RAKI

Collaborative Project

Rapid Explainable AI for Industrial Plants

Project Number: 32100687 **Start Date of Project:** 2019/09/01 **Duration:** 36 months

Deliverable 1.3
Requirement Catalog and Architecture of the RAKI (Prototyp and Framework, Version 2)

Dissemination Level	Public
Due Date of Deliverable	M6
Actual Submission Date	M6
Work Package	AP1 – Requirement Elicitation
Task	AS1.4
Type	document
Approval Status	final
Version	2.0
Number of Pages	13

Abstract:

Gefördert durch:

Bundesministerium für Wirtschaft und Energie

aufgrund eines Beschlusses des Deutschen Bundestages

This project has received funding from the Federal Minister for Economic Affairs and Energy (BMWi) under grant agreement No 32100687.

History

Version	Date	Reason	Revised by
2.0	28/01/2022	Revision of first draft	Alexander Bigerl
1.1	28/01/2022	First draft for 2.0	Caglar Demir
1.0	13/03/2020	Revision of first draft	Axel Ngonga
0.4	12/03/2020	First draft of deliverable	René Speck
0.3	11/02/2020	Prioritization of features	Axel Ngonga
0.2	31/12/2019	First draft of architecture	Michael Röder, Alexander Bigerl, Timofey Ermilov
0.1	01/12/2019	Initial requirements	Axel Ngonga

Authors In alphabetical order.

Organization	Name	Contact Information
UPB	Alexander Bigerl	alexander.bigerl@uni-paderborn.de
UPB	Axel Ngonga	axel.ngonga@uni-paderborn.de
UPB	Caglar Demir	caglar.demir@uni-paderborn.de
UPB	Diego Moussallem	diego.moussallem@uni-paderborn.de
UPB	Michael Röder	michael.roeder@uni-paderborn.de
ULEI	René Speck	speck@informatik.uni-leipzig.de
ULEI	Timofey Ermilov	ermilov@informatik.uni-leipzig.de

Contents

1 Introduction

This deliverable presents considerations pertaining to the architecture of the machine learning (ML) platform for scalable machine learning on structured data to be developed within the RAKI project. The platform is designed to be generic, ergo, to accommodate ML algorithms on knowledge graphs for both regression and classification . Moreover, it is designed to be compatible with other ML platforms. Hence, it reuses standards (e.g., SPARQL, RDF, OWL) as much as possible. The platform is also designed to scale. Hence, a packaging concept and a distribution concept are part of the specification, even if the project does not plan for them to be be part of the final version of the prototype.

2 State of the art

An extensive state of the art research reveals that the current de-facto standard for Inductive Logic Programming (ILP) on RDF knowledge graphs have similar data flows. We hence carried out an extensive analysis of the performance of a representative framework, DL-Learner, w.r.t. runtime, memory usage. We chose this framework as it currently achieves the best performance on standard ILP benchmarks.

2.1 Program Flow

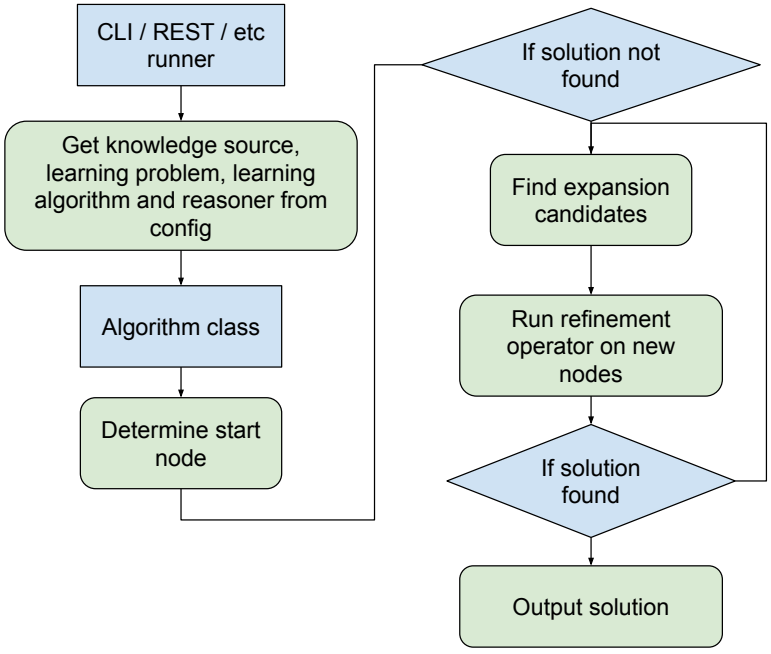

Figure 1: Overview of a typical program flow in the DL-Learner

As shown in Figure 1, the main entrypoint for the framework is the Command-Line Interface (CLI) class. This class takes on the role of reading the configuration file and preparing the specified knowledge source, learning algorithm, learning problem and optionally reasoner classes. The CLI runner relies on abstract classes for knowledge source, learning problem, learning algorithm and reasoner classes to enable extensibility. Once all the instances of these classes are constructed, the CLI passes the learning

problem and knowledge source to the learning algorithm instance and calls the '`.start()`' method. Alternatively, if the execution is requested to be done with cross-validation, the CLI runner might instantiate a special cross-validation class that runs in a similar fashion to the default execution but adds cross-validation results to the output of the ML algorithm. All the further work is executed in the learning algorithm instance. All execution is done synchronously in one thread. All data is stored in-memory in a sequence of abstracted classes. The storage itself relies either on the OWLApi or Jena packages for internal data representation.

2.2 Current Implementation Flaws

- A synchronous execution of tasks can lead to a severe under-use of the CPU. Preliminary experiments suggest that the CPU utilization can drop to under 10% even on small datasets. Java is especially problematic as roughly 90% of CPU use can be due to the garbage collector. A clear solution is to ensure that tasks can be parallelized or vectorized.
- Storing all the data (knowledge source, results of refinement operator, results for cross validation, etc.) like in many of the current frameworks can lead to a very large memory signature and is not practical for very large datasets unless an adequate and compressed storage solution is used. Here, **remote storage** and a **better in-memory storage** must be explored. Preliminary works based on an OWL to SPARQL bridge can be found in [3].

2.3 Natural Language Generation (NLG)

NLG is the process of automatically generating coherent Natural Language (NL) text from non-linguistic data [11]. Recently, the field has seen an increased interest in the development of NLG systems focusing on verbalizing resources from Semantic Web (SW) data [5]. The SW aims to make information available on the Web easier to process for machines and easier to understand for humans. However, the languages underlying this vision, i.e., Resource Description Framework (RDF), SPARQL Protocol and RDF Query Language (SPARQL) and Web Ontology Language (OWL), are rather difficult to understand for non-expert users. For example, while the meaning of the OWL class expression `Class: Professor SubClassOf: worksAt SOME University` is obvious to every SW expert, this expression ("Every professor works at a university") is rather difficult to fathom for lay persons.

Despite the plethora of recent works written on handling RDF data, only a few have exploited the generation of NL from OWL and SPARQL. For instance, [1] generates sentences in English and Greek from OWL ontologies. Also, SPARQL2NL [7] uses rules to verbalize atomic constructs and combine their verbalization into sentences.

With this aim, we developed an open-source holistic NLG framework for the SW, named LD2NL, which facilitates the verbalization of the three key languages of the SW, i.e., RDF, OWL, and SPARQL into NL. Our framework is based on a bottom-up paradigm for verbalizing SW data. Additionally, LD2NL builds upon *SPARQL2NL* as it is open-source and the paradigm it follows can be reused and ported to RDF and OWL. Thus, LD2NL is capable of generating either a single sentence or a summary of a given resource, rule, or query. LD2NL generates texts which can be easily understood by humans and is now considered the state-of-the-art approach to verbalize SW languages.[1]

[1] `https://github.com/AKSW/LD2NL`

3 General Requirements

The general requirements for this architecture are as follows:

1. **Feasibility**: The framework was to be designed so as to be implementable within the numbers of person years foreseen in the project. Consequently, local tests and evaluation were to be preferred over large-scale tests and deployments. Correctness is to be preferred over scalability. Still the framework is to designed so as to be easily ported to large-scale infrastructures.
2. **Releases**: The framework should follow the open-source principles and provide a first stable release with at least one fully implemented ML algorithm before M21. Other releases are to follow the project schedule.
3. **Deployment**: The framework should be easily deployable locally. Hence, an adequate packaging is necessary. Docker is to be preferred.
4. **Scalability**: The framework should be designed with very large datasets in mind (i.e., 1B+ triples). Hence, ports to large-scale infrastructures must be possible.
5. **Language support**: The framework must allow the extension towards supporting all possible description logics and ILP algorithms. Hence, the algorithm interfaces must be as generic as possible.
6. **Oracles (optional)**: A valuable feature would be to allow for third-party asynchronous oracles (e.g. user that gives feedback on results).
7. **Multi-user (optional)**: A valuable feature would be to allow for multi-tenancy and parallel ML jobs execution.

4 Architecture Requirements

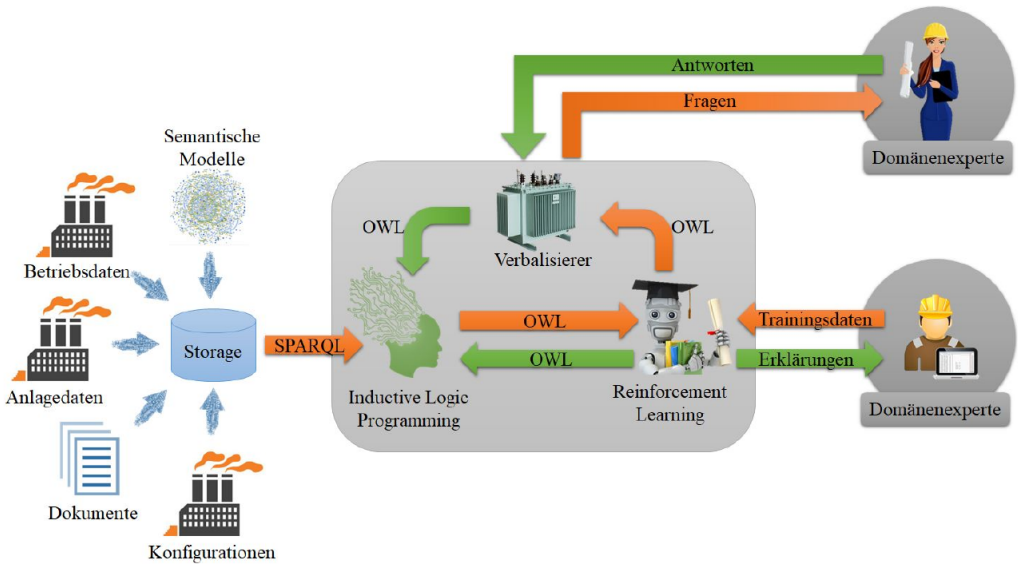

Figure 2: Overview of RAKI

The three required main components of the architecture are:

1. **Inductive Logic Programming Component**: The component defines the concept learning problem by using a knowledge representation formalism, refinement operator, and a quality metric. Initially, we choose the attributive language with complement $\mathcal{ALC}$ as a knowledge representation formalism. The input of the ILP component comprise a set of positive (and potentially negative) examples and a reference to a potentially remote knowledge graph including a corresponding ontology and a knowledge representation formalism. Initially, we choose the attributive language with complement $\mathcal{ALC}$ as a default. After receiving the input, the ILP component constructs the quasi ordered search space consisting of $\mathcal{ALC}$ concepts. The ILP component assigns a score to each $\mathcal{ALC}$ concept that is defined by a quality metric. The quality of a concept can be determined via Accuracy, Precision, Recall and F1-score. The ILP component hence constructs an environment for the Reinforcement Learning Component that traverses such environment by deciding which concept to be expanded via the refinement operator.
2. **Reinforcement Learning Component**: The RL component works in tandem with the ILP algorithm and serves to address the myopia of existing implementations. the RL component provides trainable and pre-trained RL approaches. More specifically, a RL approach can be trained at will on the given dataset. In critical cases (e.g., high variance in the Q function), it can request supplementary data to augment the training data available. Moreover, a pre-trained RL approach can be directly applied on the given dataset without retraining. The given dataset consists of a set of examples and background knowledge provided in the ILP component.
3. **Verbalization Component**: Critical decisions detected by the ML components will be verbalized and handed out to domain experts. Those experts support the Machine Learning component by providing supplementary data, answering questions pertaining to learned models and hence supporting the automatic decision making in critical states of the machine learning process.

The final architecture for such a framework can be based on a manager-workers pattern: a manager node with a group of worker nodes to run in parallel (Figure 3).

4.1 Input and Output

Figure 4 shows the input and output of the different components and their connection to each other in more detail.

1. **Reinforcement Learning**: to learn a Q-function approximator, SAMUEL will expect the A-Box and the T-Box of the data to learn upon. Practically, this means:
   (a) A reference to the ontology of the data (i.e., the T-Box), e.g., as an OWL file,
   (b) A reference to an object that can be queried via SPARQL or
   (c) A reference to a retrieval function able to return all instances of a concept in the language $\mathcal{L}$.

   The module will return a model in the form of a Q-function approximator, which can be used by the ILP module described in the following.
2. **Inductive Logic Programming**: This module will learn by using a RL model to estimate the quality of refinements. Hence, it will need
   (a) A reference to an OWL file describing the ontology of the data (T-Box),
   (b) A reference to an object that can be queried via SPARQL or

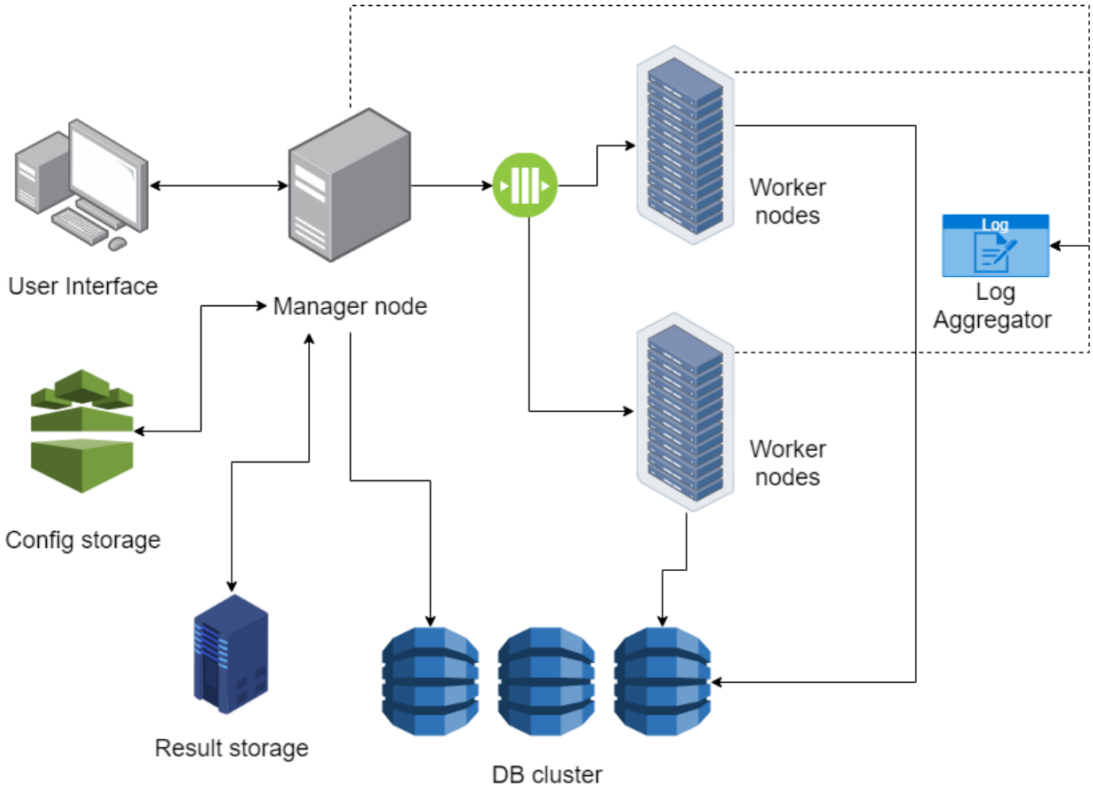

Figure 3: High-level architecture

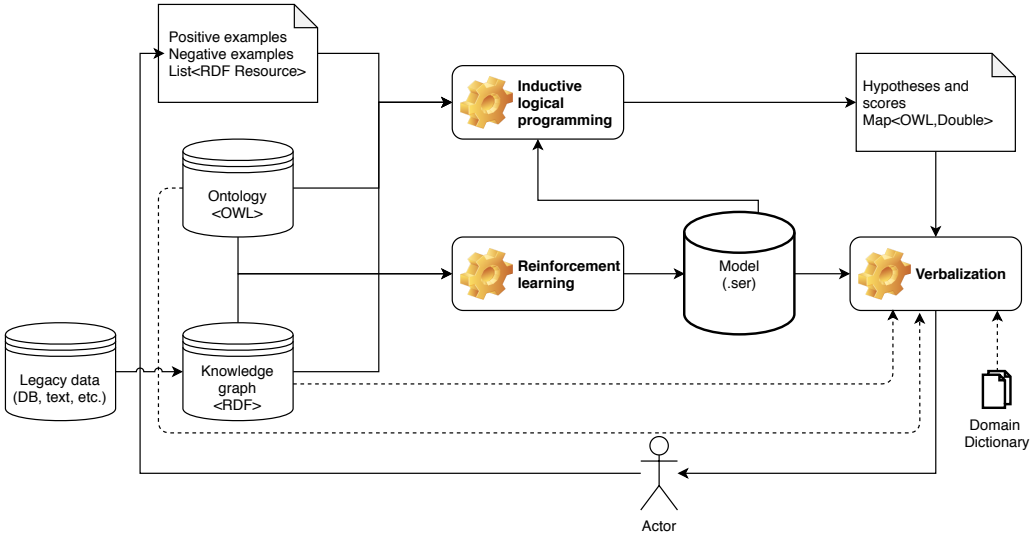

Figure 4: Overview of IO behavior of the main components of the RAKI platform

(c) A reference to a retrieval function able to return all instances of a concept in the language $\mathcal{ALC}$

(d) Positive examples and optionally

(e) Negative examples as well as

(f) A reference to a Q-function estimator, i.e., to a trained RL model.

The module will return a set of concepts mapped to scores. Note that the module is to be built so as to support receiving an existing refinement tree as input and building thereupon. Therewith, the computation can be stopped if critical decisions are to be made.

3. **Verbalization**: This module verbalizes the set of concepts, along with their scores, produced by the ILP module in a given language $\mathcal{L}$. Hence, the required input to generate the verbalizations will be

   (a) A reference to a domain dictionary

   (b) A reference to a knowledge graph which can be queried for labels using a standard language, e.g., SPARQL

   (c) A reference to a concept summarization algorithm for the language $\mathcal{L}$, which can take a concept in $\mathcal{L}$ and return a shorter concept which minimize the classification error.

   The module returns two types of outputs, (1) an automatic generated explanation pertaining to the decisions and actions taken by each of the previous modules and (2) an automatic generated question along with possible answers in a multiple-choice format to the domain expert for a human-in-the-loop scenario. Additionally, a sanity-check option will be provided, where the domain expert can register a non-sensical question-answer construct.

4.2 Deployment

For the deployment of the platform, the following requirements have been determined.

1. Platform components should be built into packaged images, e.g., Docker.
2. The platform should be deployable locally, e.g., with docker-compose.
3. The platform should allow a to be used for production-ready deployment on server clusters or in a cloud, e.g., by using Kubernetes[2] with Helm[3].

4.3 Manager Node

Manager node is responsible for:

1. Providing REST API to the interfaces
2. Managing configurations provided by users
3. Controlling the execution flow of the configurations
   (a) Splitting work across workers
   (b) Interrupting work to ask Oracle for feedback when applicable
4. Managing result sets

Manager node should be able to handle requests in async manner to allow parallel execution.The algorithms should be built so as to support an a-posteriori deployment at large scale.

[2] https://kubernetes.io/
[3] https://helm.sh/

4.4 Storage Solution

Given the size of the data, in-memory solutions might not be the most viable approach in our framework.The retrieval functions used by the algorithms must hence also support remote data sources. A family of retrieval functions must hence provide SPARQL read-only access for workers and manager. The SPARQL store itself can be either a local or a remote storage. Possible choices for an alpha version include Virtuoso, BlazeGraph and especially Tentris based on the benchmarking results presented in Figure 5.

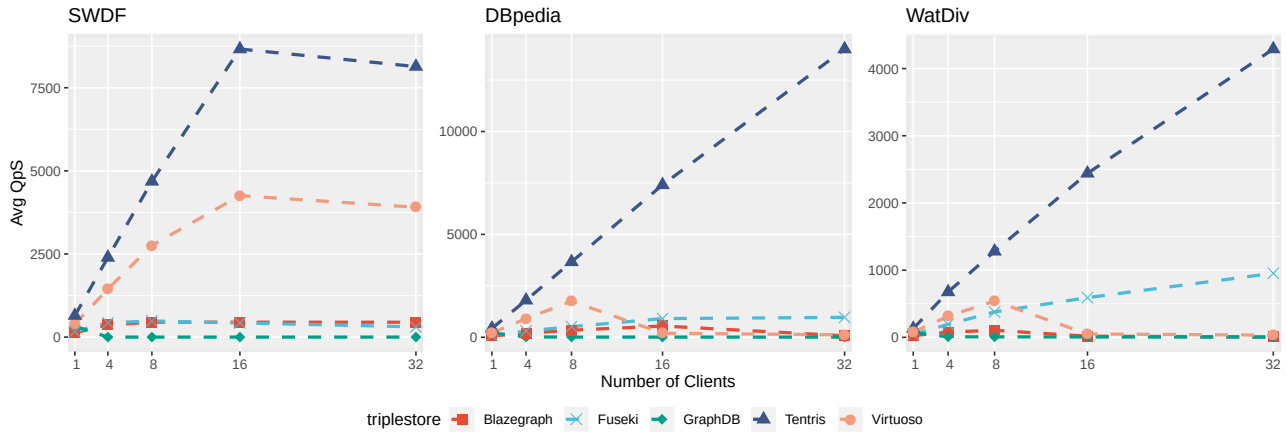

Figure 5: Comparison of storage solutions against FEASIBLE benchmarks. The performance is measured in queries per second (QpS, higher values are better).

4.5 Configuration Storage

If required, configurations could be stored either as linked data or as key-value pairs. The configuration should be provided as a file (RDF file, JSON file, YAML file, or similar) or as a database endpoint (SPARQL triple store, JSON document store, key-value store, or similar). This storage would have to provide read-write access to the manager node. The configuration could include:

1. Configuration name
2. Knowledge data (SPARQL endpoint, retrieval function)
3. Positive examples (a link to a file)
4. Negative examples (a link to a file)
5. Output (a link to a file)
6. Algorithm specification
   (a) Algorithm name
   (b) Parameters
   (c) Algorithm version
   (d) Oracle (if needed / present)
   (e) Execution status

The manager node may update the configuration status to reflect execution states.

4.6 Result Storage

Results are tied to a configuration ID. The results may be stored in a file or a separate database. Depending on the result properties, a serialization as text, table, RDF, JSON, YAML or similar file formats may be chosen. For some outputs, a combination of different files may be required. A separate database (triple store, relational database, document store, depending on the datatype) may also be used as result storage. Therefore, the manager node is granted read-write access. For local deployments the same database as for configuration can be utilized.

4.7 Oracle Workflow

The suggested workflow for including an oracle implementation looks like follows:

1. Intermediate results of the job are stored in the result storage.
2. The configuration of the job is updated by setting its status to `"needs feedback"`.
3. The platform asks the user for feedback. If the user is connected to the user interface, the user can be asked directly. If the user is not available, the manager node should send a message, e.g., via mail, to inform the user that his job needs feedback.

This workflow can be adapted for programmatic oracles by connecting the oracle to the platform using

1. A persistent connection (e.g. via websockets) or
2. using webhooks.

4.8 Worker Nodes

The worker nodes carry the main work load and at least one worker has to be available for the system to work. They receive directions from the manager and have read-only access to the knowledge base (Section 4.4). The workers should be stateless to allow for horizontal scaling. To this end, the directions should either include the user configuration along with work instructions or the workers need a read-only access to the configuration store. The results are reported back to the manager node. Based on the received results, the manager decides how to proceed further on.

Workers should be resilient to algorithm errors. If a task fails, the worker should gracefully handle the exception and notify the manager that the task has failed. The manager can react and either re-schedule the task or abort the complete job by forwarding the error to the user.

4.9 User Interfaces

User interfaces are used to (1) create a new job with a configuration and (2) provide feedback to algorithms as an oracle (if supported).

Possible solutions for user interfaces are

1. Web UI (based on a REST API),
2. Command-Line Interface (CLI),

3. REST API or
4. programmable API.

4.10 Logging and Monitoring

Thee Monitoring should notify users when one of the components of the platform has an irrecoverable failure. In addition, it should display the overall platform status in the UI. To achieve this, existing logging systems, e.g., Grafana[4] or ELK[5], and established monitoring solutions, e.g., Prometheus[6] or cAdvisor[7], should be used.

5 Framework Key Performance Indicators

The following KPIs will be used for our experiments:

Precision, Recall, Accuracy and F1-Score We apply Precision $P = \frac{TP}{TP+FP}$, Recall $R = \frac{TP}{TP+FN}$, Accuracy $A = \frac{TP+TN}{TP+FP+FN+TN}$ and F1-Score $F1 = 2 \cdot \frac{P \cdot R}{P+R}$. TP, FP, FN, and TN are determined as defined in the DL-learner framework [4].

Runtime The runtime of algorithms or queries is a central metric for evaluating the performance of a system. It measures the time that passes between starting and finishing a task.

Queries per Seconds (QpS) is a metric that measures at what speed a system can answer queries that are sent to it. Typically, average QpS are measured in a stresstest scenario where a mix of queries is sent for a certain period of time.

Query Mixes per Hour (QMpH) is a metric that measures for a fixed query mix how often a system can answer all those queries within one hour. Compared to QpS, a single long-running query has a stronger influence on the QMpH value.

5.1 Verbalization KPI

An evaluation of a given Natural Language Generation (NLG) system may be carried out either automatically or manually. Generally, the NLG community has opted to use automatic metrics to decrease human efforts and time. A common process of automatic evaluation is composed of (1) the source data (input), in our case Class Expressions, (2) generated text (output produced by an NLG system, which is also called a hypothesis), and (3) the reference text of the source data, commonly named as human reference. The human reference is compared automatically against the generated text using a given evaluation metric.

There are plenty of automatic evaluation metrics that are used in the Machine Translation and Natural Language Generation tasks. However, we plan to use in RAKI the most effective metrics according to previous NLG work, which enables a fair scientific comparison between the quality of different NLG systems.

[4] https://grafana.com/
[5] https://www.elastic.co/what-is/elk-stack
[6] https://prometheus.io/
[7] https://github.com/google/cadvisor

5.1.1 Automatic metrics

BLEU was created by Papineni [8] widely chosen for evaluating automatically generated text outputs due to its low costs. BLEU uses a modified precision metric for comparing the output with the human reference. The precision is calculated by measuring the n-gram similarity (size 1-4) at word levels. BLEU also applies a brevity penalty by comparing the length of the output with the human reference. Additionally, some BLEU variations have been proposed to improve its evaluation quality. The most common variation deals with the number variability (frequency) of useless words commonly generated by NLG systems. However, the main weakness of BLEU is its difficulty handling semantic variations (i.e., synonyms) while performing the n-gram similarity.

METEOR was introduced by Banerjee and Lavie [2] to overcome some weaknesses of BLEU, for example, the lack of explicit word-matching between translation and reference, the lack of recall and the use of geometric averaging of n-grams. The goal of METEOR is to use semantic features to improve correlation with human judgments of translation quality. To this end, METEOR considers the synonymy overlap through a shared WordNet synset of the words.

chrF proposed by Popovic [9, 10] was initially for the use of character n-gram precision and recall (F-score). In addition, chrF also works with word n-grams. Although n-gram is already used in well-known and complex automatic metrics, the investigation of n-grams as an individual metric has not been exploited before. chrF has shown a good correlation with human rankings of different automatic outputs. chrF is simple and does not require any additional information. Additionally, chrF is language and tokenization independent.

TER is different from the metrics mentioned above. TER measures the number of necessary edits in an output (generated text) to match the human reference exactly. The goal of TER is to measure how much effort is needed to fix an automated translation to make it fluent and correct[12]. The edits consist of insertions, deletions, substitutions, and shifts of words, as well as capitalization and punctuation. The TER score is calculated by computing the number of edits divided by the average referenced words.

ROUGE is essentially a set of metrics for evaluating automatic summarization of texts as well as MT and NLG[6]. It works by comparing an automatically produced summary or translation against a set of human reference summaries. ROUGE has some variations such as ROUGE-N, ROUGE-S, and ROUGE-L. ROUGE-N measures unigram, bigram, trigram, and higher-order n-gram overlap while ROUGE-L takes into account the longest matching sequence of words relying on Longest Common Subsequence (LCS). An advantage of using LCS is that it does not require consecutive matches but in-sequence matches that reflect sentence-level word order. ROUGE-S refers to any pair of words in a sentence considering a given order, usually called as skip-gram cooccurrence.

References

[1] Ion Androutsopoulos, Gerasimos Lampouras, and Dimitrios Galanis. Generating natural language descriptions from OWL ontologies: The natural owl system. *J. Artif. Int. Res.*, 48(1):671–715, October 2013.

[2] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[3] Simon Bin, Lorenz Bühmann, Jens Lehmann, and Axel-Cyrille Ngonga Ngomo. Towards sparql-based induction for large-scale rdf data sets. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 1551–1552. IOS Press, 2016.

[4] Lorenz Bühmann, Jens Lehmann, and Patrick Westphal. Dl-learner—a framework for inductive learning on the semantic web. *Journal of Web Semantics*, 39:15–24, 2016.

[5] Claire Gardent, Anastasia Shimorina, Shashi Narayan, and Laura Perez-Beltrachini. Creating training corpora for nlg micro-planning. In *Proceedings of ACL*, 2017.

[6] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[7] Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. Sorry, i don't speak sparql: translating sparql queries into natural language. In *Proceedings of the 22nd international conference on World Wide Web*, pages 977–988. ACM, 2013.

[8] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[9] Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, 2015.

[10] Maja Popović. chrf deconstructed: beta parameters and n-gram weights. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 499–504, 2016.

[11] Ehud Reiter and Robert Dale. *Building natural language generation systems*. Cambridge university press, 2000.

[12] Matthew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul. A study of translation edit rate with targeted human annotation. In *Proceedings of association for machine translation in the Americas*, volume 200, 2006.