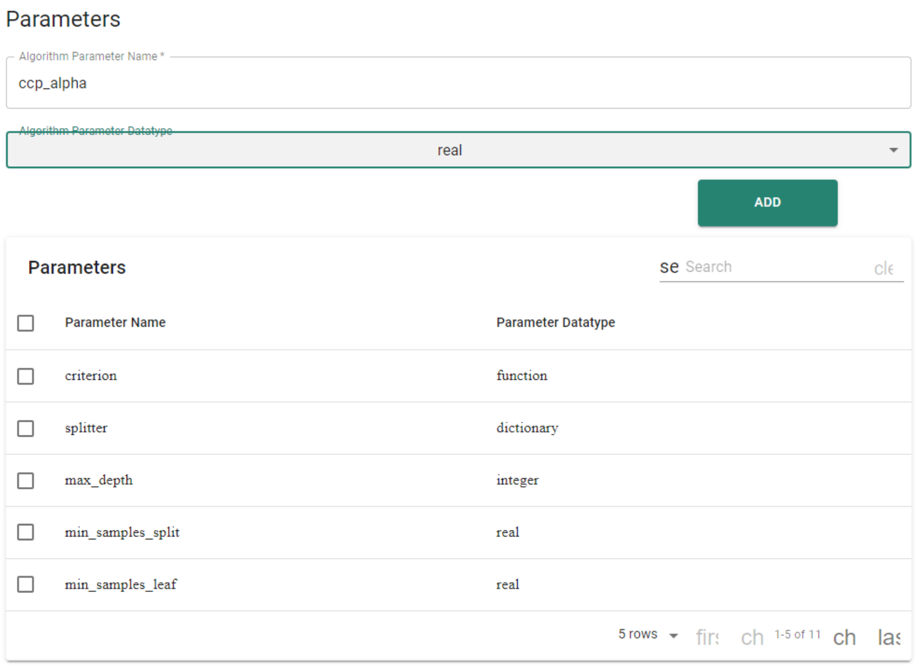
linearithmic time - O(n log n)

Maths Notation (latex) *

O(n_{samples}n_{features}\log(n_{samples}))

Add the more specific time complexity representation using latex.

(a) Complexity section



Parameters

Algorithm Parameter Name *

ccp_alpha

Algorithm Parameter Datatype

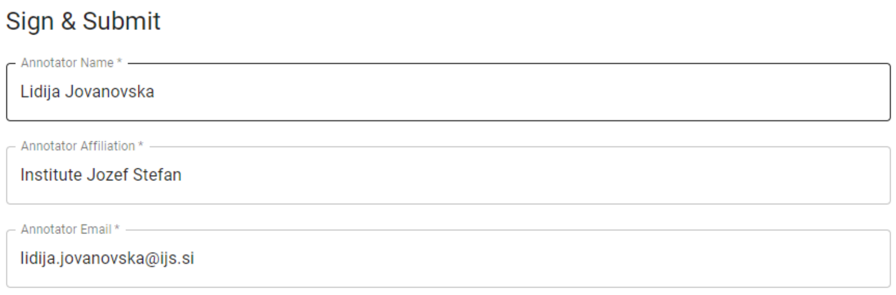
real

Parameters

se Search cle

| ☐ | Parameter Name | Parameter Datatype |
|---|---|---|
| ☐ | criterion | function |
| ☐ | splitter | dictionary |
| ☐ | max_depth | integer |
| ☐ | min_samples_split | real |
| ☐ | min_samples_leaf | real |

(b) Parameters section



Sign & Submit

Annotator Name *

Lidija Jovanovska

Annotator Affiliation *

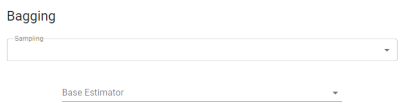
Institute Jozef Stefan

Annotator Email *

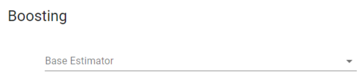
lidija.jovanovska@ijs.si

(c) Annotator information section

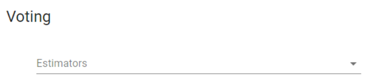
Figure 5.3: The Complexity, Parameters, and Annotator Information sections of the user interface.



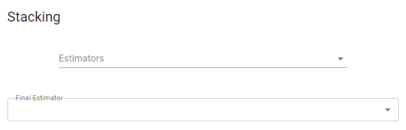
(a) Bagging section



(b) Boosting section

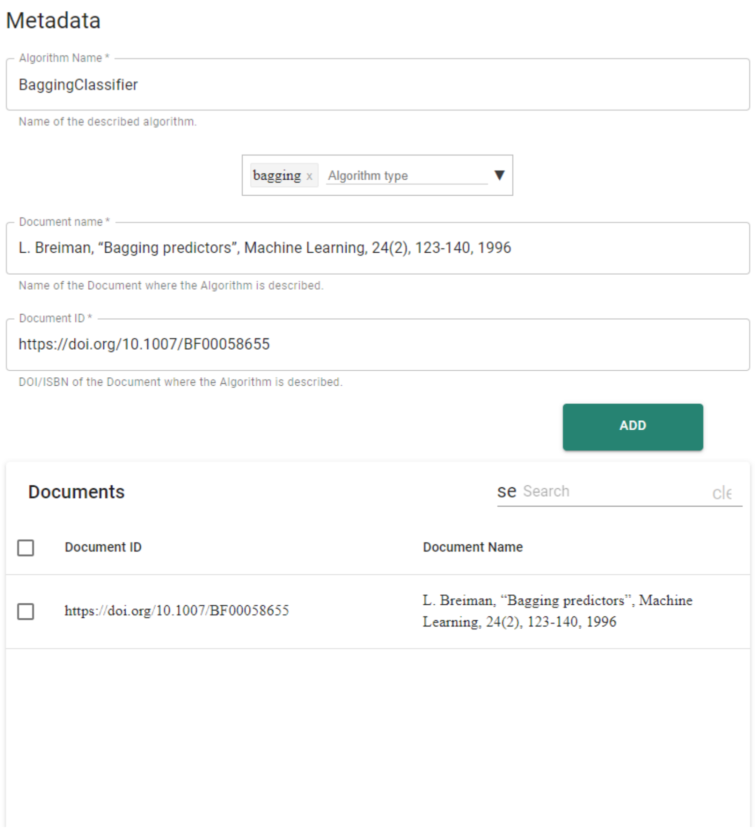


(c) Voting section



(d) Stacking section

Figure 5.4: The Bagging, Boosting, Voting, and Stacking sections of the user interface.


Metadata

Algorithm Name *
BaggingClassifier
Name of the described algorithm.

bagging x Algorithm type ▼

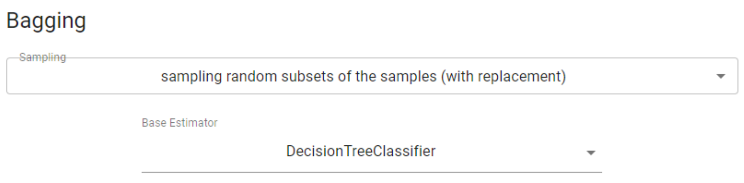
Document name *
L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996
Name of the Document where the Algorithm is described.

Document ID *
https://doi.org/10.1007/BF00058655
DOI/ISBN of the Document where the Algorithm is described.

Documents

| ☐ | Document ID | Document Name |
|---|---|---|
| ☐ | https://doi.org/10.1007/BF00058655 | L. Breiman, "Bagging predictors", Machine Learning, 24(2), 123-140, 1996 |

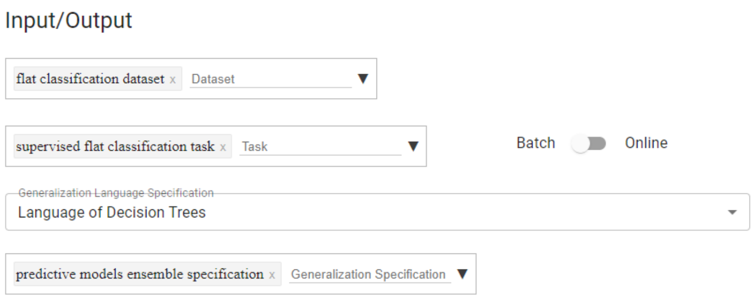
(a) Metadata section


Bagging

Sampling
sampling random subsets of the samples (with replacement)

Base Estimator
DecisionTreeClassifier

(b) Bagging section


Input/Output

flat classification dataset x Dataset ▼

supervised flat classification task x Task ▼

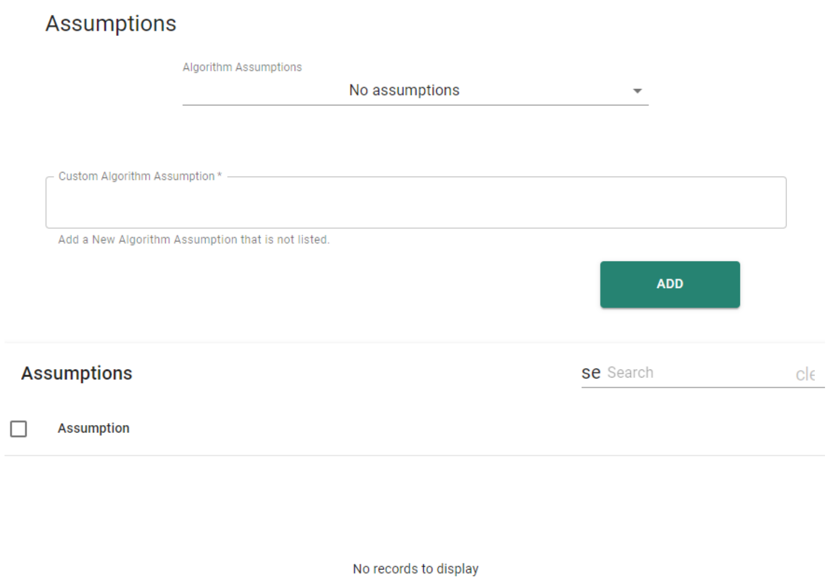
Batch Online

Generalization Language Specification
Language of Decision Trees

predictive models ensemble specification x Generalization Specification ▼

(c) Input/Output section

Figure 5.5: BaggingClassifier pt. 1/3: Complete end-to-end annotation of an ensemble ML/DM algorithm.



Assumptions

Algorithm Assumptions: No assumptions

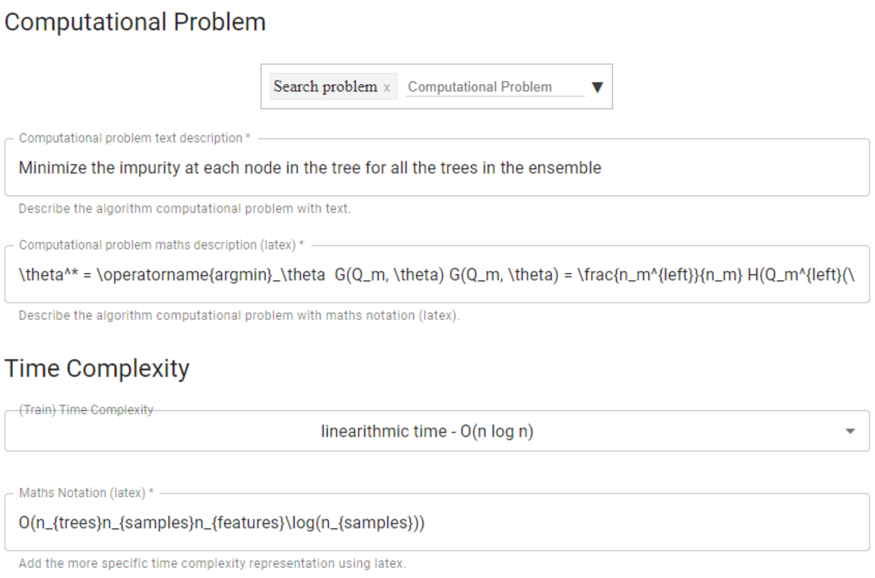
Custom Algorithm Assumption *

Add a New Algorithm Assumption that is not listed.

Assumptions

No records to display

(a) Assumptions section



Computational Problem

Computational problem text description *: Minimize the impurity at each node in the tree for all the trees in the ensemble

Describe the algorithm computational problem with text.

Computational problem maths description (latex) *: \theta^* = \operatorname{argmin}_\theta  G(Q_m, \theta) G(Q_m, \theta) = \frac{n_m^{left}}{n_m} H(Q_m^{left}(\

Describe the algorithm computational problem with maths notation (latex).

Time Complexity

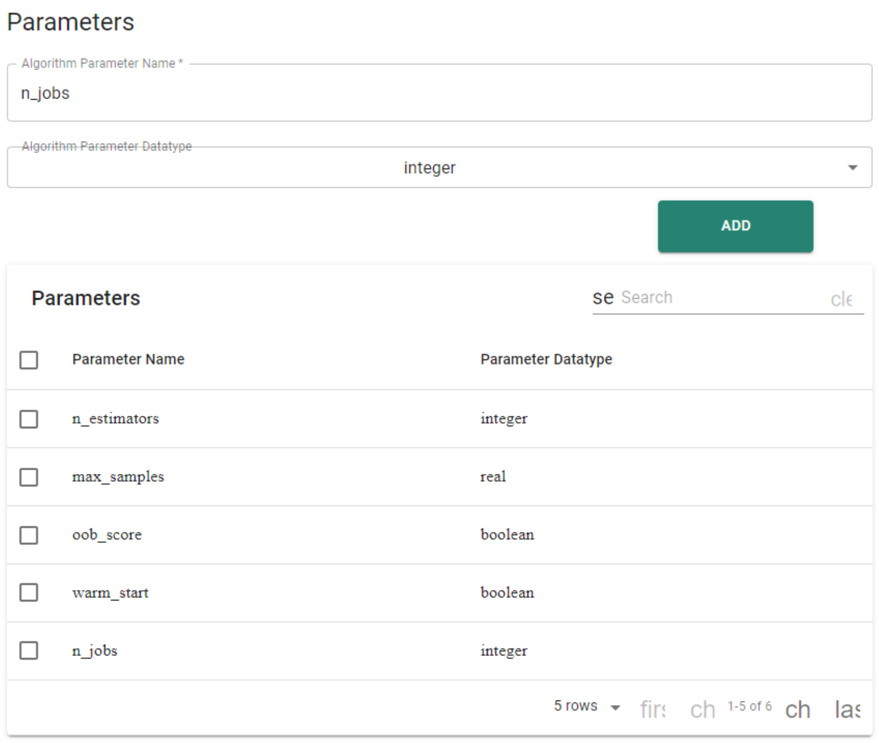
(Train) Time Complexity: linearithmic time - O(n log n)

Maths Notation (latex) *: O(n_{trees}n_{samples}n_{features}\log(n_{samples}))

Add the more specific time complexity representation using latex.

(b) Complexity section

Figure 5.6: BaggingClassifier pt. 2/3: Complete end-to-end annotation of an ensemble ML/DM algorithm.



Parameters

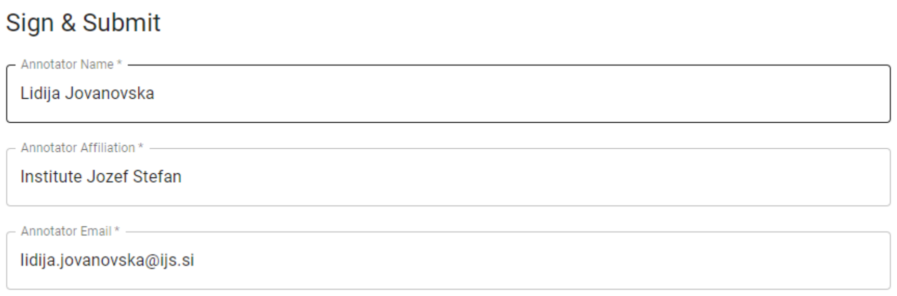
Algorithm Parameter Name *: n_jobs

Algorithm Parameter Datatype: integer

Parameters

| Parameter Name | Parameter Datatype |
|---|---|
| n_estimators | integer |
| max_samples | real |
| oob_score | boolean |
| warm_start | boolean |
| n_jobs | integer |

(a) Parameters section



Sign & Submit

Annotator Name *: Lidija Jovanovska

Annotator Affiliation *: Institute Jozef Stefan

Annotator Email *: lidija.jovanovska@ijs.si

(b) Annotator information section

Figure 5.7: BaggingClassifier pt. 3/3: Complete end-to-end annotation of an ensemble ML/DM algorithm.

## 5.2 Querying Use Case

In the querying use case, the user can retrieve relevant information from the ML/DM algorithms database. The querying process is done by specifying the values of three filters that constrain the database search operation. These filters include the type of task, the type of computational problem, and the time complexity specification. The querying tool enables the user to run a query, export the results in a format of their choice, and repeat the process quickly and easily.

For each filter, we run one query based on the condition if the filter is specified or not. This means that if the filter is not specified, we simply return all the ML/DM algorithms in the database, i.e., there is no filtering at all. Otherwise, if the filter is specified, only the ML/DM algorithms that have the specified filter value are retrieved. The process is repeated for each of the filters. Then, the results from all the queries are aggregated via an intersect operation, i.e., we retrieve only the algorithms that satisfy all of the filters. Finally, the results are presented to the user in a table view. In Table 5.1, we present all the queries that can be posed to the system using the querying tool. For a more intuitive understanding, the queries are described in natural language and the Cypher querying language.

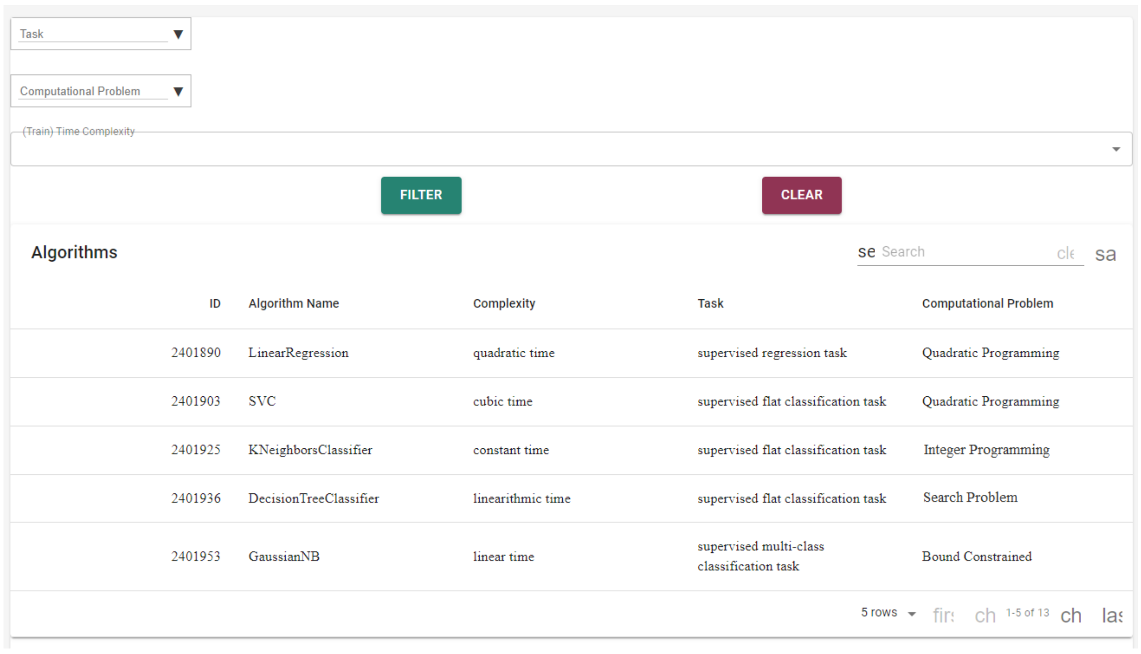
The table element used is searchable, editable, and exportable. Natively, the table supports exporting the results in CSV and PDF formats. Given that the user chooses the CSV export, they can use the data to perform analysis, or, even formulate ML problems and training algorithms using the dataset. The PDF format can be used for more presentational

Table 5.1: A complete list of the queries that the user can pose to the system. The queries are displayed in a natural language form, as well as in the Cypher querying language.

| Natural Language Query | Cypher Query | Filter Specified |
|---|---|---|
| Give me all the algorithms that address a task | MATCH (n:DataMiningAlgorithm)<br>-[:ADDRESSES]->(t:Task)<br>RETURN n | No |
| Give me all the algorithms that address the 'supervised flat classification task' | MATCH (n:DataMiningAlgorithm)<br>-[:ADDRESSES]->(t:Task)<br>WHERE t.name="supervised flat classification task"<br>RETURN n | Yes |
| Give me all the algorithms that solve a computational problem | MATCH(n:DataMiningAlgorithm)<br>-[:SOLVES]->(cp:ComputationalProblem)<br>RETURN n | No |
| Give me all the algorithms that solve the 'Linear Programming' problem | MATCH(n:DataMiningAlgorithm)<br>-[:SOLVES]->(cp:OptimizationProblem)<br>WHERE cp.name="Linear Programming"<br>RETURN n | Yes |
| Give me all the algorithms that have a train time complexity | MATCH(n:DataMiningAlgorithm)<br>-[:HAS_TRAIN_TIME_COMPLEXITY]->(c:Complexity)<br>RETURN n | No |
| Give me all the algorithms that have a Linearithmic complexity | MATCH(n:DataMiningAlgorithm)<br>-[:HAS_TRAIN_TIME_COMPLEXITY]->(c:Complexity)<br>WHERE c.name="Linearithmic complexity"<br>RETURN n | Yes |

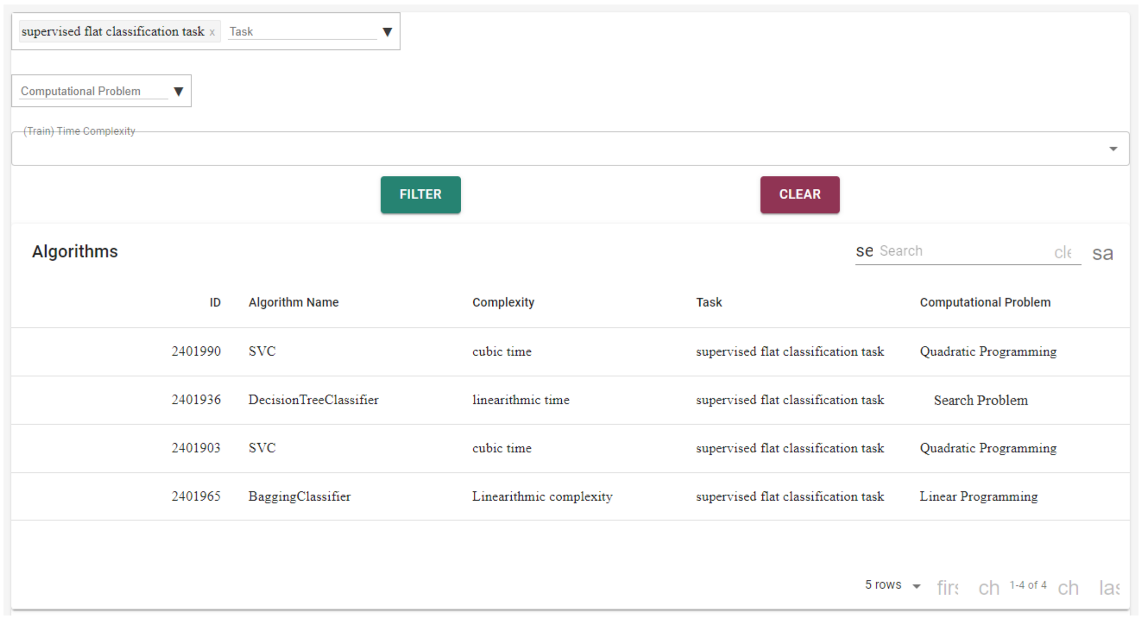
purposes. Additionally, there is the option to override the exporting functionality allowing the export to be done in a semantic format, such as OWL or RDF.

Examples of use case scenarios for the querying tool are presented in Figure 5.8. A querying use case with none of the filters specified is presented in Figure 5.8a, where information related to all the ML/DM algorithms in the repository is retrieved. In Figure 5.8b, the type of the task is specified, and set to be 'supervised flat classification task' and the corresponding algorithms that match the condition are retrieved.



| ID | Algorithm Name | Complexity | Task | Computational Problem |
|---|---|---|---|---|
| 2401890 | LinearRegression | quadratic time | supervised regression task | Quadratic Programming |
| 2401903 | SVC | cubic time | supervised flat classification task | Quadratic Programming |
| 2401925 | KNeighborsClassifier | constant time | supervised flat classification task | Integer Programming |
| 2401936 | DecisionTreeClassifier | linearithmic time | supervised flat classification task | Search Problem |
| 2401953 | GaussianNB | linear time | supervised multi-class classification task | Bound Constrained |

(a) Querying view — no filters specified



| ID | Algorithm Name | Complexity | Task | Computational Problem |
|---|---|---|---|---|
| 2401990 | SVC | cubic time | supervised flat classification task | Quadratic Programming |
| 2401936 | DecisionTreeClassifier | linearithmic time | supervised flat classification task | Search Problem |
| 2401903 | SVC | cubic time | supervised flat classification task | Quadratic Programming |
| 2401965 | BaggingClassifier | Linearithmic complexity | supervised flat classification task | Linear Programming |

(b) Querying view — the task type filter is specified

Figure 5.8: The GUI for the querying tool. The first figure shows the default view, while the second shows the results retrieved when the task filter is specified.

# Chapter 6

# Evaluation

In this thesis, we have presented a complete methodology for a bottom-up ontology design approach and a developed web-based application to semantically annotate, store, and query information related to ML/DM algorithms. While we presented multiple examples of annotation and querying scenarios, which validate our developed system, it is important to evaluate the developed resources more coherently. To do so, we turn to the FAIR principles which are commonly used as guidelines for developing digital resources, with a focus on the resource being findable, accessible, interoperable, and reusable. In addition, we assess the developed ontology extension by using OBO Foundry principles. The evaluation of the developed resources is the topic of this chapter.

In the literature, there are many ontology evaluation approaches, which can be divided into several categories: 1) gold standard-based (by comparing the developed ontology with a reference, "gold standard" ontology), 2) corpus-based (by evaluating how well the developed ontology covers the content of a text corpus in the given domain), 3) task-based (by measuring how much the ontology helps with improving the results of a certain task), 4) criteria-based (by measuring how much the ontology adheres to certain desirable criteria) (Raad & Cruz, 2015).

In the previous chapter, we presented several ML/DM algorithm annotations which have been created using an annotation schema based on the developed ontology. By doing so, we followed a corpus-based ontology evaluation approach, carried out manually, rather than by automatically extracting terms from the corpus. The choice of evaluating manually rather than automatically was based on the fact that automatic term extraction from a text corpus is an extremely difficult task in the domain of ML/DM. This will be discussed in more detail in the next chapter.

## 6.1 Assessment Using the FAIR Guiding Principles

The FAIR guiding principles for scientific data management and stewardship were conceived to serve as guidelines for those who wish to enhance the reusability and invaluableness of their data holdings (Wilkinson et al., 2016). The power of these principles lies in the fact that they are related but independent and separable. They are also simple and minimalistic in design and as such can be adapted to various application scenarios. We will now present the principles in more detail:

- *Findability* ensures that a globally unique and persistent identifier is assigned to the data and the metadata which describes the data. Also, both resources are registered or indexed in a searchable resource.

- *Accessibility* ensures that the data and the metadata can be retrieved by their identifier using a standardized communications protocol. The protocol must be open and free or in other cases, it must allow for an authentication and authorization procedure. Also, metadata should be accessible, even when the data is no longer available.
- *Interoperability* ensures that data, as well as metadata, use a formal, accessible, and shared language for knowledge representation. The data and metadata must use and reference qualified vocabularies that follow FAIR principles.
- *Reusability* ensures that data and metadata are described with accurate and relevant attributes, released with a clear and accessible license, have detailed provenance, and meet domain-relevant community standards.

Let us now describe the compliance with the FAIR guidelines presented in Table 6.1. We will go over each of the principle groups in more detail.

Table 6.1: Assessment of the developed repository using The FAIR principles.

| # | Principle | Compliance |
|---|---|---|
| F1. | (Meta)data are assigned a globally unique and persistent identifier | Yes |
| F2. | Data are described with rich metadata | Partial |
| F3. | Metadata clearly and explicitly include the identifier of the data they describe | Yes |
| F4. | (Meta)data are registered or indexed in a searchable resource | Yes |
| A1. | (Meta)data are retrievable by their identifier using a standardized communications protocol | No |
| A1.1 | The protocol is open, free, and universally implementable | Yes |
| A1.2 | The protocol allows for an authentication and authorization procedure, where necessary | Yes |
| A2. | Metadata are accessible, even when the data are no longer available | Yes |
| I1. | (Meta)data use a formal, accessible, shared and broadly applicable language for knowledge representation | Yes |
| I2. | (Meta)data use vocabularies that follow FAIR principles | Yes |
| I3. | (Meta)data include qualified references to other (meta)data | Yes |
| R1. | (Meta)data are richly described with a plurality of accurate and relevant attributes | Partial |
| R1.1. | (Meta)data are released with a clear and accessible data usage license | Yes |
| R1.2. | (Meta)data are associated with detailed provenance | Partial |
| R1.3. | (Meta)data meet domain-relevant community standards | Yes |

**Findability.** The developed resources comply with F1 because each entity in the graph database is assigned an application-generated identifier using neomodel utilities (UniqueIdProperty). Though this identifier is not globally unique, once the graph database is exported in an RDF format, the identifier combined with the ontology URI makes the data globally unique. Additionally, the developed ontology has a persistent URI which is registered in the BioPortal repository. We also rely on previously defined identifiers, such as the Digital Object Identifier (DOI), used to identify documents where a given ML/DM algorithm is described. As for F2, the developed resources comply to a certain extent. While metadata related to the annotator is stored, along with a time stamp, other metadata can also be stored, based on their further potential usage. The metadata is usually stored as node or relationship properties in the graph database, i.e, the DOI is a property of the Document entity, and the annotation metadata is stored as properties of the Annotation entity. Because of this, the metadata is explicitly linked with the data they describe, complying with F3. To comply with the final Fairness criterium — F4, the metadata and data must be registered in a searchable resource. This was ensured using the Querying Interface in the web application, where a user can view and export all the required information from the repository, as well as specify search constraints. A potential improvement to increase flexibility would be to allow for the user to query the repository using the repository query language (i.e., Cypher), however, this would mean that the user would have to know how to formulate such queries.

**Accessibility.** Closely related to F4 is the A1 criterium. While the user can in principle retrieve all the data in the repository, he cannot do so by merely specifying the identifier of the data, in this case, the entity identifier. The reason for this is that by design there is no Search feature, through which the user can specify an identifier and obtain results. Because of this, the resource does not explicitly comply with the A1 criterium. However, the protocol is completely free and open-source, i.e., anyone with a computer and an internet connection can access the complete repository. Hence, it complies with A 1.1. Moreover, all of the technologies used to develop the application are free (Neo4j), and some are even open-source (Django, React, neomodel). Since the design of the application is not user-based, one does not need to log in every time one uses the application. However, a key protocol for validating the quality of the annotation is the e-mail validation procedure described in the previous chapter (A 1.2). The annotation metadata and (meta)data related to an ML/DM algorithm are retained in the repository even if the ML/DM algorithm is removed, providing compliance with A2.

**Interoperability.** For compliance with I1., it is critical to use (1) commonly used controlled vocabularies, ontologies, thesauri, and a good data model (a well-defined framework to describe and structure (meta)data). This was ensured through our reliance on the developed ontology — OntoDM-algorithms and the ontology-based annotation schema which represented the database conceptual model. As for I2., we mainly rely on OntoDM, an ontology that follows FAIR principles, along with the ontologies which are imported in OntoDM. In I3., a qualified reference is a cross-reference that explains its intent. The goal is to create as many meaningful links as possible between (meta)data resources to enrich the contextual knowledge about the data, balanced against the time/energy involved in making a good data model. In practice, this guideline has been followed in the database conceptual model, where each of the relationships between the entities in the graph database has semantic meaning.

**Reusability.** R1. states that the (meta)data should be richly described, including information such as the scope, limitations, conditions, versioning, etc. While the data itself is to a certain extent richly described, there is always room for more explicit data, i.e., more can be done in this regard. The metadata however is not sufficiently described, since only the annotator information and the time stamp of the annotation are stored. Much more can be done to adhere to this guideline. Since this work was aimed to facilitate science, all the data is public and free to use so there is no data usage license associated with the data. Clearly stating this fact helps us meet the R 1.1 criterium. The annotator information ensures data provenance and accountability, however, the data in question is sensitive and it was not designed to be published, so R 1.2 is met only partially. Finally, since there are not many data repositories of this form, there are no specific domain-relevant community standards to comply with, as stated in R 1.3.

## 6.2 Assessment Using The OBO Foundry Principles

The OBO Foundry applies the key principles that ontologies should be open, orthogonal, instantiated in a well-specified syntax, and designed to share a common space of identifiers (Smith et al., 2007). Open means that the ontologies should be available for use without any constraint or license and also receptive to modifications proposed by the community. Orthogonal means that they ensure the additivity of annotations and comply with modular development. The proper and well-specified syntax is expected to support algorithmic processing and the common system of identifiers enables backward compatibility with legacy annotations as the ontologies evolve.

We checked the compliance of OntoDM-algorithms to the OBO Foundry principles which are presented in detail in Table 6.2. Mainly, we comply with most of the OBO Foundry principles, barring two. Regarding P3, we still do not have a Permanent URL (PURL), however, the ontology is available on BioPortal and it is linked to the ontology file in a GitHub repository which has a URL that, unless we edit manually, will persist. As for P6, definitions are provided for the top-level entities that are reused from other ontologies, such as OntoDM-core, however, we have not yet added strict definitions for the newly added entities. This is something we plan to do in future work.

## 6.3 Statistical Ontology Metrics of the OntoDM-algorithms Ontology Extension

Here, we refer to the statistical ontology metrics from the Protégé software and the BioPortal web service. This includes metrics such as the number of classes and individuals, the number of properties, maximum depth, maximum and average number of children, classes with a single child, classes with more than 25 children, and classes with no definition. The values of the statistical ontology metrics for OntoDM-algorithms are presented in Table 6.3. Some of the metrics are merely statistical and are used to determine the size of the ontology (i.e., number of properties, number of classes, etc.). Another part of the metrics gives some indication of the quality of the ontology. For example, while technically there is no problem in having only one subclass, (i.e., having multiple Classes with a single child) this situation often indicates that either the hierarchy is under-specified, or the distinction between the class and the subclass is not appropriate. Additionally, a class that has more than 25 subclasses is a candidate for additional distinctions and categorization is needed. In OntoDM-algorithms, there are no classes with more than 25 children, and the number of classes with a single child is fairly low (only 13 out of 377 classes).

Table 6.2: Assessment of OntoDM-algorithms' compliance with the OBO Foundry principles.

| # | Principle | Compliance |
|---|---|---|
| P1 | **Open** — The ontology MUST be openly available to be used by all without any constraint other than (a) its origin must be acknowledged and (b) it is not to be altered and subsequently redistributed in altered form under the original name or with the same identifiers. | Yes |
| P2 | **Common Format** — The ontology is made available in a common formal language in an accepted concrete syntax. | Yes |
| P3 | **URI/Identifier Space** — Each ontology MUST have a unique IRI in the form of an OBO Foundry permanent URL (PURL). | No |
| P4 | **Versioning** — The ontology provider has documented procedures for versioning the ontology, and different versions of ontology are marked, stored, and officially released. | Yes |
| P5 | **Scope** — The scope of an ontology is the extent of the domain or subject matter it intends to cover. The ontology must have a specified scope and content that adheres to that scope. | Yes |
| P6 | **Textual Definitions** — The ontology has textual definitions for the majority of its classes and top-level terms in particular. | No |
| P7 | **Relations** — Relations should be reused from the Relations Ontology (RO). | Yes |
| P8 | **Documentation** — The owners of the ontology should strive to provide as much documentation as possible. | Yes |
| P9 | **Documented Plurality of Users** — The ontology developers should document that the ontology is used by multiple independent people or organizations. | Yes |
| P10 | **Commitment To Collaboration** — OBO Foundry ontology development, in common with many other standards-oriented scientific activities, should be carried out collaboratively. | Yes |
| P11 | **Locus of Authority** — There should be a person who is responsible for communications between the community and the ontology developers, for communicating with the Foundry on all Foundry-related matters, for mediating discussions involving maintenance in the light of scientific advance, and for ensuring that all user feedback is addressed. | Yes |
| P12 | **Naming Conventions** — The names (primary labels) for elements (classes, properties, etc.) in an ontology must be intelligible to scientists and amenable to natural language processing. Primary labels should be unique among OBO Library ontologies. | Yes |
| P13 | **Maintenance** — The ontology needs to reflect changes in scientific consensus to remain accurate over time. | Yes |

Table 6.3: Statistical metrics for the OntoDM-algorithms ontology extension.

| Ontology Metric | # |
|---|---|
| Classes | 377 |
| Individuals | 0 |
| Properties | 1 |
| Maximum depth | 10 |
| Maximum number of children | 20 |
| Average number of children | 2 |
| Classes with a single child | 13 |
| Classes with more than 25 children | 0 |
| Classes with no definition | 178 |

# Chapter 7

# Semi-automatic Population of a Knowledge Base for ML/DM Algorithms

In the previous chapters, we explored the top-down approach to semantic annotation of ML/DM algorithms, where we used knowledge-driven (bottom-up) methods to manually create the annotations. Due to the scaling advantages of semi-automatic annotation, experiments were made by following a bottom-up approach by using data-driven methods. The findings of these experiments are the topic of this chapter. The task is to explore the quality of annotations produced by pre-trained language models, as well as to follow a bottom-to-end approach to training a NER model from scratch. First, we will describe the data resources that are used in the approach. Then, we will describe how the annotation process was conducted, as well as the obtained findings. We will then provide a qualitative evaluation of annotations produced by pre-trained language models. Finally, we will describe the procedure for training a NER model using the manually annotated resources, as well as present the results in a discussion.

**Two automation use cases — knowledge base creation and knowledge base population.** Data-driven methods for automatic semantic annotation can be used for two main use cases. First, the ontology (or the annotation schema) can be created automatically. This task can be modeled as a joint entity and relation extraction, where we could generate multiple annotation schemas, which can be ranked and reviewed by experts. Solving the task would most probably require large pre-trained language models, or fine-tuned on text data in the domain of ML/DM, like for example theoretical books.

Second, the ontology/annotation schema can be created manually, while the annotation process can be automated. If the schema is expressive enough to include relation types, the task can be modeled identically as in the first use case. Otherwise, if the schema consists of only entities, and potentially relations between entities, which designate hierarchy or simply a connection indicator, the problem is most commonly modeled as a named entity recognition task. Large language models would be utilized here as well, although trained, or fine-tuned on text data like conference and journal paper abstracts or even full papers. By following this approach, we can automatically create multiple semantic annotations of ML/DM algorithms.

The second use case is more favorable, as it addresses a less complex problem, i.e., identifying entities instead of full networks of entities and relations. Additionally, there are plenty more data resources that could be used to train the models for this task, making the data acquisition and processing stage much easier. That is why for the scope of this thesis, we will follow the second approach.

## 7.1 Data Resources

Two data corpora were utilized: the entire SCIERC corpus of scientific paper abstracts (Luan et al., 2018), and a corpus of scientific paper abstracts published in the Arxiv repository. The SCIERC corpus will be used to train a NER model from scratch, while the Arxiv corpus will be used to experiment with the annotation process and qualitatively evaluate pre-trained language models.

To prepare the Arxiv corpus for the annotation process we needed to obtain a small random sample of 50 paper abstracts, which we would further manually annotate by labeling the entities of interest. We constrain the sampling space to only the Machine Learning category, a sub-category of the Computer Science category. The annotation schema is a simplified version of the SCIERC annotation schema. It consists of three main entities: Method (i.e., Algorithm, Model), Task (i.e., Application, Problem), and Resource (i.e., Data, Corpus, Dataset). The idea was that the annotations (if sufficient in quantity) would be then used to train a neural network model which would optimally generalize well and recognize entities in unseen paper abstracts.

## 7.2 Annotation Process

The annotation process consisted of manually reading paper abstracts and selecting the named entities which occur in the text. This can be carried out either using text annotation tools such as UBIAI, or Prodigy, or simply with a text editor. The advantage of using specialized tools is that they support exporting the annotations in common formats such as spacy, IOB, CSV, etc. (Ramshaw & Marcus, 1999). However, the tools can usually be accessed only through a paid subscription, which was not in the scope of this research.

Even though there were only three entities, the annotation process turned out to be extremely difficult. This was largely because annotation of scientific text requires domain expertise which makes annotation costly with respect to time and effort. Since we only had a single annotator at our disposal, we were only able to produce 50 annotated paper abstracts, which is only 10% of the SCIERC corpus. Additionally, inter-annotator agreement, a key metric to assess the quality of the annotations, could not be measured and analyzed. Despite the modest size of the annotated corpus, we were able to draw several notable conclusions which could in turn serve other researchers working on the topic.

**The interchangeable use of Task, Problem, and Application.** The task entity types as defined in OntoDM-core ontology are rarely used in written text, apart from the leaf entities like binary classification task or regression task. In reality, the term task is synonymous with the term problem, while the term application represents the setting in which the algorithm is utilized. However, syntactically these three terms are phrased in a very similar way, for example 'this method was developed for / to address [task/problem/application]'. Instances include 'representation learning', 'pose identification', 'real estate appraisal', etc. Due to this ambiguity and the simple annotation schema, we label all the entities belonging to these vaguely (dis)similar groups, with the Task entity.

**Methods are rarely named entities.** Most machine learning practitioners do not usually name the algorithms they invent. More often the novel aspect of a research paper involves a small contribution in the form of altering or adding some method to the procedure. Instead, algorithms are usually described with more wordy sentences, exposing their inner workings, like "learnable data augmentation method that is jointly learned with the embeddings by leveraging the inherent signal encoded in the graph".

**High variability in entity word length.** Start and end positions of the entity terms, are hard to determine, i.e., (highly complex) sequential decision making, or 'predicting the structure of a protein from its sequence'. Similarly, as the SCIERC annotators, we perform a greedy annotation for spans and always prefer the longer span whenever ambiguity occurs.

## 7.3 Pre-trained Language Models for NER

The large variants of the spaCy and sciSpacy English language models were used to evaluate the quality of pre-trained language models for the NER task in the domain of ML/DM. Note that these models are not fine-tuned using domain data, rather they are used directly via the corresponding libraries. If the models proved to recognize entities effectively, judged by virtue of observance and qualitative analysis, then there would be no need for training new models from scratch. It was however somewhat expected that the results would not be ideal. This was more obvious with the spaCy language model since it was trained on general data, with generic entity labels, which were of no use to us, such as Organization, Person, Location, etc.

The prominent usage of ML/DM methods in the biomedical domain gave sciSpacy promise for improved results. Indeed, it was able to recognize a large number of entities, albeit much more than was needed for the specific use case. But, because it was trained on data from a different domain, with no fine-tuning, the results were acceptable. There were several issues with the model: the model does not distinguish between entity types; issues with entity span choice (i.e., instead of being annotated as one entity, the term 'graph neural networks' was annotated as having two entities 'graph' and 'neural networks'); the model recognizes many terms to be entities (i.e., 'limitation', 'current', 'prediction', etc.). The annotations produced by each of the models, as well as the one produced manually by us, can be seen in Figure 7.1.

## 7.4 Training a NER Model From Scratch

Since the annotations produced by the language models were not of sufficient quality, we explored the possibility of training a NER model from scratch using the SCIERC corpus. Initially, we wanted to train a model using the annotations that we produced, however, because the annotation process proved to be a very time-consuming endeavor, we ended up using the much bigger SCIERC corpus (Luan et al., 2018). The model architecture consisted of an Embedding layer, a Transformer block layer, and two pairs of Fully Connected and Dropout Layers, totaling 133,736 trainable (model) parameters. The Transformer block layer includes a Multihead Attention Layer, followed by a Fully Connected Layer, Normalization, and Dropout Layers. The Multihead Attention Layer allows the model to jointly attend to information from different representation subspaces (Vaswani et al., 2017).

To train the model we used the sparse categorical cross entropy loss function which is commonly used for multi-class classification problems because it outputs a probability distribution over the class labels. The SCIERC corpus was split into 90% training data (450 paper abstracts) and 10% validation data (50 paper abstracts). We used the Adam optimization algorithm to train the model over 100 epochs. The results obtained can be seen in Table 7.1.

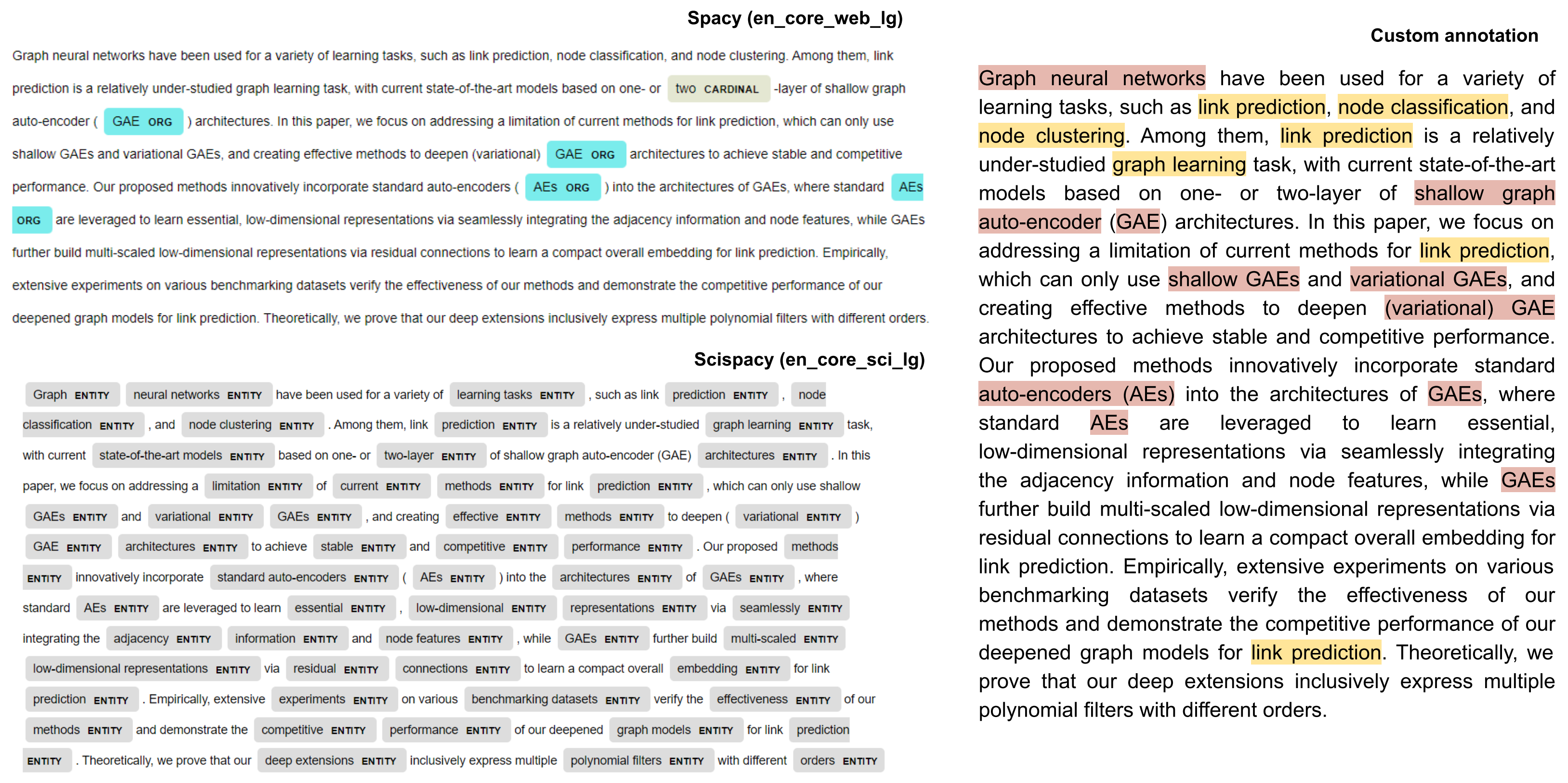
Table 7.1: The performance of the trained NER model for each entity type.

Three key metrics are covered, as well as the support for each entity type.

| | Precision | Recall | F-Score | Support |
|---|---|---|---|---|
| Overall | 42.02% | 42.22% | 42.12 | |
| Material | 92.56% | 78.34% | 84.86 | 336 |
| Method | 17.51% | 14.73% | 16.00 | 457 |
| Task | 35.31% | 43.06% | 38.80 | 861 |

## 7.5 Results and Potential Improvements

As this study was an experimental one, plenty of methods were briefly explored, while no conclusive result is presented. There is enormous potential for improving every aspect of the methodology, from the annotation process (bigger corpora, more annotators, larger annotation schema) to the model development (fine-tuning pre-trained models, experimenting with different model architectures, etc.). However, it must be noted that this task is an extremely difficult one. Essentially, there is not enough in-depth information in paper abstracts to create a detailed ML/DM algorithm annotation. Perhaps, there is sufficient information to populate higher-level concepts such as the ones we selected, but essentially specific algorithm information, such as its computational complexity, parameters, and other similar entities from our ontology-based annotation schema, is rarely found in a paper abstract. Instead, this information can probably (but not always) be found in the full paper which in turn introduces a new level of complexity with the need of utilizing a paper parsing procedure. The annotation process, which is subjected to the limitations we described previously due to the complexity and the ambiguity of the ML/DM domain, suggests that perhaps this type of annotation has to be done manually.



**Spacy (en_core_web_lg)**

Graph neural networks have been used for a variety of learning tasks, such as link prediction, node classification, and node clustering. Among them, link prediction is a relatively under-studied graph learning task, with current state-of-the-art models based on one- or two CARDINAL -layer of shallow graph auto-encoder ( GAE ORG ) architectures. In this paper, we focus on addressing a limitation of current methods for link prediction, which can only use shallow GAEs and variational GAEs, and creating effective methods to deepen (variational) GAE ORG architectures to achieve stable and competitive performance. Our proposed methods innovatively incorporate standard auto-encoders ( AEs ORG ) into the architectures of GAEs, where standard AEs ORG are leveraged to learn essential, low-dimensional representations via seamlessly integrating the adjacency information and node features, while GAEs further build multi-scaled low-dimensional representations via residual connections to learn a compact overall embedding for link prediction. Empirically, extensive experiments on various benchmarking datasets verify the effectiveness of our methods and demonstrate the competitive performance of our deepened graph models for link prediction. Theoretically, we prove that our deep extensions inclusively express multiple polynomial filters with different orders.

**Scispacy (en_core_sci_lg)**

Graph ENTITY neural networks ENTITY have been used for a variety of learning tasks ENTITY , such as link prediction ENTITY , node classification ENTITY , and node clustering ENTITY . Among them, link prediction ENTITY is a relatively under-studied graph learning ENTITY task, with current state-of-the-art models ENTITY based on one- or two-layer ENTITY of shallow graph auto-encoder (GAE) architectures ENTITY . In this paper, we focus on addressing a limitation ENTITY of current ENTITY methods ENTITY for link prediction ENTITY , which can only use shallow GAEs ENTITY and variational ENTITY GAEs ENTITY , and creating effective ENTITY methods ENTITY to deepen ( variational ENTITY ) GAE ENTITY architectures ENTITY to achieve stable ENTITY and competitive ENTITY performance ENTITY . Our proposed methods ENTITY innovatively incorporate standard auto-encoders ENTITY ( AEs ENTITY ) into the architectures ENTITY of GAEs ENTITY , where standard AEs ENTITY are leveraged to learn essential ENTITY , low-dimensional ENTITY representations ENTITY via seamlessly ENTITY integrating the adjacency ENTITY information ENTITY and node features ENTITY , while GAEs ENTITY further build multi-scaled ENTITY low-dimensional representations ENTITY via residual ENTITY connections ENTITY to learn a compact overall embedding ENTITY for link prediction ENTITY . Empirically, extensive experiments ENTITY on various benchmarking datasets ENTITY verify the effectiveness ENTITY of our methods ENTITY and demonstrate the competitive ENTITY performance ENTITY of our deepened graph models ENTITY for link prediction ENTITY . Theoretically, we prove that our deep extensions ENTITY inclusively express multiple polynomial filters ENTITY with different orders ENTITY

**Custom annotation**

Graph neural networks have been used for a variety of learning tasks, such as link prediction, node classification, and node clustering. Among them, link prediction is a relatively under-studied graph learning task, with current state-of-the-art models based on one- or two-layer of shallow graph auto-encoder (GAE) architectures. In this paper, we focus on addressing a limitation of current methods for link prediction, which can only use shallow GAEs and variational GAEs, and creating effective methods to deepen (variational) GAE architectures to achieve stable and competitive performance. Our proposed methods innovatively incorporate standard auto-encoders (AEs) into the architectures of GAEs, where standard AEs are leveraged to learn essential, low-dimensional representations via seamlessly integrating the adjacency information and node features, while GAEs further build multi-scaled low-dimensional representations via residual connections to learn a compact overall embedding for link prediction. Empirically, extensive experiments on various benchmarking datasets verify the effectiveness of our methods and demonstrate the competitive performance of our deepened graph models for link prediction. Theoretically, we prove that our deep extensions inclusively express multiple polynomial filters with different orders.

Figure 7.1: Named entity annotations produced using the spaCy and scispaCy libraries, as well as through manual annotation.

# Chapter 8

# Conclusions

## 8.1 Summary

The main goal of this thesis was to improve the semantic annotation of DM/ML algorithms. The increasing need for using DM/ML algorithms in our everyday life and consequently the increase in research in these domains led to the need for better knowledge representation in these domains. At the beginning of the thesis, we described the motivation for our work, and we gave an overview of the state of the art in knowledge representation in the domains of DM/ML. While the related work proved to be a significant foundation for our work, there was a need for more explicit knowledge representation of DM/ML algorithms. We addressed this need by developing the OntoDM-algorithms ontology extension which includes entities needed to characterize DM/ML algorithms including the computational problem taxonomy, the model and algorithm parameter entities, the computational complexity entity, the assumption specification, the ensemble algorithm taxonomy, and the sampling entity. Furthermore, an ontology-based annotation schema was designed to enable semantic annotation of DM/ML algorithms.

The annotation schema provided a basis for the conceptual model of the web-based application for semantic annotation and querying of DM/ML algorithms. We designed and implemented the application, using mainly free and open-source technologies, such as Django, React, and Neo4j. The application itself comprises two distinct functional parts, an annotation tool, and a querying tool. With the annotation tool, the user can input relevant information about any DM/ML algorithm through a simple, user-friendly interface. Once the user submits the annotation and validates their personal details, the annotation is validated and stored in a Neo4j graph database, which allows for extensive semantics, such as inference, export in semantic formats, etc. Meanwhile, using the querying tool, the user can query the database for specific DM/ML algorithms, using several designated filters. Additionally, the results of the queries can be exported in several formats.

The application was used to annotate a set of different DM/ML algorithms, evaluate the capability of the ontology to annotate representative data, as well as to test that the application is working as expected. This resulted in a small repository of DM/ML algorithms that can serve as a guide to future users of the application. What's more, the annotation process was discussed in detail, which paved the way for future work in this regard. We then evaluated the developed resources, following the FAIR Guiding Principles for scientific data management and stewardship.

Finally, we followed an approach for a semi-automatic population of a knowledge base for DM/ML algorithms. On one hand, we evaluated the quality of the annotations produced by pre-trained language models, while on another, we devised a methodology to train a NER model from scratch. To do so, we produced a small corpus of 50 annotated paper abstracts, as well as the findings from the annotation process and the methods used. We concluded that this task proved to be very difficult largely due to the nature of the data and the complexity of the domain, however much can be improved in that regard.

## 8.2 Contributions of the Thesis

This thesis contributes to the field of knowledge representation in the domains of DM/ML. More specifically, the contributions of the thesis are listed in the remainder of this section.

- SR1. An ontology schema extension of the OntoDM ontology, named OntoDM-algorithms for improved knowledge representation with respect to domain algorithms.

  We designed OntoDM-algorithms, an ontology extension focusing on improving the representation of DM/ML algorithms. The extension includes main terms from OntoDM-core, such as algorithm, dataset, task, and generalization, and novel entities such as the computational problem taxonomy, the model and algorithm parameter entities, the computational complexity entity, the assumption specification, the ensemble algorithm taxonomy, and the sampling entity.

- SR2. A web-based tool for manual semantic annotation of DM/ML algorithms.

  We designed and implemented a web-based application for manual semantic annotation, storage, and querying of DM/ML algorithms. The application enables users to contribute to an open repository of DM/ML algorithms. Additionally, the users can retrieve relevant information about DM/ML algorithms that are stored in the database, helping them to find an adequate algorithm for a given scenario.

- SR3. A populated domain knowledge base of algorithms.

  Using the developed application for manual semantic annotation we annotated a set of different DM/ML algorithms. This repository is publicly available through the application and is set to be used by the users to acquire information about DM/ML algorithms, as well as to serve as a guide for the manual annotation process.

- SR4. A pilot study that demonstrates the use of a semi-automatic approach to create and populate a domain knowledge base.

  We explored a methodology for a bottom-up approach for populating a database of DM/ML algorithms. This methodology included the evaluation of the capability of pre-trained language models to automatically annotate paper abstracts describing DM/ML algorithms. Moreover, text corpora containing DM/ML paper abstracts were used to train and evaluate a language model from scratch. Although the results were not exemplary, we used the lessons learned from this approach to formulate open questions that could be explored in the future.

## 8.3 Future Work

This thesis touched upon many difficult tasks. The contributions display the importance of this work and the amount of effort put into the thesis. However, every research begets potential improvements and undoubtedly opens up many new research questions. We will now briefly present the many ways in which we can improve what has been done so far.

**Extending the ontology.** While several novel entities were added to the ontology, the design was meant to be on a general level, and not include specifics of the inner workings of each of the DM/ML algorithms (i.e., tree-based, deep learning models, etc.). In the future, we can expand the ontology to better represent each of these algorithm groups. We can also increase the number of relationships in the ontology, by defining the relationship between, for example, the computational complexity and the size of the input dataset. Moreover, we can extend the Computational Problem taxonomy to define each problem type at a more granular level.

**Improving the web application.** Here, more improvements can be done in terms of the UI, by providing examples for each of the input fields and invaluable references, to enable a better user experience and ultimately annotations of higher quality. Additionally, we can define a larger set of validation procedures to ensure annotation correctness, i.e., minimize the number of empty fields, and check for logic consistency (e.g., the dataset and task correspondence). Finally, we can also implement an RDF/OWL export of the query results in the application.

**Improving semantics and inference.** The neosemantics module of Neo4j allows for extensive semantics capabilities. One of the modules that can be used is the inference module. For example, we can use the inferred version of the ontology to extend the annotation schema, as well as run inference on demand in the querying interface, and retrieve not only explicit but also implicit knowledge. For example, if we know that a given DM/ML algorithm solves a binary classification task, while the type of the dataset is not available in the annotation, if we run a query to retrieve all the algorithms that have a binary classification dataset as input, we can implicitly reason using the task-dataset correspondence that we can retrieve also the algorithm that solves the corresponding task.

**Improving the semi-automatic annotation approach.** This approach was not explored thoroughly, however many lessons were learned and potential venues for improvement were identified. Namely, every part of the approach can be further improved, starting from the annotation process to the data selection, and the model selection and training. First, the annotation process can be expanded to include several annotators which would enable the evaluation of inter-annotator agreement, and ultimately more consistent annotations. Next, large pre-trained language models can be fine-tuned on the annotated data. These can be further evaluated in depth, both qualitatively and quantitatively. Additionally, many model architectures can be explored and tuned to arrive at a satisfactory performance quality.

# References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 265–283.

Arp, R., Smith, B., & Spear, A. D. (2015). *Building ontologies with basic formal ontology.* Mit Press.

Axios — promise based http client for the browser and node.js [Accessed on 02.09.2022]. (2020). https://axios-http.com/

Berkeley, G. (1881). *A treatise concerning the principles of human knowledge.* JB Lippincott & Company.

Bolt protocol [Accessed on 02.09.2022]. (2015). https://boltprotocol.org/

Borgo, S., Ferrario, R., Gangemi, A., Guarino, N., Masolo, C., Porello, D., Sanfilippo, E. M., & Vieu, L. (2006). Dolce: A descriptive ontology for linguistic and cognitive engineering. *Applied Ontology*, *3*, 1–3.

Bourget, D., & Chalmers, D. J. (2020). *Philosophers on philosophy: The 2020 philpapers survey.*

Breiman, L. (1996). Bagging predictors. *Machine learning*, *24*(2), 123–140.

Breiman, L. (1999). Pasting small votes for classification in large databases and on-line. *Machine learning*, *36*(1), 85–103.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (2017). *Classification and regression trees.* Routledge.

Browning, J. (2022). Ai and the limits of language [Accessed on 02.09.2022]. https://www.noemamag.com/ai-and-the-limits-of-language/

Ceusters, W. (2012). An information artifact ontology perspective on data collections and associated representational artifacts. *MIE*, 68–72.

Chowdhary, K. (2020). Natural language processing. *Fundamentals of artificial intelligence*, 603–649.

Clifton, C. (2022). Data mining [Accessed on 02.09.2022]. https://www.britannica.com/technology/data-mining

Comesaña, J., & Klein, P. (2019). Skepticism [Accessed on 02.09.2022]. https://plato.stanford.edu/entries/skepticism/

Consortium, G. O. (2004). The gene ontology (go) database and informatics resource. *Nucleic acids research*, *32*(suppl_1), D258–D261.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, *20*(3), 273–297.

Courtot, M., Gibson, F., Lister, A. L., Malone, J., Schober, D., Brinkman, R. R., & Ruttenberg, A. (2011). Mireot: The minimum information to reference an external ontology term. *Applied Ontology*, *6*(1), 23–33.

Cypher query language [Accessed on 02.09.2022]. (2022). https://neo4j.com/developer/cypher/

Degtyarenko, K., De Matos, P., Ennis, M., Hastings, J., Zbinden, M., McNaught, A., Alcántara, R., Darsow, M., Guedj, M., & Ashburner, M. (2007). Chebi: A database and ontology for chemical entities of biological interest. *Nucleic acids research*, *36*(suppl_1), D344–D350.

Descartes, R. (1999). *Discourse on method and meditations on first philosophy*. Hackett Publishing.

Donald, E. K. et al. (1999). The art of computer programming. *Sorting and searching*, *3*, 426–458.

Džeroski, S. (2006). Towards a general framework for data mining. *International Workshop on Knowledge Discovery in Inductive Databases*, 259–300.

Esteves, D., Moussallem, D., Neto, C. B., Soru, T., Usbeck, R., Ackermann, M., & Lehmann, J. (2015). Mex vocabulary: A lightweight interchange format for machine learning experiments. *Proceedings of the 11th International Conference on Semantic Systems*, 169–176.

Forcier, J., Bissex, P., & Chun, W. J. (2008). *Python web development with django*. Addison-Wesley Professional.

Forouzan, B. A. (2002). *Tcp/ip protocol suite*. McGraw-Hill Higher Education.

Freitag, D. (2000). Machine learning for information extraction in informal domains. *Machine learning*, *39*(2), 169–202.

Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, *55*(1), 119–139.

Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of statistics*, 1189–1232.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational statistics & data analysis*, *38*(4), 367–378.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, *5*(2), 199–220.

Hinton, G. E. (2009). Deep belief networks. *Scholarpedia*, *4*(5), 5947.

Ho, T. K. (1998). The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, *20*(8), 832–844.

Hoerl, A. E., & Kennard, R. W. (1970). Ridge regression: Applications to nonorthogonal problems. *Technometrics*, *12*(1), 69–82.

Howard, R. (2017). Rdf triple stores vs. labeled property graphs: What's the difference? [Accessed on 02.09.2022]. https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/

Jackson, R. C., Balhoff, J. P., Douglass, E., Harris, N. L., Mungall, C. J., & Overton, J. A. (2019). Robot: A tool for automating ontology workflows. *BMC bioinformatics*, *20*(1), 1–10.

Jovanovska, L. (2020). Semantic analysis of lyrics for genre labeling and improving music recommendation. *International Symposium Multilinguality in Folklore*, *1*(1), 52.

Jovanovska, L., Dimovski, A., & Joksimoski, B. (2018). Interactive digital environment for learning historical events. *15th International Conference on Informatics and Information Technologies (CIIT)*.

Jovanovska, L., & Evkoski, B. (2020). Analysis of chord progression networks.

Jovanovska, L., Evkoski, B., Mirchev, M., & Mishkovski, I. (2020). Demographic analysis of music preferences in streaming service networks. *Complex networks xi* (pp. 233–242). Springer.

Jovanovska, L., Mishkovski, I., & Mirchev, M. (2019). The geographic flow of music on spotify. *16th International Conference on Informatics and Information Technologies (CIIT).*

Jovanovska, L., & Panov, P. (2020). Toward improved semantic annotation of food and nutrition data. *SiKDD 2020 - Conference on Data Mining and Data Warehouses.*

Jovanovska, L., & Panov, P. (2021). Semantic representation of machine learning and data mining algorithms. *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 205–210.

Kant, I. (1908). Critique of pure reason. 1781. *Modern Classical Philosophers, Cambridge, MA: Houghton Mifflin*, 370–456.

Keet, C. M., Lawrynowicz, A., d'Amato, C., & Hilario, M. (2013). Modeling issues & choices in the data mining optimization ontology.

Keet, C. M., Ławrynowicz, A., d'Amato, C., Kalousis, A., Nguyen, P., Palma, R., Stevens, R., & Hilario, M. (2015). The data mining optimization ontology. *Journal of web semantics*, *32*, 43–53.

Keller, J. M., Gray, M. R., & Givens, J. A. (1985). A fuzzy k-nearest neighbor algorithm. *IEEE transactions on systems, man, and cybernetics*, (4), 580–585.

Lawrynowicz, A., Esteves, D., Panov, P., Soru, T., Dzeroski, S., & Vanschoren, J. (2017). An algorithm, implementation and execution ontology design pattern. *Adv. Ontol. Des. Patterns*, *32*, 55.

LeCun, Y., Bengio, Y. et al. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, *3361*(10), 1995.

Locke, J. (1847). *An essay concerning human understanding.* Kay & Troutman.

Louppe, G., & Geurts, P. (2012). Ensembles on random patches. *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 346–361.

Luan, Y., He, L., Ostendorf, M., & Hajishirzi, H. (2018). Multi-task identification of entities, relations, and coreferencefor scientific knowledge graph construction. *Proc. Conf. Empirical Methods Natural Language Process. (EMNLP).*

Markman, A. B. (2013). *Knowledge representation.* Psychology Press.

Material design [Accessed on 02.09.2022]. (2022). https://material.io/design

Material ui [Accessed on 02.09.2022]. (2022). https://mui.com/

Matthews, B. (2005). Semantic web technologies. *E-learning*, *6*(6), 8.

Mikolov, T., Karafiát, M., Burget, L., Cernock, J., & Khudanpur, S. (2010). Recurrent neural network based language model. *Interspeech*, *2*(3), 1045–1048.

Mitchell, T. M. et al. (1997). Machine learning.

Mungall, C. (2015). Relation ontology. https://obofoundry.org/ontology/ro.html

Musen, M. A. (2015). The protégé project: A look back and a look forward. *AI matters*, *1*(4), 4–12.

Nadeau, D., & Sekine, S. (2007). A survey of named entity recognition and classification. *Lingvisticae Investigationes*, *30*(1), 3–26.

Neo4j graph data platform [Accessed on 02.09.2022]. (2022). https://neo4j.com/

Neumann, M., King, D., Beltagy, I., & Ammar, W. (2019). ScispaCy: Fast and robust models for biomedical natural language processing. *Proceedings of the 18th BioNLP Workshop and Shared Task*, 319–327. https://doi.org/10.18653/v1/W19-5034

Owl — semantic web standards [Accessed on 02.09.2022]. (2012). https://www.w3.org/OWL/

Panov, P., Džeroski, S., & Soldatova, L. (2008). Ontodm: An ontology of data mining. *2008 IEEE International Conference on Data Mining Workshops*, 752–760.

Panov, P., Soldatova, L., & Džeroski, S. (2013). Ontodm-kdd: Ontology for representing the knowledge discovery process. *International Conference on Discovery Science*, 126–140.

Panov, P., Soldatova, L., & Džeroski, S. (2014). Ontology of core data mining entities. *Data Mining and Knowledge Discovery*, *28*(5), 1222–1265.

Panov, P., Soldatova, L. N., & Džeroski, S. (2016). Generic ontology of datatypes. *Information Sciences*, *329*, 900–920.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, *32*.

Pease, A., Niles, I., & Li, J. (2002). The suggested upper merged ontology: A large ontology for the semantic web and its applications. *Working notes of the AAAI-2002 workshop on ontologies and the semantic web*, *28*, 7–10.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, *12*, 2825–2830.

Prov-o: The prov ontology [Accessed on 02.06.2022]. (2013). https://www.w3.org/TR/prov-o/

Raad, J., & Cruz, C. (2015). A survey on ontology evaluation methods. *Proceedings of the International Conference on Knowledge Engineering and Ontology Development, part of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.*

Ramshaw, L. A., & Marcus, M. P. (1999). Text chunking using transformation-based learning. *Natural language processing using very large corpora* (pp. 157–176). Springer.

Rdf — semantic web standards. (2022). https://www.w3.org/RDF/

Rdfs — semantic web standards [Accessed on 02.09.2022]. (2004). https://www.w3.org/2001/sw/wiki/RDFS

React – a javascript library for building user interfaces. (2022). https://reactjs.org/

Reboul, A. C. (2015). Why language really is not a communication system: A cognitive view of language evolution. *Frontiers in Psychology*, *6*, 1434.

Rogers Jr, H. (1987). *Theory of recursive functions and effective computability.* MIT press.

Rosse, C., & Mejino Jr, J. L. (2003). A reference ontology for biomedical informatics: The foundational model of anatomy. *Journal of biomedical informatics*, *36*(6), 478–500.

Russell, S. J. (2010). *Artificial intelligence a modern approach.* Pearson Education, Inc.

Sayak. (2021). Arxiv paper abstracts. https://www.kaggle.com/datasets/spsayakpaul/arxiv-paper-abstracts

Shacl — semantic web standards [Accessed on 02.09.2022]. (2017). https://www.w3.org/2001/sw/wiki/SHACL

Sipser, M. (1996). Introduction to the theory of computation. *ACM Sigact News*, *27*(1), 27–29.

Smith, B., Ashburner, M., Rosse, C., Bard, J., Bug, W., Ceusters, W., Goldberg, L. J., Eilbeck, K., Ireland, A., Mungall, C. J., et al. (2007). The obo foundry: Coordinated evolution of ontologies to support biomedical data integration. *Nature biotechnology*, *25*(11), 1251–1255.

Srinivasa-Desikan, B. (2018). *Natural language processing and computational linguistics: A practical guide to text analysis with python, gensim, spacy, and keras.* Packt Publishing Ltd.

Stone, H. S. (1971). *Introduction to computer organization and data structures.* McGraw-Hill, Inc.

Sun, S., Cao, Z., Zhu, H., & Zhao, J. (2019). A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, *50*(8), 3668–3681.

Vanschoren, J., & Soldatova, L. (2010). Exposé: An ontology for data mining experiments. *International workshop on third generation data mining: Towards service-oriented knowledge discovery (SoKD-2010)*, 31–46.

Vanschoren, J., Van Rijn, J. N., Bischl, B., & Torgo, L. (2014). Openml: Networked science in machine learning. *ACM SIGKDD Explorations Newsletter*, *15*(2), 49–60.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L. B., Bourne, P. E., et al. (2016). The fair guiding principles for scientific data management and stewardship. *Scientific data*, *3*.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, *5*(2), 241–259.

Yang, L., & Shami, A. (2020). On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, *415*, 295–316.

Zhang, D., Maslej, N., & Brynjolfsson, E. (2022). https://aiindex.stanford.edu/report/

Zhang, H. (2004). The optimality of naive bayes. *Aa*, *1*(2), 3.

# Bibliography

## Publications Related to the Thesis

### References

Jovanovska, L., & Panov, P. (2020). Toward improved semantic annotation of food and nutrition data. *SiKDD 2020 - Conference on Data Mining and Data Warehouses.*

Jovanovska, L., & Panov, P. (2021). Semantic representation of machine learning and data mining algorithms. *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 205–210.

## Other Publications

### Conference Paper

Jovanovska, L. (2020). Semantic analysis of lyrics for genre labeling and improving music recommendation. *International Symposium Multilinguality in Folklore*, *1*(1), 52.

Jovanovska, L., Dimovski, A., & Joksimoski, B. (2018). Interactive digital environment for learning historical events. *15th International Conference on Informatics and Information Technologies (CIIT).*

Jovanovska, L., & Evkoski, B. (2020). Analysis of chord progression networks.

Jovanovska, L., Evkoski, B., Mirchev, M., & Mishkovski, I. (2020). Demographic analysis of music preferences in streaming service networks. *Complex networks xi* (pp. 233–242). Springer.

Jovanovska, L., Mishkovski, I., & Mirchev, M. (2019). The geographic flow of music on spotify. *16th International Conference on Informatics and Information Technologies (CIIT).*

# Biography

Lidija Jovanovska was born on June 18, 1997, in Skopje, Macedonia, where she completed her primary and secondary education. In September 2015, she enrolled at the Faculty of Computer Science and Engineering (FCSE), Ss. Cyril and Methodius University in Skopje.

During her studies, she participated in a variety of extracurricular activities. In early 2017, she worked as a 3D artist for the Museum of Macedonian Struggle, creating an interactive digital replica of one of the museum rooms. In the summer of 2017, she interned at Neotel, where she worked on designing 3D printable drone parts and assembling them into a working drone. The following year, Lidija was a Laboratory Assistant for the Computer Animation course and she also led an introductory workshop for high school students to create their first computer animation project. In September 2019, she graduated with a Bachelor's degree in Computer Science and Engineering. During her studies, she published two papers at International Conference on Informatics and Information Technologies (CIIT) organized by FCSE. She won the 3rd prize in the best student paper competition for her paper "The Geographic Flow of Music on Spotify" at CIIT 2019.

After graduation, Lidija enrolled in the MSc program "Information and Communication Technologies" at the Jožef Stefan International Postgraduate School in Ljubljana, Slovenia, under the supervision of Asst. Prof. Dr. Panče Panov. Her current work is on the topic of knowledge representation and semantic technologies in the field of data mining and machine learning. She is also an avid researcher in the music information retrieval community. In that regard, she took part in the HAMR hackathon at ISMIR 2020, where she won the prize for best documentation for her work on chord progression networks. Her research is published and presented at several conferences in the areas of data mining, network science and music information retrieval.