UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Artjom Valdas

ML-TOSCA: ML pipeline modelling and orchestration using TOSCA

Master's Thesis (30 ECTS)

Supervisor(s): Chinmaya Kumar Dehury, PhD
Pelle Jakovits, PhD

Tartu 2023

ML-TOSCA: ML pipeline modelling and orchestration using TOSCA

Abstract:
In today's world, machine learning is increasingly involved in different areas. Moreover, automating machine learning workflows through AutoML enables organizations to develop and deploy machine learning solutions at scale rapidly. Additionally, leveraging the power of cloud computing can provide even greater scalability and flexibility, allowing us to efficiently process large datasets and cost-effectively train and implement complex machine learning models. Undoubtedly, these technologies will play an essential role in shaping the future across various industries. Despite many advantages, there is a lack of widespread combined implementations of AutoML and cloud-based solutions. This thesis describes a new AutoML integration approach to the TOSCA standard. TOSCA is an open-source specification used to describe the topology of cloud applications and services. Incorporating AutoML techniques into TOSCA enables users to automatically generate optimized machine learning models with the help of cloud applications, which can improve the speed and efficiency of model creation. The proposed approach is implemented in the RADON ecosystem, allowing node and relationship types to be created. The final solution allows users to create and join blocks to define a complete machine learning pipeline structure.

Keywords:
AutoML, TOSCA, ML-TOSCA, ML pipeline, Pipeline

CERCS: P170 - Computer science, numerical analysis, systems, control

ML-TOSCA: ML torujuhtme modelleerimine ja orkestreerimine TOSCA abil

Lühikokkuvõte:

Tänapäeva maailmas on masinõpe üha enam kaasatud erinevatesse valdkondadesse. Lisaks võimaldab masinõppe töövoogude automatiseerimine AutoML-i kaudu organisatsioonidel kiiresti arendada ja juurutada masinõppe lahendusi. Lisaks võib pilvandmetöötluse võimsuse võimendamine pakkuda veelgi suuremat skaleeritavust ja paindlikkust, võimaldades meil tõhusalt töödelda suuri andmestikke ning kulutõhusalt koolitada ja rakendada keerulisi masinõppe mudeleid. Kahtlemata mängivad need tehnoloogiad olulist rolli tuleviku kujundamisel erinevates tööstusharudes. Hoolimata paljudest eelistest puudub AutoML-i ja pilvepõhiste lahenduste laialdane kombineeritud rakendamine. Käesolevas töös kirjeldatakse uut AutoML integratsiooni lähenemist TOSCA standardile. TOSCA on avatud lähtekoodiga spetsifikatsioon, mida kasutatakse pilverakenduste ja -teenuste topoloogia kirjeldamiseks. AutoML tehnikate lisamine TOSCA-sse võimaldab kasutajatel automaatselt luua optimeeritud masinõppe mudeleid pilverakenduste abil, mis võib parandada mudeli loomise kiirust ja tõhusust. Kavandatavat lähenemisviisi rakendatakse RADONi ökosüsteemis, mis võimaldab luua sõlme- ja suhtetüüpe. Lõpplahendus võimaldab kasutajatel luua ja ühendada plokke, et määratleda täielik masinõppe torujuhtme struktuur.

Võtmesõnad:

AutoML, TOSCA, ML-TOSCA, ML pipeline, Pipeline

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Contents

1 Introduction 6
1.1 Problem statement 7
1.2 Thesis outline 7

2 Background 9
2.1 AutoML 9
2.1.1 Use cases 10
2.2 Related work 10
2.2.1 Managing machine learning systems 11
2.2.2 Apache Airflow 11
2.2.3 Apache Beam 12
2.2.4 Dask 13
2.2.5 Dagster 13
2.2.6 Kedro 14

3 TOSCA 16
3.1 Topology and Orchestration Specification for Cloud Applications 16
3.2 Structure 17
3.3 Execution phases 17
3.4 Workflow 18
3.5 RADON 19
3.6 Alternatives 21

4 ML-TOSCA Methodology 22
4.1 Research design 22
4.1.1 Technology stack 23
4.1.2 Data storage format 23
4.2 ML-TOSCA architectural foundations 24
4.2.1 ML-TOSCA architecture 24
4.2.2 Environment 26
4.2.3 Preprocessing 27
4.2.4 Model creation 28
4.2.5 Model evaluation 29
4.3 Technical implementation 29
4.3.1 Conda environment 29
4.3.2 Data reading 30
4.3.3 Preprocessing pipeline 32
4.3.4 Data splitting 34
4.3.5 Model creation and training 34

4.3.6 Trained model evaluation 35
4.3.7 Result exporting 36
4.4 Features and benefits 36

5 Demonstration of ML-TOSCA in action 39
5.1 Project setup 39
5.2 Classification project 39
5.2.1 Solution strategy for the Titanic classification problem 40
5.2.2 Implementing the Titanic classification solution using ML-TOSCA 40
5.2.3 Analyzing xOpera's execution flow on a Titanic problem 43
5.3 Testing results 44

6 Future research directions 46
6.1 Feature Scaling 46
6.2 Incorporating an event-driven approach 46
6.3 Visualization 47
6.4 Providing in-depth control over pipeline behavior 47
6.5 Cross-Platform Compatibility 48
6.6 Workload distribution over multiple virtual machines 48

7 Conclusion 49

References 53

Appendix 54
I. Repository 54
II. Licence 55

1 Introduction

Artificial intelligence (AI) has advanced significantly over the past decade. It is utilized more often in many applications, from self-driving cars and medical diagnosis to language translation and phishing detection. It has the power to alter several sectors and transform the way we live and work. Many organizations, including major IT companies and smaller startups, are creating and utilizing AI models for various uses.

AI is also beginning to develop in a new direction called AutoML (Automated Machine Learning). According to [21], AutoML can be defined as a combination of automating machine learning processes such as data preprocessing, feature engineering, model training, and evaluation. One of the advantages is that non-experts can apply machine learning in their domains by saving money and speeding up workflow [38].

Cloud computing is a different trend that is also gaining popularity year after year. Users can conveniently change, add, scale, and remove various resources on demand. Without creating and managing their infrastructure, consumers may access and utilize these resources whenever needed.

There are several approaches to integrating cloud computing and AutoML to create robust, effective, and affordable machine learning systems. For example, you may use the cloud to run your AutoML pipeline. This would entail storing data and model outputs in cloud storage while using VMs (virtual machines) or containers to run AutoML algorithms. As a result, you have more control over the pipeline and may alter it to suit your unique requirements. It will also provide you greater freedom to increase your resources on demand.

Nevertheless, speaking of automation, it is beneficial to have different versions of various processes, including their combined implementation, to ensure efficient management and implementation. From a technical perspective, having different standalone components [4] that can perform highly specialized tasks and communicate with each other to achieve a larger objective is critical. This approach allows for efficient distribution of workload, where each component can be optimized for a specific task, rather than trying to create a monolithic system that performs all functions in a single component. Breaking down a more significant task into smaller specialized components makes managing and maintaining the whole system easier. It necessitates a standard for all parts of the architecture, ensuring adherence to technical rules when adding or connecting new functionality and minimizing the risk of errors.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [30] is a standard for modeling the architecture of various services and cloud-based applications in a platform-agnostic manner. It enables an orchestrator to quickly set up, scale, deploy, configure, and manage cloud applications in a repeatable manner across various environments. TOSCA defines reusable components, including their relationships, infrastructure, and programming code [3], in a machine and human-readable format, enabling efficient modeling and deployment of complex systems. It allows for greater

efficiency and agility in cloud-based operations while reducing the risk of errors and inconsistencies arising from manual processes.

Even though TOSCA provides a robust framework for modeling various services, it may not cover specific domains such as data processing and machine learning. TOSCA's core blocks are abstract and can be used to model various applications. Therefore, this work aims to contribute missing blocks for AutoML to complement TOSCA's existing capabilities. The author is confident that this acquisition will provide significant value to many users by streamlining and speeding up tasks such as data cleaning, preprocessing, training machine learning models, and evaluating them. However, modeling machine learning blocks and data processing tasks in TOSCA can be complex due to their sequential steps, dependencies between them, and the various approaches and tools that require careful consideration.

Undoubtedly, similar solutions and tools already focus on AutoML pipeline creation, such as Google Cloud AutoML, DataRobot, Dataiku, and H2O.ai. However, they have several problems and difficulties, starting from the price and ending with vendor lock-in. This work solves these problems and makes it possible to use this tool in the academic and entrepreneurial fields.

1.1 Problem statement

The rise in demand for machine learning in cloud-based applications is impeded by the necessity for additional standardization in creating, building, and deploying machine learning models. TOSCA, one of the most prevalent standards in cloud-specific activities, provides a vendor-agnostic and platform-independent means of describing cloud-based applications and services. Through the utilization of TOSCA, machine learning models can be integrated more seamlessly into cloud-based applications, ensuring superior portability and interoperability across various cloud platforms and providers.

The problem addressed by this work is how to enable the use of AutoML in TOSCA-compliant cloud applications by defining a standard for describing data preprocessing and machine learning models and automating the processes for creating and deploying them. The goal is to reduce the time and effort required to create, build and optimize machine learning pipelines in TOSCA-compliant cloud applications while ensuring that AutoML components are accurate, reliable, compatible, and reusable.

1.2 Thesis outline

This thesis consists of five more significant sections. First of all, Section 2 briefly introduces automated machine learning and its various stages and use cases. The technologies that implement AutoML are also discussed in this section. Section 3 delves deeper into the foundations and principles of TOSCA by explaining its architecture and workflow. It also highlights a crucial aspect of this thesis, a RADON project used to build cloud

applications based on the TOSCA standard. After that, Section 4 will describe the thesis's practical part. It involves new technology architecture and technical implementation. Section 5 will give an overview of the project setup, the ML-TOSCA practical implementation in an actual project, and the results achieved. Finally, Section 6 introduces future work by identifying the missing parts and suggesting how they can be resolved.

Acronym	**Definition**
AI	Artificial Intelligence
API	Application Programming Interface
AutoML	Automated Machine Learning
CSAR	Cloud Service Archive
CSV	Comma-Separated Values
DAG	Directed Acyclic Graph
NLP	Natural Language Processing
ML	Machine Learning
MLOps	Machine Learning Operations
ModelOps	Model Operations
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine
YAML	Yet Another Markup Language

Table 1. Acronyms and definitions

In order to enhance the clarity and brevity of this thesis, a number of acronyms will be used to reference various technologies and concepts. By using these abbreviations consistently, the readability and conciseness of this thesis can be improved, making it easier for readers to understand the technical concepts presented. Please refer to Table 1 for a complete list of acronyms and their definitions.

2 Background

This section briefly introduces the automated machine learning process with its main stages and use cases. After that, the ML managing systems will be described. Finally, this section will provide an overview of orchestration tools, including their architecture and main concepts.

2.1 AutoML

Automated Machine Learning, or AutoML for short, is an area of artificial intelligence that focuses on automating processes from data acquisition to model evaluation. This is useful for non-experts who can build sufficiently powerful machine learning models without technical knowledge, while real experts can use their time for more priority tasks [18]. According to [27, 23, 38, 21] the AutoML pipeline consists of several stages that are traversed sequentially. However, the author has chosen to derive six main stages:

- **Data Collection** - obtaining the data needed for the particular area. This may be a user-supplied comma-separated values (CSV) file or requested data from an external database.
- **Data Preprocessing** - cleanup and reformatting of data by removing or replacing missing values, replacing categorical values with numeric values (which is also known as data labelling), enlarging data by creating synthetic data.
- **Feature Engineering** - process of creating new features or transforming existing ones into new ones, which in turn can improve the performance of machine learning models. The best-known algorithms are encoding, normalization, scaling, and feature extraction.
- **Model Selection** - once the data is prepared, the next important step is to select the right model for the job. This is best done by understanding the problem and assigning it to a specific type, such as classification, regression or clustering. If desired, the user can specify hyperparameters for the model, which can also improve the model's performance.
- **Model Training** - training of the selected model based on the processed data. Usually, during this process, the data is split into training and testing data so that user can later understand how well the model works with the same data.
- **Evaluation** - to perform this step, the user must have a test data set with correct answers. The goal is to give data to the model data and compare its prediction with the actual results.

AutoML involves a series of actions that can be customized to fit a given project's specific needs and goals. While core steps provided in a list are typically included, some can be skipped or added. The nature of the project will determine the approach and stages. For example, data visualization, model deployment, and monitoring are all potential steps to be added to a pipeline. Ultimately, the goal is the process automation of creating machine learning models, while the specific steps involved will depend on the project's specific requirements.

Automated machine learning also has its drawbacks. The user loses the freedom of choice and becomes bound to a technology provided by a service. One of the problems with AutoML is the need for more control over the system's capabilities and flexibility [23, 2]. It means not only that the choices can be minimal or some actions can even be overlooked but also that the user needs more control and understanding of how the steps are executed. In addition, there may be integration problems when the final model is ready, but it can not be exported or interacted with other systems.

2.1.1 Use cases

It may not be clear why such systems are needed and whether they have any utility at all. First of all, it should be noted that artificial intelligence is now penetrating more and more into various fields. However, there is a shortage of professionals who could develop AI.

AutoML is now used in the medical field [28] to make correct diagnoses or detect various abnormalities on X-ray images. Of course, medical professionals do not relinquish full control to the machine, but AI is a useful and sometimes invaluable assistant that can notice things that a human might not notice or even see.

Another application could be finance. The article [31] describes AutoML implementation in the area of stock price prediction. Instead of having to deal with and understand the structure and algorithms of NLP (Natural Language Processing), users can leave all the work to the model.

In summary, AI is applicable in many areas, if not all. However, various developments can often run into money. This means that the company can not afford to develop intelligence, as they do not know in absentia whether it will work well or not. In these cases, the decision to use AutoML is optimal; therefore, all further development depends on the data - namely, its quality and quantity.

2.2 Related work

Concerning automated processes connected with machine learning, the author decided to look into existing orchestrators for this area. An orchestrator is a tool that manages and coordinates the different components of a system to ensure that they work together

efficiently. Some popular orchestrators for this area include Apache Airflow, Apache Beam, Dask, Dagster, and Kedro, each with distinct features and benefits.

2.2.1 Managing machine learning systems

Managing machine learning systems has become more challenging and complex in recent years. It involves not only model building and training but also deployment, versioning, monitoring, and support [14, 40]. The model deployment to production requires much time, while it could be spent developing or improving the model itself. In addition, it is essential to ensure that models are reliable, accurate, and secure. In the developers' community, new directions, such as MLOps, AIOps, and ModelOps, have appeared taken from DevOps and applied to machine learning to eliminate this pain [36].

The paper [22] introduces the ModelOps (Model Operations) framework, a cloud-based lifecycle management approach for building and deploying artificial intelligence models. ModelOps focuses on managing AI and ML models throughout their lifecycle, starting from model development and ending with its deployment. The main goal behind that is to make sure that the developing process of models is flexible, best practices of operational principles are applied, and final models are easy to deploy, test and monitor. Another similar approach is MLOps (Machine Learning Operations) [35], which is being used interconvertible. It is considered that ModelOps is more general, as it is not only about machine learning models [36]. However, in our opinion, they represent the same meaning.

AIOps is an approach to increasing the efficiency and effectiveness of various operations and services with the help of AI and ML techniques [11]. Described service will help to predict the cost, quality, and error appearance. Thus, this solution is more about optimizing processes using the received data. It has nothing to do with the manipulation of various operations. However, when combined with MLOps, this will speed up the model deployment time and make valuable predictions and tips.

2.2.2 Apache Airflow

Apache Airflow is an open source platform for creating, scheduling and monitoring various pipelines and workflows [20]. Like other workflow managers, Airflow allows the definition of a directed acyclic graph (DAG) in which each job or task is defined as a node and the edges represent dependencies between them (see Figure 1). With this approach, different tasks can be executed both sequentially and in parallel.

Airflow DAGs are defined in DAG files with Python code. In other words, it is a script that defines a flow of executable functions in a predefined order. The predefined structure consists of the following parts:

- **Task operators** - any work to be performed in a DAG is represented as an operator.

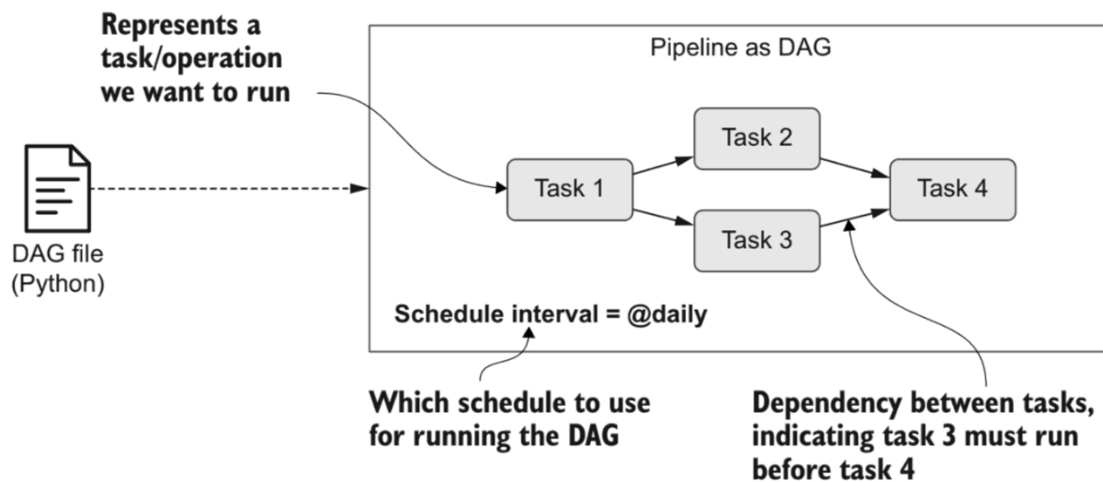

Figure 1. Airflow pipelines defined as DAGs [20].

It also defines the type of a task - for example, it can be a Python function or a SQL query.

- **Dependencies** - all tasks in a DAG are interconnected. These connections are called dependencies, which define an execution order. They allow building complex workflows and passing attributes and properties to each other.

- **Scheduling** - user can set up a scheduler that will execute tasks based on predefined scheduling rules (e.g. specific time, API call).

It is worth noting that the users must write the functions by themselves. And this, in turn, means that the user must have at least some minimal knowledge of Python programming and be familiar with the Airflow API.

2.2.3 Apache Beam

Apache Beam is another open-source framework for building data processing pipelines. Beam provides a software development kit (SDK), which allows building a pipeline that is also a DAG [25]. Besides, using SDK allows execution pipelines on different computing engines, which makes it flexible. The core concept is to define a sequence of manipulations and transformations that will be applied to a data source.

Beams pipeline consists of *PCollections* and *PTransforms*. The first one is a distributed dataset that will be processed by the pipeline. *PTransform* is a function that takes one or more *PCollections*, executes some preprocessing on the data and, as a result, returns one or more *PCollections*. Simply, *PTransform* is a function that maps one

PCollection to another. Additionally, Beam also provides the ability to write a custom *PTransform*, which means that the users can create their own complex pipeline for a specific task.

2.2.4 Dask

Dask is an open-source flexible parallel computing library [9]. All calculations and work executions are described as a DAG, which is built on top of the Python dictionaries and tuples. Dask's main benefit is that it can distribute computational power across multiple clusters. This allows the processing of larger datasets by splitting them into smaller chunks.

```python
# Numpy example
import numpy as np
a = np.random.normal(size = (1000, 10000))
b = np.random.normal(size = (10000, 10000))
answer = a.dot(b)

# Dask example
import dask.array as da
a = da.random.normal(size = (1000, 10000), chuncks = 1000)
b = da.random.normal(size = (10000, 10000), chuncks = 1000)
answer = a.dot(b).compute()
```

Figure 2. Numpy and Dask matrix multiplication example [9]

Dask also provides a high-level interface parallel of Pandas and Numpy packages. Developer can write the same code with the similar syntax (see Figure 2), while in the background are executed different data structures. Using Dask specific structures, user does not have to take care of distribution of the computations, it all is done by the system.

2.2.5 Dagster

Dagster [1] is also an open-source data orchestrator. As well as others, Dagster constructs acyclic graphs from solids and dependencies that as a result create a data pipeline. Solids are self-contained units of computation and edges represent the dependencies between solids. Such architecture makes sure that nodes will be executed in the correct order and that the correct inputs are provided to each solid.

[1] https://docs.dagster.io/0.12.0/concepts

```python
from kedro.pipeline import node, pipeline

# First node
def return_greeting():
    return "Hello"

return_greeting_node = node(
    func=return_greeting, inputs=None, outputs="my_salutation"
)

# Second node
def join_statements(greeting):
    return f"{greeting} Kedro!"

join_statements_node = node(
    join_statements, inputs="my_salutation", outputs="my_message"
)

# Assemble nodes into a pipeline
greeting_pipeline = pipeline(
    [return_greeting_node, join_statements_node]
)
```

Figure 3. Kedro pipeline comprised of two nodes [1]

Dagster also provides a web-tool, which provides a visualization of pipeline schemes and executions. User can see and validate intermediate results, inputs and outputs of solids. This is very convenient when pipeline becomes complex and some errors occur. Given tool allows to execute each function separately and debug whole pipeline.

2.2.6 Kedro

Finally, Kedro [1] is another open-source Python framework for building modular, scalable, and maintainable data pipelines. It provides a standardized method for building data science code that helps improve code quality and reproducibility. Development teams can use it to create complex pipelines that include multiple stages of data cleaning, pre-processing, feature engineering, model training and evaluation.

One of the core concepts of Kedro is nodes. Nodes are units of a pipeline that perform a specific task. Each node represents a function that takes input data and returns output

data. Nodes can be chained together to form a pipeline. A pipeline is a collection of nodes that describe the entire flow of data through the pipeline. The code implementation is shown in Figure3. It is important to note that the pipeline executes the nodes in the correct order depending on their dependencies. This means that the nodes can be executed in a different order than the one in which they were passed.

Another core concept of Kedro is the data catalog. The data catalog is a registry of all input and output data sources and their respective locations. It provides a standardized way to manage data, which simplifies the development process. The data catalog also ensures that data is easily accessible, versioned and documented.

Kedro also provides an integrated visualization tool Kedro-Viz [2]. It provides a visual representation of the pipeline structure, helping to better understand and debug it. The tool provides an interactive directed graph, where the nodes are represented as circles and the dependencies between them are represented as arrows. The user can click on them to get information about the nodes and their inputs and outputs.

[2]https://github.com/kedro-org/kedro-viz

3 TOSCA

This chapter introduces the specification of TOSCA and describes its technical aspects, structure, and execution flow. In addition, a RADON project which is built on TOSCA, is also described. Finally, there will be presented TOSCA alternatives.

3.1 Topology and Orchestration Specification for Cloud Applications

Topology and Orchestration Specification for Cloud Applications (TOSCA) is an open-source standard that aims to enable the creation of cloud applications that can be easily moved between different environments and to automate the process of deployment and management [5]. The standard in question was developed by the Open Standards for Cloud Incubator Group (OSCIG) under the Organization for the Advancement of Structured Information Standards (OASIS).

This specification enables users to model and manage cloud-based applications (such as IaaS, PaaS, FaaS and SaaS) - from provisioning to scaling resources - using a common language for all tasks. It enables users to automate processes, which in turn saves resources and time and generates fewer errors during the workflow.

For example, in FaaS-based (serverless) applications, TOSCA can be used to define functions, languages, API endpoints, triggers and relationships between them. The user can define a model that specifies which web resource the function can send requests to or which database it should connect to. These specifications and information allow you to automate the deployment process, making it many times faster and easier.

Another example is the implementation of TOSCA when modeling data-intensive applications. [19]. The engineer must decide which storage and stream processing components to use by ensuring communication between them. TOSCA is able to take control of the properties and attributes required for the entire system, including cloud resources and storage location. As in the previous example, the user wins for many reasons, from simplicity and convenience to the ability to change the entire system instantly.

The papers [12] and [13] describe data processing modeling and deployment using TOSCA language. This standard has been adopted because it simplifies the deployment and management of complex data pipeline applications. In addition, it enables users to integrate with other cloud services and technologies effectively. The authors proposed a TOSCA-based approach to model data-intensive applications as it demonstrated its effectiveness in applying business requirements to cloud services.

It is also worth noting that TOSCA is a standard that is not tied to a specific language, so it can be used with a variety of orchestration engines and cloud platforms. This feature makes the process of deploying and managing cloud-based services more adaptable and flexible. Despite the fact that the given specification has many advantages, AutoML, which is described in Section 2.1, is not yet implemented in TOSCA. As mentioned

earlier, TOSCA is capable today of describing the composition and orchestration of various cloud technologies and services. However, if developers want to build, train, and deploy a machine learning model to the cloud, they must do so manually. Consequently, there is a gap in TOSCA, when it comes to fully automating the machine learning process. This is an area where further research and development is needed to close the gap and bring the benefits of AutoML to TOSCA users.

3.2 Structure

TOSCA defines a meta-model to describe the management and structure of services, represented as *Topology Template* [32]. This is the main part where the user sets up properties, requirements and actions of a service to be considered during the service lifecycle. The template again consists of two parts: *Relationship Templates* and *Node Templates*, which in combination result in a graph of nodes.

Node Templates describe and define the properties, attributes, actions, capabilities and requirements of a service component. A node template is, in turn, an instance of a Node Type [32, 30] that provides a persistent way to model components. In other words, the Template is responsible for providing explicit values for attributes and properties, while the Type describes those attributes and properties, which can also be reused in multiple Templates.

The same is for the Relationship Template, where Template is an instance of Relationship Type [32]. The main goal is to describe the relationship between the Node Templates. For example, we can have two components web server and a database. For them to communicate with each other, we need to define the relationship between them by describing how and where they are connected.

There can be multiple relationship types such as *HostedOn*, *ConnectsTo*, *AttachesTo*, *RouteTo*, *DependsOn* [30]. It is also possible to create a custom relationship for a specific task. When developing the application, the user must take into account the fact that the relationship types execute the nodes in a predefined order. When two node types are connected, it also means that these node types have a matching capability.

3.3 Execution phases

The dependencies between nodes define the order of execution of relations and nodes in TOSCA. Regarding relationships, they have their lifecycle and can go through various execution phases parallel to the node execution order. For instance, a relationship can only be executed after the target node has been created and configured. The execution phases for nodes and relationships are shown in Figure 4.

Execution phases are essential to ensure that a TOSCA application's deployment flows reliably and predictably in provided order. By defining the order of operations, TOSCA ensures that each step is executed precisely and that the dependencies between

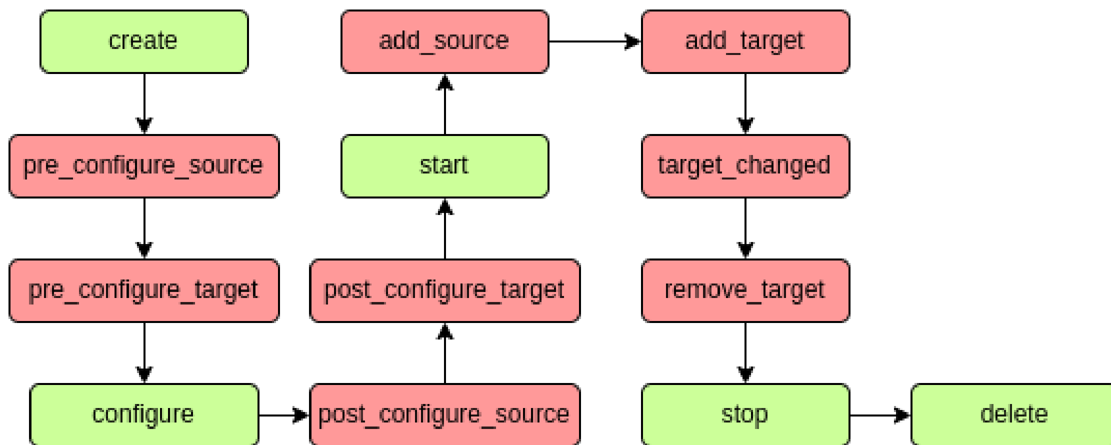

Figure 4. The order of execution phases for nodes (represented by the green blocks) and relationships (represented by the red blocks)

nodes and relationships are appropriately managed. In addition, execution phases can be extended or customized to meet specific deployment needs, making TOSCA a versatile and adaptable standard for deploying complex applications.

3.4 Workflow

As described above, the architecture of TOSCA is built like a graph, consisting of nodes and relationships between them. However, these components alone cannot interact and function. The workflow process is described in *topology* [33]. It is part of a template that can be parsed at TOSCA runtime to execute all services step by step.

Once a developer has created an application model and entered all the required data, it can be executed. However, the question arises as to how it should be executed. It should be noted that TOSCA uses YAML (Yet Another Markup Language) as the file format in which all components are described [32]. It can also be customized to meet the needs of a particular application. And, of course, the entire workflow process is described there, including all processes and required steps to be started during deployment. The workflows are executed through a TOSCA-compliant orchestration engine that manages the execution of the tasks defined in the workflow.

The TOSCA standard provides a Cloud Service Archive (CSAR), a standardized packaging format for cloud applications. It ensures that all necessary artifacts and templates are available. When an application is executed, an orchestrator manages and executes the application within a single environment [24]. CSAR archive contains multiple files describing the workflow, organized in hierarchical subdirectories. A mandatory subdirectory, TOSCA-Metadata [15], which should always exist within

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3
imports:
  - db.yaml
  - web.yaml
topology_template:
  node_templates:
    database:
      type: tosca.nodes.DB
      properties:
        db_name: postgres
        db_user: admin
        db_password: admin
    web_app:
      type: tosca.nodes.Web
      properties:
        app_name: web
      requirements:
        - database:
            node: database
            relationship: tosca.relationships.Uses
```

Figure 5. Example of a TOSCA file connecting a web application to a database

CSAR, describes information about the artifacts, including the path inside a container and the execution file format.

In Figure 5 you can see a simple example that describes all the necessary information for connecting a web application to the database. The script has *imports* that contains information about the node types, which are contained in a separate file with all the information about them. This script also has two node templates - database and web application, the latter of which also has a relationship requirement to the database. This, in turn, means that when the entire model is ready, the user only needs to run a single script file that sets up and creates all the services automatically.

3.5 RADON

RADON [3] is a project focused on developing a DevOps framework that encompasses all phases of building and deploying complex software in the cloud. In other words, the goal of this project is to accelerate the process of various cloud application actions by

[3]https://radon-h2020.eu/

providing RADON components. It is important to mention that OASIS TOSCA serves as baseline modeling language [7] for describing RADON models.

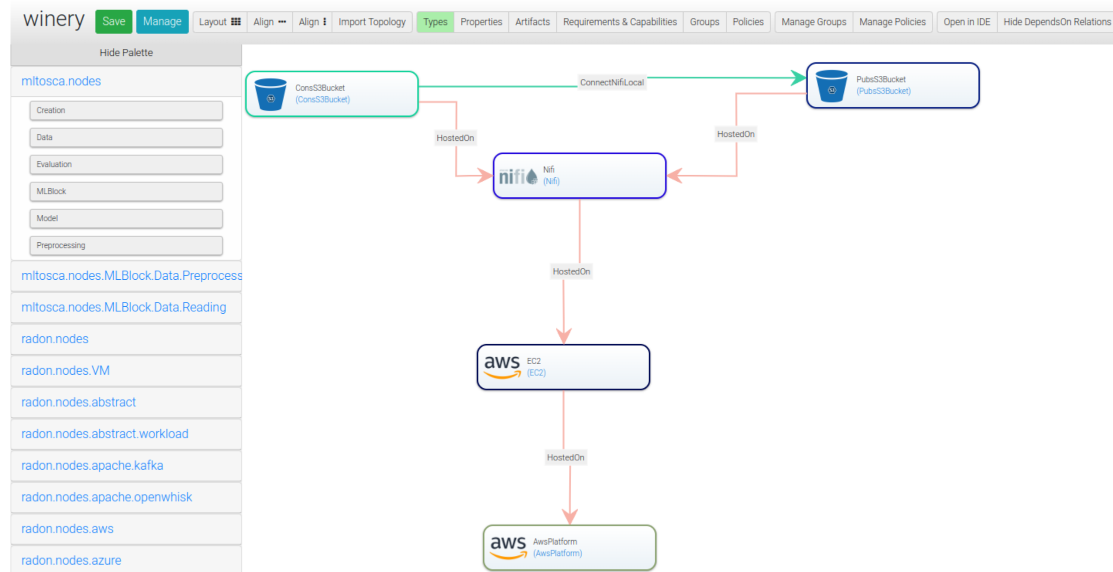

Figure 6. Eclipse Winery web interface

A model-based approach is used in this project to setup and control distribution and orchestration of modern cloud applications, where microservice architecture is commonly used [10]. Here, TOSCA is used and acts as an orchestrator of cloud services and also validates those components' topology. That is, it turns out that RADON is a kind of wrapper for TOSCA while giving the following benefits:

- **Graphical Modeling Tool (GMT)** - web-application (see Figures 6), which allows users to model the TOSCA application instead of describing components and their relationships manually. When the model is finalized, Eclipse Winery [4] (web-application) allows users to export their models as CSAR files [16], which include all necessary information to deploy cloud application.
- **xOpera Orchestrator** [5] - RADONS' Ansible-focused lightweight open-source orchestrator. It is responsible for executing scripts, including correct orders and models, that are located in the above mentioned CSAR file.
- **RADON Particles** [6] - a repository which is hosted on GitHub [16]. This repository consists of TOSCA blueprints, or, in other words, reusable components that are

[4]https://projects.eclipse.org/projects/soa.winery
[5]https://github.com/xlab-si/xopera-opera
[6]https://github.com/radon-h2020/radon-particles

predefined by the organization. It enables users to use and combine them in order to create a variety of cloud applications. It also is organized in a way so that it fulfills the GMT requirements so that, for example, users can easily connect to an Azure virtual machine and start a NGINX server in just a few clicks.

RADON also includes a *verification tool* [10] that checks whether the designed model complies with the constraint sets (e.g., pattern violations, security, missing inputs or attributes) before the deployment is executed. If the DevOps engineer violates constraints during system modeling in GMT, the system notifies him. In addition, the engineer can review an artifact generated during model creation and use it for troubleshooting by reviewing the logs, which provide more detailed information about errors that occurred.

With these features, RADON also allows for the creation of custom components. Since this project largely describes services related to DevOps, it simply lacks some components that are also needed for software development. One of the missing areas is machine learning blocks.

3.6 Alternatives

While TOSCA is a widely accepted standard for automating processes, other solutions offer similar functionality. Among the most popular are Terraform[7], Cloudify[8], Puppet[9], Chef[10], and Ansible[11] [7]. Each of these technologies has its strengths and weaknesses, so it is essential to understand how they compare to TOSCA.

Terraform uses a declarative approach to infrastructure as code, defining the desired state of infrastructure and managing necessary changes automatically. Cloudify, built on top of TOSCA, provides a higher-level abstraction than TOSCA and includes additional features like support for multiple clouds and Kubernetes integration.

Puppet and Chef are configuration management tools that can be used for infrastructure automation using a procedural approach to configuration management. Finally, Ansible is another popular configuration management tool that uses a declarative approach. It is designed to be lightweight but less powerful and extensible than TOSCA.

Overall, the decision to use TOSCA as a starting point for process automation was based on several factors, including its status as a widely accepted standard and flexibility. TOSCA can describe a wide range of services, not just cloud applications, making it flexible. TOSCA also provides a high level of abstraction, so the user does not have to dig deep into the details of the components. Finally, it can extend beyond a specific technology.

---

[7] https://www.terraform.io/
[8] https://cloudify.co/
[9] https://www.puppet.com/
[10] https://www.chef.io/
[11] https://www.ansible.com/

4 ML-TOSCA Methodology

This section provides a precise technical overview of all stages involved in the practical work. It covers the algorithms, technologies, and ideas implemented throughout the project and any difficulties encountered. Additionally, the section delves into the project's technology architecture and technical implementation, providing detailed insights into the underlying system design and implementation. Finally, the author will outline the project's advantages, highlighting the key benefits that the project brings.

During the writing process for this thesis, the author used the Grammarly[12] tool, an online grammar checker and writing assistant. Grammarly helped identify and correct grammatical, spelling, and punctuation errors and provided suggestions for improving the clarity and coherence of writing. Also, during the thesis writing, ChatGPT[13] was used. This tool was used for text rephrasing and certain concept clarifications, mainly in the background and TOSCA sections.

4.1 Research design

The work began with introducing the technology TOSCA, namely familiarizing with the documentation and various articles on the subject. In parallel, the author began to solve simple data science tasks, which are described below. Furthermore, ultimately, to ensure that the author understands the working principle of TOSCA, several simple components were created, connected, and performed the most specific functions.

Since the topic is directly related to machine learning, it was decided to solve four tasks [14], two of them related to classification and the other two to regression. These tasks clarified what transformations and data processing are repeated, including the actions associated with the machine learning models, so that it becomes clear which components these tasks can be replaced. The tasks were solved to get a clear indication of what should facilitate and speed up this work.

The tasks were solved using the Python language. The reason was that it has a vast collection of libraries and frameworks specifically designed for data science. Each task imported the external libraries Pandas, XGBoost, and Scikit-Learn. These libraries were chosen because they are among the most commonly used in data science and have become industry-standard tools. The author decided that the creation of neural networks would not be implemented in this project but would be left for further development. In addition, the author has decided not to develop a component for data visualization, as this is already a separate topic and needs to be studied separately.

[12] https://www.grammarly.com
[13] https://openai.com/blog/chatgpt
[14] https://github.com/chinmaya-dehury/TOSCA-ML

4.1.1 Technology stack

This project aimed to create automated machine-learning components using the TOSCA specification, an open standard for describing cloud services. Several tools and technologies were utilized to achieve this goal, including the Python programming language and the Ansible orchestration engine, both of which were used to automate the deployment and management of services. To facilitate automation and orchestration, XOpera was installed, a cloud orchestration tool that supports the TOSCA specification. This tool allowed automation of the deployment and management of services, ensuring they were configured and optimized for specified use cases. For developing a particular service, it was necessary, among other technologies, to install Docker [15], which in turn served as a platform for hosting Eclipse Winery.

Throughout the project, were used the following versions of the tools: Python 3.8, Ansible 2.12, XOpera 0.6.9, and Docker 20.10.16. The author found these tools reliable, efficient and well-suited for the current project.

4.1.2 Data storage format

Before implementation, the author considered how to handle the intermediate data that would be transmitted and read by different nodes. In this work, 'data' refers to both the input data and the trained model, which can be saved as a file and transmitted as data. To ensure compatibility and ease of handling, the author decided to store all data in a unified format using the pickle [16] library in Python. This way, whether the data is a file or a data object, it can be saved and transmitted consistently.

Pickle is a module that can convert Python objects into a binary format that can be saved as a file. The Pickle module can handle almost all Python objects, including lists, dictionaries, functions, and classes. Of course, it is worth mentioning that these pickle files can be reconstructed into the original objects.

Number of rows per file	**100 000**	**1 000 000**	**10 000 000**
CSV write time in seconds	0.823	8.625	81.819
Pickle write time in seconds	0.008	0.066	0.68
CSV read time in seconds	0.145	1.07	8.883
Pickle read time in seconds	0.007	0.052	0.502
CSV file size	10.2 MB	103 MB	1.04 GB
Pickle file size	4 MB	40 MB	400 MB

Table 2. Benchmarking results in CSV and Pickle files comparison

[15] https://docs.docker.com/get-started/overview/
[16] https://docs.python.org/3/library/pickle.html

Below will be mentioned that also all CSV files will be converted to the Pickle format. The reason was several factors that can be seen in Table 2. This experiment was carried out by the author of the work [17]. It can be seen that Pickle is better in all aspects, so the author decided to read the CSV file only once, rewrite it into Pandas data frame object, and save it as a Pickle file.

4.2 ML-TOSCA architectural foundations

After reviewing the projects mentioned in Section 4.1, the author concluded that these projects could be classified into four major subgroups: Data Reading, Data Preprocessing, Model Training, and Model Evaluation. This categorization was based on an analysis of each project's technical details and requirements, which revealed that they shared similar characteristics and functionality. However, the main difficulty is their combination and the ability to transfer data between the different stages of the machine learning workflow, such as from Data Reading to Data Preprocessing, from Data Preprocessing to Model Training, and from Model Training to Model Evaluation.

To get a complete picture, the author has created a diagram that visually shows and explains which components exist and to which subgroups they belong. Figure 7 shows that this architecture has a tree structure - each component has a parent component and can also be a parent component for child components. Parent components allow for the description of properties inherited by child components. At the same time, only the components that are leaves of the tree (purple squares) are directly relevant for solving machine learning tasks and can be used in conjunction with Eclipse Winery to create and manage these tasks efficiently.

The task was to create autonomous and, at the same time, compatible nodes that do not have to wait for the execution of a particular task. In this case, the block responsible for creating and training the machine learning model does not have to wait until the data is prepared. The block may already start preparing and creating the model and be ready for the data to arrive at some point. In addition, it was necessary to figure out exactly how the communication between the components would take place and how and which nodes could be connected.

4.2.1 ML-TOSCA architecture

The main advantage of this project is that it is open for additions. Although the main components have been described in Figure 7, they are more specific to the existing solution. However, this project is open to more than just these nodes, which can be added occasionally. It consists of abstract components on which the entire system is built.

[17] https://github.com/chinmaya-dehury/TOSCA-ML/blob/main/scripts/csv_pickle_benchmark.ipynb

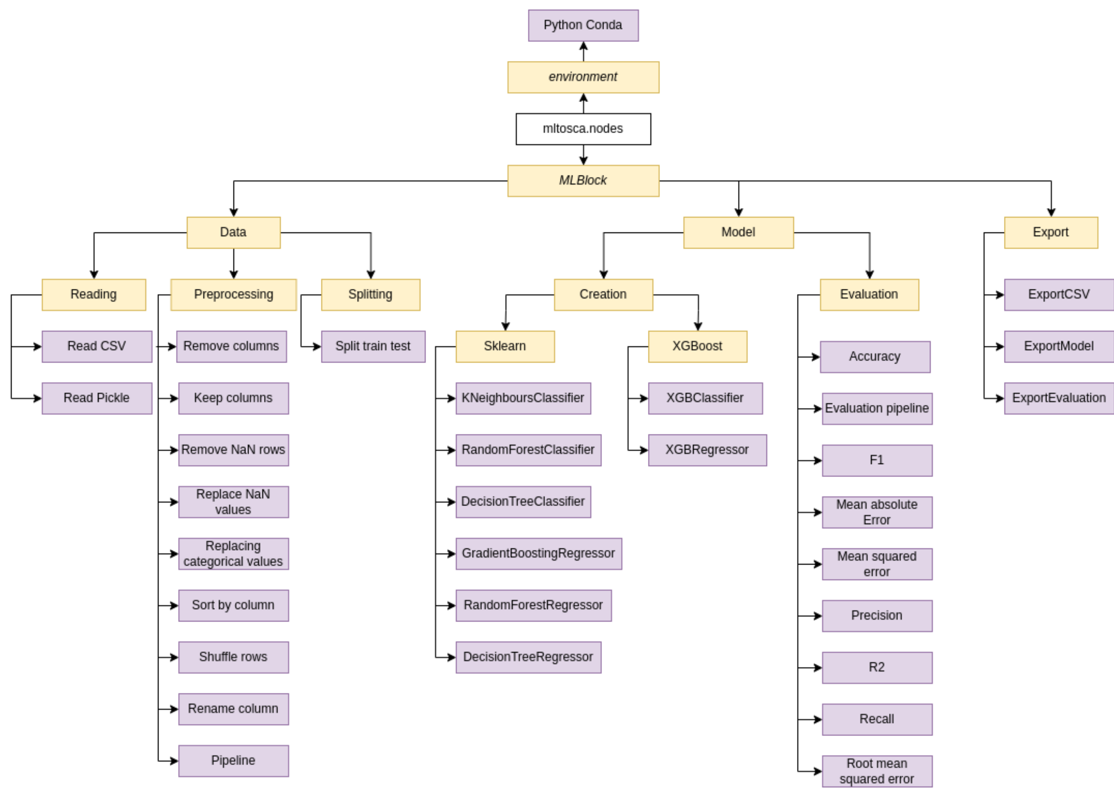

Figure 7. ML-TOSCA key hierarchy. The yellow rectangles represent parent components, while the purple rectangles represent the final components or leaves.

By looking at Figure 8, it can be seen that, similarly to key hierarchy, all components depend on some parent node. At the top level is the computational resource, which executes the script on the machine. On top of that comes a virtual environment. This component serves an important role, as it, in turn, provides a development environment, delivering everything necessary for child components. It means that in the further development of components, taking care of any required technologies or libraries will not be necessary.

The following upcoming node types are unique and will be designed for specific tasks. Each node is responsible for solving its concrete task and does not depend on its neighbors. For example, the leftmost part of Figure 8 is responsible for reading the data and does not depend on data splitting. Data can be read from the local machine or any remote location. A data processing block follows the reading. It consists of a plugin created to build up various preprocessing algorithms. The last block associated with the data is responsible for splitting the data into subsets using various strategies commonly employed in data science. Each of these strategies serves a specific purpose and can be

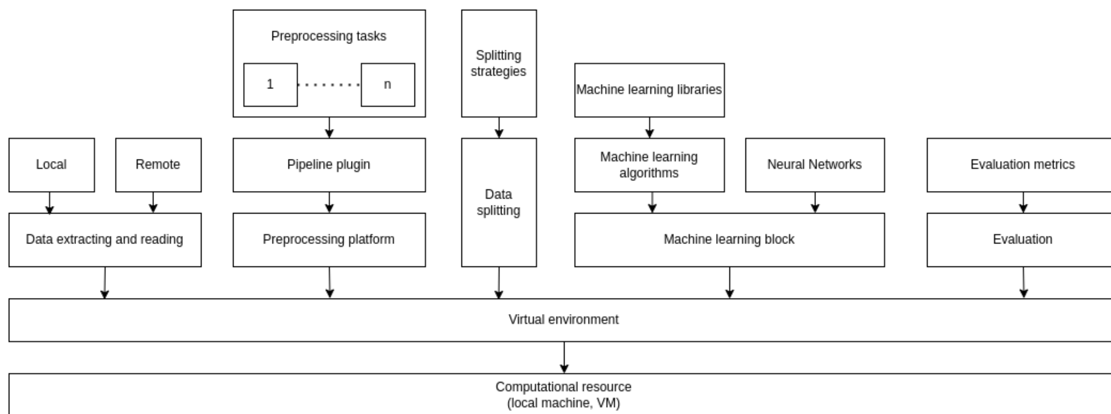

Figure 8. ML-TOSCA architecture

tailored to the specific requirements of the downstream tasks.

The remaining two parts are responsible for machine learning models. The first and the most complicated is a model creation module. Machine learning is divided into machine learning algorithms and deep learning. The current solution has only a "traditional" machine learning algorithms implementation, but there is a space for neural networks. In turn, algorithms also have different variations of implementations. All these parts are independent but can solve different problems in combination.

4.2.2 Environment

One of the most critical requirements for this project was to provide a seamless user experience. From the user's point of view, they should not have to download or install anything additional when running a program or script. This requirement is essential because it ensures that users can easily access and use the tool without encountering technical barriers. To achieve this goal, creating a common platform or environment that would automatically install and load the necessary dependencies to run the final program was necessary.

Since all nodes are implemented with Python scripts, providing an environment that supported Python in a particular version was essential while installing all necessary libraries. Of course, one could rely on Python already being installed on the machine, but this has several problems. First, Python may have an old or inappropriate version. Second, the required libraries may have the wrong version or not be installed. However, changing the version is also not the best solution since the user can use a specific version for a specific project. Therefore, to address these challenges, *ML-TOSCA* supports the creation of Conda-based [18] virtual environments, allowing users to efficiently manage the

[18]https://docs.conda.io/en/latest/

required Python version and libraries needed for their specific project without interfering with other Python projects that may require a different environment setup.

A *Python virtual environment* is a self-contained directory tree that contains a Python installation and a set of packages and dependencies [26]. It allows developers to create an isolated workspace with its own Python interpreter separate from the global Python environment. This separation is practical when working with multiple projects that require different versions of Python or third-party libraries. At the same time, Conda is an open-source package and environment management system responsible for installing, updating, and sharing software packages. In other words, Conda allows the creation of Python virtual environments by specifying the Python version the user wants to use [39]. The virtual environment created by Conda behaves similarly to a Python virtual environment but is more potent because it also allows managing non-Python packages.

Therefore, the first step was to implement this type of node. As shown in Figure 7, this node is separate from the machine learning block. The fact is that the environment itself is applicable not only in data science but also in other fields, making it platform agnostic. In addition to creating a virtual Python environment with Conda, adding other environments for different technologies is possible. For instance, creating environments for other programming languages such as R, Node.js, C/C++ and others should be possible. As mentioned earlier, this approach allows the use of the same Python version and the required versions of the libraries for all scripts. Therefore, this component is the foundation for all other nodes. Each subsequent node must be connected to it.

4.2.3 Preprocessing

This part was one of the most difficult and, at the same time, crucial in this project. In itself, the data processing means that the program receives, processes, and outputs the processed data in the same format. However, the problem arises when the file to be processed is enormous. Assuming that the processing consists of n different steps, the file is loaded, processed, and overwritten each time. This is not the most optimal solution, although the user may not feel the difference if the file being processed is small.

In this project, the data processing module comprises several autonomous blocks that do not rely on each other. To ensure efficient data processing, it was necessary to find a solution that would allow loading data only once and writing it only once. After thorough research, it was concluded that the Pandas Dataframe [19] structure would be the most suitable tool for data analysis in the Python ecosystem. Pandas is a widely used library that offers a powerful function called *pipe()* [20], which can perform multiple operations on a DataFrame object without creating temporary variables or modifying the original DataFrame. This feature allows for a certain number of modifications to be

[19] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html
[20] https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pipe.html

made to the data without loading it several times, thus improving the efficiency of the data processing module. The problem was how to combine these same node types and, in addition, save the sequence arranged by the user.

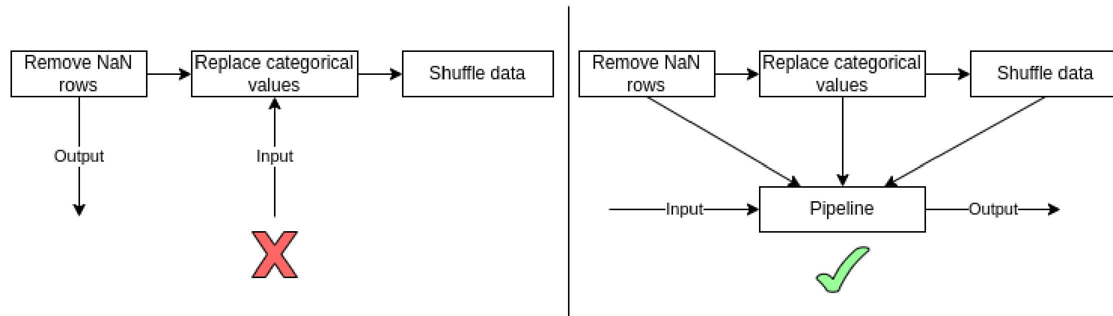

Figure 9. ML-TOSCA data preprocessing schema. An incorrect solution is shown on the left side, where the user can violate the sequence. At the same time, a solution is shown on the right side that forces the user not to break the flow.

From such a description, it is emerging that there are too many responsibilities for each node, but simultaneously, they are similar. That is, each node must be able to perform its unique task related to data processing, but at the same time, it must be able to understand the execution sequence and transfer and store data. Continuing this list, the last component must understand that it is the last and has to create a pipe that collects and runs all the functions. Hence, all these components must have a parent component that should perform generalized tasks, such as reading and writing data and building the correct order.

It was decided to create a basis component responsible for all the essential functions. At the same time, all child nodes should directly depend on it and transfer the necessary information related specifically to themselves. Also, looking more globally at this solution eliminates the possibility of interrupting the chain of actions. Having such a component, other components pass input data to one place, and the following components know they will receive data from the same place. Thus, we exclude the possibility of interrupting the chain and reading the processed data from several places, shown in Figure 9. This node is also shown in Figure 7, called *Pipeline*, and belongs to the *Preprocessing* parent component.

4.2.4 Model creation

The choice and creation of a machine learning model are crucial links for predictions. Choosing a suitable model with the proper parameters increases the model's accuracy dramatically. The current work implemented two machine learning libraries - XGBoost

and Scikit-learn. Each of those libraries has models for classification and regression tasks.

Each model is unique and has parameters that may not be repeated with other models. Therefore, in this case, it was unnecessary to find all models' common functionalities and attach them to a single block that provided some execution logic. However, the parent component is still needed to describe similar properties inherent in each model. In this case, the main difficulty was determining the format of the received data. The previous node could create one final file or several (for example, training and testing), assuming that the machine learning component should understand how to dispose of them.

4.2.5 Model evaluation

Moreover, finally, having a ready-made trained model, the user should be able to evaluate it. There are various metrics for this. However, to evaluate and obtain any result, we need to have data on which the prediction should be made. To be more precise, the data should be in the same processed form as during training.

There were several difficulties with this implementation. First, it was necessary to decide how to make it so that the same metrics could be applied to several models. For example, if the user wants to know the accuracy, recall, and precision, then it is enough to connect the models with only one block that would be responsible for the results. In this implementation, a significant difference was that the sequence of metrics in the final result was unimportant. Also, an essential integral part is the export of the results; however, this will be described afterward in the technical description.

Another important part was the transmission of the location of the data that was used to train the model. In data science, there is a concept of splitting data into training and test data. It makes it possible to evaluate the model on new data and determine how well it can generalize to new data. In a given project, passing the location of the training data, the evaluation component must be able to find the data intended for the test.

4.3 Technical implementation

The last section gave brief but, at the same time, architectural explanations without delving into technical details. This part will give a more detailed explanation of the implementation of nodes and relationships. Namely, the workflow of the nodes, their life cycle, properties, and attributes will be described, as well as an algorithm for transferring data between nodes.

4.3.1 Conda environment

As mentioned above, to create a single environment, it was decided to use Conda, which allows the creation of a Python environment of any version. Since the libraries used to

require a minimum version of Python 3.8, the author decided to use the latest provided by Conda.

The purpose of this node type is to directly create an environment if it does not exist on the host machine. It is known that to create an environment in Conda, Anaconda should be installed on the computer. As a property, it takes only possible environment name, which by default is *mltosca*. Also, at the beginning of the work, the author decided to install all kinds of libraries used in machine learning; however, they will not necessarily be used in one blueprint. Therefore, it was decided to transfer this logic to nodes, which directly need one or another library.

Another important task of this component is to create a unique name for the newly created blueprint. The work is based on mutual communication between nodes. All the needed data must be in the exact location for this. Data refers to the configuration files that are created when creating any node. For this reason, it is vital to separate project data from others without allowing them to overlap. It was decided to create a root folder containing the environment's name and the start time. For example, *~/mltosca/170323_11-50-39*, where the first part of the number until underscore represents the current data (in this case, it is 17th of March 2023) and another part represents execution time (11:50:39). Such a structure gives uniqueness since two blueprints cannot be launched simultaneously on the host machine.

All other nodes in the exact blueprint must be associated with the environment component. This means that other nodes must be connected to it using the *HostedOn* relationship. This rule is also programmed; the finished script will not start throwing an error if violated. It is also necessary so that all subsequent nodes know where to write and where to read data. This realization is implemented using the *project_location* node attribute, which is available to other components.

4.3.2 Data reading

The first step in data science is, of course, getting or reading data. In this project, two nodes are implemented that read data. The difference lies in the format of the read file. In one case, it is CSV, and in the other pickle.

Both components have similarities in important properties - they need to know the file's location to be read. Since both nodes are inherited from the same parent component, it was decided to move this property there. The difference lies in the additional parameters required to read the file. For example, the node responsible for reading the CSV also asks the user for the delimiter and encoding of the file, which is shown in Figure 10. It can be noticed that the properties are merged into a single parameter, which the Ansible script will use as input. Ansible will run the Python script using the environment described in the last section, using the same inputs. In its turn, Python firstly creates *configuration.json* file; the name of the newly created file will be described there. The next step will be splitting the input parameters and reading the content into a pickle file.

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3
node_types:
  mltosca.nodes.Reading.ReadCSV:
    derived_from: mltosca.nodes.Reading
    metadata:
      targetNamespace: "mltosca.nodes.Reading"
      abstract: "false"
      final: "false"
    attributes:
      function_id:
        type: string
        default: { get_attribute: [ SELF, tosca_id ] }
      parameters:
        type: string
        default: {concat: [{get_property: [SELF, delimiter]}
                          , "#_#"
                          , {get_property: [SELF, encoding]}]}
    properties:
      delimiter:
        type: string
        description: The character used to separate the values
        default: ","
      encoding:
        type: string
        description: The character encoding used in the CSV file
        default: "UTF-8"
```

Figure 10. Example of a TOSCA file connecting a web application to a database

However, the question of where the configuration and read files are saved may arise. Again, in the last section, it was described that the environment creates a unique name for the root folder and passes it to all connected nodes, including the read node. In addition, all nodes have the *function_id* parameter (see Figure 10), setting up a unique name for it inside a blueprint. This parameter serves as a unique name for the folder in which the created files of the read node will be stored. According to the previous example, the newly created folder location will be *~/mltosca/170323_11-50-39/ReadingCSV_0_0*, which will contain two files - *configuration.json* and *df.pkl*.

It is also worth noting that the final file format will be a pickle. Again, referring to Table 2, this format allows one to read and write data many times faster while winning in the size of the saved file. In the case of a CSV, the file is read into the Pandas data frame and saved as a Pickle. Moreover, if the transferred file is already a Pickle, then it is copied to a new location.

Also, an important point used in each node - when creating a final file (for example, df.pkl), a file with a different name is initially created, different from the name written

into the configuration file. It is done to ensure that other nodes waiting for the file will only read it after the whole data is written into it. Otherwise, the data writing process into a file may only be halfway, but the next component starts reading it. This can lead to various anomalies or missing data.

4.3.3 Preprocessing pipeline

Data processing is one of the largest and most complex families of nodes. As mentioned in subsection 4.2.3 given solution should consist of one main block, to which all other direct processing blocks will be attached. Before going to the more technical part, it is worth noting that just like the previous node, this node must have a *HostedOn* relationship, and so have a *DependsOn* relationship with the previous node. In this case, the previous node was the data reading node. This means that we depend on data, and preprocessing pipeline will not be executed before the data reading node is started.

Maintaining the correct order of operations is essential when building a data preprocessing pipeline. For instance, if the user wants to transform data into a numerical representation, it should be completed before any subsequent operations depend on it. However, maintaining the correct order can be challenging, as discussed in subsection 4.2.3. One issue is knowing when the last node from a given pipeline was launched. One possible solution is to call the callback function on the data file directly when creating any processing node. This way, the processed file can be transferred to the next node for further processing. Nevertheless, this approach may not be the most efficient.

Again, the correct order is implemented using a *DependsOn* relationship. It turns out that by combining the necessary nodes for the user, the order of the called blocks can be written into a file, where the function's name from top to bottom describes the order. In this case, it will be possible to reproduce the given queue, but it will be necessary to comprehend how to understand that at some point, the pipeline has reached the end, and it needs to run all the functions. The author thought that if it was possible to know the number of functions in advance, the primary node could check the file into which the correct order is written and verify whether the number of lines matches the number of preprocessing nodes. This would provide an additional check to ensure that all functions in the pipeline are executed in the correct order.

It was decided to do this with the help of a custom relationship called *HostedOnPipeline*. This was specifically designed to ensure that all those functions that connect to the main pipeline write their function names to a separate file. This way, the parent pipeline node can figure out how many nodes there will be before running the *DependsOn* relationship. To do this, the pipeline itself counts the number of lines and runs a script in the background to check the number of lines in the file describing the correct order.

When the specified number of nodes is reached, the same script reads the names of all functions from the file responsible for the queue while maintaining the sequence. In

```
remove_columns-|-PassengerId,Name,Cabin,Ticket
replace_nan_values-|-Age-|-mean-|-None
remove_nan_rows
replace_categorical_values-|-Embarked
replace_categorical_values-|-Sex
shuffle_rows
```

Figure 11. Example of a data preprocessing order file

addition to the name, some functions have mandatory parameters passed in the file. All parameters are separated by a special separator which can be seen in Figure 11. This example shows that first will be executed *remove_columns* function, which takes one argument. Secondly, will be executed *replace_nan_values*, which takes three arguments.

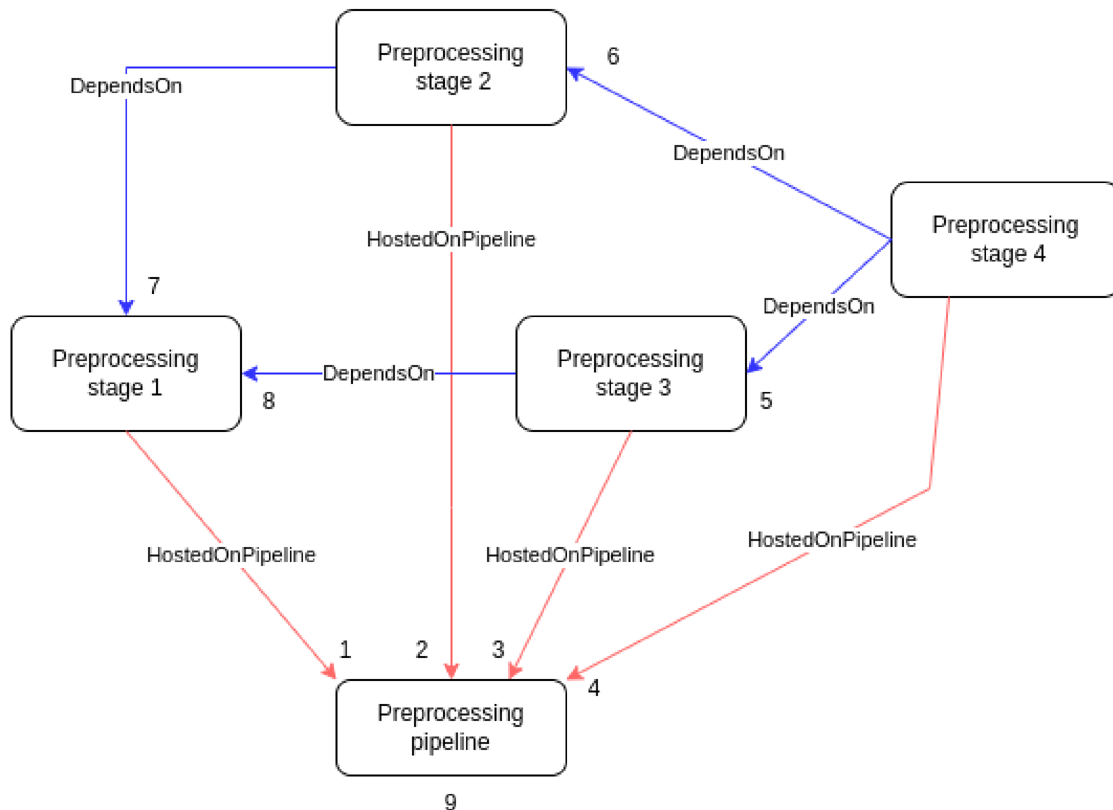

Figure 12. Example flowchart of the preprocessing stage.

After gathering all the information regarding the functions, they are collected in a

single pipe. As mentioned above, this solution will run all functions simultaneously and only once per data frame from the previous node. Following successful processing, the data will be saved in the same way to a new file, which will be ready for further processing or reading.

To better understand the preprocessing flow, take a look at Figure 12. The correct execution order is provided with the numbers at each arrow's end. Primarily, preprocessing stages tell to parent node about their existence (the number from one to four). Next, the same stages set up the correct order (the numbers from five to eight). Finally, the parent node, which knows everything about stages, will execute them.

4.3.4 Data splitting

Before training a machine learning model on the prepared data, we must consider how to evaluate it in the future. Usually, the data is divided into training and test data. There are other strategies; for example, in addition, to testing data, evaluation data is also created (hold-out method). This allows for fine-tuning the model while not checking the metrics on the final test data, avoiding overfitting. Alternatively, an equally popular method, K-Fold cross-validation, divides the dataset into k subsets. In this case, the model is trained on k-1 subsets and evaluated on the remaining subset. This process is repeated k times, averaging the performance metrics over the k runs.

In this project, only the first method was applied - data division into training and testing. For this, one node was used, which asked the user for one property: the percentage of the test data to the train data, which by default is 0.2. Specifically, this node takes this argument and creates two files defined in the configuration file.

4.3.5 Model creation and training

Finally, when the data is processed and prepared, the machine learning models can start training. Just like the others, these nodes must be connected to the environment and also have a dependency on the previous node that passes the location of the configuration file. It is worth mentioning once again that the task is to create components that are independent of each other, which, in conjunction with each other, can solve problems. Therefore, the configuration file is the only link between the various components, which provides all the necessary information about the work accomplished.

First, by running the script, the model node will wait until the file for training is ready. To do this, as in other components, an infinite loop is launched that checks for the existence of a file every second. Once the file is ready, the script creates the model specified by the user and the parameters that can be set for each model. The author has tried to place certain restrictions; however, the primary responsibility for the correctness of the parameters provided is the user's responsibility. Thus, having received the necessary file for training, the script separates a column from it, which the model

must learn to predict. The user must also specify the name of the column in properties. Finally, the model training starts.

At the end of the model training, the model is converted to pickle format and saved to a file. Meanwhile, this node also keeps the file's location used for training. This will be necessary so that the node responsible for the evaluation can take the same processed data but already a file intended for testing.

4.3.6 Trained model evaluation

Having a ready-trained model user should be able to evaluate it. This naturally requires a model and data with the same appearance as the training data. The evaluation node depends on the node responsible for creating the model and, like the others, must be hosted on the environment node.

In general, there are a large number of metrics. Users must also consider that some metrics are suitable for classification models, and some are only for comparing regression models. In the *ML-TOSCA* project, are implemented eight metrics in total for both situations (4 for classification and 4 for regression). However, given that this project uses directed acyclic graph implementation, it can be assumed how many links need to be laid if the user wants to combine four different models with four identical metrics (four models and four metrics give in total 4x4 = 16 relationships). The author thought that this solution in this situation is to create a parent component to which all metrics will be attached. Similarly, a data processing pipeline was implemented.

Having a parent node reduces the number of relationships and allows more logical group metrics, which is more important. That is, the user can group data depending on the type of models and the selected metrics. Delving into more technical aspects, in this case, the order of metrics is not essential, eliminating unnecessary connections between them. It is enough to determine the total metrics using the *HostedOn* connection and run the evaluation script.

To connect the metrics with the parent metric pipeline, a custom relationship was used so that when they were connected, the used metrics were written to a file for storage. That is, the task execution phase was programmed in such a way that the data of all associations were first read, and only after that the main script was launched. It is essential to mention that Ansible is used in launching the script itself (as in all other notes). To launch any script, Ansible first creates a unique folder whose name is the unique id of the node. Then, depending on the Python script being run, the necessary libraries are downloaded to the environment if they are not present. Only after that, the Python script is launched in the background not to slow down the entire orchestration.

Thus, the primary node responsible for the metrics and hosting the functions of various metrics also connects with the models. This relationship is also custom and is called *DependsOnModel*, inherited from *DependsOn*. When this relation is connected, a script is also launched that writes the connected model name to the file and, necessarily,

```
DecisionTreeClassifier_0_0#_#SplitTrainTest_0_0#_#Survived
RandomForestClassifier_0_0#_#SplitTrainTest_0_0#_#Survived
KNeighborsClassifier_0_0#_#SplitTrainTest_0_0#_#Survived
```

Figure 13. Example of a file storing model names, data folder location, and column for the prediction

the name of the folder from where the model took data for training with the column name that should be predicted. An example of the file structure is also shown in Figure 10. It can be seen that specific characters (#_#) separate all three mandatory parameters.

Having all the mandatory information needed, the script, which was running in the background, can start evaluating models. The program first reads the file with the models, namely folder names where the models are located. After that, a configuration file is read in each folder, which describes the full location of the models which will be uploaded. In parallel, the data folder is loaded, where the file for testing is located. Then, one by one, each model makes a prediction. The results obtained are compared in the user-specified metrics algorithms, and the final results are written to separate files, where the file name corresponds to the model name.

4.3.7 Result exporting

Working on data science projects, a programmer can look at the processed data at any time, evaluate it from a professional point of view, or even send it to the client. In addition, of course, the developer has the opportunity to save the trained model in order to use it later. In the given project, it was also important to provide similar functionality.

Using *ML-TOSCA*, the user can export three things - data (downloaded, processed, split) in the CSV format, machine learning models, and the results obtained during model testing. To apply this functionality, export nodes must be connected to those that produce the data or objects the user needs, whether a data processing pipeline or a KNearestNeighbours node. These nodes will similarly run in the background and wait for the necessary files to appear. The only required property the user must specify is the exported file's location.

4.4 Features and benefits

This work is unique in the TOSCA area. The uniqueness of ML-TOSCA lies in the used technology but also the functionality provided. It has many advantages and can be helpful in numerous aspects. Below is a list of all the comprehensive features implemented and ready for execution.

- **Machine learning democratization** - machine learning has become even closer to everyone, depending on whether the user is a specialist. Offering a user-friendly interface, anyone interested in data science can get started. The user should not get bogged down in the technical details of the programming language or algorithm. It is enough to understand the steps to achieve the final goal. The whole technical part will be implemented within this technology.

- **Time-saving** -ML-TOSCA has the potential to save a significant amount of time by simplifying the process of creating machine learning pipelines. Rather than writing code from scratch, users can select and configure pre-built blocks using a user-friendly interface. The modular design of the system and its drag-and-drop interface help speed up development and reduce the time spent on repetitive tasks.

- **Simplicity of use** - This project provides a user-friendly tool for simplifying the process of data preprocessing and building and training ML models. While a learning curve may be associated with using the tool, it is designed to be intuitive and easy for experts and non-experts. Users can quickly and easily build a complete machine-learning pipeline using an intuitive graphical user interface.

- **Modularity and reusability** - speaking of the technical implementation, this work is also notable for the fact that all components were developed in an interoperable and reusable way. It allows us to build and supplement existing components rather than starting from scratch each time. That is, adding new independent components from each other is possible while not changing the existing ones. As a result, it speeds up the software development process, enhances system functionality, and improves code quality.

- **Flexibility** - based on the previous point, the more developers build components in ML-TOSCA, the more tasks can be solved. It means this project is not restricted to any specific technology or problem. The user, potentially, can solve any problem that can be solved using a machine learning approach. It makes this project flexible and platform agnostic.

- **Process automation** - eventually, the final benefit, which in one way or another relates to all others, is the automation of all processes. Using ML-TOSCA, the user does not have to write a single line of code. Each process that should be executed is done automatically in the background. It saves significant time and reduces error occurrence, ultimately leading to higher data quality and better model performance.

TOSCA has many advantages. Nevertheless, it is worth noting that all the features turn into advantages when the user has at least some idea of the world of machine learning

and its algorithms. In other words, this tool becomes powerful in the hands of someone with theoretical knowledge about ML but is not strong in its technical implementation.

5 Demonstration of ML-TOSCA in action

In this section, the author will explain and show an implementation of *ML-TOSCA* nodes and their relationship in the example of an actual project. For clarity, the project will solve the classification problem, including all neccesary steps to train a machine learning model. It will also explain the basic rules needed to run the entire application.

5.1 Project setup

Since this project is not part of the official repository at the time of the release of the thesis, the first step is to download the project from the custom *TOSCAML* repository as shown in Listing 1.

```
$ git clone https://github.com/TOSCA-ML/tosca-ml-models.git
```

Listing 1. Command for cloning *ML-TOSCA* repository to local machine

The user can run the project in a docker container using the Listing 2 command. The ⟨*path_on_your_host*⟩ should be replaced with the full directory path where the project was downloaded.

```
$ docker run -it -p 8080:8080 \
    -e PUBLIC_HOSTNAME=localhost \
    -e WINERY_FEATURE_RADON=true \
    -e WINERY_REPOSITORY_PROVIDER=yaml \
    -v <path_on_your_host>:/var/repository \
    -u `id -u` \
    opentosca/radon-gmt
```

Listing 2. Command for executing a custom TOSCA model repository [8]

After executing the above commands, the user can open Winery GMT on the machine where the container was launched. To do this, he must open any browser and navigate to `http://localhost:8080`. A web interface opens to the user, allowing him to view and try out all the node types, relationship types, blueprints, and other elements.

5.2 Classification project

In this section, the author will explain and show an implementation of *ML-TOSCA* nodes and their relationship in the example of an actual project. It will also explain the basic rules to run the application and cover results gained during the program execution. For clarity, the project will solve the classification problem.

5.2.1 Solution strategy for the Titanic classification problem

The Titanic dataset[21] is a popular machine learning dataset that contains information about the passengers on the Titanic's voyage. The dataset includes passengers' information, demographics, ticket class, and whether or not they survived the disaster. A classification problem with this dataset aims to predict whether a passenger survived or not based on their attributes. It is a fundamental binary classification problem.

Before applying any machine learning algorithms to this problem, it needs to be performed some data preprocessing steps. All applicable phases are shown below.

1. Removing unnecessary columns that do not provide useful information. As a rule, these columns carry unique values repeated only once.

2. Filling in missing values. Multiple strategies can be applied, for example, speaking of numbers filling empty values with the median or average of the entire column.

3. Converting categorical variables into numerical values. Applying this algorithm is a good practice because many machine learning algorithms are designed to work with numerical data.

4. Scaling the features. It ensures that all features contribute equally to the machine learning model's training process [34].

5. Shuffle whole dataset. Sometimes it can lead to the model not being able to generalize well to unseen data if the dataset is not randomly ordered.

Once the data is preprocessed, we can split it into training and testing sets. After that, machine-learning algorithms can be applied to train models on the training set. Ultimately, model performance can be evaluated with test data using different metrics such as accuracy, precision, recall, and F1 score. This flow can be applied to this project to train the models, make predictions, and get results. The author's task was to reproduce these steps using his system.

5.2.2 Implementing the Titanic classification solution using ML-TOSCA

The same project can be solved with *ML-TOSCA* without writing a single line of code. At the same time, there should still be a basic understanding of the algorithms used for working with this data. This section will show the complete solution to this problem which can be found in the project repository service template [22].

[21] https://www.kaggle.com/competitions/titanic/overview

[22] https://github.com/TOSCA-ML/tosca-ml-models/tree/master/servicetemplates/mltosca.blueprints/titanic_classification_problem

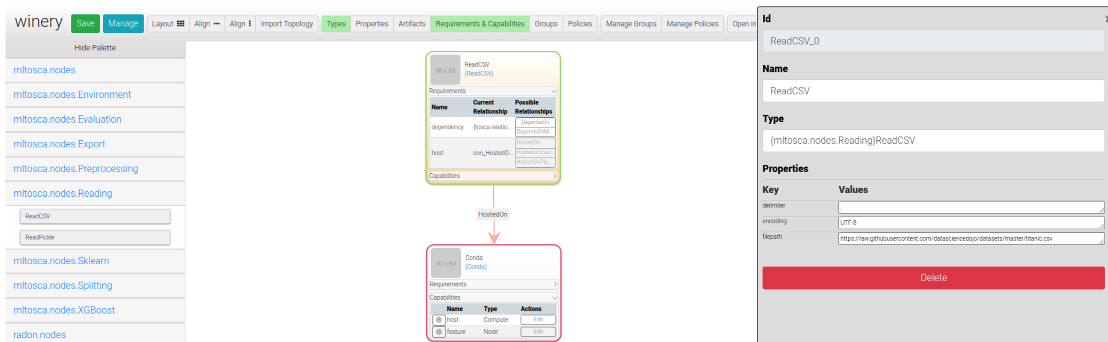

Figure 14. Setting up Conda environment connected with ReadCSV node type.

The first step is to create a new template service and add the first node from the *mltosca.nodes.Evaluation* subgroup called *Conda*. After that, the user can drag the *ReadCsv* node from the *mltosca.nodes.Reading* subgroup and connect them with the mandatory *HostedOn* relationship as shown in Figure 14. It can also be noted that the *ReadCsv* node type has three properties that the user should provide - delimiter, encoding, and file path. In this example, author did not add any virtual machine, meaning this script will be executed on a local machine (however, the user has this opportunity). Users can download this topology and run it already, having such a setup.

In this example, the user has already configured the system to create Conda if it does not exist and reads the CSV file. The next logical step is data processing. As mentioned above, a processing pipeline node has been created to serve as the central processing unit. The user must find and *Pipeline* node type under the *mltosca.nodes.Reading* subgroup and drag it to the platform. This node must be hosted on *Conda*. Since processing cannot logically start before data is read, the pipeline node must have a *DependsOn* relationship with the *ReadCsv* node. The nodes directly responsible for the processing algorithms will be hosted on the pipeline but already using a custom relationship *HostedOnPipeline*. In addition, this correct execution order will be implemented again with the help of *DependsOn* connection. A newly added preprocessing pipeline can be seen in Figure 15.

It may seem incomprehensible why the arrows go from right to left. However, the name of the *DependsOn* dependency speaks for itself. If we have two nodes, A and B, where B should be executed only after A, then B should depend on A. In other words, the arrow should go from B to A. It is also worth clarifying that properties are also provided to some nodes. For example, *RemoveColumns* has an input string of columns separated by a comma we want to remove. *ReplaceNanValues* took a column name *Age* and method *mean*. Finally, two node types *ReplaceCategoricalValues* take a column name as input. In this case, it was *Sex* and *Embarked* columns.

Before training machine learning models, we need to divide the dataset into training and testing. The split node *SplitTrainTest* is under the *mltosca.nodes.Splitting* section. It

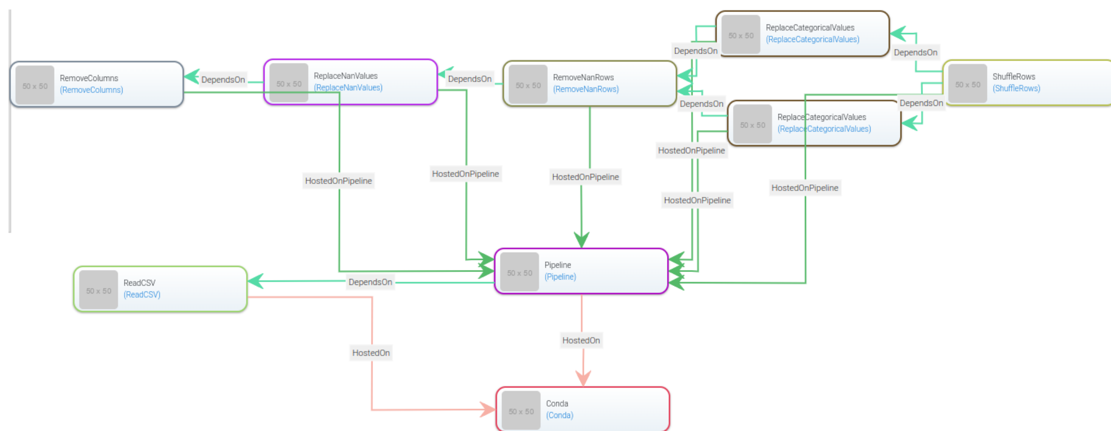

Figure 15. Adding preprocessing pipeline.

has only one property which is *test_size*. By default, this value is 0.2. Given node will also be hosted on *Conda* and will have *DependsOn* relationship with *Pipeline* node type. We could not split the data but directly pass the data to the models. However, thinking ahead, we assume that we want to evaluate models in addition to training.

Finally, the user can train the models when the data is prepared. In this example, the user can use two different libraries - Sklearn and XGBoost. To be more precise, the user has to select classification models, which can be *XGBoostClassifier*, *KNeighboursClassifier*, *DecisionTreeClassifier*, and *RandomForestClassifier*. All models have multiple properties with default values except 1 - the column name to predict. In all other respects, those nodes are similar to others.

The conclusive step is the evaluation of the models. Again, in a similar way as in data preprocessing, a *EvaluationPipeline* is responsible for hosting evaluation metrics on it. As well as others, it should be hosted on *Conda*, but it also should have a custom relationship *DependsOnModel* with the models. In turn, metrics that should be hosted on *EvaluationPipeline* also have a custom relationship *HostedOnEvaluation*. After connecting all of those, the user will get a complete orchestration which will execute all necessary scripts and produce the final result. The whole orchestration is shown in Figure 16.

However, described orchestration can be modified further. Received results on each step - data preparation, model training, and results evaluation can be exported to any preferable location. For this action, the user has to add an *export* block to any node he is interested in. Furthermore, the whole pipeline can be executed on a remote host. There is also a separate node for this, where the user must specify all the necessary properties to connect to the VM and, undoubtedly, connect it to the environment node.

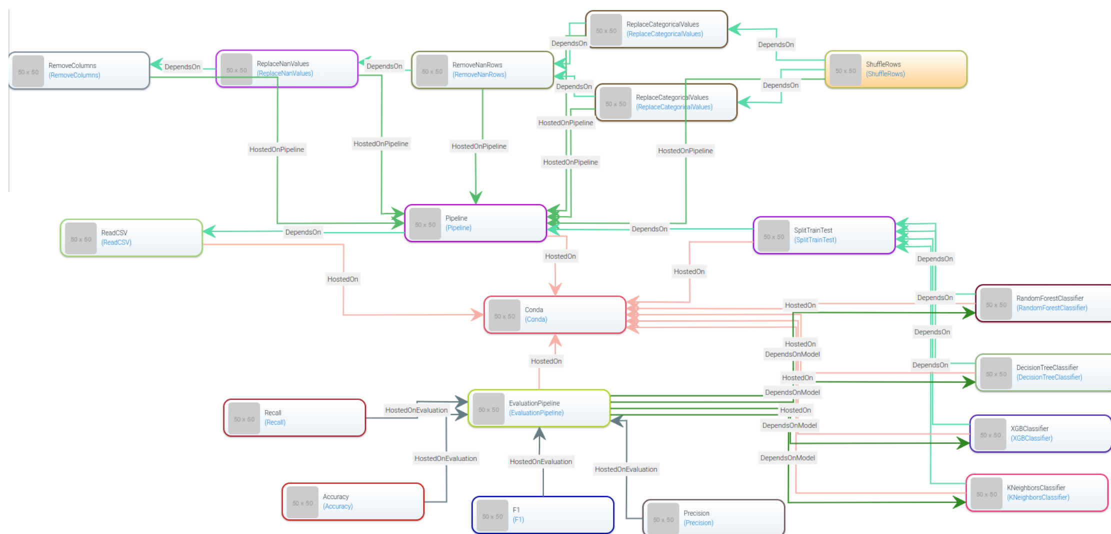

Figure 16. Final ML-TOSCA orchestration for the Titanic classification problem.

5.2.3 Analyzing xOpera's execution flow on a Titanic problem

The last logical step is to run the resulting orchestration. We need to download the CSAR archive and execute it using xOpera. As mentioned above, the ML-TOSCA folder is created after launch. Inside it, a folder with the date and time of the launch will be created with all the execution information. The user does not have to know how and which order scripts were executed; however, it can be tracked in the command line from which the CSAR was executed.

In order to better understand how it works, let us rely on the same orchestration created in the last subsection for simplicity. The result of the execution of described orchestration can be seen in Figure 17. It can be seen that each element was involved. First, as expected, the node responsible for the environment was launched. Then, the node *ReadCSV* reads the data, which will be subsequently processed and transferred to the models for training.

The following preprocessing execution phase consists of multiple stages. Firstly, the *Pipeline* node is deployed but does not execute any script. The following six lines show stages that will be executed inside a pipeline. They write their function names in a file. After that, the *Pipeline* node starts, which will read the previous file with function names and know how many phases should be executed. The pipeline will run in the background and wait until a file exists, showing the function names with the correct execution order. The figure shows that those six execution phases will be executed later, but this time they will write down the execution order. Similarly, an *evaluation pipeline* is executed later.

In other cases, which are left, such as data splitting and model creations, they are

executed straight forward, one by one. The reason is that in other cases, the execution order is not crucial. For example, it does not play any role if XGBClassifier will be executed before RandomForestClassifier and visa versa.

5.3 Testing results

One of the challenges of testing technology is that specific characteristics can be difficult to measure objectively. For example, it is not fair or even possible to test and compare the speed. By speed, we mean the speed of writing the code and the speed of the execution. At the same time, if ML-TOSCA is solving the same task that can be solved using any programming language, it should also be possible to compare the intermediate and final results.

We decided to compare the results generated during the Titanic classification project solution. As each stage of the execution was storing its result, extracting and comparing the outcomes with the results gained with the coding implementation was easy. The data obtained after processing were identical except for the ordering. Regarding the data splitting, it was also possible to check the size and the proportion of the train and test datasets. The sizes of compared datasets were also identical.

Model	**Manual Code**				**ML-TOSCA**			
	Acc	**Prec**	**Rec**	**F1**	**Acc**	**Prec**	**Rec**	**F1**
KNeighboursClassifier	0.68	0.64	0.52	0.6	0.68	0.54	0.56	0.55
RandomForestClassifier	0.8	0.78	0.7	0.74	0.83	0.86	0.61	0.72
DecisionTreeClassifier	0.76	0.7	0.73	0.71	0.82	0.88	0.58	0.7
XGBClassifier	0.82	0.81	0.74	0.77	0.79	0.68	0.73	0.7

Table 3. Comparison of models performance (accuracy, precision, recall, and F1-score) between Manual Code and ML-TOSCA

Machine learning models we were able to test only by comparing their predictions. Undoubtedly, comparing results using the same random state could be better, but the current ML-TOSCA solution does support it. Nonetheless, by looking at the outcomes, it can be seen that the evaluation metrics results are near each other. The results can be seen in Table 3. Thus, based on all the data obtained, it can be argued that ML-TOSCA works and produces the same results as if the developer wrote the code from scratch.

```
[Worker_0]   Deploying Conda_0_0
[Worker_0]     Executing create on Conda_0_0
[Worker_0]   Deployment of Conda_0_0 complete
[Worker_0]   Deploying ReadCSV_0_0
[Worker_0]     Executing start on ReadCSV_0_0
[Worker_0]   Deployment of ReadCSV_0_0 complete
[Worker_0]   Deploying Pipeline_0_0
[Worker_0]     Executing pre_configure_target on ShuffleRows_0_0--Pipeline_0_0
[Worker_0]     Executing pre_configure_target on ReplaceCategoricalValues_1_0--Pipeline_0_0
[Worker_0]     Executing pre_configure_target on RemoveColumns_0_0--Pipeline_0_0
[Worker_0]     Executing pre_configure_target on ReplaceCategoricalValues_0_0--Pipeline_0_0
[Worker_0]     Executing pre_configure_target on RemoveNanRows_0_0--Pipeline_0_0
[Worker_0]     Executing pre_configure_target on ReplaceNanValues_0_0--Pipeline_0_0
[Worker_0]     Executing start on Pipeline_0_0
[Worker_0]   Deployment of Pipeline_0_0 complete
[Worker_0]   Deploying RemoveColumns_0_0
[Worker_0]     Executing start on RemoveColumns_0_0
[Worker_0]   Deployment of RemoveColumns_0_0 complete
[Worker_0]   Deploying SplitTrainTest_0_0
[Worker_0]     Executing start on SplitTrainTest_0_0
[Worker_0]   Deployment of SplitTrainTest_0_0 complete
[Worker_0]   Deploying DecisionTreeClassifier_0_0
[Worker_0]     Executing pre_configure_target on EvaluationPipeline_0_0--DecisionTreeClassifier_0_0
[Worker_0]     Executing start on DecisionTreeClassifier_0_0
[Worker_0]   Deployment of DecisionTreeClassifier_0_0 complete
[Worker_0]   Deploying RandomForestClassifier_0_0
[Worker_0]     Executing pre_configure_target on EvaluationPipeline_0_0--RandomForestClassifier_0_0
[Worker_0]     Executing start on RandomForestClassifier_0_0
[Worker_0]   Deployment of RandomForestClassifier_0_0 complete
[Worker_0]   Deploying KNeighborsClassifier_0_0
[Worker_0]     Executing pre_configure_target on EvaluationPipeline_0_0--KNeighborsClassifier_0_0
[Worker_0]     Executing start on KNeighborsClassifier_0_0
[Worker_0]   Deployment of KNeighborsClassifier_0_0 complete
[Worker_0]   Deploying XGBClassifier_0_0
[Worker_0]     Executing pre_configure_target on EvaluationPipeline_0_0--XGBClassifier_0_0
[Worker_0]     Executing start on XGBClassifier_0_0
[Worker_0]   Deployment of XGBClassifier_0_0 complete
[Worker_0]   Deploying ReplaceNanValues_0_0
[Worker_0]     Executing start on ReplaceNanValues_0_0
[Worker_0]   Deployment of ReplaceNanValues_0_0 complete
[Worker_0]   Deploying RemoveNanRows_0_0
[Worker_0]     Executing start on RemoveNanRows_0_0
[Worker_0]   Deployment of RemoveNanRows_0_0 complete
[Worker_0]   Deploying ReplaceCategoricalValues_1_0
[Worker_0]     Executing start on ReplaceCategoricalValues_1_0
[Worker_0]   Deployment of ReplaceCategoricalValues_1_0 complete
[Worker_0]   Deploying ReplaceCategoricalValues_0_0
[Worker_0]     Executing start on ReplaceCategoricalValues_0_0
[Worker_0]   Deployment of ReplaceCategoricalValues_0_0 complete
[Worker_0]   Deploying ShuffleRows_0_0
[Worker_0]     Executing start on ShuffleRows_0_0
[Worker_0]   Deployment of ShuffleRows_0_0 complete
[Worker_0]   Deploying EvaluationPipeline_0_0
[Worker_0]     Executing pre_configure_target on Precision_0_0--EvaluationPipeline_0_0
[Worker_0]     Executing pre_configure_target on Recall_0_0--EvaluationPipeline_0_0
[Worker_0]     Executing pre_configure_target on Accuracy_0_0--EvaluationPipeline_0_0
[Worker_0]     Executing pre_configure_target on F1_0_0--EvaluationPipeline_0_0
[Worker_0]     Executing configure on EvaluationPipeline_0_0
[Worker_0]   Deployment of EvaluationPipeline_0_0 complete
[Worker_0]   Deploying Precision_0_0
[Worker_0]   Deployment of Precision_0_0 complete
[Worker_0]   Deploying Recall_0_0
[Worker_0]   Deployment of Recall_0_0 complete
[Worker_0]   Deploying Accuracy_0_0
[Worker_0]   Deployment of Accuracy_0_0 complete
[Worker_0]   Deploying F1_0_0
[Worker_0]   Deployment of F1_0_0 complete
```

Figure 17. xOpera execution flow

6 Future research directions

This project implemented a wide range of functions and features to provide a comprehensive solution to the problem at hand. However, due to the complexity and scope of the project, some possible implementations were not included in the final product. This section provides a more detailed description of the missing features that could be implemented.

6.1 Feature Scaling

Feature scaling, also known as feature standardization [17], is a data preprocessing technique that transforms the features of a data set to have a similar scale or range. It involves scaling the features of a data set to a standard scale, such as so that they have a mean of zero and a unit variance, or scaling them to a specific range, usually between 0 and 1 or -1 and 1. It is an essential technique in data science because it helps improve the performance of machine learning algorithms [29, 6, 34].

An important point to note is that when feature scaling is performed on the entire dataset before it is split into training and test datasets, the test set will contain information about the mean and variance of the entire dataset. This, in turn, means that the model may learn relationships between the features and the target variable that do not generalize well to new, unseen data. By scaling the training and test data sets separately, we ensure that the scaling parameters for each set are optimized independently and that the test set remains completely independent of the training process.

For clarity, the entire data set was scaled before being split into train and test. The model does not know how to scale data. This means that when the model receives new data for predictions, it must scale it with different scaling parameters than the training data, leading to potential problems with its performance.

In this project, all data preprocessing is done only once, and only those hosted on the preprocessing pipeline node. Of course, the user can reuse the preprocessing after the data has been split, but currently, the model or the parameters used to scale are not transferred in any way, so this makes no sense. Nevertheless, it is integral to processing and preparing the data for model training and prediction. This implies a separate major work, where storing the algorithm with parameters for further test data processing must be considered.

6.2 Incorporating an event-driven approach

Currently, our solution is using a data-event approach. This approach triggers events when data is changed, and the software system responds accordingly. For example, when data is processed, the model will know that and start training. Nevertheless, this process

will happen only once during the script execution. In the future, an event-driven approach needs to be implemented.

The event-driven technique implies that action will start after a specific event is triggered, for example, a user button press or inserting additional data into the system. This is necessary so that in the future, the model can make predictions and train in parallel. Alternatively, after the data processing pipeline is configured, the user can add data and be sure it will also be processed. That is, it will allow observing a particular continuous integration when the system can be updated without stopping.

6.3 Visualization

Another critical component of data science that should have been implemented in this project is data visualization. Data scientists can use various visualization techniques to uncover hidden insights and trends that inform business decisions. By visually representing data, researchers can quickly identify trends, anomalies, and outliers to understand better the relationships and patterns within the data [37]. Data visualization is also useful for testing hypotheses and validating assumptions. It allows developers and researchers to uncover insights and patterns not readily apparent in the raw data.

Again, such an important component still needs to be implemented, as it was not the primary goal of this work. Nevertheless, the author considers it an essential and valuable component that should be considered in future implementation. It should be a separate, standalone block that should have many different charts and, at the same time, be intelligent enough to understand how and what data should be displayed.

6.4 Providing in-depth control over pipeline behavior

In the *ML-TOSCA* project, the user has a large selection of different properties for configuring various nodes. The choice of one or another property can play an essential role in the quality of the final data or the accuracy of the model. The more the developer knows about them, the better result he can get.

Although the author of this work tried to add as many different properties as possible to various objects, it is impossible to describe everything. It is a big job that requires precision and attention to detail. Sometimes it needs to check if the specified parameters are compatible, for example, to create and train a machine-learning model. Since AutoML does not imply a deep understanding of all attributes, at the moment, having the properties that are provided now, a scientist can train a good model already. However, if a professional wants to use this tool, he may need some parameters slightly.

Thus, a potential future improvement could be revisiting and adding properties to the various nodes. To do this, it will need to read the documentation of the corrected technology and, following the rules, add them to the nodes.

6.5 Cross-Platform Compatibility

This project has an environment block responsible for creating an independent platform inside a system. Regardless, the current implementation only works on Linux, which limits its accessibility to a broader audience. This is because the environment setup script is only for Linux systems.

The solution to the problem consists of several changes consisting of multiple steps. However, those changes are not global, but they should be tested. Since most of the scripts are run through a virtual environment, then in this case, the operating system does not play a role. However, the Anaconda installation script is already different. It means that its location may be different. In addition, the folder structure may be different.

6.6 Workload distribution over multiple virtual machines

It must learn how to distribute work between several computers to get the most out of this implementation. Currently, the project can run all work on a virtual machine. However, distributing the load between several machines will help speed up the whole process and correctly distribute the load. It can be assumed that model training requires GPU, but it does not have to be during the data preprocessing.

The workload can be split and distributed over multiple virtual machines. It divides work into more manageable units, which can be processed in parallel. This means that the whole system's efficiency will increase because each virtual machine can focus on its allocated workload without being overwhelmed by the entire workload. Additionally, workload distribution can help ensure the system is reliable and scalable because additional virtual machines can be added on demand. Furthermore, workload distribution can help improve fault tolerance, enabling the system to continue functioning even if one or more virtual machines fail.

7 Conclusion

This thesis aimed to introduce automated machine learning concepts into the TOSCA standard, focusing on developing independent and reusable components related to the machine learning pipeline. The goal was to create a novel solution that automates time-consuming tasks, such as data preprocessing and model design, enabling non-experts to create and train AI models efficiently. By creating specific components for the ML pipeline, this research has made it possible to integrate them with other TOSCA cloud computing components (e.g., those created in the RADON project) to create a complete machine learning application from data reading to model evaluation. This integration was impossible before creating the independent and reusable components described in this research.

During the research process, author encountered several challenges, such as inventing a way to create versatile components, jobs execution in parallel, and optimizing the machine learning pipeline. It was important not only to create the ability to connect different blocks but also to do it as efficiently as possible in terms of speed and memory used. In addition, as was mentioned, it was also mandatory to construct platform-agnostic components allowing the user to decide where and how to use them. Despite facing challenges, author overcame them by evaluating various technologies, different options and testing the proposed approach before implementing them into the final solution.

The final version was tested on the classification and regression projects and gained results were the same as when they were solved manually by writing a code. As a result, an ML-TOSCA project can solve easy and middle-level tasks. However, even though this project has already implemented most of the various algorithms, much work still lies ahead. Author should consider implementing more cloud approaches to distribute the load without losing speed. Nevertheless, the current solution is enough to create a full-fledged machine learning pipeline.

References

[1] Sajid Alam, Nok Lam Chan, Gabriel Comym, Yetunde Dada, Ivan Danov, Deepyaman Datta, Tynan DeBold, Jannic Holzer, Rashida Kanchwala, Ankita Katiyar, Amanda Koh, Andrew Mackay, Ahdra Merali, Antony Milne, Huong Nguyen, Nero Okwa, Juan Luis Cano Rodríguez, Joel Schwarzmann, Jo Stichbury, and Merel Theisen. Kedro, 2 2023.

[2] Maroua Bahri, Flavia Salutari, Andrian Putina, and Mauro Sozio. AutoML: state of the art with a focus on anomaly detection, challenges, and research directions. *International Journal of Data Science and Analytics*, 14(2):113–126, February 2022.

[3] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, pages 527–549. Springer New York, New York, NY, 2014.

[4] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*. Springer, New York, NY, 08 2014.

[5] Antonio Brogi, Jacopo Soldani, and PengWei Wang. Tosca in a nutshell: Promises and perspectives. In *Service-Oriented and Cloud Computing*, pages 171–186, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[6] Xi Hang Cao, Ivan Stojkovic, and Zoran Obradovic. A robust data scaling algorithm to improve classification accuracies in biomedical data. *BMC Bioinformatics*, 17(1), September 2016.

[7] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Presenza, A. Russo, and et al. Radon: Rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1-2):77–87, 2019.

[8] Eclipse Winery Contributors. Getting started. `https://winery.readthedocs.io/en/latest/user/getting-started.html`. Accessed: 2022-11-15.

[9] James Crist. Dask and numba: Simple libraries for optimizing scientific python code. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2342–2343, 2016.

[10] Stefano Dalla Palma, Martin Garriga, Dario Di Nucci, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. Devops and quality management in serverless computing: The radon approach. In Christian Zirpins, Iraklis Paraskakis, Vasilios

Andrikopoulos, Nane Kratzke, Claus Pahl, Nabil El Ioini, Andreas S. Andreou, George Feuerlicht, Winfried Lamersdorf, Guadalupe Ortiz, Willem-Jan Van den Heuvel, Jacopo Soldani, Massimo Villari, Giuliano Casale, and Pierluigi Plebani, editors, *Advances in Service-Oriented and Cloud Computing*, pages 155–160, Cham, 2021. Springer International Publishing.

[11] Yingnong Dang, Qingwei Lin, and Peng Huang. Aiops: Real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5, 2019.

[12] Chinmaya Dehury, Pelle Jakovits, Satish Narayana Srirama, Vasilis Tountopoulos, and Giorgos Giotis. Data pipeline architecture for serverless platform. In Henry Muccini, Paris Avgeriou, Barbora Buhnova, Javier Camara, Mauro Caporuscio, Mirco Franzago, Anne Koziolek, Patrizia Scandurra, Catia Trubiani, Danny Weyns, and Uwe Zdun, editors, *Software Architecture*, pages 241–246, Cham, 2020. Springer International Publishing.

[13] Chinmaya Kumar Dehury, Pelle Jakovits, Satish Narayana Srirama, Giorgos Giotis, and Gaurav Garg. Toscadata: Modeling data pipeline applications in tosca. *Journal of Systems and Software*, 186:111164, 2022.

[14] I Sessione di Laurea. *Mlops-standardizing the machine learning workflow*. PhD thesis, University of Bologna Bologna, Italy, 2021.

[15] Beniamino Di Martino, Giuseppina Cretella, and Antonio Esposito. Defining cloud services workflow: A comparison between tosca and openstack hot. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 541–546, 2015.

[16] D Di Nucci and DA Tamburri. Radon adoption handbook. 2021.

[17] Thara D.K., PremaSudha B.G, and Fan Xiong. Auto-detection of epileptic seizure events using deep neural network with different feature scaling techniques. *Pattern Recognition Letters*, 128:544–550, 2019.

[18] Pieter Gijsbers, Erin LeDell, Janek Thomas, Sébastien Poirier, Bernd Bischl, and Joaquin Vanschoren. An open source automl benchmark. *CoRR*, abs/1907.00909, 2019.

[19] Abel Gómez, José Merseguer, Elisabetta Di Nitto, and Damian A. Tamburri. Towards a uml profile for data intensive applications. In *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*, QUDOS 2016, page 18–23, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Bas P Harenslak and Julian de Ruiter. *Data Pipelines with Apache Airflow*. Simon and Schuster, 2021.

[21] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[22] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. Modelops: Cloud-based lifecycle management for reliable and trusted ai. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 113–120, 2019.

[23] Shubhra Kanti Karmaker, Md. Mahadi Hassan, Micah J. Smith, Lei Xu, Chengxiang Zhai, and Kalyan Veeramachaneni. Automl to date and beyond: Challenges and opportunities. *ACM Comput. Surv.*, 54(8), oct 2021.

[24] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery – a modeling tool for tosca-based cloud applications. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing*, pages 700–704, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] Shen Li, Paul Gerver, John MacMillan, Daniel Debrunner, William Marshall, and Kun-Lung Wu. Challenges and experiences in building an efficient apache beam runner for ibm streams. *Proc. VLDB Endow.*, 11(12):1742–1754, aug 2018.

[26] Omar Abdulwahabe Mohamad. New virtual environment based on python programming. *Iraqi Journal For Computer Science and Mathematics*, 3(2):111–118, 2022.

[27] Felix Mohr and Marcel Wever. Replacing the ex-def baseline in autoML by naive autoML. In *8th ICML Workshop on Automated Machine Learning (AutoML)*, 2021.

[28] Akram Mustafa and Mostafa Rahimi Azghadi. Automated machine learning for healthcare and clinical notes analysis. *Computers*, 10(2), 2021.

[29] Calpephore Nkikabahizi, Wilson Cheruiyot, and Ann Kibe. Chaining zscore and feature scaling methods to improve neural networks for classification. *Applied Soft Computing*, 123:108908, 2022.

[30] OASIS. Tosca simple profile in yaml version 1.3. `https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html`. Accessed: 2023-23-01.

[31] Hilal Pataci, Yunyao Li, Yannis Katsis, Yada Zhu, and Lucian Popa. Stock price volatility prediction: A case study with AutoML. In *Proceedings of the Fourth Workshop on Financial Technology and Natural Language Processing (FinNLP)*, pages

48–57, Abu Dhabi, United Arab Emirates (Hybrid), December 2022. Association for Computational Linguistics.

[32] Rawaa Qasha, Jacek Cala, and Paul Watson. Towards automated workflow deployment in the cloud using tosca. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1037–1040, 2015.

[33] Rawaa Qasha, Jacek Cala, and Paul Watson. Dynamic deployment of scientific workflows in the cloud using container virtualization. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 269–276, 2016.

[34] V N Ganapathi Raju, K Prasanna Lakshmi, Vinod Mahesh Jain, Archana Kalidindi, and V Padma. Study the influence of normalization/transformation process on the accuracy of supervised classification. In *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*. IEEE, August 2020.

[35] Damian A. Tamburri. Sustainable mlops: Trends and challenges. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 17–23, 2020.

[36] Mark Treveil, Nicolas Omont, Clément Stenac, Kenji Lefevre, Du Phan, Joachim Zentici, Adrien Lavoillotte, Makoto Miyazaki, and Lynn Heidmann. *Introducing MLOps*. O'Reilly Media, 2020.

[37] Anthony Unwin. Why is data visualization important? what is important in data visualization? *2.1*, January 2020.

[38] Quanming Yao, Mengshuo Wang, Hugo Jair Escalante, Isabelle Guyon, Yi-Qi Hu, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *CoRR*, abs/1810.13306, 2018.

[39] Moshe Zadka. Installing python. In *DevOps in Python: Infrastructure as Python*, pages 1–6. Springer, 2022.

[40] Yue Zhou, Yue Yu, and Bo Ding. Towards mlops: A case study of ml pipeline platform. In *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, pages 494–500, 2020.

Appendix

I. Repository

All components, including relationship and node types, are available on the GitHub repository at `https://github.com/TOSCA-ML/tosca-ml-models`.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Artjom Valdas**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   ML-TOSCA: ML pipeline modelling and orchestration using TOSCA,

   supervised by Chinmaya Kumar Dehury and Pelle Jakovits.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Artjom Valdas
09/05/2023