FACULTY OF ENGINEERING AND ARCHITECTURE

Automatic semantic annotation and visualization of machine learning pipelines with FnO

Pol Nachtergaele
Student number: 01808572

Supervisor: Prof. dr. Femke Ongenae
Counsellors: Dr. Stijn Verstichel, Kushagra Singh Bisen, Tom Windels

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Information Engineering Technology

Academic year 2023-2024

GHENT UNIVERSITY

Acknowledgements

Academic Support and Guidance

I extend my deepest gratitude to my supervisor, Prof. Dr. Femke Ongenae, for her invaluable guidance and support while allowing me the freedom to grow and choose my own path. I am also deeply thankful to my co-supervisors, Dr. Stijn Verstichel, Tom Windels, and Kushagra Singh Bisen, for their insightful revisions and for steering me in the right direction when necessary.

A special thanks to Dr. Ir. Ben De Meester for providing crucial insights into the Function Ontology. Your guidance was instrumental in helping me grasp this concept and tackle the challenges of this thesis.

Finally, I extend my heartfelt gratitude to all those who have supported me in my academic endeavors and consistently believed in my abilities. Your encouragement has been truly invaluable. I am especially grateful to my parents for their unwavering support and their assistance in financing my studies.

Use of Large Language Models

In the process of writing this thesis, I used large language models such as ChatGPT and Gemini to check spelling and structure. Suggestions made by these models were incorporated to enhance the writing style. These tools were used solely for structural aid, and no insights or sources were gathered through them. I found Gemini particularly useful, as it provides clearer insights into why certain rephrasing is suggested compared to ChatGPT.

Clarification Regarding the Master Thesis

This master's dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master's dissertation are not included in this text.

Abstract

This paper presents a novel tool for automatically generating RDF descriptions of machine learning pipelines to facilitate their use on the semantic web, promoting transparency and compliance with GDPR legislation. The tool leverages the Function Ontology (FnO) to semantically describe Python functions, methods, and classes, and integrates the Provenance Ontology (ProvO) to capture data transformations performed during processing. By describing function implementations users are able to execute and replicate these pipelines while capturing detailed provenance information. Additionally, MLFlow and the MLSchema Ontology are utilized to describe machine learning models and their training processes in RDF. These documents are visualized using a flowchart, enhancing accessibility and user interaction. By semantically describing machine learning operations, the tool allows individuals to retain control over their data.

Automatic semantic annotation and visualization of machine learning pipelines with FnO

Pol Nachtergaele

Supervisors: Prof. Dr. Femke Ongenae, Dr. Stijn Verstichel, Kushagra Singh Bisen, Tom Windels

***Abstract*— This paper presents a novel tool for automatically generating RDF descriptions of machine learning pipelines to facilitate their use on the Semantic Web, promoting transparency and compliance with GDPR legislation. The tool leverages the Function Ontology (FnO) to semantically describe Python functions, methods, and classes, and integrates the Provenance Ontology (ProvO) to capture data transformations performed during processing. By describing function implementations users are able to execute and replicate these pipelines while capturing detailed provenance information. Additionally, MLFlow and the MLSchema Ontology are utilized to describe machine learning models and their training processes in RDF. These documents are visualized using a flowchart, enhancing accessibility and user interaction. This approach aims to ensure that machine learning operations within decentralized Solid Pods maintain high transparency, allowing individuals to retain control over their data. Future extensions may include more advanced NLP methods for enhanced function matching and integration with defined taxonomies for model descriptions.**

***Keywords*— machine learning, data processing, Solid, Semantic Annotation, RDF, Provenance, Function Ontology, Provenance Ontology, MLSchema**

I. Introduction

The Semantic Web represents an evolution of the current web, aiming to create a universal medium for data, information, and knowledge exchange. At its core, it involves semantically describing instances using the Resource Description Framework (RDF), a standard model for data interchange. RDF enables the encoding of information in a machine-readable format, allowing computers to understand and process the semantics, or meanings, of data. By linking data across different sources, the Semantic Web enhances the FAIR principles—Findability, Accessibility, Interoperability, and Reusability—facilitating more efficient data sharing and integration, and enabling smarter applications that can leverage this interconnected web of data.

Modern applications leverage machine learning to enhance user experience. A challenge lies in continuing to utilize these powerful machine learning technologies while ensuring transparency, allowing individuals to have constant insight into the handling of their data. This paper focuses on the automatic description of machine learning operations using RDF, enabling these projects to be shared on the Semantic Web. By ensuring maximum transparency and compliance with GDPR legislation [1], this approach facilitates better data integration and accessibility while adhering to important privacy regulations.

P. Nachtergaele is a master student in Information Engineering Technology at Ghent University (UGent), Gent, Belgium. E-mail: Pol.Nachtergaele@UGent.be .

To tackle this problem a tool is developed that automatically generates RDF documents describing a complete machine learning pipeline, from data preprocessing through to model predictions, using semantic ontologies. This aims to facilitate the replication of the machine learning pipeline, enabling users to safely deploy and execute it, all while generating provenance information capturing all data trasnformations.

II. Problem

The General Data Protection Regulation (GDPR) is a comprehensive data protection framework that originated in the European Union (EU) and was adopted on April 14, 2016 [1]. The regulation introduces robust privacy rights for data subjects, imposes obligations on organizations processing personal data, and outlines mechanisms for enforcing compliance. Applications that use machine learning models trained on personal data must adhere to these laws. According to the right related to **automatic decision-making (Article 22)** and the right to **object (Article 21)** individuals may choose not to be subject to decisions based solely on automated processing. Nowadays, models used in machine learning pipelines are often complex and act as complete black boxes, making it challenging to assess the reliability of the decision making process. Understanding the data preprocessing and knowing which models are used is a first step to enable data owners to evaluate this reliability.

In the context of machine learning, a preprocessing pipeline transforms raw input data into meaningful feature data for use by a machine learning model. This pipeline consists of an ordered series of data transformations where each transformation's output serves as the input for the next, thus creating a continuous data flow that incrementally refines raw input. These transformations are often carried out by functions. To fully describe a pipeline, it is essential to capture and transform all used functions, their effects on data, and their interactions into RDF descriptions using semantic ontologies. To enable users to replicate the described pipelines, the RDF document should include the necessary implementation details, allowing the transformation of the description into an executable format.

The reliability of a model largely depends on the training process. Key questions to consider include: Which data was used for training? What hyperparameters were set? Who created the model, and for what purpose? Access to this information is crucial for anyone employing a model, as it allows for better understanding, evaluation, and trust in the model's performance and suitability for specific tasks.

III. Motivation

Providing semantic descriptions of machine learning pipelines facilitates their use on the semantic web. By leveraging semantic web technologies, users can ensure that their work is easily accessible, interpretable, and reusable by others in the community, promoting collaboration and accelerating innovation in the field of machine learning.

Furthermore, this approach represents a crucial first step in enabling machine learning within a Solid environment. By incorporating provenance information into the data preprocessing stages, an individual can make more informed decisions about whether or not to continue providing access to their data.

IV. Approach

This paper focuses on the Python programming language, which is popular for machine learning projects. By parsing code into an Abstract Syntax Tree (AST) [2], the underlying structure can be recursively visited, allowing for a step-by-step conversion into RDF. This hierarchical tree encapsulates data transformations and control flow structures as nodes. The following sections will explore how these AST nodes contribute to the generation of the RDF document.

A. Semantically Describing Functions

Before handling nodes, a correct semantic description of a Python function must be established. For this purpose, the Function Ontology (FnO) [3][4] can be employed. FnO allows developers to describe functions, as shown in Figure 1, regardless of the implementation environment, and link these function descriptions to their implementations. However, the current specification does not support the description of Python implementations. Therefore, this paper extends the specification with three new RDF classes for Python functions, methods and classes. By providing the name, module, and package, the corresponding Python objects can be retrieved from the implementation description. It is important to note that this paper assumes the necessary Python environment is already correctly set up by the user through the installation of required libraries. FnO Functions and FnO Implementations are connected through FnO Mappings, which map the implementation parameters to the corresponding parameters defined in the function description.

The developed tool allows users to provide a link to RDF documents containing FnO function descriptions. These documents are then queried to find possible matches between Python functions and their descriptions. For a match to occur, both the function and the description must have the same name. This approach is basic and does not account for lexical differences between functions with the same functionality. While the matching process could be enhanced with NLP to calculate similarity, this is beyond the scope of this paper.

When no description is found, the Python function object is used to generate a new description. By utilizing the Python function signature, all parameter names and optional type annotations are captured, allowing for seamless conversion into an FnO description. The Python object is also used to capture all metadata concerning the implementation by extracting the function name and the module in which it is defined.

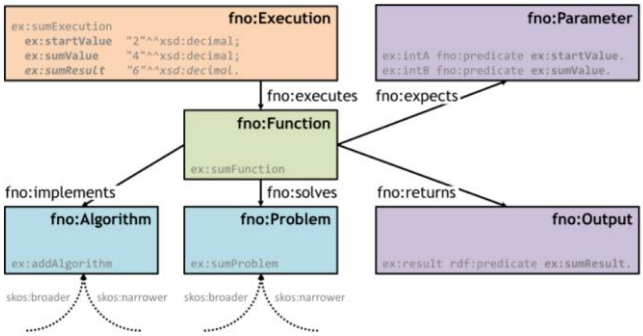

Figure 1: Function Ontology

B. Semantically Describing Pipelines

Merely identifying data transformations and describing them in FnO does not provide a complete understanding of a pipeline. It is crucial to recognize how these transformations are interconnected, specifically how inputs and outputs from each step are linked, illustrating the flow of data through the pipeline. To describe this flow comprehensively the FnO Composition Vocabulary can be employed. Function composition, in this, context, refers to the application of one function result to the to the input of another, effectively merging multiple functions into a single cohesive function. Thus, a pipeline can be conceptualized as a single function composed of all its constituent data transformations.

An FnO Composition consists of multiple mappings, each defining where to map from and where to map to by specifying the function and its corresponding parameter. Parameters that act as containers, such as lists, can hold a variable number of elements. The current FnO specification treats these parameters as a whole, thereby losing any internal structural information. To address this shortcoming, this paper introduces mapping strategies that allow mapping to a specific element within a container by specifying an index or a property.

To allow functions to be used multiple times in a composition, a distinction is made between a function and a function call. A function call is linked to a specific function and has a unique identifier, ensuring it can be referenced unambiguously within the pipeline.

C. Interpreting AST Nodes as Data Transformations

An AST (Abstract Syntax Tree) node in Python represents a single element of the source code structure, such as an expression, statement, or function definition, allowing the hierarchical organization of code components for analysis and transformation. Each node can be conceptualized as a data transformation that uses a specific function. This transformation yields an output, which can subsequently be utilized by other nodes in the data flow. As outlined in B, establishing a mapping needs both the function and the parameter/output it generates. Therefore, the handling of each node should entail returning an output or parameter along with its contextual information, such as the function call it belongs to. When the corresponding function object is derived from the AST node, the system generates a description as outlined in A to be used in the composition. The system will attempt to describe this function as a pipeline if possible. By providing a composition for the used function, users can choose to either

use the high-level function description or delve into the function body to expose more internal details. This flexibility provides multiple levels of detail when capturing provenance information when executing a pipeline.

D. Visualizing Described Pipelines

To make the complex descriptions more accessible to users, a visualization tool was developed with the PyQt library [7]. The Flowchart class from pyqtgraph [6] serves as a solid foundation for displaying the data flow described in FnO compositions. This class was modified and extended to enable interaction with RDF and to facilitate easy conversion from an RDF document. Figure 1 shows a visualized description of a pipeline that processes tweets to train a sentiment prediction model implemented with TensorFlow. Each data transformation is depicted as a node, along with its corresponding parameters and outputs, while the data flow is illustrated by the blue lines.

Figure 2: Visualization of a training pipeline

Function nodes that have a corresponding composition can be expanded to reveal the internal data flow using the interface shown in Figure 3. This interface displays all the functions used within the pipelines and allows the user to decide whether to include each function in the provenance generation process.

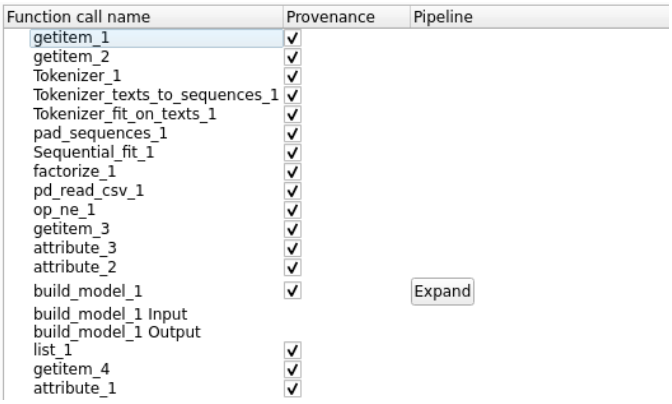

Figure 3: Interface of used functions

E. Generating Provenance for Described Pipelines

When creating a flowchart from a generated RDF document, the implementations are used to import the correct function objects and link them to the corresponding nodes. By providing input to the flowchart, it will sequentially process nodes and propagate the results, allowing full execution of the pipeline. Inputs with simple types, such as integers, strings, or floats, can be entered directly through an input field. However, when the input requires more advanced Python objects, such as a pandas DataFrame, they must be provided as a pickle file. The pickle library allows Python objects to be stored as binary files.

When processing a node, the system generates provenance information using the function description and the generated values. The Provenance Ontology (ProvO) [8] is used to describe the lifecycle of a data object in RDF format and can be integrated with FnO [9] to represent a data transformation, as shown in Figure 4. In this context, a data transformation is identified as a provenance activity that uses raw input data and a function, both categorized as entities. This activity generates an output, which is also recognized as an entity. By combining the metadata generated from each node, a comprehensive provenance document of the pipeline, including all intermediate data instances, can be created.

A Python instance can be very complex, making direct conversion into an RDF format challenging. To address this, the string representation of the instance is provided as a Literal, with the datatype, which is a Python class, provided as an FnO implementation. While this approach results in a significant loss of information, providing a full conversion from instances to RDF would require a separate, dedicated paper.

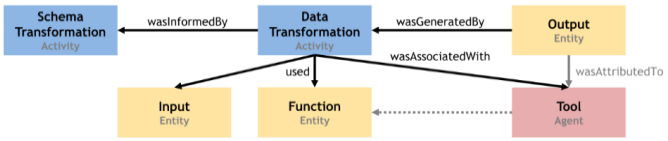

Figure 4: Integration of ProvO with FnO

MLFlow [10] is a tool that automatically captures metadata during the training of machine learning models, including the generated model and the hyperparameters set. Using the MLSchema Ontology [11], as shown in Figure 5, the training process can be semantically described in RDF. Python supports machine learning through various libraries such as TensorFlow or scikit-learn. An implementation specifies the algorithm used within such a library, while an Algorithm is a framework-independent description. These descriptions are derived from the model instance captured by MLFlow.

The challenge of representing model instances in RDF arises here as well. Currently, the system supports a basic description of implementations for models described in TensorFlow and Python. The system could be expanded to obtain these descriptions from defined taxonomies, but this would require NLP methods to be effective. Any metrics used to evaluate the model, such as accuracy, are defined using ModelEvaluation.

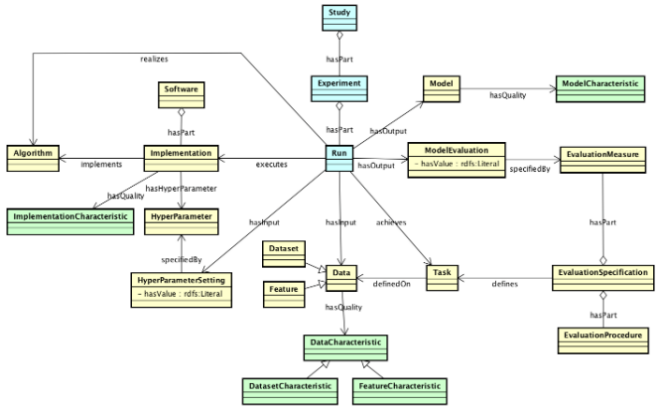

Figure 5: The MLSchema Ontology

V. Results

A. Discovered Limitations and Breaking Conditions

This paper presents a proof of concept and acknowledges its limitations in addressing all potential scenarios. Conditional statements, in particular, exert an unforeseen influence on data flow within a pipeline. The flow of data can bifurcate and subsequently reunite, possibly leading to the overwrite of each other's results. An instance of this occurrence arises with early stopping due to return statements within conditional statements. Certain code structures introduce cycles in node dependencies, rendering it impossible to ascertain a correct processing order.

When configuring a pipeline based on a pre-existing RDF graph, the tool encounters difficulty converting certain simple literals, like tuples, from their RDF representation into a Python instance. However, it has been verified that when the graph is generated from Python code and subsequently employed to establish a pipeline, the same RDF literal successfully yields a Python instance. Despite efforts, no documentation regarding RDF literals has been uncovered that might elucidate this behavior. This issue significantly impedes the utility of the generated description, as enabling other users to execute a pipeline transparently becomes unfeasible for certain projects. Given the complexity of the problem, it is highly probable that there exist other scenarios where the system exhibits unexpected behavior.

B. Performance

The tool was tested on 7 machine learning projects to asses the generated graphs and measure the effect the tool has on execution time. Table 1 shows measurement pertaining to generation time. It is immediately evident that generating the flowchart requires more time. This most likely due to the slow SPARQL querying. More generated triples equate to higher generation time for both the description graph and the flowchart. Figure 6 plots demonstrates a linear correlation between the number of generated nodes and the number of triples. This suggests that an increase in triples leads to a proportional increase in the time required to generate a flowchart. Nonetheless, there is a noticeable variance, indicating that factors beyond the number of triples influence generation time. This variance likely stems from the complexity of the code being described.

Table 1: Generation time

Name	Graph gen. time (ms)	# Triples	Chart gen. time (ms)	# Nodes
Sentiment Analysis [12]	1277	1783	7111	37
Digit Recognition [13]	2828	3402	12129	59
Iris Flower Classification [14]	1193	1856	3728	21
Recommend TedTalks [15]	1497	2038	5177	28
IPL Score Prediction [16]	2952	3163	11457	55
Music Genre Clustering [17]	1462	1927	5003	28
Stock Price Prediction [18]	2728	3272	8605	40

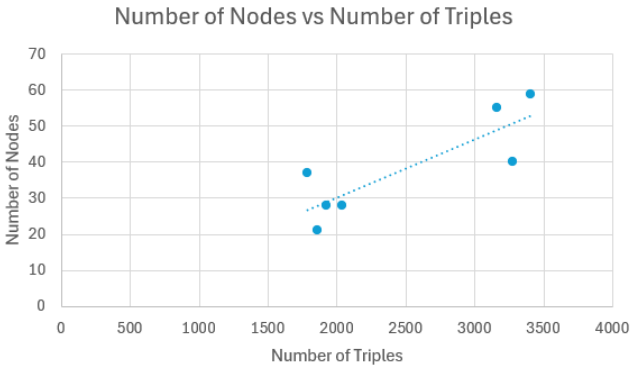

Figure 6: Numer of nodes vs Number of triples

The *stock price prediction* project is excluded from the following performance test due to ongoing errors encountered during execution via the tool. Table 2 outlines the performance impact observed when executing a pipeline through a flowchart. The number of open nodes represents the amount of data transformations that are captured to provide provenance information. Performance impact is quantified as the ratio of the chart execution time to the default execution time. Notably pipelines executing swiftly experience a more significant impact. Figure 7 shows that the overhead introduced by the flowchart becomes obsolete as execution time increases.

Table 2: Performance impact

Name	Chart exe. Time (ms)	Default exe. Time (ms)	Perf. impact	Time diff (ms)	# open nodes
Sentiment Analysis	524241	483198	1,08	41042	19
Digit Recognition	16873	5532	3,05	11342	59
Iris Flower Classification	4183	4	1006,35	4179	21
Recommend TedTalks	3304	533	6,20	2771	28
IPL Score Prediction	79912	49398	1,62	30514	55
Music Genre Clustering	4249	45	94,53	4204	28

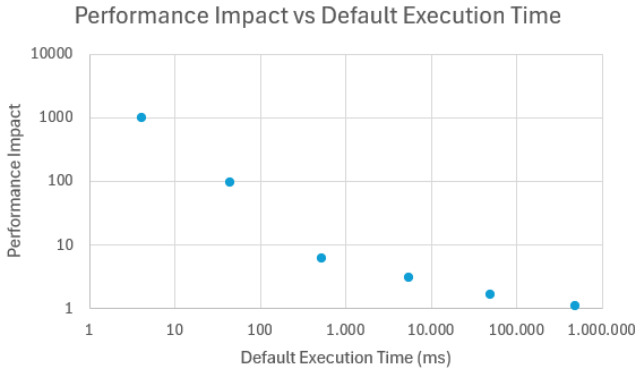

Figure 7: Performance impact vs Default execution time

Figure 8 plots the actual time difference against the number of open nodes. Surprisingly, no correlation between these two variables is observed. Figure 9 illustrates the time difference relative to the default execution time, clearly showing a constant overhead for pipelines that execute quickly. Beyond a certain point, the overhead increases linearly with the

default execution time. This constant overhead can be attributed to the systems that must be set up before executing the flowchart. Other factors, such as determining process order and provenance generation, were anticipated to introduce latency proportional to the number of nodes, but this was not observed. No explanation for this behaviour was identified.

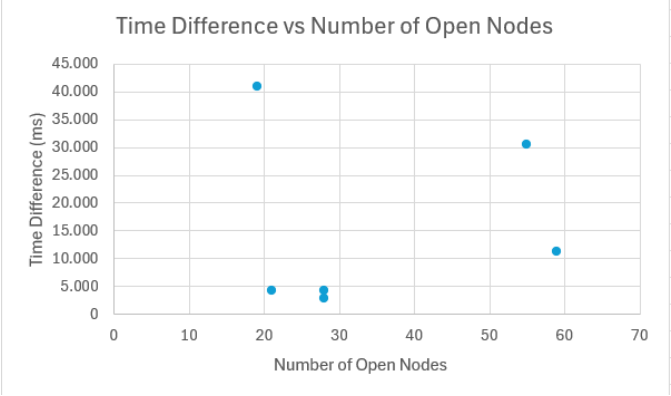

Figure 8: Time difference vs Number of open nodes

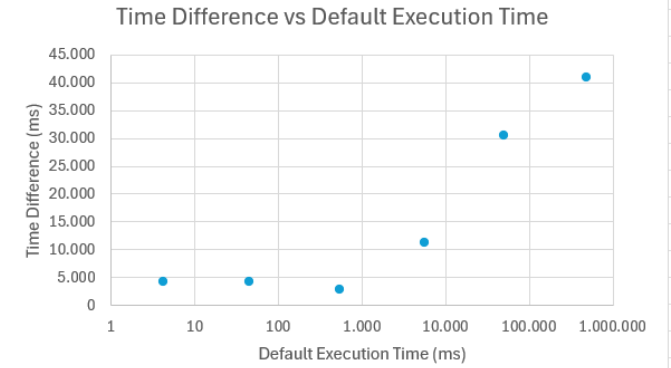

Figure 9: Time difference vs Default execution time

VI. Conclusion

The tool successfully describes Python pipelines using FnO to semantically describe every data transformation present, allowing the pipeline to be executed while generating a provenance document combining FnO, ProvO and MLSchema. This provides a broad understanding of what happens with data inside a pipeline, enabling users to capture training of a model and share the generated provenance document alongside the trained model. Others can then acquire more insights about the training process and make a more informed decision on whether to trust the model or not. To make these graphs more accessible, the tool can be employed to visualize such provenance graphs, exposing the internal structure of the described pipeline.

However, when dataflow gets more complex due to conditional statements or other control flow structures, the system is prone to failing. The handling of conditional statements could be further improved to be more robust. There are still code structures that are not parsed into FnO descriptions, such as while-loops, try-except clauses and with statements. When these are present, the dataflow will not be able to be described using the tool. In other words, this tool is still in development and must be used with care to avoid potential errors.

A significant limitation is the fact that Python Instances cannot always be converted into an RDF representation that allows conversion back into a Python instance. Future research should focus on making this possible. This would be a crucial step forward to allow these descriptions to be used to safely set up machine learning pipelines while keeping track of transformations applied on their data.

References

[1] European Parliament and Council of the European Union, “Regulation (EU) 2016/679 of the European Parliament and of the Council.” [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj

[2] Priya C., Bala, “Abstract Syntax Tree (AST) - Explained in Plain English”, 2022, [Online]. Available: Abstract Syntax Tree (AST) - Explained in Plain English - DEV Community

[3] B. De Meester, A. Dimou, and F. Kleedorfer, “The function ontology,” 2023. [Online]. Available: https://fno.io/spec/

[4] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, “Implementation-independent function reuse,” Future Generation Computer Systems, vol. 110, pp. 946–959, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19303723

[5] B. De Meester and F. Kleedorfer, “The Function Ontology – Composition vocabulary”, 2021. [Online]. Available: The Function Ontology - Composition vocabulary (fno.io)

[6] Pyqtgraph - scientific graphics and gui library for python,” 2021. [Online]. Available: https://www.pyqtgraph.org/39

[7] “Pyqt6” 2024. [Online]. Available: https://www.riverbankcomputing.com/software/pyqt/

[8] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, “Prov-o: The prov ontology,” April 30, 2013 2013. [Online]. Available: https://www.w3.org/TR/prov-o/

[9] B. De Meester, A. Dimou, R. Verborgh, and E. Mannens, “Detailed provenance capture of data processing,” Proceedings of the First Workshop on Enabling Open Semantic Science (SemSci), vol. 1931, pp. 31–38, 2017. [Online]. Available: http://ceur-ws.org/Vol-1931/#paper-05

[10] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. A. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, “Developments in mlflow: A system to accelerate the machine learning lifecycle,” in Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning, ser. DEEM’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399579.3399867

[11] D. Esteves, A. Ławrynowicz, P. Panov, L. Soldatova, T. Soru, and J. Vanschoren, “Ml schema core specification,” Oct. 2016. [Online]. Available: http://mlschema.github.io/documentation/ML%20Schema.html#overview

[12] Sentiment analysis of tweets using an lstm.” [Online]. Available: https://techvidvan.com/tutorials/python-sentiment-analysis/11

[13] Handwritten digit recognition.” [Online]. Available: https://data-flair.training/blogs/python-deep-learning-project-handwritten-digit-recognition/

[14] Iris flower classification.” [Online]. Available: https://dataflair.training/blogs/iris-flower-classification/

[15] “Tedtalk reccomendation.” [Online]. Available: https://www.geeksforgeeks.org/ted-talks-recommendation-system-with-machine-learning/

[16] Ipl score prediction using deep learning.” [Online]. Available: https://www.geeksforgeeks.org/ipl-score-prediction-using-deep-learning/?ref=rp

[17] Clustering music genres using python.” [Online]. Available: https://thecleverprogrammer.com/2022/04/05/clustering-music-genres-with-machine-learning/#google_vignette

[18] Stock price prediction with lstm.” [Online]. Available: https://thecleverprogrammer.com/2022/01/03/stock-price-prediction-with-lstm/

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence.

EDA Exploratory Data Analysis.

FnO Function Ontology.

FnOC FnO Composition Vocabulary.

FnOI FnO Implementation Vocabulary.

FnOM FnO Mapping Vocabulary.

GDPR General Data Protection Regulation.

ML Machine Learning.

NLP Natural Language Processing.

PROV-O PROV-O Ontology.

RDF Resource Description Framework.

SPARQL SPARQL Protocol and RDF Query Language.

URI Uniform Resource Identifier.

List of Algorithms

List of code Fragments

1

Introduction

The vast amount of information available online presents challenges in data discovery, integration, and interpretation. The Semantic Web represents an evolution of the current web, aiming to create a universal medium for data, information, and knowledge exchange. At its core, it involves semantically describing instances using the Resource Description Framework, a standard model for data interchange. RDF enables the encoding of information in a machine-readable format, allowing computers to understand and process the semantics, or meanings, of data. By linking data across different sources, the Semantic Web enhances the FAIR principles—Findability, Accessibility, Interoperability, and Reusability—facilitating more efficient data sharing and integration, and enabling smarter applications that can leverage this interconnected web of data [1]. This interconnected web of data has the potential to address a key challenge in modern machine Learning (ML) applications: the lack of transparency in complex models.

Modern applications leverage ML to enhance user experience, providing deep insights into user behavior based on sensor measurements and personalized content recommendations. A challenge lies in continuing to utilize these powerful ML technologies while ensuring transparency, allowing individuals to have constant insight into the handling of their data. ML models used in ML pipelines are often complex and act as complete black boxes, making it challenging to assess the reliability of the decisions making process. Understanding how data is transformed and which models are used is a first step to enable data owners to evaluate this reliability. This requires complete provenance information, i.e., metadata describing the transformations applied, when it was modified, etc., throughout the entire ML process.

Semantic annotation of ML pipelines using RDF can facilitate the discoverability and distribution of these projects on the Semantic Web. This approach enables users to efficiently identify projects aligned with their specific needs. Furthermore, by providing provenance information regarding the training process and meaningful insights into the preprocessing steps, users are more likely to trust and grant access to their personal data. Ultimately, this strategy fosters improved collaboration in a trust-based environment.

1 Introduction

The primary objective of this master's thesis is to develop a system that automatically generates RDF documents describing a complete ML pipeline, from data preprocessing through to model predictions, using semantic ontologies. This documentation aims to facilitate the replication of the ML pipeline, enabling users to effectively execute the pipeline. The generated RDF documents are crucial as they allow users to safely deploy the ML pipeline and generate provenance metadata for any given input based on the documentation provided. Furthermore, the system will visually present this information, enhancing accessibility for users.

chapter 2 delves into related work, providing a foundation for the subsequent chapters. Following this, chapter 3 presents an overview of the problem, decomposing it into manageable parts and introducing the proposed solution. This chapter will also lay out the structure and content of the remaining chapters, providing a roadmap for the rest of the thesis.

2

Related work

2.1 The Semantic Web

The Semantic Web and Linked Data represent groundbreaking approaches to structuring and understanding data on the internet. At their core, these concepts rely on technologies like RDF, ontologies, and protocols to provide data with semantics, creating a web of interconnected information.

2.1.1 Describing Ontologies with RDF

RDF is a cornerstone of the Semantic Web. At its essence, RDF provides a standardized way to express relationships between resources on the web. It employs triples, comprised of subject-predicate-object statements, to create a graph-like structure that conveys not only the data but also the relationships and context within which it exists. RDF enables the creation of machine-readable data structures, forming the basis for a more semantically rich web.

Ontologies play a pivotal role in the Semantic Web by establishing a shared understanding of specific domains. An ontology defines a standardized vocabulary and a set of relationships within a particular knowledge domain. This enables the classification and/or explanation of different entities. By adhering to ontologies, disparate systems and applications can interpret and exchange data with a consistent understanding. Ontologies can be implemented using RDF which makes this knowledge domain available on the semantic web.

2.1.2 Linked Data and SPARQL

Linked Data is a set of best practices and principles for publishing and connecting structured data on the web [2]. Following these principles, data is not merely isolated within silos but is interlinked to other related data points across the web. This is achieved through the use of Uniform Resource Identifiers (URIs) and RDF triples, creating a network of interconnected information. Linked Data principles enable applications to traverse this web of data, discovering and integrating information from various sources.

SPARQL (SPARQL Protocol and RDF Query Language) serves as the query language for the Semantic Web [3]. It allows for the retrieval and manipulation of data stored in RDF format. SPARQL queries are capable of traversing and querying interconnected data. This querying capability is fundamental for applications to derive meaningful insights from the wealth of linked information on the Semantic Web.

2.2 Machine Learning

Machine Learning is a powerful field of artificial intelligence (AI) that empowers computers to learn patterns, make decisions, and improve their performance over time without explicit programming. This is done using huge amounts of data.

There are multiple forms of machine learning:

- **Supervised Learning**: The algorithm is trained on a labeled dataset, where each input is associated with a corresponding output. It learns to map inputs to outputs.
- **Unsupervised Learning**: The algorithm explores unlabeled data to identify patterns, relationships, or groupings without explicit guidance.
- **Reinforcement Learning**: The algorithm learns by interacting with an environment and receiving feedback in the form of rewards or penalties.

A ML model is a mathematical algorithm that undergoes training on extensive datasets to subsequently perform specific tasks, such as classification. The training data, essential for the model's proficiency, needs to be well-structured and comprise attributes known as features. The selection of these features is a critical decision as it significantly impacts the model's capabilities. Notably, various models exist, each characterized by a distinctive approach to the learning process, with some being more intricate than others. The full ML process is discussed further in 2.4.

2.3 General Data Protection Regulation

The General Data Protection Regulation (GDPR) is a comprehensive data protection framework that originated in the European Union (EU) and was adopted on April 14, 2016 [4]. The primary objective of the GDPR is to enhance and harmonize data protection laws across EU member states, providing individuals with greater control over their personal data. The regulation introduces robust privacy rights for data subjects, imposes stringent obligations on organizations processing personal data, and outlines mechanisms for enforcing compliance. GDPR principles emphasize transparency, fairness, and accountability in the handling of personal data.

Applications that use ML models trained on personal data must adhere to these laws. But before delving into how such applications can conform to the GDPR and respect the rights of the subjects, it's crucial to acknowledge the areas where current ML workflows often lack the needed transparency.

2.3.1 Lack of Transparency in Machine Learning Models

According to the right related to **automatic decision-making (Article 22)** and the right to **object (Article 21)**, individuals may choose not to be subject to decisions based solely on automated processing and must be able to prevent their data from being processed. As such, a user can decide which ML models may use his/her data or not. To let the user make an informed decision, it must be able to understand the model. Several questions one might ask are: How is my data used? What model is used and what is it doing? To enable the user to make an informed decision, the user must have metadata concerning the ML process.

In the context of data objects and ML, provenance (also known as data provenance or data lineage) refers to the record of the origins and lifecycle of data throughout its journey from creation to its final form and use. This includes details about how data is collected, transformed, and processed before it is used by a ML model for prediction. The PROV-O Ontology (PROV-O) is effectively utilized to describe the lifecycle of a data object in RDF format [5]. By having access to the provenance of their personal data that has been used by a ML model, an individual can make more informed decisions about whether to continue allowing the model access to their data. Figure 2.1 illustrates the three starting points for employing the ontology.

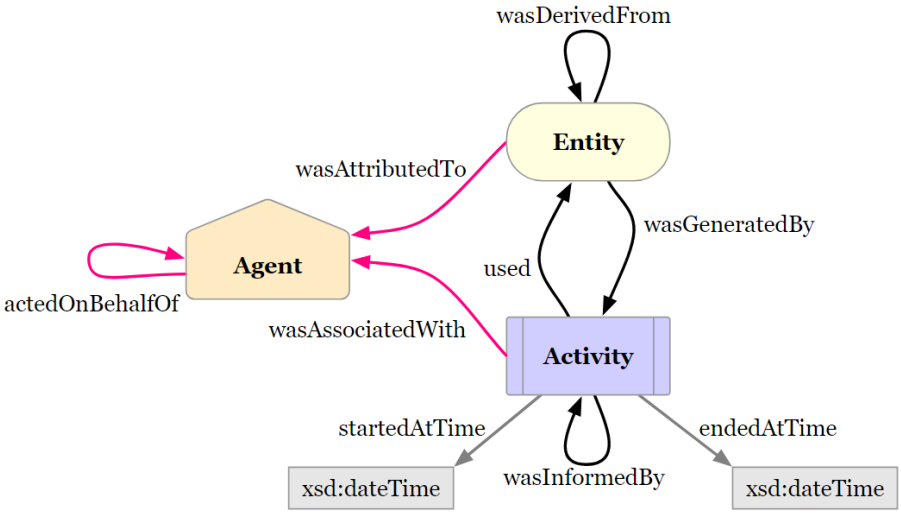

Figure 2.1: PROV-O starting points.

2.4 Semantically describing a machine learning process

As stated in 2.3.1, a user must be able to understand what is done with their data and make an informed decision about whether or not they trust the application using a ML model. These challenges can be overcome by describing the ML process semantically.

To describe the entire ML process, it is useful to break it down into successive pieces. Using [6, 7], the ML process can be divided into the following parts:

1. preprocessing
2. Model training
3. Evaluation
4. Prediction

In the upcoming sections, the examination focuses on how provenance data enhances transparency and reliability for each component of the model. Within this context, discussion centers on existing technologies and ontologies capable of semantically describing these components.

2.4.1 preprocessing and model training

Provenance data plays a crucial role in enhancing transparency and reliability for the preprocessing and model training stages of the ML process. This section explores how existing ontologies can be leveraged to semantically describe the transformations applied to the data during these stages. preprocessing encompasses the entire process of converting raw data into usable training data. The initial step involves conducting Exploratory Data Analysis (EDA) to collect statistical attributes of the data. Currently, no ontology has been identified to describe these statistical features, such as class distribution and mean value. Subsequently, the raw data undergoes adjustments for the creation and selection of features. These adjustments significantly influence the model and, as such, are vital for model evaluation.

For the semantic representation of these transformations, the Function Ontology (FnO) can be employed [8, 9]. FnO empowers developers to delineate functions, as shown in Figure 2.2, regardless of the implementation environment, thus promoting code reusability and compatibility. In [10], the combination of FnO with PROV-O facilitates the capture of schema and data transformations, encompassing the entire preprocessing process. Figure 2.3 shows how FnO and PROV-O can be integrated with each other.

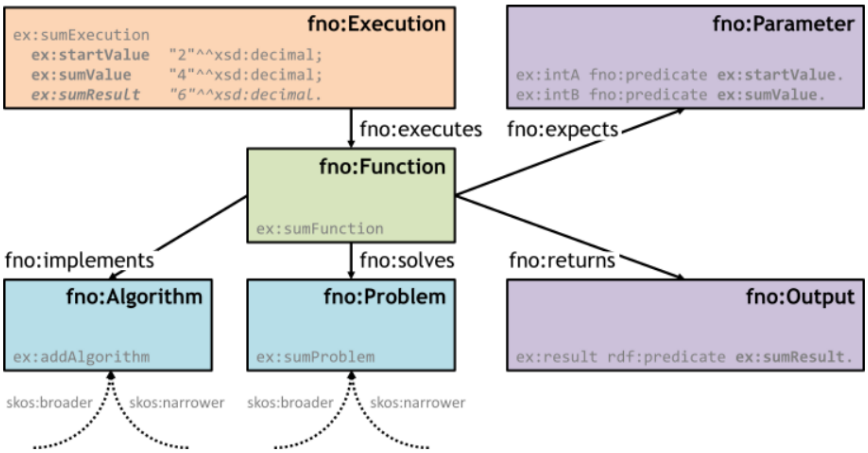

Figure 2.2: Graphic presentation of a FnO Function.

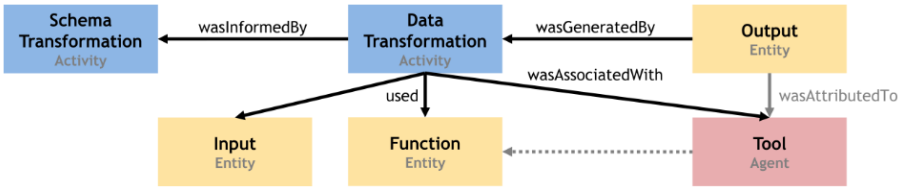

Figure 2.3: Combination of FnO and PROV-O

The MLSchema Ontology is a valuable resource in the realm of machine learning, designed to comprehensively represent various facets of the ML process [11]. Its primary aim is to facilitate the structured description of ML experiments, making it a powerful tool for both researchers and practitioners. It offers a holistic approach to ML experiment descriptions. This entails covering every phase of the process, from data preprocessing and model training to the evaluation of results. Experiments consist of multiple runs. Each run yields a specific ML model. The ontology allows for granular definitions of ML models, including the algorithms used, their implementations, and the settings of hyperparameters. This level of detail is crucial for replicability and model understanding.

2.4.2 Evaluation & Prediction

Model performance metrics, including sensitivity, precision, recall, F1 score, and others, serve as essential tools for evaluating the quality and reliability of ML models [12]. These metrics provide insights into distinct aspects of a model's performance. Various methods can be employed to test ML models, each with distinct attributes and implications for trustworthiness assessment. Common testing methods include Cross-Validation and Hold-Out Validation [12].

The choice of performance metrics and testing methods is pivotal in assessing a model's trustworthiness. It determines how well the model generalizes to new data, whether it prioritizes precision or recall, and how robust the model is to data variations. A careful selection of these elements is crucial in ensuring that the model is fit for its intended purpose.

Once a model is deemed trustworthy, it transitions from the training phase to the prediction phase. During prediction, the model is presented with new data for tasks like classification or making predictions. In essence, this prediction pipeline utilizes a pre-trained model rather than training a new one. Fortunately, the same methods employed for describing training pipelines, such as those discussed in previous sections, can be applied to describe prediction pipelines as well.

2.5 Automatically generating linked data from program code

In order to optimize the utilization of existing ontologies, automating the generation of linked data from program code is crucial. This not only reduces the barrier for creating a transparent model but also ensures consistent descriptions. In this section, various technologies are explored that can assist in this process. For this research only Python code is taken into consideration. This is a very popular coding language in the ML world.

2.5.1 Describing preprocessing

Capturing the preprocessing stage of the ML process can be challenging. Primarily, it involves data transformations, where the contents of a data sample undergo changes for purposes such as data cleaning, feature selection, or feature creation. These transformations vary widely, spanning from simple functions to more intricate processes. Creators of the model may define these transformations, or they can be extracted from an existing library. Moreover, the preprocessing stage is frequently conceptualized as a pipeline. This pipeline represents a sequence of transformations, where the raw data serves as the input and the output consists of refined feature data that is suitable for training a ML model. The pipeline organizes and streamlines the various steps involved in preparing the data for model training.

A data transformation, being a function, can be effectively described using FnO [8]. Fortunately, a Python function descriptor is already available [13]. This module leverages annotations within a Python function to capture information about input and output types. It utilizes a user-defined type map to convert these Python types into RDF. Furthermore, it outlines the mapping that facilitates linking the abstract function description to its implementation. While this module may have some limitations it serves as a foundational framework that can be extended to capture the complex preprocessing pipeline.

A limitation of this module is its focus on function descriptions without examining the internal structure. To fully understand the preprocessing steps, capturing the operations the function performs on sample data is crucial. Extracting a suitable RDF representation directly from the Python objects used in the preprocessing function could address this limitation and guarantee consistency across documents, eliminating the need for user-defined type maps. This approach could be facilitated by parsing the function body using libraries like `inspect` [14] and `ast` [15] for a more comprehensive description.

Describing the impact of a data transformation on a data sample is a crucial aspect, and the PROV-O is well-suited for this purpose. As illustrated earlier in Figure 2.3, the PROV-O ontology can be effectively integrated into the previously outlined data transformation pipeline. This becomes particularly valuable when capturing the execution of the pipeline on a given data sample. The resulting provenance information allows users to gain a comprehensive understanding of the transformations applied to their data.

2.5.2 Describing Model & Evaluation

MLflow, a tool designed to capture and organize details of ML workflows [16], offers valuable information for generating RDF descriptions. By parsing the directory structure created by MLflow, RDF triples can be generated using the MLSchema ontology. This approach enriches the provenance data captured with PROV-O by adding semantic meaning to the ML process stages, hyperparameters, and evaluation metrics. [17] is an example where MLflow is succesfully used to automatically generate provenance information. A Python API is provided as the library `mlflow`.

In conclusion, this chapter explored existing research on related topics that contribute to achieving transparency and understanding in Machine Learning processes. The Semantic Web with its core technologies like RDF and ontologies provides a foundation for semantically describing these processes. Ontologies such as FnO, PROV-O, and MLSchema offer vocabularies to capture details about data transformations, provenance, and model characteristics. Additionally, tools like MLflow can facilitate the automation of generating linked data from ML workflows. By leveraging these technologies, we can enhance the explainability and trustworthiness of ML models, empowering users to make informed decisions about their data and the models that utilize it.

3

Component Overview

This chapter provides an overview of the research conducted in this paper, which aimed to achieve three primary objectives: describing pipelines, visualizing pipelines, and capturing provenance during pipeline execution. Figure 3.1 illustrates a use case diagram that outlines the necessary functionalities to implement these objectives. To facilitate transparent pipeline descriptions (e.g., specifying inputs, outputs, and transformations), ML developers must be able to provide their Python code based on FnO. This approach enables developers to share their transparent descriptions with data owners, thereby increasing the likelihood of obtaining consent to access personal data. However, it's important to acknowledge that the current implementation is designed and tested on fairly straightforward pipelines. Some limitations regarding highly complex pipelines were discovered during evaluation (further discussed in chapter 8). These limitations are being addressed, but they currently restrict the tool's full functionality.

To facilitate transparent pipeline descriptions, ML developers must be able to provide their Python code based on FnO, which can then be used to replicate a fully executable pipeline. This approach enables developers to share their transparent descriptions with data owners, thereby increasing the likelihood of obtaining their consent to access personal data. By allowing data owners to set up and execute the pipeline in a secure environment while generating provenance, they can make more informed decisions about trusting the developed pipeline.

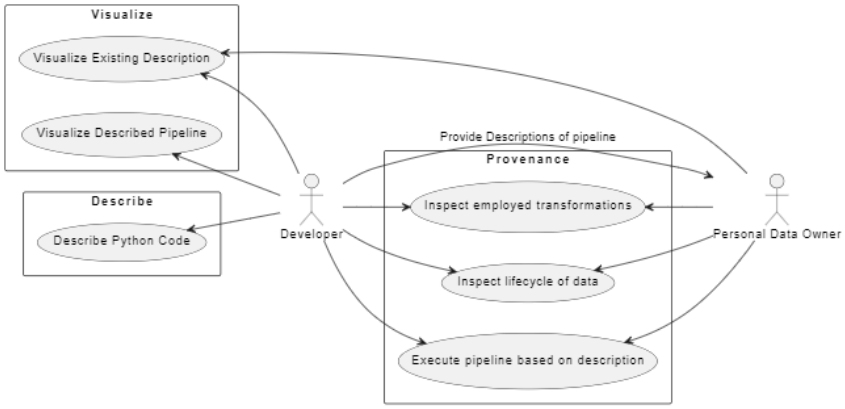

Figure 3.1: Usecase diagram

3 Component Overview

The class diagram provided in Appendix 9.2 gives an overview of the key classes implemented to accomplish the objectives of this work. Three main generator classes were developed:

1. `PipelineRDFGenerator`: Responsible for generating an RDF graph representation of a provided function, which is further explained in chapter 5.

2. `PipelineChartGenerator`: Uses the RDF graph generated by `PipelineRDFGenerator` and transforms it into an executable format, as discussed in section 6.2.

3. `ProvenanceGenerator`: Describes the execution process in RDF, providing transparent provenance information by combining ProvO and FnO, as discussed in section 6.3.

Additionally, the `FunctionRDFGenerator` class was introduced to transform function objects into FnO Function descriptions, which are then used by the `PipelineRDFGenerator` to build pipeline descriptions. The functionality of this class is elaborated upon in chapter 4, where it is discussed in detail. The sequence diagram in Appendix 9.2 visualizes the interactions between these generator classes and how they collaborate to achieve the overall functionality. All source code is implemented in the project repository [1].

Many code fragments throughout this document contain RDF descriptions expressed in the Turtle format [18]. To avoid cluttering, Table 3.1 provides an overview of the prefixes used within these RDF descriptions.

Table 3.1: Ontology Prefixes

Ontology Name	**Prefix**	**Namespace URL**
base		http://www.example.com#
RDF	rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns#
RDFS	rdfs	http://www.w3.org/2000/01/rdf-schema#
XSD	xsd	http://www.w3.org/2001/XMLSchema#
Function Ontology	fno	https://w3id.org/function/ontology#
FnO Implementation Vocabulary	fnoi	https://w3id.org/function/vocabulary/implementation#
FnO Mapping Vocabulary	fnom	https://w3id.org/function/vocabulary/mapping#
FnO Composition Vocabulary	fnoc	https://w3id.org/function/vocabulary/composition#
PROV-O Ontology	prov	http://www.w3.org/ns/prov#
Machine Learning Schema	mls	http://www.w3.org/ns/mls#
Python Functions	pf	http://www.example.com/pythonfunctions#
GREL Functions	grel	http://users.ugent.be/~bjdmeest/function/grel.ttl#
Dublin Core Terms	dcterms	http://purl.org/dc/terms/
DOAP	doap	http://usefulinc.com/ns/doap#

[1] https://gitlab.ilabt.imec.be/kbisen/thesis-solid-protego-pol

4

Semantically describing functions

In this chapter, the semantic description of Python functions using Fno will be explored. The initial focus is on introducing FnO and its role in establishing standardized function descriptions. Subsequently, the discussion will shift towards the use of existing descriptions and their linkage to correct implementations, a crucial step in enabling the replication of the ML pipeline. Finally, the chapter will detail the conversion process from Python function objects to RDF and vice versa.

4.1 Understanding the Structure of FnO Descriptions

4.1.1 Describing Functions

A function is defined by its name, along with any number of input parameters and outputs it may have. It's important to note that a description of a function using FnO does not reveal any details about the function's internal mechanics. These details are addressed through FnO Composition, which is discussed further in 5.

To facilitate the description of a function, the `FnoDescriptor` class was implemented. This class includes several helper functions to create a `PipelineGraph` that contains the function description. The `PipelineGraph` class inherits from `rdflib.Graph`. Its sole purpose is to provide functions that implement frequently used queries to ease programming.

Function

Code Fragment 4.1 presents a template for describing a function in FnO. It outlines the expected parameters and the outputs returned using RDF Lists, with the **expects** and **returns** properties respectively. Further details on parameters and outputs can be found in section 4.1.1.

```
:func a fno:Function ;
    fno:name "The function name"^^xsd:string
    fno:expects [
        a rdf:Seq ;
        rdf:_1 :funcParameter1 ;
        rdf:_2 :funcParameter2 ;
        ...
    ] ;
    fno:returns [
        a rdf:Seq ;
        rdf:_1 :funcOutput
    ] .
```

Code Fragment 4.1: FnO Function Description

Parameter & Output

In Code Fragment 4.2, a template is provided for defining a parameter using FnO. The **predicate** relationship is crucial for associating values with the parameter during a function's execution. This concept is further explained in section 5.2.2. Moreover, the linkage of a parameter to a function is facilitated by the **expects** predicate, as previously discussed. An output is defined similarly, with the primary distinction being that it is categorized as a FnO Output.

In Python, functions often specify default values for their parameters, allowing these arguments to be omitted when the function is called. In formal descriptions, this characteristic can be indicated using the predicate **required**. However, FnO currently lacks a mechanism to explicitly define the default values of these parameters. To address this limitation, the introduction of a new predicate **default** is proposed. This predicate will associate a literal value with a parameter. It is important to note that **default** should only be applied to parameters that are not mandatory, as indicated by **required**.

```
:par a fno:Parameter ;
    fno:predicate :index ;
    fno:type :int ;
    fno:default 0 ;
    fno:required false .

:out a fno:Output ;
        fno:predicate :outResult ;
        fno:type :Any .
```

Code Fragment 4.2: FnO Parameter Description

4.1.2 Describing Implementations

A function can be implemented in various ways, each with its specific characteristics. In the context of this thesis, although all functions are implemented in Python, it is critical to distinguish between user-defined functions and those from established Python libraries, particularly in terms of trustworthiness. Functions from well-known Python libraries typically offer a higher level of reliability. In contrast, user-defined functions, while offering flexibility and customization, may not always undergo rigorous testing and peer review. This makes it crucial to convey metadata about the function's implementation.

The FnO ontology addresses this by using the FnO Implementation and Mapping Vocabulary (FnOI, FnOM). An FnO Implementation captures critical metadata about the function's source and characteristics, whereas an FnO Mapping serves to connect a specific implementation with a general function description. Additionally, it delineates the transformation process between the implementation and function description.

To ground the upcoming concepts in a practical example, we leverage the `Tokenizer` class and its `text_to_sequences` method from `keras` [19], a popular deep learning library. While this example introduces some structures explained in later sections, Appendix 9.2 provides the detailed illustration.

Implementation

The current FnO specification [8] does not currently support descriptions of Python implementations. This section proposes an extension to accommodate the description of Python functions, methods and classes by introducing the FnOI types PythonFunction, PythonMethod and PythonClass.

In Python a function or class is defined within a module, while a package is a collection of modules. In order to import the corresponding object it is enough to know the name of the implementation, the module and the package if relevant. A method, a distinct type of function, is defined within the scope of a Python class. To encapsulate this concept, the predicate **self** is suggested to specify from which class the method should be invoked, while **static** indicates whether it is a static method or not. The description offers the code documentation, aiding in a better understanding of how to use and comprehend the inner workings of the implemented function.

Mapping

An FnO Mapping serves as a crucial linkage between a function description and its practical implementation. The FnOM StringMethodMapping is employed to specify the implementation name of the function. Additionally, the FnOM PositionParameterMapping is used to define the positional order of parameters when the function is invoked while FnOM PropertyParameterMapping is used for arguments that can be given with a keyword. For outputs, the FnOM DefaultReturnMapping directly maps the return values.

Execution

The **Execution** class integrates all previous concepts to accurately describe the actual execution of a function. These descriptions are essential for providing provenance about the pipeline's processing. To illustrate this concept, an example involving sorting a list alphabetically is provided. The function `:sortAlph` and its implementation `:sortAlphPythImp` are used in this scenario. Detailed definitions are omitted for simplicity. The execution of this data transformation is demonstrated with an extra example in Code Fragment 4.3.

```
:SortTransformation a fno:Execution, prov:Activity ;
    fno:executes :SortAlphFunction ;
    fno:implementation :SortAlphPythImplementation ;
    fno:uses :SortAlphMapping ;
    :listInput :Input ;
    :orderedOutput :output .

:Input a prov:Entity ;
    rdf:value [ rdf:_1 "Mango"; rdf:_2 "Apple"; rdf:_3 "Banana" ] .

:Output a prov:Entity ;
    rdf:value [ rdf:_1 "Apple"; rdf:_2 "Banana"; rdf:_3 "Mango" ] .
```

Code Fragment 4.3: Execution of sorting strings alphabetically

4.2 Connecting FnO Functions and Python Implementations

4.2.1 Enabling Reusability of FnO Functions

RDF and linked data are integral components of the data ecosystem that must adhere to the FAIR principles—making data Findable, Accessible, Interoperable, and Reusable. Therefore, it is recommended to link the Python implementation to an existing function description for optimal reusability.

To achieve this seamlessly, the developed tool employs the `FnODescriptionMap` class, storing links to RDF documents containing FnO statements. By providing the name of the Python function, the tool searches for corresponding matches in the provided documents. However, it's crucial to note a limitation: the RDF subject of the description must precisely match the name of the implementation, excluding any prefixes. For instance, *average* and *mean*, despite having the same semantic meaning, are not linked. While this shortcoming could potentially be mitigated through NLP methods to assess similarity [20, 21], such approaches were beyond the scope of this thesis.

Generating FnO Functions from Python Functions

In cases where no FnO description is found within the provided documents, a description needs to be generated from the Python object itself. The `inspect` module in Python can be utilized to extract the signature of a function object. This signature encompasses all parameters, their names, and any type annotations if present. The `FunctionRDFGenerator` class employs this method to enable a straightforward conversion of the function signature into an RDF representation. All derived function descriptions are automatically added to the `PipelineGraph` of the `PipelineRDFGenerator`.

To ensure clarity and avoid ambiguity in the `PipelineGraph`, each parameter description is systematically named by concatenating the function name with a sequential numerical count, such as `:SortParameter1`. This methodological naming ensures that each parameter is uniquely identified within the RDF representation. Additionally, the names of the parameters as defined in the function definition are utilized as predicates.

An attribute call is a Python function that is invoked on another object. These calls can potentially modify the object they are called upon. For example, calling `append` on a list adds an element to that list. To capture this, a second output is defined that not only returns the regular output but also returns the modified object when executed. Consequently, a function description can have two outputs, which necessitates an expansion of FnOM. This expansion involves the use of a FnOM ValueReturnMapping. The correct usage is demonstrated in Code Fragment 4.4.

```
:append a fno:Function
    fno:returns [ rdf:_1 :appendOutput ; rdf:_2 :appendSelfOutput ] .

:appendMapping a fno:Mapping ;
    fno:returnMapping [ a fnom:ValueReturnMapping ;
            fnom:functionOutput :appendSelfOutput ],
        [ a fnom:DefaultReturnMapping ;
            fnom:functionOutput :appendOutput ] .
```

Code Fragment 4.4: Describing attribute calls

Built-in functions in Python, written in C, do not require importation. These functions lack a signature, making it impossible to automatically generate an FnO Function. However, the context in which a function is called provides valuable information. For instance, the number of arguments and whether the function is called on an object can help define its description. Note that this approach does not support type annotations or default values, resulting in reduced transparency. Therefore, it is recommended to manually define a function description and add it to the dictionary for improved clarity and completeness. This approach should only be used as a last resort.

4.2.2 Enabling Conversion between Implementation and RDF

Given the necessity to set up ML pipelines from generated RDF documents, a seamless transition between the described implementations and callable Python objects is imperative. This transition is facilitated by the `ImpMap` class. The following sections explore the mechanics of this conversion process and how mappings connect an FnO Implementation to a function description.

Please note that this thesis does not explore the capability to semantically describe Python environments. Therefore, it assumes that the necessary Python environment is already correctly set up by the user through the installation of required libraries.

Translating Python Objects to and from FnO Implementations

Capturing metadata like name, module, and docstring of Python objects is achieved through double underscore methods. The `importlib` module facilitates dynamic object importation, searching for the appropriate files along the system path [22, 23]. User-defined functions without installation within a library are identified by their filepaths, which are referenced using the **file** predicate. Importation of these functions is conducted via the module spec.

Because Python types are essentially classes, this conversion method facilitates the transformation of type annotations into RDF, allowing them to be used in parameter and output descriptions. This approach provides valuable semantic insights into a function's behavior. Parameters and outputs without a type annotation are mapped to the `Any` type.

Mapping an FnO Implementation to an FnO Function

After creating an FnO Implementation, it must be linked to an FnO Function through a mapping process. The significance of this mapping lies in its ability to offer users a selection among various available implementations with identical functionality. For instance, a machine learning pipeline document could be mapped to implementations using TensorFlow [24] and PyTorch [25], two well-known deep learning libraries. This could allow users to decide which library to utilize for execution.

The `inspect` module allows the determination of parameter types, which is essential for understanding how parameters are passed to a function. Positional parameters must be mapped using an index, while keyword parameters are mapped using a property. Some parameters can be both positional and keyword, thus requiring mappings for both the index and the keyword. Additionally, Python supports special parameters: variable positional arguments (`*args`) and variable keyword arguments (`**kwargs`). Variable positional arguments are passed as a tuple, while variable keyword arguments are passed as a dictionary, both of which are unpacked when provided to the function. To support these special parameters, this thesis proposes two new parameter mapping strategies: FnOM VarPositionalParameterMapping and FnOM VarPropertyParameterMapping. These strategies require only a link to the parameter using **functionParameter**. To exemplify these new mappings, consider the definition of the `train_test_split` function from the `scikit-learn` library [26]. This function takes a variable number of arrays and creates corresponding training and test sets. Code Fragment 4.5 demonstrates how the new strategy is applied.

```
:mapping a fno:Mapping ;
    fno:parameterMapping [ a fnom:VarPositionalParameterMapping ;
        fnom:functionParameter [ a fno:Parameter ;
            fno:predicate :arrays ]] .
```

Code Fragment 4.5: Parameter mapping for `train_test_split`

However, the mapping algorithm assumes that the arguments possess the same name as the FnO Parameter predicates to link them. This poses no issue when the function description is generated from the implementation. However, when the FnO Function originates from a pre-defined description, this correspondence isn't always guaranteed. This challenge could potentially be addressed using NLP techniques to map each argument to the correct FnO Parameter using its predicate.

5

Semantically describing pipelines

In the context of machine learning, a preprocessing pipeline transforms raw input data into meaningful feature data for use by a ML model. This pipeline consists of an ordered series of data transformations where each transformation's output serves as the input for the next, thus creating a continuous data flow that incrementally refines raw input.

Each python function can be conceptualized as a pipeline, with its parameters serving as the raw input data. Functions are described as outlined in Chapter 4, though such descriptions omit details about the function's internal mechanisms. This Chapter explores the comprehensive process of describing a Python function as a pipeline in RDF. It begins by establishing the concept of data flow, and how to describe this using RDF. Following this foundation, the discussion will focus on the methods for automatically generating an RDF description from a Python function pipeline.

5.1 Data Flow

Merely identifying individual data transformations does not provide a complete understanding of a pipeline. It is crucial to recognize how these transformations are interconnected, specifically how inputs and outputs from each step are linked, illustrating the flow of data through the pipeline. To describe this flow comprehensively, the FnO Composition Vocabulary (FnOC) can be employed. Function composition, in this context, refers to the application of one function to the results of another, effectively merging multiple functions into a single cohesive function. Thus, a pipeline can be conceptualized as a single function composed of all its constituent data transformations. An example of a simple composition is:

```
def sum3(a, b, c):
    return sum(sum(a, b), c)
```

5.1.1 Compositions

A FnO Composition consists of one or more FnOC CompositionMappings, where each mapping serves to connect the output of one function to the input of another. It consists of two FnOC CompositionMappingEndpoints, one for the source and one for the target. The properties used for this are **mapFrom** and **mapTo**. To specify which functions are involved, the predicate **constituentFunction** is used. Additionally, the specific parameters and outputs involved are identified using **functionParameter** and **functionOutput** respectively. Alternatively, a FnOC CompositionMapping can link to a constant term via **mapFromTerm** instead of **mapFrom**. An FnO Function is connected to the composition describing its body using the **composition** property.

To illustrate this concept further, consider the example of summing three numbers in Code Fragment 5.1. For brevity, only a subset of mappings is provided. The full composition can be found in Appendix 9.2. Two key mappings of the FnO Composition are highlighted. The first mapping demonstrates how the output of the first `sum` function serves as parameter `a` for the second sum function. The second mapping links the output of the second `sum` function to the output of the `sum3` function. To avoid ambiguity between the two instances of the sum function, the **applies** predicate is employed.

These complex compositions can be effectively visualized to clearly demonstrate the data flow within the pipeline. A visualization of the `sum3` example is provided in Figure 5.1. For further details on how this visualization is constructed and the tools used to create it, please refer to chapter 6.

```
:sum_1 fnoc:applies :sum .
:sum_2 fnoc:applies :sum .

:sum3Pipeline a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum_2 ;
                         fnoc:functionOutput :sumOutput ] ;
            fnoc:mapTo [ fnoc:constituentFunction :sum_1 ;
                         fnoc:functionParameter :sumParameter0 ] ],
        ...
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum_1 ;
                         fnoc:functionOutput :sumOutput ] ;
            fnoc:mapTo [ fnoc:constituentFunction :sum3 ;
                         fnoc:functionOutput :sum3Output ] ] .
```

Code Fragment 5.1: Composition of the sum3 function

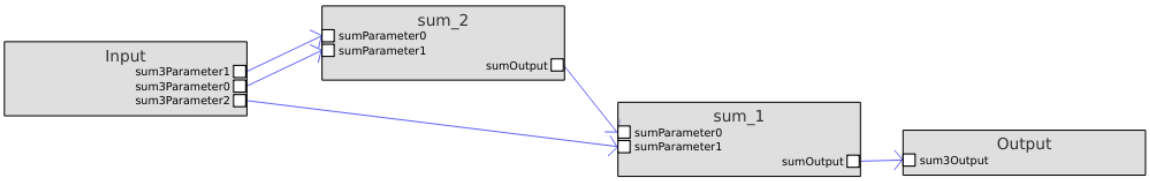

Figure 5.1: Visualized composition of the sum3 function

5.2 Describing Dataflow using the Abstract Syntax Tree

Abstract Syntax Trees (ASTs) are a valuable tool for analyzing data flow within Python functions. These tree-like structures represent the program's syntactic structure, with nodes capturing fundamental elements such as statements, expressions, and function calls. By traversing the AST, one can gain insights into the interconnections between these components and understand how data is transformed as it progresses through the code. This section will commence by examining the primary approach for extracting data flow from an AST, focusing on the process of handling individual nodes and establishing connections through assignments. Subsequently, the discussion will shift towards addressing the disruptive impact of loops and conditional expressions on data flow analysis.

5.2.1 Traversal of the Abstract Syntax Tree

Upon retrieving the source code of a function object using the `inspect` module, the `ast` module is used to capture the function's name and its body as an AST. After generating an RDF representation of the function object, the AST's body is traversed. For traversal purposes, the `ASTUtil` class is employed to capture all top-level nodes in the correct order. To delve deeper into the tree structure, each node is recursively handled using the `PipelineRDFGenerator`. This recursive approach ensures comprehensive coverage of the AST, enabling the extraction of metadata necessary for constructing the FnO Composition.

The `Importer` class plays a pivotal role in describing the functions used within a function as pipelines. The importer operates by parsing source code to extract and execute all import statements and importing all functions and classes defined in the same file. This method allows to process `import <> (from <>) as <>` statements to facilitate access to modules or objects via aliases. By offering multiple pipelines as FnO Compositions within the document, the tool empowers users with the flexibility to decide whether or not to include the internal body of used functions into the provenance generation process. This adaptable approach ensures that users can tailor the level of detail and inclusion of used functions based on their specific requirements and preferences.

Each node can be conceptualized as a data transformation that uses a specific function. This transformation yields an output, which can subsequently be utilized by other nodes in the data flow. As outlined in section 5.1.1, establishing a mapping needs both the function and the parameter/output it generates. Therefore, the handling of each node should entail returning an output or parameter along with its contextual information. Further sections will delve deeper into how each type of node contributes to the generation of an FnO Composition, representing the data flow. An example of an AST can be found in Appendix 9.2.

5.2.2 Handling Function Calls

A `Call` node consists of the function and its arguments (positional or keyword) represented as nodes. To determine the object for the call, the `Importer` class is used if it's a standalone function call. Otherwise, the context of the call (method, class, or imported module) determines the object. This object's internal body can then be translated into an FnO Composition as described previously.

To ensure unique identification within a single FnO Composition, each function call is enumerated and connected to the corresponding FnO Function using **applies**. The accurate linkage between parameters and arguments is achieved using the FnO Function generated and its corresponding FnO Mapping. Algorithm 1 presents the pseudocode for this process. This approach ensures that arguments are correctly mapped according to position, keyword, or variable parameters by comparing argument name to FnO Parameter predicates. Mapping strategies are employed to link to a variable parameter, as further discussed in section 5.2.6. Parameters without mapped arguments are assigned their default values, if any exist.

The `FnODescriptor` class manages the mapping of outputs and parameters, generating both the FnOC CompositionMappingEndpoints based on the provided context and parameters/outputs. Figure 5.2 illustrates the flow of this process with an example. To ensure the mapping is correctly incorporated into the appropriate FnO Composition, the `PipelineRDFGenerator` maintains awareness of the current scope, signifying which pipeline is undergoing conversion. This scope management is crucial for allowing pipelines to be recursively described and converted. When encountering a recursive call, it is crucial to distinguish between the recursive call itself and the outer function. For parameters or outputs, the URI of the function is used without enumeration. The recursive call is treated like any other function call, properly enumerated, and linked with **applies**.

Algorithm 1 Linking Arguments to Parameters

Require: `f`: FnO Function, `call`: FnO Function call, `args`: arguments given to the call, `value`: contains value if function is a method

```
varpos ← variable positional parameter of f
varkey ← variable keyword parameter of f
defaults ← positional and keyword parameters of f
if f is a method then
    value_output ← handle node value
    Map value_output to (call, self Parameter)
end if
for i, arg in enumeration of args do
    arg_output ← handle node arg
    if there is a positional parameter for the argument then
        par ← parameter at position i
        Map arg_output to (call, par)
        Remove par from defaults
    else if function allows variable positional arguments then
        Map arg_output to (call, varpos) at index i
    end if
end for
for key, value, in kargs do
    value_output ← handle node value
    if there is a keyword parameter with predicate key then
        par ← parameter with predicate key
        Map value_output to (call, par)
        Remove par from defaults
    else if function allows variable keyword arguments then
        Map value_output to (call, par) with property key
    end if
end for
for par in defaults do
    if par has a default value then
        default ← default value of par
        Map (None, default) to (call, par)
    end if
end for
```

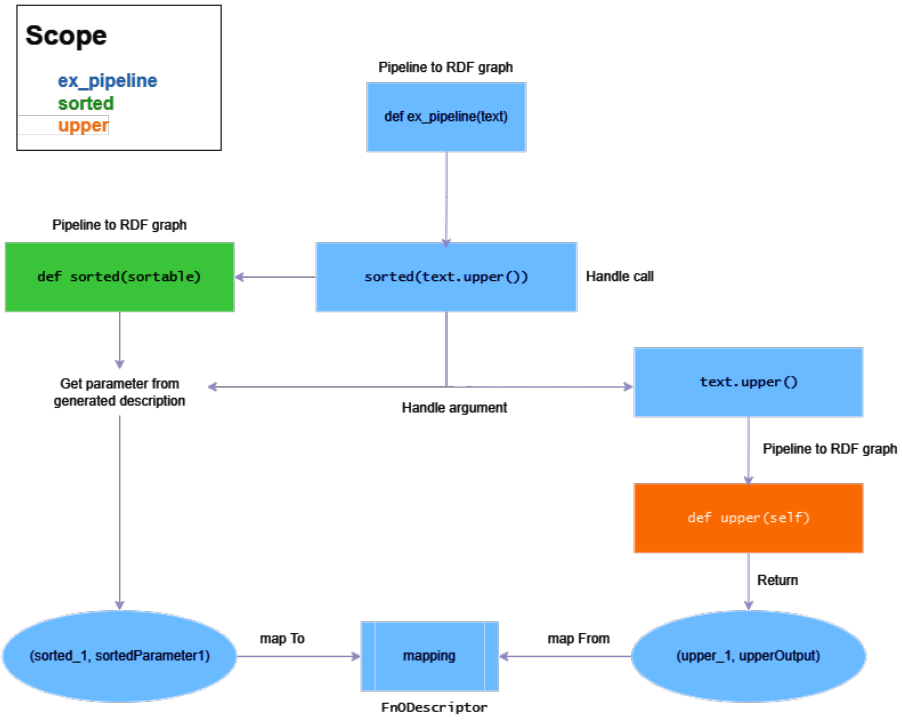

Figure 5.2: Mapping process

When a function is called on a variable, this variable should be assigned to the output mapped with a FnOM ValueReturnMapping. This ensures that further functions use the potentially modified variable. How assignment is handled is discussed in the next section.

5.2.3 Handling Assignments

Assignment statements play a crucial role in data flow as they determine how the output from one node is used by subsequent nodes within the pipeline. To maintain a concise representation, the variable itself is omitted, and instead, a connection is established between the assigned value and the node where it's used. This eliminates the need to explicitly include the variable name within the FnO Composition.

A central mechanism tracks assigned values using a dictionary. This dictionary allows nodes to directly reference the assigned output and potentially use it as input for their own parameters. When encountering a variable name (represented by a `Name` node), the system checks the assignment dictionary. If present, it retrieves the assigned output instead of simply returning the variable name. Figure 5.3 illustrates this process with an example.

A dictionary keeps track of the types associated with each variable by examining the type annotation of the generated value output. In cases where type annotations are absent, this dictionary plays a crucial role in propagating the type throughout the pipeline. For instance, the variable `text` is identified as a string through type annotation in the function definition. Subsequently, when `text.upper()` is invoked, the system consults the dictionary to determine that `str.upper()` must be imported.

When constructing a pipeline, FnO Parameter predicates function as variables. In the provided example, the `sorted` function uses the variable `text` as its argument. To facilitate mapping to the input parameter `ex_pipeline-Parameter0`, these predicates need to be assigned using the assignment dictionary.

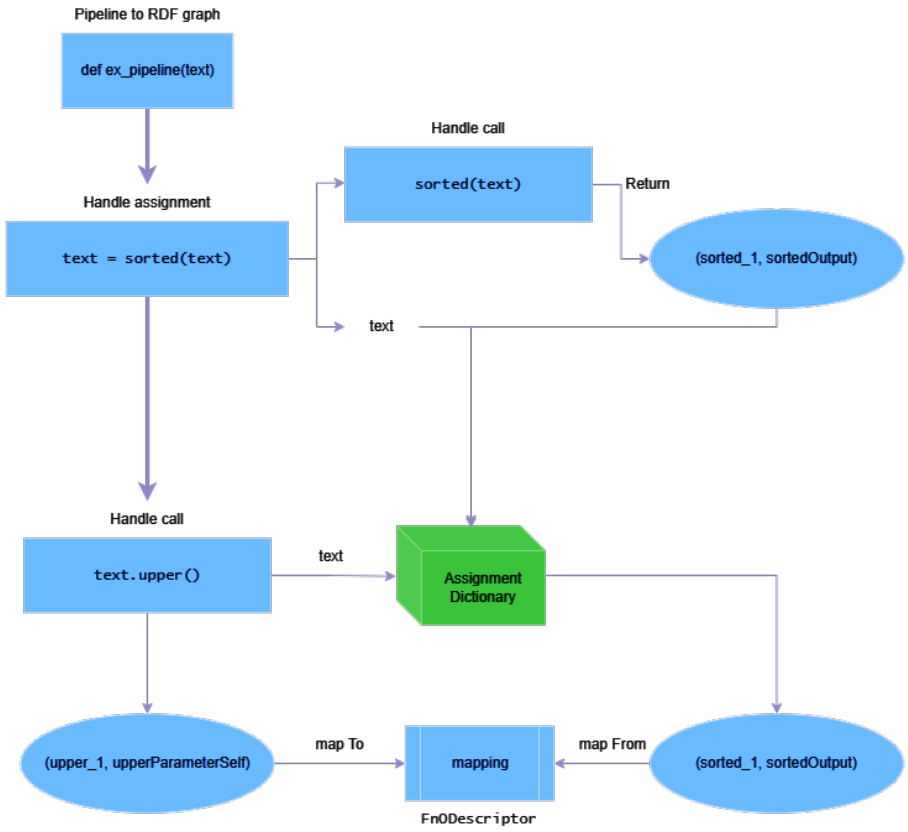

Figure 5.3: Assignment process

5.2.4 Mapping From Term

A `Constant` node represents an immutable value, such as a string or an integer. When this node is processed, it returns the value directly, without providing context. If no context is given, the system utilizes **mapFromTerm** to map an instance directly to a function parameter or output. These terms can be represented in RDF using the `Literal` class from the `rdflib` library. A literal presenting the number 4 is represented as follows: `"4"^^xsd:integer`. Notice how the datatype is an RDF URI.

The primary advantage of using this class is that it allows deriving the constant value from the RDF description, although this feature is exclusive to constant types. Consider the function `apply` that applies a function on each element in an array. In the AST, the function object is represented as a `Name` node. When processed, the system first checks if it is a variable within the assignment dictionary. If not, the system uses the `Importer` to locate an object with that name. In this scenario, the function object is found and returned without context.

However, for **mapFromTerm** to work, the function object needs to be represented as a `literal` with its string representation as the value and a specific RDF datatype indicating it's a function implementation. Section 4.2.2 details how this datatype is constructed. Code Fragment 5.2 illustrates this example. This approach facilitates conversion for any described term, as function objects can be determined from the datatype URI. More complex instances are constructed from nodes that provide context. These instances are abstracted as function outputs and used with **mapFrom**.

```
:comp a fnoc:Composition ;
    fnoc:composedOf [
        fnoc:mapFromTerm "<str.upper object at
        ↪ 0x7f2bb399b550>"^^:str_upperImplementation ;
        fnoc:mapTo [ fnoc:functionParameter :applyParameter1 ;
            fnoc:constituentFunction :apply_1 ]
    ] .
```

Code Fragment 5.2: Mapping a function object as a term

5.2.5 Accessing containers

In Python, container access is achieved through subscription using square brackets []. This notation invokes the `getitem` function to retrieve elements from the container. Thus, to properly manage subscription, the node needs to be transformed and treated as a `Call` node with the index as argument. When assigning a value to a container element using indexing (e.g., `container[index] = value`) a problem arises. While handling assignments, the output value is typically mapped to each target. However, in this case, the target refers to an element within the container rather than the container itself. Consequently, when the container is treated as a variable, it remains assigned to the original output, failing to encompass the newly made change. This issue is depicted in Figure 5.4. To address this limitation, the `setitem` function is used. This function includes an additional parameter for assignment, allowing the variable storing the container to be assigned to the output of this function. This ensures that the modified version is appropriately used thereafter. The process is illustrated in Figure 5.5 with a detailed example.

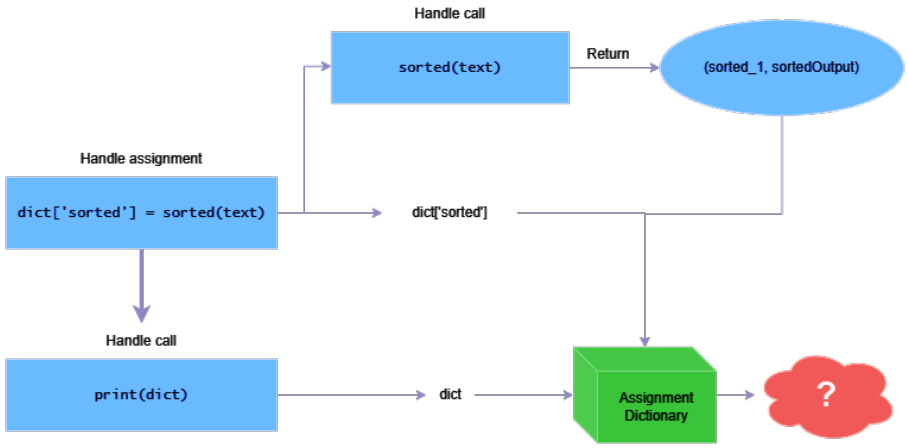

Figure 5.4: Subscript assignment problem

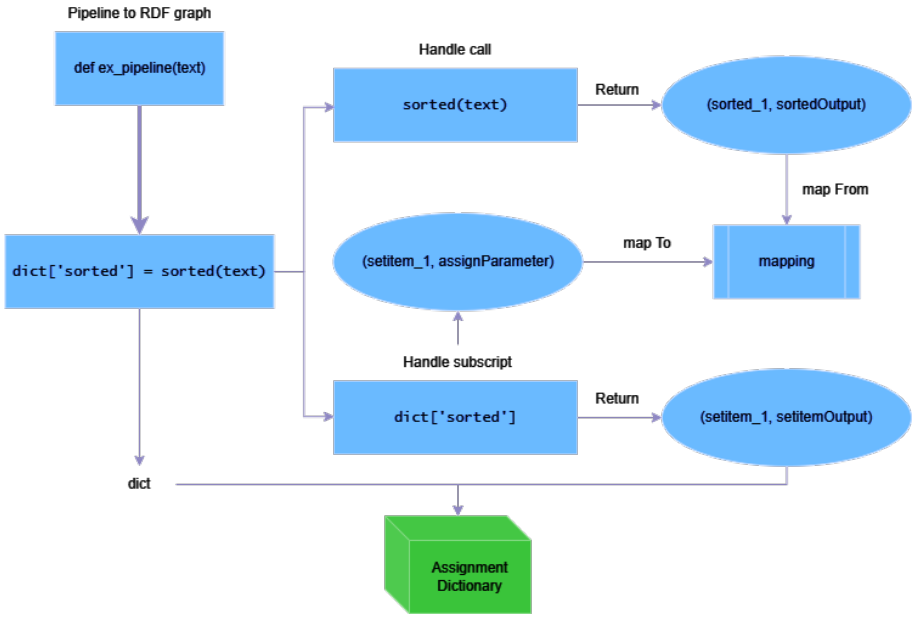

Figure 5.5: Subscript Assignment process

Slicing enables access to multiple elements within a container by specifying lower and upper bounds along with a step parameter. This is achieved by providing a slicing object to the `getitem` function. Unfortunately, `slice` is a built-in class and cannot be automatically converted. Therefore, a standard function description is provided, shown in Code Fragment 5.3. When handling a slicing node, the output of this description must be mapped to the index parameter of `getitem`.

```
pf:slice a fno:Function ;
         fno:expects [ a rdf:Seq ;
                       rdf:_2 pf:LowerIndexParameter ;
                       rdf:_3 pf:UpperIndexParameter ;
                       rdf:_4 pf:StepParameter ] ;
         fno:returns [ a rdf:Seq ;
                       rdf:_1 pf:SliceOutput ] ;
         dcterms:descrition "Create a slicing object." .
```

Code Fragment 5.3: FnO Description for the `slice` constructor

5.2.6 Creating containers

Containers like lists, tuples, and dictionaries are common data structures in pipelines. To ensure a concise representation, standard descriptions are provided for their constructors. However, the current FnO specification doesn't support mapping to parameters that allow an unspecified amount of arguments.

This thesis introduces the FnOC SetItem mapping strategy to address this limitation. It simplifies container construction by eliminating the need for a series of append functions. Code Fragment 5.4 demonstrates this approach using a list construction scenario. Here, the key takeaway is that FnOC SetItem allows assigning elements at specific indices within the container being built, using the **index** property.

```
:comp a fnoc:Composition
    fnoc:composedOf
        [ fnoc:mapFrom [ fnoc:constituentFunction :amax_1 ;
                fnoc:functionOutput :amaxOutput ] ;
            fnoc:mapTo [ fnoc:constituentFunction pf:list ;
                fnoc:functionParameter pf:Elements ;
                fnoc:mappingStrategy fnoc:SetItem ; fnoc:index 0 ]],
        [ fnoc:mapFrom [ fnoc:constituentFunction :amax_2 ;
                fnoc:functionOutput :amaxOutput ] ;
            fnoc:mapTo [ fnoc:constituentFunction pf:list ;
                fnoc:functionParameter pf:Elements ;
                fnoc:mappingStrategy fnoc:SetItem ; fnoc:index 1 ]] .
```

Code Fragment 5.4: FnOC SetItem strategy example

The dictionary constructor takes a list of tuples representing pairs using FnOC SetItem. This approach allows for greater flexibility by not limiting keys to simple types but also accommodating more diverse outputs of functions, as the **index** property expects a `Literal`.

The FnOC SetItem strategy becomes essential when functions accept a variable number of arguments, denoted by `*args` or `**kwargs`. Again, consider the `train_test_split` function. A dataset containing the samples and a dataset containing labels can be mapped to `*arrays` using the FnOC SetItem strategy, as shown in Code Fragment 5.5.

```
:comp a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFromTerm "[...]"^^:DataFrame ; # samples
            fnoc:mapTo [ fnoc:constituentFunction :train_test_split ;
                fnoc:functionParameter :train_test_splitParameter0 ;
                fnoc:mappingStrategy fnoc:SetItem ; fnoc:index 0 ]],
        [ fnoc:mapFromTerm "[...]"^^:DataFrame ; # labels
            fnoc:mapTo [ fnoc:constituentFunction :train_test_split ;
                fnoc:functionParameter :train_test_splitParameter0 ;
                fnoc:mappingStrategy fnoc:SetItem ; fnoc:index 1 ]] .
```

Code Fragment 5.5: Mapping arguments to variable positional parameter of `train_test_split`

5.3 Disruptive Impact of Control Flow Structures

Control flow structures like for loops and if-else statements influence the data flow within a pipeline in unique ways. This is because the exact path the data will traverse in these structures can only be determined during execution. As a result, it is crucial to handle such structures correctly to ensure both a clear representation and proper execution of the pipeline. The following section will delve deeper into these challenges and explore strategies for effectively managing them.

5.3.1 For statements

In a for loop, the body is executed for each target available from an iterator. However, the number of times this body will be executed cannot be known during the parsing of the AST. Therefore, a description is needed that correctly encompasses a for-loop to allow a user to execute it. A standard function description is provided, which can be seen in Code Fragment 5.6. When encountering a loop, a call is instantiated using **applies** with the corresponding description. The body of the for-loop is treated as a FnO Composition and mapped to the call using the **composition** property.

```
pf:for a fno:Function ;
       fno:expects [ a rdf:Seq ;
                     rdf:_1 pf:IterParameter ;
       fno:returns [ a rdf:Seq ;
                     rdf:_1 pf:TargetOutput ]
       dcterms:descrition "The for function iterates over ..." .
```

Code Fragment 5.6: FnO Description for for-loop

A `For` node in the AST contains three elements: the node representing the value to be iterated, the variable assigned to each iteration, and the body represented as an AST. After mapping the iterator and creating variables for the target output, the nodes within the body are iterated and processed. Figure 5.6 illustrates the complete handling process. To correctly construct the body of the for-loop, the scope is adjusted to match that of the loop's call. This adjustment ensures that all mappings are added to the FnO Composition of the loop call. When processing a `Return` node, the nodes are mapped to the output of the current scope. However, `Return` nodes inside a for-loop must be mapped to the output of the function where the loop is defined, rather than the output of the loop itself. Consequently, it is essential for the system to track both the current scope, which is used defining pipelines of functions and control flow structures, and the outer scope, which is only set upon entering a function and is used when handling return nodes.

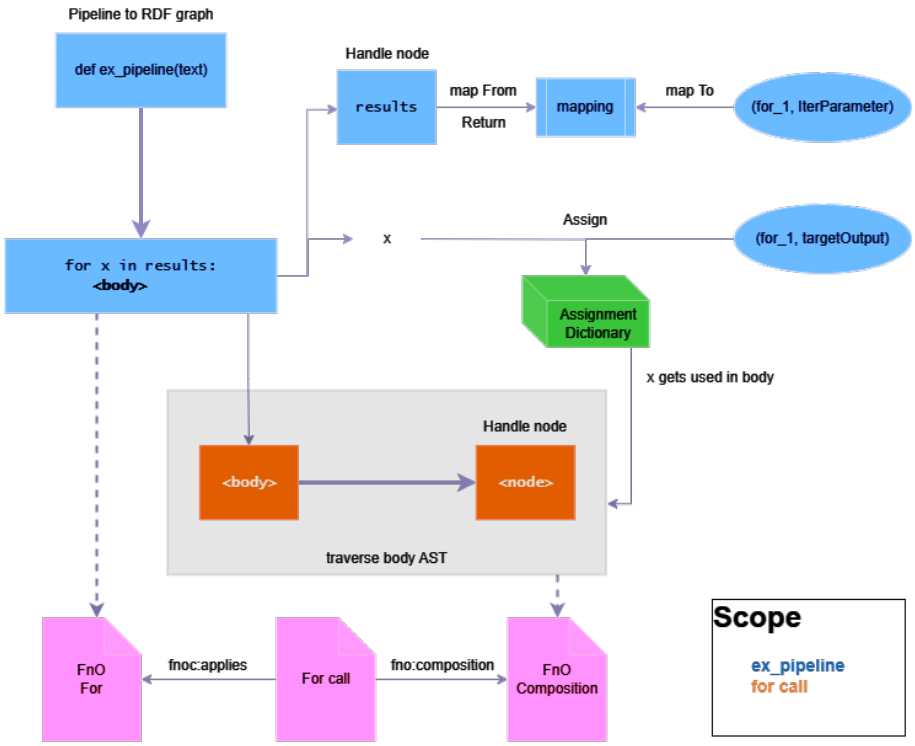

Figure 5.6: For-loop process

The iterator results are often unpacked to assign multiple targets at once, for example when enumerating a variable: `for i, text in enumerate(texts)`. To avoid cluttering the RDF documents, an alternative mapping strategy is proposed. The FnOC GetItem strategy enables concise mapping from an item within a container. Code Fragment 5.7 demonstrates how this strategy can be used to map the `text` target to the `upper` function.

```
:unpack a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFrom [ foc:constituentFunction :for_1 ;
                fnoc:functionOutput :TargetOutput ;
                fnoc:mappingStrategy fnoc:GetItem ; fnoc:index 1 ] ;
            fnoc:mapTo [ fnoc:constituentFunction :upper_1
                fnoc:functionParameter :UpperParameterSelf ]] .
```

Code Fragment 5.7: For Loop unpacking with mapping strategy

5.3.2 Conditional statement

Conditional statements present multiple paths of execution. It is impossible to determine in advance whether the body of the if-clause or the body of the else-clause will be executed. The current FnO specification has no means to convey if a mapping must only be made under a certain condition. A problem arises when a variable is modified in both the if- and else-clauses, as the assignments will override each other, as illustrated in Figure 5.7. When the variable is used outside the conditional statement, it becomes unclear which assignment should be relied upon for subsequent operations. The following chapters will discuss the solutions to resolve these issues.

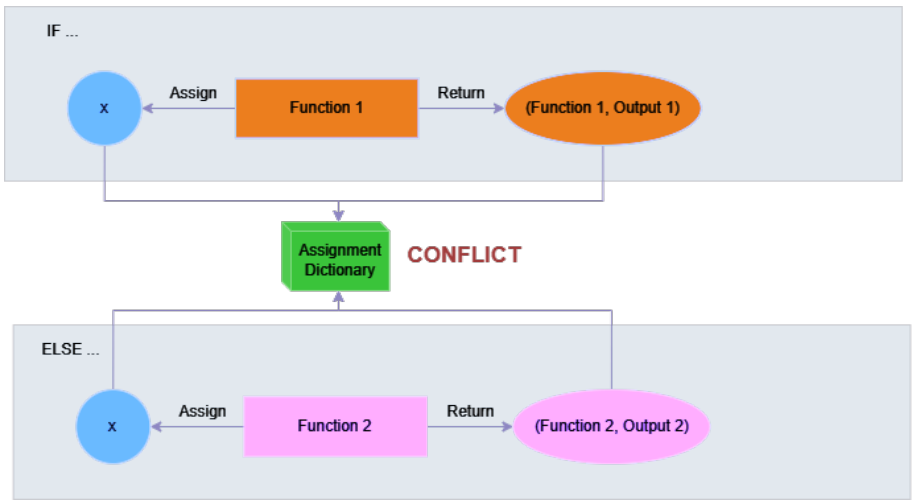

Figure 5.7: If-Else clause problem

Conditional Mapping

This paper proposes expanding a FnOC CompositionMapping with a FnOC with an endpoint that defines a condition that must be met for the mapping to be executed. Consider a scenario where the minimum value of an array of integers is used for division. To prevent division by zero, a conditional statement first checks whether the provided value is greater than zero. Code Fragment 5.8 illustrates how this can be described using a conditional mapping. By defining the strategy FnoC IsTrue, this mapping is only relevant when `:gtOutput` is `True`. The complementary mapping strategy is IsFalse.

To implement this, the system must track multiple elements. After handling the test condition, the provided output is used to identify each if-else statement. Additionally, it distinguishes whether it is processing the if-clause or the else-clause. Whenever a FnOC CompositionMapping is made, a list containing all condition outputs with an indication of which conditional body is being processed is provided. The FnOC IsTrue strategy is employed when processing the if-clause, and IsFalse otherwise.

These conditions are essential when replicating a pipeline from a FnO Composition for execution. Further details on how these conditional mappings are used during execution can be found in section 6.2.2.

```
:comp a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFrom [ fnoc:functionOutput :minOutput ;
                fnoc:constituentFunction :min_1 ] ;
            fnoc:mapTo [ fnoc:functionParameter :divParameter1 ;
                fnoc:constituentFunction :div_1 ] ;
            fnoc:mapCondition [ fnoc:functionOutput :gtOutput ;
                fnoc:constituentFunction :gt_1 ;
                fnoc:mappingStrategy fnoc:IsTrue ] ] .
```

Code Fragment 5.8: Conditonal mapping to prevent division by zero

Merging Results into If Expression

A key challenge arises when handling assignments within both branches of an if-else statement. To prevent overwriting assignments, variables are combined into a single if expression. This expression takes a test condition, a value to return if true, and a value to return if false. Figure 5.8 illustrates this concept. The system employs a dictionary (`ifmap`) to track if expressions for each variable within a test. Algorithm 2 outlines how assignments are handled when processing a test. It takes the value to be assigned, the target variable, the current test, and the `ifmap` dictionary as input. The algorithm ensures proper handling of assignments within both if- and else-clauses.

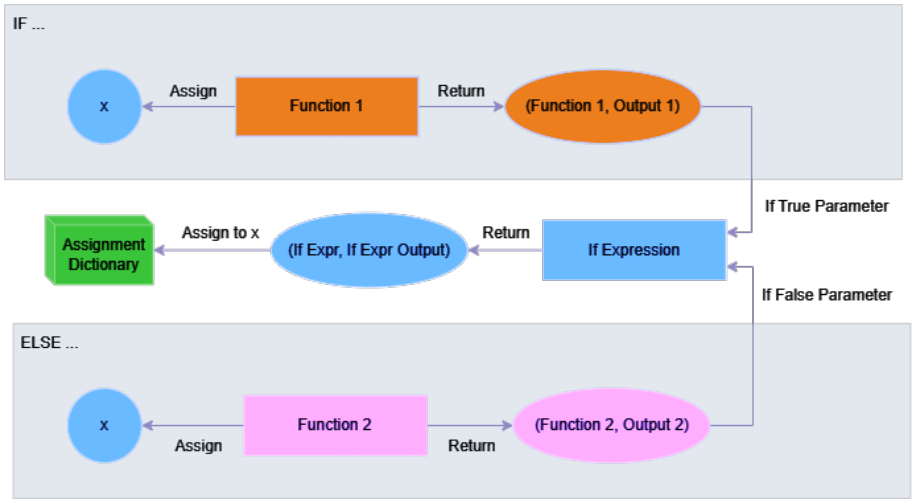

Figure 5.8: If-Else expression solution

Algorithm 2 Handling assignments in an If-Else clause

Require: `value`: value that must be assigned, `target`: variable that gets value, `test`: current test being handled, `ifmap`: dictionary mapping variables to an if expression for each test

```
value_output ← handle node value
if test is not None then
    if test is processing if-clause then
        expr ← if expression that returns value_output if True
        ifmap[test][target] ← expr
        value_output ← output of expr
    end if
    if inside the ifelse body then
        if target in ifmap[test] then
            expr ← ifmap[test][target]
            update expr such that it returns value_output if False
        else
            expr ← if expression that returns value_output if False
            ifmap[test][target] ← expr
        end if
        value_output ← output of expr
    end if
end if
assign target to value_output
```

The system also needs to accommodate nested if-else statements. Algorithm 3 addresses this by correctly assigning the output of each if expression within a nested structure. It iterates through the `ifmap` dictionary and updates expressions based on the current and previous test conditions.

Algorithm 3 Handling nested If-Else clauses

Require: `test`: current test being handled, `prev_test`: previous test being handled, `ifmap`: dictionary containing mapping variables to an if expression for each test

```
for target, expr in ifmap do
    if prev_test is processing if-clause then
        outer_expr ← if expression that returns the output of expr if prev_test is True
    else
        if target in ifmap[prev_test] then
            outer_expr ← ifmap[prev_test]
            Update outer_expr such that it returns the output of expr if prev_test is False
        else
            outer_expr ← if expression that returns output of expr if prev_test is False
        end if
    end if
    assign target to output of outer_expr
    ifmap[prev_test] ← outer_expr
end for
```

Another scenario that must be addressed is capturing assignments made outside a conditional statement. Consider a conditional statement that only has an if-clause. The generated if expression will return `None` if the test evaluates to `False`. If the variable already had a value assigned, the if expression should return this value instead of `None`. Therefore, each test should first store a copy of all assigned values before handling the body. After processing the bodies, all expressions can be completed with these stored values if the corresponding parameter is not linked. To make this possible for nested if-else statements, each test condition must update its stored assignments with the actual values. The inner test condition can then access these assignments of the outer test. Before handling an else-clause after processing the if-clause, the stored copy must be reverted to its previous state to undo any assignments made in the conditional body. To clarify, consider the following example. The function `binaryCount` takes two bits, either 0 or 1, and calculates the decimal output. The source code for this function is shown in Code Fragment 5.9. The visualized composition is depicted in Figure 5.9. In this visualization, `ifexpr_1` represents the if-else statement in the first if-clause, while `ifexpr_3` represents the if-clause within the else-clause. Both results are combined in `ifexpr_2`, which evaluates the first bit. Notably, `ifexpr_3` will return 0 if its condition evaluates to `False`, as it retains the assigned value from the outer scope.

```
def binaryCount(bit1, bit2):
    out = 0
    if bit1:
        if bit2:
            out = 3
        else:
            out = 2
    else:
        if bit2:
            out = 1

    return out
```

Code Fragment 5.9: source code of `binaryCount`

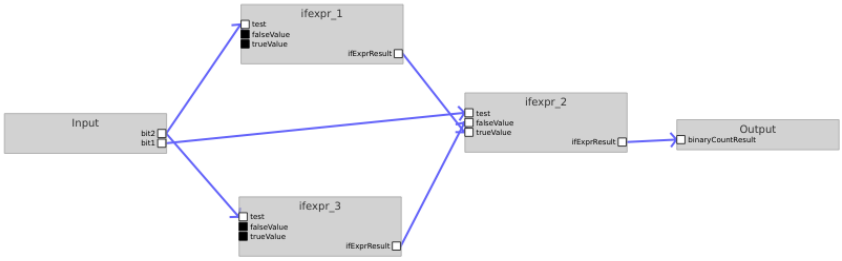

Figure 5.9: `binaryCount` composition

6

Generating Provenance for described pipelines

The descriptions generated in chapters 4 and 5 can now be used to provide provenance information for data processed through a pipeline. To achieve this, the descriptions must be transformed into an executable format that can track all transformations applied to these data objects. This chapter begins by discussing the `Pipeline` class, which facilitates execution. Following this, the process of deriving a `Pipeline` from an RDF description is explored. Next, the chapter explains how this class generates provenance information. It concludes with a discussion on integrating ML Schema to enhance provenance for ML operations.

6.1 Providing an executable format

The `Pipeline` class is built upon the foundation of the `Flowchart` class from the `pyqtgraph` library [27]. A `Flowchart` facilitates visual programming by offering nodes connected through wires. Each node represents a Python function with input and output terminals. This architecture provides an excellent framework for representing data flow and enabling its step-by-step execution. However, the `Flowchart` class has limitations. It allows users to modify the structure, which can lead to inconsistencies. Additionally, it cannot execute loops or incorporate conditional links between nodes. To address these limitations, the source code was modified to develop the `Pipeline` class.

In a `Pipeline`, each function call in an FnO Composition is represented as a node, while its parameters and outputs are represented as terminals. Terminals store values and can be connected to each other. A node can use the values stored in its input terminals and use them for processing to set its own output terminals. The nodes and terminals are implemented using the `FunctionNode` and `ParameterTerminal` classes, respectively.

Figure 6.1 depicts the node for the function `compile`, responsible for compiling a TensorFlow model for training. The figure illustrates the parameters it expects on the left and the outputs it generates on the right. As observed, the function is invoked on a model and subsequently returns the compiled model through terminal `self_output`.

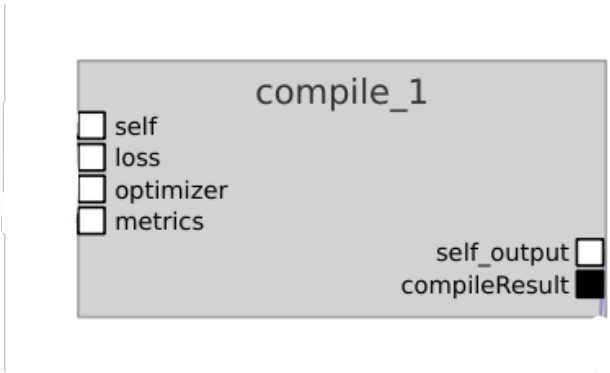

Figure 6.1: Visualization of node `compile_1`

Terminals can be connected to simulate data flow. These connections propagate the values from an output terminal to an input terminal of another node during processing. Figure 6.2 shows an example where the function `create_data` returns a `DataFrame`. Subsequently, a function is applied to a column using the `apply` function.

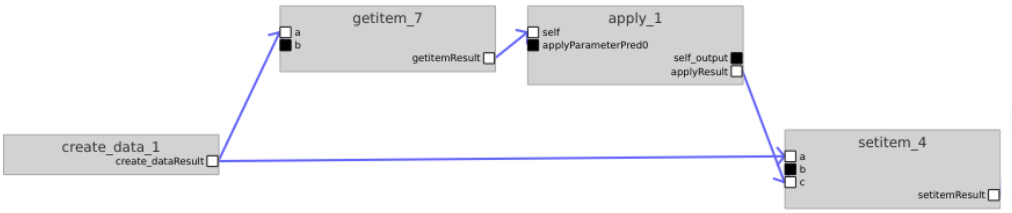

Figure 6.2: Visualization of applying a function to a `Dataframe` row

A `Pipeline` stores all nodes. Each node contains its respective terminals, and each terminal maintains a reference to the `ConnectionItem` that connects it to another terminal. This hierarchical structure ensures that the pipeline has access to all structural information about the data flow. The primary function of a `Pipeline` is to incrementally construct a data flow by adding nodes and establishing connections between their terminals. Once the configuration is complete, its responsibility shifts to calculating the correct processing order for the nodes based on their dependencies. It then orchestrates the execution by propagating results through the nodes in this calculated order.

6.2 Replicating Pipeline from FnO Descriptions

With the foundation of the `Pipeline` class established, the process of constructing a pipeline from an RDF document can be examined in detail. The `PipelineChartGenerator` provides all necessary functionality. The RDF document must be provided in the form of a `PipelineGraph`.

The first step involves transforming an FnO Function into an executable node. Following this, the method for using compositions to connect terminals is explored. Subsequently, the integration of compositions from utilized nodes into the pipeline is discussed, highlighting how to reveal or hide their internal structure. Finally, the setup process for loops that ensure correct execution is addressed.

6.2.1 Transforming FnO Functions to Nodes

The first step involves transforming an FnO Function into an executable node. A `FunctionNode` representing a function call is uniquely identified by the URI of the FnO Function. To enhance readability, a display name is derived from this URI, excluding its prefix. The graph is first queried to construct the terminals based on the function description of the call. FnO Parameters are converted into input terminals, while FnO Outputs are converted into output terminals. Similar to a node, each `ParameterTerminal` has a unique URI-based identifier, and the predicate is used as its displayed name. The expected type of a terminal is determined by converting the type description into a Python type using the `ImpMap`. This type information is used to verify the correctness of values assigned to terminals.

When creating a node, a subgraph containing all relevant information of the FnO Function is provided. This includes the function description with its FnO Parameters and Outputs, its FnO Mapping, and the FnO Implementation details. During node initialization an attempt is made to retrieve a function object using the `ImpMap`. This function object will be used when processing the node. If the FnO Implementation lacks sufficient information and the object cannot be retrieved, it will be fetched during execution. This process will be detailed further in section 6.3. Chapter 5 introduced standard descriptions for functionalities that cannot be represented by a function object, such as if expressions. When encountering such a description, a special node must be created. This node inherits from `FunctionNode`, with the primary difference being how these nodes handle processing.

6.2.2 Connecting Terminals based on FnO Compositions

A `Pipeline` is initialized in the same manner as a node, by providing the corresponding URI and the terminals generated from the function description. It then creates two nodes: an `inputNode` and an `outputNode`. The `inputNode` contains all FnO Parameters as output terminals, while the `outputNode` contains all FnO Outputs as input terminals. The `inputNode` is used to feed arguments into the pipeline provided by the user, while the `outputNode` captures the generated output. Figure 6.3 shows an example of a pipeline based on the function `sort_and_upper`. This function takes a string value as a parameter and returns the string sorted and converted to uppercase. Once the pipeline is initialized, all functions used in the corresponding FnO Composition are extracted. These FnO Functions are then iterated over to create `FunctionNode` instances. The nodes are subsequently added to the pipeline. After this iteration, the pipeline is ready to establish connections based on the FnOC CompositionsMappings.

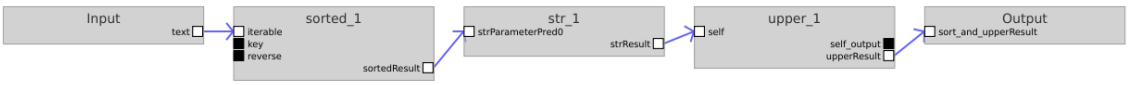

Figure 6.3: Visualization of pipeline for `sort_and_upper`

Using the provided `PipelineGraph`, all FnOC CompositionMappings within a FnO Composition are fetched. These mappings include the URIs referenced in both FnOC CompositionMappingEndpoints. The URIs are used to identify the appropriate nodes in the `Pipeline` along with the corresponding terminals. The `PipelineChartGenerator` then establishes connections using the `connectTerminals` function of the `Pipeline`. This creates a `ConnectionItem` that references its source and target terminal. Subsequently, the item is stored in each terminal. If the source is a term, the corresponding instance is retrieved using the method described in section 5.2.4. The value of the target terminal is subsequently set to this instance using the `setValue` method of the `ParameterTerminal`.

Sections 5.2.6 and 5.3.1 introduced mapping strategies aimed at neatly describing access to containers. During the mapping of two terminals, each FnOC CompositionMappingEndpoint is examined for any strategy. Upon encountering a strategy, the `ConnectionItem` will store the corresponding index. For a FnoC SetItem strategy, the index is stored in the attribute `toIndex`, whereas for a FnOC GetItem strategy, it is stored in the attribute `fromIndex`. In the absence of a strategy, the corresponding attributes default to `None`. These indices serve as crucial markers during processing to ensure the correct functionality.

Input terminals utilizing the FnOC SetItem strategy can accept input from multiple output terminals. These terminals are flagged as multi-value terminals and require a dictionary that maps each index to its designated target, rather than a single value. Since a terminal might be added multiple times to a container, a `ConnectionItem` should store a list of indices instead of just one. When creating containers from terms, no connection is made to store an index; the term is directly added to the dictionary stored in the terminal's value.

Conditional Connections

Section 5.3.2 introduced conditional mappings, requiring special handling during connection establishment. A `Condition` node, derived from `FunctionNode`, acts as a gatekeeper, regulating data flow based on a provided function's output and strategy. The `Condition` class effectively divides the connection into two parts: one mapping the terminal providing input to its input terminals, and the other mapping its output terminal to the terminal receiving input. In scenarios with nested if-else statements, an FnOC CompositionMapping can be subject to multiple conditions. Consequently, several `Condition` nodes will sequentially segment the connection. Considering these factors and the mapping strategies discussed in the previous section, Algorithm 4 outlines the procedure for mapping two terminals within a pipeline.

If no conditions are present, the algorithm directly connects `term1` to `term2`. When conditions exist, it creates condition nodes, positioning them as intermediaries between the source and target terminals. The condition nodes are connected sequentially, preserving the connection chain's integrity. The final step connects the last condition node to `term2`, completing the terminal connection process. The initial connection uses `fromIndices`, while the final connection uses `toIndices`. Figure 6.4 illustrates an example where the output must be `None` if the denominator is zero or lower. Green `Condition` nodes represent the FnOC IsTrue strategy, while red nodes represent the IsFalse strategy.

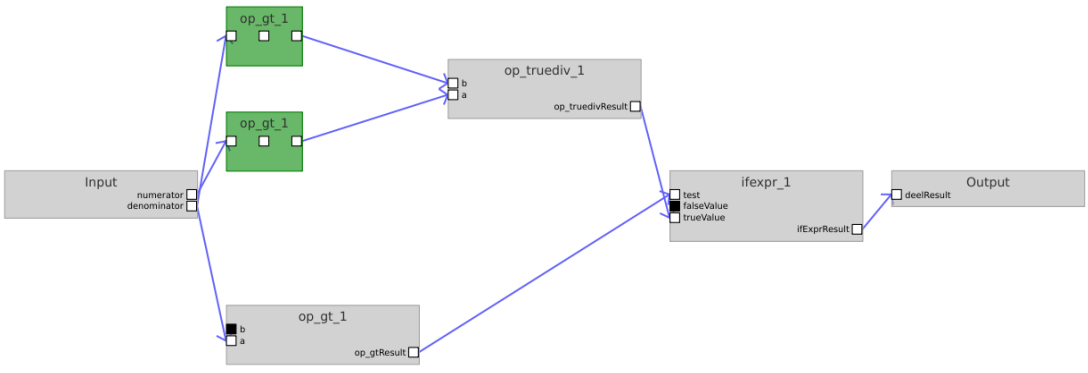

Figure 6.4: Divide by zero with conditional mapping

Algorithm 4 Connecting Terminals

Require: Source terminal `term1`, Target terminal `term2`, list of mapping strategies `indices`, list of conditions `conditions`

```
if no conditions present then
    Connect term1 to term2 using indices
else
    if indices is not None then
        fromIndices ← (indices[0], None)
        toIndices ← (None, indices[1])
    end if
    for each test, strategy in conditions do
        node ← Condition(test, strategy)
        Add node node to self
        Connect test terminal to test terminal of node
        Append node to nodes
    end for
    source ← term1
    Connect source to input of nodes[0] with fromIndices
    source ← output of nodes[0]
    for node in remaining nodes do
        Connect source to input of node
        source ← output of node
    end for
    Connect source to term2 with toIndices
end if
```

6.2.3 Incorporating Nested Compositions

When creating nodes, the system may encounter functions that are described using a FnO Composition, which can either be for-loops or other functions. In such cases, the composition is processed to ensure the `Pipeline` incorporates all necessary calls and their connections. Special handling is required for for-loops and their internal nodes to facilitate proper execution. The tool provides users with the option to exposes all internal nodes of a function or to represent is as a simple node, hiding its function body. This section first addresses the handling of for-loops and then discusses the handling of nested functions. Note that when a node is created in the following sections, it might involve a one of these special cases. By recursiely handling the creation of nodes the system incorporates all FnO Compositions. To prevent infinite looping, the internal nodes of a recursive call are not added to the **Pipeline**.

Setting up For-Loops for Execution

Incorporating a for-loop by its FnO Composition is straightforward. The input node is a specialized `ForNode`, representing the function described in Code Fragment 5.6. Then, all functions used in its composition are iterated, creating a corresponding node for each function. Nodes within the loop contain a reference to the `ForNode`. During execution, the system processes these nodes for each target generated by the input node, a topic that is discussed in greater detail in section 6.3.3.

The challenge lies in identifying which functions are inside the loop, and which functions simply provide an argument from outside the loop. To exemplify Code Fragment 6.1 shows the function `filter_list` that returns a list based on another, but without any numbers below a minimum bound. Code Fragment 6.2 shows the mapping for `append` which is called on the empty list defined outside the for-loop. When executing the constructed pipeline it should not repeatedly execute the `list_1` node that creates an empty list as this will override the list stored in the `append_1` node. Only calls used with **mapFrom** are considered inside the loop and contain a reference to the input node. Figure 6.5 shows the fully visualized pipeline, where nodes inside a for-loop are grouped by color.

```python
def filter_list(numbers, min):
    filtered = []
    for number in numbers:
        if number >= min:
            filtered.append(number)
    return filtered
```

Code Fragment 6.1: Filtering a list of numbers with `filter_list`

```
:for_1Pipeline a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFrom [ fnoc:functionOutput :listOutput ;
                fnoc:constituentFunction :list_1 ] ;
            fnoc:mapTo [ fnoc:functionParameter :appendParameterSelf ;
                fnoc:constituentFunction :append_1 ] ;
            fnoc:mapCondition [ fnoc:functionOutput :gteOutput ;
                fnoc:constituentFunction :gte_1 ] ] .
```

Code Fragment 6.2: Composition of `filter_list`

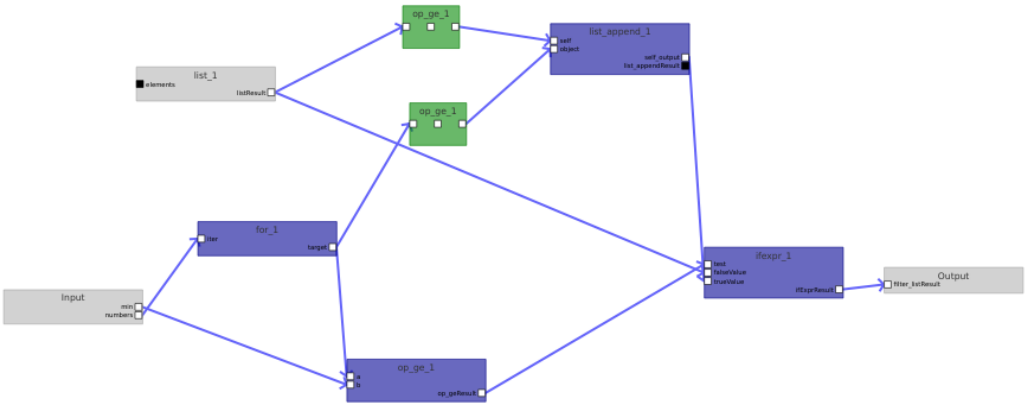

Figure 6.5: Filtering a list with a minimum bound

Including Internal Nodes of Used Functions

Several considerations must be taken into account when including the internal nodes of a function. Since the FnO Composition describes the function independently, calling the function multiple times results in the same nodes being used for each call. To ensure that each call can provide input to these internal nodes and produce the appropriate output for other nodes, each call should have two linking nodes: one for connecting the input and another for connecting the output. Figure 6.6 illustrates a pipeline that returns two sorted strings using the `my_sort` function. When multiple nodes are expanded that use the same internal nodes a problem arises during execution. This is further discussed in section 8.1.2.

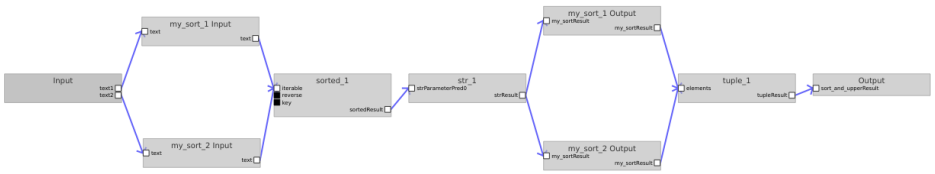

Figure 6.6: Including internal nodes of used functions

The tool allows users to choose whether to display the internal body of a function or not. Consequently, the system must manage both a node representing the function call and a collection of nodes representing its body. The system must also enable quick transitions between these representations. Figure 6.7 illustrates the same pipeline as before, with only one node showing its internal structure. To facilitate this, the concept of opening and closing nodes is introduced. A closed node is not visualized and is not considered during pipeline execution. When encountering a function with a composition, both the call node and all internal nodes with linking nodes are created and added to the pipeline. Initially, the call node is open while the other nodes are closed.

When closing a pipeline, all internal nodes (including the linking nodes), are closed using a bottom-up approach. Before a terminal is closed, it first closes its own connection and the connection to the connected terminal. Once all terminals of a node are closed, the node itself is closed. Conversely, opening a pipeline is performed with a top-down approach. After a node is opened, it then opens all its terminals. When opening a terminal, the system attempts to open its connection, which is only possible if the connected terminal is also open. Therefore, the last connected terminal to be opened triggers the opening of the connection.

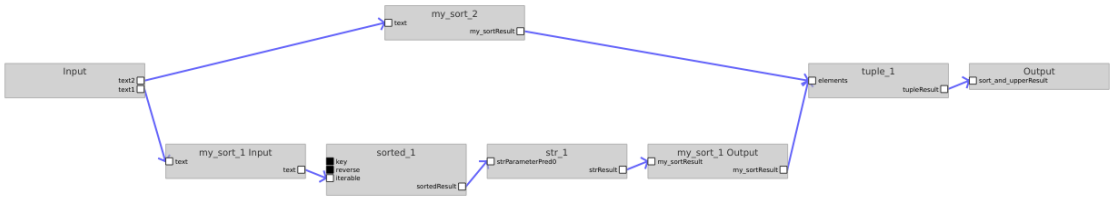

Figure 6.7: Opening or closing a pipeline representation

To ensure these mechanisms work effectively, three key aspects must be tracked: the internal nodes of a function, each function call that uses these internal nodes through linking nodes, and the number of function calls currently displaying their internal body. The last aspect is crucial to ensure that internal nodes are only closed when no other calls are using the internal nodes. The process of creating a node with a FnO Composition, as detailed in Figure 6.8, involves several steps. First, link nodes are created as instances of the special node type `LinkNode`, whose sole function is to propagate inputs to the pipeline as arguments. If the internal nodes are not already present in the pipeline, they are created using the FnO Composition. During this creation, other pipelines and/or for-loops can be recursively added. Connections are then created using FnOC CompositionMappings; if a mapping is made using the parameter of the composition's function, it is connected to the terminal of a linking node. The internal nodes are subsequently mapped to the function URI, ensuring they are created only once and can be easily retrieved. The function call then subscribes to the pipeline, storing its input and output nodes and referencing the relevant internal nodes. The process concludes by closing all input, output, and internal nodes. The linking nodes are properly connected to the internal nodes, but they must also be mapped to external terminals to receive external data. To achieve this, each time a nested function is handled, the outside function stores the linking nodes. When the outside function encounters a call that has linking nodes while iterating the FnOC CompositionMappings, it creates a connection to both the terminals of the call node and the link nodes.

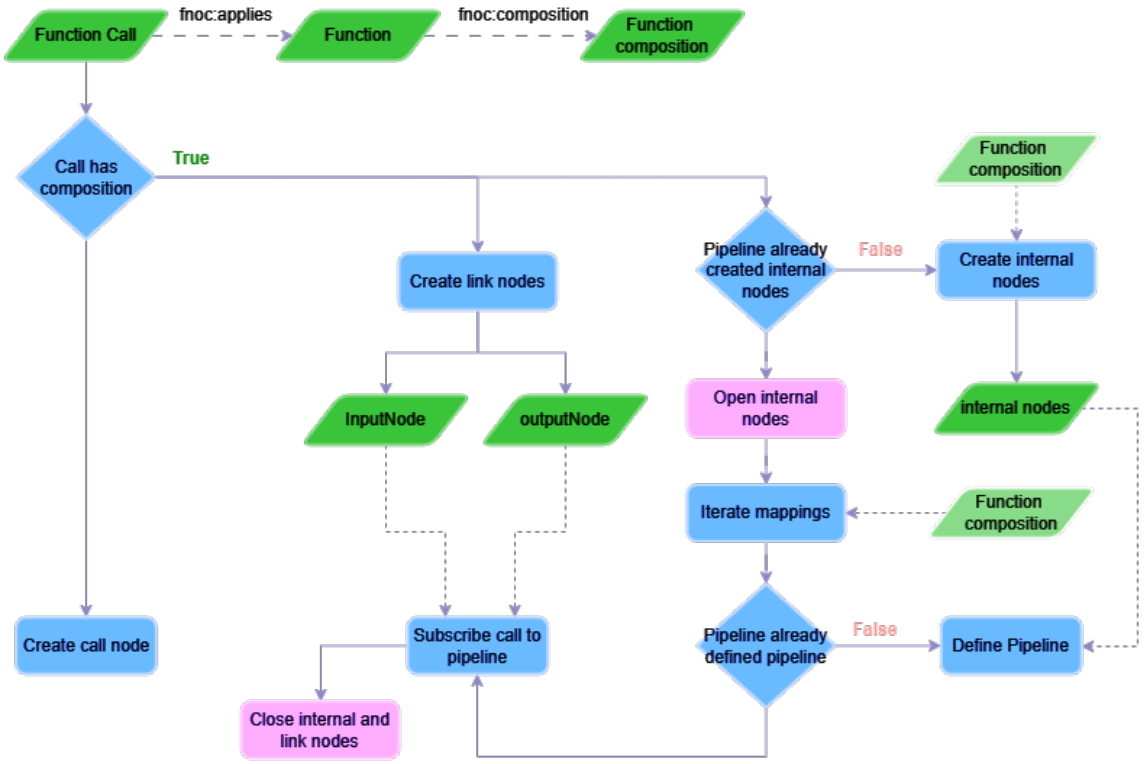

Figure 6.8: Creating a node with a composition

6.3 Processing a Generated Pipeline

With the `Pipeline` instance set up, it can now be used for data processing while ensuring provenance. First, this section will examine in detail how a single `FunctionNode` processes its inputs to generate outputs. The processing of specialized nodes, such as the `LinkNode`, will also be explained. Next, an overview of how the `Pipeline` orchestrates the processing of all nodes will be provided. Finally, the discussion will cover how the structures introduced in previous chapters enable the execution of for-loops and conditional statements.

6.3.1 Processing a Node

Section 6.2.1 explains how a node contains a function object based on the described implementation, along with a subgraph containing relevant FnO descriptions to offer structural information. Processing is initialized with the `process` method. To start, it checks if it is a method call by looking for a self parameter. If present, the corresponding value is loaded from the terminal and the method object is acquired using `getattr`. This ensures that nodes who failed to derive the function object are still able to be executed. Using the method object instead of the function object is also necessary to work with the `mlflow` library. More on why this is in section 6.5.

When executing a function node, terminal values need to be provided to the function object in a correct order. The FnO Mapping helps determine if a terminal corresponds to a variable parameter, keyword argument, or positional argument. Algorithm 5 outlines this calculation. Variable parameters are multi-value terminals that store a dictionary, th variable positional terminal values using indices as keys. Arguments are provided by unpacking in this specific order: `args`, `vargs`, `keyargs`, and `vkeyargs`. The node returns a dictionary containing the output URI and its corresponding return value. Additionally, for methods, the dictionary includes a URI mapping to the object the method was called on.

Algorithm 5 Calculating correct order of arguments with an FnO Mapping

Require: Node to be processed `node`

```
args ← [], vargs ← [], keyargs ← {}, vkeyargs ← {}
for each terminal in node do
    param ← terminal URI
    value ← terminal value
    if param is a varpositional parameter then                ▷ Multi-value terminal
        for i, val in value do
            vargs[i] ← val
        end for
    else if param is a varkeyword parameter then              ▷ Multi-value terminal
        vkeyargs ← value
    else
        index, prop ← FnO ParameterMapping for param
        if prop is not None then
            keyargs[prop] ← value
        else if index is not None then
            args[index] ← val
        end if
    end if
end for
if node represents a method then
    if 'self' in keyargs then
        delete keyargs['self']
    else
        args ← args[1:]
    end if
end if
```

Processing Specialized Nodes

This section will explore the processing of nodes created from standard functions. When creating a node, the first step is to check if the function URI corresponds to a standard function description. If it does, the corresponding specialized node is created. If it does not, a standard `FunctionNode` is created.

Containers are generated using either a `ListNode`, `TupleNode`, or a `DictNode`. Each of these nodes features a multi-value input terminal with a dictionary indicating the position of each value with its keys. In the case of a dictionary, the indices themselves are not used; only the pairs stored in the values are relevant. As mentioned earlier, a `LinkNode` simply propagates its values from input terminals to the corresponding output terminals. The `IfNode` implements the functionality of an if expression. It evaluates the value of the test terminal to determine which of the two other terminals' values to return. A `SliceNode` is responsible for creating a slice object. The `AttributeNode` is used when accessing an attribute of an instance. This operation is straightforward and involves using the `getattr` function during processing. If the instance is a string, it might represent the name of a module. In such cases, the node first attempts to obtain the module object by importing it before fetching the attribute.

6.3.2 The `Pipeline` Class as Orchestrator for Processing

A `Pipeline` instance can be executed by calling `process`. At its core, the `Pipeline` oversees the sequential processing of nodes while handling the flow of data between them. It begins by setting input data for the `inputNode` and then determines the order of operations. During processing, the `process_node` function constructs input dictionaries, accommodating multi-value terminals and supporting unpacking. Error handling ensures robustness throughout execution. Overall, the `Pipeline` serves as a reliable conductor, ensuring smooth data processing.

The `Flowchart` class provides the `processOrder` function for determining the order of operations necessary to process nodes, ensuring that all dependencies are respected. Algorithm 6 outlines the approach. The returned order of operations should look like `[('p', node1), ('p', node2), ('d', terminal1), ...]`, where each tuple specifies either (p)rocess this node or (d)elete the result from this terminal. The algorithm first constructs two dependency graphs: one mapping each node to the nodes it depends upon and another mapping each terminal to the nodes that recieve input from it. Using topological sorting, the code establishes a processing order for the nodes. It then constructs a list of operations to process each node in this order. To manage resource cleanup efficiently, the code identifies the appropriate points at which it is safe to delete terminal values, ensuring these deletions occur only after all dependent nodes have been processed. The resulting sequence of operations ensures an efficient and dependency-aware execution of the pipeline, balancing processing tasks with resource management.

Algorithm 6 Determine Order of Operations for Processing Nodes

Require: Set of nodes `nodes`
Ensure: Order of operations `ops`

```
deps ← {}                                              ▷ Dependencies of each node
tdeps ← {}                                             ▷ Dependencies of each terminal
for each node in nodes do
    deps[node] ← nodes which node depends upon
    for each terminal in output terminals of node do
        tdeps[terminal] ← nodes which depend on terminal
    end for
end for
Compute topological sort order using deps
ops ← []
for each node in order do
    Append ('p', node) to ops                          ▷ Process node
end for
dels ← []
for each terminal, nodes depending on terminal in tdeps do
    lastInd ← 0
    lastNode ← None
    for each node in nodes do
        ind ← order.index(node)
        if lastNode is None or ind > lastInd then
            lastNode ← node
            lastInd ← ind
        end if
    end for
    Append (lastInd+1, terminal) to dels
end for
Sort dels in descending order based on index
for each i, terminal in dels do
    Insert ('d', terminal) at index i in ops           ▷ Delete terminal
end for
return ops
```

Algoritm 7 shows how the `Pipeline` orchestrates the sequential processing of nodes. It manages all generated values by mapping them to their respective output terminals in the `data` variable. It traverses through the processing order, crafting an input dictionary for each node that links input terminals to the values they must receive. it accesses each input terminal of the node, identifying the terminal that provides input and looking in `data` for the generated value. In cases where an input terminal is flagged as multi-value, indicating it receives input from multiple terminals, the algorithm constructs an index map using the indices stored in each connection item. To accommodate unpacking, the algorithm checks if the connection item has a `fromIndex` set, utilizing this index to subscript the generated value appropriately. Finally, the constructed input dictionary is passed to the `setInput` method to assign values to all input terminals of the node. This action initiates the processing of the node and returns the value for each output. These are then stored in the `data` variable for use by downstream nodes. Delete operations simply remove the terminal and its generated value from `data`.

When the output node is encountered in the list of operations, the generated output is stored immediately. The processing does not halt at this point; it continues to process any remaining endpoints in the pipeline. Once all nodes have been processed, the final returned values are serialized and saved as a binary file using the `pickle` library. This approach ensures that the generated values can be efficiently reused by other pipelines, facilitating data sharing across different pipeline executions.

6.3.3 Enabling For-Loops

Section 6.2.3 details the setup of for-loops within the `Pipeline`. Each node inside the loop holds a reference to the input node that enables iteration. When a `ForNode` is processed, it creates an iterator based on the `iterParameter`. Every subsequent time the node is processed, it will return the current iterator item and try to get the next. The first time a `ForNode` is encountered, it is added to `for_loops`, which contains all active loops.

Algorithm 8 illustrates the necessary handling required before processing a node to enable for-loop execution. Before a node is processed, it verifies whether the node is part of an existing loop recorded in `for_loops`. If the node is part of such a loop, it appends the node to the corresponding entry in `for_loops` and delays its execution. This loop is then checked to see if it should be executed by calling `check_loop`. This method increments an internal counter each time it is called, and once this counter matches the expected number of nodes within the loop body, it will commence execution. The loop continues to iterate as long as there are items to process (attribute `has_next` is True). During each iteration, all nodes associated with the loop are processed, followed by the loop node itself, advancing the iterator. After the loop completes all iterations, the loop entry is removed from `for_loops`. If the node is not part of any loop, it is processed immediately. Processing is done with Algorithm 7 which is expanded to store loops in `for_loops`.

Algorithm 7 Construct Input Value Dictionary

Require: List of input terminals `ins`, dictionary of generated values `data`
Ensure: Input value dictionary `args`

```
args ← {}
for each input terminal inp in ins do
    Retrieve list of input terminals connected to inp
    if number of inputs is 0 then
        Continue to next iteration
    else if inp is multi-value then
        indexMap ← {}
        for each connected input terminal i do
            for each (fromIndex, toIndex) in connection between i and inp do
                value ← data[i]
                if fromIndex is not None then
                    value ← value[fromIndex]
                end if
                indexMap[toIndex] ← value
            end for
        end for
        args[inp] ← IndexMap
    else
        i ← input terminal
        value ← data[i]
        Retrieve fromIndex from connection between i and inp
        if fromIndex is not None then
            value ← value[fromIndex]
        end if
        args[inp] ← value
    end if
end for
return args
```

This method ensures that the dependent nodes of every node within the for-loop body are processed and ready to provide input. Notably, when calculating the order, every node in the for-loop body is marked as dependent on the `ForNode`. This dependency ensures that nodes without a connection to the target output of the `ForNode` are not processed before the loop is correctly initiated.

Algorithm 8 Process Node with Loop Handling

Require: Node to be processed `node`, dictionary of generated values `data`, dictionary containing active for-loops `for_loops`

```
if node has a loop l and l is in for_loops then
    Append node tp for_loops[l]
    if l encountered all nodes then
        while iterator of l has a next target do
            for each node n in for_loops[l] do
                Process node n with data                    ▷ Algorithm 7
            end for
            Process loop l with data                        ▷ Algorithm 7
        end while
        Remove l from for_loops
    end if
else
    Process node with data                                  ▷ Algorithm 7
end if
```

6.3.4 Conditional Connections

Conditional mappings are implemented with the `Condition` class. This class, when processed, behaves like a `LinkNode` by propagating its values. However, it includes a `check` method that evaluates the condition output against the specified strategy. Before processing a node, the system first checks if any of its connections involve a `Condition` node and whether any of the `check` functions return `False`. If this is the case, the node will not be processed. Currently, a `Condition` node must be added to every connection along the subsequent path to prevent each further node from being processed. Therefore, the idea that a condition acts as a 'gateway' is not entirely accurate. Instead, it is a collection of `Condition` nodes that collectively block a path. Further research is needed to execute conditional statements more efficiently.

6.4 Capturing & Describing Data Transformations

A `Pipeline` uses an instance of the `ProvenanceGenerator` to bring all metadata concerning node execution together. This stores a `PipelineGraph` and provides functionality to add executions to it. Before discussing how it collects provenance of data transformations during execution, it is essential to further establish the concept of data transformations. This section explores how these transformations can be described using PROV-O.

6.4.1 PROV-O Entities, Activities & Agents

A data transformation refers to the operation or series of operations applied to raw input data to convert it into a format more suitable for analysis. The three principal components of data transformations within a preprocessing pipeline include the input data, the transformation function, and the output data generated. These components provide comprehensive insights into the processing of the provided data. PROV-O is particularly effective for describing how these components interconnect to delineate a complete data transformation.

A PROV-O Entity is defined as any item that has provenance; essentially, anything that can be tracked, undergoes changes over time, or is derived from other entities. A PROV-O Activity is how new entities get created, often using other entities in the process. A PROV-O Agent holds responsibility for the existence of an entity or the occurrence of an activity. This indicates accountability in the context of data provenance, highlighting the roles of individuals, systems, or organizations in creating or modifying data.

Section 4.1.2 outlined the method for describing executions using FnO with Code Fragment 4.3 as an example. To ensure that this description is accessible and interpretable by machines through PROV-O, it needs to be expanded. In this context, a data transformation is identified as a PROV-O Activity that employs raw input data and a FnO Function (both categorized as entities) connected via the **used** predicate. This activity generates an output, which is also recognized as an entity and is linked to the activity through the **wasGeneratedBy** predicate. The FnO Implementation of the function acts as the PROV-O Agent of the transformation, linked using the **wasAssociatedWith** relation. This association indicates that the implementation is responsible for the generation of the output in this data transformation. Finally, the output and its implementation are connected with the **wasAttributedTo** predicate. This concept is introduced in [10] and visualized in Fig 2.3. The resulting extended graph is displayed in Code Fragment 6.3. By incorporating the predicates **startedAtTime** and `endedAtTime`, performance metrics can be easily evaluated.

6.4.2 Provenance of a Processed Node

Following execution, a node leverages its input and output terminal values, along with its function description and implementation, to generate PROV-O annotations describing the execution process (explained previously). The `Pipeline` achieves this by calling the `provenance` method on the node after execution. This method provides the `Pipeline`'s `ProvenanceGenerator` as an argument, which the node can use to add any relevant execution metadata.

```
:sortAlph a fno:Function, prov:Entity .
:sortAlphPythImp a fnoi:PythonFunction, prov:Agent .

:sortTransformation a fno:Execution, prov:Activity ;
    prov:used :sortAlph, :input ;
    prov:wasAssociatedWith :sortAlphPythImp ;
    prov:qualifiedAssociation [ a prov:Association ;
        prov:agent :sortAlphPythImp ;
        prov:hadPlan :sortAlph ;
        prov:hadRole :implementation ] ;
    prov:startedAtTime "XXX"^^xsd:dateTime
    prov:endedAtTime "YYY"^^xsd:dateTime

:output a prov:Entity ;
    prov:wasGeneratedBy :sortTransformation
    prov:wasAttributedTo :sortAlphPythImp
```

Code Fragment 6.3: Provenance information for Code Fragment 4.3

All generated values used must be represented in RDF. Section 5.2.4 details how this is accomplished using the `Literal` class. However, when describing terminal values, more complex instances can be encountered. For example, Code Fragment 6.4 shows a `DataFrame` storing TedTalks represented as a `Literal`. The string representation formats a small part of the first five rows. While this approach introduces a significant loss of information, providing an RDF format that supports two-way conversion is challenging because each instance is structured very differently from another.

```
"""      main_speaker         details
0        Alexandra Auer       The intangible effects of walls ...
1        Elizabeth Gilbert    It's OK to feel overwhelmed. ...
2        Butterscotch         Accept Who I Am Firing off her ...
3        Ethan Lisi           What it's really like to have ...
4        Daniel Finkel        Can you solve the sea monster ...
...      ...                  ...
[4466 rows x 2 columns]"""^^:DataFrame
```

Code Fragment 6.4: Literal representation of a `DataFrame` storing TedTalks

```
:Int_1 a prov:Entity ;
    prov:wasGeneratedBy :amax_1
    rdf:value "9"xsd:integer .
```

Code Fragment 6.5: Generated output of an `amax` function call

By invoking the `execute` method on the `ProvenanceGenerator`, its graph is expanded with a node execution. Alongside providing the URI of the function, implementation, and corresponding mapping, it also requires information about the used and generated values. Code Fragment 6.5 illustrates the description of an integer instance obtained from an array using the `amax` function. To prevent ambiguity, instances use an enumerated name of its type. No entity should be created for instances that are used as constants. The `Literal` representation can be directly used. To enable reuse of outputs by other executions, the `ProvenanceGenerator` maps each output to the URI of the corresponding parameter within the context of the function call. When a node utilizes the outputs of another, the generator will employ this store to reuse the previously described output rather than creating a new instance with the exact same value. Additionally, nodes store performance metrics, including the start and end times of the execution. These metrics are also incorporated into the execution graph.

The tool enables users to bypass nodes, which will still execute but won't be incorporated into the provenance document. Nodes that depend on outputs of bypassed nodes must create a new entity, as no output was previously described. The `LinkNode`, `IfNode`, and `Condition` should never be included in a provenance document, as they simply propagate values. When `provenance` is called, they will simply link their outputs to the generated outputs of their inputs without adding an execution.

6.5 Capturing & Describing Machine Learning Operations

All previously discussed functionality allows for a semantic description of any Python pipeline. However, the tool was specifically developed with ML pipelines in mind. To enhance transparency, the MLSchema ontology is employed to further semantically annotate the ML operations that occur. These operations can be automatically captured using MLFlow. After establishing the concepts introduced within MLSchema and their connection to PROV-O, the use of MLFlow and its Python library is explained in greater detail.

6.5.1 Combining MLSchema and PROV-O for Enhanced Transparency

Figure 6.9 presents an overview of MLSchema. At the center of this ontology is the Run class, which represents the execution of a pipeline involving ML operations. A Run compiles a model based on a specific algorithm and trains it on a given dataset. The outcome includes a trained model and any evaluations generated during testing. MLSchema provides RDF classes for all these concepts, along with the necessary predicates to accurately convey their relationships. A Run shares many similarities with an FnO Execution. It is PROV-O Activity that utilizes hyperparameters, an algorithm, and input data to produce a model and corresponding evaluation metrics. Similar to an execution, the implementation serves as a PROV-O Agent in this context.

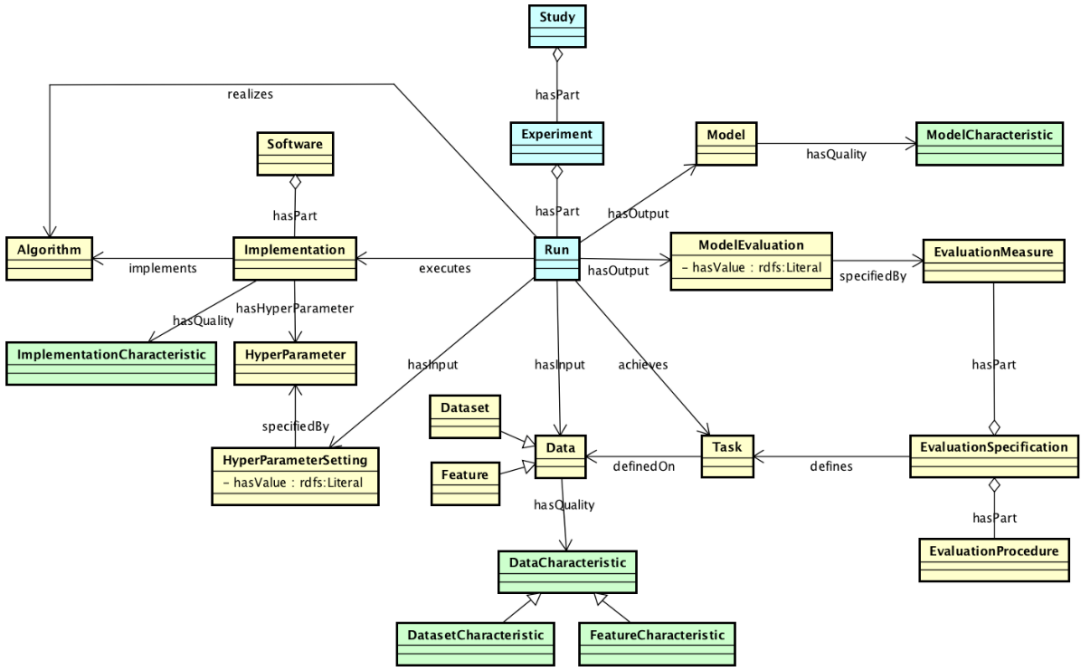

Figure 6.9: The MLSchema Ontology

The `mlflow` library provides an API for working with MLFlow, allowing easy logging and retrieval of parameters, metrics, and artifacts. The `mlflow.autolog()` function enhances this capability by automatically capturing metadata from popular ML libraries (like TensorFlow, Keras, and scikit-learn) during training. Before execution, the `Pipeline` instance will enable auto logging. All metadata is stored in a local directory structure and can be accessed via the `mlflow.Run` instance, which is created by the `autolog` function. Both hyperparameters and metrics contain their names and values. A model is stored as an artifact which contains the used ML library and trained model instance.

To simplify, this paper assumes that only one model is trained in a pipeline. For the MLSchema Implementation and Model, the PROV-O Entities generated from the FnO Execution that trained the model are used. The implementation instance is identified by the `self` predicate, while the model instance is identified by the `self_output` predicate. To handle potential conflicts where functions with the same name are used in different contexts, each execution instance is type-checked. The type derived from the artifact is transformed into RDF and compared to the types of the found instances. If the types match, we can safely assume that this FnO Execution used the implementation to produce a trained model. Code Fragment 6.6 shows the query used to fetch every URI.

```
SELECT ?imp ?impval ?model ?modelval WHERE {
    ?exe a fno:Execution ;
        fno:uses ?mapping ;
        :self ?imp ;
        :self_output ?model .
    ?mapping fno:methodMapping ?method .
    ?method fnom:method-name "<name of train function>" .
    ?imp rdf:value ?impval .
    ?model rdf:value ?modelval .
}
```

Code Fragment 6.6: Query to get the generated model and its implementation

One limitation is the hardcoded search for the `fit` function when encountering a `tensorflow` or `sklearn` model, chosen as proof of concept. Extending the tool to support models from other libraries is feasible; options include allowing users to provide the function name or using semantic reasoning to infer the training function. Additionally, the broad semantic meaning of the MLSchema Algorithm class makes it challenging to determine the actual algorithm used. Currently, it is defined by the class name of the model instance. Future research could explore methods for effectively extracting the algorithm from a model instance, thereby enhancing our understanding of the model's behavior and enabling more insightful analysis.

Using `autolog`, the system only captures the shape of input data, providing limited insights. Additionally, the system relies on the FnO Execution of the training function to obtain input data, which is then described using the MLSchema Data class. However, the input format varies depending on the library used. Presently, input capture is supported only for the `tensorflow` and `sklearn` libraries. Due to the current inability to recreate the dataset using the `Literal` representations, no analysis can be performed to extract data characteristics such as mean and standard deviation.

7

User Manual

To ensure comprehensive accessibility, a specialized tool was developed featuring a user-friendly interface that enables the conversion of Python files, visualization of described pipelines, and the provision of input for executing replicated pipelines. This tool is designed for developers with a foundational understanding of Python code and RDF. It exclusively supports the generation of RDF documents for use by other applications and does not offer querying options for further inspection. This chapter presents an in-depth user manual for the tool, primarily focusing on the user interface. For installation, see the project repository.

7.1 User Interface

7.1.1 Account Tab

Upon starting the application, you will be directed to the **Account** tab, shown in Figure 7.1. In this section, you are required to enter your name and email address to establish provenance for the generated graph. Additionally, at the bottom of this tab, you have the option to define a new prefix base for the generated document. If no prefix base is specified, it defaults to 'http://www.example.com#'. To navigate to the other tabs, simply click on any of the tabs highlighted by the red border.

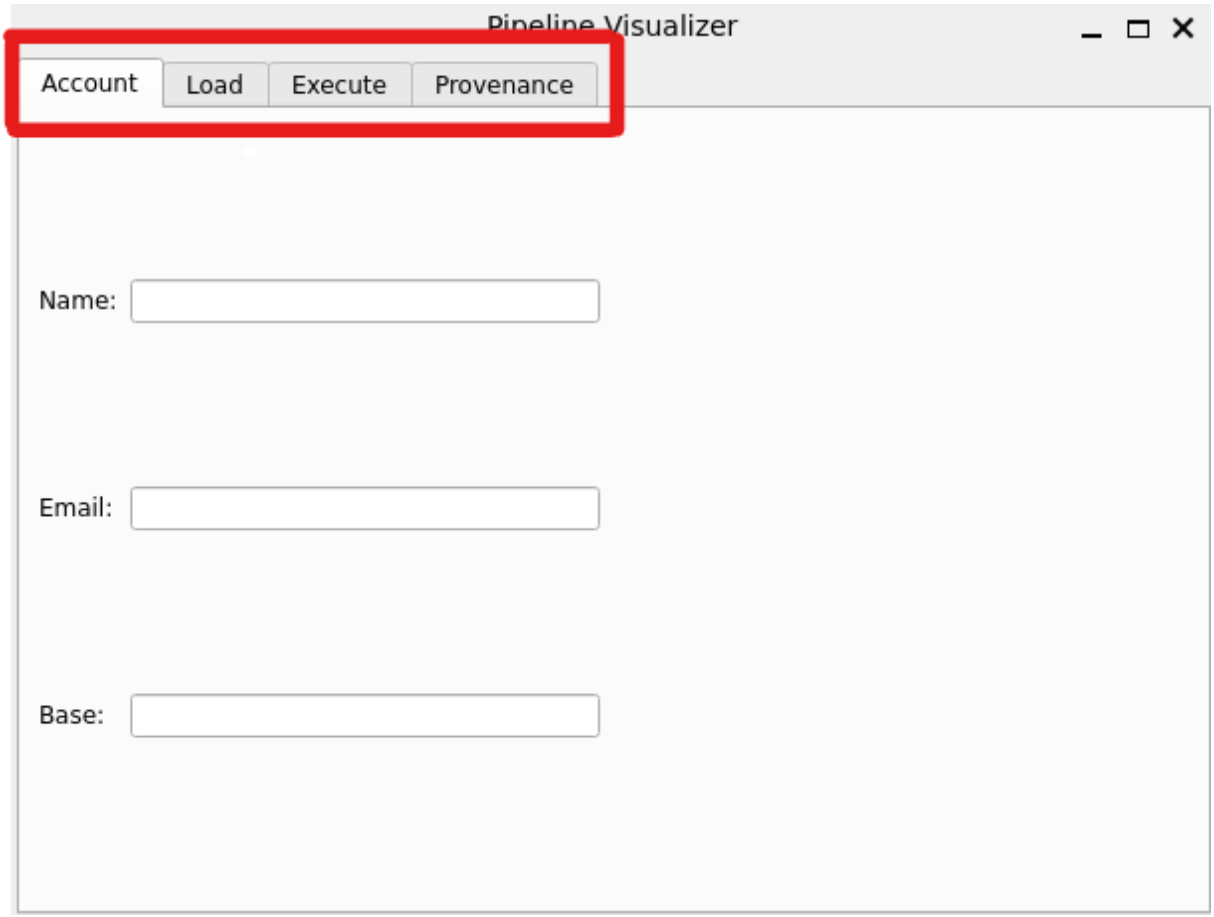

Figure 7.1: The **Account** tab

7.1.2 Load Tab

The Load tab enables users to import a Python file containing the source code of a pipeline or a Turtle file with a previously described pipeline. Figure 7.2 illustrates the Load tab with the widgets for loading documents highlighted. The widget at the top is used to load a Python file. Once the file is loaded, the source code will be displayed in the left textbox. The dropdown menu, shown in Figure 7.3, allows you to select the desired function to be described. Upon pressing the 'Load' button, a graph is generated. When the generation is complete, the serialized graph is shown in the right textbox. Pre-generated graphs are directly displayed in the right textbox. Each textbox includes a search widget at the top, highlighted in Figure 7.4. By entering a search term and clicking 'search', all occurrences of the term will be highlighted in yellow.

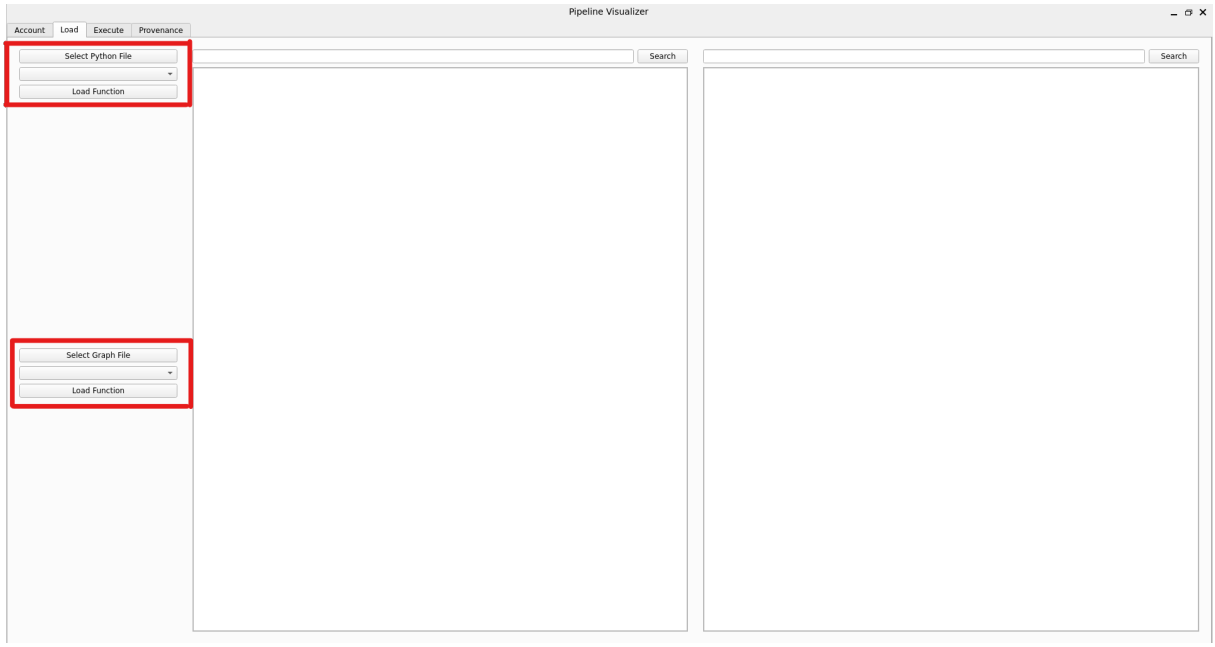

Figure 7.2: The **Load** tab

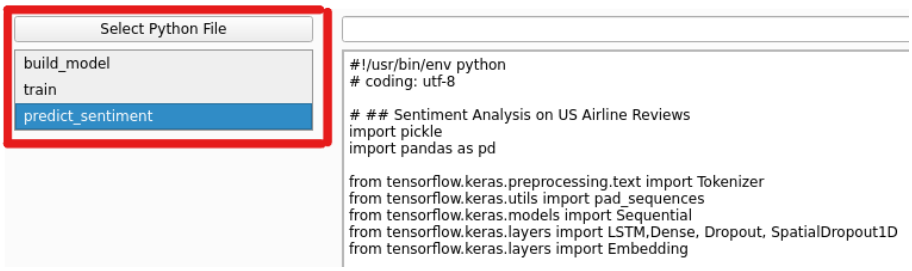

Figure 7.3: Dropdown menu to pick function to be described

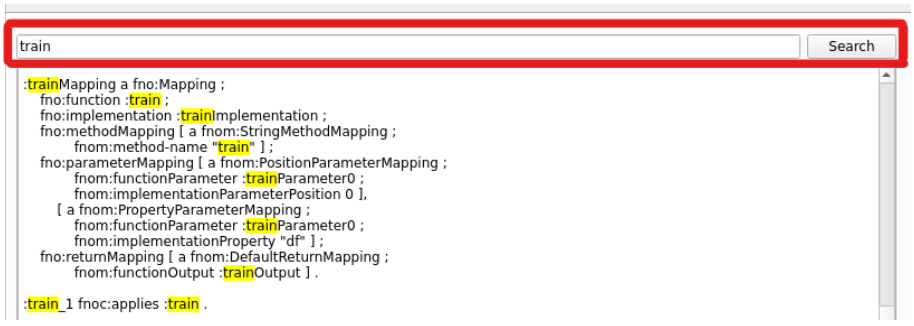

Figure 7.4: Search a text by higlighting

7.1.3 Execute Tab

When a graph is displayed in the **Load** tab, a `Pipeline` instance is automatically generated and visualized in the **Execute** tab. On the right, you can see the visualized pipeline along with two `DockArea` widgets, as shown in Figure 7.5. A `pyqtgraph DockArea` can be stretched or minimized by dragging its top edge. The 'Hover Info' section displays the value of the terminal you hover over with the mouse. Selecting a node by clicking on it will display the subgraph in the 'Selected Node' section, containing all relevant information for that node. If the layout algorithm does not present the nodes in a clear manner, you can rearrange the nodes by dragging and dropping them to improve the view.

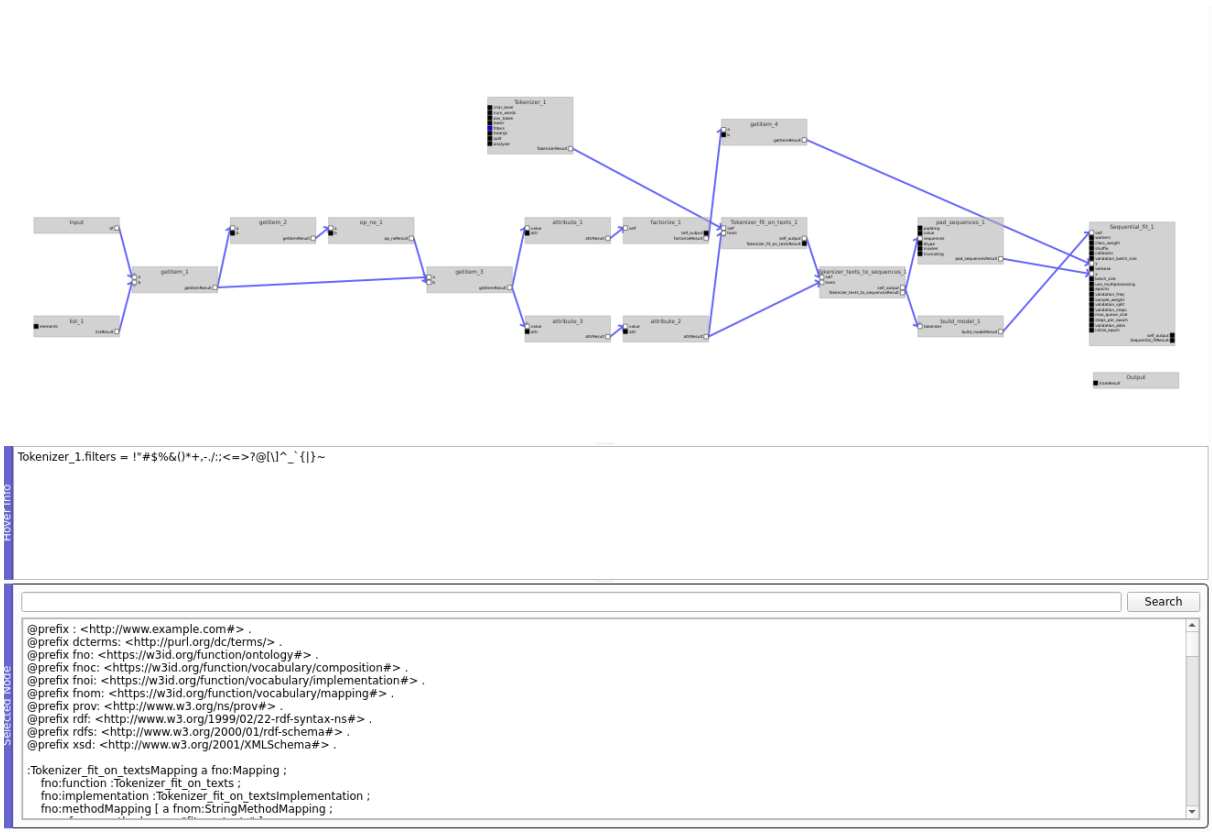

Figure 7.5: The `Pipeline` view

On the left side, you'll encounter the input widget alongside an overview of all nodes. This widget, as depicted in Figure 7.6, serves as an interface facilitating input provision for the execution of the `Pipeline` instance. It accommodates various data types such as `string`, `integer`, or `float`, each equipped with a validator to ensure accurate conversion. However, for other instance types, such as complex objects, users are required to upload pickle files via a file dialog window. Expanding the tool's functionality could involve incorporating additional input methods tailored to commonly used file formats. For instance, when anticipating a `DataFrame`, users might be provided with the option to upload a `.csv`-file directly.

Upon pressing 'process', the pipeline undergoes execution utilizing the provided input. Any nodes encountering errors during this process will be visually highlighted in red. Users can select these nodes to view the traceback of the exception, offering insight into the nature of the encountered error.

The node overview, illustrated in Figure 7.7, furnishes a comprehensive list of all included nodes. Each node is accompanied by a checkbox enabling users to select whether it should be included in the provenance document or not. Nodes equipped with internal pipelines feature an additional button allowing for the expansion or concealment of the internal nodes.

Figure 7.6: The Input widget

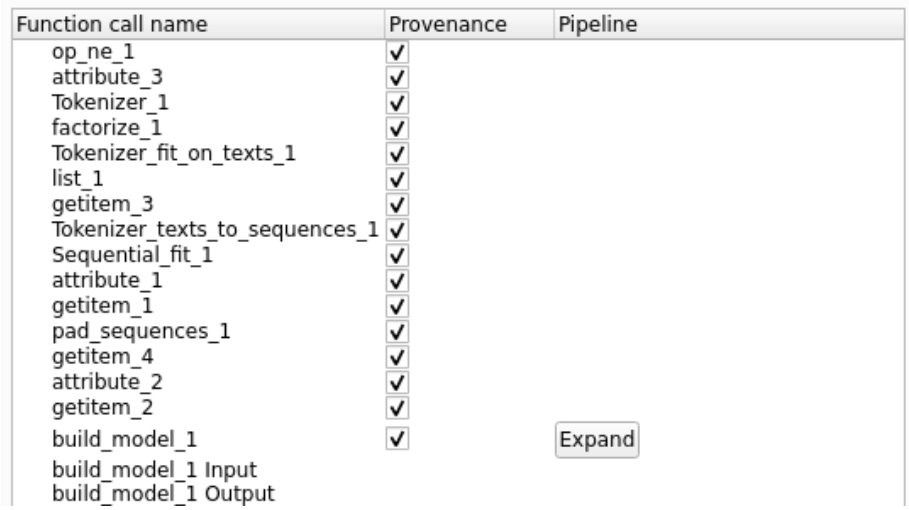

Figure 7.7: The `FunctionNode` overview

7.1.4 The Provenance Tab

After pipeline execution, the **Provenance** tab displays a graph with provenance information. Clicking 'save' in the bottom right lets you save the execution details and pipeline description into a turtle file. More RDF formats may be supported in the future. See Figure 7.8 for the tab layout.

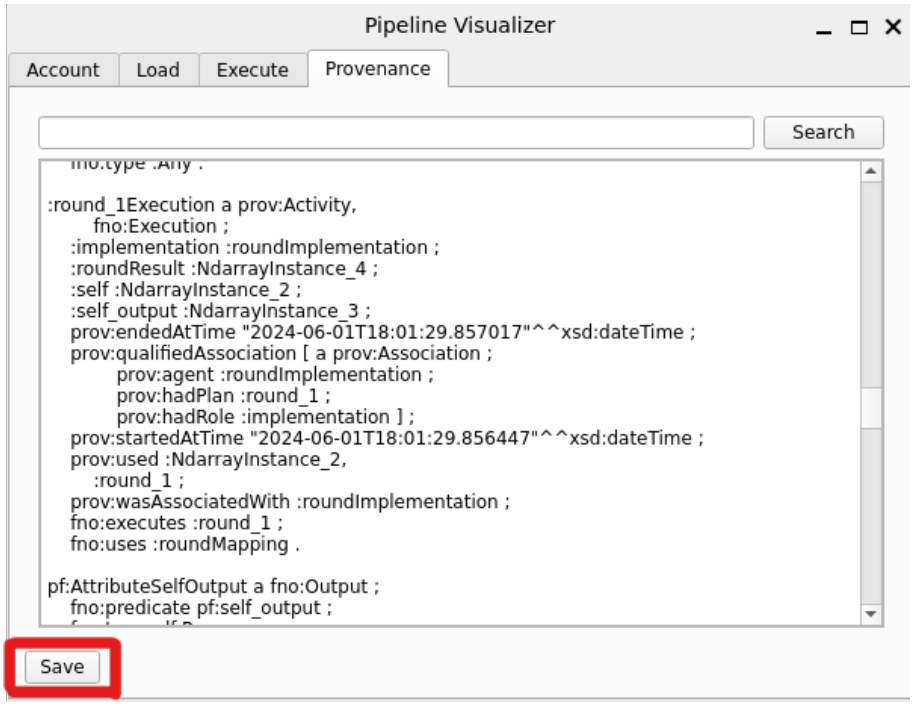

Figure 7.8: The **Provenance** tab

8

Evaluation

This chapter focuses on evaluating the developed tool by applying it to annotate a variety of machine learning projects sourced from the internet. This Chapter will begin by addressing the current know limitations. Section 8.2 will analyze the behavior and inspect the correctness of the generated graphs. Section 8.3 will assess the performance of graph generation and examine the impact of provenance extraction on execution time. Table 8.1 presents an overview of the ML projects used for testing purposes. This paper does not include fully generated RDF documents due to their large size. However, they are available in the project repository for access, together with the corresponding source code.

Table 8.1: Overview of tested machine learning projects

Name	Library	Model	Description
Sentiment Analysis [28]	TensorFlow	LSTM	Model trained to classify the sentiment of a tweet as either *positive* or *negative*
Digit Recognition [29]	Keras	CNN	Model trained to classify handwritten digits
Iris Flower Classification [30]	Scikit-learn	SVC	Model trained to classify flowers based on numerical features
TedTalk Reccomendations [31]	TensorFlow	Vectorizer	Text vectorizer with the Pearson correlation metric to compute similarity.
IPL Score Prediction [32]	TensorFlow	RNN	Model trained to predict score based on the current match situation for an IPL cricket match
Music Genre Clustering [33]	Scikit-Learn	KMeans	Clustering music genres from Spotify
Stock Price Prediction [34]	TensorFlow	LSTM	Model to predict the source code based on financial data of the previous 5000 days

8.1 Discovered Limitations and Breaking Conditions

This paper serves as a proof of concept and acknowledges its limitations in addressing all possible scenarios. Prior to evaluating the tool on machine learning pipelines, potential breaking conditions will be discussed. These conditions entail identifying erroneous behaviors and proposing future solutions wherever feasible.

8.1.1 Early Stopping with Return Statements

A return statement in a conditional statement facilitates early stopping. The system correctly merges return statements encountered in both conditional bodies. However, it fails to merge return statements from independent clauses or the return statement at the end of the pipeline. Code Fragment 8.1 provides the trivial function `early_stop`. The generated graph is visualized in Figure 8.1. The if expressions are not merged; instead, they all point to the pipeline output, which means they override each other depending on the process order. When the `stop_index` is zero and the corresponding if expression is processed, the output will be set to zero. If another `IfNode` is processed, it will set the output to `None`, overriding the correct output. Simply stopping further executions when the output receives input will not work because, for example, if another `IfNode` is processed first, the correct if expression won't even have a chance to provide output.

```
def early_stop(stop_ind: int):
    if stop_ind == 0:
        return 0

    if stop_ind == 1:
        return 1

    if stop_ind == 3:
        return 3

    return None
```

Code Fragment 8.1: Function definition of `early_stop`

The issue arises because, when encountering a return statement within a conditional, it is immediately mapped to an if expression based on the current test, without considering future return statements. A potential solution could be to delay handling these statements until the end, then merge all statements by coupling multiple if expressions together.

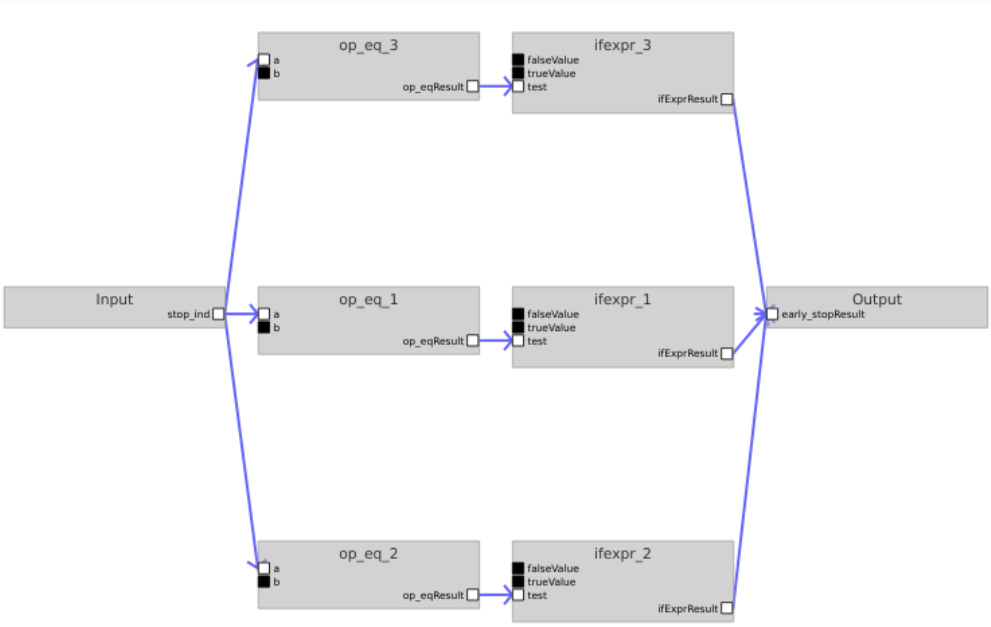

Figure 8.1: Visualization of the function `early_stop`

This problem reveals a deeper issue with the current handling of conditional statements. The decision to incorporate conditional mappings was made later in the development process, and it was combined with the already implemented if expressions. Currently, if a node encounters a connection with an unsatisfied condition, it will not process. However, an `IfNode` with one satisfied and one unsatisfied condition still needs to be processed. Consequently, it was decided not to use conditional statements with if expressions. The concept of conditional statements requires further development to handle such situations effectively, but due to timing constraints, these improvements were not implemented.

8.1.2 Multiple Nodes based on the Same Composition

A complication arises when processing a pipeline that contains multiple nodes based on the same composition. When more than one node is expanded, multiple `LinkNode` instances provide input to the same input, and multiple `LinkNode` instances receive output from one terminal. This issue was illustrated in Figure 6.3.

The current algorithm for calculating processing order does not consider that internal nodes should be processed as a single entity. Consequently, all `LinkNode` instances are processed first. Subsequently, the internal nodes are processed with the values from the last processed `LinkNode`, and their generated outputs are provided to all `LinkNode` instances at the end.

A more significant problem arises when a node depends on the output of another node based on the same composition, introducing a cycle. There is no correct way to calculate processing order, as topological ordering is only feasible for directed acyclic graphs. Figure 8.2 visualizes a pipeline that utilizes the `double_string` function to repeat a string four times. As depicted a cycly is introduced by the connection between `double_string_1 Output` and `double_string_2 Input`.

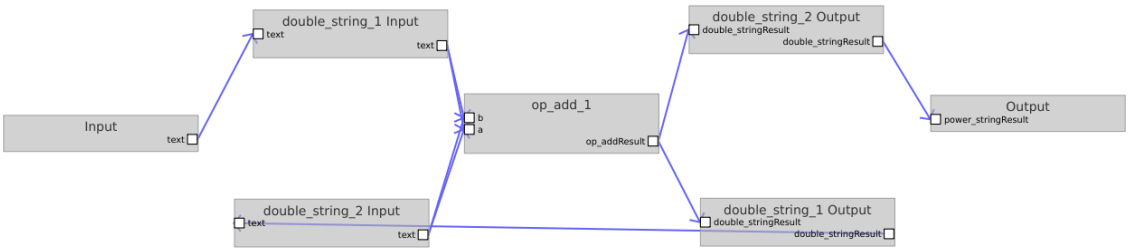

Figure 8.2: Introducing cycles by expanding dependent nodes

Several solutions can address this issue. Firstly, the algorithm for calculating processing order must be enhanced to compute the order recursively upon encountering an expanded node. Alternatively, internal nodes can be replicated for each call, significantly increasing the number of nodes present in a `Pipeline` instance. Currently, these situations can be avoided by never expanding more than two nodes based on the same composition. However, it is crucial to address this problem correctly in future iterations.

8.1.3 Generating `Pipeline` from Existing Descriptions

Unexpected behavior is observed when generating a `Pipeline` from a previously-generated RDF description. The *digit recognizer* project uses tuples as default values. When a `Pipeline` is generated from Python code, a tuple literal encountered in a **mapFromTerm** mapping is correctly converted into a tuple instance. However, when this process uses an existing description, the literals are not converted and instead return `None`. This is peculiar because both literals share the exact same RDF representation. No documentation for the `Literal` class could be found to explain this behavior. This issue significantly hinders the distribution of generated descriptions, making it difficult for others to replicate the Python pipeline.

8.2 Qualitative Evaluation

To perform a qualitative evaluation, two ML projects were selected to analyze the generated graphs. This evaluation tests how effectively the graphs store and provide provenance information about the pipeline. The first project examined is the *TedTalk recommendation* project, which recommends TedTalks based on a search term by training a vectorizer and combining it with similarity measures. The second project is the *sentiment* project, which trains an LSTM model to classify tweets as positive or negative.

8.2.1 TedTalk Recommendation

The initial logical step is to inspect the execution that trained the model. Since `mlflow` does not capture the training of a vectorizer, it is necessary to know the training function. By consulting the documentation of `TfidVectorizer`, it is determined that the input is provided to the argument `raw_documents`. Code Fragment 8.2 shows a SPARQL query that fetches the FnO Execution of the `fit` function together with its input data.

```
SELECT ?exe ?docs ?val WHERE {
    ?exe a fno:Execution ;
        fno:executes :TfidfVectorizer_fit_1 .
        :raw_documents ?docs .
    ?docs rdf:value ?val .
```

Code Fragment 8.2: Query to get vectorizer training provenance

The input data, represented as `SeriesInstance_16`, can have its provenance extracted using the function `provenance_tree`. This function implements Algorithm 9 to visualize a provenance tree with `graphviz`. Starting from a given entity, the function recursively follows the PROV-O predicate **wasGeneratedBy** to trace back through the execution steps until no further execution can be found. This constructs a tree that reveals the provenance of the entity. Additional links between derived entities provide deeper insights. The tree generated for `SeriesInstance_16` in Appendix **??** demonstrates how the graph accurately represents the execution process.

Section 6.3.2 stated that the output gets binarized into a `.pkl` file. This allows the binarized file to be used in other pipelines. The `recommend_talks` function expects a vectorizer, a dataset containing vectorized TED talks, and, of course, a vectorizer. By using this tool, the pipeline is successfully executed, providing recommendations while also capturing provenance. Unfortunately, the system treats the vectorizer as a new input and is unable to link it to its training process. This issue arises because the system cannot correctly convert Python instances to and from RDF literals, preventing the use of RDF representations as inputs.

Algorithm 9 Provenance Tree Construction

```
Require: entity, graph
    dot ← Digraph()
    visited ← []
    stack ← [entity]
    Add node entity to dot
    Add entity to visited
    while stack ≠ ∅ do
        current_entity ← stack.pop()
        exe ← FnO Execution that generated current_entity
        if exe ≠ None then
            Add node exe to dot
            Connect node current_entity to node exe with label 'wasGeneratedBy'
            used_entities ← PROV-O entities used by exe
            dv ← FnO Entity that current_entity was derived from
            for each entity in used_entities do
                Add node used to dot with label val
                Connect node exe to node used with label 'used'
                if used = dv then
                    Connect node current_entity to node used with label 'wasDerivedFrom'
                end if
                if used ∉ visited then
                    Add used to stack
                    Add used to visited
                end if
            end for
            if dv ∉ visited then
                Add used to stack
                Add used to visited
            end if
        end if
    end while
    Save dot as a PNG file
```

8.2.2 Sentiment Classification

This section delves into the provenance generated by ML operations using MLSchema. Leveraging the MLSchema Run grants access to all entities associated with ML operations. This includes fetching the implementation and the model, enabling the derivation of the training function, which represents the execution utilizing the implementation to train the model. Additionally, querying the run provides insights into the input data.

For the classification model, detailed evaluation metrics are obtained through MLSchema ModelEvaluations. These metrics provide crucial information regarding the model's performance. The metrics indicate a strong performance by the model. However, it's important to note that accuracy alone may not offer a complete picture and could potentially be misleading. For a more comprehensive evaluation, metrics such as precision and recall are essential. It's worth mentioning that Mlflow only records these metrics if they are explicitly defined in the pipeline's source code. Therefore, the availability of comprehensive metrics depends on the practices of the programmer. Code Fragment 8.3 presents an example output of chained queries using the MLSchema Run as the starting point. Further examination of the provenance document can be achieved by generating a provenance tree for fetched all entities. As an illustration, a provenance tree is generated for the samples and presented in Appendix 9.2.

```
Got run:    sedate-shoat-608
Got imp:    SequentialInstance_1
Got model:  SequentialInstance_2
Got train:  Sequential_fit_1Execution
Got data:   ModelInput
    samples:    NdarrayInstance_2
    labels:     NdarrayInstance_3
Got Metrics:
    validation_accuracy:    0.9203118085861206
    validation_loss:        0.20567485690116882
    val_loss:               0.20567485690116882
    loss:                   0.4047428369522095
    accuracy:               0.8330805897712708
    val_accuracy:           0.9203118085861206
```

Code Fragment 8.3: Output of chained queries inspecting the MLSchema Run

8.3 Quantitave Evaluation

Performance metrics were gathered for all ML projects listed in Table 8.1. This evaluation aims to determine the time required to generate RDF descriptions from Python code and identify potential factors influencing this process. Additionally, the time taken to convert RDF descriptions into `Pipeline` instances for visualization is measured. Finally, the impact on execution time when the code is executed using a `Pipeline` instance is assessed to evaluate the tool's feasibility.

8.3.1 Generation Time

Table 8.2 illustrates the time needed for generating an RDF graph compared to generating the corresponding `Pipeline` instance. It is immediately evident that generating the `Pipeline` instance requires significantly more time. This is most likely due to the slow SPARQL querying of `rdflib` [35]. As expected, the generation time increases with the number of triples produced. This number, in turn, is influenced by the complexity of the source code. Traditionally, complexity could be gauged using metrics such as cyclomatic and Halstead complexity measures [36]. However, upon employing the `radon` tool for these calculations [37], it appeared that all projects exhibited similar levels of complexity. As a result, these metrics were not considered further in our performance analysis.

Table 8.2: Generation time

Name	Graph gen. Time (ms)	# Triples	`Pipeline` gen. time (ms)	# Nodes
Sentiment Analysis	1277	1783	7111	37
Digit Recognition	2828	3402	12129	59
Iris Flower Classification	1193	1856	3728	21
TedTalk Reccomendations	1497	2038	5177	28
IPL Score Prediction	2952	3163	11457	55
Music Genre Clustering	1462	1927	5003	28
Stock Price Prediction	2728	3272	8605	40

The time required to create a `Pipeline` instance exhibits a linear relationship with the number of nodes created, as depicted in Figure 8.3. However, a more intriguing aspect to explore is the relationship between the number of nodes created and the number of triples in an RDF graph. Figure 8.4 demonstrates a linear correlation between the number of generated nodes and the number of triples. This suggests that an increase in triples leads to a proportional increase in the time required to generate a `Pipeline`. Nonetheless, there is noticeable variance, indicating that factors beyond the number of triples influence generation time. This variance likely stems from the complexity of the code being described.

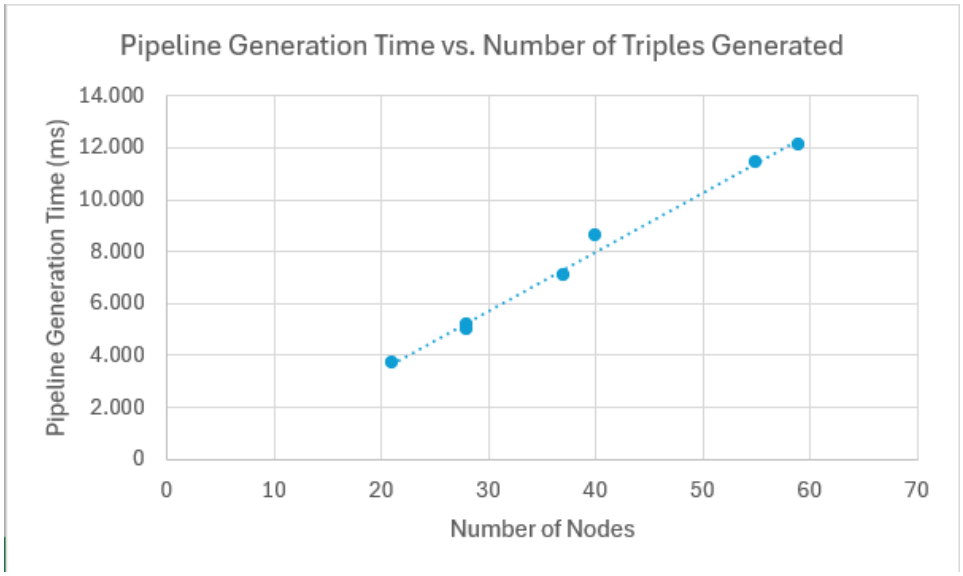

Figure 8.3: `Pipeline` generation time vs. number of nodes

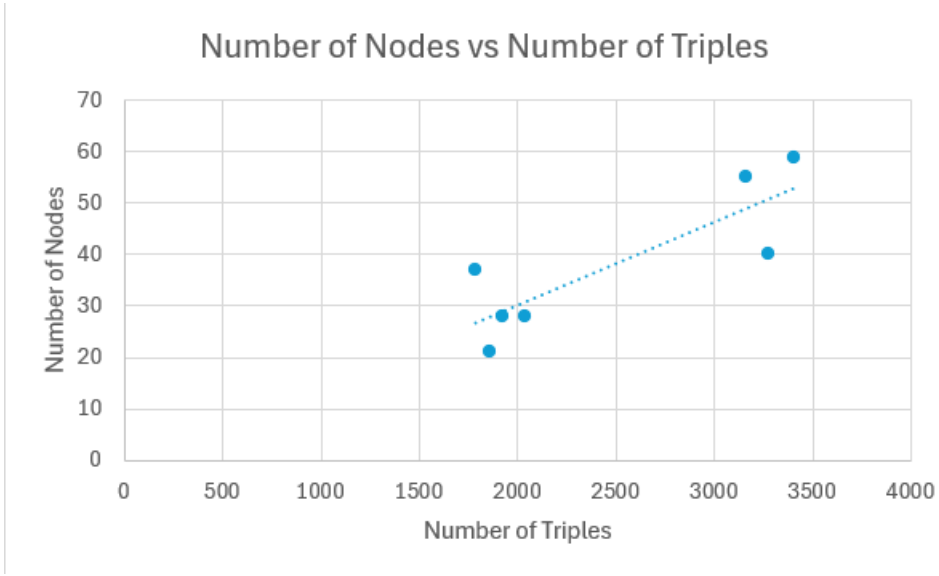

Figure 8.4: Number of nodes vs. number of triples

8.3.2 Execution Time

The *stock price prediction* project is excluded from this section due to ongoing errors encountered during execution via the tool. Table 8.3 outlines the performance impact observed when executing a pipeline through a `Pipeline` instance. Performance impact is quantified as the ratio of the `Pipeline` execution time to the default execution time. Notably, pipelines executing swiftly experience a more significant impact. This suggests that tool execution introduces a certain constant amount of overhead which slowly increases. The rate at which the overhead increases seams to slow and becomes obsolete as pipeline execution duration increases. Figure 8.5 visually represents this trend.

Table 8.3: Performance Impact

Name	`Pipeline` exe. time (ms)	Default exe. time (ms)	Performance Impact
Sentiment Analysis	524241	483198	1,08
Digit Recognition	16873	5532	3,05
Iris Flower Classification	4183	4	1006,35
TedTalk Reccomendations	3304	533	6,20
IPL Score Prediction	79912	49398	1,62
Music Genre Clustering	4249	45	94,53

A final test was conducted to examine the relationship between the number of processed nodes (i.e., open nodes) and the actual increase in execution time. The data used to plot Figure 8.6 is presented in Table 8.4. Surprisingly, no correlation between these two variables was observed. Figure 8.7 illustrates the time difference relative to the default execution time, clearly showing a constant overhead for pipelines that execute quickly. Beyond a certain point, the overhead increases linearly with the default execution time. This constant overhead can be attributed to the setup of autologging with `mlflow`, which requires a fixed amount of time. Other factors, such as determining process order and generating provenance, were anticipated to introduce overhead proportional to the number of open nodes, but this was not observed. No explanation for this behavior was identified.

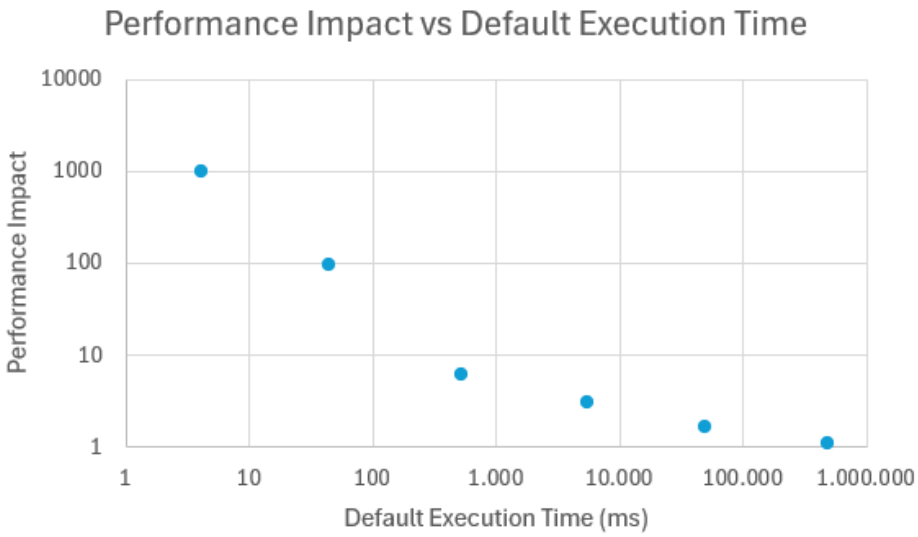

Figure 8.5: Performance Impact vs default execution time

Table 8.4: Time difference vs. number of open nodes

Name	Time Difference (ms)	# Open nodes
Sentiment Analysis	41042	19
Digit Recognition	11342	59
Iris Flower Classification	4179	21
TedTalk Reccomendations	2771	28
IPL Score Prediction	30514	55
Music Genre Clustering	4204	28

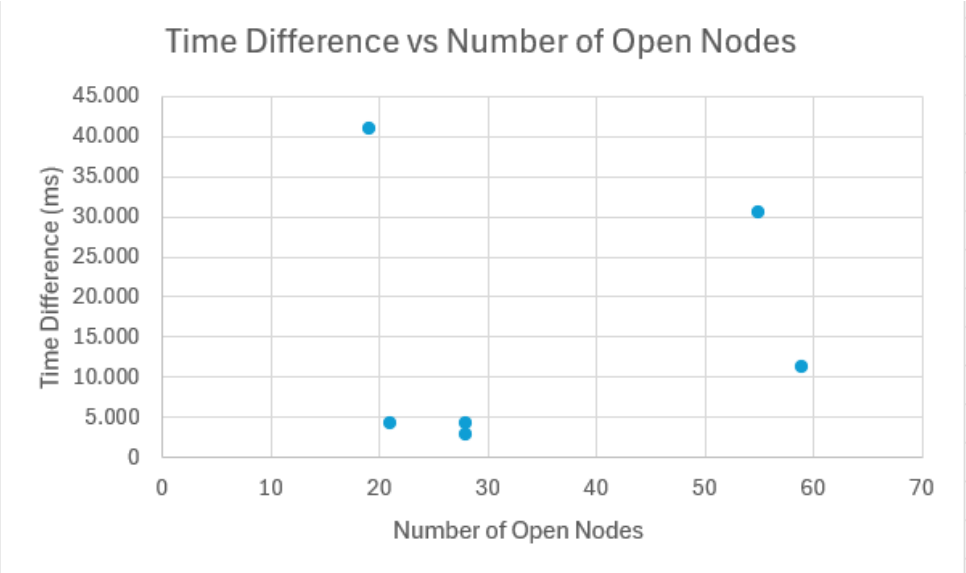

Figure 8.6: Time difference vs number of open nodes

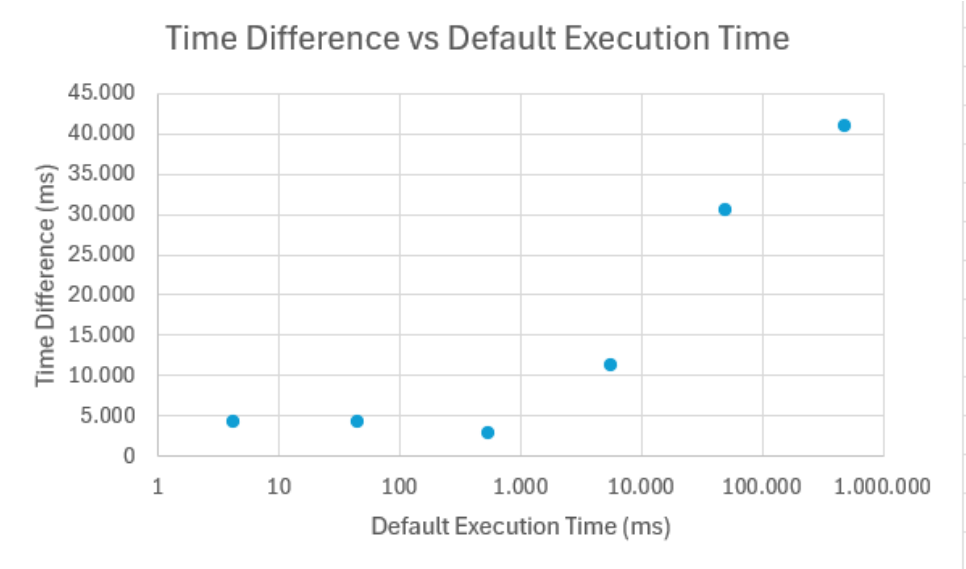

Figure 8.7: Time difference vs number of open nodes

9

Future Work

9.1 Enhancing Tool Capabilities

Our current tool offers a valuable foundation for describing machine learning pipelines. However, to ensure broader applicability and handle complex workflows, further development is necessary. This section highlights key areas for improvement.

1. **Addressing Order Calculation and Conditional Handling**: The existing algorithm for processing order determination and handling of conditional statements within pipelines requires refinement. Specific focus should be directed towards improving the identification and processing of currently unsupported or incorrectly processed AST nodes.

2. **Universal Instance Representation**: Ideally, a universal conversion algorithm that accurately represents any Python instance in RDF format without information loss would be developed. However, if such a solution proves infeasible, establishing transformation schemes for commonly used data types can significantly enhance the reusability of generated pipeline descriptions. This could potentially solve the current problems that arise when executing certain pre-generated graphs.

3. **Environment Representation**: By embedding environment setup instructions within the RDF descriptions, pipelines could be distributed independently, eliminating the current environment dependency.

4. **Enhancing FnO Description Reuse**: The current limitations of the `FnoDescriptionMap` class hinder the effective reuse of existing descriptions due to strict matching requirements. This could be addressed by incorporating NLP techniques or schema alignment algorithms. These enhancements would enable the tool to identify relevant FnO descriptions that closely match Python objects, leveraging the full potential of linked data for improved efficiency.

9.2 Integration with Solid Pods

While currently envisioned as a standalone tool, our solution can potentially become invaluable within the Solid environment for Machine Learning operations. Solid pods empower users with greater control over their personal data by storing it in decentralized data stores. This approach contrasts with traditional methods where large companies amass user data in centralized locations. The Solid project aims to address these concerns by facilitating data processing from individual pods, granting companies controlled access to authorized data while storing newly generated information back into the user's pod. ([38])

Solid's interoperability across data pods relies heavily on semantic technologies like RDF ([38]). By employing our tool to semantically describe machine learning operations, these descriptions can be incorporated within the Solid ecosystem. This integration offers several advantages:

1. **Incorporation of Machine Learning Workflows**: Machine learning pipelines described using our tool can be seamlessly integrated within the Solid environment.
2. **Enhanced User Control and Provenance Tracking**: Users gain greater control over how their data is used for machine learning by authorizing access to specific data within their pods. Additionally, the semantic descriptions provide provenance information, allowing users to understand how their data is processed.

However, significant development is required before full integration with Solid pods can be achieved.

Conclusion

This thesis successfully developed a tool that automatically generates semantic descriptions (using FnO) of machine learning pipelines written in Python code. These descriptions provide a machine-readable and universal format, exposing the inner workings of a pipeline and its data transformations with clear semantic meaning. This newfound transparency offers valuable insights into how pipelines function.

While developing the tool, challenges arose due to the complexities of Python code, particularly when dealing with intricate structures like loops and conditional statements. Additionally, generated descriptions needed to facilitate conversion back into an executable format, necessitating an expansion of the current FnO ontology to incorporate this functionality. Despite these hurdles, the tool demonstrably generates descriptions for simple, straightforward ML pipelines. Moreover, it allows pipeline setup for execution while capturing provenance information. This paves the way for ensuring transparency in ML operations, a crucial aspect of complying with GDPR legislation – current ML pipelines handling personal data often lack such transparency.

Looking forward, the tool's resilience can be further enhanced. Additionally, exploring potential integration within the Solid environment holds promise for enabling transparent ML operations over Solid pods. This approach would empower users to distribute their machine learning projects transparently, fostering stronger collaboration within the field. While the tool introduces some overhead, this cost becomes negligible when dealing with long-running pipelines, which are common in ML.

In conclusion, this thesis presents a significant step towards achieving transparent and explainable Machine Learning. The developed tool offers valuable capabilities and paves the way for further advancements in responsible and collaborative ML practices.

Ethical and Societal Reflection

The development presented in this thesis aligns with several Sustainable Development Goals (SDGs) established by the United Nations [39]. By promoting transparency and explainability in machine learning pipelines, this tool contributes to **SDG 9: Industry, Innovation and Infrastructure**. It fosters responsible development and application of technological advancements.

The increased automation of decision-making processes through machine learning necessitates vigilance against potential biases and discrimination. This aligns with **SDG 10: Reduced Inequalities**, emphasizing the need to bridge societal divides and ensure equitable access to technological benefits. Furthermore, the emphasis on user control and data provenance aligns with **SDG 16: Peace, Justice and Strong Institutions**. By empowering users with insights into data processing, the tool promotes transparency and accountability within AI development. Moving forward, continued development efforts should prioritize fairness, transparency, and user control to ensure this technology serves humanity and contributes positively to achieving the SDGs.

References

[1] J. Frey, S. Hellmann, and M. Assoc Comp, "Fair linked data - towards a linked data backbone for users and machines," in *30th World Wide Web (WWW) Conference (WebConf)*, 2021, Conference Proceedings, pp. 431–435, frey, Johannes Hellmann, Sebastian. [Online]. Available: <GotoISI>://WOS:000749534900072

[2] T. Berners-Lee. (2009) Linked data design principles. [Online]. Available: https://www.w3.org/wiki/LinkedData

[3] E. Prud'hommeaux and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, 2008. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[4] European Parliament and Council of the European Union, "Regulation (EU) 2016/679 of the European Parliament and of the Council." [Online]. Available: https://data.europa.eu/eli/reg/2016/679/oj

[5] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao, "Prov-o: The prov ontology," April 30, 2013 2013. [Online]. Available: https://www.w3.org/TR/prov-o/

[6] S. Scherzinger, C. Seifert, L. Wiese, and I. C. Soc, "The best of both worlds: Challenges in linking provenance and explainability in distributed machine learning," in *39th IEEE International Conference on Distributed Computing Systems (ICDCS)*, ser. IEEE International Conference on Distributed Computing Systems. LOS ALAMITOS: Ieee Computer Soc, 2019, Conference Proceedings, pp. 1620–1629. [Online]. Available: <GotoISI>://WOS:000565234200150

[7] P. I. Nakagawa, L. F. Pires, J. L. R. Moreira, L. Santos, and F. Bukhsh, "Semantic description of explainable machine learning workflows for improving trust," *Applied Sciences-Basel*, vol. 11, no. 22, p. 18, 2021. [Online]. Available: <GotoISI>://WOS:000724818600001

[8] B. De Meester, A. Dimou, and F. Kleedorfer, "The function ontology," 2023. [Online]. Available: https://fno.io/spec/

[9] B. De Meester, T. Seymoens, A. Dimou, and R. Verborgh, "Implementation-independent function reuse," *Future Generation Computer Systems*, vol. 110, pp. 946–959, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X19303723

[10] B. De Meester, A. Dimou, R. Verborgh, and E. Mannens, "Detailed provenance capture of data processing," *Proceedings of the First Workshop on Enabling Open Semantic Science (SemSci)*, vol. 1931, pp. 31–38, 2017. [Online]. Available: http://ceur-ws.org/Vol-1931/#paper-05

[11] D. Esteves, A. Ławrynowicz, P. Panov, L. Soldatova, T. Soru, and J. Vanschoren, "Ml schema core specification," Oct. 2016. [Online]. Available: http://ml-schema.github.io/documentation/ML%20Schema.html#overview

9 References

[12] P. Domingos, "A few useful things to know about machine learning," *Communications of the Acm*, vol. 55, no. 10, pp. 78–87, 2012. [Online]. Available: <GotoISI>://WOS:000309215800024

[13] G. De Mulder, "fno-descriptor-python," 2022. [Online]. Available: https://github.com/FnOio/fno-descriptor-python

[14] "inspect - inspect live objects," 2023. [Online]. Available: https://docs.python.org/3/library/inspect.html

[15] "ast - abstract syntax tree," 2023. [Online]. Available: https://docs.python.org/3/library/ast.html

[16] A. Chen, A. Chow, A. Davidson, A. DCunha, A. Ghodsi, S. A. Hong, A. Konwinski, C. Mewald, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, A. Singh, F. Xie, M. Zaharia, R. Zang, J. Zheng, and C. Zumar, "Developments in mlflow: A system to accelerate the machine learning lifecycle," in *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, ser. DEEM'20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399579.3399867

[17] M. Schlegel and K.-U. Sattler, "Mlflow2prov: Extracting provenance from machine learning experiments," in *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*, ser. DEEM '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3595360.3595859

[18] D. Beckett, I. Schreiber, and P. Hayes, "Turtle: Terse rdf triple language," in *World Wide Web Consortium (W3C) Recommendation*, 2014. [Online]. Available: https://www.w3.org/TR/turtle/

[19] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[20] J. McCrae and P. Buitelaar, "Linking datasets using semantic textual similarity," *Cybernetics and Information Technologies*, vol. 18, pp. 109–123, 03 2018.

[21] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song, and D. Zhao, "Semantic sparql similarity search over rdf knowledge graphs," *Proc. VLDB Endow.*, vol. 9, no. 11, p. 840–851, jul 2016. [Online]. Available: https://doi.org/10.14778/2983200.2983201

[22] Python Software Foundation, "The import system," 2024. [Online]. Available: https://realpython.com/python-import/

[23] Geir Arne Hjelle, "Python import: Advanced Techniques and Tips."

[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and

X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library."

[26] "scikit-learn - train_test_split," 2007. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

[27] "Pyqtgraph - scientific graphics and gui library for python," 2021. [Online]. Available: https://www.pyqtgraph.org/

[28] "Sentiment analysis of tweets using an lstm." [Online]. Available: https://techvidvan.com/tutorials/python-sentiment-analysis/11

[29] "Handwritten digit recognition." [Online]. Available: https://data-flair.training/blogs/python-deep-learning-project-handwritten-digit-recognition/

[30] "Iris flower classification." [Online]. Available: https://data-flair.training/blogs/iris-flower-classification/

[31] "Tedtalk reccomendation." [Online]. Available: https://www.geeksforgeeks.org/ted-talks-recommendation-system-with-machine-learning/

[32] "Ipl score prediction using deep learning." [Online]. Available: https://www.geeksforgeeks.org/ipl-score-prediction-using-deep-learning/?ref=rp

[33] "Clustering music genres using python." [Online]. Available: https://thecleverprogrammer.com/2022/04/05/clustering-music-genres-with-machine-learning/#google_vignette

[34] "Stock price prediction with lstm." [Online]. Available: https://thecleverprogrammer.com/2022/01/03/stock-price-prediction-with-lstm/

[35] M. Bamboat, A. Hafeez, and A. Wagan, "Performance of rdf library of java, c and python on large rdf models [scopus index y category journal]," pp. 25–30, 01 2021.

[36] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 2, pp. 96–104, 1979.

[37] "Various code metrics for python code." [Online]. Available: https://github.com/rubik/radon

[38] Solid project. [Online]. Available: https://solidproject.org/

9 References

[39] U. Nations, “Transforming our world: the 2030 agenda for sustainable development.” [Online]. Available: https://sdgs.un.org/2030agenda

Appendices

Appendix A

This appendix contains the class diagram, which presents a simplified representation of the implemented classes and their functionalities. Some functions have been omitted, and multiple functions may be abstracted into a single representative function. Classes used for visualization are not included. For complete details, please refer to the project code.

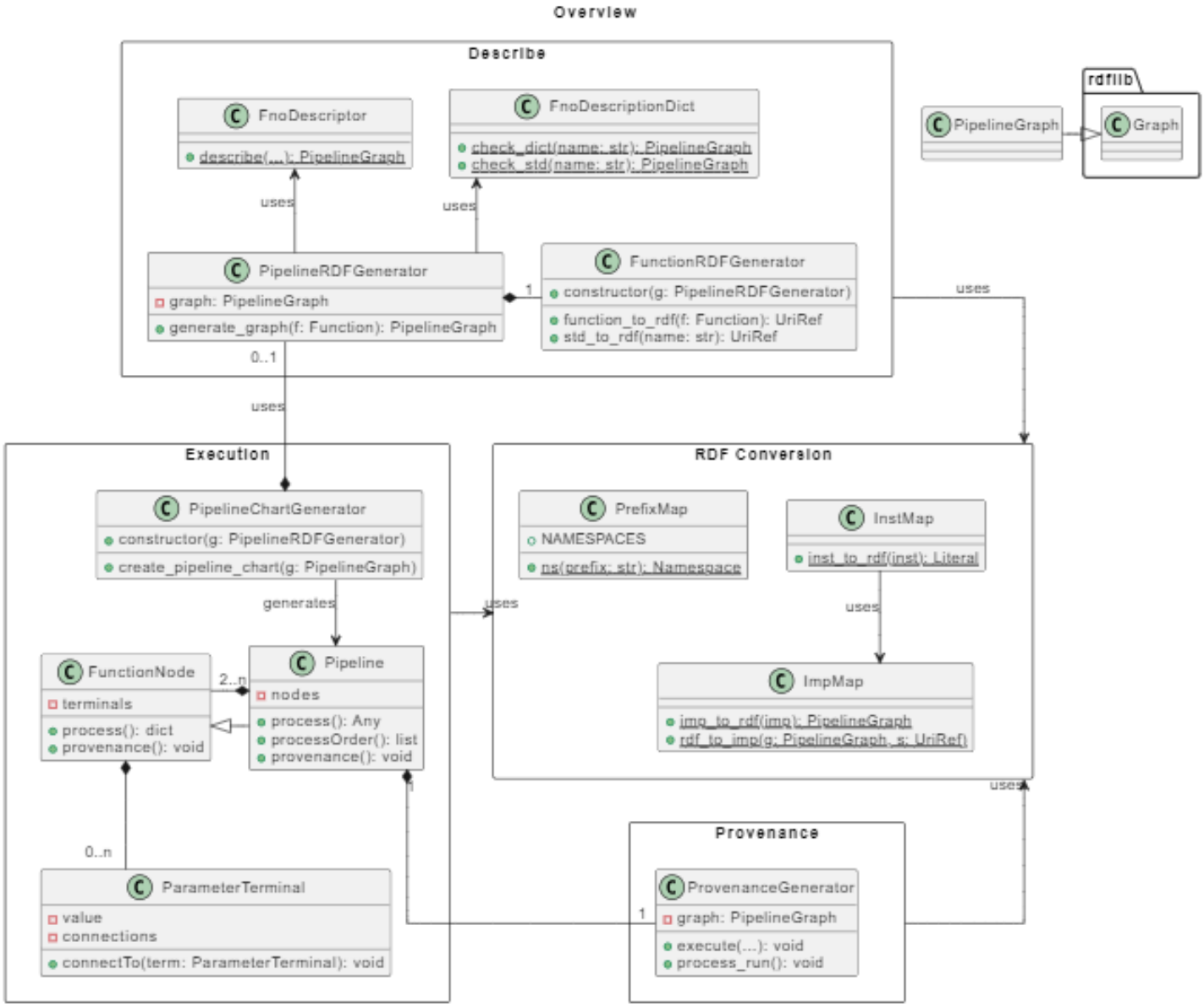

Figure 1: Class diagram

Appendix B

This sequence diagram provides a simplified representation of the internal interactions within the system when utilizing the user interface. The diagram focuses on the generation of the `PipelineGraph` and the execution of the `Pipeline`, while presenting all components of the user interface into a single entity, `MainWindow`.

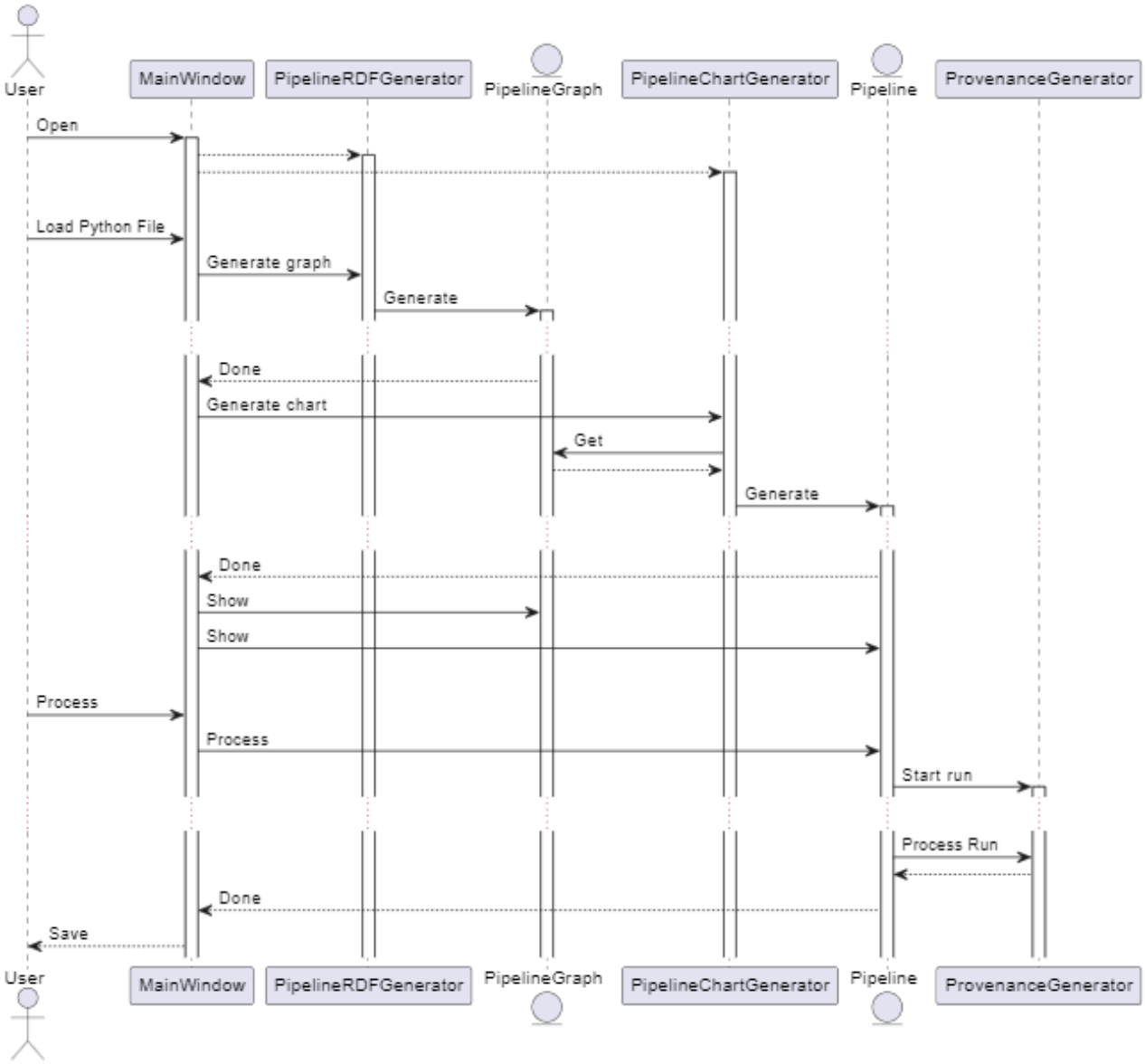

Figure 2: Sequence diagram

Appendix C

This appendix details the automatically generated FnO description for the `keras` library's `Tokenizer` class and its `texts_to_sequences` method. A tokenizer breaks text into its parts (words/characters) for machine learning. The `texts_to_sequences` method specifically converts a collection of text documents into sequences of integers, preparing text data for numerical processing by machine learning algorithms.

```
:Tokenizer a fnoi:PythonClass ;
    rdfs:label "Tokenizer" ;
    dcterms:description "Text tokenization utility class. ..." ;
    fnoi:module "keras.preprocessing.text" ;
    fnoi:package "keras" .

:Tokenizer_texts_to_sequences a fno:Function ;
    fno:name "texts_to_sequences" ;
    fno:expects [ a rdf:Seq ;
            rdf:_1 :Tokenizer_texts_to_sequencesParameter0 ;
            rdf:_2 :Tokenizer_texts_to_sequencesParameterSelf ] ;
    fno:returns [ a rdf:Seq ;
            rdf:_1 :Tokenizer_texts_to_sequencesSelfOutput ;
            rdf:_2 :Tokenizer_texts_to_sequencesOutput ] .

:Tokenizer_texts_to_sequencesParameter0 a fno:Parameter ;
    fno:predicate :texts ;
    fno:required true ;
    fno:type :Any .

:Tokenizer_texts_to_sequencesParameterSelf a fno:Parameter ;
    fno:predicate :self ;
    fno:required true ;
    fno:type :Tokenizer .

:Tokenizer_texts_to_sequencesOutput a fno:Output ;
    fno:predicate :Tokenizer_texts_to_sequencesResult ;
    fno:type :Any .
```

```
:Tokenizer_texts_to_sequencesSelfOutput a fno:Output ;
    fno:predicate :self_output ;
    fno:type :Tokenizer .

:texts_to_sequencesImplementation a fnoi:PythonMethod ;
    rdfs:label "texts_to_sequences" ;
    dcterms:description "Transforms each text in texts to a sequence of
    ↪  integers. ..." ;
    fnoi:module "keras.preprocessing.text" ;
    fnoi:package "keras" ;
    fnoi:self :Tokenizer ;
    fnoi:static false .

:texts_to_sequencesMapping a fno:Mapping ;
    fno:function :Tokenizer_texts_to_sequences ;
    fno:implementation :Tokenizer_texts_to_sequencesImplementation ;
    fno:methodMapping [ a fnom:StringMethodMapping ;
            fnom:method-name "texts_to_sequences" ] ;
    fno:parameterMapping [ a fnom:PositionParameterMapping ;
            fnom:functionParameter :texts_to_sequencesParameter0 ;
            fnom:implementationParameterPosition 1 ],
        [ a fnom:PropertyParameterMapping ;
            fnom:functionParameter :texts_to_sequencesParameter0 ;
            fnom:implementationProperty "texts" ],
        [ a fnom:PropertyParameterMapping ;
            fnom:functionParameter :texts_to_sequencesParameterSelf ;
            fnom:implementationProperty "self" ],
        [ a fnom:PositionParameterMapping ;
            fnom:functionParameter :texts_to_sequencesParameterSelf ;
            fnom:implementationParameterPosition 0 ] ;
    fno:returnMapping [ a fnom:ValueReturnMapping ;
            fnom:functionOutput :texts_to_sequencesSelfOutput ],
        [ a fnom:DefaultReturnMapping ;
            fnom:functionOutput :texts_to_sequencesOutput ] .
```

```
:texts_to_sequencesExecution a fno:Execution ;
    :texts_to_sequencesResult :ListInstance_2 ;
    :implementation :texts_to_sequencesImplementation ;
    :self :TokenizerInstance_2 ;
    :self_output :TokenizerInstance_2 ;
    :texts :ListInstance_1 ;
    fno:executes :texts_to_sequences ;
    fno:uses :texts_to_sequencesMapping .
```

Appendix D

This appendix presents the complete code for Code Fragment 5.1, which demonstrates the use of the `sum` function to add three integers. This example is an excellent reference for understanding FnOC Compositions.

```
:sum3 a fno:Function ;
    fno:composition :sum3Pipeline ;
    fno:name "sum3" ;
    fno:expects [ a rdf:Seq ;
            rdf:_1 :sum3Parameter0 ;
            rdf:_2 :sum3Parameter1 ;
            rdf:_3 :sum3Parameter2 ] ;
    fno:returns [ a rdf:Seq ;
            rdf:_1 :sum3Output ] .

:sum3Parameter0 a fno:Parameter ;
    fno:predicate :a ; fno:required true ; fno:type :int .

:sum3Parameter1 a fno:Parameter ;
    fno:predicate :b ; fno:required true ; fno:type :int .

:sum3Parameter2 a fno:Parameter ;
    fno:predicate :c ; fno:required true ; fno:type :int .

:sum3Output a fno:Output ;
    fno:predicate :sum3Result ; fno:type :int .

:sum a prov:Entity,
        fno:Function ;
    fno:composition :sumPipeline ;
    fno:expects [ a rdf:Seq ;
            rdf:_1 :sumParameter0 ;
            rdf:_2 :sumParameter1 ] ;
    fno:name "sum" ;
    fno:returns [ a rdf:Seq ;
            rdf:_1 :sumOutput ] .
```

```
:sumParameter0 a fno:Parameter ;
    fno:predicate :a ; fno:required true ; fno:type :int .

:sumParameter1 a fno:Parameter ;
    fno:predicate :b ; fno:required true ; fno:type :int .

:sumOutput a fno:Output ;
    fno:predicate :sumResult ; fno:type :int .

:sum_1 fnoc:applies :sum .
:sum_2 fnoc:applies :sum .

:sum3Pipeline a fnoc:Composition ;
    fnoc:composedOf
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum3 ;
                         fnoc:functionParameter :sum3Parameter2 ] ;
          fnoc:mapTo [ fnoc:constituentFunction :sum_1 ;
                       fnoc:functionParameter :sumParameter1 ] ],
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum3 ;
                         fnoc:functionParameter :sum3Parameter1 ] ;
          fnoc:mapTo [ fnoc:constituentFunction :sum_2 ;
                       fnoc:functionParameter :sumParameter1 ] ],
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum_1 ;
                         fnoc:functionOutput :sumOutput ] ;
          fnoc:mapTo [ fnoc:constituentFunction :sum3 ;
                       fnoc:functionOutput :sum3Output ] ],
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum3 ;
                         fnoc:functionParameter :sum3Parameter0 ] ;
          fnoc:mapTo [ fnoc:constituentFunction :sum_2 ;
                       fnoc:functionParameter :sumParameter0 ] ],
        [ fnoc:mapFrom [ fnoc:constituentFunction :sum_2 ;
                         fnoc:functionOutput :sumOutput ] ;
          fnoc:mapTo [ fnoc:constituentFunction :sum_1 ;
                       fnoc:functionParameter :sumParameter0 ] ] .
```

Appendix E

In this appendix, various concise representations of AST nodes available in Python's `ast` module are provided, along with an example.

FunctionDef

```
def add(a, b):
    return a + b
```

```
FunctionDef(
  name='add', args=arguments(posonlyargs=[], args=[arg(...), arg(...)],
  ↪  ...), body=[Return(value=BinOp(left=Name(...), op=Add(),
  ↪  right=Name(...)))],
  decorator_list=[])
```

Return

```
return a + b
```

```
Return(value=BinOp(left=Name(id='a', ctx=Load()), op=Add(),
↪  right=Name(id='b', ctx=Load())))
```

Assign

```
x = 10
```

```
Assign(targets=[Name(id='x', ctx=Store())], value=Constant(value=10),
↪  type_comment=None)
```

For

```
for i in range(10):
    print(i)
```

```
For(target=Name(id='i', ctx=Store()), iter=Call(func=Name(...),
↪  args=[...], ...), body=[Expr(...)], orelse=[], type_comment=None)
```

If

```python
if x > 0:
    print("Positive")
else:
    print("Non-positive")
```

```
If(test=Compare(left=Name(...), ops=[Gt()], comparators=[Constant(...)]),
↪  body=[...], orelse=[...])
```

Call

```python
print("Hello, World!")
```

```
Call(func=Name(id='print', ctx=Load()), args=[Constant(value='Hello,
↪  World!')], keywords=[])
```

Dict

```python
{'key': 'value'}
```

```
Dict(keys=[Constant(value='key')], values=[Constant(value='value')])
```

List

```python
[1, 2, 3]
```

```
List(elts=[Constant(value=1), Constant(value=2), Constant(value=3)],
↪  ctx=Load())
```

Tuple

```python
(1, 2, 3)
```

```
Tuple(elts=[Constant(value=1), Constant(value=2), Constant(value=3)],
↪  ctx=Load())
```

Comprehension

```
[x for x in range(10)]
```

```
ListComp(
  elt=Name(id='x', ctx=Load()),
  generators=[comprehension(target=Name(id='x', ctx=Store()),
  ↪  iter=Call(...), ...)])
```

```
{k: v for k, v in zip(range(10), range(10))}
```

```
DictComp(
  key=Name(id='k', ctx=Load()), value=Name(id='v', ctx=Load()),
  generators=[comprehension(target=Tuple(...), iter=Call(...), ...)])
```

Appendix F

This appendix contains the proposed extension for FnO and its vocabularies.

FnO Extensions

```
@prefix owl:    <http://www.w3.org/2002/07/owl#> .
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:    <http://www.w3.org/2001/XMLSchema#> .
@prefix vs:     <http://www.w3.org/2003/06/sw-vocab-status/ns#> .
@prefix fnoc:   <https://w3id.org/function/vocabulary/composition#> .
@base           <https://w3id.org/function/ontology#> .

:default rdf:type owl:ObjectProperty ;
    rdfs:domain :Parameter ;
    rdfs:label "default"@en ;
    rdfs:comment "Defines a default value for a parameter that is not
    ↪ required." ;
    rdfs:isDefinedBy <https://w3id.org/function/ontology> ;
    vs:term_status "proposed" .

:composition rdf:type owl:ObjectProperty ;
    rdfs:domain :Function ;
    rdfs:range fnoc:Composition ;
    rdfs:label "composition"@en ;
    rdfs:comment "Connects a function to a composition that describes its
    ↪ internal structure."
    rdfs:isDefinedBy <https://w3id.org/function/ontology> ;
    vs:term_status "proposed" .
```

FnOC Extensions

```
@prefix fno: <https://w3id.org/function/ontology#> .
@base <https://w3id.org/function/vocabulary/composition#> .

:mapCondition rdf:type owl:ObjectProperty ;
    rdfs:domain :CompositionMapping ;
    rdfs:range :CompositionMappingEndpoint ;
    rdfs:label "mapCondition"@en ;
    rdfs:comment "Defines a `condition` [:CompositionMappingEndpoint] for
    ↪ a [:CompositionMapping]"@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .

:mappingStrategy rdf:type owl:ObjectProperty ;
    rdfs:domain :CompositionMappingEndpoint ;
    rdfs:range :MappingStrategy ;
    rdfs:label "mappingStrategy"@en ;
    rdfs:comment "Defines a `strategy` [:MappingStrategy] for a
    ↪ [:CompositionMappingEndpoint]"@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .

:index rdf:type owl:ObjectProperty ;
    rdfs:domain :CompositionMappingEndpoint ;
    rdfs:range xsd:integer ;
    rdfs:label "index"@en ;
    rdfs:comment "Defines the index to acces a container"@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .

:MappingStrategy rdf:type owl:Class ;
    rdfs:label "Mapping Strategy"@en ;
    rdfs:comment "Defines a strategy for a
    ↪ [:CompositionMappingEndpoint]"@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .
```

```
:IsTrue rdf:type owl:Class ;
    rdfs:subClassOf :MappingStrategy ;
    rdfs:label "IsTrue Mapping Strategy"@en ;
    rdfs:comment "Indicates that the `condition`
    ↪ [:CompositionMappingEndpoint] of a [:CompositionMapping] must be
    ↪ True."@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .

:IsFalse rdf:type owl:Class ;
    rdfs:subClassOf :MappingStrategy ;
    rdfs:label "IsFalse Mapping Strategy"@en ;
    rdfs:comment "Indicates that the `condition`
    ↪ [:CompositionMappingEndpoint] of a [:CompositionMapping] must be
    ↪ False."@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status   "proposed" .

:SetItem rdf:type owl:Class ;
    rdfs:subClassOf :MappingStrategy ;
    rdfs:label "SetItem Mapping Strategy"@en ;
    rdfs:comment "Indicates that the [fno:Parameter] is a container and
    ↪ the `source` [:CompositionMappingEndpoint] maps to an element
    ↪ within."@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .

:GetItem rdf:type owl:Class ;
    rdfs:subClassOf :MappingStrategy ;
    rdfs:label "GetItem Mapping Strategy"@en ;
    rdfs:comment "Indicates that the [fno:Parameter] is a container and
    ↪ the `target` [:CompositionMappingEndpoint] is mapped to an
    ↪ element within."@en ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/composition> ;
    vs:term_status "proposed" .
```

FnOI Extensions

```
@base <https://w3id.org/function/vocabulary/implementation#> .

:PythonImplementation rdf:type owl:Class ;
    rdfs:subClassOf fno:Implementation ;
    rdfs:label "Python Implementation"@en ;
    rdfs:comment "An fno:Implementation, coded in Python"@en ;
    rdfs:isDefinedBy
     ↪  <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:PythonFunction rdf:type owl:Class ;
    rdfs:subClassOf :PythonImplementation ;
    rdfs:label "Python Function"@en ;
    rdfs:comment "An fno:Implementation, defined as a function in
     ↪  Python"@en ;
    rdfs:isDefinedBy
     ↪  <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:PythonMethod rdf:type owl:Class ;
    rdfs:subClassOf :PythonFunction ;
    rdfs:label "Python Method"@en ;
    rdfs:comment "An fno:Implementation, defined for a class in
     ↪  Python"@en ;
    rdfs:isDefinedBy
     ↪  <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:PythonClass rdf:type owl:Class ;
    rdfs:subClassOf :PythonImplementation ;
    rdfs:label "Python Class"@en ;
    rdfs:comment "An fno:Implementation, defined as a class in Python"@en
     ↪  ;
    rdfs:isDefinedBy
     ↪  <https://w3id.org/function/vocabulary/implementation> ;
```

```
    vs:term_status "proposed" .

:module rdf:type owl:ObjectProperty ;
    rdfs:domain :PythonImplementation ;
    rdfs:range xsd:string ;
    rdfs:label "module"@en ;
    rdfs:comment "Indicates the module this Python implementation is
    ↪ defined in"@en ;
    rdfs:isDefinedBy
    ↪ <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:package rdf:type owl:ObjectProperty ;
    rdfs:domain :PythonImplementation ;
    rdfs:range xsd:string ;
    rdfs:label "package"@en ;
    rdfs:comment "Indicates the package that contains this Python
    ↪ implemantions's module"@en ;
    rdfs:isDefinedBy
    ↪ <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:file rdf:type owl:ObjectProperty ;
    rdfs:domain :PythonImplementation ;
    rdfs:range xsd:string ;
    rdfs:label "file"@en ;
    rdfs:comment "Indicates the file on the local system this Python
    ↪ implementation is defined in"@en ;
    rdfs:isDefinedBy
    ↪ <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:self rdf:type owl:ObjectProperty ;
    rdfs:domain :PythonMethod ;
    rdfs:range :PythonClass ;
    rdfs:label "self"@en ;
```

```
    rdfs:comment "Connects a method implementation to the Python class it
    ↪ is defined for."@en ;
    rdfs:isDefinedBy
    ↪ <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .

:static rdf:type owl:ObjectProperty ;
    rdfs:domain :PythonMethod ;
    rdfs:range xsd:boolean ;
    rdfs:label "static"@en ;
    rdfs:comment "Indicates whether the Python method is static or
    ↪ not"@en ;
    rdfs:isDefinedBy
    ↪ <https://w3id.org/function/vocabulary/implementation> ;
    vs:term_status "proposed" .
```

FnOM Extensions

```
@base <https://w3id.org/function/vocabulary/implementation#> .

:ValueReturnMapping rdf:type owl:Class ;
    rdfs:subClassOf fno:ReturnMapping ;
    rdfs:label "Value Return Mapping"@en ;
    rdfs:comment "Indicates this [fno:Output] returns the value a method
    ↪ is called upon" ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/mapping> ;
    vs:term_status "proposed" .

:VarPositionalParameterMapping rdf:type owl:Class ;
    rdfs:subClassOf  fno:ParameterMapping ;
    rdfs:label "Variable Positional Parameter Mapping"@en ;
    rdfs:comment "Indicates this [fno:Parameter] is a list with an
    ↪ unexpected amount of elements." ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/mapping> ;
    vs:term_status "proposed" .
```

```
:VarPropertyParameterMapping rdf:type owl:Class ;
    rdfs:subClassOf fno:ParameterMapping ;
    rdfs:label "Variable Keyword Parameter Mapping"@en ;
    rdfs:comment "Indicates this [fno:Parameter] is a dictionary with an
    ↪  unexpected amount of key-value pairs." ;
    rdfs:isDefinedBy <https://w3id.org/function/vocabulary/mapping> ;
    vs:term_status "proposed" .
```

Appendix G

This appendix displays the provenance trees generated for evaluation. Figure 3 visualizes the input data used for training the `vectorize` function within the *TedTalk Recommendation* project. To ensure clarity and avoid clutter in the visualization, executions involving more than 10 entities are not explored further. Additionally, to manage space efficiently, instance values with substantial content, such as `DataFrame` instances, are represented by their names rather than displaying their full contents.

Figure 4 displays the tree generated for the samples utilized in training the LSTM model within the *sentiment prediction* project. The entity referenced was retrieved through the MLSchema Run as defined in the provenance document.

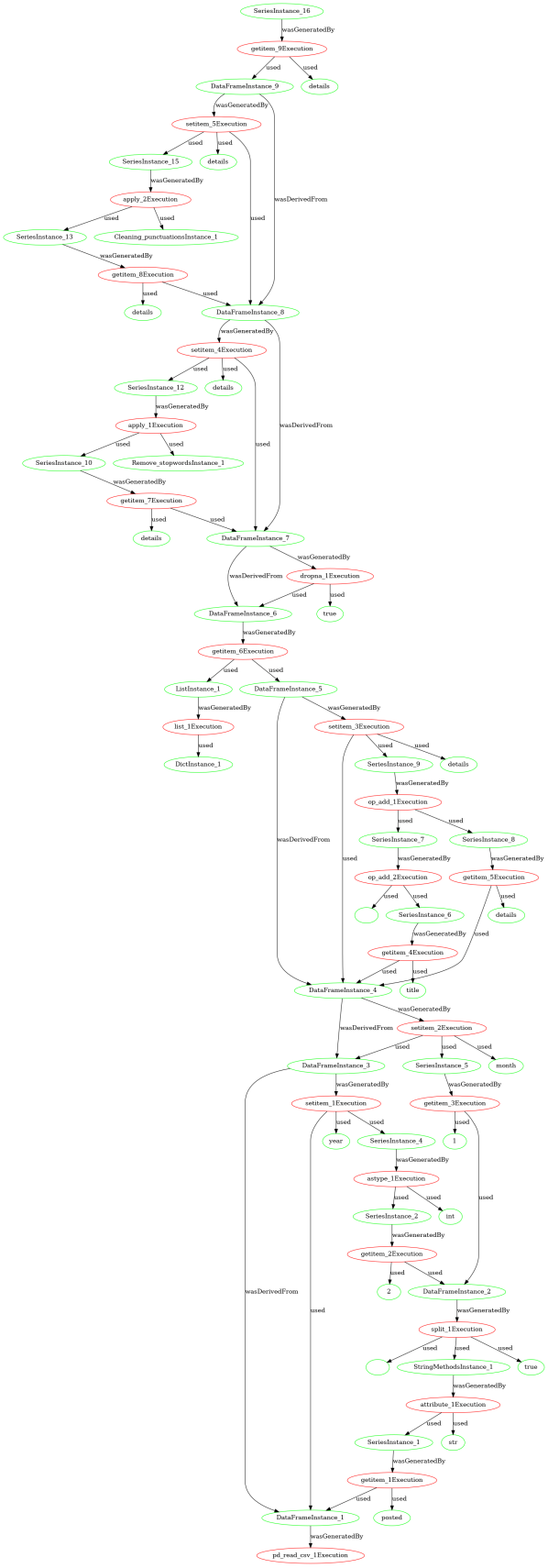

Figure 3: Provenance tree for TedX `vectorizer` input

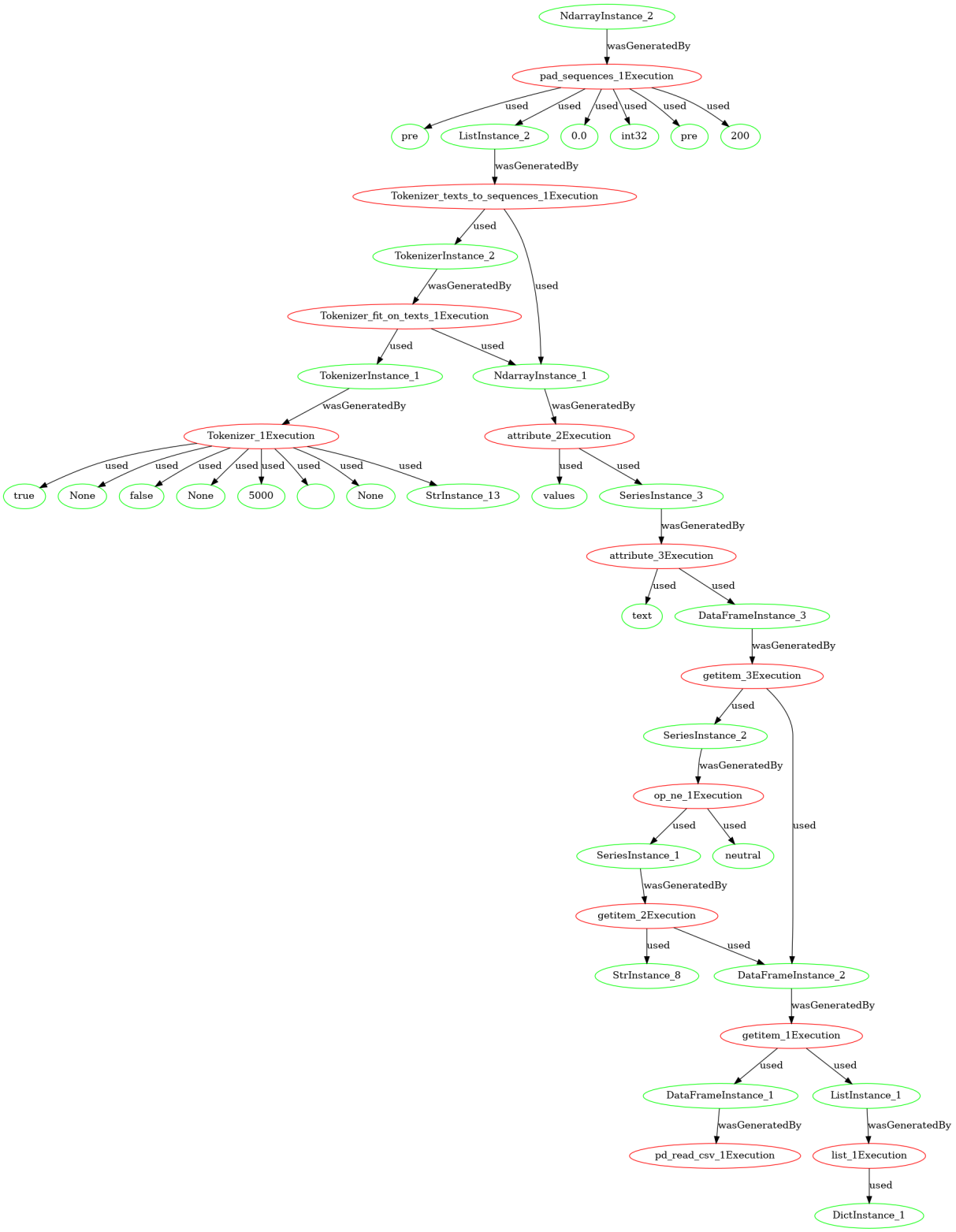

Figure 4: Provenance tree for TedX `vectorizer` input