Democratization of Machine Learning using Semantic Web Technologies

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim
vorgelegt von
Antonios Klironomos
aus Amarousio Attikis, Griechenland

Mannheim, 2026

Dekan:	Prof. Dr. Claus Hertling, Universität Mannheim
Referent:	Prof. Dr. Heiko Paulheim, Universität Mannheim
Korreferent:	Prof. Dr. Evgeny Kharlamov, Universität Oslo

Abstract

The growth of machine learning (ML) has led to a vast and expanding ecosystem of artifacts, including libraries, algorithms, datasets, and pipelines. As a result, creating and identifying effective ML pipelines requires ML expertise and remains complex and time-consuming. This highlights the need for tools that support pipeline creation and meta-learning methods that exploit prior experiment records. This thesis considers two meta-learning tasks: *pipeline performance estimation* (*PPE*) and *dataset performance-based similarity estimation* (*DPSE*), which help identify promising pipelines and relevant test datasets without costly simulations. Existing free pipeline tools do not effectively incorporate linked open data, while meta-learning approaches often overlook experiment and pipeline metadata. Such data is rich in structural relationships (*e.g.*, pipeline components and their connections), so knowledge graphs (KGs) are a suitable representation. This thesis proposes Executable KGs (*ExeKGs*), a semantic representation of ML pipelines in terms accessible to non-ML experts. Building on *ExeKGs*, we introduce *ExeKGLib*, an open-source Python library for their programmatic creation and execution, and *ExeKGLab*, a no-code graphical interface for interacting with them. We further propose *KGmetaSP*, a KG embedding (KGE)-based approach that unifies *ExeKGs* into *MetaExe-KG* to model dataset–pipeline interactions from existing experiment data and improve both *PPE* and *DPSE*. We learn embeddings that enable pipeline-agnostic meta-models for *PPE* and distance-based retrieval for *DPSE*. To capture numeric literals in *MetaExe-KG*, such as hyper-parameters and meta-features, we introduce `ReaLitE`, a relation-centric KGE model that integrates them into relation embeddings. Link prediction and node classification benchmarks show its effectiveness, especially for infrequent relations and relations between entities with correlated literal values. We also apply `ReaLitE` within *KGmetaSP*. We evaluate *KGmetaSP* on our new benchmark of 144,177 OpenML experiments, which enables extensive cross-dataset evaluation. *KGmetaSP* achieves accurate *PPE* with a single pipeline-agnostic meta-model and outperforms *DPSE* baselines.

Zusammenfassung

Das Wachstum des maschinellen Lernens (ML) hat zu einem umfangreichen und stetig wachsenden Ökosystem von Artefakten geführt, darunter Bibliotheken, Algorithmen, Datensätze und Pipelines. Dadurch erfordert die Erstellung und Identifikation effektiver ML-Pipelines weiterhin ML-Expertise und bleibt komplex und zeitaufwendig. Dies verdeutlicht den Bedarf an Werkzeugen, die die Pipeline-Erstellung unterstützen, sowie an Meta-Learning-Methoden, die frühere Experimentaufzeichnungen nutzen. Diese Dissertation betrachtet zwei Meta-Learning-Aufgaben: *Pipeline Performance Estimation* (*PPE*) und *Dataset Performance-based Similarity Estimation* (*DPSE*). Diese Aufgaben helfen dabei, vielversprechende Pipelines und relevante Testdatensätze zu identifizieren, ohne kostenintensive Simulationen durchführen zu müssen. Bestehende frei verfügbare Pipeline-Werkzeuge binden Linked Open Data nicht effektiv ein, während Meta-Learning-Ansätze Experiment- und Pipeline-Metadaten häufig vernachlässigen. Solche Daten enthalten reichhaltige strukturelle Beziehungen, etwa Pipeline-Komponenten und deren Verbindungen, weshalb Knowledge Graphs (KGs) eine geeignete Repräsentation darstellen. Diese Dissertation schlägt Executable KGs (*ExeKGs*) als semantische Repräsentation von ML-Pipelines in Begriffen vor, die auch für Nicht-ML-Experten zugänglich sind. Aufbauend auf *ExeKGs* werden *ExeKGLib*, eine quelloffene Python-Bibliothek für deren programmatische Erstellung und Ausführung, sowie *ExeKGLab*, eine grafische No-Code-Oberfläche für die Interaktion mit ihnen, eingeführt. Darüber hinaus wird *KGmetaSP* vorgeschlagen, ein auf Knowledge Graph Embeddings (KGE) basierender Ansatz, der *ExeKGs* in *MetaExe-KG* vereinheitlicht, um Datensatz–Pipeline-Interaktionen aus bestehenden Experimentdaten zu modellieren und sowohl *PPE* als auch *DPSE* zu verbessern. Dabei werden Embeddings gelernt, die pipeline-agnostische Meta-Modelle für *PPE* sowie distanzbasierte Suche für *DPSE* ermöglichen. Um numerische Literale in *MetaExe-KG*, wie Hyperparameter und Meta-Features, zu erfassen, wird `ReaLitE` eingeführt, ein relationszentriertes KGE-Modell, das diese in Relations-Embeddings integriert. Benchmarks für Link Prediction und Node Classification zeigen seine Effektivität, insbesondere für seltene Relationen und Relationen zwischen Entitäten mit korrelierten Literalwerten. Zudem wird `ReaLitE` innerhalb von *KGmetaSP* angewendet. *KGmetaSP* wird auf einem neuen Benchmark mit 144.177 OpenML-Experimenten evaluiert, der eine umfassende datensatzübergreifende Evaluation ermöglicht. Die Ergebnisse zeigen, dass *KGmetaSP* eine genaue *PPE* mit einem einzigen pipeline-agnostischen Meta-Modell erreicht und bestehende *DPSE*-Baselines übertrifft.

Acknowledgements

I would like to express my sincere and heartfelt gratitude to my supervisors, Prof. Dr. Evgeny Kharlamov, Dr. Mohamed Gad-Elrab, and Prof. Dr. Heiko Paulheim. Their collective guidance, insightful discussions, and continuous support were the cornerstone of this thesis. I am immensely grateful for their mentorship, which was instrumental in helping me navigate the complexities of this PhD and in shaping a clear and impactful research direction. They fostered a positive and intellectually stimulating environment that made this entire process both productive and enjoyable.

A special note of thanks is due to Assistant Professor Baifan Zhou. In the formative stages of my PhD, he was a true mentor to me, offering patient and invaluable guidance, particularly with the writing process. His support was crucial in building the foundation upon which this thesis rests.

I also wish to thank my co-authors and my colleagues in the Data and Web Science Group and the wider research community. Your collaboration, feedback, and insights have been instrumental to the progress and success of this work.

Finally, I want to express my deepest gratitude to my family and friends. Their constant emotional support, patience, and encouragement were essential throughout this academic journey, and this achievement would not have been possible without them.

Contents

Abstract **iii**

Zusammenfassung **iv**

Acknowledgements **v**

1 Introduction **2**
1.1 Motivation . . . 2
1.2 Contributions . . . 4
1.3 Thesis Outline . . . 4
1.4 Publications . . . 5

2 Background **7**
2.1 The Semantic Web: A Web of Data . . . 7
2.1.1 Core Technologies: RDF, RDFS, and OWL . . . 7
2.1.2 Querying the Web of Data: SPARQL . . . 9
2.2 Knowledge Graphs: Structuring Human Knowledge . . . 9
2.3 Ontologies and Data Validation . . . 10
2.3.1 Formalizing Domains with Ontologies . . . 10
2.3.2 Ensuring Data Quality with SHACL . . . 11
2.4 Knowledge Graph Embeddings . . . 12
2.5 The Democratization of Machine Learning . . . 14
2.5.1 Facilitating ML Pipeline Development . . . 14
2.5.2 Identifying Effective Pipelines and Similar Datasets . . . 15

3 Challenges in ML Democratization **17**
3.1 General Challenges in ML Democratization . . . 17
3.2 The Landscape of No-Code ML Tools . . . 18
3.3 The Challenge of Semantic Representation for ML . . . 20
3.4 The Challenge of Incorporating Numeric Literals in KGEs . . . 20
3.5 The Untapped Potential of KGs for Meta-learning . . . 21
3.6 Synthesizing the Challenges . . . 22

4 Executable Knowledge Graphs for Machine Learning **25**
4.1 Introduction . . . 25
4.2 The *ExeKG* Ontological Foundation . . . 25
4.2.1 The Top-Level Data Science Ontology . . . 25
4.2.2 Lower-level Specialized Ontologies . . . 26
4.2.3 Semi-automatic Generation of Lower-level Ontologies . . . 27
4.3 The Executable Knowledge Graph . . . 28
4.3.1 Executability Rules . . . 29
4.4 Conclusion . . . 31

5 Tools for Democratizing ML: *ExeKGLib* and *ExeKGLab* 32
5.1 Introduction . . . 32
5.2 *ExeKGLib*: The Semantic Execution Engine . . . 32
5.2.1 Architecture . . . 33
5.2.2 Programmatic Pipeline Construction . . . 34
5.2.3 Pipeline Execution Mechanism . . . 35
5.2.4 Extending *ExeKGLib* with New Methods . . . 37
5.3 *ExeKGLab*: A No-Code Interface for ML Pipelines . . . 37
5.3.1 GUI Design and Features . . . 37
5.3.2 LLM-Powered Pipeline Recommendation . . . 38
5.4 Evaluation Setup . . . 39
5.4.1 Automated Evaluation Setup for LLM Pipeline Recommendations . . . 39
5.4.2 User Study Design and Procedure . . . 41
5.5 Evaluation Results . . . 43
5.5.1 Results of the Automated LLM Pipeline Evaluation . . . 43
5.5.2 User Study Results . . . 43
5.6 Conclusion . . . 44

6 ReaLitE: A Relation-Centric Literal-Aware Knowledge Graph Embedding Model 46
6.1 Introduction . . . 46
6.2 Related Work . . . 47
6.2.1 Literal-aware Link Prediction . . . 47
6.2.2 Node Classification . . . 47
6.3 The `ReaLitE` Method . . . 48
6.3.1 Problem Description . . . 48
6.3.2 The `ReaLitE` Architecture . . . 48
6.3.3 Aggregation of Numeric Literals . . . 49
6.3.4 Fusing Literals with Relation Embeddings . . . 50
6.3.5 Model Complexity . . . 50
6.4 Experimental Setup . . . 51
6.4.1 Datasets . . . 51
6.4.2 Models . . . 51
6.4.3 Training . . . 52
6.4.4 Evaluation Settings . . . 53
6.5 Results . . . 53
6.5.1 Link Prediction Results . . . 53
6.5.2 Node Classification Results . . . 57
6.6 Conclusion . . . 57

7 *ExeKG* Embeddings for Meta-Learning 58
7.1 Introduction . . . 58
7.2 Related Work . . . 59
7.3 Problem Description . . . 60
7.4 Methodology . . . 60
7.4.1 *MetaExe-KG*: A Knowledge Graph of Datasets and Pipelines . . . 61
7.4.2 *KGmetaSP*: Knowledge Graph Embedding-Based Meta-Learning 62
7.5 Experiments . . . 63
7.5.1 Experimental Setup . . . 63
7.5.2 Evaluation Results . . . 65

7.6 Conclusion 68

8 Summary **70**

9 Limitations and Future Work **71**
9.1 Limitations 71
9.2 Future Work 71
9.2.1 Extensions of the Thesis Contributions 71
9.2.2 Guiding LLM-based Pipeline Recommendations with *KGmetaSP* 72

A Evaluation Details for Chapter 5 **73**
A.1 Implementation Details for the Automated LLM Evaluation 73
A.2 Additional Analysis for the Automated LLM Evaluation 74
A.3 User Study Questionnaires and Responses 75

B Implementation Details and Additional Results for Chapter 6 **81**
B.1 Implementation details 81
B.2 Relation-focused evaluation details 82
B.3 Detailed link prediction results 83
B.4 Literal aggregation choice evaluation 83
B.5 Evaluating model's use of numeric literals for link prediction 84

C Implementation Details for Chapter 7 **85**
C.1 Experimental Configuration 85
C.2 Analysis of Computational Cost 85
C.3 Ablation Study: Impact of *MLSeaKG* Integration 86
C.4 Detailed PPE Results 86
C.4.1 Unseen Datasets 87
C.4.2 Unseen Pipeline Configurations 87

Bibliography **90**

List of Figures

2.1 An illustration of the Semantic Web vision, contrasting the traditional Web of Documents (left), where hyperlinks connect unstructured pages for human interpretation, with the Web of Data (right), where machine-readable predicates form explicit, meaningful relationships between structured data entities. 8
2.2 A sample KG representing facts as a directed, labeled graph. Ellipses denote entities (*e.g.*, ex:Marie_Curie), rectangles denote literal values (*e.g.*, "1903"), and labeled edges represent predicates (*e.g.*, ex:awardWinner) that connect them to form subject-predicate-object triples. 10
2.3 An example of an ontology that provides the formal schema for the scientific awards KG (Figure 2.2). It defines the classes of entities (*e.g.*, ex:NobelPrize, dbo:Scientist), and the rules for the properties that connect them by specifying rdfs:domain and rdfs:range constraints. 10
2.4 An illustration of performance-based dataset similarity. It is calculated by representing each dataset's performance profile across a common set of pipeline configurations as a vector (left) and then measuring the distance between these vectors (right). 15

4.1 The hierarchical structure of the ontologies used in the *ExeKG*. Specialized lower-level ontologies inherit from and extend the core concepts of the top-level Data Science ontology. 26
4.2 Data Science (DS) ontology 26
4.3 Machine Learning ontology 27
4.4 The two lower-level ontologies used in the *ExeKG*: (A) Statistics and (B) Visualization. 28
4.5 An example of a simple classification pipeline configuration represented as an *ExeKG*. The graph explicitly models the sequence of tasks, the methods used, the flow of data entities, and the corresponding hyper-parameters 29

5.1 Comparison between conventional code and *ExeKGLib* for a classification problem. Fig. 5.3 details how this is achieved. 33
5.2 The object-oriented class hierarchy of *ExeKGLib*, which mirrors the structure of the *ExeKG* ontologies. 34
5.3 Executable KG construction. Linking semantic concepts (*e.g.*, *Task* KG entity) with programming elements (*e.g.*, *Task* Python object). Sample code in Fig. 5.1. 35
5.4 The graphical user interface of *ExeKGLab*, showing the sidebar with available components and the main canvas for constructing pipeline configurations. 40
5.5 The LLM-powered AI Assistant in *ExeKGLab*, which provides pipeline recommendations based on natural language queries. 40
5.6 LLM-Powered Recommendations Flow 41

6.1 An example showing the potential benefit of the mathematical relationship between people's income and houses' rent during link prediction. 47
6.2 The ReaLitE pipeline for relation embedding enrichment. Numeric literals from head and tail entities are aggregated (Step 1) and then combined with the relation embedding (Step 2) before being used by the base KGE model's scoring function. 49

7.1 Comparison of the information leveraged by our proposed approach (green) versus existing approaches (orange) for representing datasets in meta-learning. For a pair of datasets, A and B, existing approaches represent datasets using only intrinsic meta-feature vectors (top-right). Our proposed approach also includes past pipeline-configuration metadata (top left) as additional context into a KG, to represent datasets. 59
7.2 *KGmetaSP* overview. 61
7.3 High-level view of *MetaExe-KG*. 63
7.4 Embedding aggregation process. 63
7.5 High-level view of the *PPE* evaluation setting. (a) Standard approach: one configuration-specific meta-model predicts the performance of a single pipeline configuration from a dataset representation. (b) Proposed approach: a single configuration-agnostic meta-model predicts performance from both dataset and pipeline representations. 64

A.1 Pre-study questionnaire 76
A.2 User satisfaction 77
A.3 Ease of use and confidence 77
A.4 Impact of AI assistant and general comments 78
A.5 Resulting pipeline for classification task from participant 79
A.6 Participant logs example 1 79
A.7 Participant logs example 2 79
A.8 User comments example 1 80
A.9 User comments example 2 80
A.10 User comments example 3 80
A.11 User comments example 4 80
A.12 User comments example 5 80

B.1 Boxplot of relation frequencies for YAGO15k. The frequency is measured as the number of training triples that contain the relation. The boxplot's upper extreme is in bold. Outliers are depicted with dots. . . 82
B.2 Heatmaps of pairwise Pearson's correlation ("$corr.\ coef.$") between head (suffix 'h') and tail (suffix 't') numeric attributes per relation in YAGO15k's training set. These relations have at least one pair of head and tail attributes with $|corr.\ coef.| \geq 0.2$ in the training set. They group Table 6.6's test triples. Correlations with $|corr.\ coef.| < 0.05$ are blank. 82
B.3 Fluctuation of model performance for various choices of literal aggregation method during HPO. "all_learnable" is function **y** from Eq. 6.3. **Left**: FB15k-237's validation set. **Right**: YAGO15k's validation set. . . 84

List of Tables

3.1 Comparison of existing free open-source tools for no-code or low-code ML pipeline development. Checkmarks indicate explicit support for the corresponding feature. Missing features that are central to the proposed approach are denoted by '▲'. 19
3.2 Comparison of existing KG schemata and ontologies relevant to ML. Checkmarks indicate explicit support for the corresponding focus area or representation capability. Missing features that are central to the proposed representation are denoted by '▲'. 21

4.1 ExeKG ontology statistics 27

5.1 Indicative runtime overhead of the semantic execution layer for a four-task Random Forest pipeline. 36
5.2 Automated evaluation results for LLM-generated pipeline recommendations against OpenML reference pipelines. 43
5.3 Quantitative user-study metrics across system variants. Completion times are rounded average durations in minutes, computed separately for each task over 24 attempts per system. Task success and task search use are reported as counts and percentages over 48 task attempts per system. Errors and AI assistant use are reported as means per task attempt over the same 48 attempts. 44
5.4 Participant ratings across system variants on a 5-point Likert scale. Values indicate mean ratings aggregated over participants. 44

6.1 Number of trainable parameters for LiteralE and for variations of `ReaLitE`. B represents the number of trainable parameters of base models (*e.g.*, TransE). The parentheses in $(+12)$ indicate that this number is added only if the learnable function $\mathbf{y}$ is used to aggregate literals. 51
6.2 Datasets used for link prediction 51
6.3 Datasets used for node classification 52
6.4 Link prediction results. In this context, `ReaLitE` uses g_{lin} as it yielded better results than g_{gated}. The baseline results are sourced from the current state-of-the-art paper, with the best results per metric and base KGE model in bold. 54
6.5 MRR for the filtered test set of YAGO15k based on relation types grouped by their frequency. "Long-tail" refers to the test triples that contain relations present in $\leq 2.55\%$ of the training triples. "Frequent" refers to the remaining test triples. "All triples" refers to all test triples. The best results per base model per triple group are in bold. The percentages indicate the MRR increase of `ReaLitE` relative to the corresponding vanilla baseline. 55

6.6 MRR for the filtered test set of YAGO15k based on relation types grouped by literal correlation. "Corr. lit." refers to the test triples where the head and tail attributes are more correlated (the relation has at least one pair of head and tail attributes with $|corr.coef.| \geq 0.2$ in the training set). "Less corr. lit." refers to the remaining test triples where the head and tail attributes are less correlated. "All triples" refers to all test triples. The best results per base model per triple group are in bold. The percentages indicate the MRR increase of `ReaLitE` relative to the corresponding vanilla baseline. 56
6.7 Aggregation function used in the best hyper-parameter configurations of `ReaLitE`. Function $\mathbf{y}$ is a learnable combination of all supported aggregation types (Eq. 6.3). 57
6.8 Node classification results – Micro-F1 score. In this context, `ReaLitE` uses g_{gated} as it yielded better results than g_{lin}. The best results per dataset are underlined. The best results per base KGE model per classifier are in bold. 57

7.1 *MetaExe-KG* statistics. 64
7.2 *PPE* meta-classification results for unseen datasets. Best overall scores are underlined, and best scores per meta-feature group are bold. 66
7.3 *PPE* results for unseen pipeline configurations. For each target, *KGmetaSP*$_{\text{PE}}^{\text{REALITE}}$ denotes the LP-based variant among ComplEx, DistMult, and TransE with the highest meta-classification F1. *KGmetaSP*$_{\text{PE}}^{\text{RDF2VEC}}$ denotes the walk-based *RDF2Vec* variant. Best scores are underlined. . 67
7.4 Hit@k of similarity measures according to performance-based ground truth. A hit is recorded when a dataset retrieved within the top-k has a ground-truth similarity with the query dataset greater than a similarity threshold (ST). Results are reported for $ST = 0.8$ and $ST = 0.9$. The highest score per metric is in bold and the second highest is underlined. 67
7.5 NDCG@k of similarity measures according to performance-based ground truth. The highest score per metric is in bold and the second highest is underlined. 67

9.1 Preliminary results for *KGmetaSP*-guided LLM pipeline generation on six unseen OpenML classification datasets. 72

A.1 LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 10 and Run 1847757 (Jaccard: 0.3333, Hit Rate: 1.0000). 74
A.2 LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 20 and Run 2087085 (Jaccard: 0.3333, Hit Rate: 0.0000).) 74
A.3 LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 39 and Run 1860346 (Jaccard: 0.3333, Hit Rate: 0.5000). 75

B.1 Hyper-parameter Values for Model Combinations 81

B.2 MRR for the filtered test set of YAGO15k based on relation types grouped by symmetry. 'Symm. rel.' refers to the test triples that contain symmetric relations. 'Asymm. rel.' refers to the remaining test triples that contain asymmetric relations. 'All triples' refers to all test triples. The best results per base model per triple group are in bold. The best results per triple group are underlined. 83
B.3 Accuracy on the synthetic dataset for literal usage evaluation. The baseline results are sourced from the paper that proposed the evaluation method. 84

C.1 Ablation of *MLSeaKG* integration for *PPE*. Values report Δ = With − Without. For MSE, negative values indicate improvement; for R^2, Accuracy, and F1, positive values indicate improvement. 86
C.2 Ablation of *MLSeaKG* integration for *DPSE*. Values report Δ = With − Without. For Hit@k, datasets with similarity > 0.9 are considered hits. 87
C.3 Detailed *PPE* results for unseen datasets. 88
C.4 Detailed *PPE* results for unseen pipeline configurations. 89

List of Abbreviations

AI	**A**rtificial **I**ntelligence
AMLB	**A**uto**ML** **B**enchmark
API	**A**pplication **P**rogramming **I**nterface
AutoML	**Auto**mated **M**achine **L**earning
Chi2	**Chi-Squared** Feature Selection
CLI	**C**ommand **L**ine **I**nterface
ComplEx	**Compl**ex **E**mbeddings
DCAT	**D**ata **Cat**alog Vocabulary
DistMult	**Dist**ributional **Mult**iplication
DPSE	**D**ataset **P**erformance-based **S**imilarity **E**stimation
DT	**D**ecision **T**ree
DV	**D**ict **V**ectorizer
ET	**E**xtra **T**rees
ExeKG	**Exe**cutable **K**nowledge **G**raph
FB15k	**F**ree**b**ase **15k**
FB15k-237	**F**ree**b**ase **15k** with **237** relations
FH	**F**eature **H**asher
GB	**G**radient **B**oosting
GED	**G**raph **E**dit **D**istance
GNN	**G**raph **N**eural **N**etwork
GUI	**G**raphical **U**ser **I**nterface
HistGB	**Hist**ogram **G**radient **B**oosting
HPO	**H**yperparameter **O**ptimization
IRI	**I**nternationalized **R**esource **I**dentifier
kNN	**k**-**N**earest **N**eighbors
KBLN	**K**nowledge **B**ase Representations from **L**atent and **N**umerical Features
KBLRN	**K**nowledge **B**ase Representations from **L**atent, **R**elational, and **N**umerical Features
KG	**K**nowledge **G**raph
KGA	**K**nowledge **G**raph **A**ugmentation
KGE	**K**nowledge **G**raph **E**mbedding
LCWA	**L**ocal **C**losed **W**orld **A**ssumption
LiteralE	**Literal** **E**mbeddings
LLM	**L**arge **L**anguage **M**odel
LOD	**L**inked **O**pen **D**ata
LP	**L**ink **P**rediction
LR	**L**ogistic **R**egression
LRCV	**L**ogistic **R**egression with **C**ross-**V**alidation
Meta-F.	**Meta**-**f**eature P-Norm Distance
Meta-F. N	**Meta**-**f**eature P-Norm **N**ormalized Distance
MIC	**M**utual **I**nformation **C**lassification
MKGA	**M**ultimodal **K**nowledge **G**raph **A**ugmentation
ML	**M**achine **L**earning

MLP	**M**ulti-**L**ayer **P**erceptron
MLSeaKG	**M**achine **L**earning **Sea**mantic Layer **K**nowledge **G**raph
MLSO	**M**achine **L**earning **S**chema **O**ntology
MMS	**M**in**M**ax **S**caler
MRR	**M**ean **R**eciprocal **R**ank
MSE	**M**ean **S**quared **E**rror
MuFi-Dat.	**Mu**lti-**Fi**eld Document Ranking (**Dat**aset Meta-features)
NDCG	**N**ormalized **D**iscounted **C**umulative **G**ain
OE	**O**rdinal **E**ncoder
OHE	**O**ne-**H**ot **E**ncoder
OpenML	**Open M**achine **L**earning
OWL	**W**eb **O**ntology **L**anguage
PCA	**P**rincipal **C**omponent **A**nalysis
PMLB	**P**enn **M**achine **L**earning **B**enchmark
PPE	**P**ipeline **P**erformance **E**stimation
PT	**P**ower **T**ransformer
QT	**Q**uantile **T**ransformer
RDF	**R**esource **D**escription **F**ramework
RDF2Vec	**RDF to Vec**tor
RDFS	**RDF S**chema
ReaLitE	**REl**A**t**ion-centric **LIT**eral-aware knowledge graph **E**mbedding
RF	**R**andom **F**orest
RFE	**R**ecursive **F**eature **E**limination
RMSE	**R**oot **M**ean **S**quared **E**rror
RotatE	**Rotat**ional **E**mbeddings
RS	**R**obust **S**caler
SHACL	**Sha**pes **C**onstraint **L**anguage
SiFi-Dat.	**Si**ngle-**Fi**eld Document Ranking (**Dat**aset Meta-features)
SiFi-Pip.	**Si**ngle-**Fi**eld Document Ranking (**Pip**eline Configuration Description)
SKB	**S**elect **K** **B**est
SP	**S**elect **P**ercentile
SPARQL	**SPARQL P**rotocol **a**nd **R**DF **Q**uery **L**anguage
SS	**S**tandard **S**caler
SVM	**S**upport **V**ector **M**achine
TE	**T**arget **E**ncoder
TransE	**Trans**lating **E**mbeddings
TuckER	**Tuck**er Embeddings for **E**ntities and **R**elations
XSD	**X**ML **S**chema **D**atatypes
YAGO	**Y**et **A**nother **G**reat **O**ntology

List of Namespaces

`:`	Default/local namespace, used for locally defined resources in examples and SHACL shapes, such as `:AtomicTask`.
`dbo`	DBpedia ontology namespace, used in background examples for external ontology terms such as `dbo:Scientist`.
`ds`	Data Science ontology, used for general data science concepts such as data, tasks, methods, and pipelines.
`ex`	Example namespace, used in illustrative RDF and ontology examples, such as `ex:Dog`, `ex:Person`, and `ex:name`.
`ml`	Machine Learning ontology, used for ML-specific tasks, methods, data entities, parameters, and constraints.
`owl`	Web Ontology Language vocabulary, used for ontology classes, properties, individuals, and axioms.
`rdf`	Resource Description Framework vocabulary, used for core RDF terms such as `rdf:type` and `rdf:Property`.
`rdfs`	RDF Schema vocabulary, used for class hierarchies and schema-level descriptions, such as `rdfs:subClassOf`, `rdfs:domain`, and `rdfs:range`.
`sh`	SHACL vocabulary, used for validation constraints over RDF graphs, such as `sh:NodeShape`, `sh:property`, and `sh:minCount`.
`stats`	Statistics ontology, used for statistical tasks and methods, including statistical measures and data transformations.
`visu`	Visualization ontology, used for visualization-related tasks and methods, such as canvas and plotting tasks.
`xsd`	XML Schema Datatypes vocabulary, used for datatypes such as `xsd:string`, `xsd:int`, and `xsd:float`.

Part I

Introduction and Foundations

Chapter 1

Introduction

This chapter provides an overview of the research presented in this thesis. It begins by outlining the motivation behind the work, followed by a summary of the key contributions. The chapter then presents the structure of the thesis and concludes with a list of publications that have resulted from this research.

1.1 Motivation

Thanks to the remarkable progress in Computer Science and specifically the field of machine learning (ML), there is a plethora of ML artifacts, such as pipelines and datasets, publicly accessible. In both academia and industry, the popularity of ML is continuously increasing [130]. Many experts in multiple domains are learning ML and want to apply it for their questions, *e.g.*, biologists [89, 74], oncologists [81, 3], and engineers in the industry [99, 120].

A *pipeline* refers to a data mining workflow consisting of sequentially executed components such as preprocessing, feature extraction, feature selection, model estimation, and post-processing, which together transform raw data into a predictive model [39]. A *pipeline configuration* denotes an instantiation of a pipeline, where all component hyper-parameters are specified.

The development of ML pipelines remains a complex and time-consuming endeavor, posing a barrier for domain experts who have limited ML background and time for acquiring new skills. This complexity is increased by the vast variety of potential pipeline configurations, which makes finding an optimal solution a challenge both for experts and non-experts.

The availability of publicly shared ML experiment data on the web has stimulated growing interest in meta-learning, which uses past experiments to improve performance [67]. Within this paradigm, two central tasks have emerged: *pipeline performance estimation (PPE)* [29] and *dataset performance-based similarity estimation (DPSE)* [86]. *PPE* focuses on predicting how a given pipeline configuration will perform on a target dataset by training predictive models, commonly referred to as meta-models [29]. *DPSE* seeks to quantify similarity between datasets based on how consistently they perform with the same pipeline configurations [48]. Solving these tasks enables early identification of promising pipelines and reduces the search space, thereby mitigating the need for computationally expensive training and optimization procedures [48, 139].

Although the idea of lowering the barrier of ML pipeline development is not new, existing solutions have notable limitations. A group of solutions are AutoML systems [68] that, despite their success, often require substantial computational resources due to exhaustive search and repeated pipeline evaluation, while at the same time providing limited customizability and overlooking related tasks such as data visualization and feature engineering. Similarly, tools like RapidMiner [101]

and KNIME [13] offer user-friendly interfaces, but they do not deeply incorporate linked open data (LOD) principles, which limits the interoperability and reuse of the pipelines created. Our work utilizes LOD throughout the entire pipeline lifecycle.

Existing approaches to *PPE* and *DPSE* often characterize datasets using numeric descriptors known as meta-features [30, 71]. These features capture structural properties (*e.g.*, number of instances), statistical characteristics (*e.g.*, skewness), and landmarker signals [11] (*e.g.*, classifier performance). While using only meta-features, these methods overlook the extensive experimental metadata publicly available. To capture the relationships between ML pipelines and the datasets on which they were evaluated, past experiment metadata can be structured using semantic web standards and represented within knowledge graphs (KGs). These links enable *DPSE* even when datasets do not share pipeline configurations and capture historical dataset–pipeline interaction patterns that can inform *PPE*. When combined with literal-aware KGE models that incorporate numeric attributes (*e.g.*, hyper-parameters), this representation can further enhance both *PPE* and *DPSE*.

This thesis addresses these interconnected challenges of ML democratization by introducing executable KGs (*ExeKGs*), a representation of ML pipelines in simple terminology that is understandable to non-ML experts. Within an *ExeKG*, a pipeline configuration is a sequence of steps encoded as entities, which are linked to entities representing their input and output data, while their hyper-parameters are captured as attributes. The construction of *ExeKGs* is based on a hierarchical suite of ontologies, comprising a top-level data science ontology for generic concepts and three specialized lower-level ontologies for ML, statistics, and visualization operations. Based on this semantic foundation, we propose *ExeKGLib*, an open-source Python library for the programmatic creation and execution of *ExeKGs*. We further introduce *ExeKGLab*, a no-code graphical user interface that facilitates interaction with *ExeKGs*.

Then, we propose *KGmetaSP*, a new approach for exploiting latent relationships between datasets and pipelines. It encodes past experiment data as *ExeKGs*, integrates it into a unified *MetaExe-KG*, and learns KGEs. We use pipeline embeddings to train meta-models for *PPE* and use vector distance measures for *DPSE*. Since *MetaExe-KG* includes numeric information such as pipeline hyper-parameters and dataset meta-features, effective performance prediction requires embedding models that can explicitly leverage this information. For this reason, we present `ReaLitE`, a relation-centric KGE model designed to incorporate numeric literal information into relation embeddings. `ReaLitE` aggregates numeric attributes of the head and tail entities connected by a relation and integrates them with the relation embedding through a learnable mechanism, enabling the modeling of correlations between entity literals. On link prediction and node classification benchmarks, `ReaLitE` consistently outperforms existing KGE models, with particularly strong gains for infrequent relations and relations between entities with correlated numeric values. We apply `ReaLitE` within *KGmetaSP* to address *PPE* using a meta-model that combines dataset meta-features with pipeline embeddings. To validate *KGmetaSP*, we created a large-scale performance benchmark after running 144,177 pipeline experiments on OpenML [148] datasets. The results show that *KGmetaSP* achieves accurate *PPE*, while also delivering improved performance in *DPSE*.

1.2 Contributions

This thesis explores the use of Semantic Web technologies, particularly KGs and ontologies, to democratize discovery of ML datasets and pipelines, and creation of ML pipelines. The contributions are summarized below:

1. **A Suite of Ontologies for the ML Domain:** To make the domain accessible to non-ML experts and to enable automated pipeline validation and execution, we propose a hierarchical group of four ontologies. It comprises a general, top-level Data Science ontology and three specific, lower-level ontologies covering ML, Statistics, and Visualization.

2. **A Semantic Framework for ML Pipelines using Executable Knowledge Graphs (*ExeKGs*):** We introduce a framework for representing ML pipelines as *ExeKGs*. This includes two tools: *ExeKGLib*, our open-source Python library for the programmatic creation, manipulation, and execution of *ExeKGs*; and *ExeKGLab*, a GUI-based platform powered by *ExeKGLib*. *ExeKGLab* uses an LLM-powered assistant, making pipeline development accessible to non-ML experts.

3. **A Novel Literal-Aware KGE Model:** We developed `ReaLitE`, a novel KGE model that serves as a literal-aware extension to triple-scoring KGE models. Our model enriches relation embeddings by incorporating numeric information from the head and tail entities it connects. It aggregates the numeric attributes for the head and tail entities into two respective vectors, then integrates them with the original relation embedding to capture potential correlations. Its superiority over state-of-the-art methods was demonstrated through evaluation on public link prediction and node classification benchmarks.

4. **Performance Prediction and Dataset Similarity with Knowledge Graph Embeddings:** We propose *KGmetaSP*, a novel approach to predict pipeline performance and estimate performance-based dataset similarity. Our approach encodes past experiment data into our proposed *MetaExe-KG* comprising 2,616 *ExeKGs*. The *MetaExe-KG* captures the connections between datasets and the pipeline configurations applied to them, as well as the internal structure of these configurations. *KGmetaSP* applies KGE techniques to generate vector representations of pipeline configurations and datasets. It uses configuration vectors as input to a meta-model trained to predict pipeline performance on a given dataset, and vector distances to find performance-similar datasets. For evaluation purposes, we constructed a large performance benchmark after running 144,177 experiments on OpenML datasets. We evaluated *KGmetaSP* against baselines using our benchmark as ground truth, showing accurate pipeline-agnostic *PPE* and improved *DPSE*.

1.3 Thesis Outline

This thesis is organized into four parts.

Part I: Introduction and Foundations provides the motivation and foundational knowledge for the research.

- **Chapter 1: Introduction.** This chapter introduces the research, outlines the motivation, summarizes the main contributions, and presents the structure of the thesis.

- **Chapter 2: Background.** This chapter introduces the foundational concepts of this thesis.
- **Chapter 3: Challenges in ML Democratization.** This chapter provides a detailed analysis of the landscape of no-code ML tools and the specific opportunity of using knowledge graphs in meta-learning, which together motivate the contributions of this thesis.

Part II: A Semantic Framework for ML Pipelines details the core framework developed in this thesis.

- **Chapter 4: Executable Knowledge Graphs for ML Pipelines.** This chapter details the core of our approach, presenting the ontologies and the formal structure of *ExeKGs* for representing ML pipelines.
- **Chapter 5: Tools for Democratizing ML: *ExeKGLib* and *ExeKGLab*.** This chapter describes the design and implementation of our tools, detailing the functionality of the open-source Python library and the features of the no-code graphical user interface.

Part III: Knowledge Graph Embeddings for Link Prediction and Downstream Tasks presents the novel methods developed for learning from and on top of the proposed framework.

- **Chapter 6: `ReaLitE`: A Relation-Centric Literal-Aware Knowledge Graph Embedding Model.** This chapter presents the design, implementation, and evaluation of our novel KGE model.
- **Chapter 7: Integrating Meta-Features with *ExeKG* Embeddings for Meta-Learning.** This chapter presents our novel approach for predicting pipeline performance and identifying performance-similar datasets using our *ExeKG* framework.

Part IV: Conclusion and Outlook summarizes the work and outlines future directions.

- **Chapter 8: Summary.** This chapter summarizes the main findings and contributions of the thesis.
- **Chapter 9: Limitations and Future Work.** This chapter discusses the limitations of the presented work and outlines directions for future research.

1.4 Publications

The work presented in this thesis has been published in the following peer-reviewed conference papers and workshops:

- Klironomos, A., Zhou, B., Tan, Z., Zheng, Z., Gad-Elrab, M. H., Paulheim, H., & Kharlamov, E. (2023). ExeKGLib: Knowledge Graphs-Empowered Machine Learning Analytics. In C. Pesquita, H. Skaf-Molli, et al. (Eds.), *The Semantic Web: ESWC 2023 Satellite Events* (pp. 123–127). Springer Nature Switzerland.
- Klironomos, A., Gad-Elrab, M. H., & Kharlamov, E. (2024). Empowering Industry Professionals with Machine Learning through Knowledge Graphs. In: *Proceedings of the International Conference on Semantic Systems (SEMANTiCS)*.

- Klironomos, A. (2025). Towards Democratized Machine Learning: A Semantic Web Approach. In *Companion Proceedings of the ACM on Web Conference 2025* (pp. 697–700). Association for Computing Machinery.
- Sajid, S., Klironomos, A., Kharlamov, E., Hopfgartner, F., & Gad-Elrab, M. H. (2025). No-Code ML Pipeline Development: Leveraging Knowledge Graphs and Language Models. In: *European Semantic Web Conference*.
- Klironomos, A., Zhou, B., Zheng, Z., Gad-Elrab, M. H., Paulheim, H., & Kharlamov, E. (2025). ReaLitE: Enrichment of Relation Embeddings in Knowledge Graphs using Numeric Literals. In *The Semantic Web: ESWC 2025* (pp. 41–58). Springer Nature Switzerland.
- Klironomos, A., Zhou, B., Tan, Z., Zheng, Z., Gad-Elrab, M. H., Paulheim, H., & Kharlamov, E. (2025). ExeKGLib: A Platform for Machine Learning Analytics based on Knowledge Graphs. In *Proceedings of the 24th International Semantic Web Conference (ISWC 2025)*.
- Klironomos, A., Dasoulas, I., Periti, F., Gad-Elrab, M. H., Paulheim, H., Dimou, A., & Kharlamov, E. (2026). Integrating Meta-features with Knowledge Graph Embeddings for Meta-learning. In *The Semantic Web: ESWC 2026* (pp. 336–357). Springer Nature Switzerland.

Chapter 2

Background

This chapter establishes the scientific foundations upon which this dissertation is built. It provides an overview of the core concepts and technologies that surround the research presented here. The chapter begins with an exploration of the Semantic Web, detailing its vision and the key standards that enable a machine-readable web, including the Resource Description Framework (RDF), RDF Schema (RDFS), and the Web Ontology Language (OWL). It then introduces the concept of KGs as a means of structuring real-world knowledge. Following this, the chapter discusses the mechanisms for ensuring data quality and consistency within these graphs through the use of ontologies and the Shapes Constraint Language (SHACL). The subsequent sections present KGE as a method for representation learning on graphs, and finally, the chapter explains the principles of ML democratization and the methodologies of dataset similarity for meta-learning. These sections provide the necessary concepts for understanding the contributions of this thesis.

2.1 The Semantic Web: A Web of Data

The current World Wide Web is mainly a web of documents designed for human consumption [12] (Figure 2.1). While humans can directly interpret the meaning and context of text and images on a webpage, machines cannot. Computers can parse the structure of a webpage for layout and routine processing, but they have no reliable way to process the semantics—the underlying meaning—of the content [64]. The Semantic Web, a vision articulated by Tim Berners-Lee, proposes an extension of the current web to address this limitation [9]. The goal is to create a "Web of Data" (Figure 2.1) where information is given a well-defined, machine-interpretable meaning, enabling software agents to carry out sophisticated tasks on behalf of users and increase cooperation between people and computers [60].

This vision is realized through a common framework of standards, led by the World Wide Web Consortium (W3C), that allows data to be shared and reused across applications, enterprises, and communities. A key realization of this vision is the LOD paradigm, which promotes the publication of structured data on the Web using open standards and explicit interlinking between data sources. The core principle is to move from a web of linked documents to a web of linked data, where the relationships between things are as important as the things themselves [15].

2.1.1 Core Technologies: RDF, RDFS, and OWL

The foundation of the Semantic Web is a stack of technologies designed to represent structured data and the rules that govern it.

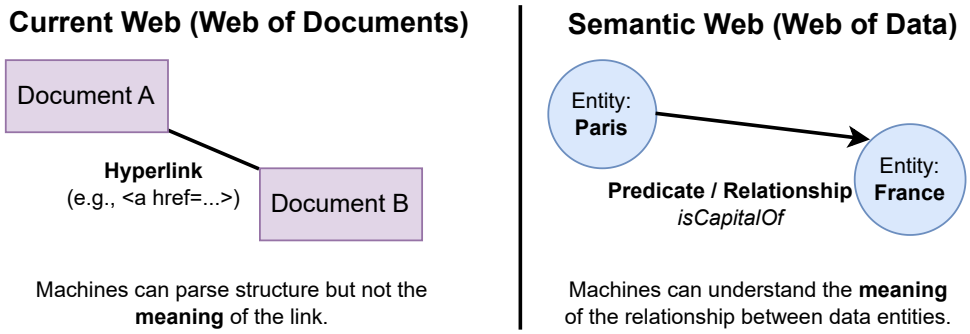


FIGURE 2.1: An illustration of the Semantic Web vision, contrasting the traditional Web of Documents (left), where hyperlinks connect unstructured pages for human interpretation, with the Web of Data (right), where machine-readable predicates form explicit, meaningful relationships between structured data entities.

Resource Description Framework (RDF) The Resource Description Framework (RDF) is the standard data model for the Semantic Web, adopted as a W3C recommendation in 1999 and subsequently updated [56]. It provides a method for describing resources and the relationships between them. The core structure of RDF is the triple, an expression of the form 'subject-predicate-object' [35]. This structure allows RDF to model information as a directed, labeled graph, where subjects and objects are nodes, and predicates are the labeled, directed edges connecting them, as illustrated in Figure 2.2.

RDF resources are identified using Internationalized Resource Identifiers (IRIs). An IRI is a unique global identifier used to name a resource, ensuring that data can be linked across different sources without ambiguity. Since IRIs can be long and difficult to read, RDF serializations commonly use namespace prefixes as compact aliases. A namespace denotes the base IRI of a vocabulary, while a prefix is the shortened label used to refer to terms from that vocabulary. For example, `rdfs:subClassOf` refers to the term `subClassOf` in the RDF Schema namespace. Throughout this thesis, such prefixes are used to improve readability when referring to ontology classes, properties, and constraints. The namespaces used in the thesis are summarized in the List of Namespaces.

The components of an RDF triple are defined as follows [64]: The subject is the resource being described. It must be an IRI or a blank node. The predicate is a property or characteristic of the subject and defines the relationship between the subject and the object. It must be an IRI. The object is the value of the property. It can be an IRI, a blank node, or a literal value. A blank node is an existential node used when a resource is known to exist but does not have a specific IRI. A literal is a datatype value, such as a string, number, or date, often defined using XML Schema Datatypes (XSD). A collection of such triples forms an RDF graph [156].

RDF Schema (RDFS) While RDF provides a way to make statements, it does not provide a vocabulary for defining the types of things being described. RDF Schema (RDFS) is a W3C recommendation that extends RDF with a basic vocabulary for this purpose [25]. RDFS allows for the creation of simple ontologies by introducing core concepts such as 'rdfs:Class', 'rdfs:subClassOf', 'rdf:Property', 'rdfs:domain', and 'rdfs:range'. These constructs allow developers to define hierarchies of classes and properties, specifying which properties can be applied to which classes and what kind of values those properties can have. For example, one can state that 'Dog' is

a subclass of 'Animal' ('ex:Dog rdfs:subClassOf ex:Animal'), and that the property 'hasOwner' ('ex:hasOwner') has a domain of 'Animal' and a range of 'Person'.

Web Ontology Language (OWL) For applications requiring more expressive power than RDFS can provide, the W3C recommends the Web Ontology Language (OWL) [151, 61]. Derived from earlier work on DAML+OIL, an early ontology language for the Semantic Web, OWL is a family of knowledge representation languages designed to formally describe the semantics of a domain in rich detail [98, 66]. OWL allows for the creation of complex ontologies by providing a much richer vocabulary than RDFS. It supports defining more sophisticated class relationships (*e.g.*, disjointness, equivalence), complex property characteristics (*e.g.*, transitive, symmetric, functional properties), and cardinality constraints [61]. In its original specification, OWL distinguished between three variants: OWL Lite, OWL DL, and OWL Full. Each reflects different trade-offs between expressiveness and computational decidability. OWL DL was designed to provide maximum expressiveness while retaining decidable reasoning, OWL Lite offered a more restricted language for simpler use cases, and OWL Full imposed no formal restrictions but does not guarantee decidable reasoning. OWL has since evolved into OWL 2, which introduces a revised set of dialects, or profiles, with varying levels of expressiveness and computational complexity, including OWL 2 EL, OWL 2 QL, and OWL 2 RL, designed to support polynomial-time reasoning for specific application scenarios [103]. This expressiveness enables automated reasoning, allowing computer systems to infer facts that are not explicitly stated in the data, a key component of the Semantic Web vision [9].

2.1.2 Querying the Web of Data: SPARQL

To retrieve and manipulate data stored in RDF format, the W3C standardized the SPARQL Protocol and RDF Query Language [138]. SPARQL is a semantic query language designed to query data across diverse sources, whether stored natively as RDF or viewed as RDF through middleware [111, 118]. A SPARQL query consists of triple patterns, conjunctions, and disjunctions that are matched against an RDF graph. It allows users to express complex queries that can join data across different datasets, filter results, and aggregate data, making it an essential tool for interacting with the Web of Data. The SPARQL protocol also defines a standard way for clients to send queries and receive results from a SPARQL service, mainly over HTTP [43].

2.2 Knowledge Graphs: Structuring Human Knowledge

The term "Knowledge Graph" (KG) has gained attention from both industry and academia as a vessel for accumulating and conveying real-world knowledge [109]. While the term is used in various contexts, a KG can be formally defined as a graph of data whose nodes represent entities of interest and whose edges represent the relationships between these entities [69]. Prominent examples of large-scale KGs include DBpedia [85], YAGO [95], and Wikidata [150], which are modeled in RDF and serve as foundational resources for many intelligent applications.

A KG's structure is that of a directed, labeled graph, often built upon the RDF data model [44]. The nodes of the graph are entities, which can be real-world objects (*e.g.*, a city), abstract concepts (*e.g.*, a scientific theory), or events. To avoid ambiguity, especially when integrating data from multiple sources, entities are identified using

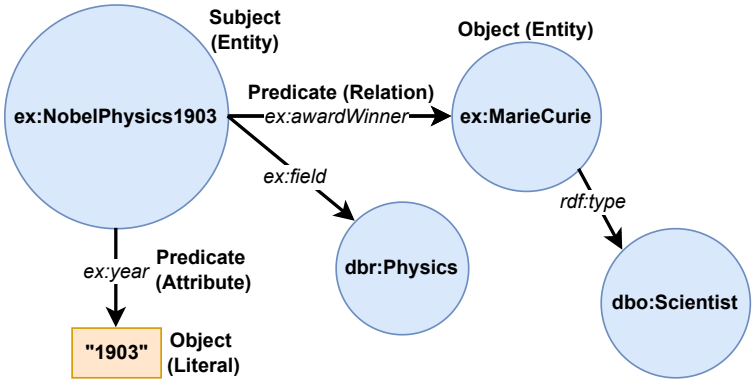


FIGURE 2.2: A sample KG representing facts as a directed, labeled graph. Ellipses denote entities (*e.g.*, ex:Marie_Curie), rectangles denote literal values (*e.g.*, "1903"), and labeled edges represent predicates (*e.g.*, ex:awardWinner) that connect them to form subject-predicate-object triples.

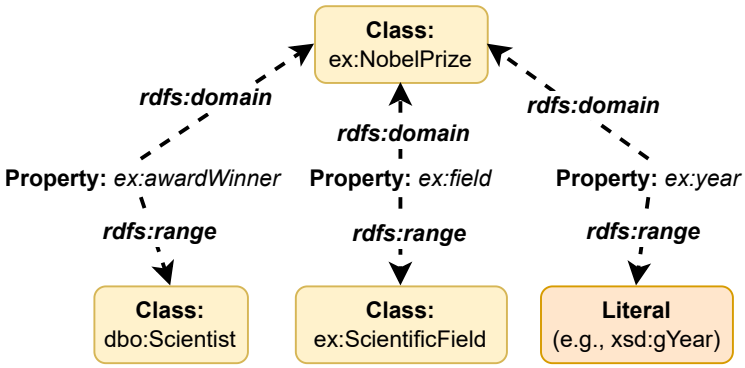


FIGURE 2.3: An example of an ontology that provides the formal schema for the scientific awards KG (Figure 2.2). It defines the classes of entities (*e.g.*, ex:NobelPrize, dbo:Scientist), and the rules for the properties that connect them by specifying rdfs:domain and rdfs:range constraints.

persistent and globally unique identifiers, specifically IRIs [41]. The edges represent binary relations between entities and are labeled with the type of relation (*e.g.*, 'isLocatedIn', 'isAuthorOf'). In addition to entity nodes, KGs also contain literal nodes, which represent datatype values like strings, numbers, and dates, providing concrete attributes for the entities they describe [69].

2.3 Ontologies and Data Validation

For a KG to be reliable and useful, the data within it must be consistent and adhere to a set of predefined rules. Ontologies provide the semantic schema for defining these rules, while constraint languages provide the mechanism for validating data against them.

2.3.1 Formalizing Domains with Ontologies

As introduced in Section 2.1.1, an ontology is a formal specification of a conceptualization [140]. It serves as the semantic backbone for a KG by defining the classes of objects that can exist in a domain, the properties that can be used to describe them,

and the relationships that can exist between them [9]. The practice of using ontologies to create a shared, unambiguous understanding of a domain is well-established across many scientific and commercial disciplines.

The biomedical field, for example, has long relied on ontologies to integrate and analyze complex biological data. The Gene Ontology (GO) [142] is a cornerstone of bioinformatics, providing a controlled vocabulary to describe the molecular function, biological process, and cellular component of gene products. In e-commerce, the GoodRelations ontology [62] provides a standard vocabulary for describing products, services, and business entities, enabling more sophisticated search and comparison shopping applications. Similarly, in the geography domain, the GeoNames ontology [16] provides a global, hierarchical database of geographical features and names. At a broader, cross-domain level, schema.org [57] provides a widely adopted vocabulary for annotating web content, describing entities such as people, organizations, events, and creative works, and supporting the publication of structured data across diverse application areas. By providing a formal, computable representation of domain and cross-domain knowledge, these ontologies enable sophisticated data integration and analysis.

2.3.2 Ensuring Data Quality with SHACL

An ontology defines the vocabulary of a domain, but does not always enforce structural constraints on the data. The Shapes Constraint Language (SHACL) is a W3C recommendation designed specifically for the purpose of validating RDF graphs [78, 79]. SHACL allows users to define a set of conditions, or "shapes", that the data must conform to [34]. It is a language for describing the expected structure of a graph, rather than for inferring new knowledge [77].

A SHACL validation process takes two graphs as input: a data graph (the data to be validated) and a shapes graph (containing the SHACL rules). The output is a validation report that indicates whether the data conforms to the shapes and, if not, details the specific violations. For instance, a simple SHACL rule to enforce that every instance of a "Person" class must have exactly one value for the "ex:name" property can be expressed as follows:

LISTING 2.1: A simple SHACL shape enforcing a cardinality constraint.

```
:PersonShape
    a sh:NodeShape ;
    sh:targetClass ex:Person ;
    sh:property [
        sh:path ex:name ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
    ] .
```

SHACL shapes can specify a wide range of constraints, such as the cardinality constraints (shown in the above example), datatype constraints, value constraints, and logical constraints that combine other conditions using boolean operators. By defining and applying these shapes, SHACL provides a standardized and code-free mechanism for ensuring the quality, consistency, and integrity of the data within a KG [20].

2.4 Knowledge Graph Embeddings

While KGs are suitable for representing structured, symbolic knowledge, their discrete nature makes them difficult to use directly with ML models, which typically operate on continuous numeric data in vector spaces [154]. KGE methods bridge this gap by learning low-dimensional vector representations of the entities and relations in a KG [28]. These embeddings aim to preserve relevant structural and semantic information from the graph and can then be used as input features for downstream ML tasks.

In general, a KG can be represented as a set of triples $G = \{(h, r, t) \mid h, t \in E, r \in R\}$, where E denotes the set of entities and R denotes the set of relations. KGE methods learn a mapping from these symbolic entities and relations to a continuous vector space. Each entity $e \in E$ is mapped to an embedding vector $\mathbf{v}_e \in \mathbb{R}^d$, and each relation $r \in R$ is represented by an embedding vector, matrix, or transformation, depending on the embedding model.

KGE models can be broadly categorized according to how they learn and use these representations. Score-based methods evaluate the plausibility of individual triples using a scoring function. Walk-based methods generate sequences from the graph and learn embeddings from these sequences using techniques inspired by natural language processing. Deep learning-based methods, especially graph neural networks (GNNs), learn embeddings by aggregating information from neighboring nodes through message passing. These categories are discussed in the following paragraphs.

Score-based KGE Methods. Score-based KGE methods learn embeddings by assigning a plausibility score to triples. Given a triple (h, r, t), a scoring function $f(h, r, t)$ measures how likely the triple is to be true. The model is trained such that true triples, *i.e.*, triples present in the KG, receive higher scores than false or corrupted triples [128]. False triples are usually generated by replacing either the head or tail entity with another entity from the KG. A well-known score-based model is TransE, which represents relations as translations in the embedding space [21]. Given a triple (h, r, t), TransE assumes that the embedding of the head entity plus the embedding of the relation should be close to the embedding of the tail entity. Its scoring function is commonly defined as $f(h, r, t) = -\|\mathbf{h} + \mathbf{r} - \mathbf{t}\|_p$, where a higher score indicates a more plausible triple. For example, for the triple $(\mathit{Paris}, \mathit{capitalOf}, \mathit{France})$, TransE aims to learn embeddings such that $\mathbf{Paris} + \mathbf{capitalOf}$ is close to $\mathbf{France}$. Score-based methods are often directly optimized for link prediction because their training objective is closely related to distinguishing valid triples from invalid ones. However, they may face challenges when modeling complex relation patterns or when applied to very large and heterogeneous KGs [163].

Walk-based KGE Methods. Walk-based methods learn embeddings by converting the graph into sequences of entities and relations. These sequences are typically generated through random walks and are treated similarly to sentences in natural language processing. Algorithms such as Word2Vec are then used to learn embeddings from the resulting corpus of graph walks. RDF2Vec is a prominent example of a walk-based KGE method [124]. It first generates sequences from the RDF graph using graph walks or graph kernels and then applies Word2Vec-style training to learn vector representations of entities. For example, a walk such as

Paris → *locatedIn* → *France* → *memberOf* → *EuropeanUnion* provides contextual information about the entity *Paris*. By observing many such walks, RDF2Vec can capture broader neighborhood information rather than only individual triples. Walk-based methods are often suitable for sparse graphs because they can exploit the wider graph context around an entity [126].

Deep Learning-based KGE Methods. Deep learning-based KGE methods use neural architectures to learn representations from graph structure. A major group within this category consists of GNNs, which compute node representations by aggregating information from neighboring nodes. In multi-relational KGs, this aggregation must take different relation types into account. Relational Graph Convolutional Networks (R-GCNs) extend graph convolutional networks to multi-relational data by using relation-specific transformations during message passing [131]. Instead of scoring triples independently, R-GCNs update the representation of each entity based on messages received from its neighbors. For example, when computing the representation of an entity such as *Paris*, an R-GCN may aggregate information from neighboring entities connected through relations such as *locatedIn*, *capitalOf*, and *hasPopulation*. R-GCNs have been used for tasks such as link prediction and entity classification. GNN-based methods can capture rich neighborhood structure, but they can also be computationally expensive. In addition, they often require careful modeling choices when applied to heterogeneous or highly multi-relational graphs [157].

Literal-aware KGE Methods. A limitation of many KGE models is that they focus on the topological structure of the graph, while literal attributes associated with entities are either ignored or only indirectly represented. Literals include values such as numbers, dates, strings, and textual descriptions. These values may contain information that is essential for understanding entities and their relationships. For example, in a KG containing countries, numeric literals such as population, area, gross domestic product, or life expectancy may provide important signals for distinguishing entities or predicting relations. Similarly, in a KG about ML experiments, numeric literals such as the number of instances, number of features, or model performance can be important for comparing datasets and pipelines. To address this limitation, several literal-aware KGE approaches have been proposed. These methods differ mainly in how they incorporate literal information into the embedding process. Some approaches modify the KGE model architecture, while others transform the graph through preprocessing so that literals become part of the graph structure.

Architecture-based Literal Integration. LiteralE is an example of an architecture-based approach [82]. It extends existing KGE models by combining each entity embedding with a vector of its corresponding literal values through a learnable parametric function. For example, assume that an entity representing a city has literal values such as population, area, and elevation. LiteralE represents these values as a literal vector and combines it with the original entity embedding to produce a literal-enriched entity representation. This enriched representation is then used by a standard scoring function, such as DistMult or ComplEx. Thus, LiteralE keeps the general structure of the underlying KGE model but changes the entity representations by injecting literal information. KBLRN is another early approach that incorporates numerical information into knowledge base completion [53]. It combines latent features, relational features, and numerical features in an end-to-end learning framework. The numerical component captures differences between numeric attribute values of the head and tail entities. For example, for a relation such as

hasHigherGDPThan, the difference between the GDP values of two country entities can be informative for predicting whether the relation holds.

Preprocessing-based Literal Integration. Preprocessing-based methods transform literals into additional graph structure before training. KGA is a representative example of this approach [152]. Instead of modifying the scoring function or embedding architecture, KGA discretizes numeric and temporal values into bins and adds these bins as new nodes in the graph. For example, if the entity *BarackObama* has the literal *birthYear* with value 1961, KGA may assign this value to a bin such as *birthYear_1960–1969* and connect the entity to that bin node. KGA can also connect neighboring bins to model continuity between close numeric values and can use hierarchical bins to represent different levels of granularity. The MKGA framework extends this preprocessing idea to multiple literal modalities [116]. It proposes universal preprocessing operators for transforming numerical, temporal, textual, and image literals into graph structures that can be processed by standard embedding methods. For numerical literals, MKGA also relies on a binning-based strategy. The advantage of this approach is that it is model-agnostic: after preprocessing, the transformed KG can be embedded using different KGE methods, including score-based and walk-based approaches.

2.5 The Democratization of Machine Learning

The democratization of ML refers to the process of making ML technologies and techniques more accessible to a broader audience, particularly to individuals who are not ML experts. This movement is driven by the recognition that in many fields, deep domain expertise is as critical as technical skill for building effective ML solutions. The plethora of ML algorithms and libraries, and the fragmented nature of ML pipelines and datasets might prevent domain experts from using ML for their problems. The core idea is to lower this barrier, enabling people in both academia and industry to build, discover, and execute ML pipelines and datasets without requiring specialized training or extensive programming knowledge.

2.5.1 Facilitating ML Pipeline Development

Early efforts to simplify the ML process focused on creating libraries such as scikit-learn in Python [110], which standardized interfaces for many common algorithms but still required programming expertise. Based on these libraries, the field of Automated Machine Learning (AutoML) [59] tries to automate the end-to-end process of applying ML to real-world problems. A central challenge that AutoML addresses is the Combined Algorithm Selection and Hyper-parameter optimization (CASH) problem [49, 80]. For any given dataset, a practitioner must not only choose a suitable learning algorithm from a large selection (*e.g.*, Support Vector Machine, Random Forest, Neural Network) but also tune the hyper-parameters associated with that algorithm. The CASH problem is to find the combination of algorithm and hyper-parameters that maximizes performance on the given dataset. Tools like Auto-WEKA [80] provide a concrete example of this automation. Auto-WEKA treats the CASH problem as a single, complex optimization problem over a structured and hierarchical search space. It leverages Bayesian optimization techniques, specifically using models like Tree-structured Parzen Estimators (TPE), to intelligently and efficiently search this joint space of algorithms and their hyper-parameters. By automatically exploring different pipeline configurations and learning from the results,

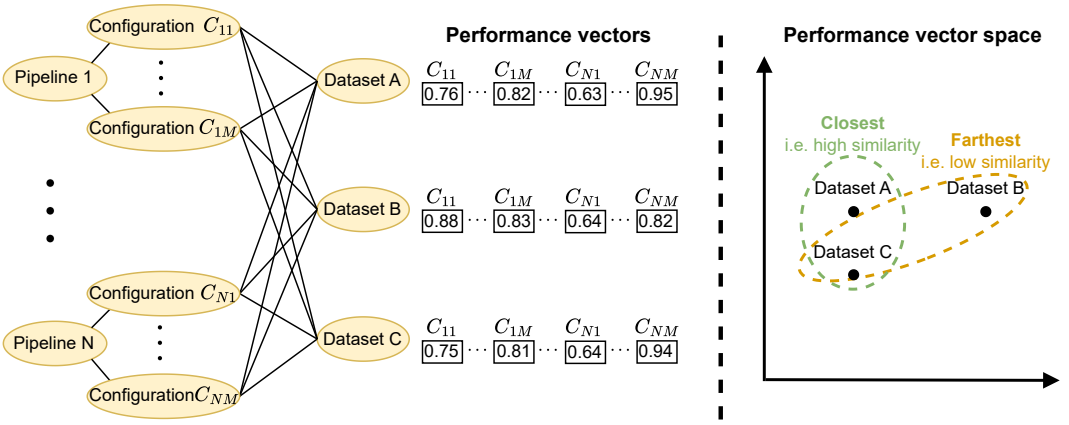


FIGURE 2.4: An illustration of performance-based dataset similarity. It is calculated by representing each dataset's performance profile across a common set of pipeline configurations as a vector (left) and then measuring the distance between these vectors (right).

Auto-WEKA can discover a high-performing, customized ML pipeline for a given dataset with minimal human intervention, making advanced ML techniques accessible to users without requiring them to perform manual algorithm selection and tuning.

A step towards greater accessibility came with the development of GUI-based platforms that abstract away the underlying code. A prime example is the Konstanz Information Miner (KNIME) [13], an open-source platform for data analytics, reporting, and integration. KNIME's core paradigm is visual workflow programming. Users construct pipelines, or "workflows", by dragging nodes from a repository onto a canvas and connecting them to define the flow of data. Each node represents a specific, modular task, such as reading data from a file, filtering rows, training a predictive model, or visualizing results. By connecting the output of one node to the input of another, users can create arbitrarily complex data processing pipelines without writing a single line of code. This node-based, visual approach lowers the technical barrier, allowing users with strong analytical skills but limited programming experience to perform sophisticated data science tasks.

2.5.2 Identifying Effective Pipelines and Similar Datasets

A central goal of meta-learning is to make the process of building effective ML pipelines more efficient by learning from past experience [146]. In the context of ML pipelines, such experience is typically captured through experiment records that describe how pipeline configurations performed on different datasets. These records can support complementary tasks such as *PPE* and *DPSE*, which help prioritize promising pipelines and identify datasets with similar empirical behavior.

PPE is important because the space of possible ML pipeline configurations is often large. A pipeline configuration may include multiple preprocessing steps, a learning algorithm, and its hyper-parameter settings. Exhaustively training and evaluating many candidate configurations can therefore be computationally expensive. By estimating the likely performance of candidate pipelines in advance, meta-learning methods can help prioritize promising configurations and reduce the number of expensive evaluations required [47].

DPSE supports meta-learning by identifying datasets that exhibit similar empirical behavior under ML pipeline configurations. Such information can be used to

warm-start pipeline search, since pipelines that performed well on similar datasets may be promising candidates for a new dataset. It can also support benchmark construction by helping select relevant datasets for testing and comparing pipeline configurations. In performance-based similarity, datasets are represented by performance vectors over a common set of pipeline configurations, and distances between these vectors are used to estimate how similarly the datasets behave [87], as illustrated in Figure 2.4. This goes beyond simply separating easy datasets from difficult ones. For example, two datasets may have similar average performance but favor different pipelines, while datasets with different overall difficulty may still be similar if the pipelines have a similar performance ordering on both datasets.

Together, *PPE* and *DPSE* provide complementary perspectives on how prior experiment data can support meta-learning. While *PPE* focuses on estimating the suitability of a given pipeline configuration for a target dataset, *DPSE* focuses on identifying datasets with similar empirical behavior. Both tasks can help reduce the cost of experimentation by guiding the selection of promising pipelines and relevant benchmark datasets.

Chapter 3

Challenges in ML Democratization

The democratization of ML aims to make its techniques accessible to a broader audience, especially domain experts who are not ML specialists. This chapter begins with an overview of the general obstacles in the field before focusing on the specific gaps this thesis aims to address. We first survey the landscape of existing no-code and low-code ML tools, which represent the current state of the art in simplifying pipeline creation. We then analyze the limitations of existing semantic models for ML, the challenge of incorporating numeric literals in KGEs, and the untapped potential of KGs for meta-learning.

3.1 General Challenges in ML Democratization

The path to democratizing ML is impeded by several interconnected obstacles that span the entire lifecycle of an ML system: from data preparation and development to deployment and governance.

The Skills and Knowledge Gap. Developing effective ML systems demands a multidisciplinary skill set, including a deep understanding of statistical theory, algorithm design, data engineering, and software development [7]. This requirement makes it difficult for organizations and domain experts to leverage ML without hiring dedicated specialists. While no-code tools manage to lower the barrier between domain experts and ML, effective and responsible AI development still requires a foundational understanding of the underlying principles [134].

Data Management and Accessibility. ML is fundamentally data-driven, and managing the data lifecycle is one of the most important challenges in production ML [115]. This includes issues of data discovery, versioning, cleaning, and ensuring data quality. For many organizations, data is siloed, inconsistent, or not in a format that is readily usable for ML tasks. The process of preparing data for an ML model is one of the most time-consuming and labor-intensive parts of the ML workflow [63].

Lack of Transparency and Interpretability. Many ML models, particularly in deep learning, operate as "black boxes", making it difficult to understand how they arrive at a particular prediction. This lack of transparency is a major barrier to adoption, especially in high-stakes domains like healthcare and finance, where accountability and the ability to explain a decision are critical. If domain experts cannot understand or trust a model's reasoning, they may not adopt it in their practice [65]. This lack of trust is linked to the broader ethical challenge to ensure AI systems operate with fairness and dignity in diverse contexts [51].

Resource Constraints and Scalability. Training state-of-the-art ML models might require significant computational resources, including powerful GPUs and large amounts of storage. These resource requirements can be a prohibitive barrier for

small organizations, researchers, and individual practitioners, contributing to a divide between resource-rich and resource-constrained regions [160]. Furthermore, scaling ML systems from a prototype to a production environment introduces a new set of engineering challenges related to reliability, latency, and maintainability [7].

Risks of Widespread Access and Misuse. A primary goal of democratization is to broaden access. Unconstrained access to powerful AI tools can enable malicious use. For example, AI drug discovery tools could be repurposed to design novel toxins [145]. Similarly, open-sourcing powerful models, while beneficial for innovation and auditing, makes it difficult to enforce controls and prevent misuse, as demonstrated by instances where restricted models were quickly leaked and made public [136, 149]. This creates a tension between the goals of broad accessibility and public safety.

Algorithmic Bias and Societal Injustice. AI systems can replicate or even amplify existing societal biases. It is widely documented that algorithmic bias can lead to unfair outcomes in critical areas such as hiring [121], loan appraisals [76], and criminal justice [8]. If AI development is dominated by a narrow demographic, the resulting products may perform disparately and inequitably for users from different backgrounds, reinforcing systemic injustices rather than alleviating them.

Equitable Distribution of Economic Benefits. There is a prevalent concern that if the profits from AI are concentrated only to a handful of companies, it could dramatically widen the socioeconomic divide between AI-leading and lagging nations and communities [134]. Leaders in the AI field like Microsoft's CTO argue that such a concentration of value would represent a failure of democratization [133], and DeepMind's CEO expresses that the benefits should ideally be distributed to all of humanity [134].

Concentration of Power and Governance Deficit. The immense cost of cutting-edge AI research has led a few large tech companies to dominate the field, creating a concentration of power. This results in a governance deficit: pivotal decisions about AI are made privately, driven by competitive pressures that encourage risk-taking over caution [114, 127]. Such a dynamic creates a danger that AI will advance faster than the safety protocols and regulations needed to manage it, leading to calls for greater restraint [134]. This lack of public oversight means AI's development may conflict with societal values.

These general challenges underscore the need for new frameworks and tools that not only simplify the process of building ML pipelines but also make them more transparent, fair, reusable, and discoverable. The following sections of this chapter will narrow the focus to the specific limitations of existing tools and the untapped potential of semantic technologies to address these issues.

3.2 The Landscape of No-Code ML Tools

The drive to make ML more accessible has led to the development of platforms that abstract away the complexity of programming. Table 3.1 compares representative free open-source tools according to six criteria. *Python implementation* denotes whether the tool provides a Python-based implementation or interface that supports integration with the PyData ecosystem. *Semantic pipeline validation* denotes whether the tool can check pipeline correctness using semantic constraints, such as type compatibility or operator constraints. *Semantic pipeline creation and execution* denotes whether pipelines can be created, represented, validated, and executed from

TABLE 3.1: Comparison of existing free open-source tools for no-code or low-code ML pipeline development. Checkmarks indicate explicit support for the corresponding feature. Missing features that are central to the proposed approach are denoted by '▲'.

Feature	Weka	RapidMiner	Orange	KNIME	Ludwig
Python Implementation			✓		✓
Semantic Pipeline Validation		✓			
Semantic Pipeline Creation & Execution ▲					
Batch Pipeline Generation & Execution ▲					
Open Pipeline Format		✓			✓
LOD Pipeline Format ▲					

a standardized, machine-interpretable semantic model rather than only from an internal workflow representation. *Batch pipeline generation and execution* denotes native support for programmatically generating and executing multiple pipelines. *Open pipeline format* denotes whether pipelines are stored in a documented, non-binary, reusable format, while *LOD pipeline format* denotes whether this representation follows LOD principles using RDF standards.

The dominant tool category is visual pipeline programming. Platforms such as KNIME [13], RapidMiner [101], and Orange [37] exemplify this approach, with some tools offering a *Python implementation* to support extensibility and integration. They provide a graphical canvas where users construct data processing pipelines by selecting nodes from a repository and connecting them in a directed graph. Each node represents a discrete unit of work, such as reading a CSV file, filtering rows, training a classifier, or generating a plot. The connections between nodes represent the flow of data through the pipeline. These systems typically store pipelines in tool-specific formats, rather than in an *open pipeline format*, which limits interoperability and reuse across platforms. The Weka workbench, while not strictly a workflow tool in the same vein, offers a similar GUI-based environment for applying a wide array of data mining algorithms, making it suitable for academic teaching and research.

A more recent group of solutions is declarative ML, with Ludwig [102] being a prominent example. Instead of constructing a visual workflow, the user specifies the pipeline in a high-level YAML configuration file, which constitutes an *open pipeline format* but remains tightly coupled to the Ludwig execution engine. The framework then handles code generation, model construction, and training.

As shown in Table 3.1, existing tools support some of the criteria considered in this comparison, but they do not provide the combination required by this thesis. Some platforms provide partial semantic pipeline validation, for example by checking type compatibility or operator constraints, but none support full semantic creation and execution of pipelines using a standardized semantic representation. Moreover, none of the examined tools provide native support for batch generation and execution of pipeline variants, nor do they represent pipelines using a LOD-based pipeline format that enables semantic interoperability and reuse. Although RapidMiner incorporates semantic annotations for ML pipelines [73], it does not leverage LOD principles to the extent proposed in this thesis.

3.3 The Challenge of Semantic Representation for ML

Recognizing the limitations of non-semantic formats, several research efforts have proposed ontologies for describing ML artifacts. Table 3.2 compares these efforts according to their *core focus* and *representation capabilities*. In the table, *ML experiment tracking* denotes support for describing experiment runs, datasets, algorithms, parameters, and results. *ML pipeline representation* denotes support for modeling a pipeline as an executable workflow composed of connected steps. *Software description* and *general scientific concepts* denote broader vocabularies for software artifacts and scientific entities, respectively. *Data flow/sequences* denotes representation of ordering or dependencies between pipeline steps. *Code artifacts* denotes support for representing software implementations. *Algorithmic technical details* denotes support for method-specific parameters, assumptions, or algorithm characteristics. *Broad ML concepts/tasks* denotes coverage of general ML task types such as classification, regression, clustering, and preprocessing. Finally, *hierarchical ML structure* denotes a task and method hierarchy designed to support both user-facing pipeline construction and automated execution.

A significant portion of existing work has concentrated on *ML experiment tracking*. Ontologies such as Exposé [147], ML-Schema [119], the MEX Vocabulary [42], and PROV-ML [137] provide vocabularies for describing the setup and results of an ML experiment. They are valuable for provenance, allowing researchers to document which algorithm was run on which dataset with what parameters to produce a given result. However, their core focus is on the experiment as a whole, and they generally lack the granularity required for *ML pipeline representation*, in particular the explicit modeling of internal *data flow and sequences* between pipeline steps.

Other ontologies have a broader, more general scope. The Software Ontology (SWO) [97] and the Semanticscience Integrated Ontology (SIO) [40], for instance, are designed for *software description* [2] and *general scientific concepts*, respectively. While these ontologies can capture *code artifacts* and abstract scientific entities, they are not specifically tailored to the ML domain and therefore lack the specialized concepts needed to describe ML pipelines in detail.

Closer to the needs of ML pipeline modeling are ontologies developed for the *data mining* domain, such as OntoDM [108] and the Data Mining OPtimization ontology (DMOP) [72]. These models are more aligned with the notion of a pipeline, supporting the representation of algorithmic *technical details* and, in some cases, explicit *data flow and sequences*. DMOP further introduces abstractions for *broad ML concepts and tasks*. However, these ontologies do not provide the *hierarchical ML structure* required to model ML pipelines in a way that is both intuitive for user-facing applications and sufficiently detailed to support automated execution.

In summary, while existing ontologies can describe individual aspects of the ML ecosystem, none provide an end-to-end semantic model for ML pipelines that combines explicit data flow, technical algorithm details, and a hierarchical structure designed for executability. This thesis addresses this gap by proposing a semantic, executable representation of ML pipelines.

3.4 The Challenge of Incorporating Numeric Literals in KGEs

The numeric nature of most hyper-parameters, included in past experiment data, creates a challenge for effectively incorporating them into the KG learning process. Specifically, within the field of KGEs for link prediction, the effective incorporation

TABLE 3.2: Comparison of existing KG schemata and ontologies relevant to ML. Checkmarks indicate explicit support for the corresponding focus area or representation capability. Missing features that are central to the proposed representation are denoted by '▲'.

Feature	Exposé	ML-Schema	MEX Vocab.	PROV-ML	SWO	SIO	OntoDM	DMOP
Core Focus								
ML Experiment Tracking	✓	✓	✓	✓				
ML Pipeline Representation ▲								
Software Description					✓			
General Scientific Concepts						✓		
Data Mining							✓	✓
Representation Capabilities								
Data Flow/Sequences			✓	✓				✓
Code Artifacts					✓	✓		
Algo. Technical Details							✓	✓
Broad ML Concepts/Tasks								✓
Hierarchical ML Structure ▲								

of literal values remains a challenge. Traditional KGE models like TransE [22] tend to ignore the quantitative information encoded in entity attributes, failing to capture the numeric patterns that often exist between the attributes of connected entities.

While literal-aware KGE models have been developed, they have some limitations. Some methods augment entity embeddings but not the relations themselves (*e.g.*, LiteralE [83]), rely on preprocessing steps (*e.g.*, KGA [153]), or struggle with the sparse literal data common in real-world KGs (*e.g.*, KBLRN [53]).

These methods ignore the potential patterns between the numeric attributes of head and tail entities. Thus, there is the opportunity to integrate head and tail numeric information in a learnable manner, enabling a KGE model to capture underlying numeric relationships. Since the relation is the connection point between head and tail entities, the challenge lies in using the relation's embedding as a vessel for this information fusion.

This challenge extends to downstream tasks like node classification. In this domain, the performance of triple-scoring KGE methods (*e.g.*, TransE) is not thoroughly examined. Also, numeric literals are often either excluded or treated as conventional nodes [125, 132, 26]. Recent work has focused on KG preprocessing methods for literal-aware node classification [117]. So, there is a need for further comparison with methods that integrate literals with the embeddings.

3.5 The Untapped Potential of KGs for Meta-learning

Meta-features provide a compact and practical representation of datasets and are widely adopted in meta-learning. However, they treat datasets as isolated statistical objects, detached from the pipelines that have been evaluated on them. In doing so, they abstract away the relational and structural information embedded in past experiment records.

In practice, performance behavior is not determined solely by intrinsic dataset properties, but by the interaction between datasets and pipeline configurations. These interactions are reflected in past experiment logs, which encode which pipelines were evaluated, how they were structured, which hyper-parameters were used, and how they performed. Conventional meta-feature-based approaches incorporate only a small fraction of this information, typically through landmarkers, and require overlapping pipeline evaluations to make datasets comparable.

The core limitation, therefore, is the absence of a representation that jointly models datasets and pipeline configurations as first-class entities. A flat feature vector representation cannot naturally encode heterogeneous elements such as pipeline components, hyper-parameters, execution structure, and dataset metadata within a unified space.

KGs offer a natural way to address this limitation. By representing datasets, pipeline configurations, and their relationships within a unified relational structure, KGs preserve the topology of dataset – pipeline interactions. Learning embeddings over such graphs enables the construction of pipeline-agnostic meta-models for performance estimation, while simultaneously supporting performance-based dataset retrieval without requiring identical evaluation sets.

Consequently, this thesis focuses on improving two meta-learning tasks: pipeline performance estimation and performance-based dataset similarity estimation. It addresses the challenge of representing past ML experiments in a form that supports knowledge transfer across datasets and pipeline configurations, including settings where not all datasets have been evaluated on the same set of pipelines.

3.6 Synthesizing the Challenges

This chapter identified several barriers that limit the democratization of ML, ranging from skills gaps and data management issues to transparency, governance, and misuse risks. Within this broader landscape, the chapter focused on four specific challenges addressed by this thesis.

First, existing no-code and low-code ML tools lower the barrier to pipeline creation, but they typically rely on tool-specific or non-semantic representations. As a result, they provide limited support for interoperable, reusable, and machine-interpretable pipeline descriptions. In particular, they do not offer a standardized LOD-based format that supports the semantic creation, validation, batch generation, and execution of ML pipelines.

Second, existing semantic models for ML capture important parts of the ML lifecycle, such as experiment tracking, provenance, software description, or data mining concepts. However, they do not provide a complete end-to-end representation of executable ML pipelines. In particular, they lack the combination of explicit data flow, technical algorithm details, and a hierarchical ML structure needed to support both user-facing pipeline construction and automated execution.

Third, the numeric information contained in KGs, such as hyper-parameters in ML experiment records, remains difficult to exploit effectively in KGE models. Existing literal-aware methods often focus on enriching entity embeddings, rely on preprocessing steps, or do not explicitly model numeric patterns between connected entities. This limits their ability to represent quantitative relationships that may be important for downstream tasks.

Fourth, conventional meta-learning approaches for performance estimation rely mainly on dataset meta-features and therefore treat datasets as isolated statistical

objects. This overlooks the relational structure of past experiment records, including which pipelines were evaluated on which datasets, how these pipelines were structured, which hyper-parameters were used, and how these choices affected performance. As a result, there is a need for representations that jointly model datasets and pipeline configurations as first-class entities and exploit dataset–pipeline interactions for more generalizable performance estimation.

Together, these challenges motivate the need for semantic and executable representations of ML pipelines that can support democratized pipeline construction while also enabling learning from past experiments. The following parts of this thesis address this need through Executable KGs, literal-aware KGEs, and KG-based meta-learning methods for performance estimation.

Part II

A Semantic Framework for ML Pipelines

Chapter 4

Executable Knowledge Graphs for Machine Learning

4.1 Introduction

The preceding chapters have established the foundational concepts of the Semantic Web and highlighted the critical challenges facing the democratization of ML. We identified a significant gap in the landscape of existing tools and ontologies: the lack of a standardized, semantic, and executable representation for ML pipelines. Conventional approaches, whether based on scripts, proprietary configuration files, or non-standard graph models, suffer from a lack of interoperability, transparency, and reusability. This chapter introduces the proposed solution to these challenges: the Executable KG (*ExeKG*).

The main idea of the *ExeKG* is to model an ML pipeline configuration as a KG using simple terms understandable to non-ML experts. The *ExeKG* is based on a specialized set of ontologies and conforms to a defined set of SHACL constraints. This way, the *ExeKG*'s executability can be validated. In this chapter, we will introduce the hierarchical suite of ontologies that provide the semantic backbone, describe the formal structure of an *ExeKG*, and discuss the SHACL constraints surrounding an *ExeKG*.

4.2 The *ExeKG* Ontological Foundation

For the *ExeKG* to be robust and interoperable, it must be grounded in a formal, shared vocabulary. This is achieved through a hierarchical suite of ontologies that define the concepts and relationships within the data science domain. This ontological structure, illustrated in Figure 4.1, is composed of a general, top-level Data Science ontology and several specialized, lower-level ontologies (statistics on Table 4.1) that are semi-automatically generated from popular data science libraries.

4.2.1 The Top-Level Data Science Ontology

The Data Science (DS) ontology (namespace: `ds`) is the upper-level ontology to which the rest refer. It represents the general concepts of Data Science. Specifically, it contains the upper-level *owl:Class* entities: *ds:Data*, *ds:Method*, and *ds:Task*. The *ds:Data* class includes all concepts related to data, such as *ds:DataSemantics*, which describes the meaning of data, and *ds:DataStructure*, which specifies the format of data (*e.g.*, a *ds:Numerical* can have the format *ds:Vector*). The *ds:Method* class includes algorithms and functions that operate on data, which are represented by the *ds:AtomicMethod* class. The *ds:Task* class has two subclasses: *ds:AtomicTask* and *ds:Pipeline*. The former defines the separate, executable tasks of a pipeline, while the latter represents the ordered series of tasks that constitutes the complete workflow.

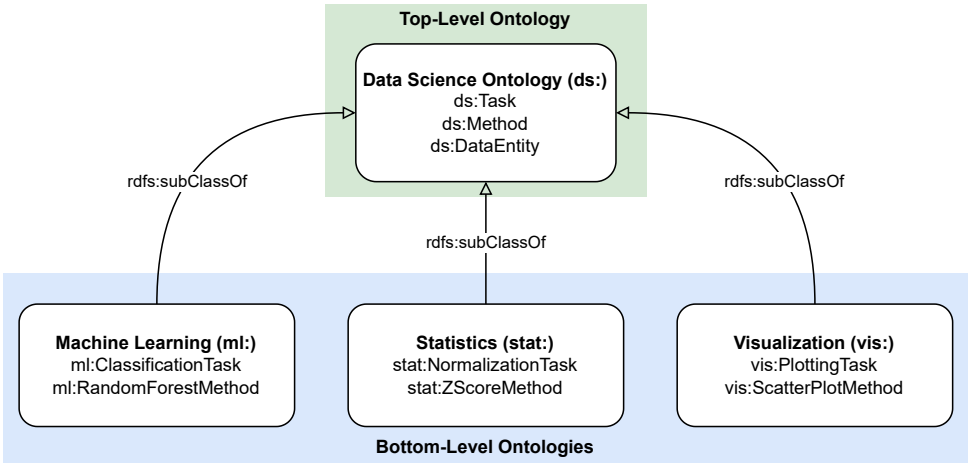


FIGURE 4.1: The hierarchical structure of the ontologies used in the *ExeKG*. Specialized lower-level ontologies inherit from and extend the core concepts of the top-level Data Science ontology.

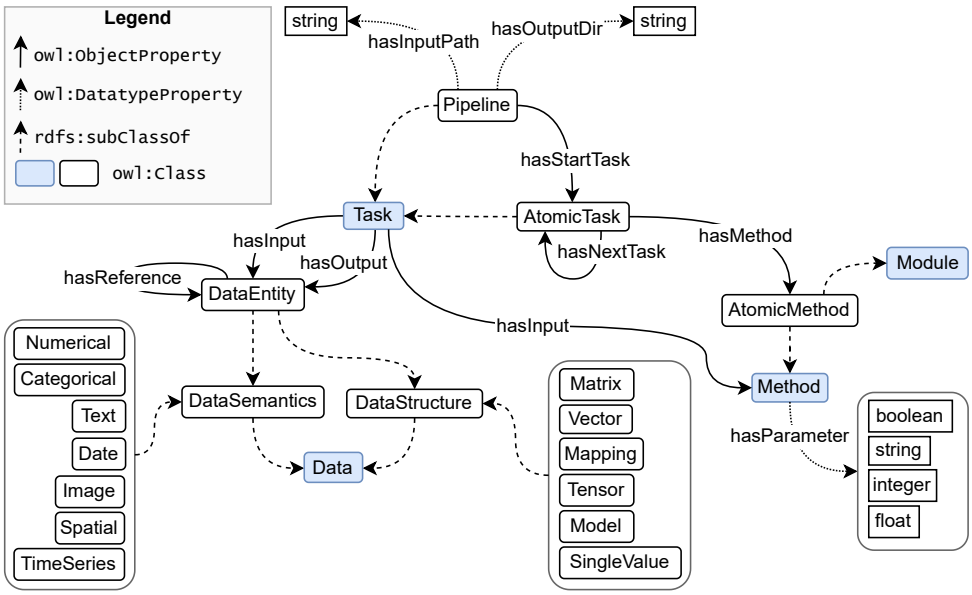


FIGURE 4.2: Data Science (DS) ontology

4.2.2 Lower-level Specialized Ontologies

Machine Learning Ontology. The ML ontology (namespace: `ml`) (Figure 4.3) is an example of a lower-level ontology. It contains entities of type *owl:Class* that are subclasses (*rdfs:subClassOf*) of *ds:AtomicTask* and *ds:AtomicMethod* which are defined in the DS ontology. Each lower-level subclass of *ds:AtomicTask* refers to a task type that can be solved using a specific group of methods (*i.e.*, *ds:AtomicMethods*). The boxes with dashed frames can be seen as templates for methods that implement specific algorithms suitable for solving the connected task types. For instance, tasks of type *ClassificationTrain* can be solved using a method that implements the kNN algorithm. Task types are categorized according to the common ML processes of data splitting, training and testing regression, classification, and clustering models, and calculating the models' performance.

Statistics and Visualization Ontologies. The other two KG ontologies (namespaces: `stats` and `visu` respectively) (Figure 4.4) follow the same structural principles as the ML ontology. The difference is the contents of the lower level subclasses which in this case consist of tools for statistics and data visualization. For the former, the task types are grouped based on the

Ontology	Classes	Object properties	Datatype properties	Triples
ML	337	40	351	4727
Statistics	28	8	20	269
Visualization	53	16	150	1093

TABLE 4.1: ExeKG ontology statistics

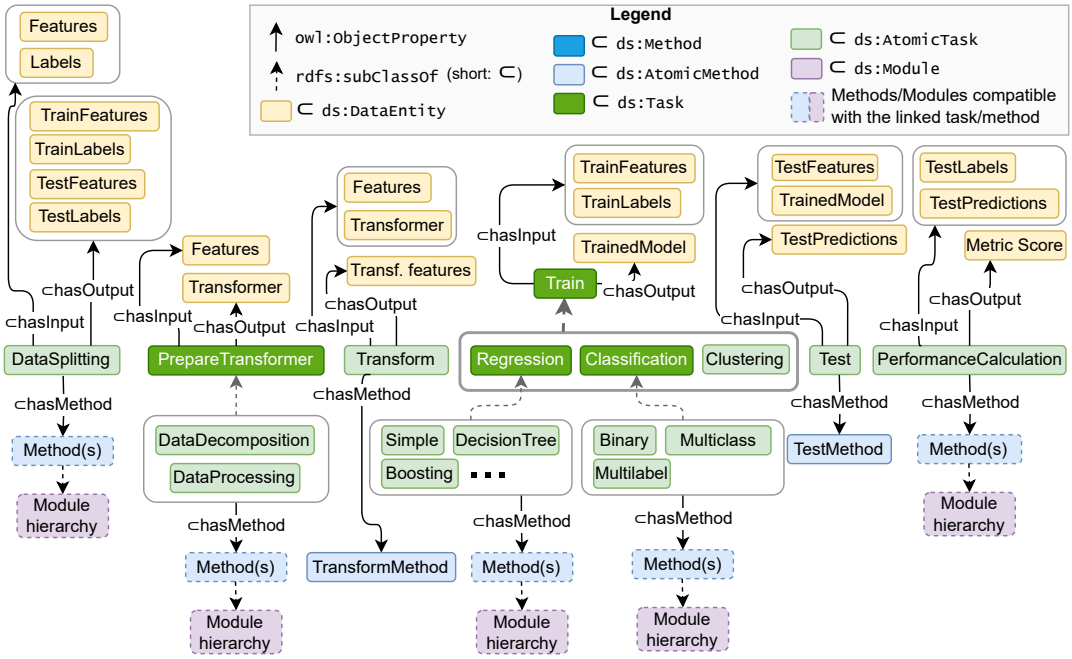


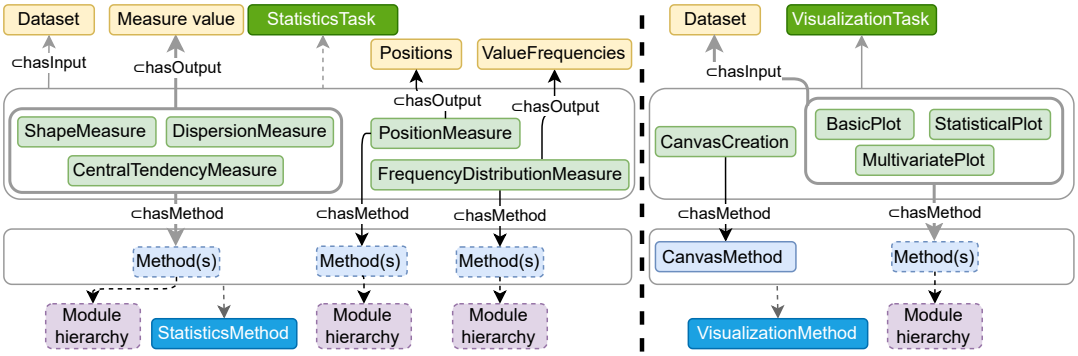
FIGURE 4.3: Machine Learning ontology

statistical measure to be calculated or the transformation to be applied to the data. Various descriptive statistics can be calculated such as central tendency measures (*e.g.*, median), position measures (*e.g.*, percentile), and frequency distributions (*e.g.*, grouped frequency distributions). Also, one of the statistics task types refers to preparing the data for ML by applying normalization, scaling, etc. As for data visualization, the task types are divided depending on whether the user wants to do basic (*e.g.*, scatter), statistical (*e.g.*, boxplot), or multivariate (*e.g.*, heatmap) plotting.

4.2.3 Semi-automatic Generation of Lower-level Ontologies

The lower-level ontologies were constructed using a hybrid approach that combines manually curated ontology cores with semi-automatic generation. In each domain-specific ontology (ML, statistics, and visualization), a hand-crafted core defines the task hierarchy and schema-level design, while the semi-automatic process extends this core by extracting API-level knowledge from widely used Python data science libraries, including `scikit-learn`, `matplotlib`, and `numpy`. This approach allows the ontologies to remain conceptually stable and curated at the task level, while systematically incorporating library-specific methods.

Extraction from Python Libraries. The process crawls the selected library modules and programmatically inspects their exported functions and classes, while applying curated whitelists and blacklists to exclude irrelevant modules and generic components (*e.g.*, private members, mixins, and base utility classes). This filtering step limits noise at the source and ensures that only public, user-facing functions and classes corresponding to concrete data processing, modeling, or visualization steps are considered. For each retained function or



(A) Statistics ontology (B) Visualization ontology

FIGURE 4.4: The two lower-level ontologies used in the *ExeKG*: (A) Statistics and (B) Visualization.

class, the process extracts callable signatures, parameter names, and available type information using Python introspection and structured docstring parsing. The extracted information is serialized into an intermediate representation.

Ontology Construction and Mapping. The process maps each extracted callable to an ontology subclass of *ds:AtomicMethod*, representing an atomic pipeline operation. The process then connects these atomic methods to a hierarchy of module concepts, represented by *ds:Module* classes, which are automatically generated to reflect the original library structure. The original documentation of each callable is preserved as `rdfs:comment` annotations, while method parameters are materialized as datatype properties attached to the corresponding *ds:AtomicMethod* subclasses. The ranges of these properties are derived from a controlled mapping between automatically extracted Python types and XML Schema datatypes, with conservative defaults used when type information is ambiguous. In addition, atomic methods are linked to existing *ds:AtomicTask* subclasses via an explicit method-to-task mapping. This conversion step ensures consistency by extending the manually curated core of each lower-level ontology. Semi-automatically generated method classes inherit from existing handcrafted classes in the curated cores and are connected to predefined task classes through newly introduced relations. Likewise, all generated datatype properties are derived from and constrained by existing property definitions. As a result, all automatically generated ontology axioms remain anchored to handcrafted semantic concepts.

Automatic SHACL Constraint Generation. Alongside the ontology axioms, the process automatically generates SHACL shapes that formalize task–method compatibility constraints implied by the ontology design. For each existing *ds:AtomicTask* subclass, a corresponding `sh:NodeShape` is generated, enforcing that every *ds:AtomicTask* instance is connected to exactly one *ds:AtomicMethod* instance via the appropriate property, using explicit `sh:minCount` and `sh:maxCount` cardinality constraints. Compatibility between tasks and methods is expressed through `sh:or` constraints that allow only the compatible *ds:AtomicMethod* subclasses for each task. In addition, SHACL shapes are generated for datatype properties representing method parameters, constraining their allowed XML Schema datatypes.

4.3 The Executable Knowledge Graph

An *ExeKG* is a concrete instantiation of the concepts defined in the ontological suite. It is an RDF graph that represents an end-to-end ML pipeline configuration, as illustrated in Figure 4.5. The structure of an *ExeKG* is defined by four key aspects. The overall workflow is modeled as a task sequence, an ordered chain of task *owl:Individuals* linked by the *ds:hasNextTask* property. Each task in the sequence has a task–method association, where the *ds:hasMethod* property links it to a method *owl:Individual* that defines its implementation. The data flow is explicitly modeled by connecting *ds:DataEntity owl:Individuals* to tasks via *ds:hasInput* and *ds:hasOutput* properties, creating a traceable data lineage. Finally, the

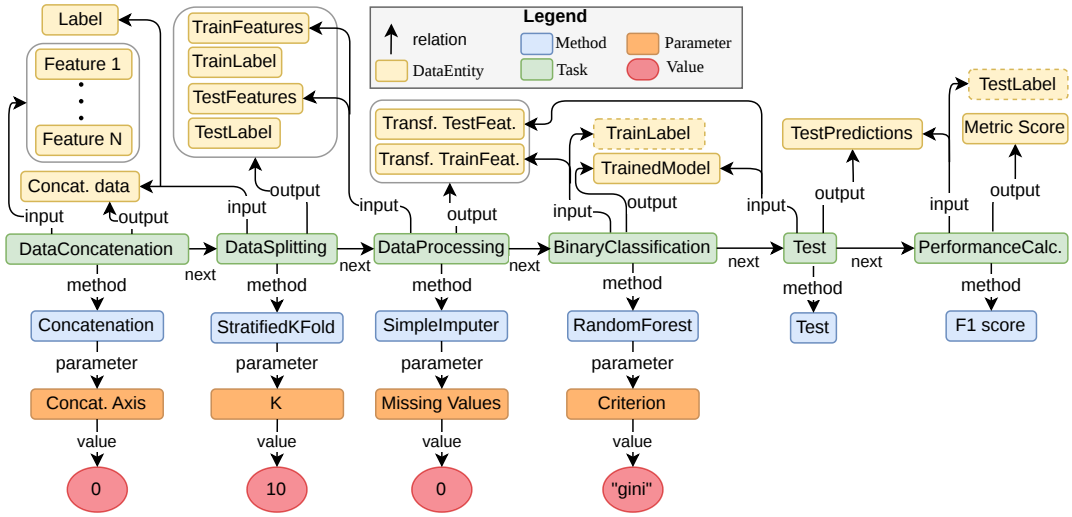


FIGURE 4.5: An example of a simple classification pipeline configuration represented as an *ExeKG*. The graph explicitly models the sequence of tasks, the methods used, the flow of data entities, and the corresponding hyper-parameters

parameterization of each method is represented by attaching literal values to the method *owl:Individual*, thereby specifying its hyper-parameters.

ExeKGs are designed to be executable. This is accomplished by an execution engine that translates the semantic representation into code. We implement this engine in our *ExeKGLib* Python library detailed in Chapter 5.

4.3.1 Executability Rules

We use SHACL constraints to ensure the executability of an *ExeKG*. The initial SHACL shape graph was generated using `sheXer` Python library [45], which can automatically extract SHACL shape constraints for RDF graphs. At first, the generated shape graph was slightly modified to reflect the constraints inferred by the KG ontologies, *e.g.*, from `rdfs:range` property. With this shape graph as a baseline, we added special constraints required for our case. These additional constraints can be categorized according to the following three aspects of an *ExeKG*.

Pipeline Structure. To formulate a valid ML pipeline, a sequence of *ds:Task owl:Individuals* which invoke specific *ds:Method owl:Individuals* should be connected with each other. Some examples are: (a) Each *ds:AtomicTask owl:Individual* must be followed by at most one *ds:AtomicTask owl:Individual* via *ds:hasNextTask* (Constraint 4.1), (b) each *ds:AtomicTask owl:Individual* must have exactly one compatible *ds:AtomicMethod owl:Individual* connected to it (Constraint 4.2). In addition, the sequence of *ds:Task owl:Individuals* of specific *ds:AtomicTask* subclasses should be in a particular order. For instance, before any *visu:PlotTask owl:Individual*, a *visu:CanvasTask owl:Individual* has to be added to the pipeline because it defines the grid layout for the plots.

LISTING 4.1: Example constraint 1: Each pipeline task must be followed by at most one task.

```
:AtomicTask a sh:NodeShape ;
    sh:property [ a sh:PropertyShape ;
            sh:maxCount 1 ;
            sh:path ds:hasNextTask ] ;
    sh:targetClass ds:AtomicTask .
```

LISTING 4.2: Example constraint 2: Each pipeline task must be linked to a compatible method.

```
:PerformanceCalculationTaskMethodShape a sh:NodeShape ;
    sh:targetClass ml:PerformanceCalculation ;
```

```
    sh:property [
        sh:path ml:hasPerformanceCalculationMethod ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:or (
            [ sh:class ml:AccuracyScoreMethod ]
            [ sh:class ml:F1ScoreMethod ]
            ...
            [ sh:class ml:PrecisionScoreMethod ]
            [ sh:class ml:R2ScoreMethod ]
        ) ;
        sh:message "Tasks of type PerformanceCalculation must be
            connected with exactly one compatible atomic method." ;
    ] .
```

Data Entities. The tasks of a pipeline are related to *ds:DataEntity owl:Individuals* that represent their inputs and outputs, which are constrained based on the *ds:Task owl:Individuals*. In particular, the type of data represented by these *ds:DataEntity owl:Individuals* is determined using the subclasses of *ds:DataStructure owl:Class* as shown in Figure 4.2. This allows for enforcing SHACL constraints such as: The output of each *ml:Train owl:Individual* must be a *ds:DataEntity owl:Individual* (representative of the trained ML model) that is connected to the *ds:SingleValue* subclass of *ds:DataStructure*. Besides, the number of *ds:DataEntity owl:Individuals* for some *ds:Task owl:Individuals* is also constrained. For instance, each *ml:Train owl:Individual* must have at least 2 inputs (representing data for features and labels) and only 1 output (Constraint 4.3). In this case, the third input is optional as it is used only by *ml:TrainMethod owl:Individuals* that correspond to ensemble models or hyper-parameter tuners.

LISTING 4.3: Example constraint 3: Each training task must have a single value data entity as an output.

```
:TrainTaskInputOutputShape a sh:NodeShape ;
    sh:targetClass ml:Train ;
    # input
    sh:property [
        sh:path ml:hasTrainInput ;
        sh:qualifiedMinCount 1 ;
        sh:qualifiedMaxCount 1 ;
        sh:qualifiedValueShape [
            sh:class ml:DataInTrainX ;
        ] ;
        sh:message "Tasks of type Train must be connected with exactly
            one input of type DataInTrainX." ;
    ] ;
    sh:property [
        sh:path ml:hasTrainInput ;
        sh:qualifiedMinCount 1 ;
        sh:qualifiedMaxCount 1 ;
        sh:qualifiedValueShape [
            sh:class ml:DataInTrainY ;
        ] ;
        sh:message "Tasks of type Train must be connected with exactly
            one input of type DataInTrainY." ;
    ] ;
    sh:property [
        sh:path ml:hasTrainInput ;
        sh:qualifiedMaxCount 1 ;
        sh:qualifiedValueShape [
            sh:class ml:InputModelAsMethod ;
        ] ;
        sh:message "Tasks of type Train must be connected with at most
            one input of type InputModelAsMethod." ;
    ] ;
```

```
    # output
    sh:property [
        sh:path ml:hasTrainOutput ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:class ml:DataOutTrainModel ;
        sh:message "Tasks of type Train must be connected with exactly
            one output of type DataOutTrainModel." ;
    ] .
```

Method Parameter Values. Method parameter values are represented as literals in an executable KG. SHACL constraints are used to validate the types of these literal values based on the standard XSD. As an example, *ds:AtomicMethod owl:Individuals* that represent MLP classifiers must have at most one integer literal for 'batch size' (Constraint 4.4).

LISTING 4.4: Example constraint 4: The 'batch size' parameter for the MLP classifier must be an integer.

```
:MLPClassifierMethodParameterShape a sh:NodeShape;
    sh:targetClass ml:MLPClassifierMethod ;

    sh:property [
        sh:path ml:hasParamBatchSize ;
        sh:maxCount 1 ;
        sh:or (
            [ sh:datatype xsd:int ]
        ) ;
        sh:message "Method MLPClassifierMethod must have at most one
            compatible value for parameter hasParamBatchSize." ;
    ] .
```

4.4 Conclusion

This chapter has detailed the formal structure of *ExeKGs*. By grounding the representation of ML pipeline configurations in a hierarchical suite of ontologies and a set of SHACL constraints, the *ExeKG* data model is both semantically rich and executable. The structure of an *ExeKG*, with its simple and granular modeling of tasks, methods, data flow, and parameters, creates a representation of an ML pipeline configuration that is understandable to both humans and machines. This approach addresses the core challenges of interoperability, transparency, and reusability that were identified in Chapter 3. Having established this formal representation, the following chapter will describe the software tools that were developed to enable users to create, manipulate, and interact with *ExeKGs*.

Chapter 5

Tools for Democratizing ML: *ExeKGLib* and *ExeKGLab*

5.1 Introduction

The preceding chapter established the *ExeKG* as an ontology-driven representation for ML pipeline configurations, designed to address the challenges of interoperability and transparency. The practical utility of such a semantic representation depends upon the availability of tools that enable users to create, execute, and manage these artifacts. This chapter introduces the two software contributions built for this purpose.

As analyzed in Chapter 3, the need for such tools is driven by the complexity and time-intensive nature of pipeline development. This challenge particularly affects domain experts in fields like smart manufacturing, who possess specialized domain knowledge but often lack a deep background in programming and ML. While existing solutions, such as AutoML systems and no-code platforms like KNIME [13], aim to simplify this process, they often come with trade-offs between automation and customizability, and do not leverage LOD to ensure the interoperability and reusability of the pipeline configurations they produce.

To fill this gap, this chapter presents a two-part software solution. The first component is *ExeKGLib*, an open-source Python library that serves as the programmatic engine for the framework. It provides developers with the interface to create, validate, and execute *ExeKGs*. By utilizing a hierarchical suite of ontologies and SHACL constraints, *ExeKGLib* ensures that all generated pipeline configurations are both semantically coherent and verifiably executable.

Recognizing that ML democratization extends beyond developers, the second component is *ExeKGLab* (Fig. 5.4), a GUI built on top of *ExeKGLib*. *ExeKGLab* offers a no-code, visual environment for pipeline construction, further facilitating entry to ML by incorporating an LLM-powered assistant to guide users through the process.

The effectiveness of these tools is evaluated in the remainder of this chapter through (i) an automated benchmark-based analysis of recommendation quality and (ii) a controlled user study assessing usability and task performance across alternative workflows.

5.2 *ExeKGLib*: The Semantic Execution Engine

ExeKGLib is a Python library that implements the logic of manipulating *ExeKGs*. It handles the creation of ML pipeline configurations as *ExeKGs*, their validation against SHACL shapes, and their translation into executable Python code.

The practicality of this approach is illustrated in Fig. 5.1, which compares a conventional setup for a classification pipeline against one built with *ExeKGLib*. In a typical scenario, a user must import and manage multiple libraries like `pandas`, `scikit-learn`, and `matplotlib`. In contrast, *ExeKGLib* provides a unified interface, allowing the user to discover and invoke the required functionalities from a single library. This simplifies the development process, making it faster and more intuitive by reducing the need to consult the extensive documentation of various external libraries.

By building upon standard PyData tools, `rdflib`, and the SHACL standard for validation, the library is capable of:

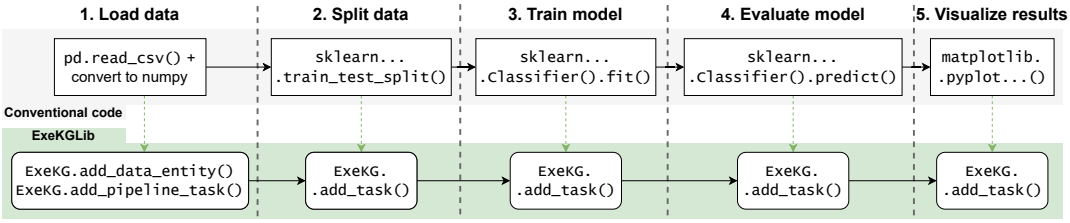


FIGURE 5.1: Comparison between conventional code and *ExeKGLib* for a classification problem. Fig. 5.3 details how this is achieved.

1. Generating executable ML pipeline configurations as KGs, covering data visualization, preprocessing and feature engineering, and model training and testing.
2. Validating the constructed KGs to guarantee their executability.
3. Transforming the KGs into Python scripts and executing them.

Furthermore, the use of semantic technologies provides a foundation for systematic extensibility. New ML algorithms, preprocessing operations, feature engineering methods, and visualization tasks can be added by extending the ontology with the corresponding semantic descriptions, without requiring changes to the pipeline execution logic.

5.2.1 Architecture

The architecture of *ExeKGLib* is based on a custom object-oriented class hierarchy that mirrors the concepts defined in the *ExeKG* ontologies. As shown in Figure 5.2, core classes such as *Entity*, *Data Entity*, and *Task* provide Python-native abstractions for ontology classes (*owl:Class*). The objects that *ExeKGLib* instantiates from these Python classes correspond to KG entities (*owl:Individual*).

ExeKGLib leverages standard Semantic Web technologies for its backend operations. It uses the `rdflib` library for creating and manipulating the underlying RDF graphs and the `pyshacl` library for validating the constructed *ExeKGs* against the predefined SHACL shapes detailed in Section 4.3.1.

***ExeKG* Class.** The primary class is *ExeKG*. It contains several fields, including custom *Namespaces* and two `rdflib` Graph objects designated as *Input KG* and *Output KG*. When constructing a pipeline configuration, the *Input KG* holds the base ontologies, and the *Output KG* is populated with the resulting *ExeKG*. A series of *KG creation methods* are used to add components, such as *owl:Individuals*, to the *Output KG*. For programmatic KG construction, higher-level methods like `ExeKG.add_task()` are available. During execution, the *Input KG* holds the *ExeKG* representing the pipeline configuration to be executed, and the *Output KG* is not utilized. The execution is handled by *KG execution methods*, which traverse the graph, translate each task and its properties into Python code, and then run it. This process is supported by the *Task* class and its specialized child classes.

***Entity* Class.** The *Entity* class is a base class for the *Data Entity* and *Task* classes, representing a generic entity within the KG. It stores fundamental information such as its *IRI*, *Name*, *Type*, *Parent Entity*, and *Namespace*. Both the KG creation and execution methods use this class to temporarily hold information about parsed KG entities.

***Task* Class.** The *Task* class extends the *Entity* class with fields specific to pipeline operations. It includes a *Next Task IRI* to link to the subsequent step, along with *Input Data Entity objects* and *Output Data Entity objects*. These are dictionaries that map a task's named inputs and outputs to their corresponding *Data Entity* objects. The inputs can be either the initial data for the pipeline configuration or the outputs from preceding tasks.

Each *Task* class contains an abstract Python method, `run_method()`, which is implemented in its child classes to execute a specific algorithm. These child classes correspond to tasks associated with various methods from domains like ML (*e.g.*, model training), statistics

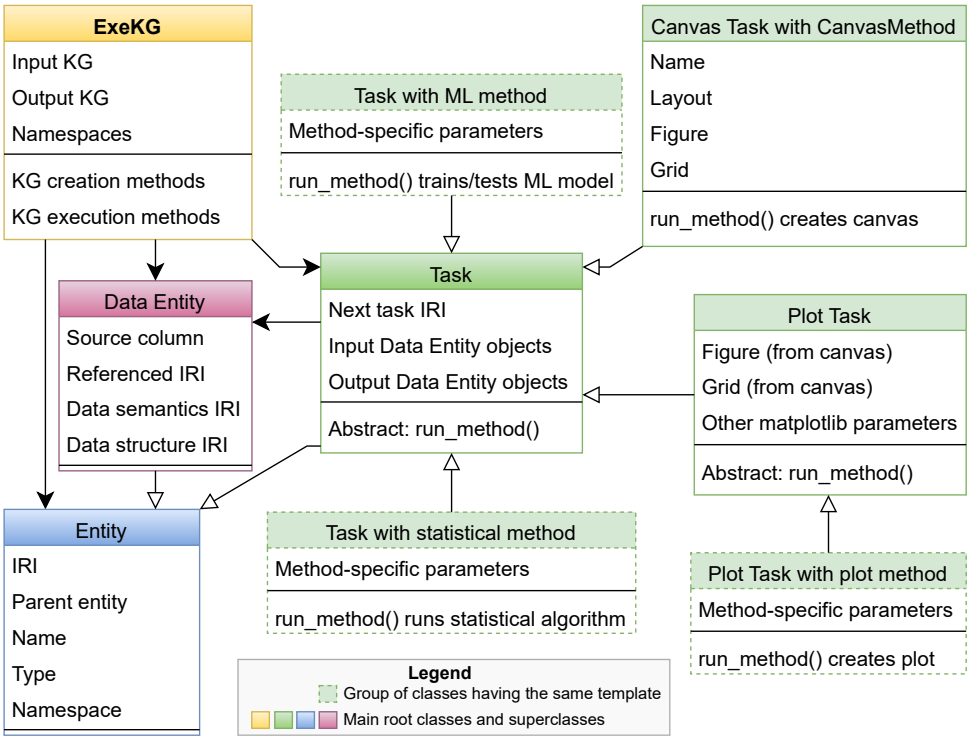


FIGURE 5.2: The object-oriented class hierarchy of *ExeKGLib*, which mirrors the structure of the *ExeKG* ontologies.

(*e.g.*, normalization), and data visualization (*e.g.*, plotting). For visualization tasks, a common parent class, *Plot Task*, manages canvas information, which is initialized by the *Canvas Task with Canvas Method* class.

***Data Entity* Class.** The *Data Entity* class represents the data that flows between the tasks of a pipeline configuration. It has a *Source column* field, which specifies a column from an input CSV file, and a *Referenced IRI* field, which links to an output from a previous task. For any given data entity, only one of these two fields will contain a value. The class also includes a *Data semantics IRI* and a *Data structure IRI* to define the data's meaning (*e.g.*, time series) and its format (*e.g.*, vector).

5.2.2 Programmatic Pipeline Construction

An *ExeKG* can be created programmatically, through the provided `Typer` CLI, or using an internal GUI that is not yet published. The following description focuses on the programmatic usage of the library to provide a clear understanding of its mechanisms.

The process of creating an *ExeKG*, outlined in Figure 5.3, begins with the user instantiating a Python object of the *ExeKG* class. This action initializes an empty KG and loads the required ontologies.

First, the user defines the data source of the pipeline configuration by providing the input CSV file path via the `ExeKG.add_pipeline_task()` method. With the source file specified, individual data columns can be added to the KG using `ExeKG.add_data_entity()`. Internally, *ExeKGLib* populates the KG with *ds:DataEntity owl:Individuals* that represent these columns, which can then be used as inputs for subsequent pipeline tasks.

Once the data entities are defined, the user can specify the sequence of operations by calling the `ExeKG.add_task()` function for each step in the pipeline configuration. For each call, the user selects a task to perform and a compatible method, which correspond to subclasses of *ds:Task* and *ds:Method* respectively. The user then decides on the input data entities for this operation, which can be either the initial CSV columns or the outputs of previously

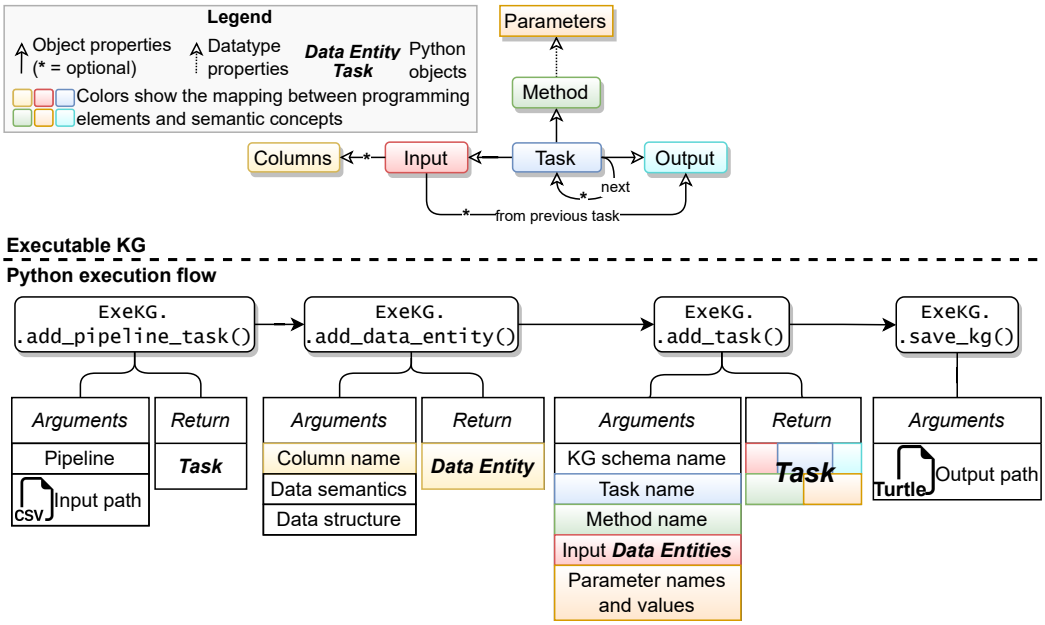


FIGURE 5.3: Executable KG construction. Linking semantic concepts (*e.g.*, *Task* KG entity) with programming elements (*e.g.*, *Task* Python object). Sample code in Fig. 5.1.

added tasks. To configure the operation, the user can also specify values for the parameters of the chosen method.

When `ExeKG.add_task()` is executed, *ExeKGLib* adds the *owl:Individuals* for the specified task and method to the KG. It then links the current task with its method, input data, datatype properties, and the next task in the sequence. The user does not need direct knowledge of the ontology structures, as the names for all required components are provided in the library's documentation.

Finally, with a call to `ExeKG.save_kg()`, *ExeKGLib* validates the created *ExeKG* according to the rules described in Section 4.3.1. By performing this validation we minimize the chance of user error and give the appropriate feedback to the user. After successful validation, *ExeKGLib* serializes the created *ExeKG* and saves it on the disk in Turtle.

5.2.3 Pipeline Execution Mechanism

As with *ExeKG* creation, the execution of an ML pipeline configuration represented as a KG can be performed in two ways: via code or CLI. Achieving this is also possible through the internal GUI, which has not been published yet. To execute a given KG, *ExeKGLib* parses the KG with the help of the above KG ontologies (Section The *ExeKG* Ontological Foundation). *ExeKGLib* begins the execution process by traversing the *ds:Task owl:Individuals* of the pipeline configuration in sequence, following the *ds:hasNextTask* object property. For each task, its properties are retrieved from the KG and dynamically mapped to a Python object. This dynamic mapping allows the library to be extended with new components without modifying the core execution code. Finally, the Python implementation corresponding to the task's linked *ds:Method owl:Individual* is invoked to perform the operation.

The internal process that *ExeKGLib* uses for executing a pipeline configuration can be divided into the below steps.

1. **KG and Dataset Loading**: The execution begins by parsing the *ExeKG* file of the pipeline configuration using the `rdflib` package, which constructs an *ExeKG* Python object. The *ds:Pipeline owl:Individual* is then retrieved from the graph. By parsing this individual, *ExeKGLib* finds the *ds:hasInputDataPath* data type property and uses the specified path to load the input CSV dataset.

TABLE 5.1: Indicative runtime overhead of the semantic execution layer for a four-task Random Forest pipeline.

Scenario	Setting	T_{semantic}	$T_{\text{execution}}$	Relative Overhead
Dataset scaling	500 rows	1.85 s	0.05 s	97.4%
Dataset scaling	50,000 rows	1.76 s	2.01 s	46.7%
Model scaling	20 trees	1.82 s	0.18 s	91.0%
Model scaling	300 trees	1.76 s	2.43 s	42.0%

2. **Task Parsing**: *ExeKGLib* traverses the pipeline configuration by following the *ds:hasNextTask* object property from one task to the next. For each *ds:Task owl:Individual* encountered, the library extracts its associated *ds:Method*, its input and output *ds:DataEntity owl:Individuals*, and the method's datatype properties. This information is then stored in a Python object for convenient access during execution.
3. **Task Execution**: The Python object created for each task contains an abstract method, `run_method()`, whose specific implementation is determined by the names of the parsed *ds:Task* and *ds:Method owl:Individuals*. *ExeKGLib* calls this method internally, providing it with two sets of arguments: the input data parsed from *ds:DataEntity owl:Individuals*, which has been resolved to either columns from the initial dataset or the outputs from a previous task, and the method's parameters parsed from datatype properties, which have been translated into Python primitive types.

Automatic Mapping of KG Components to Programming Components. As the *ExeKG* is traversed, each *ds:Task owl:Individual* is dynamically mapped to a corresponding Python class, and an object is instantiated for it (Algorithm 2). The properties of the *ds:Task* are then automatically assigned as fields of this object. For most properties, such as literal values or standard IRIs, this conversion is handled directly by Python's `setattr()` function. A specific process is required for the IRIs of *ds:DataEntity owl:Individuals* (Algorithm 1), which must be resolved to account for the *Referenced IRI* field that links data entities within the pipeline configuration. This dynamic mapping is a key feature, as it allows users to extend the library with new tasks and custom methods without needing to modify the underlying code for KG construction or execution.

Runtime Considerations. The semantic execution layer introduces runtime overhead compared to directly executing a hand-written Python script. This overhead mainly stems from loading the *ExeKG*, traversing the task sequence, resolving input and output data entities, parsing method parameters, and dynamically mapping KG components to Python objects. To quantify this effect, we measured the generation overhead (T_{semantic}) separately from the execution time of the underlying ML operations ($T_{\text{execution}}$). The relative overhead was computed as:

$$\text{Relative Overhead} = \frac{T_{\text{semantic}}}{T_{\text{semantic}} + T_{\text{execution}}} \times 100.$$

The measurements were conducted using a four-task classification pipeline consisting of data loading, train-test splitting, model training, and evaluation. The input data was a synthetic tabular classification dataset with two numerical features. As shown in Table 5.1, the semantic overhead remains almost constant, at around 1.8 s across the measured scenarios, while its relative contribution decreases as the dataset size and model complexity increase. For example, when scaling the dataset from 500 to 50,000 rows, the relative overhead decreases from 97.4% to 46.7%. Similarly, when increasing the Random Forest size from 20 to 300 trees, the relative overhead decreases from 91.0% to 42.0%. This indicates that the semantic layer behaves mainly as a fixed setup cost. It is therefore most noticeable for small pipelines and inexpensive operations, but becomes less significant for larger datasets, iterative training procedures, and more complex models. This trade-off is acceptable in the context of *ExeKGLib*, whose primary objective is not low-level execution speed, but the provision of semantically validated, interoperable, and reusable pipeline configurations.

5.2.4 Extending *ExeKGLib* with New Methods

A central design goal of *ExeKGLib* is to make the framework extensible beyond the currently supported set of ML, preprocessing, feature engineering, and visualization methods. This extensibility is achieved by separating the semantic specification of a pipeline component from its Python implementation. Adding a new algorithm therefore requires extending two layers: the bottom-level *ExeKG* schema and SHACL shapes, which describe and validate the new component, and the corresponding Python task implementation, which defines how the method is executed.

At the semantic level, a new method is added to the appropriate bottom-level KG schema, such as the ML, statistics, or visualization schema. If the method belongs to an existing task type, such as classification or normalization, it can be linked directly to that task by extending the corresponding task-to-method property and updating the associated SHACL shape. In this case, the existing task definitions, including their input and output constraints, can be reused. If the new method introduces a task that does not thematically fit under an existing task class, a new subclass of *ds:Task* or *ds:AtomicTask* is added, together with its task-to-method property, input and output data entities, and corresponding SHACL constraints.

New methods are represented as subclasses of *ds:AtomicMethod* and are placed under the appropriate method hierarchy. Their implementation location is also encoded semantically through module classes that mirror the Python module hierarchy. Method parameters are added as datatype properties, with their domains, ranges, and SHACL constraints. As a result, newly introduced methods become discoverable and validatable in the same way as existing methods. They can also be used by *ExeKGLab*, since the GUI derives the available tasks, methods, parameters, and compatibility constraints from the ontology.

At the implementation level, the developer extends the relevant subclass of *Task* in the *ExeKGLib* codebase. For example, when extending the ML schema, the corresponding implementation is added in the ML task classes. If the new method is associated with an existing task type, only the relevant `run_method()` implementation needs to be extended to call the Python function or module that implements the new algorithm. If a new standalone task type is introduced, a corresponding Python task class is added following the same structure as the existing task classes.

This design limits the amount of code that must be changed when new functionality is added. The general KG construction, parsing, validation, and execution mechanisms remain unchanged. The ontology and SHACL shapes define what the new component is, how it can be connected, and which parameter values are valid, while the task implementation defines how it is executed. Consequently, *ExeKGLib* can be incrementally extended with new algorithms and domain-specific operations while preserving semantic validation, executability, and interoperability.

5.3 *ExeKGLab*: A No-Code Interface for ML Pipelines

While *ExeKGLib* provides a tool for developers to interact with ML pipeline configurations, the goal of ML democratization requires a more accessible interface for non-programmers. *ExeKGLab* is a GUI-based tool designed for this purpose, with its functionality and structure guided by the *ExeKG* ontologies. It provides a visual, no-code environment for creating, editing, and executing ML pipeline configurations, all powered by *ExeKGLib* in the backend.

5.3.1 GUI Design and Features

The *ExeKGLab* interface, shown in Figure 5.4, is composed of two main areas: a sidebar and a canvas. The sidebar provides a palette of available components, including the variables of the loaded dataset and the available tasks, which are organized according to the *ExeKG* ontology hierarchy. The canvas serves as the workspace where users construct their pipeline configurations using a drag-and-drop methodology. Users can drag task nodes from the sidebar onto the canvas and connect them with edges to define the data flow between pipeline steps. Selecting a task node on the canvas reveals its associated methods and parameters, which can be configured through interactive widgets.

Algorithm 1: Converting a property value of a *ds:Task owl:Individual* to a field value of a programming class

Input: propertyValue: String, KG: Graph
Output: propertyValue as is *OR DataEntity* object

if *propertyValue is not an IRI* **then**
  return propertyValue;
dataEntityOwlClassIRI ← *KG.queryGetDataEntityType*(*propertyValue*);
if *dataEntityOwlClassIRI is empty* **then**
  `/* propertyValue is an IRI, but does not refer to a rdfs:subClassOf ds:DataEntity */`
  return propertyValue;
dataEntityIRI ← *propertyValue*;
dataEntityOwlClassObj ← *Entity*(*dataEntityOwlClassIRI*);
dataEntityObj ← *DataEntity*(*dataEntityIRI*, *dataEntityOwlClassObj*);
refDataEnIRI←*KG.queryRefIRI*(*dataEntityIRI*);
if *refDataEnIRI is empty* **then**
  `/* ds:DataEntity owl:Individual does not link to another one */`
  refDataEnIRI ← *dataEntityIRI*;
else
  dataEntityObj.setValue('*hasRef*', *refDataEnIRI*);
end
foreach *t* ∈ *{triples of KG, with subject IRI = refDataEnIRI}* **do**
  predicateLabel ← part of *t.predicateIRI* after '#';
  fieldName ← *camelCaseToSnakeCase*(*predicateLabel*);
  if *not dataEntityObj.hasField(fieldName)* **then**
    continue;
  `/* Recursion */`
  fieldValue ← *propertyValueToFieldValue*(*t.objectDataOrIRI*);
  dataEntityObj.setValue(*fieldName*, *fieldValue*);
end
return dataEntityObj;

The user can initiate the conversion of the visual graph into an *ExeKG* by pressing the "Run Pipeline" button. Before execution, *ExeKGLab* validates the generated *ExeKG* using the SHACL constraints defined for the *ExeKG* ontologies. This validation is used to identify modeling issues in the user-created pipeline, such as missing required inputs, invalid task connections, or incomplete parameter configurations. When such issues are detected, *ExeKGLab* provides feedback through the GUI by reporting the validation error and indicating the affected task node, connection, or parameter.

If the generated *ExeKG* passes validation, *ExeKGLab* sends it to the backend (*ExeKGLib*) for execution. The execution results are then displayed in the GUI.

5.3.2 LLM-Powered Pipeline Recommendation

To further assist users, especially those with limited ML expertise, *ExeKGLab* integrates an LLM-powered recommendation engine, shown in Figure 5.5. This feature allows users to express their goals in natural language during pipeline construction. The recommendation engine follows a prompt-based approach (Figure 5.6), which begins by dynamically constructing a structured prompt that includes the user's natural-language query, dataset-specific metadata, and ontology-derived constraints restricting recommendations to tasks and methods supported by *ExeKGLib*. The recommendation process is carried out in two stages. First, the LLM identifies an ordered sequence of suitable ML tasks that defines the high-level pipeline structure. Second, for each recommended task, the system retrieves the set of semantically compatible methods from the *ExeKG* ontology, and the LLM selects the most appropriate method based on the refined prompt context. By iteratively grounding intermediate LLM inputs in ontology classes and constraints, the system ensures that the resulting pipeline recommendations remain semantically coherent and executable.

Algorithm 2: Parsing a *ds:Task* from the KG and storing the relevant information in a programming object

```
   Input: taskIRI: String, KG: Graph
   Output: Task object
1  taskOwlClassIRI ← KG.queryGetTaskType(taskIRI);
2  taskOwlClassObj ← Task(taskOwlClassIRI);
3  taskObj ← Task(taskIRI, taskOwlClassObj);
4  linkedMethodIRI ← KG.queryGetLinkedMethodOfTask(taskIRI);
5  linkedMethodOwlClassIRI←KG.queryGetMethodType(linkedMethodIRI);
6  methodAsEntityObj ← Entity(linkedMethodIRI, linkedMethodOwlClassIRI);
7  childTaskClassName ← taskObj.type + methodAsEntityObj.type;
8  ChildTaskClass ← getChildTaskClass(childTaskClassName);
   /* ChildTaskClass implements the logic indicated by the ds:Method
      owl:Individual that is linked to the ds:Task owl:Individual      */
9  childTaskObj ← ChildTaskClass(taskIRI, taskOwlClassObj);
10 foreach t ∈ {triples of KG, with subject IRI = taskIRI} do
11     predicateLabel ← part of t.predicateIRI after '#';
12     fieldName ← camelCaseToSnakeCase(predicateLabel);
13     if not childTaskObj.hasField(fieldName) then
14         continue;
15     fieldValue ← propertyValueToFieldValue(t.objectDataOrIRI);
16     if fieldName = 'hasInput' OR fieldName = 'hasOutput' then
17         dataEntityList ← childTaskObj.getFieldVariable(fieldName);
18         dataEntityList.add(fieldValue);
19     else
20         childTaskObj.setValue(fieldName, fieldValue);
21     end
22 end
23 return childTaskObj
```

ExeKGLab's features aim to improve both the correctness of recommended pipelines and the accessibility of ML pipeline construction for non-expert users. These claims require systematic empirical validation. In particular, it is necessary to assess (i) to what extent the LLM-based recommendations align with established ML practice, and (ii) whether the combination of a no-code GUI and AI assistance leads to measurable improvements in user performance and experience compared to alternative workflows.

5.4 Evaluation Setup

This section describes the methodology used to evaluate the two tool contributions of this chapter: *ExeKGLib* as the execution backbone for ontology-grounded pipeline configurations, and *ExeKGLab* (including its LLM assistant) as the user-facing interface for constructing pipeline configurations. The evaluation combines an automated, offline comparison against established OpenML benchmarks with a controlled user study involving human participants. Together, these complementary evaluations enable assessment of both algorithmic recommendation quality and practical user-facing impact.

5.4.1 Automated Evaluation Setup for LLM Pipeline Recommendations

To assess the intrinsic quality of the LLM-generated pipeline recommendations independently of human interaction effects, we first conduct an automated, offline evaluation using OpenML benchmarks. This evaluation focuses on how closely the assistant's recommendations align with high-performing reference pipelines at both the method and task level.

Evaluation Dataset. The evaluation uses OpenML benchmark records that capture ML goals defined on datasets together with their corresponding pipeline executions, including dataset metadata, pipeline structure, and performance metrics such as F1 score. Each record corresponds to a single pipeline configuration. Since multiple pipeline configurations may be

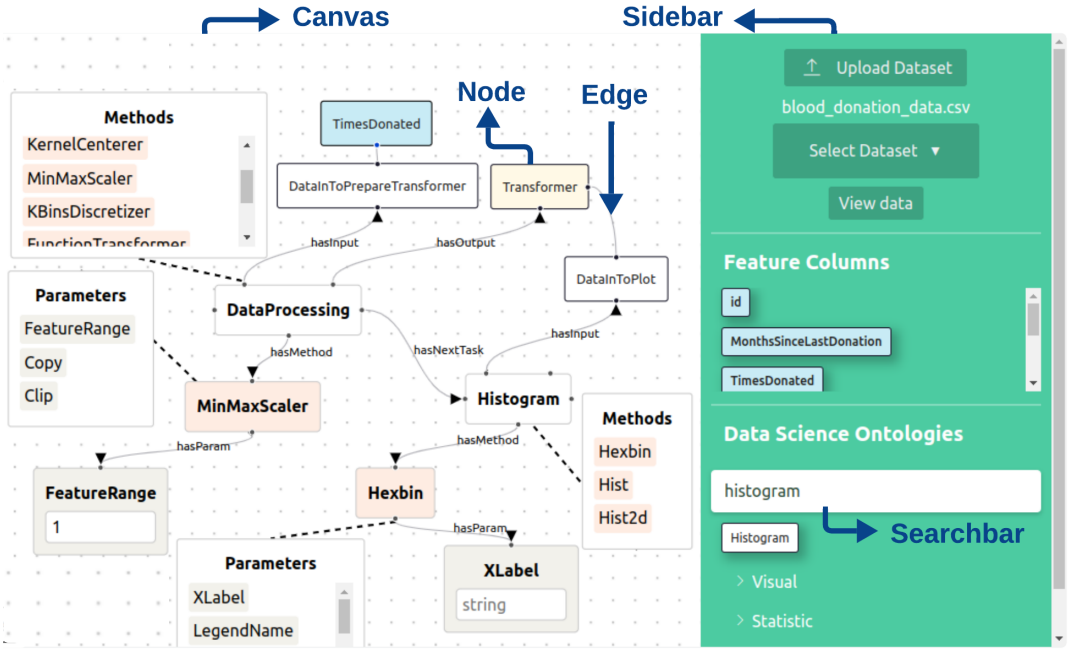


FIGURE 5.4: The graphical user interface of *ExeKGLab*, showing the sidebar with available components and the main canvas for constructing pipeline configurations.

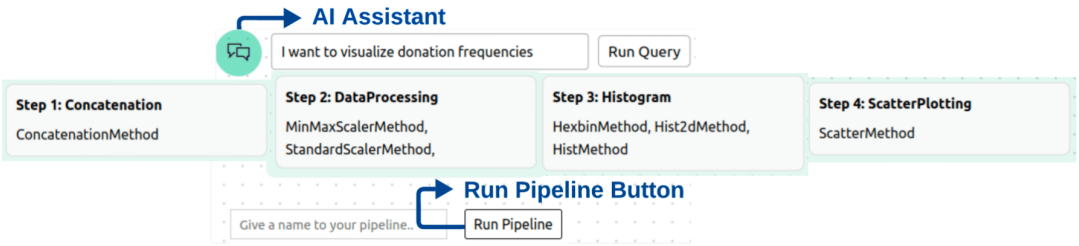


FIGURE 5.5: The LLM-powered AI Assistant in *ExeKGLab*, which provides pipeline recommendations based on natural language queries.

available for the same OpenML benchmark instance, we select the configuration with the highest F1 score and define its corresponding pipeline as the reference pipeline for that task. Across all evaluated datasets, this resulted in a total of 108 benchmark pipelines.

Evaluation Process. For each OpenML benchmark instance, a structured natural-language query is constructed from the ML goal and dataset metadata. This query is used to obtain the pipeline task sequence and to generate method recommendations, with up to five recommended methods per pipeline task. The resulting recommended pipelines are then compared against the reference pipeline for the same OpenML benchmark instance at both the method and pipeline-task levels. Details on the LLM and prompt template are provided in Appendix A.

Evaluation Metrics. Recommendation quality is quantified using two complementary metrics: *Hit Rate* and the *Jaccard Coefficient*, capturing method-level accuracy and task-level structural similarity, respectively. The Hit Rate measures the extent to which methods used in the benchmark pipeline are recommended by the LLM. For a benchmark method m_i, a hit is defined as:

$$\text{Hit}(m_i) = \begin{cases} 1, & \text{if } m_i \in M_{\text{recommended}} \\ 0, & \text{otherwise} \end{cases}$$

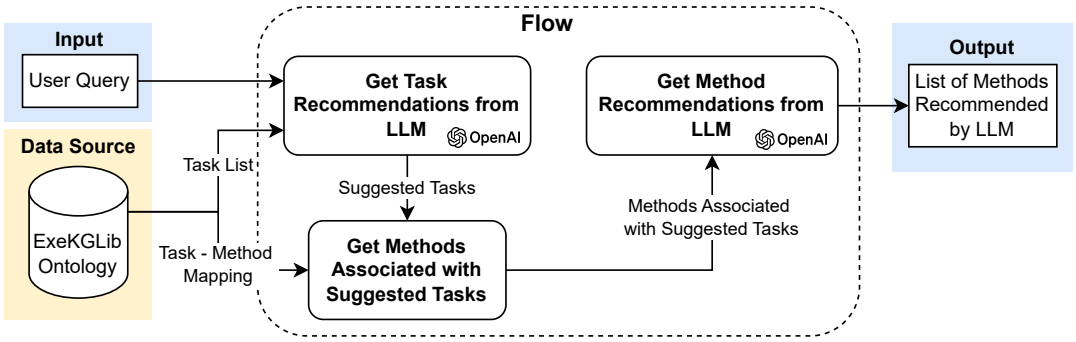


FIGURE 5.6: LLM-Powered Recommendations Flow

where $M_{\text{recommended}}$ denotes the set of methods in the LLM-recommended pipeline. The overall Hit Rate is computed as Hit Rate $= \frac{1}{N}\sum_{i=1}^{N} \text{Hit}(m_i)$, with N denoting the total number of methods in the benchmark pipeline. In contrast, the Jaccard Coefficient measures task-level similarity between the LLM-generated and reference pipelines by comparing their task sets A and B, defined as $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$. More information about the choice of these metrics is provided in Appendix A.2. Together, these metrics provide a quantitative assessment of how accurately the assistant selects relevant methods and how closely the overall structure of the recommended pipelines aligns with high-performing OpenML reference pipelines, prior to evaluating their impact on user experience.

5.4.2 User Study Design and Procedure

The user study was designed to evaluate the usability and effectiveness of *ExeKGLab* as a platform for ML pipeline construction, and to quantify the incremental benefits of (i) integrating an LLM-based assistant and (ii) providing a no-code GUI compared to a conventional code-based workflow. To this end, three system variants were evaluated: *System A*, *ExeKGLab* with the LLM-based AI assistant enabled, allowing users to request and apply pipeline recommendations through a chat interface; *System B*, *ExeKGLab* without the AI assistant, in which users constructed pipelines manually using the no-code GUI; and *System C*, a conventional code-based workflow, where participants implemented equivalent pipelines using standard Python ML libraries. Results are reported jointly for all three variants, enabling direct and consistent comparisons between *System A*, *System B*, and *System C*.

Participants and Study Design. The study involved 24 participants with varying levels of ML expertise. It followed a within-subject design in which each participant evaluated all three system variants: *System A* (ExeKGLab with the LLM-based assistant), *System B* (ExeKGLab without the assistant), and *System C* (a conventional code-based workflow). To mitigate learning and order effects, the six possible system orders were counterbalanced, with four participants assigned to each order. The order of the two tasks was also counterbalanced.

Dataset. All tasks were performed using a publicly available blood donation dataset (OpenML ID: 1464), originally published in ARFF format and converted to CSV for the study. The dataset consists of 748 instances describing individual blood donors and includes four numerical input features: *MonthsSinceLastDonation*, *TimesDonated*, *BloodAmountDonated*, and *MonthsSinceFirstDonation*. The target binary variable indicates whether an individual donated blood in March 2007. The dataset was selected due to its moderate complexity and suitability for both classification and feature engineering tasks, allowing the study to focus on tool usability rather than domain-specific modeling challenges.

Tasks. Participants completed two tasks designed to reflect common ML pipeline construction activities while remaining accessible to users with varying levels of ML expertise. Task complexity was intentionally kept low to moderate in order to emphasize system usability and interaction quality rather than advanced modeling knowledge.

1. **Task 1: Classification.** The first task focused on end-to-end supervised learning pipeline construction. Participants were asked to create a classification pipeline to predict the

target variable of the provided dataset. The task required participants to perform basic data exploration, define an appropriate train-validation-test split with stratification to preserve class balance, select and configure a classification model, and execute the resulting pipeline. Depending on the assigned system condition, pipeline construction was supported either through LLM-based recommendations (*System A*), manual configuration via the no-code GUI (*System B*), or a conventional code-based implementation (*System C*). Successful completion required producing a valid, executable pipeline and inspecting its output.

2. **Task 2: Feature Engineering and Visualization.** The second task emphasized feature-level analysis and data preprocessing. Participants were asked to analyze a numerical feature (*MonthsSinceLastDonation*) by inspecting its distribution, apply a suitable transformation (*e.g.*, normalization), and compare the original and transformed feature distributions through visualization. This task assessed the participants' ability to reason about data transformations, construct simple preprocessing pipelines, and interpret the effects of feature engineering steps within each system variant.

Evaluation Metrics. The evaluation combined qualitative and quantitative metrics to capture both subjective user experience and objective task performance. The same set of metrics was applied consistently across all system variants to enable direct comparison, with the exception of assistant-specific measures, which are only applicable to *System A*.

- **Qualitative Metrics.** Subjective user experience was assessed using a 5-point Likert scale. Participants rated the ease of use of the platform, reflecting how intuitive they perceived the pipeline construction process to be. Overall user satisfaction captured participants' holistic assessment of the platform and the outcomes achieved during the tasks. User confidence measured how confident participants felt in their ability to construct correct and meaningful ML pipelines. In addition, for *System A* only, participants rated the perceived usefulness of the AI assistant, reflecting its contribution to task completion and decision-making.
- **Quantitative Metrics.** Objective performance was assessed through logging and outcome-based metrics collected during task execution. *Task completion time* was measured as the time, in minutes, from the beginning of a task until the participant submitted or executed the final pipeline. For each system variant, completion times were computed separately for the classification and feature engineering tasks, with 24 attempts per task and system. *Task success* denotes the number and percentage of task attempts that resulted in a valid and executable pipeline that produced the expected output. *Errors* denote the mean number of configuration or implementation errors per task attempt. For the no-code systems, errors included invalid task connections, incompatible method selections, missing required inputs or outputs, and incomplete parameter configurations. For the code-based system, errors included runtime exceptions, incorrect preprocessing, invalid train-test splits, and incomplete or non-executable scripts. *AI assistant use* denotes the mean number of assistant requests per task attempt and is applicable only to *System A*. *Task search use* denotes the number and percentage of task attempts in which the participant used the task-search functionality at least once and is applicable to *System A* and *System B*. Except for task-specific completion times, aggregate metrics are computed over 48 task attempts per system, corresponding to two tasks completed by each of the 24 participants.

Data Collection and Procedure. Participants first completed a pre-study questionnaire after being briefed on the study goals and providing informed consent. This questionnaire captured background information on ML expertise and familiarity with similar tools. Participants then completed both tasks with all three system variants according to their assigned counterbalanced system order. The order of the two tasks was also counterbalanced across participants. During task execution, interaction data such as completion time, number of clicks, assistant usage frequency, and task search usage were logged automatically. After completing all tasks, participants filled out a post-study questionnaire consisting of Likert-scale ratings and open-ended questions addressing usability, satisfaction, and perceived strengths and weaknesses of the systems. Figures illustrating the study materials and representative examples are provided in Appendix A.

TABLE 5.2: Automated evaluation results for LLM-generated pipeline recommendations against OpenML reference pipelines.

Metric	Evaluation Level	Mean Score
Hit Rate	Method	0.495
Jaccard Coefficient	Task	0.587

5.5 Evaluation Results

This section presents the results of the evaluation described in the previous section. We first report the outcomes of the automated offline evaluation, which assesses the alignment between LLM-generated pipeline recommendations and established OpenML benchmark pipelines. We then present the results of the controlled user study, which evaluates the impact of the LLM-based assistant and the no-code interface on task efficiency, correctness, and user experience during interactive ML pipeline construction.

5.5.1 Results of the Automated LLM Pipeline Evaluation

Pipelines generated by the LLM assistant were compared against OpenML reference pipelines at both the method and task level. As shown in Table 5.2, the method-level evaluation yielded an average Hit Rate of 0.495. This indicates that approximately half of the methods used in the high-performing OpenML reference pipelines were also proposed by the LLM assistant. At the task level, the mean Jaccard similarity was 0.587, indicating moderate structural overlap between the task sequences proposed by the assistant and those present in the reference pipelines.

In addition to these aggregate metrics, manual inspection of selected pipelines suggests that the assistant can recommend semantically appropriate alternative methods for a given dataset and ML goal. The assistant also sometimes extends pipelines by introducing reasonable tasks, such as preprocessing or feature selection, that may improve pipeline robustness even when they are absent from the reference pipeline. A detailed analysis of this behavior is provided in Appendix A.

This automated validation provides a necessary foundation for the subsequent user study, which evaluates how these recommendations affect human users during interactive pipeline construction.

5.5.2 User Study Results

The user study evaluated the usability and effectiveness of *ExeKGLab* across three system variants: *System A* (no-code GUI with LLM assistant), *System B* (no-code GUI without assistant), and *System C* (conventional code-based workflow). Results are summarized using subjective ratings on a 5-point Likert scale (Table 5.4) and objective performance measures derived from task logging (Table 5.3). All reported user-study results are aggregated over $n = 24$ participants. For each system variant, the two tasks completed by each participant result in 48 task attempts. Completion times are reported separately per task, while task success, errors, assistant use, and search use are aggregated at the task-attempt level.

Efficiency, success, and errors. Participants completed tasks fastest with *System A*, followed by *System B*, while *System C* required the most time. For the classification task, the rounded average completion time was 20 minutes with *System A*, 33 minutes with *System B*, and 47 minutes with *System C*. For the feature engineering task, the corresponding durations were 10, 22, and 27 minutes. Across both tasks and all participants, *System A* achieved the highest task success, with 34 out of 48 task attempts resulting in a valid executable pipeline. *System B* achieved 24 successful attempts, while *System C* achieved 17. The same pattern was observed for errors: *System A* produced the fewest logged errors, with an average of 4.0 errors per task attempt, followed by *System B* with 9.0 and *System C* with 15.0.

User experience. Subjective ratings show a consistent trend across system variants. *System A* received the highest ratings for ease of use (3.75), user satisfaction (3.50), and confidence

TABLE 5.3: Quantitative user-study metrics across system variants. Completion times are rounded average durations in minutes, computed separately for each task over 24 attempts per system. Task success and task search use are reported as counts and percentages over 48 task attempts per system. Errors and AI assistant use are reported as means per task attempt over the same 48 attempts.

Metric	System A No-code + LLM	System B No-code	System C Code-based
Classification time	20 min	33 min	47 min
Feature engineering time	10 min	22 min	27 min
Task success	34/48 (71%)	24/48 (50%)	17/48 (35%)
Errors	4.0	9.0	15.0
AI assistant use	1.5	n/a	n/a
Task search use	29/48 (60%)	48/48 (100%)	n/a

(4.00). *System B* achieved moderate ratings across the same dimensions (3.00 for ease of use, 3.00 for satisfaction, and 3.50 for confidence), reflecting the benefits of the no-code GUI even without AI support. In contrast, *System C* consistently received lower ratings (2.7 for ease of use, 2.6 for satisfaction, and 2.8 for confidence), highlighting the higher cognitive and technical demands of a code-based workflow. Participants rated the LLM assistant as highly useful in *System A* (4.50), and open-ended feedback emphasized reduced cognitive load through concrete recommendations and decision validation.

Interaction behavior. Interaction logs show that participants consulted the assistant 1.5 times per task attempt on average in *System A*. The task-search functionality was used in 29 out of 48 task attempts in *System A*, compared with all 48 task attempts in *System B*. This suggests that the assistant reduced the need for manual component search in some cases, while search remained essential in the no-code-only condition.

Overall, the results indicate that the no-code GUI improves usability and efficiency compared to the conventional code-based workflow in this study. They further suggest that integrating the LLM-based assistant provides additional benefits in task efficiency, task success, and error reduction.

TABLE 5.4: Participant ratings across system variants on a 5-point Likert scale. Values indicate mean ratings aggregated over participants.

Criterion	System A No-code + LLM	System B No-code	System C Code-based
Ease of Use	3.75	3.00	2.70
User Satisfaction	3.50	3.00	2.60
Confidence	4.00	3.50	2.80
Perceived AI Assistant Usefulness	4.50	n/a	n/a

5.6 Conclusion

This chapter has presented the software tools developed to interact with *ExeKGs*. *ExeKGLib* provides the core semantic engine, enabling the programmatic creation, validation, and execution of ML pipelines as *ExeKGs*. *ExeKGLab* builds upon this foundation to offer a user-friendly, no-code graphical interface, further lowering the barrier to entry through visual programming and an LLM-powered assistant. Together, these tools provide a practical platform for ontology-grounded ML pipeline construction and execution, supporting the broader goal of ML democratization. The following chapters shift the focus to methods that learn from and on top of these semantic ML representations.

Part III

Knowledge Graph Embeddings for Link Prediction and Downstream Tasks

Chapter 6

ReaLitE: A Relation-Centric Literal-Aware Knowledge Graph Embedding Model

6.1 Introduction

As mentioned in Chapter 2, a challenge in the field of KGEs is the effective incorporation of literal values, such as numbers and dates, which are often associated with entities. Traditional KG embedding models, such as TransE [22] and ComplEx [144], primarily focus on the topological structure of the KG, but neglect the literal attributes of entities. Recently, the field of multimodal KG completion has gained significant attention [135, 90], and the importance of relation representation has been stressed by multiple works [93, 90]. Each relation in a KG establishes a connection between a pair of entities. Different types of relations may exhibit patterns in how the numeric attributes of their head and tail entities are linked. For instance, a pattern may arise where *people* (head entities) with a greater *monthly income* (literal) tend to *rent* (relation) *houses* (tail entities) with a higher *monthly rent* (literal), as illustrated in Figure 6.1.

By encoding the mathematical relationship between the literals of head and tail entities, it becomes possible to more accurately predict missing links. In the provided example (Figure 6.1), if we observe that *monthly income* is approximately three times the *monthly rent* for known triples, we can incorporate this learned pattern when ranking potential tail entities for a new head entity.

Existing KGE models that handle numeric literal values employ various strategies [54]. The LiteralE model [83] incorporates literals directly into entity embeddings, while the KGA model [153] preprocesses literals by converting them into new entities within the graph. To our knowledge, KBLRN [53] is the only method that attempts to use numeric literals as relation-specific features to enhance link prediction.

However, these methods generally overlook the potential relational patterns between the attributes of connected entities, as exemplified in Figure 6.1. KBLRN also faces challenges when literals are absent, which can limit the inclusion of numeric information. We argue that there is an opportunity for improvement by integrating head and tail numeric information in a learnable manner, enabling the model to capture these underlying quantitative relationships. As the relation serves as the connection point between head and tail entities, we propose using the relation's embedding as a vessel for this information fusion.

In the domain of node classification, the performance of triple-scoring KGE methods has been a subject of limited assessment, and numeric literals are often either excluded or treated as conventional nodes [125, 132, 26]. A recent study proposed a literal preprocessing method based on the current state-of-the-art and evaluated it for node classification [117], providing a suitable baseline for comparison.

This chapter introduces `ReaLitE`, a relation-centric method designed to enhance vanilla KGE models by infusing numeric information from both head and tail entities into the embeddings of their connecting relations. For each relation, the values of each numeric attribute are aggregated separately for its head and tail entities, generating two numeric vectors. This aggregated numeric information is then combined with the relation embedding

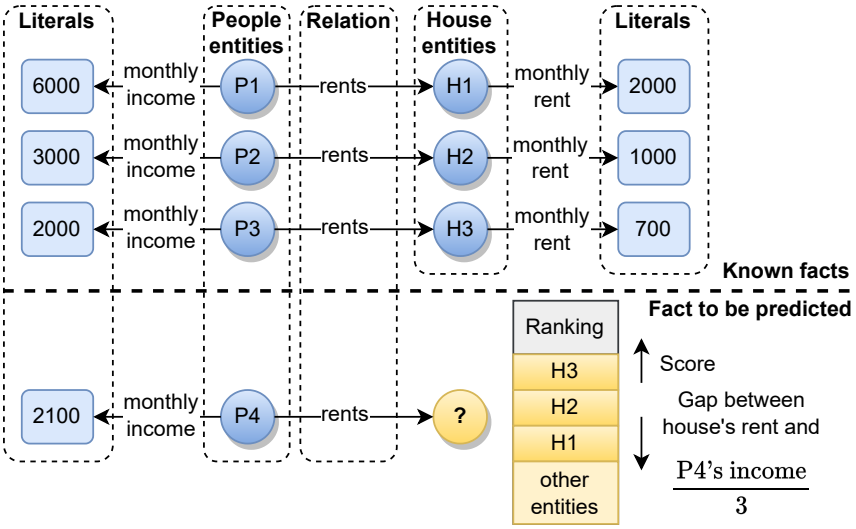


FIGURE 6.1: An example showing the potential benefit of the mathematical relationship between people's income and houses' rent during link prediction.

through a machine-learnable strategy. This infusion enriches the relation embedding vectors with information about the correlations between the numeric attributes of its head and tail entities. This chapter provides a description of the `ReaLitE` method, its architecture, and a thorough evaluation on public benchmarks for both link prediction and node classification tasks, where it demonstrates superior or comparable performance to existing state-of-the-art methods.

6.2 Related Work

Several literal-aware KGE approaches have been proposed, as discussed in Section 2.4. This section positions `ReaLitE` with respect to the most relevant prior work on literal-aware link prediction and node classification.

6.2.1 Literal-aware Link Prediction

KBLRN [53], LiteralE [83], and KGA [153] are among the most relevant approaches for incorporating numeric literals into link prediction. KBLRN uses numerical differences between the head and tail entities, LiteralE injects literal values into entity embeddings, and KGA transforms literals into additional graph structure through discretization. These approaches are described in more detail in Section 2.4.

Although these methods demonstrate the usefulness of numeric literals, they do not directly model relation-specific interactions between the numeric attributes of head and tail entities. KBLRN uses numerical differences but requires the relevant numeric values to be available for both entities. LiteralE focuses on enriching entity embeddings rather than relation embeddings. KGA is model-agnostic, but it represents numeric values through discretized bins, which may lose fine-grained numerical information.

In contrast, `ReaLitE` directly incorporates numeric information into relation representations and models how the numeric attributes of head and tail entities interact for different relation types. In this work, we therefore evaluate `ReaLitE` not only in the standard link prediction setting but also on specific relation categories, including long-tail relations and relations with correlated head and tail attributes.

6.2.2 Node Classification

Node classification is commonly addressed using message-passing models such as R-GCNs [26, 132, 155] or feature extraction methods such as RDF2Vec [125]. Triple score-based KGE methods are less frequently evaluated on downstream tasks such as node classification.

Recent work has shown that strong link prediction performance does not necessarily imply strong performance on downstream tasks such as recommendation or question answering [162]. This motivates evaluating ReaLitE beyond link prediction. In this chapter, we therefore also study whether the embeddings learned by ReaLitE are useful for node classification.

Prior work has evaluated triple score-based KGE methods on node classification. However, these studies either use datasets that are not well suited for such models [158] or do not include literals in their multi-relational KGs [1]. The most relevant baseline for our setting is the multimodal preprocessing approach proposed by Preisner et al. [117], which extends KGA-style preprocessing to different literal modalities. We use this approach as a primary baseline and compare it against ReaLitE on the node classification task.

6.3 The ReaLitE Method

The ReaLitE approach is designed as a modular enhancement that can be combined with any existing KGE model that learns relation embeddings. The method involves a two-step process: first, for a given relation, the numeric attributes of its head and tail entities are aggregated; second, this aggregated numeric information is fused with the relation's embedding vector.

6.3.1 Problem Description

A KG with numeric literals can be defined as a set of triples $G = \{(h,r,t)|h,t \in E, r \in R\} \cup \{(e,a,v)|e \in E, a \in A, v \in \mathbb{R}\}$, where E is the set of entities, R is the set of relations, and A is the set of numeric attributes.

The link prediction task aims to predict the plausibility of a triple (h,r,t) given the graph G. This can be modeled as a ranking problem, where for a given pair (h,r), all entities $e_i \in E$ are ranked based on the probability of (h,r,e_i) being a true triple, computed using a scoring function $f(h,r,t)$.

Node classification is a prominent downstream task concerned with predicting the category of a given entity. Formally, for a given graph G and a set of entity labels Y, the task is to learn a function $f(e_i) \rightarrow Y_i$ for each entity $e_i \in E$. A common approach to this task is to first learn a vector representation for each entity using a pre-trained KGE model, $emb(e_i)$, and then train a classifier on these embeddings, such that $emb(e_i) \rightarrow Y_i$.

6.3.2 The ReaLitE Architecture

Our proposed method, ReaLitE, enhances traditional KGE models by following a two-step procedure, as illustrated in Figure 6.2. First, it aggregates the numeric literals associated with the head and tail entities of a given relation. Second, it integrates this aggregated numeric information into the relation's embedding. A key design principle of ReaLitE is its modularity; these steps are independent of the KGE model's scoring function, allowing our approach to serve as a complementary enhancement for a wide range of existing models, as will be demonstrated in the experiments.

The process begins with a matrix of literals, $\mathbf{L} \in \mathbb{R}^{|E|\times|A|}$, where $|E|$ is the number of entities and $|A|$ is the number of numeric attributes. Each entry, $\mathbf{L}_{i,k}$, holds the value of the k-th numeric attribute for the i-th entity, with a value of zero if the attribute is not present. In line with standard practice, this matrix is normalized using min-max normalization [83].

For each relation, two matrices, **Lh** and **Lt** (both of dimension $\mathbb{R}^{|E|\times|A|}$), are created. These matrices isolate the literal vectors from $\mathbf{L}$ that correspond to the head and tail entities of that specific relation, respectively, with all other rows being zeroed out. An aggregation method is then applied column-wise to these two matrices, producing a pair of summary vectors, $\mathbf{l_h}$ and $\mathbf{l_t}$, both in $\mathbb{R}^{|A|}$.

The core of the ReaLitE architecture is a fusion function, g, with the signature $g : \mathbb{R}^{|A|} \times \mathbb{R}^{D_r} \times \mathbb{R}^{|A|} \rightarrow \mathbb{R}^{D_r}$, where D_r is the dimensionality of the relation embedding. This function takes as input the original relation embedding $\mathbf{r}$ and the two aggregated literal vectors, $\mathbf{l_h}$

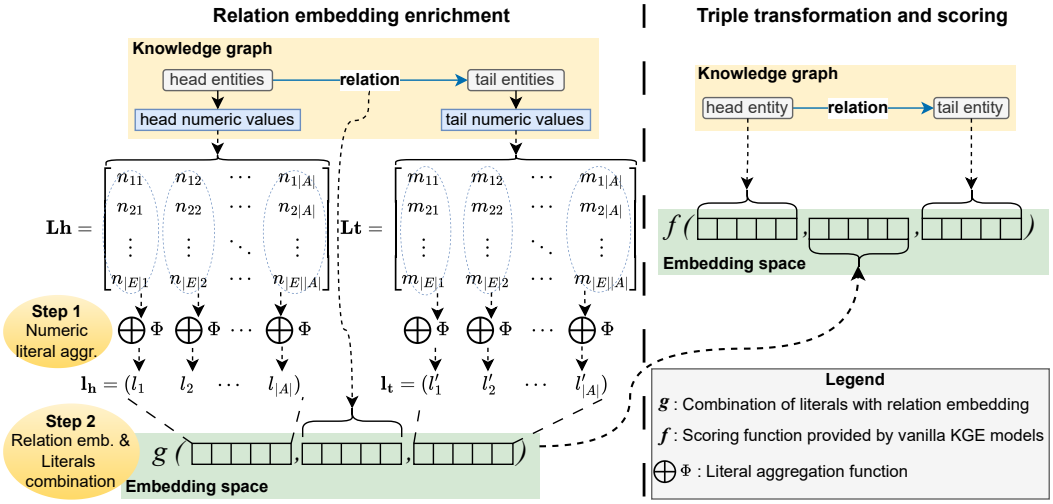


FIGURE 6.2: The ReaLitE pipeline for relation embedding enrichment. Numeric literals from head and tail entities are aggregated (Step 1) and then combined with the relation embedding (Step 2) before being used by the base KGE model's scoring function.

and $\mathbf{l_t}$. Its output is a new, literal-enhanced relation embedding, $\mathbf{r_i}^{lit} = g(\mathbf{l_h}, \mathbf{r_i}, \mathbf{l_t})$, which has the same dimensionality as the original. This new vector is designed to be a drop-in replacement for the original relation embedding $\mathbf{r_i}$ within the scoring function of any base KGE model, leaving the other components of the scoring function unchanged.

The following sections will elaborate on the two main steps of the ReaLitE method in detail.

6.3.3 Aggregation of Numeric Literals

To reduce space complexity and the number of trainable parameters, the literal matrices **Lh** and **Lt** are transformed into vectors. This is achieved by aggregating the values for each numeric attribute, a column-wise operation that preserves the dimensionality corresponding to the number of attributes ($|A|$). This design ensures that all available numeric attributes can be incorporated, allowing the model to learn their relative importance during training. This process corresponds to "Step 1" in Figure 6.2.

$$\mathbf{l_h} = \left(\bigoplus_{i=1}^{|E|} \Phi(\mathbf{Lh}_{i,1}) \quad \cdots \quad \bigoplus_{i=1}^{|E|} \Phi(\mathbf{Lh}_{i,|A|})\right) \tag{6.1}$$

$$\mathbf{l_t} = \left(\bigoplus_{i=1}^{|E|} \Phi(\mathbf{Lt}_{i,1}) \quad \cdots \quad \bigoplus_{i=1}^{|E|} \Phi(\mathbf{Lt}_{i,|A|})\right) \tag{6.2}$$

The aggregation method, denoted by $\bigoplus \Phi$, is treated as a tunable hyper-parameter. This provides flexibility, as the optimal aggregation strategy may vary depending on the statistical distribution of the literal values for each attribute. The available choices for $\bigoplus \Phi$ include standard measures of central tendency $\{mean, median, mode\}$, other descriptive statistics $\{min, max, sum, count\}$, and measures of dispersion $\{variance, std, IQR, range\}$. Additionally, we provide an option to use a learnable linear combination $\mathbf{y}$ of all these aggregation types, defined as:

$$\mathbf{y} = \sigma(\mathbf{U} \cdot \mathbf{W}_a + \mathbf{b}_1) \tag{6.3}$$

where, for each aggregation function ϕ in the set of 11 types, there is a corresponding row $\mathbf{u}$ in the matrix $\mathbf{U}^\intercal$ such that:

$$\mathbf{u} = \left(\bigoplus_{i=1}^{|E|} \phi(\mathbf{L}_{i,1}) \quad \cdots \quad \bigoplus_{i=1}^{|E|} \phi(\mathbf{L}_{i,|A|})\right) \tag{6.4}$$

The input of the linear function $\mathbf{y} \in \mathbb{R}^{|A|\times 1}$ is $\mathbf{U} \in \mathbb{R}^{|A|\times 11}$. The learnable parameters are the weight matrix $\mathbf{W}_a \in \mathbb{R}^{11\times 1}$ and the bias term $\mathbf{b}_1 \in \mathbb{R}$, with σ representing the sigmoid function. This learnable aggregation is applied separately to both **Lh** and **Lt**.

6.3.4 Fusing Literals with Relation Embeddings

In the second step of the ReaLitE method, the aggregated literal vectors, $\mathbf{l_h}$ and $\mathbf{l_t}$, are fused with the original relation embedding via a learnable function g ("Step 2" of Figure 6.2). We propose two variants for this function: a simple linear transformation (g_{lin}) and a more complex gated mechanism (g_{gated}). The gated version follows LiteralE [83] and adopts the intuition of gated neural units introduced by Cho et al. [32]. Cho et al. proposed a recurrent neural network (RNN) encoder–decoder architecture for phrase-based statistical machine translation, where one RNN encodes a source phrase into a fixed-length vector and another RNN decodes it into a target phrase. To improve the memory capacity and trainability of the model, they introduced gated hidden units with reset and update gates. These gates are learnable vectors with values between 0 and 1 that control how much previous information should be retained and how much newly computed information should update the hidden state. Inspired by this idea, ReaLitE uses a gate to control, for each embedding dimension, how much of the literal-enriched relation representation should replace or preserve the original relation embedding.

$$g_{lin} = \mathbf{W}_r^T[\mathbf{l_h}, \mathbf{r}, \mathbf{l_t}] + \mathbf{b}_2 \tag{6.5}$$

$$g_{gated} = \mathbf{z} \odot \mathbf{h} + (1 - \mathbf{z}) \odot \mathbf{r} \tag{6.6}$$

where:

$$\mathbf{z} = \sigma(\mathbf{W}_{zlh}^T \mathbf{l_h} + \mathbf{W}_{zr}^T \mathbf{r} + \mathbf{W}_{zlt}^T \mathbf{l_t} + \mathbf{b}_3)$$
$$\mathbf{h} = h'(\mathbf{W}_r^T[\mathbf{l_h}, r, \mathbf{l_t}])$$

In addition to the base relation embedding $\mathbf{r} \in \mathbb{R}^{D_r}$, the trainable parameters for these functions include the weight matrices $\mathbf{W}_r \in \mathbb{R}^{|A|+D_r+|A| \times D_r}$, $\mathbf{W}_{zr} \in \mathbb{R}^{D_r \times D_r}$, $\mathbf{W}_{zlh}, \mathbf{W}_{zlt} \in \mathbb{R}^{|A| \times D_r}$ and the bias vectors $\mathbf{b}_2, \mathbf{b}_3 \in \mathbb{R}^{D_r}$. The symbol $\odot$ denotes element-wise multiplication, σ is the sigmoid function, and h' is an element-wise nonlinear activation function such as tanh.

6.3.5 Model Complexity

Computational Complexity. The computational complexity of ReaLitE can be analyzed in two phases. The preprocessing phase involves a single traversal of all triples (N_t) to populate the literal matrices, **Lh** and **Lt**, which has a complexity of $O(N_t)$. This is followed by an aggregation step for each attribute across every relation, with a complexity of $O(|E| \cdot |A| \cdot |R|)$. Thus, the total computational complexity of the preprocessing step is $O(N_t + |E| \cdot |A| \cdot |R|)$. During the training phase, the function g introduces a computational overhead. For g_{lin}, this consists of one matrix multiplication and one vector addition per operation. For g_{gated}, it increases to three matrix multiplications and one vector addition. An additional matrix multiplication and vector addition are required if the learnable aggregation function $\mathbf{y}$ is used.

Trainable Parameters. In terms of model complexity, ReaLitE introduces a number of trainable parameters beyond those of the base KGE model (Table 6.1). The exact number of additional parameters is determined by the choice of the fusion function g and the literal aggregation function $\mathbf{\Phi}$. The fusion function contributes parameters through the weight matrices $\mathbf{W}$ used in either Eq. 6.5 (g_{lin}) or Eq. 6.6 (g_{gated}). The only aggregation method that adds trainable parameters is the learnable function $\mathbf{y}$, which introduces the parameters of the weight matrix $\mathbf{W}_a$ and bias vector $\mathbf{b}_1$. As the number of aggregation types is fixed at 11, this overhead is constant. When compared to LiteralE, another literal-aware model that modifies base embeddings, the additional complexity introduced by ReaLitE is constant and therefore negligible.

Space Complexity. The space complexity of the literal preprocessing phase is determined by the two matrices, **Lh** and **Lt**, created for each relation, resulting in the storage of $2 \cdot |E| \cdot |A| \cdot$

TABLE 6.1: Number of trainable parameters for LiteralE and for variations of ReaLitE. B represents the number of trainable parameters of base models (*e.g.*, TransE). The parentheses in $(+12)$ indicate that this number is added only if the learnable function $\mathbf{y}$ is used to aggregate literals.

Model	Parameters
LiteralE	$B + 2D_e^2 + 2\|A\|D_e + D_e$
ReaLitE-g_{lin}	$B + D_r^2 + 2\|A\|D_r + D_r(+12)$
ReaLitE-g_{gated}	$B + 2D_r^2 + 4\|A\|D_r + D_r(+12)$

TABLE 6.2: Datasets used for link prediction

	Entities	Relations	Triples	Attributes	Literals
FB15k-237	14,541	237	310,116	116	29,220
YAGO15k	15,136	32	138,056	7	23,520

$|R|$ numeric values, which corresponds to a complexity of $O(|E| \cdot |A| \cdot |R|)$. These matrices are then aggregated into the vectors $\mathbf{l}_h$ and $\mathbf{l}_t$ during preprocessing. Consequently, the space required during the model training and testing phases is reduced to $2 \cdot |A| \cdot |R|$ values.

6.4 Experimental Setup

To validate the effectiveness of the ReaLitE method, a set of experiments was conducted on both link prediction and node classification tasks.

6.4.1 Datasets

Link Prediction Datasets. The evaluation was conducted on two standard benchmarks for literal-aware link prediction. The first, **FB15k-237** [143], is a well-known subset of the Freebase KG [19] that has been modified to increase the difficulty of the link prediction task by removing inverse relations. The dataset, which was later enriched with numeric literals using a SPARQL endpoint [83], covers diverse domains such as movies, sports, and awards. The second benchmark, **YAGO15k** [52], is a subset of the YAGO general-domain KG [96]. It was extended with numeric literals in [92] and is notable for having a higher proportion of valid numeric triples than FB15k-237. For both datasets, the data splits used in our experiments align with the standards set by recent literal-aware KGE research [83, 153]. A summary of the datasets' metadata is provided in Table 6.2.

Node Classification Datasets. For this evaluation, we selected three multimodal datasets from the kgbench collection [17]: dmgfull, which contains data on monuments in the Netherlands; dmg777k, a subset of dmgfull; and mdgenre, which consists of movie data extracted from Wikidata. This choice facilitates a direct comparison with recent literature [117]. While the original datasets include various data modalities, our experiments focused exclusively on numeric attributes. To this end, we used only the number and date literals, converting all date values to the decimal format YYYY.MMDD. The detailed statistics for these datasets are provided in Table 6.3.

6.4.2 Models

Link Prediction Models. We evaluated the ReaLitE framework by combining it with five well-established KGE models: TransE [22], DistMult [159], ComplEx [144], RotatE [141], and TuckER [10]. The ConvE model [38] was excluded from our evaluation, as its architectural

TABLE 6.3: Datasets used for node classification

	dmgfull	dmg777k	mdgenre
Entities	593,291	288,379	1,001,791
Relations	62	60	154
Triples	1,850,451	777,124	1,252,247
Numbers	88,168	10,706	14,352
Dates	–	–	113,463
Classes	14	5	12
Nodes	842,550	341,270	349,344

requirement for inverse triples is incompatible with our chosen training strategy. The performance of the `ReaLitE`-enhanced models was compared against two groups of baselines: the five aforementioned KGE models in their vanilla form, and three state-of-the-art literal-aware models, namely KBLN [53], LiteralE [83], and KGA [153]. For each dataset and base KGE model, we conducted experiments to find the optimal configuration of `ReaLitE`, including the best literal aggregation strategy. The results reported are those of the best-performing configuration, as measured by the MRR metric.

Node Classification Models. We combined `ReaLitE` with the TransE and DistMult embedding models. This choice allows for a direct comparison with the results reported in recent literature [117]. To establish a strong baseline, we also evaluated the performance of TransE and DistMult when trained using the literal preprocessing methods from the same reference. This baseline framework, which we refer to as MKGA for simplicity, employs two main strategies: KL-REL+LOF, which bins numeric values based on relation similarity while removing outliers, and DATBIN, which converts date literals into timestamps before binning [117]. For the final classification step, the entity embeddings produced by each model configuration were used to train both a Support Vector Machine (SVM) [23] and a k-Nearest Neighbors (KNN) [50] classifier.

6.4.3 Training

Link Prediction Training. We adopted a symmetric training strategy that scores both head and tail predictions for every true triple in the training data. This approach was selected because it preserves the original semantics of the relation types by avoiding the creation of synthetic inverse relations, a common practice in methods like the Local Closed World Assumption (LCWA) [5]. Preserving the original relations is important for the `ReaLitE` framework, which focuses on enriching these specific relation embeddings.

This training paradigm, implemented in PyKEEN as "symmetric LCWA", is based on the concept of "instantaneous multi-class log-loss" as initially proposed in [84]. We utilized the Cross-Entropy loss function, which has been shown to yield robust results for KGE training [129]. The loss ℓ_{ijk} for a given training triple (i, j, k), a normalized ground truth tensor y, and a score tensor s, is given by eq. 6.7. This loss consists of two components: the loss for predictions after perturbing the tail entity k (eq. 6.8) and the loss for predictions after perturbing the head entity i (eq. 6.9).

$$\ell_{ijk} = \ell^{(1)}_{ijk} + \ell^{(3)}_{ijk} \tag{6.7}$$

$$\ell^{(1)}_{ijk} = -\sum_{k'} y_{ijk'} \log \left(\frac{\exp(s_{ijk'})}{\sum_{k'} \exp(s_{ijk'})} \right) \tag{6.8}$$

$$\ell^{(3)}_{ijk} = -\sum_{i'} y_{i'jk} \log \left(\frac{\exp(s_{i'jk})}{\sum_{i'} \exp(s_{i'jk})} \right) \tag{6.9}$$

Node Classification Training. The entity embeddings pre-trained on link prediction serve as the feature vectors. We used these embeddings to train two classifiers. The SVM was trained using a hinge loss function, which is designed to find a hyperplane that optimally separates the classes by maximizing their margin. The kNN classifier, as a non-parametric algorithm, does not undergo a training phase involving a loss function; it makes predictions based on the classes of the nearest neighbors at inference time.

6.4.4 Evaluation Settings

Overall Link Prediction. The evaluation protocol assesses the model on two prediction tasks for every test triple (h, r, t). For tail prediction, the model scores all candidate triples of the form (h, r, e'), where e' is drawn from the set of all entities E. A symmetric task is then performed for head prediction, scoring all candidates of the form (e', r, t). The rank of the true test triple is then determined by its position within the sorted list of these candidate scores. To ensure a fair evaluation, all other true triples that exist in the training set are filtered from the candidate lists before the ranking is performed. The final results are reported using established rank-based metrics, namely Mean Reciprocal Rank (MRR) and Hits@k (for k=1 and k=10).

Relation-focused Link Prediction. This analysis involved partitioning the test set into groups based on the properties of the relation in each triple and then calculating the MRR for each group separately. We employed two partitioning strategies. The first strategy evaluated performance on relations of varying frequency by splitting the test set into "frequent" and "long-tail" groups, using a frequency of 2817 triples (2.55% of the training set) as the threshold (Table 6.5). The second strategy aimed to assess the impact of numeric patterns by grouping relations based on the correlation of their associated numeric attributes. For this, we calculated the pairwise Pearson's correlation coefficient (denoted by "$corr.\ coef.$") between the numeric attributes of the head and tail entities for each relation and used a threshold of 0.2 to divide the relations into high and low correlation groups (Table 6.6).

Node Classification. We evaluated performance using the micro-F1 score, a standard metric that calculates the harmonic mean of precision and recall. To compute this score, the trained classification model was first used to predict labels for all entities in the test set. These predictions were then compared against the ground truth labels to aggregate the total number of true positives, false positives, and false negatives across all classes. This data was then used to calculate the overall precision and recall, from which the final micro-F1 score was derived. This approach was chosen because the micro-F1 score treats each sample equally, making it a robust measure for overall classification performance.

6.5 Results

This section presents the results of our experimental evaluation, where we combined `ReaLitE` with several vanilla KGE models to assess its performance. We first detail the results of the link prediction task, where the `ReaLitE`-enhanced models are compared against both their vanilla counterparts and other state-of-the-art literal-aware methods. We then present the results for the downstream task of node classification, investigating the effectiveness of the embeddings produced by `ReaLitE` in this application.

6.5.1 Link Prediction Results

Overall Link Prediction. In the standard link prediction setting, the results presented in Table 6.4 demonstrate that enhancing vanilla KGE models with `ReaLitE` leads to notable performance improvements across most configurations. On the FB15k-237 dataset, the increase in MRR reached a maximum of 17% when `ReaLitE` was combined with ComplEx. Similarly, on YAGO15k, which contains higher quality numeric literals, the MRR gains were as high as 10%, again with ComplEx. The effectiveness of our approach on this dataset is further underscored by the fact that the best overall MRR and Hits@1 scores were achieved by

TABLE 6.4: Link prediction results. In this context, ReaLitE uses g_{lin} as it yielded better results than g_{gated}. The baseline results are sourced from the current state-of-the-art paper, with the best results per metric and base KGE model in bold.

	FB15k-237			YAGO15k		
Model	**MRR**	**H@1**	**H@10**	**MRR**	**H@1**	**H@10**
TuckER	0.354	0.263	0.536	0.433	0.360	0.571
+LiteralE	0.353	0.262	0.536	0.421	0.348	0.564
+KBLN	0.345	0.253	0.530	0.420	0.349	0.556
+KGA	**0.357**	**0.265**	**0.540**	0.454	0.380	0.592
+ReaLitE	0.347	0.254	0.533	**0.463**	**0.387**	**0.608**
TransE	0.315	0.217	0.508	0.459	0.376	0.615
+LiteralE	0.315	0.218	0.504	0.458	0.376	0.612
+KBLN	0.308	0.210	0.496	0.466	0.382	0.621
+KGA	0.321	0.223	0.516	0.470	0.387	**0.623**
+ReaLitE	**0.335**	**0.242**	**0.520**	**0.491**	**0.422**	0.620
DistMult	0.295	0.212	0.463	0.457	0.389	0.585
+LiteralE	0.309	0.223	0.481	0.462	0.396	0.587
+KBLN	0.302	0.220	0.470	0.449	0.377	0.581
+KGA	0.322	0.233	0.502	0.472	0.402	0.606
+ReaLitE	**0.340**	**0.248**	**0.525**	**0.489**	**0.419**	**0.622**
ComplEx	0.288	0.205	0.455	0.441	0.370	0.572
+LiteralE	0.295	0.212	0.462	0.443	0.375	0.570
+KBLN	0.293	0.213	0.451	0.451	0.380	0.583
+KGA	0.305	0.219	0.478	0.453	0.380	0.591
+ReaLitE	**0.338**	**0.244**	**0.528**	**0.487**	**0.417**	**0.621**
RotatE	0.324	0.232	0.506	0.451	0.370	0.605
+LiteralE	0.329	0.237	0.512	0.475	0.400	0.619
+KBLN	0.314	0.222	0.500	0.469	0.393	0.613
+KGA	0.335	0.242	0.521	0.473	0.392	**0.626**
+ReaLitE	**0.344**	**0.249**	**0.535**	**0.483**	**0.409**	0.624

the TransE+ReaLitE model. With the exception of TuckER on FB15k-237, the integration of ReaLitE consistently improved or matched the performance of the underlying vanilla KGE model. The results with ComplEx provide an interesting case for the superiority of a relation-centric approach. The performance gain is pronounced because the ReaLitE enhancement is applied to both the real and imaginary parts of ComplEx's relation embeddings, effectively doubling its impact. This amplified effect, which is not observed to the same degree with the entity-centric LiteralE, highlights the benefit of directly enriching the relational component with numeric context.

Relation-focused Link Prediction. To gain a more granular understanding of model performance, we conducted a relation-focused analysis on the YAGO15k test set. This involved partitioning the test triples into groups based on the characteristics of their relations and evaluating the MRR for each group separately.

First, we analyzed performance based on relation frequency by splitting the test set into "Frequent" and "Long-tail" subsets, with the results shown in Table 6.5. The results show that ReaLitE provides a significant performance increase for the long-tail relations across most base models. This suggests that by incorporating numeric context, ReaLitE is better able to model relations with fewer training examples, which constitute the majority in most KGs. The exceptional performance of TuckER+KGA is likely due to the synthetic relations

TABLE 6.5: MRR for the filtered test set of YAGO15k based on relation types grouped by their frequency. "Long-tail" refers to the test triples that contain relations present in $\leq$ 2.55% of the training triples. "Frequent" refers to the remaining test triples. "All triples" refers to all test triples. The best results per base model per triple group are in bold. The percentages indicate the MRR increase of `ReaLitE` relative to the corresponding vanilla baseline.

Model	Frequent	Long-tail	All triples
TuckER	0.455	0.263	0.433
+KBLN	0.463	0.280	0.420
+LiteralE	0.464	0.282	0.421
+KGA	0.480	**0.288**	0.454
+ReaLitE	+8% **0.491**	+4% 0.274	+7% **0.463**
TransE	0.492	0.222	0.459
+KBLN	0.496	0.235	0.466
+LiteralE	0.489	0.233	0.458
+KGA	0.502	0.237	0.470
+ReaLitE	+7% **0.525**	+18% **0.262**	+7% **0.491**
DistMult	0.490	0.236	0.457
+KBLN	0.480	0.240	0.449
+LiteralE	0.492	0.250	0.462
+KGA	0.501	0.258	0.472
+ReaLitE	+6% **0.519**	+20% **0.284**	+7% **0.489**
ComplEx	0.467	0.251	0.441
+KBLN	0.478	0.258	0.451
+LiteralE	0.470	0.253	0.443
+KGA	0.479	0.256	0.453
+ReaLitE	+10% **0.514**	+20% **0.300**	+10% **0.487**
RotatE	0.475	0.284	0.451
+KBLN	0.480	0.279	0.469
+LiteralE	0.486	0.285	0.475
+KGA	0.494	0.298	0.473
+ReaLitE	+7% **0.510**	+6% **0.302**	+7% **0.483**

and entities created by KGA's preprocessing step. This method results in a KG structure that may benefit the TuckER model, a phenomenon also discussed in [18]. For details, see Appendix B.

Second, to investigate the impact of numeric patterns directly, we partitioned the test set based on the correlation between the numeric attributes of the head and tail entities for each relation, as shown in Table 6.6. The results show that `ReaLitE` consistently improves performance on relations with less correlated literals ("Less corr. lit."), indicating its ability to capture nuanced quantitative signals. For relations with more highly correlated literals ("Corr. lit."), the impact was even more pronounced for certain models; TransE and ComplEx, for example, saw their MRR increase by up to 72% and 65% respectively. This demonstrates that the proposed mechanism of infusing literal information directly into the relation embedding can be highly beneficial for specific KGE architectures. It is noteworthy that for the TuckER and RotatE models on the highly correlated subset, the KGA baseline achieved a higher MRR. We attribute this to the synthetic entities and relations created by KGA's preprocessing, which may result in a graph structure that is advantageous for certain tensor factorization models, as discussed in [18]. For details, see Appendix B.

Analysis of Literal Aggregation Methods. For the link prediction task, the choice of literal aggregation function was treated as a hyper-parameter. To ensure a manageable search space

TABLE 6.6: MRR for the filtered test set of YAGO15k based on relation types grouped by literal correlation. "Corr. lit." refers to the test triples where the head and tail attributes are more correlated (the relation has at least one pair of head and tail attributes with $|corr.coef.| \geq 0.2$ in the training set). "Less corr. lit." refers to the remaining test triples where the head and tail attributes are less correlated. "All triples" refers to all test triples. The best results per base model per triple group are in bold. The percentages indicate the MRR increase of `ReaLitE` relative to the corresponding vanilla baseline.

Model	Less corr. lit.	Corr. lit.	All triples
TuckER	0.454	0.265	0.433
+KBLN	0.464	0.271	0.420
+LiteralE	0.465	0.272	0.421
+KGA	0.477	**0.298**	0.454
+ReaLitE	+7% **0.488**	+9% 0.289	+7% **0.463**
TransE	0.503	0.151	0.459
+KBLN	0.505	0.160	0.466
+LiteralE	0.500	0.146	0.458
+KGA	0.510	0.176	0.470
+ReaLitE	+4% **0.524**	+72% **0.259**	+7% **0.491**
DistMult	0.491	0.220	0.457
+KBLN	0.482	0.216	0.449
+LiteralE	0.494	0.227	0.462
+KGA	0.501	0.245	0.472
+ReaLitE	+5% **0.517**	+30% **0.287**	+7% **0.489**
ComplEx	0.476	0.190	0.441
+KBLN	0.478	0.238	0.451
+LiteralE	0.477	0.193	0.443
+KGA	0.479	0.245	0.453
+ReaLitE	+7% **0.511**	+65% **0.314**	+10% **0.487**
RotatE	0.477	0.237	0.451
+KBLN	0.476	0.300	0.469
+LiteralE	0.482	0.310	0.475
+KGA	0.488	**0.333**	0.473
+ReaLitE	+7% **0.509**	+25% 0.296	+7% **0.483**

during hyper-parameter optimization (HPO), the set of candidate aggregation types was limited to $\{mean, median, mode, min, \mathbf{y}\}$. This subset was chosen to include all measures of central tendency, a simple descriptive statistic, and the learnable function $\mathbf{y}$, which provides a combination of all supported aggregation types.

The analysis of the best-performing configurations, presented in Table 6.7, reveals that the optimal aggregation function is dependent on both the dataset and the base KGE model. A degree of consistency was observed for the TransE, RotatE, and TuckER models, which favored the same aggregation function across both the YAGO15k and FB15k-237 datasets. Overall, the *mode* aggregation was the most frequently selected, appearing in 50% of the best configurations. The *mean* and *min* aggregations were each selected in 20% of cases, while the learnable function $\mathbf{y}$ was chosen in 10% of the optimal configurations. Notably, the *median* function, despite being included in the HPO search, was not part of any of the final best-performing models. Further analysis is included in the Appendix B.

TABLE 6.7: Aggregation function used in the best hyper-parameter configurations of ReaLitE. Function **y** is a learnable combination of all supported aggregation types (Eq. 6.3).

	TransE	DistMult	ComplEx	RotatE	TuckER
FB15k-237	*mode*	**y**	*mode*	*mode*	*mean*
YAGO15k	*mode*	*min*	*min*	*mode*	*mean*

TABLE 6.8: Node classification results – Micro-F1 score. In this context, ReaLitE uses g_{gated} as it yielded better results than g_{lin}. The best results per dataset are underlined. The best results per base KGE model per classifier are in bold.

	dmg777k		dmgfull		mdgenre	
Model	**KNN**	**SVM**	**KNN**	**SVM**	**KNN**	**SVM**
TransE	0.506	0.528	**0.649**	**0.662**	**0.634**	0.646
+MKGA	0.439	0.472	0.583	0.573	0.607	0.606
+ReaLitE	**0.609**	**0.638**	0.597	0.605	0.632	**0.662**
DistMult	0.548	0.542	0.619	0.658	0.605	0.622
+MKGA	0.472	0.495	0.605	0.647	0.630	0.640
+ReaLitE	**0.582**	**0.642**	**0.639**	**0.682**	**0.659**	**0.674**

6.5.2 Node Classification Results

The performance of the learned embeddings was also assessed on the downstream task of node classification, with the results presented in Table 6.8. The embeddings produced by the ReaLitE-enhanced models demonstrated strong performance. Notably, a model enhanced with ReaLitE achieved the highest overall micro-F1 score on every dataset evaluated. The practical utility of the enriched embeddings is highlighted by the results on the dmg777k dataset, where the integration of ReaLitE led to an average F1 score increase of 16% over the vanilla KGE models.

6.6 Conclusion

This chapter introduced ReaLitE, a novel, relation-centric approach for incorporating numeric literals into KGEs. By aggregating the numeric attributes of connected entities and fusing this information with the relation's embedding, ReaLitE is able to model quantitative relationships that are ignored by existing KGE methods. The thorough evaluation on link prediction and node classification tasks demonstrated the effectiveness of this approach. The results show that ReaLitE often improves overall performance and is especially effective for long-tail relations and relations with correlated numeric attributes.

Chapter 7

Integrating Meta-Features with *ExeKG* Embeddings for Meta-Learning

7.1 Introduction

The rapid growth of data science has led to the creation of a vast collection of ML artifacts available on the web. Platforms such as OpenML [148] host a large number of experiment records covering diverse datasets, pipelines, and pipeline configurations.

The availability of ML experiment data on the web has driven increasing interest in meta-learning. Among the core tasks in this setting are *PPE* [29] and *DPSE* [86]. Addressing these tasks enables the identification of promising pipelines early and helps narrow the search space, thereby reducing the need for expensive training and optimization runs [48, 139].

In this chapter, we investigate whether encoding past experiment data in a KG and learning numerical representations of its entities can improve both *PPE* and *DPSE*. To this end, we introduce a novel approach, named *KGmetaSP*, that employs KGEs to model the latent structure relating datasets and their top-performing pipeline configurations (Figure 7.1). Our approach builds on the ability of KGs to capture hidden patterns and transitive relationships [104].

Specifically, we construct *MetaExe-KG*, a unified KG that integrates semantic representations of datasets and pipeline configurations, together with the links between them, using experiment data sourced from OpenML [148]. For dataset representation, we adopt *MLSea-KG* [36], which models dataset metadata including meta-features. For pipeline configuration representation, we adopt *ExeKGs*, introduced in Chapter 4, which capture the data flow of ML pipeline configurations, including preprocessing steps, models, and hyper-parameters. We then apply KGE methods to learn vector representations for both datasets and pipeline configurations. The learned dataset embeddings are used with vector distance measures to estimate performance-based dataset similarity, while the pipeline embeddings are used to train pipeline-agnostic meta-models for pipeline performance estimation.

To evaluate our methodology, we created *MetaExe-Bench*, a large-scale performance benchmark containing 144,177 scikit-learn pipeline evaluations. These cover 2,616 end-to-end classification and regression pipeline configurations executed sparsely across 170 OpenML datasets. To the best of our knowledge, this is the first benchmark of its kind at this scale for existing OpenML datasets and pipelines. The evaluation of *KGmetaSP* was conducted in two settings. First, we assess the effectiveness of the learned embeddings for *PPE* by training a pipeline-agnostic meta-model and comparing it against commonly used specialized models across multiple prediction metrics, considering both unseen datasets and unseen pipeline configurations. Second, we evaluate *DPSE* by using distances between learned dataset embeddings to retrieve and rank performance-similar datasets, and we compare the results against prominent meta-feature-based baselines. The results show that *KGmetaSP* enables accurate *PPE* with a single meta-model that jointly learns to predict the performance of multiple pipeline configurations, including unseen ones. In addition, the learned dataset embeddings improve *DPSE*.

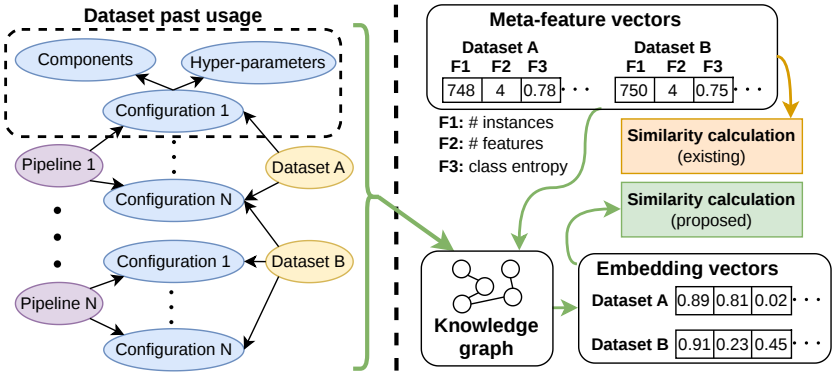


FIGURE 7.1: Comparison of the information leveraged by our proposed approach (green) versus existing approaches (orange) for representing datasets in meta-learning. For a pair of datasets, A and B, existing approaches represent datasets using only intrinsic meta-feature vectors (top-right). Our proposed approach also includes past pipeline-configuration meta-data (top left) as additional context into a KG, to represent datasets.

7.2 Related Work

This section surveys prior research relevant to *PPE* and *DPSE*, and then discusses semantic and graph-based representations for ML knowledge.

Pipeline Performance Estimation (*PPE*). Estimating how well a pipeline will perform on a target dataset is a long-standing problem in meta-learning [88]. A common solution is to train predictive meta-models on previous experimental results, using dataset meta-features as the main input representation [29]. In many cases, one predictor is trained per pipeline configuration, so that the model learns to estimate the expected performance of that specific configuration on previously unseen datasets [14]. Such methods are useful for warm-starting optimization [107]. Their main limitation, however, is that they are tied to the configurations observed during training and therefore cannot naturally handle unseen pipelines. More recent approaches, such as DeepPipe [113], address this issue by learning representations of pipeline structure. Still, these methods focus primarily on the pipelines themselves and do not explicitly incorporate historical pipeline usage or dataset descriptions. Our approach jointly learns representations for datasets and pipelines, making it possible to address *PPE* both for unseen datasets and pipeline configurations.

Dataset Performance-based Similarity Estimation (*DPSE*). Determining which datasets behave similarly with respect to pipeline performance is important in many meta-learning scenarios [139]. One representative example is the work of Feurer et al. [46], where similarity is defined through the behavior of different pipeline configurations across datasets. Most existing approaches estimate such similarity through dataset meta-features [30, 71]. These representations may capture basic descriptive properties, information-theoretic characteristics, statistical properties, or landmarker-based information derived from the performance of simple ML models [100, 11]. Another direction is to replace manually designed descriptors with learned representations obtained from neural models trained on whole datasets [70, 75]. Although these methods can produce useful dataset representations, they ignore valuable past experiment data. Even landmarker-based approaches incorporate only limited performance information and assume that datasets are compared under the same pipeline configurations. By embedding both datasets and pipelines in a common latent space, our method aims to compare datasets even when the evaluated pipeline configurations differ.

Performance Benchmarks. A number of benchmarking initiatives have been proposed to study model and pipeline behavior across datasets. PMLB [106], for example, collects 165 classification datasets from sources such as UCI and KEEL and evaluates a set of 13 models. AMLB [55] benchmarks 9 AutoML systems on a collection of 71 classification and 33 regression datasets, most of them obtained from OpenML. Alcobaça et al. [4] also provide large-scale meta-knowledge derived from pipeline configuration variations evaluated on 211

OpenML datasets. Despite their usefulness, such resources either depend on artificially generated pipeline variants or cover only a small portion of pipelines for each dataset. In our work, we rely on scikit-learn pipeline configurations already published on the web through OpenML and use their cross-dataset evaluations as the basis for meta-learning analysis.

Ontologies and Graphs for ML Knowledge Representation. KGs and ontologies have increasingly been used to organize and describe ML-related artifacts. Examples include HuggingKG [31], MLTaskKG [91], DCAT [94], ML-Schema [119], and MLSO [36]. MLSea-KG [36] applies MLSO to integrate ML knowledge from OpenML, Kaggle, and Papers with Code. It represents dataset metadata, ML tasks, pipeline descriptions, experiment runs, configuration settings, evaluation results, implementations, algorithms, code notebooks, repositories, publications, models, and related agents. However, its pipeline representation is mainly experiment-oriented: it links datasets, tasks, runs, implementations, settings, and results, but does not model the executable structure of a pipeline in terms of connected processing components. This fine-grained view is provided by *ExeKGs*, introduced in Chapter 4, which represent ML pipelines as executable graphs of components, parameters, and dataflow connections. The KG introduced in this chapter builds on the combination of these two complementary representations to cover both dataset-level semantics and detailed pipeline structure.

7.3 Problem Description

This section defines the setting considered in this chapter and formalizes the two meta-learning tasks addressed by our approach.

Context. Let $\mathcal{D}$ denote a set of datasets. Each dataset $d \in \mathcal{D}$ is associated with a description C_d, representing its characteristics, such as meta-features. Let $\mathcal{P}$ denote a set of ML pipeline configurations. Each pipeline configuration $p \in \mathcal{P}$ is described by a set of characteristics C_p, including, for example, preprocessing steps, predictive models, and hyper-parameter settings. We assume that each pipeline configuration has been evaluated on at least one dataset in $\mathcal{D}$. Whenever a pipeline configuration $p_j \in \mathcal{P}$ is evaluated on a dataset $d_i \in \mathcal{D}$, it yields a performance value $v_{i,j}$.

Tasks. We focus on two tasks that support the reduction of the pipeline search space in meta-learning. For *PPE*, the objective is to estimate the performance value of a pipeline configuration p_j on a dataset d_i. In other words, given a dataset–pipeline pair (d_i, p_j), the task is to predict the corresponding value $v_{i,j}$. For *DPSE*, the objective is to identify, for a target dataset $d_i \in \mathcal{D}$, another dataset $d_j \in \mathcal{D}$ with $j \neq i$ whose behavior across pipeline configurations is the most similar. To express this more precisely, let $P_i = \{p \in \mathcal{P} \mid p \text{ was evaluated on } d_i\}$ be the set of pipeline configurations evaluated on d_i. The performance profile of d_i is then defined as the vector of observed performance values over these configurations: $V_i = [v_{i,1}, v_{i,2}, \ldots, v_{i,|P_i|}]$. The goal of *DPSE* is to retrieve datasets whose performance profiles are most similar to that of the target dataset.

Formalization. We formalize *PPE* as a prediction function f that takes as input the characteristics of a dataset d_i and a pipeline configuration p_j and outputs an estimated performance value $\hat{v}_{i,j}$: $\hat{v}_{i,j} = f(C_{d_i}, C_{p_j})$, where the objective is for $\hat{v}_{i,j}$ to approximate $v_{i,j}$ as closely as possible, *i.e.*, the difference $|v_{i,j} - \hat{v}_{i,j}|$ should be minimal. For *DPSE*, we define a similarity function $sim(d_i, d_j)$ that measures the similarity between two datasets d_i and d_j based on C_d and C_p. Given a target dataset d_i, the retrieval task consists of finding the dataset d_j^* with maximum similarity: $d_j^* = \underset{d_j \in \mathcal{D}, j \neq i}{\arg\max}\, sim(d_i, d_j)$.

7.4 Methodology

This chapter proposes *KGmetaSP*, a meta-learning approach built on KGEs (Fig. 7.2). The central idea is to represent datasets, pipeline configurations, and their internal elements within a common graph structure, and then learn vector representations that reflect their

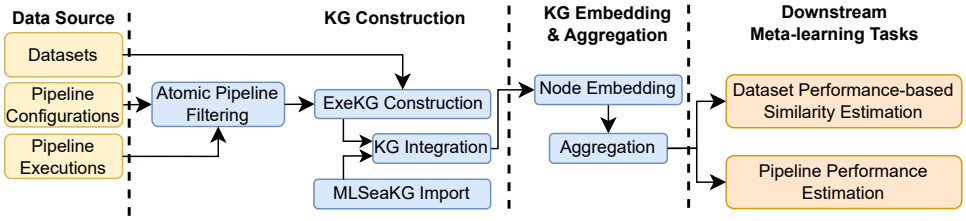


FIGURE 7.2: *KGmetaSP* overview.

relational context. In this way, entities that participate in similar graph neighborhoods are mapped to nearby points in the embedding space, enabling similarity-based reasoning for downstream meta-learning tasks.

To support this idea, we first construct a unified KG, denoted as *MetaExe-KG*, that brings together information about datasets and pipeline configurations. The graph is obtained by combining *MLSeaKG* [36], which provides semantic and statistical descriptions of ML datasets, with *ExeKGs* derived from OpenML experiments, which capture the structure of pipeline configurations together with their components and hyper-parameters.

Once *MetaExe-KG* has been constructed, we embed it into a latent vector space in order to obtain joint representations of datasets and pipelines. These representations are then used for the two tasks considered in this chapter, namely *PPE* and *DPSE*. More specifically, pipeline embeddings are used as input to pipeline-agnostic meta-models for *PPE*, while dataset embeddings are compared through cosine similarity for *DPSE*. Additional implementation details and an analysis of computational cost are provided in the Appendix.

7.4.1 *MetaExe-KG*: A Knowledge Graph of Datasets and Pipelines

A key requirement of the proposed approach is a KG that captures both descriptive information about datasets and fine-grained structural information about ML pipelines. On the dataset side, MLSO, as used in *MLSeaKG* [36], offers a detailed semantic description of dataset metadata and meta-features. However, it does not explicitly represent the internal composition of pipelines, such as their operators, components, and hyper-parameters. To model these aspects, we align *MLSeaKG* with the ontology underlying *ExeKGs*, which provides a modular representation of ML pipeline configurations. The resulting integrated schema forms the basis of *MetaExe-KG* and is illustrated in Fig. 7.3.

We instantiate this schema using datasets and pipeline configurations retrieved from OpenML. In particular, through the OpenML API[1], we collect a set of datasets $\mathcal{D}$. For each dataset $d_i \in \mathcal{D}$, we retrieve the top 10 configurations for every pipeline that has been evaluated on d_i. Restricting the graph to top-performing configurations serves three purposes: it reduces the influence of poor-quality runs, keeps the resulting KG at a manageable size, and preserves the most informative performance patterns for representation learning. We focus on scikit-learn pipelines so that components and hyper-parameters can be represented consistently across configurations. To ensure that the selected configurations correspond to executable end-to-end pipelines for their target datasets, we developed a filtering module (Fig. 7.2) that validates the recorded OpenML evaluations and removes configurations that are not atomic, *i.e.*, those that depend on additional preprocessing outside the recorded pipeline. After this filtering stage, we model datasets and pipeline configurations using MLSO and *ExeKGs*, including the individual pipeline steps and hyper-parameter settings shown in Fig. 4.5. The resulting graph links each dataset to the pipeline configurations that achieved the best recorded performance on it.

For every OpenML dataset, its features and target variable are mapped to the corresponding semantic elements in *ExeKG* (Fig. 4.5). For each associated pipeline configuration, we parse the full sequence of pipeline steps together with their hyper-parameters. We create an *ExeKG* `DataEntity` node for each dataset variable, an *ExeKG* `Task` node for each pipeline stage, such as splitting, transformation, training, or testing, and an *ExeKG* `Method` node for

[1] https://www.openml.org/apis

the concrete implementation of that stage. In this way, each pipeline configuration is represented as a separate *ExeKG* graph.

These pipeline configuration graphs are then connected with *MLSeaKG*, which contributes richer dataset metadata, including descriptions, keywords, and meta-features. The result is *MetaExe-KG*, a unified graph that combines execution-oriented pipeline structure with detailed dataset characterization. In its current form, *MetaExe-KG* comprises 170 OpenML datasets and 2,616 OpenML pipeline configurations for classification and regression, with an average of 15 steps per pipeline. Overall, the graph contains approximately 4.5 million triples.

7.4.2 *KGmetaSP*: Knowledge Graph Embedding-Based Meta-Learning

After constructing *MetaExe-KG*, we learn vector representations for its nodes using both walk-based and LP-based KGE models. For the walk-based setting, we use *RDF2Vec* [124] and rely on *MKGA* [116] to incorporate numeric attributes through binning. For the LP-based setting, we evaluate ComplEx [144], DistMult [159], and TransE [22], using `ReaLitE` to incorporate numeric literals. From these node-level embeddings, we derive one vector representation for each pipeline configuration and one for each dataset. For datasets, we investigate three alternative aggregation schemes. These learned representations are then used in the downstream tasks of *PPE* and *DPSE*.

Embedding aggregation. For each dataset, we derive two representations based on different subsets of nodes, as well as a third representation that combines them (Fig. 7.4). The first, denoted $\mathbf{DE_{var}}$, is obtained by averaging the embeddings of the `DataEntity` nodes corresponding to the dataset variables. The second, denoted $\mathbf{DE_{pip}}$, is obtained by averaging embeddings associated with the pipeline configurations linked to that dataset. More specifically, each pipeline configuration is first represented by averaging the embeddings of its `Method` nodes, yielding a pipeline embedding $\mathbf{PE}$. The dataset-level representation $\mathbf{DE_{pip}}$ is then computed as the average of the pipeline embeddings associated with the dataset. Finally, the third representation, $\mathbf{DE_{comb}}$, is defined as the average of $\mathbf{DE_{var}}$ and $\mathbf{DE_{pip}}$. These aggregation choices are motivated by the two main sources of information that influence performance behavior. On the one hand, variable-level nodes reflect intrinsic dataset properties [24, 71]. On the other hand, method-level nodes capture how the dataset has interacted with modeling components in past experiments [46, 86]. In contrast, `Hyper-parameter` nodes mainly describe execution-specific settings, while `Task` nodes indicate processing stages rather than the concrete operations applied. The combined representation $\mathbf{DE_{comb}}$ aims to merge these complementary views, following the general intuition behind meta-embedding approaches [33]. Formally, we define:

$$\mathbf{DE_{var}}(D_i) = \frac{1}{f}\sum_{j=1}^{f}\mathbf{E}(\mathtt{de}_j), \qquad \mathbf{PE}(P_l) = \frac{1}{s_l}\sum_{k=1}^{s_l}\mathbf{E}(M_{lk}),$$

$$\mathbf{DE_{pip}}(D_i) = \frac{1}{p}\sum_{l=1}^{p}\mathbf{PE}(P_l), \qquad \mathbf{DE_{comb}}(D_i) = \frac{1}{2}(\mathbf{DE_{var}}(D_i) + \mathbf{DE_{pip}}(D_i))$$

where $\mathbf{E}(\mathtt{de}_j)$ denotes the embedding of the j-th `DataEntity` node of dataset D_i, f is the number of `DataEntity` nodes of D_i, $\mathbf{E}(M_{lk})$ denotes the embedding of the k-th `Method` node in the l-th pipeline configuration associated with D_i, s_l is the number of `Method` nodes in pipeline configuration P_l, and p is the number of pipeline configurations associated with D_i.

KGEs for *PPE* and *DPSE*. For *PPE*, the learned pipeline embeddings $\mathbf{PE}(P_l)$ are combined with dataset meta-features and used as input to pipeline-agnostic meta-models (Fig. 7.5). Because the model is not tied to one specific configuration, it can be trained to predict performance across multiple pipeline configurations within a single predictive framework. For *DPSE*, the similarity between two datasets D_1 and D_2 is approximated through the cosine similarity between their embedding representations, denoted $\mathbf{DE}_*^{(1)}$ and $\mathbf{DE}_*^{(2)}$, where $*$ specifies the aggregation variant used, *i.e.*, $\mathbf{DE_{var}}$, $\mathbf{DE_{pip}}$, or $\mathbf{DE_{comb}}$.

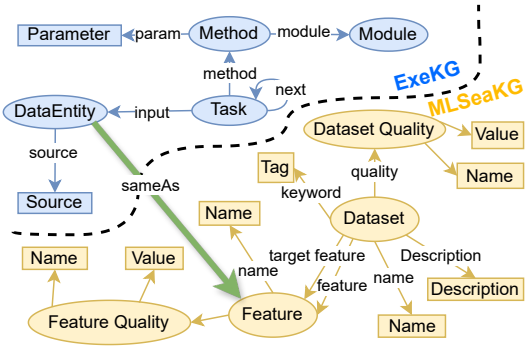


FIGURE 7.3: High-level view of *MetaExe-KG*.

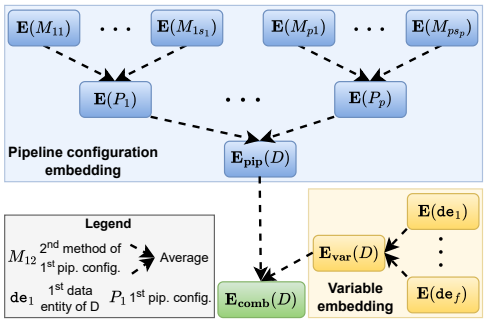


FIGURE 7.4: Embedding aggregation process.

7.5 Experiments

This section evaluates the proposed approach on the two tasks considered in this chapter, namely *PPE* and *DPSE*. To support this evaluation, we construct *MetaExe-Bench*, a benchmark with ground-truth performance values obtained by executing pipeline configurations from *MetaExe-KG* across datasets contained in *MetaExe-KG*. The resulting benchmark is used both to train predictive models for *PPE* and to derive reference similarity values for *DPSE*.

7.5.1 Experimental Setup

Data Source and *MetaExe-Bench*. Table 7.1 reports the statistics of the data source used in our experiments, both with and without the integration of *MLSeaKG*. This distinction allows us to later assess the contribution of semantic enrichment. To prevent information leakage, performance values collected from OpenML are removed from *MetaExe-KG* before learning embeddings. Otherwise, the embedding process could encode target performance directly, rather than capturing structural and semantic regularities.

To obtain an evaluation benchmark, we created *MetaExe-Bench* by sparsely executing the 2,616 pipeline configurations in *MetaExe-KG* on the 170 datasets represented in *MetaExe-KG*. For this purpose, OpenML pipeline components and hyper-parameter metadata were mapped to scikit-learn `Pipeline` objects. Since the OpenML records span multiple scikit-learn versions, deprecated parameters and outdated components were translated to their counterparts in scikit-learn v1.5.1. This process yielded 144,177 distinct evaluations involving existing OpenML datasets and pipeline configurations.

For each dataset $d_i \in D$, we evaluate the subset of pipeline configurations P that are compatible with its ML goal, such as classification or regression. If a pipeline configuration p_j cannot be fitted on d_i, the corresponding experiment is treated as invalid and the resulting performance value $V_{i,j}$ is set to the worst possible value for the metric under consideration. In addition, we record fit time and penalize configurations whose execution time is excessively larger than in their original OpenML execution. In the current benchmark version, a configuration is considered invalid when its fit time is at least ten times larger than the original one. In total, 300,543 dataset–pipeline configuration pairs were found to be invalid.

For *PPE*, the recorded performance value of each dataset–pipeline pair serves as the ground-truth target. For *DPSE*, ground-truth similarity between two datasets is derived from their performance profiles over the set of pipeline configurations that were successfully evaluated on both datasets. Following prior work [86], we use cosine similarity between these performance vectors as the reference measure of performance-based similarity.

Pipeline Performance Estimation (*PPE*) Evaluation. To evaluate *PPE*, we use the learned representations of *KGmetaSP* within a single pipeline-agnostic meta-model that predicts the performance of arbitrary dataset–pipeline pairs. For the dataset representation, we use the dataset's meta-feature vector. For the pipeline representation, we evaluate both walk-based *RDF2Vec* embeddings and LP-based embeddings, namely ComplEx, DistMult, and TransE, where the LP-based variants use `ReaLitE` to incorporate numeric literals. Although the

TABLE 7.1: *MetaExe-KG* statistics.

	Entities	Properties	Triples	Attributes	Literals
ExeKGs	1,502,140	293	4,492,036	149	371,534
ExeKGs +MLSea	1,514,836	317	4,522,216	155	385,246

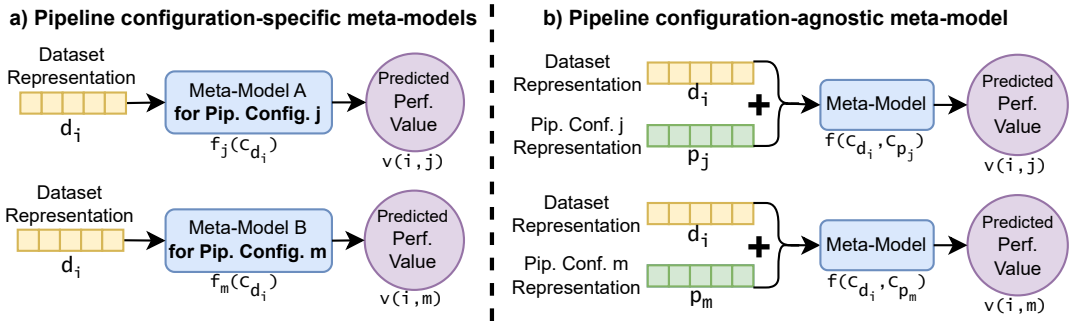


FIGURE 7.5: High-level view of the *PPE* evaluation setting. (a) Standard approach: one configuration-specific meta-model predicts the performance of a single pipeline configuration from a dataset representation. (b) Proposed approach: a single configuration-agnostic meta-model predicts performance from both dataset and pipeline representations.

framework is compatible with different predictive meta-models, in the experiments we use random forest predictors, which are widely adopted in meta-learning [29]. We focus on classification pipeline configurations and use the accuracy and precision values recorded in *MetaExe-Bench* as meta-targets. Performance is evaluated both as a meta-regression problem, where the target is the raw metric value, and as a meta-classification problem, where the target is discretized into three quantile-based bins corresponding to low, medium, and high performance.

We consider two evaluation scenarios: unseen datasets and unseen pipeline configurations. For **unseen datasets**, the task is to predict the performance of known pipeline configurations on datasets not observed during meta-model training. We retain pipeline configurations that appear in at least 50 training samples, resulting in 123,920 dataset–pipeline pairs drawn from *MetaExe-Bench* and covering 1,028 pipeline configurations. In this setting, our pipeline-agnostic meta-model, which takes as input the concatenation of dataset meta-features and pipeline embeddings, is compared against 1,028 configuration-specific meta-models trained only on dataset meta-features (Fig. 7.5). To test robustness with respect to the dataset representation, we consider several groups of meta-features:

- **MF All:** the full set of simple, statistical, information-theoretic, and landmarker meta-features.
- **MF Statistical:** meta-features describing distributional properties, such as measures of central tendency and dispersion [100].
- **MF Information Theory:** meta-features derived from entropy, mutual information, and related notions [100].
- **MF Landmarkers:** meta-features based on the performance of simple reference learners, such as Decision Tree and Naive Bayes [11].

For **unseen pipeline configurations**, the task is to predict the performance of pipeline configurations not encountered during meta-model training. This setting cannot be handled by configuration-specific meta-models, since each such model is tied to a single known configuration (Fig. 7.5). We therefore compare our method against two generalizable and pipeline-agnostic baselines. The first, *Average performance*, is a constant baseline that predicts the mean performance over the training data. The second, *Closest embedding*, is a retrieval-based baseline that predicts performance by using the nearest observed pipeline in the embedding space. Together, these baselines provide lower and stronger reference points for assessing how well *KGmetaSP* generalizes to unseen configurations.

Dataset Performance-based Similarity Estimation (*DPSE*) Evaluation. For *DPSE*, we first compute a ground-truth similarity score for every dataset pair using *MetaExe-Bench*, based on cosine similarity between their performance profiles. The rankings produced by our method are then compared to this reference using *Hit@K* and *NDCG@K*. In the case of *Hit@K*, a query is counted as successful when at least one of the top-*K* retrieved datasets has a ground-truth similarity above 0.9 with the query dataset.

We compare our method against a set of non-KGE baselines drawn from the literature:

- **Graph Edit Distance (GED).** We use GED [122] to compare two datasets through the structural similarity of their associated pipeline configurations. For datasets d_1 and d_2, with corresponding pipeline sets P_1 and P_2, similarity is defined as $\text{Sim}_{GED}(d_1, d_2) = -\frac{1}{|P_1| \cdot |P_2|} \sum_{p_i \in P_1} \sum_{p_j \in P_2} GED(p_i, p_j)$.
- **Meta-feature P-Norm Distance Ranking (MF).** Following prior work [46], we estimate similarity from the distance between meta-feature vectors, using the same meta-feature set as in *MLSeaKG*, with and without normalization. For datasets d_1 and d_2, with meta-feature vectors m_1 and m_2, similarity is computed as $\text{Sim}_{MF}(d_1, d_2) = -||m_1 - m_2||_p$.
- **Single-field Document Ranking: Dataset Meta-features (SiFi-Dat.).** Each dataset is represented as a single textual document containing OpenML metadata such as title, description, and variable names. Similarity is then estimated using cosine similarity between pre-trained bi-encoder language model embeddings [123].
- **Single-field Document Ranking: Pipeline Description (SiFi-Pip.).** For each dataset, the textual descriptions of its top-performing pipeline configurations are merged into a single document, which is embedded with a pre-trained bi-encoder and ranked by cosine similarity [123].
- **Multi-field Document Ranking (MuFi-Dat.).** Each dataset is represented as a fielded document [112, 161] with separate metadata fields such as title, description, variable names, and tags. Similarity is estimated using a mixture of language models [105], with empirically chosen field weights, together with cosine similarity over the resulting embeddings.

7.5.2 Evaluation Results

Evaluation Results for *PPE*. For the **unseen datasets** setting, the results are shown in Table 7.2. *KGmetaSP*$_{\text{PE}}^{\text{ReaLitE}}$ denotes the best LP-based variant among ComplEx, DistMult, and TransE according to meta-classification F1. Full results are reported in Appendix C. The table reports the best LP-based *KGmetaSP*$_{\text{PE}}$ variant among ComplEx, DistMult, and TransE as *KGmetaSP*$_{\text{PE}}^{\text{ReaLitE}}$, since all LP-based variants use `ReaLitE` for numeric literals. The full breakdown by LP model is provided in Appendix C. The meta-regression results show that the configuration-specific meta-models generally achieve lower MSE and higher R^2. Even so, the proposed configuration-agnostic model remains competitive. The picture changes in the more practically relevant meta-classification setting, where the proposed model consistently outperforms the average of the 1,028 specialized models across all tested dataset representations. For MF All, *KGmetaSP*$_{\text{PE}}$ with *RDF2Vec* is the strongest configuration-agnostic variant and achieves the best overall meta-classification results, reaching 0.7413 accuracy for accuracy prediction and 0.7537 accuracy for precision prediction. The results show that LP-based literal-aware embeddings are particularly useful when restricted meta-feature groups are used, while *RDF2Vec* is strongest when combined with the full meta-feature representation. MF All includes the simple meta-features represented in the KG. So, the numeric information in the dataset representation is complemented by the *RDF2Vec* graph-walk context. In contrast, those simple meta-features are absent from MF Statistical, MF Information Theory, and MF Landmarkers. Therefore, the *KGmetaSP*$_{\text{PE}}$ variants using `ReaLitE` can better compensate for this missing information than the *RDF2Vec* variant with *MKGA*.

In the **unseen pipeline configuration** setting, the proposed method clearly outperforms both the *Average performance* and *Closest embedding* baselines (Table 7.3). This holds for both meta-regression and meta-classification targets. *KGmetaSP*$_{\text{PE}}$ with *RDF2Vec* remains the best-performing variant in this setting. The results indicate that the learned representations capture enough structural information about pipelines to support reliable generalization, even when the pipeline configuration was not present during meta-model training.

TABLE 7.2: *PPE* meta-classification results for unseen datasets. Best overall scores are underlined, and best scores per meta-feature group are bold.

Dataset	Pipeline	Meta-	Meta-Regr.		Meta-Classif.	
Embedding	Strategy	Models	MSE	R2	Accuracy	F1
Target: Accuracy						
MF All	Conf.-specific	1,028	**0.0081**	**0.6748**	0.7363	0.7358
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0103	0.6104	0.7380	0.7393
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0101	0.6181	**0.7413**	**0.7427**
MF Info. Theory	Conf.-specific	1,028	**0.0249**	**-0.0737**	0.4428	0.4423
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0399	-0.5069	**0.4911**	**0.4938**
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0401	-0.5135	0.4773	0.4789
MF Landmarkers	Conf.-specific	1,028	**0.0092**	**0.6428**	0.7152	0.7139
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0107	0.5977	**0.7212**	**0.7222**
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0104	0.6094	0.7169	0.7172
MF Statistical	Conf.-specific	1,028	**0.0218**	**0.0892**	0.5499	0.5484
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0254	0.0404	**0.6077**	**0.6054**
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0259	0.0212	0.5827	0.5830
Target: Precision						
MF All	Conf.-specific	1,028	**0.0133**	**0.6311**	0.7347	0.7400
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0173	0.5198	0.7461	0.7482
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0164	0.5449	**0.7537**	**0.7562**
MF Info. Theory	Conf.-specific	1,028	**0.0330**	**-0.0933**	0.4324	0.4350
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0507	-0.4094	0.4902	0.4941
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0506	-0.4059	**0.4961**	**0.5007**
MF Landmarkers	Conf.-specific	1,028	**0.0148**	**0.6069**	0.6987	0.7006
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0183	0.4921	**0.7288**	**0.7318**
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0184	0.4889	0.7247	0.7269
MF Statistical	Conf.-specific	1,028	**0.0274**	**0.1221**	0.5490	0.5487
	$KGmetaSP_{\text{PE}}^{\text{REALITE}}$	1	0.0345	0.0416	**0.5932**	**0.5928**
	$KGmetaSP_{\text{PE}}^{\text{RDF2VEC}}$	1	0.0346	0.0381	0.5897	0.5896

Evaluation Results for *DPSE*. Tables 7.4 and 7.5 summarize the retrieval results for *DPSE*. Overall, the three embedding aggregation variants of *KGmetaSP* perform strongly. The combined representation ($KGmetaSP_{DE_{comb}}$) achieves the best results for Hit@2, Hit@5, and average Hit, and it ties for the highest Hit@1. This suggests that combining variable-level and pipeline-level information is especially beneficial for top-k retrieval. The pipeline-based representation ($KGmetaSP_{DE_{pip}}$) yields the strongest NDCG@1 and the second-best average NDCG, whereas the variable-based representation ($KGmetaSP_{DE_{var}}$) remains competitive across both metrics. Among the non-KGE baselines, SiFi-Dat. performs particularly well, especially on NDCG, indicating that textual metadata from OpenML also provides a useful signal for identifying performance-similar datasets.

TABLE 7.3: *PPE* results for unseen pipeline configurations. For each target, *KGmetaSP*$_{\text{PE}}^{\text{REALITE}}$ denotes the LP-based variant among ComplEx, DistMult, and TransE with the highest meta-classification F1. *KGmetaSP*$_{\text{PE}}^{\text{RDF2VEC}}$ denotes the walk-based *RDF2Vec* variant. Best scores are underlined.

Method	**Target: Accuracy**				**Target: Precision**			
	MSE	**R2**	**Acc.**	**F1**	**MSE**	**R2**	**Acc.**	**F1**
Average performance	0.0267	-0.0005	0.3303	0.1640	0.0352	-0.0012	0.3323	0.1658
Closest embedding	0.0127	0.5241	0.7748	0.7747	0.0225	0.3607	0.7773	0.7771
MF All + *KGmetaSP*$_{\text{PE}}^{\text{REALITE}}$	0.0080	0.7007	0.8076	0.8068	0.0148	0.5804	0.8089	0.8083
MF All + *KGmetaSP*$_{\text{PE}}^{\text{RDF2VEC}}$	**0.0070**	**0.7361**	**0.8250**	**0.8244**	**0.0131**	**0.6264**	**0.8251**	**0.8247**

TABLE 7.4: Hit@k of similarity measures according to performance-based ground truth. A hit is recorded when a dataset retrieved within the top-k has a ground-truth similarity with the query dataset greater than a similarity threshold (ST). Results are reported for $ST = 0.8$ and $ST = 0.9$. The highest score per metric is in bold and the second highest is underlined.

	Hit@k (ST=0.8)				**Hit@k** (ST=0.9)			
Method	**k=1**	**k=2**	**k=5**	**Avg.**	**k=1**	**k=2**	**k=5**	**Avg.**
GED [122]	0.5743	0.8243	0.9054	0.7680	0.4527	0.7432	0.8514	0.6824
MF [46]	0.6892	0.8243	0.8851	0.7996	0.6622	0.8041	0.8716	0.7793
MF Norm. [46]	0.6892	0.8851	0.9324	0.8356	0.7905	0.8446	0.9122	0.8491
SiFi-Dat. [27, 161]	0.8514	0.8851	0.9054	0.8806	**0.7973**	0.8649	0.8919	0.8514
SiFi-Pip. [27, 161]	0.6351	0.8176	0.9527	0.8018	0.6216	0.7568	0.9324	0.7703
MuFi-Dat. [112, 161]	0.7905	0.8851	0.8987	0.8581	0.7635	0.8378	0.8784	0.8266
KGmetaSP$_{DE_{var}}$	0.7973	0.8716	0.9324	0.8671	0.7365	0.8581	0.9324	0.8423
KGmetaSP$_{DE_{pip}}$	**0.8581**	**0.9122**	0.9460	**0.9054**	0.7905	0.8649	0.9257	0.8604
KGmetaSP$_{DE_{comb}}$	0.8176	**0.9122**	**0.9662**	0.8987	**0.7973**	**0.8986**	**0.9459**	**0.8806**

TABLE 7.5: NDCG@k of similarity measures according to performance-based ground truth. The highest score per metric is in bold and the second highest is underlined.

	NDCG@k			
Method	**k=1**	**k=2**	**k=5**	**Avg.**
GED [122]	0.5684	0.6308	0.6847	0.6280
MF [46]	0.6939	0.6874	0.7016	0.6943
MF Norm. [46]	0.8561	0.8418	0.8358	0.8446
SiFi-Dat. [27, 161]	0.8753	**0.8768**	**0.8693**	**0.8738**
SiFi-Pip. [27, 161]	0.8255	0.8073	0.8248	0.8192
MuFi-Dat. [112, 161]	0.8229	0.8338	0.8398	0.8322
KGmetaSP$_{DE_{var}}$	0.8010	0.8176	0.8136	0.8107
KGmetaSP$_{DE_{pip}}$	**0.8811**	0.8642	0.8541	0.8665
KGmetaSP$_{DE_{comb}}$	0.8436	0.8562	0.8388	0.8462

7.6 Conclusion

This chapter presented *KGmetaSP*, an approach that uses KGEs for *PPE* and *DPSE* by representing datasets, pipeline configurations, and their relationships in a unified KG. The results show that *KGmetaSP* supports *PPE* with a single pipeline-agnostic meta-model that generalizes to both unseen datasets and unseen pipeline configurations. For *DPSE*, the learned KG embeddings provide effective representations for retrieving performance-similar datasets.

These findings highlight the value of incorporating pipeline metadata and open experiment data as structured KG context for meta-learning. The approach is model-agnostic and can, in principle, be adapted to different dataset modalities, pipeline types, and meta-learning objectives.

Part IV

Conclusion and Outlook

Chapter 8

Summary

The rapid and continuous expansion of ML has created a dual challenge: the increasing complexity of developing effective pipelines has raised the barrier to entry for non-experts, while the vast landscape of ML artifacts has made the discovery of optimal pipelines and relevant datasets a challenging meta-learning task. This thesis addressed these interconnected challenges of ML democratization and discovery through a unified, semantics-driven approach.

The foundational contribution of this work is the *ExeKG*, a novel semantic basis for representing, validating, and executing ML pipelines using simple terms accessible to non-ML experts. By establishing a hierarchical suite of ontologies, the *ExeKG* provides a transparent and interoperable representation of ML pipelines. This semantic structure ensures that pipelines are not only understandable and reusable but also verifiable through SHACL constraints, guaranteeing their executability. To translate this formal representation into practical tools, this thesis introduced two software contributions: *ExeKGLib*, a Python library that provides the core engine for programmatically creating and executing *ExeKGs*, and *ExeKGLab*, a no-code GUI that makes pipeline development accessible to domain experts.

To address the challenge of ML discovery, this thesis proposes *KGmetaSP*, a meta-learning approach for *PPE* and *DPSE* that uses *ExeKGs* to integrate past experiment data into *MetaExe-KG* and exploit latent relationships between datasets and pipeline configurations through KGEs. Pipeline embeddings are used to train pipeline-agnostic meta-models for *PPE*, while dataset embeddings are compared through vector distance measures for *DPSE*. Since this setting contains rich numeric information, such as hyper-parameters, this thesis also introduces `ReaLitE`, a relation-centric KGE model that incorporates numeric literals into relation embeddings. On link prediction and node classification benchmarks, `ReaLitE` shows strong performance, particularly for infrequent relations and relations between entities with correlated numeric values, while its use within *KGmetaSP* improves *PPE* performance. *KGmetaSP* is evaluated using *MetaExe-Bench*, a large-scale benchmark comprising 144,177 scikit-learn pipeline evaluations across 2,616 end-to-end classification and regression pipeline configurations and 170 OpenML datasets. The results show that *KGmetaSP* enables accurate *PPE* with a single pipeline-agnostic meta-model that generalizes to both unseen datasets and unseen pipeline configurations, while the learned KGEs provide effective dataset representations for *DPSE*.

Together, these contributions advance the state of the art in semantic ML systems. They provide a foundational representation for executable ML pipelines, motivate and develop embedding methods for semantic ML graphs with numeric information, and demonstrate how structured experiment knowledge can support meta-learning.

Chapter 9

Limitations and Future Work

This chapter reflects on the limitations of the research presented in this thesis and outlines promising future directions.

9.1 Limitations

While the research presented has achieved its primary objectives, it is important to acknowledge its limitations, which in turn open avenues for future work.

Scope of the *ExeKG* Ontologies. The current lower-level *ExeKG* ontologies are primarily focused on operations found within the scikit-learn library and are tailored for non-neural classification and regression algorithms. This limits our framework's immediate applicability to other popular ML libraries, such as TensorFlow or PyTorch, and other ML fields, such as natural language processing, computer vision, or reinforcement learning.

Focus of the `ReaLitE` Model. The `ReaLitE` model was designed to capture the relationships between numeric literals. Its architecture does not explicitly model other important literal types, such as textual descriptions or temporal information, in the same integrated, relation-centric manner.

Scope and Dynamicity of *KGmetaSP*. The current implementation of *KGmetaSP* focuses on scikit-learn pipelines from OpenML to ensure a consistent representation of pipeline components and hyper-parameters. As a result, extending the approach to other ML libraries or experiment platforms would require mapping their pipeline representations to the *ExeKG* ontology. In addition, the embeddings used in *KGmetaSP* are static. Therefore, when new datasets, pipeline configurations, or experiment records are added, the KG must be updated and the embeddings must either be retrofitted or learned again.

9.2 Future Work

The limitations of this work reveal two directions for future research. First, the individual contributions can be extended by broadening the scope of *ExeKGs*, `ReaLitE`, and *KGmetaSP*. Second, these contributions can be integrated more tightly by using *KGmetaSP* to guide the LLM assistant in *ExeKGLab* toward empirically grounded pipeline recommendations.

9.2.1 Extensions of the Thesis Contributions

Extending the semantic framework. A primary avenue for future work is the expansion of the *ExeKG* ontology suite. New lower-level ontologies could be developed to formally represent operations from deep learning frameworks, NLP libraries, and other domains, thus broadening the applicability of the proposed representation. In addition, more sophisticated SHACL shapes could be designed to enforce deeper semantic constraints, ensuring not only executability but also logical and methodological soundness.

Extending literal-aware representation learning. Another direction is to extend `ReaLitE` beyond numeric literals. Future work could investigate how textual descriptions, temporal information, or categorical literal values can be incorporated into relation-centric KGE

TABLE 9.1: Preliminary results for *KGmetaSP*-guided LLM pipeline generation on six unseen OpenML classification datasets.

Prompt setting	Mean Accuracy	Mean F1
LLM zero-shot	0.8779	0.8758
LLM + *KGmetaSP* top-1 pipeline	0.9027	0.8966
LLM + *KGmetaSP* top-3 pipelines	0.9036	0.8984

models. This would be particularly relevant for semantic ML graphs, where dataset descriptions, algorithm documentation, version information, and time-dependent experiment metadata may provide additional signals for downstream tasks.

Dynamic embeddings for evolving KGs. Another important direction is for *KGmetaSP* to support dynamic updates of embeddings as new datasets, pipeline configurations, and experiment records become available. Future work could investigate incremental KGE methods, including recent work on incremental RDF2Vec [58], to keep *MetaExe-KG* up to date while avoiding costly retraining from scratch.

Broader applications of executable KGs. Beyond the ML setting considered in this thesis, the core concept of an *ExeKG* could be adapted to data-intensive scientific domains, such as bioinformatics, computational chemistry, or physics simulations. In these fields, executable KG representations could support standard, reproducible, and machine-readable descriptions of scientific workflows, thereby fostering greater transparency and collaboration.

9.2.2 Guiding LLM-based Pipeline Recommendations with *KGmetaSP*

A central avenue for future work is to integrate the discovery capabilities of *KGmetaSP* with the LLM assistant in *ExeKGLab*. In the current thesis, *ExeKGLab* supports users during interactive pipeline construction, while *KGmetaSP* estimates promising pipeline configurations for unseen datasets based on previous ML experiments encoded in *MetaExe-KG*. Bringing these contributions together would allow the LLM assistant to generate recommendations that are not only semantically plausible, but also guided by empirical performance evidence.

For instance, when a user uploads a new dataset, *KGmetaSP* could identify candidate pipeline configurations that are predicted to perform well for that dataset. These candidates could then be passed to the LLM assistant as contextual guidance. Rather than generating a pipeline from the dataset description alone, the LLM could use the *KGmetaSP* candidates as starting points, adapt them to the user's goal, and produce executable pipeline suggestions within *ExeKGLab*.

As an initial proof of concept, six OpenML datasets that were not contained in *MetaExe-KG* were selected. For each dataset, its meta-features were used as the dataset representation, and *KGmetaSP* was applied to identify the pipeline configurations with the highest predicted accuracy. The top-ranked pipelines were then provided as additional context to an LLM. Three settings were compared: a zero-shot LLM prompt, an LLM prompt augmented with the top-1 *KGmetaSP* pipeline, and an LLM prompt augmented with the top-3 *KGmetaSP* pipelines. The scikit-learn pipelines generated by the LLM were evaluated on the corresponding OpenML datasets using classification metrics.

Table 9.1 summarizes the preliminary results. The zero-shot LLM achieved a mean accuracy of 0.8779 and a mean F1 score of 0.8758. Providing the top-1 *KGmetaSP* pipeline increased these scores to 0.9027 and 0.8966, respectively, while providing the top-3 pipelines led to a mean accuracy of 0.9036 and a mean F1 score of 0.8984. These results suggest that *KGmetaSP* can provide useful candidate pipelines for guiding LLM-based pipeline recommendations. However, the experiment remains limited in scale and should be interpreted as an initial indication rather than as a full evaluation.

Future work should evaluate this idea on a larger set of datasets and compare different LLMs and prompting strategies. Beyond predictive performance, future evaluations should also consider pipeline robustness, interpretability, and execution cost.

Appendix A

Evaluation Details for Chapter 5

A.1 Implementation Details for the Automated LLM Evaluation

All recommendations were generated using GPT-4o through the Azure OpenAI chat-completion interface. The Azure OpenAI API version was set to `2024-02-01`. The temperature was set to 0 in order to reduce stochastic variation. For each OpenML benchmark instance, defined by an OpenML dataset and an associated ML goal such as classification, a structured natural-language query was first created from the ML goal (*e.g.*, classification) and dataset metadata. The query followed the same general form as the user requests supported by *ExeKGLab* and included the ML goal together with the feature names and their data types. The sequence of pipeline tasks was then obtained for this query using the task-selection component. In the evaluated setup, the resulting pipeline always starts with a fixed concatenation task, followed by the selected pipeline tasks.

For each selected pipeline task, the LLM was queried separately to recommend up to five suitable methods. The prompt included three elements: (i) the structured user query, (ii) the current pipeline task for which methods should be recommended, and (iii) the list of methods supported for that pipeline task. This list was derived from the task–method compatibility mapping used by *ExeKGLib*. The LLM was explicitly instructed to recommend only methods from the provided list and to return the result as an array.

The following prompt template was used for method recommendation:

> Based on the user's query: '{user_query}', recommend the 5 best methods in the format `[method1, method2, method3, ..]` for the {pipeline_task} task in the machine learning pipeline. Consider the column names and their data types provided in the query.
>
> The methods must come from this list: {method_list}.
>
> Provide only 5 methods in the format `[method1, method2, method3, ..]`. If fewer than 5 methods are available, return them in the same format.

The placeholders were filled automatically for each OpenML benchmark instance and each selected pipeline task. The `user_query` placeholder was generated from the ML goal and the feature metadata of the corresponding dataset. The `pipeline_task` placeholder corresponded to the current task in the selected pipeline. The `method_list` placeholder contained only the methods compatible with that pipeline task.

After receiving the LLM response, the output was parsed by extracting the array returned by the model. Each extracted method name was validated against the list of allowed methods for the corresponding pipeline task. Methods not present in the valid method list were discarded. If no array format was detected, a fallback extraction step searched the raw output for valid method names. The final evaluated pipeline therefore consisted of a fixed concatenation task followed by the selected pipeline task sequence, with up to five validated LLM-recommended methods for each task.

The OpenML reference pipeline was constructed by selecting, for each OpenML benchmark instance, the run with the highest F1 score among the available runs. The method–task mapping of this best-performing run was then used as the reference pipeline. The LLM pipeline was compared against this reference pipeline at both task and method level.

A.2 Additional Analysis for the Automated LLM Evaluation

To complement the quantitative metrics reported in Section 5.5.1, this appendix examines three pipelines selected to reflect a wide range of metric values. Across the examples, low or moderate Jaccard and Hit Rate values do not indicate poor recommendations. Instead, they often reflect cases where the assistant proposes a more complete or alternative pipeline structure compared to the OpenML reference pipeline.

Each table compares the LLM-generated pipeline with the best-performing OpenML reference pipeline. For each task type defined in the *ExeKG* ontology, the table reports the method used in the OpenML pipeline alongside the ordered list of methods recommended by the LLM assistant. Task-level similarity is measured using the Jaccard coefficient, since pipeline structure is represented by the set of included pipeline task types. Method-level evaluation uses Hit Rate, which measures the proportion of OpenML reference methods that also appear among the LLM-recommended methods.

Pipeline 1: High Hit Rate with Low Jaccard. Table A.1 shows a case where the LLM assistant achieves a perfect Hit Rate while exhibiting low Jaccard similarity. The OpenML benchmark pipeline consists solely of a classification task using a Random Forest, whereas the LLM proposes additional preprocessing and feature selection tasks. This behavior is justified by the dataset characteristics. Dataset 10 consists of only nominal and ordinal attributes (*e.g.*, *lymphatics*, *block_of_affere*, *changes_in_node*), and a multiclass target variable. In such settings, preprocessing steps such as one-hot or ordinal encoding are commonly required to transform categorical variables into a representation suitable for most classifiers. Similarly, feature selection methods such as Chi-squared or mutual information can help reduce redundancy for high-cardinality categorical features. This illustrates that low task-level Jaccard similarity can reflect the assistant's tendency to propose complete and robust pipelines.

TABLE A.1: LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 10 and Run 1847757 (Jaccard: 0.3333, Hit Rate: 1.0000).

Pipeline Task	OpenML Methods	LLM Assistant Methods
DataProcessing	*(none)*	OHE, OE, SS, MMS, RS
FeatureSelection	*(none)*	Chi2, MIC, SKB, SP, RFE
Classification	RF	RF, GB, LR, DT, MLP

Pipeline 2: Low Jaccard and Zero Hit Rate. As in the previous pipeline, the assistant introduces preprocessing and feature selection tasks that are absent from the OpenML pipeline, again resulting in low Jaccard similarity (Table A.2). In this case, the OpenML pipeline employs an SVM classifier and the assistant prioritizes ensemble-based methods such as Random Forest and Gradient Boosting, resulting in a Hit Rate of zero. Given that Dataset 20 contains a multiclass target, this recommendation is methodologically sound. Standard SVM implementations are inherently binary and require additional strategies (*e.g.*, one-vs-rest) for multiclass classification, while tree-based ensembles natively support multiclass learning. This shows that low method-level Hit rate can result from the assistant recommending alternative methods that are better aligned with the dataset's characteristics.

TABLE A.2: LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 20 and Run 2087085 (Jaccard: 0.3333, Hit Rate: 0.0000).)

Pipeline Task	OpenML Methods	LLM Assistant Methods
DataProcessing	*(none)*	OHE, OE, DV, FH, TE
FeatureSelection	*(none)*	Chi2, MIC, SKB, SP, RFE
Classification	SVM	RF, GB, LR, MLP, ET

Pipeline 3: Moderate Hit Rate and Jaccard. Table A.3 presents a scenario where both Hit Rate and Jaccard similarity are moderate. The OpenML pipeline includes a PCA-based decomposition step followed by Random Forest classification, while the LLM assistant omits decomposition and instead proposes data preprocessing through feature scaling. Dataset 39 contains mainly numeric features with varying scales but relatively low dimensionality. In such cases, dimensionality reduction via PCA is not as necessary as normalization or scaling which can improve classifier performance and simplify optimization. The assistant's recommendation therefore reflects a reasonable trade-off between pipeline complexity and expected benefit. Here, the partial overlap in both task types and methods yields moderate values for both metrics, illustrating a case where the assistant adapts the benchmark pipeline structure rather than simply extending it.

TABLE A.3: LLM assistant versus OpenML reference pipeline recommendation for multiclass classification on Dataset 39 and Run 1860346 (Jaccard: 0.3333, Hit Rate: 0.5000).

Pipeline Task	**OpenML Methods**	**LLM Assistant Methods**
DataProcessing	*(none)*	MMS, SS, RS, PT, QT
Decomposition	PCA	*(none)*
Classification	RF	RF, GB, LRCV, MLP, HistGB

A.3 User Study Questionnaires and Responses

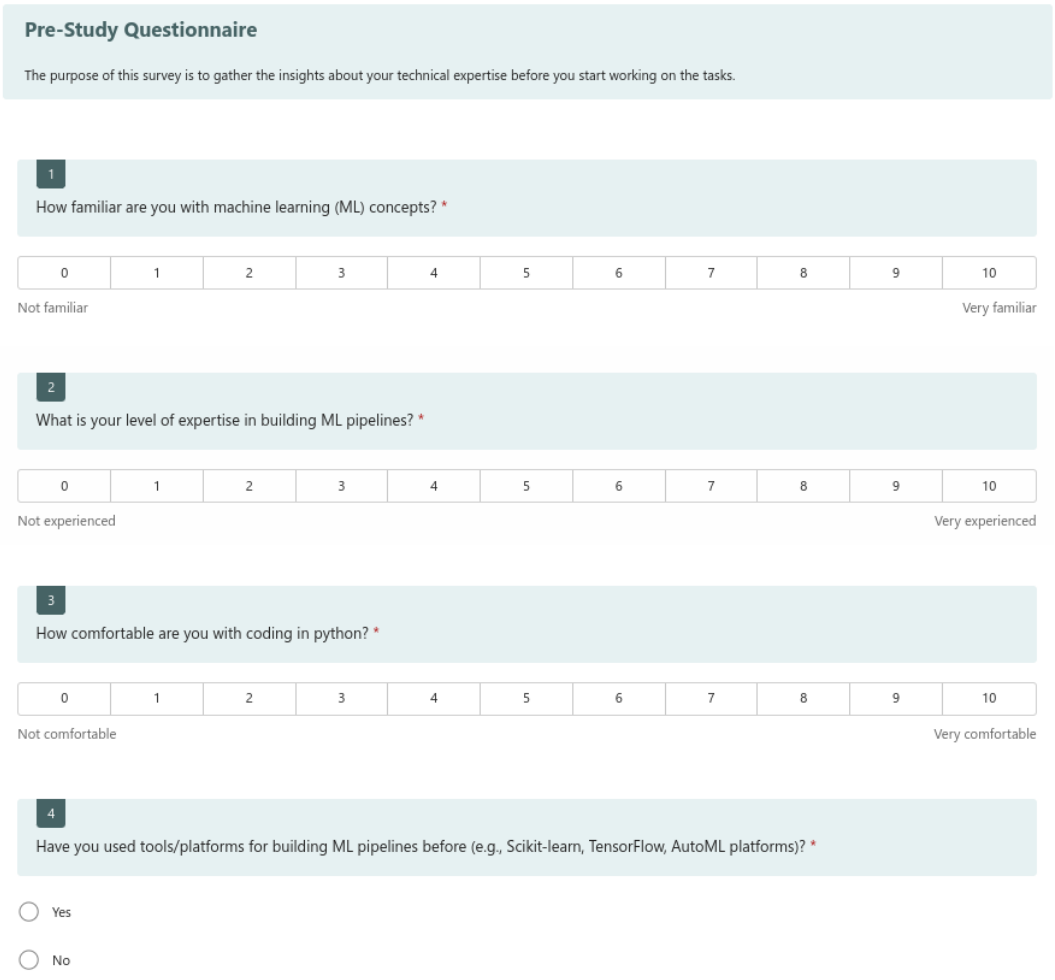


FIGURE A.1: Pre-study questionnaire

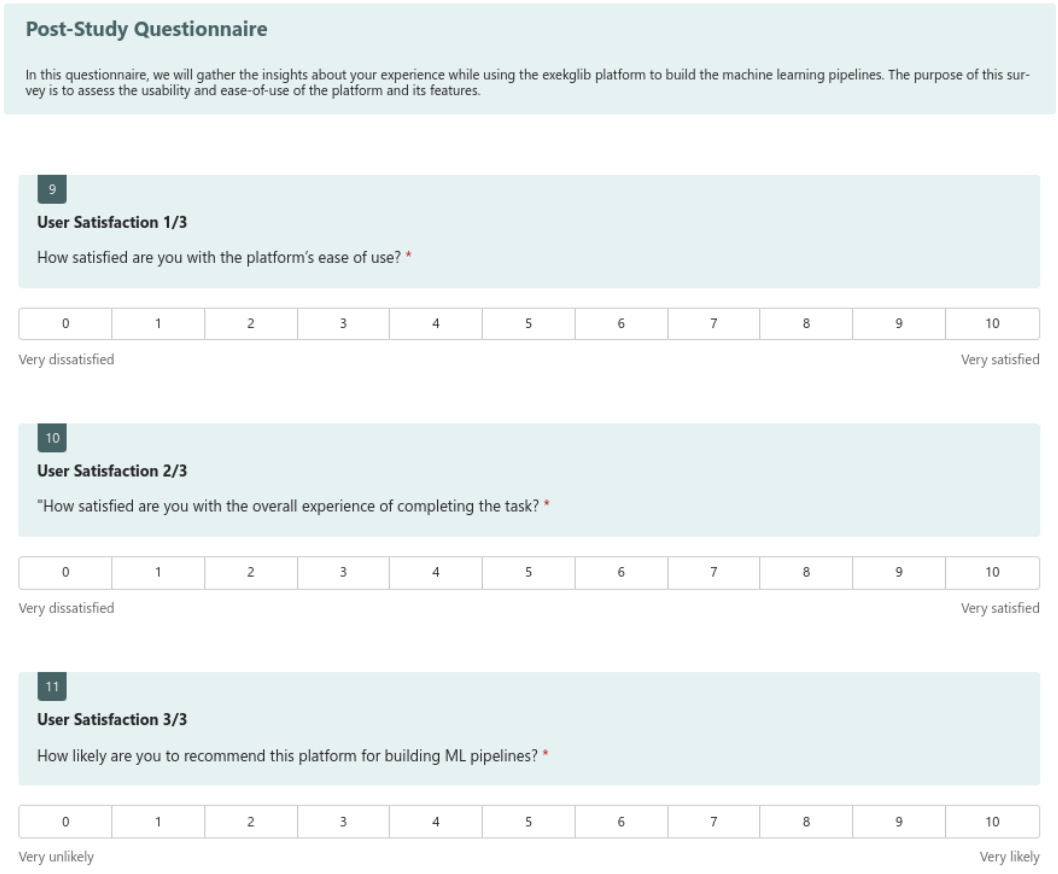


FIGURE A.2: User satisfaction

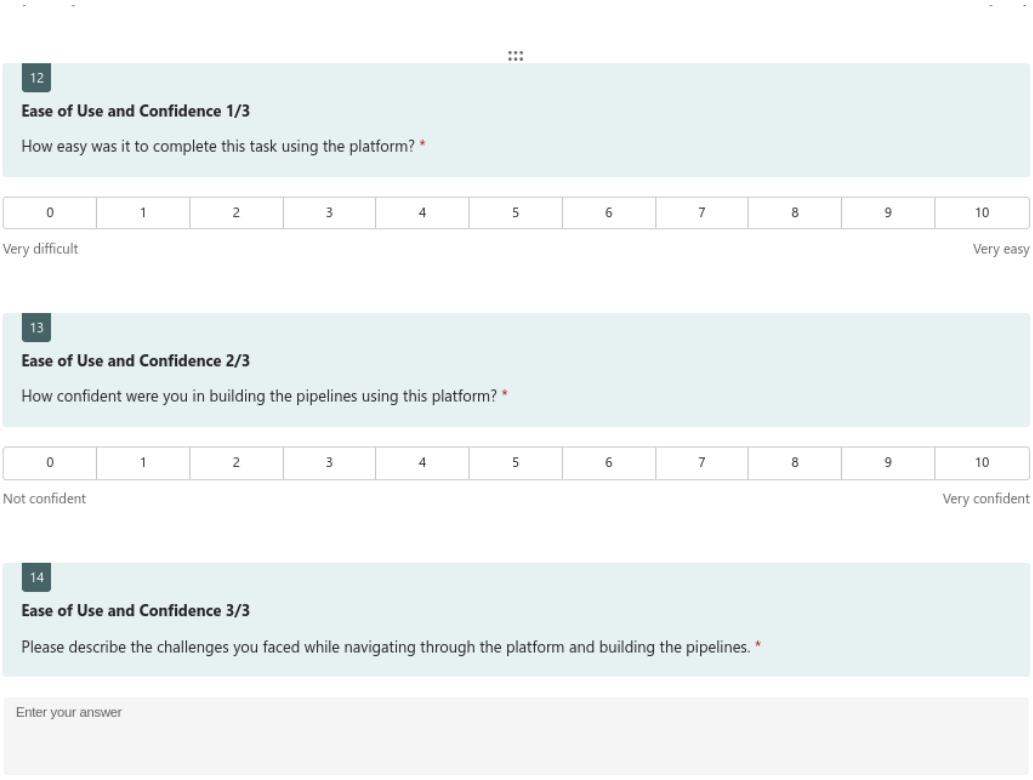


FIGURE A.3: Ease of use and confidence

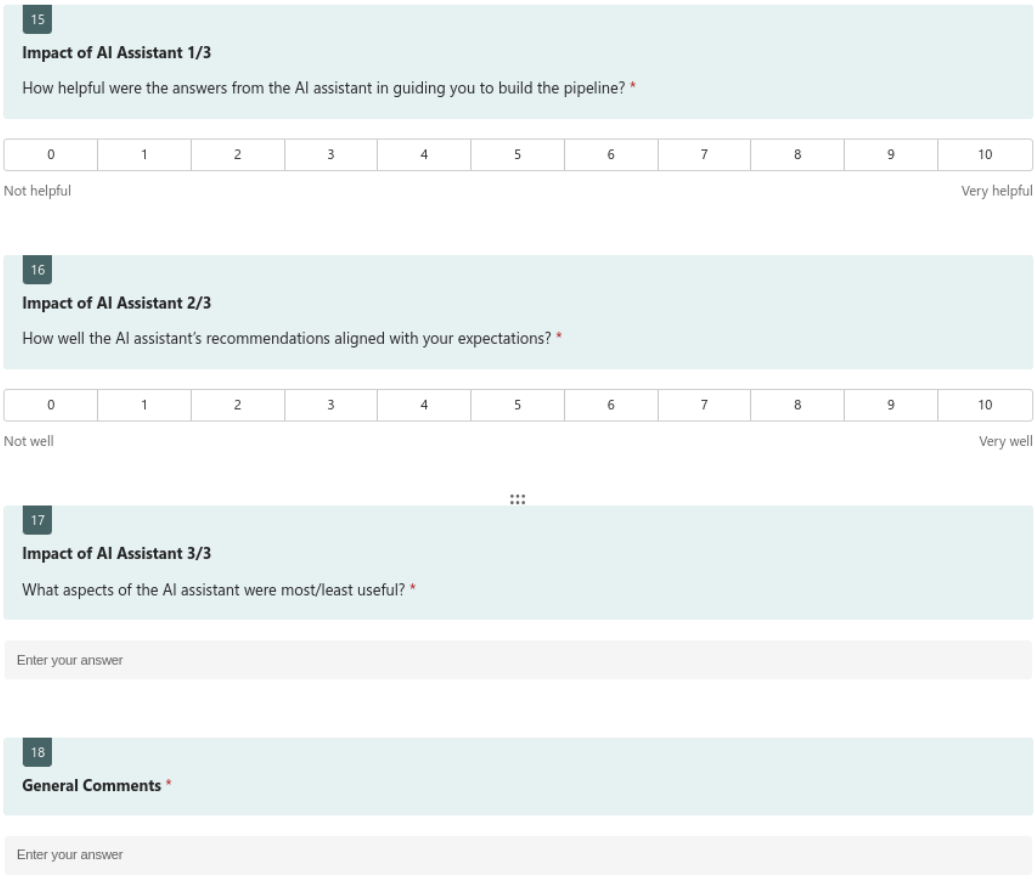


FIGURE A.4: Impact of AI assistant and general comments

```
Pipeline Details:
{
"name": "task 1",
"input_data_filename": "blood_donation_data.csv",
"data_entities": [],
"tasks": [
{
"id": "node_0",
"task_type": "Concatenation",
"method_type": "ConcatenationMethod",
"input_entity_info_dict": {
"DataInConcatenation": []
},
"method_params_dict": {},
"kg_schema_short": "ml"
},
{
"id": "node_1",
"task_type": "DataProcessing",
"method_type": "OneHotEncoderMethod",
"input_entity_info_dict": {
"DataInToPrepareTransformer": []
},
"method_params_dict": {},
"kg_schema_short": "ml"
},
{
"id": "node_2",
"task_type": "DataSplitting",
"method_type": "DataSplittingMethod",
"input_entity_info_dict": {
"DataInDataSplittingX": [],
"DataInDataSplittingY": []
},
"method_params_dict": {},
"kg_schema_short": "ml"
},
{
"id": "node_3",
"task_type": "BinaryClassification",
"method_type": "KNeighborsClassifierMethod",
"input_entity_info_dict": {
"DataInTrainY": [],
"DataInTrainX": []
},
"method_params_dict": {},
"kg_schema_short": "ml"
}
]
}
```


FIGURE A.5: Resulting pipeline for classification task from participant


```
Logging Statistics:
Total number of clicks: 47
Number of times LLM query is run: 1
Number of times task is dragged from search: 4
Number of times task is dragged from list: 0
Total time taken: 5m 26ss
```


FIGURE A.6: Participant logs example 1


```
Logging Statistics:
Total number of clicks: 71
Number of times LLM query is run: 0
Number of times task is dragged from search: 2
Number of times task is dragged from list: 1
Total time taken: 17m 47ss
```


FIGURE A.7: Participant logs example 2

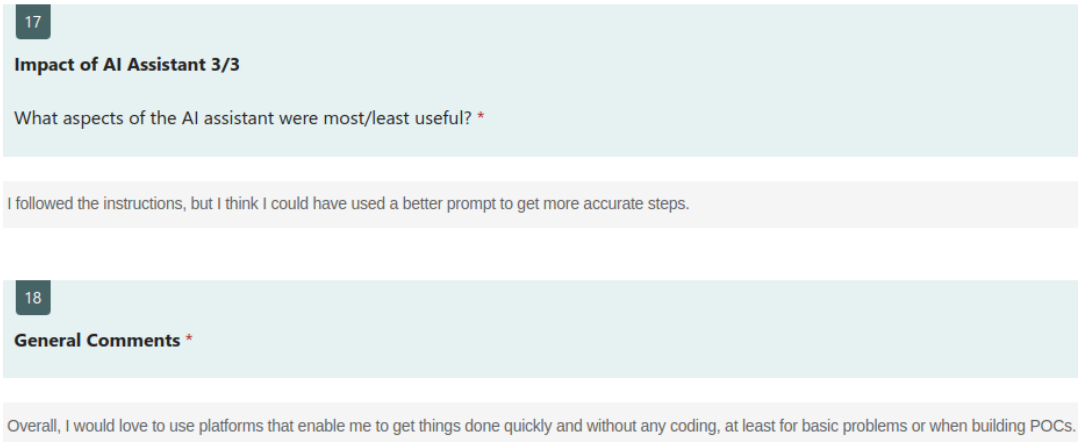


FIGURE A.8: User comments example 1

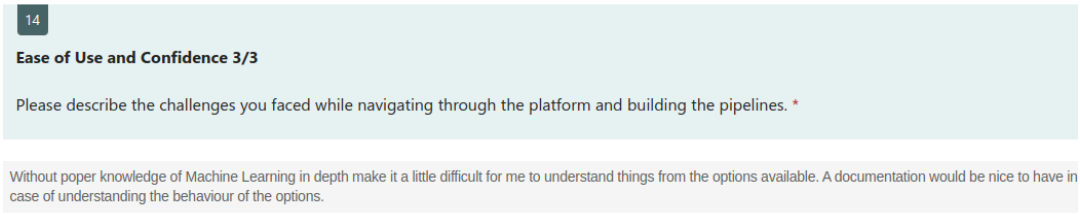


FIGURE A.9: User comments example 2

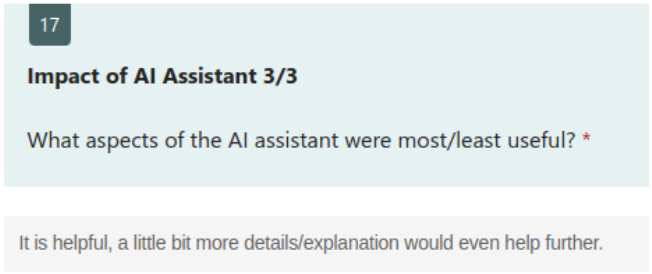


FIGURE A.10: User comments example 3

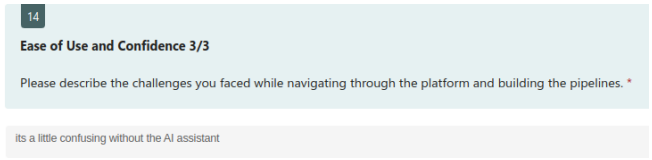


FIGURE A.11: User comments example 4

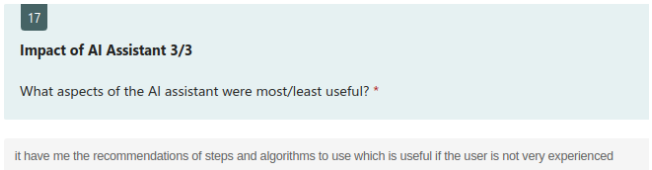


FIGURE A.12: User comments example 5

Appendix B

Implementation Details and Additional Results for Chapter 6

B.1 Implementation details

FB15k-237 preprocessing. Most numeric attributive triples (73%) in the original dataset contain duplicate entity-attribute pairs. For instance, triples `e1` *populationNumber* `427343` and `e1` *populationNumber* `399196` specify different numbers for the population of the same entity. This contributes to the lower quality of FB15k-237 than YAGO15k in terms of numeric information. For these numeric triples, the duplicates are implicitly eliminated during the conversion of numeric triples to literal matrix $\mathbf{L}$.

Link prediction models. We implement `ReaLitE` by extending the PyKEEN [6] Python library. Training, evaluation, and testing of `ReaLitE` is performed using PyKEEN. Random seeds are provided in the published code to ensure reproducibility. The overall link prediction results (Table 6.4) for the existing literal-aware methods are sourced from [153]. To provide the relation-focused results (Tables 6.5 and 6.6) for the existing literal-aware models, we train and test them using the codebase and default hyper-parameter values of [153]. The most essential hyper-parameter values for each proposed model combination are in Table B.1.

TABLE B.1: Hyper-parameter Values for Model Combinations

Task	Model	Lit. Aggr.	Emb. Dim.	In. Drop.	Batch
Link Prediction	TransE+`ReaLitE`	mode	1000	0.3	1024
	DistMult+`ReaLitE`	min	200	0.5	320
	ComplEx+`ReaLitE`	min	200	0.5	384
	RotatE+`ReaLitE`	mode	1000	0.5	1536
	TuckER+`ReaLitE`	mean	200	0.5	384
Classification	TransE+`ReaLitE`	median	50	0.1	2000
	DistMult+`ReaLitE`	median	50	0.1	75000

Node classification models. We extend the codebase of [117] to integrate our PyKEEN-based TransE+`ReaLitE` and DistMult+`ReaLitE` as embedders. After training these embedders, we perform a grid search with 5-fold cross-validation for the SVM and KNN classifiers using scikit-learn [110]. For **SVM**, the regularization parameter C is in $[0.01, 0.1, 1, 10, 100]$. For **KNN**, the number of neighbors is in $[3, 5, 8, 15]$ and the leaf size is in $[15, 30, 45]$. Random seeds are provided in the code to ensure reproducibility.

FIGURE B.1: Boxplot of relation frequencies for YAGO15k. The frequency is measured as the number of training triples that contain the relation. The boxplot's upper extreme is in bold. Outliers are depicted with dots.

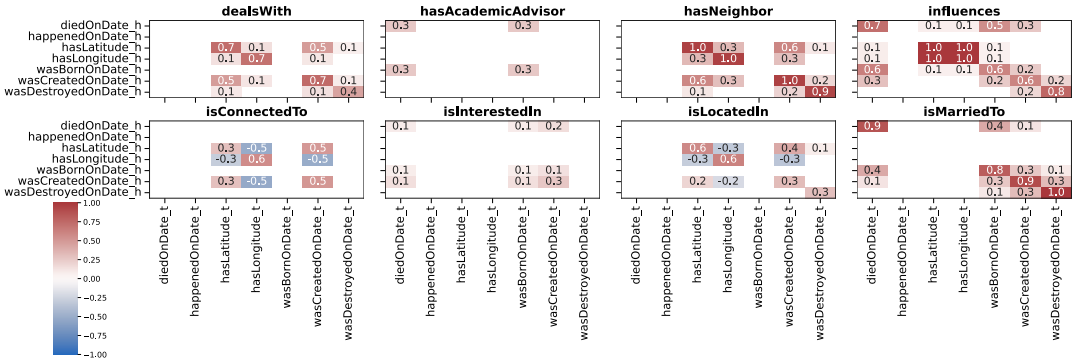


FIGURE B.2: Heatmaps of pairwise Pearson's correlation ("$corr.\ coef.$") between head (suffix 'h') and tail (suffix 't') numeric attributes per relation in YAGO15k's training set. These relations have at least one pair of head and tail attributes with $|corr.\ coef.| \geq 0.2$ in the training set. They group Table 6.6's test triples. Correlations with $|corr.\ coef.| < 0.05$ are blank.

B.2 Relation-focused evaluation details

Relations frequency. One aspect of our relation-focused link prediction evaluation, involves dividing the test triples of YAGO15k based on the frequencies of their relations (Table 6.5). The reason that we choose to use 2,817 (2.55%) as the division point is its property of being the upper extreme in the frequency boxplot (Figure B.1). As we can see, six relation frequencies are above that point. The corresponding relations (with the percentage of training triples) are: *playsFor* (33%), *isAffiliatedTo* (23%), *isCitizenOf* (13%), *isLocatedIn* (9%), *actedIn* (5%), and *wasBornIn* (4%). We consider them frequent, while the rest of the relations (below that point) are considered long-tail.

Numeric attributes correlation. As part of the relation-focused link prediction evaluation, we divide the test triples of YAGO15k based on the correlation between head and tail attributes (explained in Section 6.4.4 and Table 6.6). We choose 0.2 as a minimum threshold for the correlation coefficient. We do so because the data show that relations having at least one pair of head and tail attributes with $|corr.\ coef.|$ greater than or equal to 0.2, also have at least another pair of head and tail attributes with $|corr.\ coef.|$ greater than or equal to 0.1 (Figure B.2). This threshold choice helps us focus on meaningful correlations, considering that most head-tail attribute pairs in YAGO15k have $|corr.\ coef.| < 0.005$. Furthermore, six out of eight selected relations have at least one pair of head and tail attributes with $|corr.\ coef.| \geq 0.6$. So, our conclusion for `ReaLitE`'s performance on correlated literals (Section 6.5.1) is not sensitive to the choice of the threshold.

TABLE B.2: MRR for the filtered test set of YAGO15k based on relation types grouped by symmetry. 'Symm. rel.' refers to the test triples that contain symmetric relations. 'Asymm. rel.' refers to the remaining test triples that contain asymmetric relations. 'All triples' refers to all test triples. The best results per base model per triple group are in bold. The best results per triple group are underlined.

Model		Symm. rel.	Asymm. rel.	All triples
TuckER	+LiteralE	**0.499**	0.440	0.441
	+KGA	0.494	0.455	0.455
	+ReaLitE	0.326	**0.466**	**0.463**
TransE	+LiteralE	0.192	0.462	0.456
	+KGA	**0.197**	0.474	0.469
	+ReaLitE	0.189	**0.497**	**0.491**
DistMult	+LiteralE	0.436	0.462	0.461
	+KGA	0.439	0.470	0.470
	+ReaLitE	**0.454**	**0.490**	**0.489**
ComplEx	+LiteralE	0.434	0.442	0.442
	+KGA	0.448	0.451	0.451
	+ReaLitE	**0.499**	**0.487**	**0.487**
RotatE	+KGA	0.498	0.468	0.469
	+ReaLitE	**0.526**	**0.482**	**0.483**

B.3 Detailed link prediction results

Extended relation-focused results. We further investigate the performance of literal-aware models by dividing the test triples of YAGO15k based on their relation's symmetry (Table B.2). We see that ReaLitE is always dominant for assymetric relations (column 'Asymm. rel.'). While this effect is less pronounced for symmetric relations (column 'Symm. rel.'), we still achieve the best results on this relation group with ReaLitE (RotatE+ReaLitE).

The symmetric relations are: *dealsWith*, *isMarriedTo*, *hasNeighbor*, and *isConnectedTo*. All these relations are part of the relation groups used to create the 'Long-tail' and 'Corr. lit.' triple groups mentioned in the main paper (Tables 6.5 and 6.6). Thus, the superiority of TuckER+KGA over TuckER+ReaLitE in those groups can be explained by ReaLitE's weakness in the case of symmetric relations.

B.4 Literal aggregation choice evaluation

HPO involved training multiple model instances and evaluating them on the validation set, with literal aggregation method as a key hyper-parameter. Figure B.3 shows the resulting MRR fluctuations.

HPO trials reveal that *mode* aggregation consistently yields the highest MRR with minimal fluctuation ($\leq$ 0.02) across most model-dataset pairs. Other methods exhibit greater variability, with either larger MRR fluctuations ($\geq$ 0.1) or significant differences ($\geq$ 0.02) in maximum MRR between models. Aggregation impact is less pronounced on FB15k-237 (MRR range $\approx$ 0.15) than on YAGO15k ($\approx$ 0.3), likely due to the lower quality of numeric literals in FB15k-237.

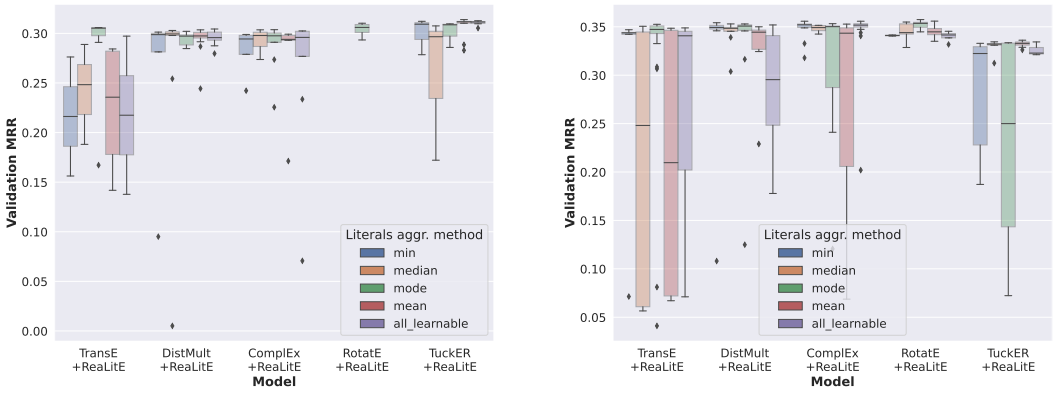


FIGURE B.3: Fluctuation of model performance for various choices of literal aggregation method during HPO. "all_learnable" is function **y** from Eq. 6.3. **Left**: FB15k-237's validation set. **Right**: YAGO15k's validation set.

TABLE B.3: Accuracy on the synthetic dataset for literal usage evaluation. The baseline results are sourced from the paper that proposed the evaluation method.

Model	Accuracy
KBLN	0.482
MTKGNN	0.472
TransEA	0.489
DistMult+KGA	0.999
TuckER+KGA	0.999
DistMult+LiteralE	0.512
ComplEx+LiteralE	0.493
DistMult+`ReaLitE`	0.505
TuckER+`ReaLitE`	0.477

B.5 Evaluating model's use of numeric literals for link prediction

Following [18], we assessed `ReaLitE`'s handling of numeric literals by creating a semi-synthetic dataset with a classification task. This involves introducing new relational triples where class assignment depends on whether an entity's attribute value is above or below a threshold. Model accuracy in predicting these classes evaluates the effective utilization of attribute information.

The results of this evaluation are shown in Table B.3. `ReaLitE` is on par with most literal-aware models in this task, with the only exception being KGA. The authors of [18] attribute this to KGA's reluctance to incorporate specific literal values. They suggest that while KGA simplifies the task posed by the semi-synthetic dataset into a more manageable graph-structure learning problem, it might face difficulties with more complex synthetic datasets.

Appendix C

Implementation Details for Chapter 7

This appendix provides implementation details for constructing *MetaExe-KG*, generating KG embeddings, and training the downstream *PPE* meta-models.

C.1 Experimental Configuration

***RDF2Vec* Embedding Hyper-parameters.** For the walk-based setting, *MetaExe-KG* is embedded using *RDF2Vec* [124] with the `gensim` Word2Vec implementation. Numeric literals are handled using MKGA [116], with the strategy `propConst_LOF_del_numbers`. We generate 10 random walks per entity with walk length 20. Word2Vec is trained for 10 epochs with embedding dimensionality 100, context window size 5, negative sampling rate 5, and minimum token frequency 0. All unspecified parameters use the default `gensim` settings.

LP-based Embedding Hyper-parameters. For the LP-based setting, we evaluate TransE, DistMult, and ComplEx using PyKEEN. All three models are extended with `ReaLitE` to incorporate numeric literals. The embeddings are trained for 1500 epochs with dimensionality 128, batch size 2048, learning rate 0.0005, three negative samples per positive triple, self-adversarial negative sampling, and margin-based loss with margin 50.

Data Splitting for *PPE*. We use a 70/30 train-test split with random seed 42. For the unseen dataset scenario, splitting is performed at the dataset level. Thus, all dataset–pipeline pairs associated with a given dataset are assigned entirely to either the training or the test split. For the unseen pipeline configuration scenario, splitting is performed analogously at the pipeline configuration level, ensuring that all pairs associated with a given pipeline configuration occur in only one split.

***PPE* Meta-Models.** Pipeline-agnostic meta-models are implemented in `scikit-learn` using Random Forests for both meta-regression and meta-classification. Hyper-parameters are selected via 10-fold cross-validation on the training set. The grid search considers the number of estimators in $[50, 200]$ for meta-classification and $[20, 200]$ for meta-regression, maximum tree depth in $\{10, 20, \text{None}\}$, and minimum samples per split in $\{2, 5, 10\}$. Other parameters use the default `scikit-learn` settings with random seed 42.

C.2 Analysis of Computational Cost

Runtime. All experiments are conducted on an AMD EPYC-7643 CPU. Constructing *MetaExe-KG* requires approximately 7.5 hours using 10 CPU cores. This includes constructing 2,616 *ExeKGs* from OpenML runs and producing approximately 4.5 million triples. The main overhead stems from repeated OpenML API calls and the construction of individual execution graphs. Training the *RDF2Vec* embeddings requires approximately 24 hours with parallel random-walk generation on 10 CPU cores. Since a single embedding model provides representations for all datasets and pipelines, this cost is incurred once per embedding model.

TABLE C.1: Ablation of *MLSeaKG* integration for *PPE*. Values report Δ = With − Without. For MSE, negative values indicate improvement; for R^2, Accuracy, and F1, positive values indicate improvement.

Setting	Target	Meta-Regression		Meta-Classification	
		Δ MSE	**Δ R^2**	**Δ Acc.**	**Δ F1**
Unseen datasets	Accuracy	-0.0002	+0.0067	+0.0111	+0.0107
Unseen datasets	Precision	-0.0003	+0.0091	-0.0045	-0.0046
Unseen pipeline configurations	Accuracy	-0.0004	+0.0115	+0.0025	+0.0024
Unseen pipeline configurations	Precision	-0.0004	+0.0101	+0.0035	+0.0036

Training ComplEx+`ReaLitE` requires approximately 3.75 hours for 54 epochs, with an average epoch time of about 233 seconds. TransE+`ReaLitE` requires approximately 10.8 hours for 64 epochs, with an average epoch time of about 563 seconds. DistMult+`ReaLitE` requires approximately 31.7 hours for 209 epochs, with an average epoch time of about 534 seconds. The downstream *PPE* meta-model training requires approximately 9.5 hours.

Memory. During KG construction, memory usage peaks at approximately 6GB RAM. *RDF2Vec* embedding training requires approximately 80GB RAM, mainly due to the in-memory random-walk corpus generated over a KG with around 1.5 million nodes. The LP-based KGE jobs show high host-level memory usage, peaking at approximately 214GB RAM, while the average process memory remains lower: approximately 11.4GB for ComplEx, 9.3GB for TransE, and 8.7GB for DistMult. Training the downstream *PPE* meta-models requires approximately 13GB RAM.

Scalability. The construction of *MetaExe-KG* scales approximately linearly with the number of recorded experiments, since each *ExeKG* is constructed independently. *RDF2Vec* scales with the number of KG nodes, the number and length of random walks, and the embedding dimensionality. LP-based KGE models scale with the number of triples, embedding dimensionality, number of negative samples, batch size, and number of training epochs. The *PPE* meta-models scale with the number of dataset–pipeline pairs and the dimensionality of the concatenated feature vectors. In our setting, embedding training is the main bottleneck, while downstream *PPE* training remains comparatively lightweight.

C.3 Ablation Study: Impact of *MLSeaKG* Integration

To examine the contribution of semantic enrichment, we compare *KGmetaSP* with and without the integration of *MLSeaKG* across both *PPE* and *DPSE*. Tables C.1 and C.2 report the effect of adding *MLSeaKG* as Δ = With − Without.

For *PPE*, the *MLSeaKG*-enhanced variant improves all meta-regression results, as indicated by lower MSE and higher R^2 values. In the unseen datasets setting, the effect on meta-classification is mixed: *MLSeaKG* improves accuracy prediction, but the non-enriched variant is slightly better for precision prediction. In the unseen pipeline configuration setting, the enriched variant improves all reported metrics, suggesting that semantic enrichment is particularly helpful when generalizing to previously unseen pipeline structures.

For *DPSE*, adding *MLSeaKG* improves top-rank retrieval, increasing Hit@1 and NDCG@1. At larger ranks, the effect is mixed: Hit@2 remains unchanged, while Hit@5 and NDCG@5 slightly decrease. This suggests that *MLSeaKG* mainly improves the quality of the highest-ranked retrieved dataset rather than broadening retrieval performance at larger k.

C.4 Detailed PPE Results

This section reports the complete *PPE* results that complement the tables in Chapter 7. Unlike the main chapter, the tables here include the full breakdown of LP-based variants. In all tables, ComplEx, DistMult, and TransE use `ReaLitE` for numeric literals. Meta-regression

TABLE C.2: Ablation of *MLSeaKG* integration for *DPSE*. Values report Δ = With − Without. For Hit@k, datasets with similarity > 0.9 are considered hits.

	Hit@k			NDCG@k		
Method	**k=1**	**k=2**	**k=5**	**k=1**	**k=2**	**k=5**
$KGmetaSP_{DE_{pip}}$	+0.0473	0.0000	-0.0270	+0.0260	+0.0036	-0.0019

predicts raw performance values, while meta-classification uses three bins. Best overall scores are underlined, and best scores per meta-feature group are bold.

C.4.1 Unseen Datasets

Table C.3 reports the full results for the unseen datasets setting.

C.4.2 Unseen Pipeline Configurations

Table C.4 reports the full results for the unseen pipeline configuration setting.

TABLE C.3: Detailed *PPE* results for unseen datasets.

Dataset	Pipeline	Meta-	Meta-Regr.		Meta-Classif.	
Embedding	Strategy	Models	MSE	R2	Accuracy	F1
Target: Accuracy						
MF All	Conf.-specific	1,028	**0.0081**	**0.6748**	0.7363	0.7358
	$KGmetaSP_{PE}$ ComplEx	1	0.0106	0.6006	0.7343	0.7361
	$KGmetaSP_{PE}$ DistMult	1	0.0103	0.6104	0.7380	0.7393
	$KGmetaSP_{PE}$ TransE	1	0.0105	0.6032	0.7351	0.7368
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0101	0.6181	**0.7413**	**0.7427**
MF Info. Theory [100]	Conf.-specific	1,028	**0.0249**	**-0.0737**	0.4428	0.4423
	$KGmetaSP_{PE}$ ComplEx	1	0.0397	-0.4964	0.4827	0.4854
	$KGmetaSP_{PE}$ DistMult	1	0.0399	-0.5069	**0.4911**	**0.4938**
	$KGmetaSP_{PE}$ TransE	1	0.0400	-0.5083	0.4877	0.4899
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0401	-0.5135	0.4773	0.4789
MF Landmarkers [11]	Conf.-specific	1,028	**0.0092**	**0.6428**	0.7152	0.7139
	$KGmetaSP_{PE}$ ComplEx	1	0.0108	0.5908	0.7156	0.7167
	$KGmetaSP_{PE}$ DistMult	1	0.0107	0.5977	**0.7212**	**0.7222**
	$KGmetaSP_{PE}$ TransE	1	0.0108	0.5918	0.7181	0.7196
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0104	0.6094	0.7169	0.7172
MF Statistical [100]	Conf.-specific	1,028	**0.0218**	**0.0892**	0.5499	0.5484
	$KGmetaSP_{PE}$ ComplEx	1	0.0254	0.0404	**0.6077**	**0.6054**
	$KGmetaSP_{PE}$ DistMult	1	0.0254	0.0409	0.6016	0.6014
	$KGmetaSP_{PE}$ TransE	1	0.0255	0.0392	0.5922	0.5913
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0259	0.0212	0.5827	0.5830
Target: Precision						
MF All	Conf.-specific	1,028	**0.0133**	**0.6311**	0.7347	0.7400
	$KGmetaSP_{PE}$ ComplEx	1	0.0175	0.5138	0.7359	0.7383
	$KGmetaSP_{PE}$ DistMult	1	0.0173	0.5198	0.7461	0.7482
	$KGmetaSP_{PE}$ TransE	1	0.0176	0.5117	0.7430	0.7457
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0164	0.5449	**0.7537**	**0.7562**
MF Info. Theory [100]	Conf.-specific	1,028	**0.0330**	**-0.0933**	0.4324	0.4350
	$KGmetaSP_{PE}$ ComplEx	1	0.0507	-0.4094	0.4902	0.4941
	$KGmetaSP_{PE}$ DistMult	1	0.0516	-0.4341	0.4913	0.4958
	$KGmetaSP_{PE}$ TransE	1	0.0511	-0.4182	0.4876	0.4915
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0506	-0.4059	**0.4961**	**0.5007**
MF Landmarkers [11]	Conf.-specific	1,028	**0.0148**	**0.6069**	0.6987	0.7006
	$KGmetaSP_{PE}$ ComplEx	1	0.0183	0.4921	**0.7288**	**0.7318**
	$KGmetaSP_{PE}$ DistMult	1	0.0180	0.4990	0.7245	0.7270
	$KGmetaSP_{PE}$ TransE	1	0.0183	0.4920	0.7278	0.7300
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0184	0.4889	0.7247	0.7269
MF Statistical [100]	Conf.-specific	1,028	**0.0274**	**0.1221**	0.5490	0.5487
	$KGmetaSP_{PE}$ ComplEx	1	0.0345	0.0416	**0.5932**	**0.5928**
	$KGmetaSP_{PE}$ DistMult	1	0.0347	0.0354	0.5708	0.5716
	$KGmetaSP_{PE}$ TransE	1	0.0345	0.0417	0.5879	0.5896
	$KGmetaSP_{PE}$ RDF2Vec	1	0.0346	0.0381	0.5897	0.5896

TABLE C.4: Detailed *PPE* results for unseen pipeline configurations.

Method	Meta-Regression		Meta-Classification	
	MSE	**R2**	**Accuracy**	**F1**
Target: Accuracy				
Average performance	0.0267	-0.0005	0.3303	0.1640
Closest embedding	0.0127	0.5241	0.7748	0.7747
MF All + $KGmetaSP_{\text{PE}}^{\text{ComplEx}}$	0.0083	0.6911	0.8072	0.8062
MF All + $KGmetaSP_{\text{PE}}^{\text{TransE}}$	0.0081	0.6976	0.8092	0.8083
MF All + $KGmetaSP_{\text{PE}}^{\text{DistMult}}$	0.0080	0.7007	0.8076	0.8068
MF All + $KGmetaSP_{\text{PE}}^{\text{RDF2Vec}}$	**0.0070**	**0.7361**	**0.8250**	**0.8244**
Target: Precision				
Average performance	0.0352	-0.0012	0.3323	0.1658
Closest embedding	0.0225	0.3607	0.7773	0.7771
MF All + $KGmetaSP_{\text{PE}}^{\text{ComplEx}}$	0.0153	0.5650	0.8088	0.8080
MF All + $KGmetaSP_{\text{PE}}^{\text{TransE}}$	0.0148	0.5792	0.8072	0.8064
MF All + $KGmetaSP_{\text{PE}}^{\text{DistMult}}$	0.0148	0.5804	0.8089	0.8083
MF All + $KGmetaSP_{\text{PE}}^{\text{RDF2Vec}}$	**0.0131**	**0.6264**	**0.8251**	**0.8247**

Bibliography

[1] Ralph Abboud and İsmail İlkan Ceylan. *Node Classification Meets Link Prediction on Knowledge Graphs*. Aug. 2021. DOI: 10.48550/arXiv.2106.07297. arXiv: 2106.07297 [cs]. (Visited on 01/08/2024).

[2] Ibrahim Abdelaziz et al. *A Toolkit for Generating Code Knowledge Graphs*. Sept. 2021. DOI: 10.48550/arXiv.2002.09440. arXiv: 2002.09440 [cs]. (Visited on 08/28/2024).

[3] Pedro Henriques Abreu et al. "Predicting Breast Cancer Recurrence Using Machine Learning Techniques: A Systematic Review". In: *ACM Computing Surveys* 49.3 (Oct. 2016), 52:1–52:40. ISSN: 0360-0300. (Visited on 12/19/2022).

[4] Edesio Alcobaça and André C. P. L. F. De Carvalho. "Exploring One Million Machine Learning Pipelines: A Benchmarking Study". In: *AutoML 2025*. 2025.

[5] Mehdi Ali et al. "Bringing Light Into the Dark: A Large-scale Evaluation of Knowledge Graph Embedding Models Under a Unified Framework". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.12 (Dec. 2022), pp. 8825–8845. ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: 10.1109/TPAMI.2021.3124805. arXiv: 2006.13365 [cs, stat]. (Visited on 01/16/2023).

[6] Mehdi Ali et al. "PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings". In: *Journal of Machine Learning Research* 22.82 (2021), pp. 1–6. ISSN: 1533-7928. (Visited on 01/22/2024).

[7] Saleema Amershi et al. "Software engineering for machine learning: a case study". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '19. Montreal, Quebec, Canada: IEEE Press, 2019, 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042. URL: https://doi.org/10.1109/ICSE-SEIP.2019.00042.

[8] Julia Angwin et al. "Machine Bias". In: *Propublica* (2016). URL: https://www.propublica.org/article/machine-bias-risk-assessments-in-criminal-sentencing.

[9] Grigoris Antoniou and Frank van Harmelen. *A Semantic Web Primer*. 3rd. The MIT Press, 2012.

[10] Ivana Balazevic, Carl Allen, and Timothy Hospedales. "TuckER: Tensor Factorization for Knowledge Graph Completion". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5185–5194. DOI: 10.18653/v1/D19-1522. (Visited on 01/08/2024).

[11] Hilan Bensusan and Christophe G. Giraud-Carrier. "Discovering Task Neighbourhoods Through Landmark Learning Performances". In: *ECML-PKDD*. 2000.

[12] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web". In: *Scientific American* 284.5 (2001), pp. 34–43.

[13] Michael R. Berthold et al. "KNIME: The Konstanz Information Miner". In: *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007. ISBN: 978-3-540-78239-1.

[14] Besim Bilalli, Alberto Abelló, and Tomàs Aluja-Banet. "On the predictive power of meta-features in OpenML". In: *International Journal of Applied Mathematics and Computer Science* 27.4 (2017), pp. 697–712. DOI: 10.1515/amcs-2017-0048. URL: https://doi.org/10.1515/amcs-2017-0048.

[15] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked Data - The Story So Far". In: *International Journal on Semantic Web and Information Systems* 5.3 (2009), pp. 1–22.

[16] Christian Bizer et al. "DBpedia-A crystallization point for the Web of Data". In: *The semantic web*. Springer. 2007, pp. 722–735.

[17] Peter Bloem et al. "Kgbench: A Collection of Knowledge Graph Datasets for Evaluating Relational and Multimodal Machine Learning". In: *Eighteenth Extended Semantic Web Conference - Resources Track*. Dec. 2020. (Visited on 01/08/2024).

[18] Moritz Blum et al. "Numerical Literals in Link Prediction: A Critical Examination of Models and Datasets". In: *Proceedings of the 23rd International Semantic Web Conference*. Nov. 2024.

[19] Kurt Bollacker et al. "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. Vancouver Canada: ACM, June 2008, pp. 1247–1250. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376746. (Visited on 01/15/2024).

[20] Ivelina Boneva et al. "Validation of RDF-star graphs with SHACL-star". In: *The Semantic Web – ISWC 2021*. 2021, pp. 19–35.

[21] Antoine Bordes et al. "Translating Embeddings for Modeling Multi-relational Data". In: *Advances in Neural Information Processing Systems 26*. 2013.

[22] Antoine Bordes et al. "Translating Embeddings for Modeling Multi-Relational Data". In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges et al. Vol. 26. Curran Associates, Inc., 2013.

[23] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. New York, NY, USA: Association for Computing Machinery, July 1992, pp. 144–152. ISBN: 978-0-89791-497-0. DOI: 10.1145/130385.130401. (Visited on 01/17/2024).

[24] Pavel Brazdil et al. *Metalearning: Applications to Data Mining*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 3540732624.

[25] Dan Brickley and R. V. Guha. *RDF Schema 1.1*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/rdf-schema/.

[26] Dan Busbridge et al. *Relational Graph Attention Networks*. Apr. 2019. DOI: 10.48550/arXiv.1904.05811. arXiv: 1904.05811 [cs, stat]. (Visited on 01/15/2024).

[27] Cafarella, Michael J. and Halevy, Alon and Wang, Daisy Zhe and Wu, Eugene and Zhang, Yang. "WebTables: exploring the power of tables on the web". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), 538–549. ISSN: 2150-8097.

[28] Hongyun Cai, Víctor G. Pérez, and Chen Qu. "A comprehensive survey of graph embedding: problems, techniques and applications". In: *IEEE Transactions on Knowledge and Data Engineering* 30.9 (2018), pp. 1616–1637.

[29] Davide Carneiro et al. "Using meta-learning to predict performance metrics in machine learning problems". In: *Expert Systems* 40.1 (2023), e12900. DOI: https://doi.org/10.1111/exsy.12900. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/exsy.12900. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/exsy.12900.

[30] Ciro Castiello, Giovanna Castellano, and Anna Maria Fanelli. "Meta-data: Characterization of Input Features for Meta-learning". In: *Modeling Decisions for Artificial Intelligence*. Ed. by Vicenç Torra, Yasuo Narukawa, and Sadaaki Miyamoto. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 457–468. ISBN: 978-3-540-31883-5. DOI: https://doi.org/10.1007/11526018_45.

[31] Qiaosheng Chen et al. "Benchmarking Recommendation, Classification, and Tracing Based on Hugging Face Knowledge Graph". In: *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '25. Padua, Italy: Association for Computing Machinery, 2025, 3433–3443. ISBN: 9798400715921. DOI: 10.1145/3726302.3730277. URL: https://doi.org/10.1145/3726302.3730277.

[32] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179/.

[33] Joshua Coates and Danushka Bollegala. "Frustratingly Easy Meta-Embedding – Computing Meta-Embeddings by Averaging Source Word Embeddings". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 194–198. DOI: 10.18653/v1/N18-2031. URL: https://aclanthology.org/N18-2031/.

[34] Julien Corman, Juan L. Reutter, and Ognjen Savković. "SHACL: A language for validating RDF graphs". In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 1033–1042.

[35] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[36] Ioannis Dasoulas, Duo Yang, and Anastasia Dimou. "MLSea: A Semantic Layer for Discoverable Machine Learning". In: *The Semantic Web*. Cham: CSAL, 2024, pp. 178–198. ISBN: 978-3-031-60635-9.

[37] Janez Demšar et al. "Orange: Data Mining Toolbox in Python". In: *Journal of Machine Learning Research* 14 (2013), pp. 2349–2353. URL: http://jmlr.org/papers/v14/demsar13a.html.

[38] Tim Dettmers et al. "Convolutional 2D Knowledge Graph Embeddings". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*. AAAI'18/IAAI'18/EAAI'18. New Orleans, Louisiana, USA: AAAI Press, Feb. 2018, pp. 1811–1818. ISBN: 978-1-57735-800-8. (Visited on 01/08/2024).

[39] Iddo Drori et al. "AlphaD3M: Machine Learning Pipeline Synthesis". English. In: *ICML 2018 AutoML Workshop*. 2021. HDL: https://orbilu.uni.lu/10993/57634.

[40] Michel Dumontier et al. "The Semanticscience Integrated Ontology (SIO) for Biomedical Research and Knowledge Discovery". In: *Journal of Biomedical Semantics* 5.1 (Mar. 2014), p. 14. ISSN: 2041-1480. DOI: 10.1186/2041-1480-5-14. (Visited on 08/28/2024).

[41] Lisa Ehrlinger and Wolfram Wöß. "Towards a Definition of Knowledge Graphs". In: *CEUR Workshop Proceedings* 1695 (2016), p. 2.

[42] Diego Esteves et al. "MEX Vocabulary: A Lightweight Interchange Format for Machine Learning Experiments". In: *Proceedings of the 11th International Conference on Semantic Systems*. SEMANTICS '15. New York, NY, USA: Association for Computing Machinery, Sept. 2015, pp. 169–176. ISBN: 978-1-4503-3462-4. DOI: 10.1145/2814864.2814883. (Visited on 08/28/2024).

[43] Lee Feigenbaum et al. *SPARQL 1.1 Protocol*. W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/sparql11-protocol/.

[44] Dieter Fensel et al. *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Springer, 2020.

[45] Daniel Fernandez-Álvarez, Jose Emilio Labra-Gayo, and Daniel Gayo-Avello. "Automatic Extraction of Shapes Using sheXer". In: *Knowledge-Based Systems* 238 (Feb. 2022), p. 107975. ISSN: 0950-7051. (Visited on 04/22/2023).

[46] Matthias Feurer, Jost Springenberg, and Frank Hutter. "Initializing Bayesian Hyperparameter Optimization via Meta-Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 2015). DOI: 10.1609/aaai.v29i1.9354. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9354.

[47] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. "Initializing Bayesian Hyperparameter Optimization via Meta-Learning". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015, pp. 1128–1135.

[48] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. "Using Meta-learning to Initialize Bayesian Optimization of Hyperparameters". In: *Proceedings of the 2014 International Conference on Meta-Learning and Algorithm Selection - Volume 1201*. MLAS'14. Prague, Czech Republic: CEUR-WS.org, 2014, 3–10. ISBN: 16130073.

[49] Matthias Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf.

[50] Evelyn Fix and Joseph L. Hodges. *Discriminatory Analysis - Nonparametric Discrimination: Consistency Properties*. Tech. rep. USAF School of Aviation Medicine, Randolph Field, Texas, 1951. Chap. Technical Reports. (Visited on 01/17/2024).

[51] Iason Gabriel. "Artificial Intelligence, Values, and Alignment". In: *Minds and Machines* 30 (Sept. 2020), pp. 411–437. DOI: 10.1007/s11023-020-09539-2.

[52] Alberto García-Durán, Sebastijan Dumančić, and Mathias Niepert. "Learning Sequence Encoders for Temporal Knowledge Graph Completion". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by Ellen Riloff et al. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 4816–4821. DOI: 10.18653/v1/D18-1516. (Visited on 01/15/2024).

[53] Alberto Garcia-Duran and Mathias Niepert. *KBLRN : End-to-End Learning of Knowledge Base Representations with Latent, Relational, and Numerical Features*. June 2018. DOI: 10.48550/arXiv.1709.04676. arXiv: 1709.04676 [cs]. (Visited on 01/08/2024).

[54] Genet Asefa Gesese et al. *A Survey on Knowledge Graph Embeddings with Literals: Which Model Links Better Literal-ly?* May 2020. DOI: 10.48550/arXiv.1910.12507. arXiv: 1910.12507 [cs]. (Visited on 01/22/2024).

[55] P. Gijsbers et al. "Amlb: an automl benchmark". In: *Journal of Machine Learning Research* 25 (2024), pp. 1–65.

[56] R. V. Guha, Dan Brickley, and Steve Macbeth. *RDF 1.1 Primer*. W3C Working Group Note. W3C, June 2014. URL: https://www.w3.org/TR/rdf11-primer/.

[57] R. V. Guha, Dan Brickley, and Steve Macbeth. "Schema.org: evolution of structured data on the web". In: *Commun. ACM* 59.2 (Jan. 2016), 44–51. ISSN: 0001-0782. DOI: 10.1145/2844544. URL: https://doi.org/10.1145/2844544.

[58] Sang Hyu Hahn and Heiko Paulheim. "RDF2vec Embeddings for Updateable Knowledge Graphs – Reuse, Don't Retrain!" In: *The Semantic Web: ESWC 2024 Satellite Events: Hersonissos, Crete, Greece, May 26–30, 2024, Proceedings, Part I*. Hersonissos, Greece: Springer-Verlag, 2025, 217–222. ISBN: 978-3-031-78951-9. DOI: 10.1007/978-3-031-78952-6_30. URL: https://doi.org/10.1007/978-3-031-78952-6_30.

[59] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021), p. 106622.

[60] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool Publishers, 2011.

[61] Jeff Heflin. "OWL Web Ontology Language". In: *IEEE Internet Computing* 8.3 (2004), pp. 53–59.

[62] Martin Hepp. "GoodRelations: An ontology for describing products and services offers on the Web". In: *Proceedings of the 16th international conference on Knowledge engineering: practice and patterns*. Springer-Verlag. 2008, pp. 329–346.

[63] Mauricio A Hernández and Salvatore J Stolfo. "Real-world data is dirty: Data cleansing and the merge/purge problem". In: *Data mining and knowledge discovery* 2.1 (1998), pp. 9–37.

[64] Aidan Hogan et al. "Knowledge Graphs". In: *ACM Computing Surveys* 54.4 (2021), 71:1–71:37.

[65] Fred Hohman et al. "Gamut: A Design Probe to Understand How Data Scientists Understand Machine Learning Models". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2019. DOI: 10.1145/3290605.3300809.

[66] Ian Horrocks. "DAML+OIL: a description logic for the semantic web". In: *IEEE Data Engineering Bulletin* 25 (2002), pp. 4–9.

[67] Timothy Hospedales et al. "Meta-Learning in Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 44.09 (Sept. 2022), pp. 5149–5169. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2021.3079209. URL: https://doi.ieeecomputersociety.org/10.1109/TPAMI.2021.3079209.

[68] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-05317-8 978-3-030-05318-5. (Visited on 12/07/2022).

[69] Shaoxiong Ji et al. "A survey on knowledge graphs: Representation, acquisition, and applications". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.2 (2021), pp. 494–514.

[70] Hadi S. Jomaa, Lars Schmidt-Thieme, and Josif Grabocka. "Dataset2Vec: learning dataset meta-features". In: *Data Min. Knowl. Discov.* 35.3 (2021), pp. 964–985.

[71] Alexandros Kalousis and Melanie Hilario. "Feature Selection for Meta-learning". In: *Advances in Knowledge Discovery and Data Mining*. Ed. by David Cheung, Graham J. Williams, and Qing Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 222–233. ISBN: 978-3-540-45357-4. DOI: https://doi.org/10.1007/3-540-45357-1_26.

[72] C. Maria Keet et al. "The Data Mining OPtimization Ontology". In: *Journal of Web Semantics* 32 (May 2015), pp. 43–53. ISSN: 1570-8268. DOI: 10.1016/j.websem.2015.01.001. (Visited on 08/28/2024).

[73] Jörg-Uwe Kietz et al. ""Semantics Inside!" But Let's Not Tell the Data Miners: Intelligent Support for Data Mining". In: *The Semantic Web: Trends and Challenges*. Ed. by Valentina Presutti et al. Cham: Springer International Publishing, 2014, pp. 706–720. ISBN: 978-3-319-07443-6.

[74] Juhyeon Kim and Insung Ahn. "Infectious Disease Outbreak Prediction Using Media Articles with Machine Learning Models". In: *Scientific Reports* 11.1 (Feb. 2021), p. 4413. ISSN: 2045-2322.

[75] Jungtaek Kim, Saehoon Kim, and Seungjin Choi. *Learning to Warm-Start Bayesian Hyperparameter Optimization*. 2018. arXiv: 1710.06219 [stat.ML].

[76] Aaron Klein. "Reducing bias in AI-based financial services". In: *Technology & Information* (2020).

[77] Holger Knublauch. "SHACL in the Industry: A Personal Perspective". In: *Companion Proceedings of the Web Conference 2022*. 2022, pp. 1173–1176.

[78] Holger Knublauch and Dimitris Kontokostas. *Shapes Constraint Language (SHACL)*. W3C Recommendation. W3C, July 2017. URL: https://www.w3.org/TR/shacl/.

[79] Holger Knublauch and Dimitris Kontokostas. "Shapes Constraint Language (SHACL)". In: *W3C Recommendation 20 July 2017*. 2017. URL: https://www.w3.org/TR/2017/REC-shacl-20170720/.

[80] Lars Kotthoff et al. "Auto-WEKA: Automatic Model Selection and Hyperparameter Optimization in WEKA". In: *Automated Machine Learning*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Cham: Springer International Publishing, 2019, pp. 81–95. ISBN: 978-3-030-05317-8 978-3-030-05318-5. (Visited on 12/07/2022).

[81] Konstantina Kourou et al. "Machine Learning Applications in Cancer Prognosis and Prediction". In: *Computational and Structural Biotechnology Journal* 13 (Jan. 2015), pp. 8–17. ISSN: 2001-0370. (Visited on 12/19/2022).

[82] Agustinus Kristiadi et al. "Incorporating Literals into Knowledge Graph Embeddings". In: *arXiv preprint arXiv:1902.00898* (2019).

[83] Agustinus Kristiadi et al. "Incorporating Literals into Knowledge Graph Embeddings". In: *The Semantic Web – ISWC 2019*. Ed. by Chiara Ghidini et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 347–363. ISBN: 978-3-030-30793-6. DOI: 10.1007/978-3-030-30793-6_20.

[84] Timothee Lacroix, Nicolas Usunier, and Guillaume Obozinski. "Canonical Tensor Decomposition for Knowledge Base Completion". In: *Proceedings of the 35th International Conference on Machine Learning*. PMLR, July 2018, pp. 2863–2872. (Visited on 12/22/2023).

[85] Jens Lehmann et al. "DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web* 6.2 (2015), pp. 167–195.

[86] Rui Leite and Pavel Brazdil. "Exploiting Performance-based Similarity between Datasets in Metalearning". In: *MetaLearning@AAAI2021*. PMLR. PMLR, 2021, pp. 90–99.

[87] Rui Leite and Pavel Brazdil. "Exploiting Performance-based Similarity between Datasets in Metalearning". In: *Proceedings of the AAAI-21 Workshop on Meta-Learning*. 2021.

[88] Christiane Lemke, Marcin Budka, and Bogdan Gabrys. "Metalearning: a survey of trends and technologies". In: *Artificial Intelligence Review* DOI: 10.1007/s10462-013-9406-y (June 2013). DOI: 10.1007/s10462-013-9406-y.

[89] Maxwell W. Libbrecht and William Stafford Noble. "Machine Learning Applications in Genetics and Genomics". In: *Nature Reviews Genetics* 16.6 (June 2015), pp. 321–332. ISSN: 1471-0064. (Visited on 12/19/2022).

[90] Jingping Liu et al. "Beyond Entities: A Large-Scale Multi-Modal Knowledge Graph with Triplet Fact Grounding". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. Mar. 2024, pp. 18653–18661. DOI: 10.1609/aaai.v38i17.29828. (Visited on 08/13/2024).

[91] Mingwei Liu et al. "Task-Oriented ML/DL Library Recommendation Based on a Knowledge Graph". In: *IEEE Trans. Softw. Eng.* 49.8 (Aug. 2023), 4081–4096. ISSN: 0098-5589. DOI: 10.1109/TSE.2023.3285280. URL: https://doi.org/10.1109/TSE.2023.3285280.

[92] Ye Liu et al. "MMKG: Multi-modal Knowledge Graphs". In: *The Semantic Web*. Ed. by Pascal Hitzler et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 459–474. ISBN: 978-3-030-21348-0. DOI: 10.1007/978-3-030-21348-0_30.

[93] Xiao Long et al. "KGDM: A Diffusion Model to Capture Multiple Relation Semantics for Knowledge Graph Embedding". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. Mar. 2024, pp. 8850–8858. DOI: 10.1609/aaai.v38i8.28732. (Visited on 08/13/2024).

[94] Fadi Maali and John Erickson. *Data Catalog Vocabulary (DCAT)*. Recommendation. World Wide Web Consortium (W3C), Jan. 2014.

[95] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. "YAGO3: A Knowledge Base from Multilingual Wikipedias". In: *CIDR*. 2013.

[96] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M. Suchanek. "YAGO3: A Knowledge Base from Multilingual Wikipedias". In: *CIDR*. Jan. 2013. (Visited on 05/07/2024).

[97] James Malone et al. "The Software Ontology (SWO): A Resource for Reproducibility in Biomedical Data Analysis, Curation and Digital Preservation". In: *Journal of Biomedical Semantics* 5.1 (June 2014), p. 25. ISSN: 2041-1480. DOI: 10.1186/2041-1480-5-25. (Visited on 08/28/2024).

[98] Deborah L. McGuinness and Frank van Harmelen. *OWL Web Ontology Language Overview*. W3C Recommendation. W3C, Feb. 2004. URL: https://www.w3.org/TR/owl-features/.

[99] Lingbin Meng et al. "Machine Learning in Additive Manufacturing: A Review". In: *JOM* 72.6 (June 2020), pp. 2363–2377. ISSN: 1543-1851. (Visited on 12/19/2022).

[100] Donald Michie et al., eds. *Machine learning, neural and statistical classification*. USA: Ellis Horwood, 1995. ISBN: 013106360X.

[101] Ingo Mierswa et al. "YALE: Rapid Prototyping for Complex Data Mining Tasks". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. New York, NY, USA: Association for Computing Machinery, Aug. 2006, pp. 935–940. ISBN: 978-1-59593-339-3. DOI: 10.1145/1150402.1150531. (Visited on 02/13/2024).

[102] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. *Ludwig: a type-based declarative deep learning toolbox*. 2019. eprint: arXiv:1909.07930.

[103] Boris Motik et al. *OWL 2 Web Ontology Language Profiles (Second Edition)*. W3C Recommendation. W3C, Dec. 2012. URL: https://www.w3.org/TR/owl2-profiles/.

[104] Maximilian Nickel et al. "A Review of Relational Machine Learning for Knowledge Graphs". In: *Proceedings of the IEEE* 104.1 (2016), pp. 11–33. DOI: 10.1109/JPROC.2015.2483592.

[105] Paul Ogilvie and Jamie Callan. "Combining document representations for known-item search". In: *SIGIR03*. SIGIR '03. New York, NY, USA: Association for Computing Machinery, 2003, 143–150. ISBN: 1581136463.

[106] R. S. Olson et al. "PMLB: a large benchmark suite for machine learning evaluation and comparison". In: *BioData mining* 10 (2017), pp. 1–13.

[107] Guilherme Palumbo et al. "Real-Time Algorithm Recommendation Using Meta-Learning". In: *Ambient Intelligence—Software and Applications—13th International Symposium on Ambient Intelligence*. Ed. by Vicente Julián et al. Cham: Springer International Publishing, 2023, pp. 249–258. ISBN: 978-3-031-22356-3. DOI: 10.1007/978-3-031-22356-3_24.

[108] Pance Panov, Sašo Džeroski, and Larisa Soldatova. "OntoDM: An Ontology of Data Mining". In: *2008 IEEE International Conference on Data Mining Workshops*. Dec. 2008, pp. 752–760. DOI: 10.1109/ICDMW.2008.62. (Visited on 08/28/2024).

[109] Heiko Paulheim. "Knowledge graph refinement: A survey of approaches and evaluation methods". In: *Semantic web* 8.3 (2017), pp. 489–508.

[110] Fabian Pedregosa et al. "Scikit-Learn: Machine Learning in Python". In: *The Journal of Machine Learning Research* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.

[111] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and complexity of SPARQL". In: *ACM Transactions on Database Systems* 34.3 (2009), pp. 1–45.

[112] Rakesh Pimplikar and Sunita Sarawagi. "Answering table queries on the web using column keywords". In: *Proc. VLDB Endow.* 5.10 (June 2012), 908–919. ISSN: 2150-8097.

[113] Sebastian Pineda Arango and Josif Grabocka. "Deep Pipeline Embeddings for AutoML". In: *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '23. Long Beach, CA, USA: Association for Computing Machinery, 2023, 1907–1919. ISBN: 9798400701030. DOI: 10.1145/3580305.3599303. URL: https://doi.org/10.1145/3580305.3599303.

[114] Kelsey Piper. "Are we racing toward AI catastrophe?" In: *Vox* (2023).

[115] Neoklis Polyzotis et al. "Data Management Challenges in Production Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, 1723–1726. ISBN: 9781450341974. DOI: 10.1145/3035918.3054782. URL: https://doi.org/10.1145/3035918.3054782.

[116] Patryk Preisner and Heiko Paulheim. "Universal preprocessing operators for embedding knowledge graphs with literals". In: *arXiv preprint arXiv:2309.03023* (2023).

[117] Patryk Preisner and Heiko Paulheim. *Universal Preprocessing Operators for Embedding Knowledge Graphs with Literals*. Sept. 2023. DOI: 10.48550/arXiv.2309.03023. arXiv: 2309.03023 [cs]. (Visited on 01/08/2024).

[118] Eric Prud'hommeaux and Andy Seaborne. *SPARQL Query Language for RDF*. W3C Recommendation. W3C, Jan. 2008. URL: https://www.w3.org/TR/rdf-sparql-query/.

[119] Gustavo Correa Publio et al. *ML-Schema: Exposing the Semantics of Machine Learning with Schemas and Ontologies*. July 2018. arXiv: 1807.05351 [cs, stat]. (Visited on 08/28/2024).

[120] Daniel Rangel-Martinez, K. D. P. Nigam, and Luis A. Ricardez-Sandoval. "Machine Learning on Sustainable Energy: A Review and Outlook on Renewable Energy Systems, Catalysis, Smart Grid and Energy Storage". In: *Chemical Engineering Research and Design* 174 (Oct. 2021), pp. 414–441. ISSN: 0263-8762. (Visited on 12/19/2022).

[121] M. S. Mohamad Raub. "Bots, Bias and Big Data: Artificial Intelligence, Algorithmic Bias and Disparate Impact Liability in Hiring Practices". In: *Arkansas Law Review* (2018). URL: https://api.semanticscholar.org/CorpusID:169863807.

[122] Sergey Redyuk et al. "Assisted design of data science pipelines". In: *The VLDB Journal* (2024), pp. 1–25.

[123] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *EMNLP-IJCNLP 2019*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3982–3992.

[124] Petar Ristoski and Heiko Paulheim. "Rdf2vec: Rdf graph embeddings for data mining". In: *International semantic web conference*. Springer. 2016, pp. 498–514.

[125] Petar Ristoski and Heiko Paulheim. "RDF2Vec: RDF Graph Embeddings for Data Mining". In: *The Semantic Web – ISWC 2016*. Ed. by Paul Groth et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 498–514. ISBN: 978-3-319-46523-4. DOI: 10.1007/978-3-319-46523-4_30.

[126] Petar Ristoski et al. "RDF2Vec: RDF graph embeddings and their applications". In: *Semantic Web* 10.4 (2019), pp. 721–752.

[127] Kevin Roose. "A Coming-Out Party for Generative A.I., Silicon Valley's New Craze". In: *The New York Times* (2022).

[128] Alberto Rossi et al. "Knowledge Graph Embedding for Link Prediction: A Comparative Analysis". In: *Proceedings of the 20th International Conference on Web Engineering*. 2021, pp. 109–124.

[129] Daniel Ruffinelli, Samuel Broscheit, and Rainer Gemulla. "You CAN Teach an Old Dog New Tricks! On Training Knowledge Graph Embeddings". In: *International Conference on Learning Representations*. Sept. 2019. (Visited on 08/03/2023).

[130] Iqbal H. Sarker. "Machine Learning: Algorithms, Real-World Applications and Research Directions". In: *SN Computer Science* 2.3 (Mar. 2021), p. 160. ISSN: 2661-8907. (Visited on 12/07/2022).

[131] Michael Schlichtkrull et al. "Modeling relational data with graph convolutional networks". In: *European semantic web conference*. Springer. 2018, pp. 593–607.

[132] Michael Schlichtkrull et al. "Modeling Relational Data with Graph Convolutional Networks". In: *The Semantic Web*. Ed. by Aldo Gangemi et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 593–607. ISBN: 978-3-319-93417-4. DOI: 10.1007/978-3-319-93417-4_38.

[133] Kevin Scott. *Democratizing & Accelerating the Future of AI With Kevin Scott*. https://exchange.scale.com/public/videos/democratizing-and-accelerating-the-future-of-ai-with-kevin-scott. Accessed: 2023-03-09. Oct. 2021.

[134] Elizabeth Seger et al. "Democratising AI: Multiple Meanings, Goals, and Methods". In: *Proceedings of the 2023 AAAI/ACM Conference on AI, Ethics, and Society*. AIES '23. New York, NY, USA: Association for Computing Machinery, 2023, 715–722. ISBN: 9798400702310. DOI: 10.1145/3600211.3604693. URL: https://doi.org/10.1145/3600211.3604693.

[135] Bin Shang et al. "LAFA: Multimodal Knowledge Graph Completion with Link Aware Fusion and Aggregation". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. Mar. 2024, pp. 8957–8965. DOI: 10.1609/aaai.v38i8.28744. (Visited on 08/13/2024).

[136] Irene Solaiman. "The Gradient of Generative AI Release: Methods and Considerations". In: *Proceedings of the 2023 ACM Conference on Fairness, Accountability, and Transparency*. FAccT '23. New York, NY, USA: Association for Computing Machinery, 2023, 111–122. ISBN: 9798400701924. DOI: 10.1145/3593013.3593981. URL: https://doi.org/10.1145/3593013.3593981.

[137] Renan Souza et al. "Provenance Data in the Machine Learning Lifecycle in Computational Science and Engineering". In: *2019 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. Nov. 2019, pp. 1–10. DOI: 10.1109/WORKS49585.2019.00006. (Visited on 08/28/2024).

[138] SPARQL Working Group. *SPARQL 1.1 Overview*. W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/sparql11-overview/.

[139] Marieke Stolte et al. "Methods for quantifying dataset similarity: a review, taxonomy and comparison". In: *Statistics Surveys* 18.none (Jan. 2024). ISSN: 1935-7516.

[140] Rudi Studer, V. Richard Benjamins, and Dieter Fensel. "Knowledge engineering: Principles and methods". In: *Data & Knowledge Engineering* 25.1-2 (1998), pp. 161–197.

[141] Zhiqing Sun et al. *RotatE: Knowledge Graph Embedding by Relational Rotation in Complex Space*. Feb. 2019. DOI: 10.48550/arXiv.1902.10197. arXiv: 1902.10197 [cs, stat]. (Visited on 01/08/2024).

[142] The Gene Ontology Consortium. "The Gene Ontology (GO) database and informatics resource". In: *Nucleic Acids Research* 32.suppl_1 (2004), pp. D258–D261.

[143] Kristina Toutanova and Danqi Chen. "Observed versus Latent Features for Knowledge Base and Text Inference". In: *Proceedings of the 3rd Workshop on Continuous Vector Space Models and Their Compositionality*. Ed. by Alexandre Allauzen et al. Beijing, China: Association for Computational Linguistics, July 2015, pp. 57–66. DOI: 10.18653/v1/W15-4007. (Visited on 01/15/2024).

[144] Théo Trouillon et al. "Complex Embeddings for Simple Link Prediction". In: *Proceedings of The 33rd International Conference on Machine Learning*. PMLR, June 2016, pp. 2071–2080. (Visited on 01/08/2024).

[145] Fabio Urbina et al. "Dual use of artificial-intelligence-powered drug discovery". In: *Nature Machine Intelligence* 4.3 (2022), pp. 189–191.

[146] Joaquin Vanschoren. "Meta-Learning: A Survey". In: *ArXiv* abs/1810.03548 (2018).

[147] Joaquin Vanschoren et al. "Experiment Databases". In: *Machine Learning* 87.2 (May 2012), pp. 127–158. ISSN: 1573-0565. DOI: 10.1007/s10994-011-5277-0. (Visited on 08/28/2024).

[148] Joaquin Vanschoren et al. "OpenML: Networked Science in Machine Learning". In: *SIGKDD Expl.* 15.2 (2013), pp. 49–60.

[149] James Vincent. "Meta's powerful AI language model has leaked online—what happens now?" In: *The Verge* (2023). URL: https://www.theverge.com/2023/3/8/23629362/meta-ai-language-model-llama-leak-online-misuse.

[150] Denny Vrandečić and Markus Krötzsch. "Wikidata: a free collaborative knowledgebase". In: *Communications of the ACM* 57.10 (2014), pp. 78–85.

[151] W3C OWL Working Group. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. W3C Recommendation. W3C, Dec. 2012. URL: https://www.w3.org/TR/owl2-overview/.

[152] Jiacheng Wang et al. "Augmenting Knowledge Graphs for Better Link Prediction". In: *arXiv preprint arXiv:2203.13965* (2022).

[153] Jiang Wang et al. *Augmenting Knowledge Graphs for Better Link Prediction*. Apr. 2022. DOI: 10.48550/arXiv.2203.13965. arXiv: 2203.13965 [cs]. (Visited on 01/09/2024).

[154] Quan Wang et al. "Knowledge Graph Embedding: A Survey of Approaches and Applications". In: *IEEE Transactions on Knowledge and Data Engineering* 29.12 (2017), pp. 2724–2743.

[155] W. X. Wilcke et al. *End-to-End Entity Classification on Multimodal Knowledge Graphs*. Mar. 2020. DOI: 10.48550/arXiv.2003.12383. arXiv: 2003.12383 [cs]. (Visited on 01/15/2024).

[156] David Wood, Markus Lanthaler, and Richard Cyganiak. *Linked Data Platform 1.0*. W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/ldp/.

[157] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE Trans. Neural Netw.* 32.1 (2020), pp. 4–24.

[158] Da Xu et al. "Product Knowledge Graph Embedding for E-commerce". In: *Proceedings of the 13th International Conference on Web Search and Data Mining*. WSDM '20. New York, NY, USA: Association for Computing Machinery, Jan. 2020, pp. 672–680. ISBN: 978-1-4503-6822-3. DOI: 10.1145/3336191.3371778. (Visited on 01/08/2024).

[159] Bishan Yang et al. "Embedding Entities and Relations for Learning and Inference in Knowledge Bases". In: *CoRR* (Dec. 2014). (Visited on 01/08/2024).

[160] Danni Yu, Hannah Rosenfeld, and Abhishek Gupta. *The 'AI divide' between the Global North and Global South | World Economic Forum*. [Online; accessed 2025-10-21]. Jan. 2023. URL: https://www.weforum.org/stories/2023/01/davos23-ai-divide-global-north-global-south/.

[161] Shuo Zhang and Krisztian Balog. "Ad Hoc Table Retrieval using Semantic Similarity". In: *WWW18*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, 1553–1562. ISBN: 9781450356398.

[162] Yuxin Zhang et al. "Fine-Grained Evaluation of Knowledge Graph Embedding Models in Downstream Tasks". In: *Web and Big Data*. Ed. by Xin Wang et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 242–256. ISBN: 978-3-030-60259-8. DOI: 10.1007/978-3-030-60259-8_19.

[163] Shuangjia Zheng et al. "PharmKG: a dedicated knowledge graph benchmark for bomedical data mining". In: *Briefings in bioinformatics* 22.4 (2021), bbaa344.