SustainML

Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems

Knowledge binning catalog and efficient practise registry - Initial

Deliverable D1.3

WP1 - Energy Consumption Optimized ML Toolkit and Methods

This project has received funding from the European Union's Horizon Europe research and innovation programme (HORIZON-CL4-2021-HUMAN-01) under grant agreement No 101070408

eProsima
The Middleware Experts

DFKI

Inria

upmem

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

IBM

Project

Title:	SustainML: Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems
Acronym:	SustainML
Coordinator:	eProsima
Grant agreement ID:	101070408
Call:	HORIZON-CL4-2021-HUMAN-01
Program:	Horizon Europe
Start:	01 October 2022
Duration:	36 months
Website:	https://sustainml.eu
E-mail:	sustainml@eprosima.com
Consortium:	**eProsima (EPROS)**, Spain **DFKI**, Germany **Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau (RPTU)**, Germany **University of Copenhagen (KU)**, Denmark **National Institute for Research in Digital Science and Technology (INRIA)**, France **IBM Research GmbH**, Switzerland **UPMEM**, France

Deliverable

Number:	**D1.3**
Title:	**Knowledge binning catalog and efficient practise registry - Initial**
Month:	24
Work Package:	WP1 - Energy Consumption Optimized ML Toolkit and Methods
Work Package leader:	DFKI
Deliverable leader:	DFKI
Deliverable type:	R, DATA
Dissemination level:	PU
Date of submission:	2024-10-15
Version:	v1.1
Status:	Finished

Version history

Version	Date	Responsible	Author/Reviewer	Comments
v1.0	15-10-2324	DFKI	DFKI	
v1.1	15-10-2324	EPROS	EPROS	Review

Executive summary

SustainML project aims to develop a design framework and an associated toolkit, so-called SustainML, that will foster energy efficiency throughout the whole life-cycle of Machine Learning (ML) applications: from the design and exploration phase that includes exploratory iterations of training, testing and optimizing different system versions through the final training of the production systems (which often involves huge amounts of data, computation and epochs) and (where appropriate) continuous online re-training during deployment for the inference process. The framework will optimize the ML solutions based on the application tasks, across levels from hardware to model architecture. It will also collect both previously scattered efficiency-oriented research, as well as novel Green-AI methods. Artificial Intelligence (AI) developers from all experience levels can make use of the framework through its emphasis on human-centric interactive transparent design and functional knowledge cores, instead of the common blackbox and fully automated optimization approaches.

This report corresponds to *Deliverable D1.3 - Knowledge binning catalog and efficient practise registry - Initial* of the SustainML project. This deliverable covers the results of T1.4 and T1.5 from WP1.

Several components of our work have already been published within international academic venues:

1. A Knowledge Distillation framework for Multi-Organ Segmentation of Medaka Fish in Tomographic Image has been accepted for publication on IEEE International Symposium on Biomedical Imaging (ISBI http://2023.biomedicalimaging.org/)
2. Latent Inspector: An Interactive Tool for Probing Neural Network Behaviors Through Arbitrary Latent Activation has been accepted for publication in the International Joint Conference on Artificial Intelligence (IJCAI https://ijcai-23.org)
3. FieldHAR: A Fully Integrated End-to-end RTL Framework for Human Activity Recognition with Neural Networks from Heterogeneous Sensors published on the IEEE International Conference on Application Specific Systems, Architectures and Processors (ASAP https://www.asap2023.org)
4. The Power of Training: How Different Neural Network Setups Influence the Energy Demand has been published on the 37th GI/ITG International Conference on Architecture of Computing Systems (ARCS https://arcs-conference.org/home)
5. TSAK: Two-Stage Semantic-Aware Knowledge Distillation for Efficient Wearable Modality and Model Optimization in Manufacturing Lines has been accepted at International COnference on Pattern Recognition 2024 (ICPR https://icpr2024.org/)

During the submission of this deliverable, the following works are submitted to international academic venues, but still within the reviewing process:

1. Talk2AIoD: Investigating an Adapted Retrieval-Augmented Generation Approach Towards Sustainable Web Agents has been submitted to CHI 2025 (https://chi2025.acm.org/)
2. Latent Boost: Leveraging Latent Space Distance Metrics to Augment Classification Performance has been submitted to the Thirteenth International Conference on Learning Representations (ICLR 2025 https://iclr.cc/Conferences/2025)
3. CoSS: Co-optimizing Sensor and Sampling Rate for Data-Efficient AI in Human Activity Recognition has been submitted to ICASSP 2025 (https://2025.ieeeicassp.org/)

Contents

Acronyms

AI	Artificial Intelligence.
LLM	Large Language Model.
ML	Machine Learning.
RAG	Retrieval Augmented Generation.
RDF	Resource Description Framework.
SOTA	State of the Art.

1 Introduction

This deliverable D1.3 will provide a comprehensive overview of the knowledge base implementation, a substantial element of the SustainML framework. Building upon the foundational research presented in previous tasks, this deliverable will focus on how knowledge is structured, propagated, and utilized across various stages of the machine-learning pipeline to enhance sustainability and efficiency.

This deliverable will focus heavily on T1.4 and T1.5, which play essential roles in shaping the structure and utility of the knowledge base. Task T1.4, "Knowledge Sorting and Abstract Level Binning," focuses on systematically categorizing machine learning tasks into abstract bins, allowing for efficient reuse of knowledge across different problem spaces. Task T1.5, "Gathering Efficient Practices Under the Problem Modeling Space," complements this effort by curating best practices in Green AI and energy-efficient methodologies. Together, T1.4 and T1.5 will enhance the knowledge base's ability to streamline machine learning processes by leveraging abstract-level binning and energy-efficient best practices.

In addition to these technical aspects, deliverable D1.3 will also cover integrating models hosted on external platforms such as Hugging Face. By linking curated models from HuggingFace with corresponding nodes in the knowledge base, the deliverable will demonstrate how this integration facilitates more informed and efficient querying, further enhancing the model development process. Moreover, future directions for expanding the knowledge base will include the incorporation of advanced techniques for human-in-the-loop modifications and model refinement, ensuring continuous improvements towards ML sustainability.

2 Progresses of WP1

2.1 Base Hierarchy of Knowledge Graph

The classic structure of the Resource Description Framework (RDF)-based knowledge graph revolves around the concept of triples, which form the foundation of the graph. Each triple consists of three elements: a subject, a predicate, and an object. In this case, the subject often represents a machine learning model or a related entity, while the predicate defines the relationship between the subject and another node, and the object is the node to which the subject is connected. For example, in a typical triple, a model node (the subject) might be connected to a problem node (the object) through a relationship like *solves* (the predicate). This forms a basic RDF triple: Model — solves — Problem.

In our Knowledge Graph, Model nodes serve as the central points, with other entities branching out through various predicates. We mimic the basic structure of Hugging Face to cover the categorized ML landscape. For instance, a model might be connected to the problem it addresses, a cover tag categorizing it, the library it uses for implementation, or any tags associated with its features or applications. Each model is represented as an entity with unique attributes, such as its name, ID, number of downloads, likes, and the date of its last modification. These attributes are stored as literals, which serve as additional metadata for each model node.

The Model node is directly connected to a Problem node through the hasProblem relationship. The Problem node encapsulates the specific challenge or task that the model is designed to address, such as classification, forecasting, or segmentation. Additionally, a CoverTag node is linked to the Model node through the hasCoverTag predicate, which provides a high-level categorization, such as whether the model pertains to "vision" or "NLP." Similarly, the Model node is connected to a Library node, which represents the machine learning framework or library (such as PyTorch or TensorFlow) used to build the model. This relationship is denoted by the usesLibrary predicate, which is crucial for understanding the computational tools behind each model.

Moreover, each Model node can be connected to one or more Tag nodes through the hasTag predi-

cate. Tags represent specific technologies, techniques, or applications related to the model, providing fine-grained classification. The tags are drawn from a predefined set of allowed tags and can include descriptors like “transformer” or “GAN,” which help users identify models that use specific architectures or methods.

In addition to the direct relationships from the Model node, the Problem node itself can also be connected to a CoverTag node, reinforcing the categorization of the task within a broader application domain. This hierarchical relationship helps to generalize tasks like classification or segmentation under a larger umbrella, such as computer vision or natural language processing.

Overall, the structure of the knowledge graph follows a model-centric hierarchy, where each model node serves as the root and branches out into related entities like problems, cover tags, libraries, and tags. This hierarchy allows for flexible queries, enabling users to retrieve models based on their tasks, domains, or specific features. The graph is dynamic and unrestricted, meaning that new models, tasks, and relationships can be added over time to keep the knowledge base up to date with the latest advancements in machine learning. The presented hierarchy serves a a basis to enable deterministic querying, however, connections can be arbitrary in order to represent complex connections and relationships. Especially for downstream tasks utilizing the Retrieval Augmented Generation (RAG) from Section 2.4, graph based knowledge representations can benefit from unrestricted structures while still perceiving the relevant relationships.

2.2 Hugging Face Integration

As mentioned in the previous sections, we integrate the strucutre and the models of Hugging Face into our knowledge graph. The process begins with a call to Hugging Face’s API to fetch the available model tags and types through the endpoint https://huggingface.co/api/models-tags-by-type. This API response provides a list of pipeline tags which represent high-level tasks or problems like text classification or image generation along with their corresponding subtypes (referred to as coverTags). These tags are essential for categorizing the models based on the problems they aim to solve and help in creating distinct problem nodes in the knowledge graph.

Once the tags are obtained, an SQL database is set up to store all the relevant details about Hugging Face models. This table contains fields such as:

- **model_id**: The unique identifier for the model on Hugging Face.
- **model_name**: The actual name of the model as found on Hugging Face.
- **problem**: The task or problem category derived from the tags, such as “classification” or “translation.”
- **tags**: A JSON-formatted list of specific tags that further describe the model’s capabilities and characteristics.
- **coverTag**: A high-level category that can group models into domains such as vision or NLP.
- **library**: The underlying library or framework (e.g., TensorFlow, PyTorch) used to build the model.
- **downloads**: The number of times the model has been downloaded, serving as an indicator of its popularity.
- **likes**: The number of likes the model has received from users.
- **lastModified**: The timestamp of the last modification to the model’s repository.

The API then fetches the list of all available models, with each model linked to one of the pipeline tags obtained earlier. Each model’s metadata, such as its ID, name, tags, and library, is inserted into the SQL

database. Special attention is given to error handling for potential issues like duplicate entries (where the same model ID is encountered more than once) or missing fields to prevent any compromising.

After fetching the models and storing their details in the database, the next step involves extracting and filtering the tags associated with these models to identify the most relevant ones. This is done by querying the tags field from the Models table in the SQL database. The code parses each tag string, ensuring it is in a valid list format, and counts the occurrences of each tag..

Once the tags have been counted, they are ranked by frequency, allowing for the identification of the most commonly used tags across models. A filtering step ensures that only tags appearing with a certain frequency (in this case, a minimum of 50 occurrences) and having more than two characters are considered relevant in order to prevent littering and excessive size of the knowledge graph

After the model data has been stored in the SQL database, the second phase of the integration process involves transferring this data into the RDF-based knowledge graph. Each model entry in the SQL database is mapped to a node in the graph, with relationships being established between the models and their associated problems, tags, and libraries. The relationships are defined as RDF triples, with the model as the subject, the relationship (such as hasProblem, hasTag, or usesLibrary) as the predicate, and the corresponding entity (problem node, tag node, or library node) as the object.

This structured approach ensures that models are categorized not only by their problem domains but also by other metadata such as the libraries they use and the tags that describe them. This semantic structuring allows for more efficient querying and retrieval of models within the knowledge graph, which can now be accessed based on a wide range of attributes such as task type, associated tags, and usage patterns.

2.2.1 Retrieving and Deploying Models

For the retrieval and deployment of models from Hugging Face, we commonly rely on the widely used `transformers` library as the majority of the latest models is integrated in such. This will allow seamless access to a large collection of pre-trained models to utilize them for fine-tuning or inference. The process involves querying the Model Hub, fetching a specific model by its unique ID, and loading it for application.

Below is an example in Python using the `transformers` library, showcasing how a pre-trained model can be retrieved and deployed for a sequence classification task:

```
from transformers import AutoFeatureExtractor, AutoModelForImageClassification
from PIL import Image
import torch

# Load pre-trained image classification model and feature extractor
model_id = "google/vit-base-patch16-224"
feature_extractor = AutoFeatureExtractor.from_pretrained(model_id)
model = AutoModelForImageClassification.from_pretrained(model_id)

# Open an image
image = Image.open("path_to_your_image.jpg")

# Preprocess the image
inputs = feature_extractor(images=image, return_tensors="pt")

# Perform inference
outputs = model(**inputs)
```

```
logits = outputs.logits

# Get predicted class
predicted_class_idx = torch.argmax(logits, dim=-1).item()
print(f"Predicted class index: {predicted_class_idx}")
```

In this example, a pre-trained Vision Transformer (ViT) is loaded from Hugging Face's Model Hub. The model uses an `AutoFeatureExtractor` to process images, converting them into the appropriate format for the model. The model then performs image classification, outputting logits from which the predicted class can be extracted.

Furthermore, to facilitate deployment on edge devices or other resource-constrained environments, we export the retrieved models to the ONNX (Open Neural Network Exchange) format.

```
import torch
# Exporting the model to ONNX format
torch.onnx.export(model, dummy_input, "vit_model.onnx", opset_version=11)
```

By converting models to ONNX format, we ensure compatibility with a wider range of platforms, particularly enabling deployment on edge hardware where low-power and efficient model inference is crucial. This aligns with the goals of the SustainML framework, promoting sustainability in AI by optimizing models for diverse deployment environments.

In the long term, we aim to extend this integration by supporting additional frameworks and custom additions of models for inference, ensuring flexibility in how models are retrieved, optimized, and deployed within the SustainML framework.

2.3 Traditional Querying

As a basis, we leverage the `rdflib` library to enable efficient extraction of information from RDF graphs programmatically. The process begins by loading graph data, typically stored in Turtle format (`.ttl`), which organizes various models, problems, and their interrelations. After loading the graph, users can retrieve cover tags to categorize models by their domains. This allows for quick identification of relevant models for specific tasks. Additionally, users can compile a list of defined problems, representing distinct categories like image classification or text generation. The querying functionality extends to exploring problems linked to specific cover tags, enabling users to find tasks associated with domains of interest. Furthermore, users can access models designed for specific problems, often sorted by download counts to highlight popular choices. Detailed metadata about individual models, including their ID, library, and download statistics, can also be retrieved.

By combining these capabilities, a user-friendly interface can be developed as part of the collaboration with WP4, facilitating easy navigation through models, problems, and cover tags. This framework forms a strong foundation for further integration with other system components.

Current Functions for Querying:

- `load_graph(file_path)`: Loads the graph from a specified .ttl file.
- `get_cover_tags(graph)`: Retrieves all cover tags defined in the graph.
- `get_problems(graph)`: Extracts a list of distinct problems represented in the graph.
- `get_problems_for_cover_tag(graph, cover_tag_literal_text)`: Finds all problems associated with a given cover tag.

- `get_models_for_problem(graph, problem_literal_text)`: Retrieves models linked to a specified problem, sorted by download counts.
- `get_model_details(graph, model_name)`: Accesses detailed information about a specific model.

2.4 Knowledge through Graph RAG

The Graph RAG approach represents an Large Language Model (LLM)-driven system designed to simplify interactions with RDF graph databases by automatically generating SPARQL queries. We aim to shift the querying from a deterministic to a more conversational interaction, especially allowing the users to address their emphasis on the model search and selection. Built on Langchain and OpenAI models, it translates natural language questions into SPARQL queries, making querying more intuitive and reducing the need for users to manually write complex queries. The system works by adhering to the schema of the RDF graph, ensuring valid queries and preventing errors.

SPARQL is the State of the Art (SOTA) query language for RDF graphs and retrieves structured data from graph-based datasets. Instead of using the `rdflib` for directly querying through SPARQL, Graph RAG automates this process, offering a more accessible way to explore interconnected data. Currently, this project is a work in progress, aiming to streamline the data retrieval process by merging AI capabilities with RDF querying, enhancing usability and efficiency. However, considerations regarding the sustainability need to be evaluated to estimate if the benefit of using Graph RAG within the SustainML framework allows for enhancing the systems energy demand compared to the traditional querying.

2.5 Code and Graph Accessibility

All resources related to the WP1 implementation are available at GitHub - DFKIEI/Knowledge2Model (https://github.com/DFKIEI/Knowledge2Model). This repository is continuously updated to track our progress and improvements throughout the project. The repository contains the complete Python code for all functionalities discussed in our documentation, providing a solid foundation for generating the graph and quering, including the Graph RAG approach and the traditional querying. Further, we provide a simple editor tool to visualize and manipulate the graph nodes and connections. We envision this tool as a form of manual validation and inspection of graphs.

In addition to the code, the knowledge graphs themself are available in Turtle (TTL) file format within the Graphs folder. We maintain multiple versions of these graphs, each differing in their level of comprehensiveness. The `graph.ttl` file consistently represents the latest and most extensive graph, ensuring users have access to the most current and relevant data for their queries.

3 Future works in the following period

In the upcoming period, we aim to integrate our implementation of WP1 more effectively into the SustainML ecosystem, enhancing its overall functionality and collaboration with the other work packages. Additionally, we plan to refine our RAG approach, with a particular emphasis on sustainability evaluations and improvements to asses the potential integration.

We also intend to conduct a comparative analysis between pure retrieval methods and pure generation techniques to assess their respective efficiencies and effectiveness. Our focus will shift towards task 1.5 in the next iteration of the project, gathering efficient practices from systematic literature research to integrate them into the graph for enhanced sustainability knowledge.

Furthermore, while we currently only utilize models from Hugging Face, we will expand our approach by incorporating custom models manually, allowing for greater flexibility and adaptability. Lastly, we aim

to integrate energy consumption information provided by WP3 into the knowledge graph to better assess and prioritize the environmental implications within the model querying process.