

Automation of Data Science Workflows via Knowledge Graphs and Table Representation Learning

Mossad Helali

**A Thesis in The Department of
Computer Science and Software Engineering**

**Presented in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy (Computer Science) at
Concordia University
Montréal, Québec, Canada**

December 2025

© Mossad Helali, 2026

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mr. Mossad Helali**

Entitled: **Automation of Data Science Workflows via Knowledge Graphs and
Table Representation Learning**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to
originality and quality.

Signed by the Final Examining Committee:

Prof. Andreas Athienitis Chair

Prof. Ziawasch Abedjan External Examiner

Prof. Juergen Rilling Examiner

Prof. Eugene Belilovsky Examiner

Prof. Sandra Céspedes Examiner

Prof. Essam Mansour Supervisor

Approved by _____
Joey Paquet , Chair
Department of Computer Science and Software Engineering

_____ 2025

Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Automation of Data Science Workflows via Knowledge Graphs and Table Representation Learning

Mossad Helali, Ph.D.

Concordia University, 2026

The rapid growth of open and collaborative data science platforms has led to large, disconnected collections of artifacts, namely, datasets and code pipelines. This separation makes it difficult to reuse knowledge and automate complex data science workflows. This thesis shows that by combining the semantic representations of Knowledge Graphs (KGs) and Table Representation Learning (TRL) into a unified foundational layer, these limitations can be overcome.

The presented research demonstrates the versatility of this semantic layer through three distinct paradigms, each addressing a critical stage of the data science lifecycle. First, data discovery is formulated as a scalable, direct query process against the KG, enabling expressive, semantic searches. Second, Automated Machine Learning (AutoML) is approached as a meta-learning task where Graph Neural Networks (GNNs) learn from the collective experience encoded in the graph's structure to recommend optimal pipelines. Third, Automated Exploratory Data Analysis (AutoEDA) is realized via a Retrieval-Augmented Generation (RAG) framework that grounds Large Language Models (LLMs) with factual, verifiable context from the KG.

Through extensive evaluations on standard benchmarks, the systems developed in this research show significant improvements in accuracy, scalability, and cost-effectiveness over state-of-the-art methods. Together, this work presents a new and powerful architectural pattern for data science automation, proving that a unified semantic representation is a practical foundation for building the next generation of intelligent and collaborative data science tools.

Acknowledgments

I offer my utmost gratitude to **God Almighty (Allah)**, whose countless blessings gave me the knowledge and perseverance to complete this work. This journey has been as much spiritual as it has been scientific, during which I have deeply felt the manifestation of His beautiful names, Al-Lateef and Al-Kareem.

I sincerely thank my supervisor, **Dr. Essam Mansour**, for his support and guidance throughout my doctoral studies. His patience and trust in letting me explore my own research path have been invaluable to my growth as a researcher. I was also fortunate to work with brilliant **collaborators** at Google, IBM Research, and other academic institutions. These collaborations were instrumental in shaping the projects in this thesis and broadening my perspective.

To my **beloved parents, brother, sisters, and wife**: I owe you a gratitude that words cannot capture. Your unconditional love, encouragement, and guidance were my constant inspiration and strength. Lastly, my **community at Al-Madinah Center** has been a home away from home, and the unforgettable memories I made with them have profoundly shaped my time in Montreal.

Contribution Statement

This thesis presents original research conducted by the author, Mossad Helali, under the supervision of Dr. Essam Mansour. The core contributions detailed in Chapters 3, 4, and 5 are based on work that has been published in peer-reviewed international conferences. This statement clarifies the author's specific contributions to each publication.

Chapter 3: Dataset Embeddings for Effective Data Discovery

- **Publication:** Mossad Helali, Niki Monjazebe, Shubham Vashisth, Philippe Carrier, Ahmed Helal, Antonio Cavalcante, Khaled Ammar, Katja Hose, and Essam Mansour. "KGLiDS: A Platform for Semantic Abstraction, Linking, and Automation of Data Science." 2024 IEEE 40th International Conference on Data Engineering (ICDE). 2024, pp. 179–192. DOI: 10.1109/ICDE60146.2024.00021.
- **My Contributions:** I was the leading author of the published KGLiDS paper. I designed and implemented in my own a new approach for scalable embedding-based data profiling (CoLR). This includes the fine-grained typing and the training of the fine-grained CoLR models. Moreover, I integrated the CoLR models into the KGLiDS system, which enabled more effective and efficient construction of the Data Global Schema. I also contributed with other students to the pipeline abstraction and the KGLiDS interfaces, which I did not report in this thesis. I thank Khaled Ammar and Katja Hose for their thorough discussion and support over the timeline of the project.

Chapter 4: AutoML via Graph Neural Networks

- **Publication:** Mossad Helali, Essam Mansour, Ibrahim Abdelaziz, Julian Dolby, and Kavitha Srinivas. "A scalable AutoML approach based on graph neural networks." Proceedings of the VLDB Endowment 15.11 (2022), pp. 2428–2436. DOI: 10.14778/3551793.3551804
- **My Contributions:** I was the main leading researcher of the project. I integrated my CoLR approach into IBM's GraphGen4Code system and utilized the integrated system to solve the AutoML problem as a graph generation problem with graph neural networks (KGpip). I thank the IBM team (Ibrahim Abdelaziz, Julian Dolby, and Kavitha Srinivas) for their thorough discussion and support over the timeline of the project.

Chapter 5: Automated Data Exploration via RAG and AI Agents

- **Publication:** Mossad Helali, Yutai Luo, Tae Jun Ham, Jim Plotts, Ashwin Chaugule, Jichuan Chang, Parthasarathy Ranganathan, and Essam Mansour. "Reliable and Cost-Effective Exploratory Data Analysis via Graph-Guided RAG". EMNLP 2025 (main conference).
- **My Contributions:** I was the main leading researcher of the project. I utilized KGLiDS and Kaggle datasets to generate a base knowledge graph (KG) for data science, and extended the semantics of this KG with exploratory data analysis (EDA) operations. Then, I used the extended KG to solve the problem of automating EDA pipeline generation using Retrieval-Augmented Generation (RAG) and AI agents. I thank the Google and Kaggle teams (Yutai Luo, Tae Jun Ham, Jim Plotts, Ashwin Chaugule, Jichuan Chang, and Parthasarathy Ranganathan) for their thorough discussion and support over the timeline of the project.

Table of Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	Motivation	3
1.3	Challenges	5
1.4	Thesis Objectives	8
1.5	Thesis Outline	9
2	Background	10
2.1	Knowledge Graphs	10
2.2	Data Discovery	11
2.3	Table Representation Learning	13
2.4	Data Discovery with Knowledge Graphs	14
2.5	Semantic Pipeline Abstraction	15
2.6	The Knowledge Graph for Linked Data Science (KGLIDS)	17
2.7	Retrieval-Augmented Generation	19
2.8	Automated Machine Learning (AutoML)	20
2.9	Exploratory Data Analysis (EDA)	21
3	Dataset Embeddings for Effective Data Discovery	23
3.1	Embedding-Based Data Profiling For Data Discovery	25
3.1.1	Fine-Grained Data Type Inference	27
3.1.2	Column Metadata and Statistics	28

3.1.3	Column and Table Embeddings	28
3.1.3.1	Column Name Embedding	29
3.1.3.2	Column Value Embedding	29
3.2	Knowledge Graph Construction	31
3.2.1	Data Global Schema Structure	31
3.2.2	Data Global Schema Construction Process	32
3.2.3	Storage and Querying	35
3.3	Experimental Evaluation	36
3.3.1	Experimental Setup.	36
3.3.2	Data Discovery Systems	37
3.3.3	Accuracy of Data Discovery	38
3.3.4	System Scalability Analysis	39
3.3.5	Ablation Study	41
3.3.5.1	Impact of column names and values on data discovery	42
3.3.5.2	Impact of fine-grained vs. coarse-grained CoLR models	42
3.3.5.3	Strong scalability test of Data Profiler and KG Builder	42
3.4	Related Work	43
3.4.1	Data Discovery Systems	43
3.4.2	Table representation learning (TRL)	44
3.5	Chapter Summary	44
4	AutoML via Graph Neural Networks	46
4.1	The KGpip Scalable Meta-Learning	48
4.1.1	Graph Representation of Code Semantics	49
4.1.2	MetaPip: from Code to Pipeline Semantics	51
4.1.3	Dataset Representation Learning	52
4.2	The KGpip Pipeline Automation	53
4.2.1	Graph Generation for ML Pipelines	53
4.2.2	Hyperparameter Optimization	55

4.3	Experiments	56
4.3.1	Benchmarks	56
4.3.2	Baselines	58
4.3.3	Training Setup	58
4.3.4	Comparison with Existing Systems	59
4.3.5	Ablation Study	62
4.3.5.1	The effectiveness of MetaPip	62
4.3.5.2	The KGpip meta-learning quality	63
4.3.5.3	The KGpip meta-learning diversity	63
4.4	Related Work	64
4.5	Chapter Summary	66
5	Automated Data Exploration via RAG and AI Agents	67
5.1	Knowledge Graph Semantic Enrichment	70
5.1.1	Base Knowledge Graph Generation	70
5.1.2	Taxonomy of EDA Operations	72
5.1.3	KG-Guided EDA Semantic Extraction	74
5.1.4	Knowledge Graph Augmentation	75
5.2	EDA Notebook Generation	76
5.2.1	Embedding-based EDA Retrieval	76
5.2.2	EDA Alignment	77
5.2.3	EDA Refinement	77
5.2.4	EDA Coding Agent	79
5.3	Experimental Evaluation	80
5.3.1	Benchmarks	80
5.3.2	Baselines	81
5.3.3	Evaluation Metrics	82
5.3.4	EDA Retrieval Accuracy	83
5.3.5	EDA Code Reliability	83

5.3.6	EDA Visual Quality	83
5.3.7	Open vs. Commercial LLMs	84
5.3.8	LLM Token Usage	85
5.3.9	Ablation Studies	86
5.3.9.1	Impact of LLM Refinement	86
5.3.9.2	Impact of EDA Coding Agent	87
5.3.9.3	Impact of EDA Retrieval Methods	87
5.3.9.4	Number of Retrieved Similar Columns	88
5.4	Related Work	89
5.5	Chapter Summary	90
6	Future Work Directions	91
6.1	Fully Agentic Systems	91
6.2	Diverse Data Modalities	92
6.3	Integration of Broader Data Science Context	92
6.4	Temporal Knowledge Graph for Data and Model Lineage	93
7	Conclusion	94
	References	96
	Appendix A	112
A.1	RAGvis Prompt Templates	112
A.2	RAGvis VLM-as-Judge Prompt Template	118

Chapter 1

Introduction

1.1 Problem Statement

Data science is a multidisciplinary field that uses scientific methods, processes, algorithms, and systems to extract knowledge and insights from structured and unstructured data. Its importance has grown exponentially, driving innovation and decision-making across nearly every sector of the modern economy and academia [11]. At its core, the practice of data science revolves around two primary artifacts: datasets, which serve as the raw material, and code, which manages the data science lifecycle. This lifecycle is a complex, iterative process comprising stages such as data discovery and acquisition, data exploration and analysis, data cleaning and feature engineering, model development and training, and deployment. The goal is to transform raw data into actionable insights, predictive models, and intelligent applications.

The recent surge in the popularity and adoption of data science is well-documented. In academia, the field's growth is staggering. A large-scale bibliometric analysis of 1.6 million research papers found that over 1.2 million non-computer scientists authored their first AI-related publication between 2012 and 2020, signaling a rapid and broad adoption across disciplinary boundaries [11]. This interdisciplinary collaboration is a key driver of AI adoption, with institutional support and access to computer science expertise being strong predictors of uptake [11]. This demand is also reflected in education; the number of AI-related curricula has increased dramatically worldwide, with one in five U.S. computer science PhDs now specializing in AI [11]. In the United States alone,

over 450 universities now offer dedicated data science degrees or certificate programs [88].

This trend is mirrored in industry, where data science is no longer a niche capability but a core business function. Kaggle’s annual State of Data Science surveys, which gather input from tens of thousands of practitioners, consistently show widespread adoption of machine learning across sectors like technology, finance, and healthcare for tasks ranging from predictive analytics to business intelligence [61]. Recent industry surveys further confirm this trend, with approximately 45% of organizations reporting the use of AI and data science in production and 66% actively considering it for core functions like engineering, R&D, and customer support [5]. To support this growth, collaborative open-source platforms like Kaggle [62] and OpenML [103] have become central hubs, hosting tens of thousands of datasets and hundreds of thousands of associated data science pipelines.

Yet, despite the scale of these efforts and the collaborative potential of these platforms, a fundamental problem persists: existing data science systems lack a holistic approach to collecting, organizing, and analyzing the artifacts they host. Datasets and the code that operates on them are treated as disconnected, isolated entities. There is little to no systemic support for users to learn from the vast, implicit knowledge embedded in the connections between datasets and pipelines. Data scientists are often forced to work in silos, unable to effectively build upon the prior work, experiences, and best practices of the broader community.

This disconnect is a well-documented source of inefficiency and redundancy. A study on data science pipeline architectures [12] found that pipelines are often constructed in an "ad hoc manner," highlighting a lack of systematic methodology. Significant inconsistencies have been observed; for example, while 96% of pipelines included preprocessing and model building stages, only 52% incorporated post-processing layers, demonstrating a fragmented and incomplete approach to workflow construction [12].

Further research on the organizational challenges of data science reveals the direct consequences of this siloed approach. A study found that because each team builds pipelines for their use case in isolation of other teams, critical activities are constantly repeated, leading to redundant storage of data and unnecessarily repeated effort [82]. This redundancy is exacerbated by severe communication gaps; the study noted that teams working on similar problems were also unaware of such

repetition. When pipelines span multiple organizations, these dependency issues are even more pronounced, with external collaborators having little knowledge of internal activities [82]. This lack of a unified, semantic view of the data science ecosystem is a significant barrier, hindering knowledge reuse, impeding collaboration, and ultimately limiting the efficiency and reliability of data science workflows.

1.2 Motivation

The limitations inherent in a landscape of disconnected data and code artifacts point to a clear and pressing need for a more integrated and intelligent data science ecosystem. The central motivation for this thesis is that by building a holistic, semantic representation of data science artifacts, we can fundamentally improve how data scientists work. Such a representation would create a connected ecosystem where the vast, implicit knowledge from countless projects is made explicit, shareable, and reusable. If this were possible, data scientists would no longer be forced to "reinvent the wheel," and could instead dedicate their expertise to tackling novel and more complex challenges.

To achieve this, a crucial prerequisite is to move beyond the superficial specifics of individual artifacts. The key is to abstract away from implementation details such as file formats (CSV, JSON), programming languages (Python, R), or specific libraries (pandas, scikit-learn). Instead, the goal is to capture their underlying semantics. This thesis demonstrates that a powerful foundation for this semantic layer can be constructed from two complementary representations: Knowledge Graphs (KGs) to model the explicit, symbolic relationships between artifacts, and dataset embeddings (specifically, Column Learned Representations or CoLR) to capture the implicit, semantic similarity of their content. The KG can represent that a specific script file reads a particular dataset, while the embeddings can reveal that a column in that dataset is semantically equivalent to another column in an entirely different dataset, even if their names differ.

However, simply creating this representation is not enough. Its true value is realized in its application. Therefore, the primary motivation of this thesis is not just to propose this unified representation, but to demonstrate its power and versatility by exploring three distinct and powerful

paradigms for its use in automating data science workflows. Each paradigm leverages the KG and embeddings in a unique way to solve critical tasks across the data science lifecycle, showcasing the breadth of possibilities unlocked by this foundational semantic layer.

The first paradigm addresses the fundamental task of data discovery by treating the KG as a queryable, symbolic database. This approach is motivated by the need to move beyond the limitations of basic keyword search, which restricts discovery to known filenames or exact feature names. By leveraging the structured nature of the KG and the power of graph query languages like SPARQL, we can enable a far more sophisticated and intentional form of discovery. Data scientists are no longer limited to asking "Where is the titanic.csv file?"; they can now pose complex, semantic questions such as, "Find all datasets that contain columns semantically similar to 'customer age' and 'monthly revenue' and can be joined with my current marketing table." This paradigm directly tackles the problem of isolated artifacts [82] by making the relationships between them explicit and queryable, transforming the data lake from a simple repository into a navigable network of interconnected knowledge.

The second paradigm addresses the task of Automated Machine Learning (AutoML) and tackles the challenge of ad-hoc and inconsistent ML pipeline construction [12]. Graph Neural Networks are utilized to learn from the collective experience of data scientists on public platforms like Kaggle [62]. While a direct query can find a single similar artifact, a GNN can learn from the entire structure of the ML pipelines graph, identifying higher-order patterns of what makes an ML pipeline successful. The KG, which captures thousands of datasets and the pipelines applied to them, serves as a perfect structure for a GNN to learn the latent relationships between dataset characteristics and optimal ML model configurations. This approach motivates a shift from manual or brute-force AutoML to a data-driven, meta-learning process. For a new, unseen dataset, the system can use its embedding to locate it within the graph and then use a GNN to recommend a pipeline architecture based on the aggregated patterns of successful pipelines for its nearest semantic neighbors. This paradigm demonstrates how the holistic representation can be used to automate the dissemination of best practices at scale.

The third paradigm addresses the problem of Automated Exploratory Data Analysis (Auto EDA) and is motivated by the need to bridge the gap between the immense potential of Large

Language Models (LLMs) and their practical limitations in enterprise or specialized data science contexts. While LLMs have shown promising results on a variety of tasks, they lack specific, factual knowledge about a given data lake, leading to generic or incorrect outputs. However, a Retrieval-Augmented Generation (RAG) paradigm can address this gap by using the knowledge graph as a reliable, external "brain" to guide LLM reasoning. Instead of asking an LLM to "explore this dataset" from scratch, a RAG system would first retrieve concrete context from the KG, such as examples of successful analyses performed on semantically similar datasets. This structured information is then provided to the LLM as part of its prompt, guiding it to generate code and insights that are accurate, relevant, and directly applicable to the dataset at hand. This demonstrates how the KG and embeddings can serve as a critical component in the modern AI stack, transforming LLMs from general-purpose tools into context-aware, expert assistants for data science.

By exploring these three distinct paradigms, this thesis aims to provide comprehensive evidence that a unified semantic representation is a practical and powerful foundation for the next generation of automated, intelligent, and collaborative data science platforms.

1.3 Challenges

While the vision of a semantically unified and automated data science ecosystem is compelling, realizing it requires overcoming significant technical and conceptual obstacles, which are to be addressed in this thesis. The challenges can be broadly categorized into foundational issues that affect the entire task spectrum and specific difficulties that arise within each of the three targeted automation paradigms.

Foundational Challenges in Representation and Scalability

At the most fundamental level, the primary challenge is the lack of a suitable representation for data science artifacts. Existing platforms and systems [65, 13, 29, 1, 6] treat datasets, code, and models as disconnected entities, providing no mechanism to capture the rich web of relationships between them. Therefore, the first major hurdle is designing a holistic and interoperable knowledge graph to capture these semantics. Such a KG must link entities like datasets, columns, and pipelines

with both explicit, structured relations and implicit, semantic similarity captured by embeddings. Designing the node and edge types in this KG and algorithmically generating it from raw artifacts is a major challenge, as no such comprehensive representation currently exists in the literature or in practice.

Beyond the design of the knowledge graph is the critical challenge of ensuring end-to-end scalability. Building this KG for a real-world data lake is a massive computational task. It requires an architecture that can scale across the entire information lifecycle: from the initial, parallelized profiling of thousands of heterogeneous datasets and the generation of high-dimensional embeddings for hundreds of thousands of columns, to the scalable processing of millions of existing data science pipelines. Furthermore, the system must support complex hybrid queries that combine traditional graph traversal with vector similarity search, all while maintaining interactive performance. This necessitates a carefully architected system that is scalable and efficient at every stage.

Challenges in Scalable, Embedding-Based Data Discovery

For the first paradigm of data discovery, a key challenge is that the use of deep learning models to generate dataset profiles and perform embedding-based discovery has seen limited attention. While embedding-based search has the potential to dramatically improve the scalability of indexing and querying in data lakes with tens of thousands of datasets, existing table representation learning techniques are often not optimized for the specific task of data discovery and require significant adaptation to be effective.

This leads to the challenge of achieving accurate similarity matching across heterogeneous data types. A successful discovery system must learn to combine signals from both human-provided column names (label-based embeddings) and the raw data itself (content-based embeddings). The system must be nuanced enough to correctly identify both unionable and joinable tables while avoiding the common pitfall of generating a high number of false positive matches. Finally, there is the challenge of enabling an expressive yet usable discovery interface. The goal is to empower users with a powerful, yet simple, method of querying the underlying KG and embeddings, without burdening them with the need to learn an overly complex new query language.

Challenges in Meta-Learning for Automated Machine Learning

In the AutoML paradigm, a central difficulty is extracting representative pipeline semantics from code at scale. Real-world data science notebooks are noisy, often containing a mix of exploratory code, visualizations, and multiple experimental pipelines. Developing static analysis techniques that can robustly parse these evolving artifacts to abstract the core machine learning logic and reliably link it to the correct input datasets is a major challenge.

Once semantics are extracted, the next hurdle is to effectively learn from them. This involves selecting appropriate Graph Neural Network architectures that are well-suited for learning from the heterogeneous graph of datasets and pipelines. Subsequently, the system must generate valid and task-appropriate pipeline graphs. This requires a generative model that can balance diversity (exploring novel pipeline structures) against determinism (recommending proven, high-quality solutions) to compose preprocessors and estimators into executable skeletons. Finally, a significant integration challenge is to seamlessly combine these meta-learned starting points with existing Hyperparameter Optimization (HPO) frameworks to construct fully optimized ML pipelines under the realistic time and computational budgets faced by data scientists.

Challenges in Automating Exploratory Data Analysis

For the automated EDA paradigm, a distinct set of challenges emerges. A foundational problem is the construction of a purpose-built, EDA-aware knowledge graph from a variety of real-world EDA pipelines. This KG must accurately represent EDA operations and link them with the specific columns being analyzed and the corresponding code statements. Creating these high-level semantic nodes requires interpreting the analytical intent behind code blocks in EDA pipelines, a task complicated by the diversity of EDA operations and the common practice of extensive data preprocessing. The system must backtrack through complex data dependencies, handling variable renames and transformations that obscure the data lineage.

After extraction, the system must overcome the challenge of retrieving and aligning these historical EDA operations to new, unseen datasets. This is a complex mapping problem that involves ensuring column type compatibility and accurately ranking retrieved EDA operations. Perhaps the

most critical challenge is ensuring the reliability of the generated code. This requires the implementation of a robust method that allows an LLM to generate source code and debug it. In addition, the end-to-end RAG approach must be efficient across LLMs of varying capabilities (from large proprietary models to smaller open-source ones), while minimizing LLM token usage. Finally, a significant challenge is the objective evaluation of automated EDA. Given the inherent subjectivity of what constitutes a "good" exploration, this requires developing a novel, multi-faceted evaluation framework that combines quantitative and qualitative metrics to demonstrate the system's true utility and its ability to generalize.

1.4 Thesis Objectives

To address the previously outlined problems and challenges, this thesis pursues the central objective of designing, implementing, and evaluating a unified framework for automating data science workflows based on a holistic, semantic representation of artifacts. The goal is to demonstrate that by combining Knowledge Graphs (KGs) and Column Learned Representations (CoLR), it is possible to create a foundational semantic layer that makes data science more scalable, efficient, and intelligent.

To validate this central thesis, the research is guided by three specific, interconnected objectives. These have been strategically chosen to demonstrate the framework's utility across critical, successive stages of the data science lifecycle, from initial data acquisition to final model creation. Each objective corresponds to a distinct paradigm for leveraging the semantic representation:

- (1) **To develop a scalable and accurate system for embedding-based data discovery.** This objective involves creating a novel data profiling pipeline that uses fine-grained, type-aware CoLR models to generate high-quality column embeddings. The goal is to construct a Data Global Schema—a knowledge graph of the data lake—and demonstrate that executing discovery tasks as direct queries against this graph leads to significant improvements in both accuracy and performance compared to state-of-the-art systems.

- (2) **To design a data-driven, meta-learning approach for automated machine learning (AutoML).** This objective focuses on leveraging the entire knowledge graph to guide the generation of ML pipelines. The goal is to use Graph Neural Networks (GNNs) to learn from the collective experience encoded in thousands of existing datasets and pipelines. The objective is to show that this paradigm can effectively recommend high-performing pipeline architectures for new datasets, significantly improving the efficiency of existing AutoML systems.
- (3) **To create a reliable and cost-effective system for automated exploratory data analysis via Retrieval-Augmented Generation (RAG).** For the crucial exploratory stage that precedes and informs modeling, the third objective is to mitigate the limitations of LLMs by using the knowledge graph as a factual, external memory. This research proposes and evaluates a RAG framework where the KG provides specific, verifiable context to an LLM. A key outcome will be to demonstrate that this grounding enables even smaller, open-source LLMs to generate reliable and relevant EDA code, making automated EDA more feasible in low-resource and privacy-sensitive scenarios.

By achieving these three objectives, this thesis aims to provide comprehensive evidence that a unified semantic representation is a practical and powerful foundation for a new generation of data science automation tools.

1.5 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 reviews the relevant background and related work. Chapter 3 presents the data discovery framework, which uses direct semantic queries against a knowledge graph built with fine-grained CoLR models. Chapter 4 details the AutoML approach, which leverages Graph Neural Networks to learn from the knowledge graph and recommend optimal machine learning pipelines. Chapter 5 introduces the AutoEDA system, which employs a Retrieval-Augmented Generation framework to ground Large Language Models with the knowledge graph. Finally, Chapters 6 and 7 conclude the thesis with directions for future research and a summary of the contributions presented.

Chapter 2

Background

This section reviews important concepts and techniques we utilize in our proposed research.

2.1 Knowledge Graphs

Knowledge graphs have emerged as a powerful framework for representing and organizing information in a way that is both machine-readable and intuitive for humans to visualize. They provide a structured representation of real-world entities, their attributes, and relationships, enabling a wide range of intelligent applications across various domains.

A knowledge graph is a directed graph consisting of nodes and edges. Nodes represent real-world or abstract entities (e.g., people, places, concepts), while edges represent the relationships between them. Each edge is assigned a predicate that defines the type of relationship (e.g., "is employed by," "is located in," "is a type of"). This structured representation allows for the encoding of complex networks of interconnected information, capturing both factual knowledge and semantic relationships. [48]

The key components of a knowledge graph include: 1) Triples, where each triple represents a single fact as a subject-predicate-object statement (e.g., Montreal - *isLocatedIn* - Canada), 2) A schema or ontology, which are formal definitions of the types of entities and relationships that can exist in the knowledge graph; knowledge graph schemas enable reasoning over the graph, i.e. inferring relationships that are not explicitly defined (e.g. if DepA - *subDepartmentOf* - DepB and

DepB - *subDepartmentOf* DepC, then DepA - *subDepartmentOf* - DepC), 3) A graph store, which stores the graph triples, builds indices to facilitate graph search, and provides support to query the graph.

The ability of knowledge graphs to represent and connect information in a meaningful way has led to their adoption in a wide range of applications, including: search [57], product recommendation [25], question answering [51], and data integration [21]. Some example large-scale knowledge graphs used in practice are: The Google Knowledge Graph, which contains billions of facts and is used to enhance Google’s search engine, DBPedia [7], which is extracted from Wikipedia and contains information on millions of entities, and Wikidata [107], which is also extracted from Wikipedia but is collaboratively edited.

These examples highlight the diverse applications of knowledge graphs and their potential to transform how we access, organize, and utilize information. In our research, we propose a novel use case of knowledge graphs for data science.

2.2 Data Discovery

Data discovery is the task of efficiently retrieving the most similar tables or columns in a data lake to a given query table or column. With the growing sizes of data lakes and the growing interest in data science, data discovery has become a significant task for data scientists to find datasets of interest efficiently and accurately [76].

The definition of table or column similarity is based on a data discovery system’s objective. Common discovery objectives include unionability, where the goal is to find tables that can be combined row-wise to increase data instances, and joinability, which seeks tables that can be combined column-wise to add features. The field of data discovery research is concerned with improving one or more of the key components involved in the process, including data profiling, data catalog construction, and discovery operations.

Data profiling involves extracting metadata and statistics to create data profiles that characterize and summarize individual columns or tables. Data profiling techniques range from basic statistics to machine learning-based profiling. Data catalog construction is the task of creating specialized

data structures to improve the efficiency of data discovery systems. Two common approaches for creating data catalogs are Locality-Sensitive Hashing (LSH) indexes and graph-based structures. Data discovery operations are the main techniques, where the generated profiles and data catalogs are utilized to discover relevant datasets in a data lake. There are various approaches with different levels of complexity. Keyword search is a basic and widely used technique implemented, for example, in Google Dataset Search[85]. Ontology-based discovery, utilized for instance by SANTOS[65], leverages knowledge graphs to model the domain and relationships between concepts. Profile-based discovery, employed in systems like Starmie[29], utilizes column or table profiles to identify relevant data items.

Data discovery evaluation focuses on accuracy (finding the right tables) and efficiency (speed of finding them). Benchmarks used for this evaluation can be real-world (e.g., [13]) or synthetic (e.g., [65]). The most common evaluation metrics for data discovery systems are precision and recall. Given a query table q , a data lake $\mathcal{D}$, and a set of tables $\mathcal{T} \in \mathcal{D}$ recommended by the data discovery system, true positives and false positives are tables $t \in \mathcal{T}$ related and not related to q , respectively. Further, false negatives are tables $t \in \mathcal{D}$ related to q but are not retrieved by the system, i.e. $t \notin \mathcal{T}$.

Precision is the ratio of relevant tables retrieved out of all retrieved tables:

$$precision = \frac{TP}{TP + FP} \quad (1)$$

Systems with high precision retrieve mostly relevant tables. **Recall** is the ratio of relevant tables retrieved out of all relevant tables in the data lake:

$$recall = \frac{TP}{TP + FN} \quad (2)$$

Systems with high recall retrieve most of the relevant tables in the data lake. It is also common to calculate precision@ k and recall@ k at different values of k , where k is the number of related tables desired.

2.3 Table Representation Learning

Table representation learning is a rapidly evolving subfield of machine learning focused on automatically learning meaningful and efficient representations of structured tabular data. The increasing availability of large-scale tabular datasets through open data portals and data science platforms has led to significant research interest in this area. Unlike traditional hand-crafted approaches to table summarization [53, 119], which require extensive feature engineering for each specific task, learned representations are task-agnostic and can be applied to various downstream applications. Moreover, these learned representations have demonstrated superior accuracy compared to simpler statistical methods, capturing complex relationships and dependencies within tabular data [22].

Several approaches have been proposed for table representation learning including transformer-based approaches (e.g., [8]), graph neural network-based approaches (e.g., [68]), and deep neural network-based approaches (e.g., [80, 10]).

One particular work of interest is that of Mueller and Smola [80]. They proposed a column representation learning approach using deep neural networks trained on a large repository of open data tables. Given two columns ($\mathcal{C}_i$ and $\mathcal{C}_j$), the authors propose to train two types of deep neural networks (h_0 and g_ψ) simultaneously to infer the likelihood of the two columns representing the same variables, i.e., whether the two columns have the same semantics. The loss function of the two networks is the binary cross-entropy loss:

$$Loss = y_{ij}C_{ij} + (1 - y_{ij}) \log \left(\frac{1}{1 - e^{-C_{ij}}} \right) \quad (3)$$

where $y_{ij} = 1$ when the two columns are similar and 0 otherwise, and C_{ij} is the learned probability that two columns are similar and is defined as:

$$\begin{aligned} C_{ij} &= \|h_0(\mathcal{C}_i) - h_0(\mathcal{C}_j)\|_2^2 + g_\psi(\mathcal{C}_i) + g_\psi(\mathcal{C}_j) \\ &= \left\| \frac{1}{|\mathcal{C}_i|} \sum_{x \in \mathcal{C}_i} h_\theta(x) - \frac{1}{|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j} h_\theta(x) \right\|_2^2 + \frac{1}{|\mathcal{C}_i|} \sum_{x \in \mathcal{C}_i} g_\psi(x) + \frac{1}{|\mathcal{C}_j|} \sum_{x \in \mathcal{C}_j} g_\psi(x) \end{aligned}$$

The two networks h_0 and g_ψ are applied at the value level then aggregated per column to have a fixed-size embedding regardless of the column size. The output of h_0 is the embedding vector of a column, while the output of g_ψ is a scalar that adjusts the matching probability when the two columns have common distributions (e.g., booleans).

The authors train separate embedding models for each data type. They consider three different data types: numeric, language, and string. The featurization of column values also depends on the data type. The embedding models are evaluated on the tasks of schema matching [41] and column retrieval.

In this thesis project, the technique of Mueller and Smola is adapted to the context of open data science platforms. One shortcoming addressed in the proposed research is the granularity of the embedding models, where having only three embedding models does not yield accurate column matches.

2.4 Data Discovery with Knowledge Graphs

While the area of data discovery using knowledge graphs remains largely unexplored, some research works explored this area. An example is KGLac [45], a data discovery and enrichment platform for data science. KGLac performs data discovery in large data lakes by analyzing datasets at the column level using data profiling techniques [2] to collect column features. KGLac uses these features to determine different types of similarities between columns including inclusion dependency, primary-key-foreign-key similarity, and column name similarity. Utilizing these similarities, KGLac constructs a knowledge graph that represents the semantics of the data lake. The ontology of GLac graph is shown in Figure 2.1. Nodes in this graph include datasets, tables, columns, and insights, while edges include hierarchical relationships (e.g. between column node and table node), and the different similarity relationships, annotated with their similarity score using RDF-Star. KGLac is the first work to propose data discovery using SPARQL queries on a knowledge graph of dataset semantics.

In this thesis project, the KGLac knowledge graph and ontology is utilized for open data science

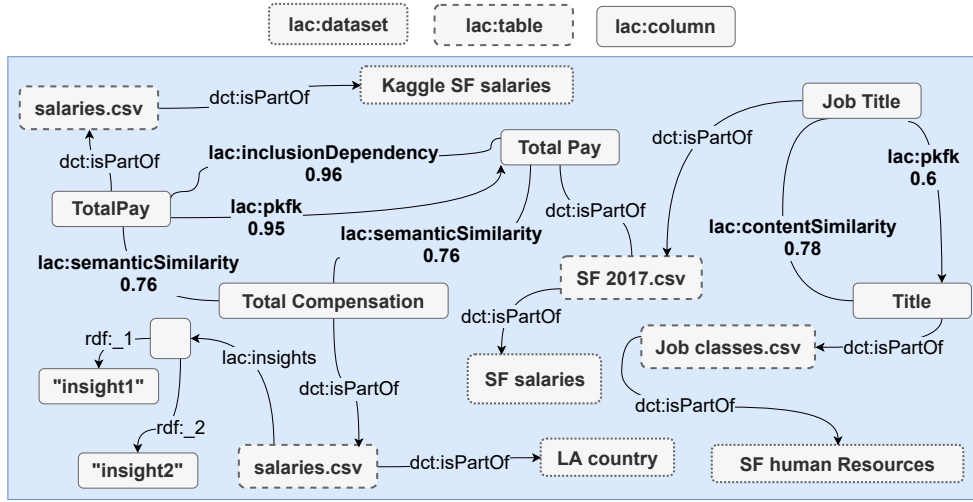


Figure 2.1: Ontology of GLac, a dataset knowledge graph proposed by [43].

platforms while addressing the following shortcomings: first, KGLac considers only the semantics of datasets, while the proposed research considers all data science artifacts including datasets, pipeline scripts, and data science libraries. Second, unlike KGLac, which uses hand-crafted features and basic statistics for column similarity detection, we utilize fine-grained deep dataset embedding models, which yield significantly more accurate similarities.

2.5 Semantic Pipeline Abstraction

One of the primary challenges in the exploration and automation of data science pipelines is to understand their semantics and represent them in a machine-readable format. Many techniques have been developed for semantic code understanding, which enabled several applications such as code search, code automation, refactoring, bug detection, and code optimization [3].

Semantic code understanding involves two main tasks: code analysis and semantic representation. **Code analysis** refers to the set of techniques aimed at analyzing source code to collect more information about its logical and semantic structures. **Semantic representation** involves storing the captured information from code analysis in a suitable data structure that allows for retrieval and reuse of information.

Code analysis of data science pipelines is done using two approaches: dynamic and static code analysis. Dynamic analysis involves running the program with various inputs and observing its

behavior, including CPU usage, memory traces, and variable types. This provides a detailed understanding of the program’s behavior at each step but is limited by its input-specific nature and the time required to execute the pipeline, specially for data science pipelines, which are data-intensive. Static analysis, on the other hand, examines the source code without executing it, making it efficient in terms of time and memory consumption. While it lacks the detailed information obtained through dynamic analysis, it can be enhanced with techniques tailored to data science libraries, making it suitable for analyzing large numbers of publicly available pipelines. Techniques used in static code analysis of data science pipelines include Abstract Syntax Tree (AST) analysis, control flow analysis, and data flow analysis. AST analysis explores the syntactic structure of the code, represented as a tree, to understand the overall organization of the pipeline. Control flow analysis tracks the execution path of the code, creating a directed graph to illustrate the order of operations, such as data preprocessing and model training. Data flow analysis focuses on how data is used and modified throughout the pipeline, also using a directed graph to represent the movement and transformation of variables. These methods collectively provide a comprehensive understanding of the pipeline’s structure, execution flow, and data manipulation.

Code representation methods describe the techniques to materialize the analyzed code in a machine-readable format. The outputs of static code analysis techniques are the abstract syntax tree (AST), control flow graph (CFG), and data flow graph. A natural representation that encapsulates these outputs is a single directed acyclic graph (DAG) whose nodes are statements, functions, classes, and packages, while edges are structural relationships, control flow, and data flow.

The most relevant approach to the proposed research is GraphGen4Code [1], a toolkit to generate knowledge graphs from source code, library documentation, and open forum questions. GraphGen4Code uses static code analysis and is tailored for Python without a specific support for data science pipelines. In this proposed research, GraphGen4Code is adapted to open data science platforms to address three shortcomings. First, the GraphGen4Code graph does not include information about datasets. Second, GraphGen4Code abstracts local script syntax (e.g., statement line number), resulting in a large amount of redundant information. Third, GraphGen4Code, misses important semantics such as data reads, i.e., the linking between code and data .

2.6 The Knowledge Graph for Linked Data Science (KGLIDS)

The work presented in this thesis is situated within the context of a larger, collaborative research project called KGLiDS (Knowledge Graph for Linked Data Science). The overarching goal of the KGLiDS platform is to address the challenge of disconnected artifacts in data science by creating a single, unified knowledge graph that captures the rich semantics of datasets, code pipelines, and their intricate interconnections. The platform is designed as a fully-fledged system with three main components: a KG Governor responsible for profiling data and abstracting code to build the graph, a Storage layer for the KG and embeddings, and Interfaces for user interaction and automation [47].

While the KGLiDS platform is a broad, collaborative effort, this thesis focuses specifically on the author’s primary and individual contributions to the project, namely the design and implementation of the dataset subgraph (the Data Global Schema) and its application to data discovery, as will be detailed in Chapter 3. However, to provide the necessary context for the subsequent chapters on AutoML (Chapter 4) and Automated Exploratory Data Analysis (Chapter 5), which leverage the full, unified KGLiDS graph, this section briefly describes the overall graph structure.

The complete KGLiDS graph, as illustrated in Figure 2.2, is composed of three interconnected subgraphs: the Dataset Graph, the Pipeline Graphs, and the Library Graph. The first major component is the Dataset Graph, also referred to as the Data Global Schema, which serves as the semantic representation of the data lake itself. In this subgraph, nodes represent data sources (e.g., Kaggle), datasets, tables, and columns. The edges capture both the hierarchical structure (e.g., a column being part of a table) and the semantic similarity between columns. As this subgraph represents the core data-centric contribution of this thesis, its detailed construction and application for data discovery will be the exclusive focus of Chapter 3.

To capture the semantics of data science workflows, KGLiDS constructs individual Pipeline Graphs. This process uses a combination of lightweight static analysis of Python scripts, documentation analysis, and dataset usage analysis. In these graphs, nodes represent individual code statements, while directed edges represent the control flow and data flow between them. This rich representation of historical workflows is foundational for the meta-learning component for AutoML, as will be presented in Chapter 4. However, while this graph contains detailed semantics at

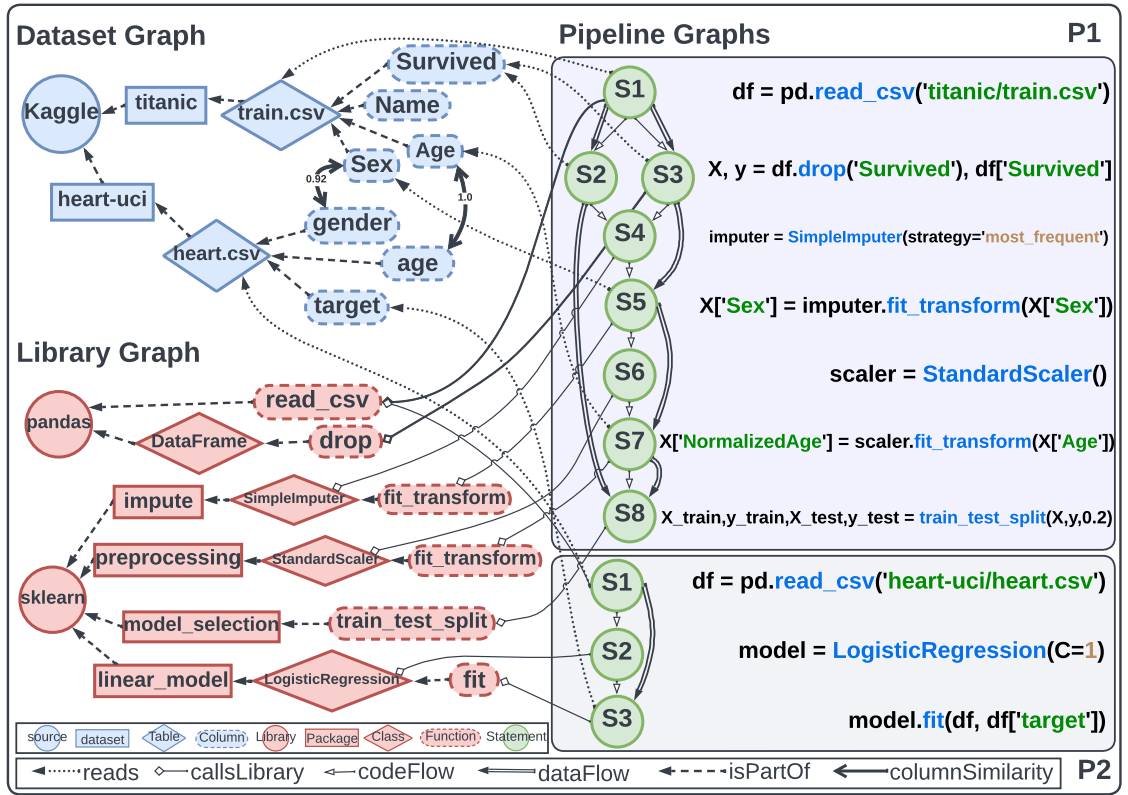


Figure 2.2: Two sample pipeline scripts and illustration of the corresponding KGLiDS graph composed of the Dataset Graph (Data Global Schema explained in Chapter 3), Pipeline Graphs, and Library Graph.

the statement level, it lacks an understanding of higher-level analytical tasks such as which statements perform data discovery, data cleaning and transformation, data exploration, or data modeling. In Chapter 5, this structure will serve as a base KG that is further enriched with EDA-specific semantics to enable the RAG framework.

Finally, the Library Graph provides deeper semantic context to the pipeline graphs. It is constructed by systematically parsing the official documentation of common data science libraries like Pandas and Scikit-learn. In this subgraph, nodes represent the components of these libraries, such as modules, classes, and functions, while its edges model the hierarchical relationships between them (e.g., a function belonging to a class).

By interlinking these three subgraphs, the complete KGLiDS graph creates a holistic view of the data science ecosystem. For example, a statement node in a pipeline graph can be linked to

both the specific function it calls in the library graph and the particular data column it reads from the dataset graph. This unified structure is what enables the advanced, cross-artifact automation paradigms explored in Chapters 4 and 5.

2.7 Retrieval-Augmented Generation

Large language models (LLMs) have demonstrated remarkable capabilities in natural language understanding and generation [109, 83, 102]. However, their reliance on their training set introduces limitations such as inability to provide up-to-date information, factually incorrect answers (i.e. hallucinations), and inability to adapt to specific domains or tasks. Retrieval-augmented generation (RAG) emerged as a promising solution to address these challenges by integrating LLMs with external knowledge sources [30].

RAG-based systems operate by retrieving relevant information from external repositories, such as databases, knowledge graphs, or text corpora, and incorporating this information into the LLM’s generation process. This augmentation enables the LLM to access up-to-date information, ground its responses in factual data, and tailor its output to specific domains or tasks.

A RAG-based system is typically composed of: 1) **Knowledge Source**, which comprises the external data that the LLM can access. The choice of knowledge source depends on the specific application and can range from general-purpose knowledge bases like Wikipedia to domain-specific datasets. 2) **Retrieval Model**, which is responsible for identifying and retrieving relevant information from the knowledge source based on the user’s query. Techniques such as dense retrieval, which leverages embeddings to represent textual information, are commonly employed. 3) **LLM**, which serves as the generation engine, utilizing both the user’s query and the retrieved information to produce the final output. [36]

RAG-based systems offer several advantages over LLM-based systems including improved accuracy by accessing external and up-to-date knowledge sources, reduced errors or hallucinations since LLMs can cross-check the facts before generation, and explainability, because RAG-based systems can reveal the knowledge sources used to generate the answers [30]. We develop a RAG-based approach in our proposed research that utilizes knowledge graphs and dataset embeddings

in the retrieval component and combines it with an LLM agent for code generation as detailed in Chapter 5.

2.8 Automated Machine Learning (AutoML)

AutoML is the process of automatically building optimized machine learning pipelines for new datasets. A machine learning pipeline consists of a set of data transformations $\mathcal{T}$ and a learning algorithm $\mathcal{L}$. Given an unseen dataset $\mathcal{D}$, the objective of an AutoML system is to search for the combination of transformations $t \in \mathcal{T}$, learning algorithm $l \in \mathcal{L}$, and its hyperparameters $\theta_{\mathcal{L}}$, such that a scoring metric $\mathcal{S}$ is maximized on $\mathcal{D}$ within a time budget T . AutoML systems have two primary components: the first is the meta learner, which selects the optimal starting point for the search space by designing a set of meta-features, collecting them for the unseen dataset $\mathcal{D}$, and matching them against previously collected meta-features. The second component is the hyperparameter optimizer, which implements the search algorithm that navigates the search space by repeatedly selecting a combination $\{t, l, \theta_{\mathcal{L}}\}$, evaluating the scoring metric $\mathcal{S}$, and deciding the next direction of search space navigation until the time budget T runs out. The optimal ML pipeline is the one that had the highest score $\mathcal{S}$ during hyperparameter optimization. Therefore, research in AutoML has focused on improving meta-features, meta-learning techniques, hyperparameter optimization algorithms, or multiple of these factors [46, 69].

Because the search space for the optimal combination $\{t, l, \theta_{\mathcal{L}}\}$ is prohibitively large, exhaustive search, i.e., evaluating the scoring metric $\mathcal{S}$ at every search point, is infeasible. AutoML systems, therefore, aim to select the best starting point for the search as well as the best search strategy. Some meta-learning approaches include mining existing data science code and their associated datasets to learn from human expertise. The AL system [17] mined existing Kaggle notebooks using dynamic analysis, that is, actually running the scripts and collecting run-time traces during execution such as number of rows and columns, value distributions of dataset features, etc. Although the AL showed promise, it does not scale because it is onerous to execute a large number of pipeline scripts on datasets, which is time- and resource-intensive or sometimes infeasible due to the evolution of the ML library over time. Therefore, it is not surprising that AL performed a dynamic analysis on just

nine datasets. Other meta-learning approaches such as [32, 95] devise a set of hand-crafted meta-features, compute them for a set of sample datasets from multiple domains, and compute the best search starting point associated with every sample dataset. When an unseen dataset is provided, the meta-features are computed, matched against those of the sample datasets, and the starting point associated with the closest dataset (via e.g. cosine similarity) is selected. However, hand-crafted features of datasets have been shown to be deficient in summarizing dataset properties [53].

In this thesis, a data-driven meta-learning system is developed using graph neural networks, knowledge graphs, and deep dataset embeddings. The proposed approach is combined with existing hyperparameter optimizers and is demonstrated to generate better starting points for the hyperparameter search, measured by the ML accuracy under the same time budget.

2.9 Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is a critical first step in any data science pipelines. This includes uncovering underlying patterns, identifying potential anomalies, and formulating hypotheses for further investigation. EDA is a data-intensive and time-consuming task, often forming the foundation for subsequent analysis. Surveys of data scientists by Kaggle [61] and Anaconda [4] confirm this, indicating that data analysis and visualization consume a significant portion (29%) of a data scientist’s time, with a majority (57%) considering it their primary responsibility.

EDA operations can be categorized into three types: univariate, bivariate, and multivariate analysis. Univariate analysis focuses on exploring the distributions of individual columns. Bivariate and multivariate analyses investigate the relationship between two or more column, respectively, exploring joint distributions, correlations, and dependencies. Data scientists utilize a large set of visualization charts including bar charts, line charts, scatter plots, pie charts, etc.

The primary challenge in automating EDA is the vast number of choices involved including dataset columns, analysis types, and chart types. Generating all possible visualizations, while technically feasible, would result in an overwhelming output, limiting the gained insights. An effective automated EDA system must intelligently navigate this search space choices, prioritizing combinations that are most likely to yield meaningful discoveries. Furthermore, evaluating the effectiveness

of an auto EDA system is inherently complex due to the subjective nature of EDA itself. What constitutes a "good" or "insightful" visualization can vary between individuals and analytical goals. To address this, research works in this area [9, 72] often rely on user studies, where data scientists examine and rate the outputs generated by different auto EDA systems. These studies employ metrics like comprehensiveness, informativity, and clarity to assess the quality and usefulness of the generated visualizations.

In this thesis, an auto EDA system is developed using a retrieval-augmented generation (RAG) approach, combining knowledge graphs, deep dataset embeddings, and large language models to generate EDA notebooks for unseen datasets.

Chapter 3

Dataset Embeddings for Effective Data Discovery

The task of data discovery is essential in data science, with numerous applications ranging from data integration [15] to automated machine learning [46]. With the growing success of data science, open data portals now offer access to tens of thousands of datasets[62, 85]. However, manually discovering tables related to a given query table in these vast data lakes is a time-consuming and effort-intensive process. This challenge has consequently motivated significant research focused on the automation of data discovery.

Data discovery systems are architecturally composed of two fundamental parts. The first is the **data discovery algorithm**, which defines the logic for comparing tables and columns and is the primary factor affecting the system’s **accuracy**. The second is the **data indexing structure**, which organizes the data and intermediate values for efficient retrieval and directly impacts the system’s **scalability**. Several approaches have been proposed to address this task. For instance, D3L [13] uses a collection of hand-crafted features to construct column profiles and performs discovery by comparing these profiles with Locality Sensitive Hashing (LSH) [55]. SANTOS [65] leverages open ontologies like YAGO [89] and DBPedia [7] to find relationships between columns, supplementing them with a synthesized ontology customized for the data lake. More recently, Starmie [29] employs embeddings of raw values, obtained by training a language model on the data lake, and calculates

embedding similarities using the Hierarchical Navigable Small World (HNSW) [75] algorithm.

Despite these advances, existing data discovery systems exhibit two primary shortcomings. First, their accuracy is often limited by a reliance on hand-crafted features (D3L), customized ontologies (SANTOS), or language models (Starmie) that are inherently designed for textual data and are less effective for numerical data. Second, these systems often lack scalability. This can be due to the absence of an optimized indexing structure (D3L, SANTOS), the need to analyze data at the granular level of individual values (SANTOS), or the requirement for lengthy training periods to achieve satisfactory accuracy (Starmie).

Separately, table representation learning (TRL) has emerged as an active research area focused on developing machine learning methods to embed raw column values into fixed-size, dense vector representations that capture column semantics. Approaches include training deep learning embedding networks [80], or using pre-trained [22] and purpose-trained [52] language models. While these methods perform well in representing individual columns and could serve as column profiles, they require significant adaptation to achieve competitive accuracy in the context of data discovery.

This chapter introduces the KGLiDS Data Discovery system, which directly addresses the aforementioned issues of previous research. KGLiDS Data Discovery is composed of two components: data profiling and knowledge graph (KG) construction. The data profiling component employs a simple yet effective strategy for data discovery: it profiles data at the granularity of individual columns, categorizes them into seven fine-grained data types, and strictly performs discovery by comparing columns only of the same type (e.g., integer columns are only compared with other integer columns, never with date columns). The data profiler adapts an existing TRL technique [80] by training a Column Learned-Representation (CoLR) model per fine-grained type to embed raw column values. Additionally, column names are embedded via word embeddings. This utilization of column embedding techniques enables accurate data discovery. The KG construction component builds a KG of columns, tables, and their semantic relationships. This formulation allows data discovery to be executed as KG queries, which scale effectively to tens of thousands of tables due to the efficient indexing in modern graph stores.

Through experimental evaluation on four data discovery benchmarks, we demonstrate that the KGLiDS data discovery system achieves at least 1.5x higher precision and recall while being 1.8x

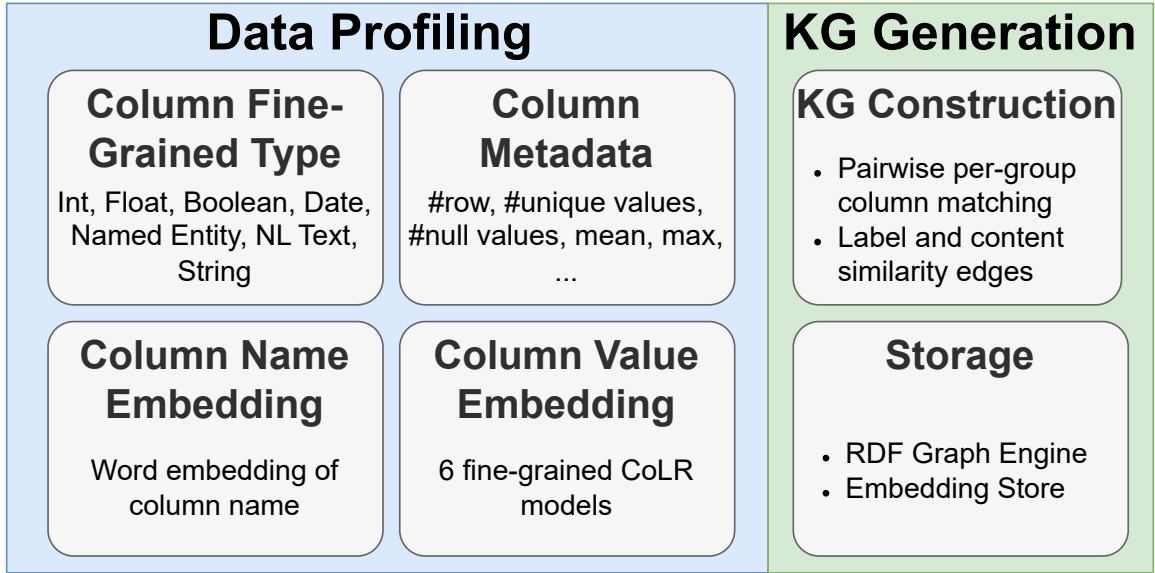


Figure 3.1: Architecture of the KGLiDS data discovery system..

faster in preprocessing time and 3.3x faster in query time than the state-of-the-art systems.

The main contributions of this chapter are as follows:

- A novel data profiler that utilizes fine-grained column types and specialized embedding models to improve the accuracy of semantic matching.
- A scalable knowledge graph construction method that models data elements and their relationships, enabling efficient data discovery through KG queries.
- A comprehensive evaluation on four standard data discovery benchmarks that demonstrates the superior accuracy and scalability of the proposed approach.

3.1 Embedding-Based Data Profiling For Data Discovery

The KGLiDS data profiling component analyzes datasets at the column level to generate a detailed column profile for each column in a data lake. This profile serves as a structured summary of a column’s key characteristics, which are foundational for building the final knowledge graph. The first and most critical step in this profiling process is to accurately infer a column’s fine-grained data type. The entire profiling component is developed in PySpark to ensure distributed computation and scalability across large data lakes.

Algorithm 1: Data Profiling

Input: Datasets $\mathcal{D}$, CoLR Models $\mathcal{H}_{\theta, \mathcal{T}}$, Word Embeddings $\mathcal{W}$, Pre-trained NER Model f_{σ}

1 Main Node:

2 $\text{columns}_{rdd} = \{c; c \in t; t \in \mathcal{D}\}$ ▷ Columns in all tables

3 $\text{columns}_{rdd}.map(\text{profile_column})$

4 Worker (Parallel) $\text{profile_column}(col)$:

5 $M = \text{col.metadata}$ ▷ Table and dataset membership

6 $\text{fgt} = \text{infer_fine_grained_type}(col, \mathcal{W}, f_{\sigma})$

7 $S = \text{collect_stats}(col, \text{fgt})$ ▷ Statistics e.g. #NaNs

8 $E_l = \mathcal{W}(\text{col.name})$ ▷ Label embedding of column name

9 $E_c = [0]^{300 \times 1}$ ▷ 300-Dimensional content embedding

10 **for** $\text{val} \in \text{col.sample}(\max(0.1|\text{col}|, 1000))$ **do**

11 $E_c \leftarrow E_c + \frac{1}{|\text{col}|} h_{\theta, \text{fgt}}(\text{val})$ ▷ Avg. CoLR over values

12 $\mathcal{CP} = \{M, \text{fgt}, S, E_l, E_c\}$ ▷ Column profile

13 $\text{json.dump}(\mathcal{CP})$ ▷ Store

Algorithm 1 details the distributed process for generating a profile for each column. It takes as input the datasets to be processed, pre-trained word embeddings, a pre-trained Named Entity Recognition (NER) model, and a set of pre-trained Column Learned Representation (CoLR) models $\mathcal{H}_{\theta, \mathcal{T}}$. The process operates in a master-worker paradigm using PySpark, where the main node creates a distributed collection of all columns from and assigns them to worker nodes for parallel execution.

Each worker executes the `profile_column` function on an individual column. This function begins by extracting the column’s basic metadata and summary statistics. Next, it performs fine-grained type inference to classify the column. Following classification, it generates two embeddings: a label embedding from the column’s name, and a 300-dimensional content embedding. The content embedding is derived from a sample of the column’s raw values, using the corresponding CoLR model for the inferred type. The sample size is 10% of the column size but with a max size of 1000 values. The CoLR embedding is obtained by applying the CoLR model on each value in the sample to get a value embedding then averaging all value embeddings. This ensures the embedding is order-invariant.

The function concludes by assembling the extracted metadata, statistics, inferred type, and both embeddings into a single column profile. This final profile is then stored as a JSON document.

3.1.1 Fine-Grained Data Type Inference

KGLiDS employs a simple yet effective strategy for data discovery. It first categorizes all columns into fine-grained semantic types. It then constrains the discovery process by comparing only columns that belong to the same type. This approach drastically reduces false positives in similarity predictions by preventing illogical comparisons, such as matching an integer column with a date column.

The number of defined types presents a crucial trade-off. Too few types result in heterogeneous groups and lower accuracy, while too many types can create overly specific groups that may not contain enough columns for meaningful discovery or may not generalize across different data lakes. To address this, KGLiDS defines seven semantically meaningful data types:

- **Integer:** Columns containing whole numbers.
- **Float:** Columns containing decimal numbers.
- **Boolean:** Columns representing binary values. The system normalizes various formats, such as `True/False`, `1/0`, and `"Yes"/"No"`.
- **Date:** Columns containing temporal data. The system recognizes and handles multiple common datetime formats.
- **Named Entity:** Columns containing entities like persons, organizations, countries, and currencies. These are identified using a pre-trained Named Entity Recognition (NER) model [91] trained on the OntoNotes 5 dataset [110]. The NER model recognizes 18 different entity types including persons, organizations, locations, currencies, etc.
- **Natural Language Text:** Columns with long-form text, such as product reviews, where values have corresponding word embeddings.
- **String:** Columns with generic string values that do not fit other categories, such as postal codes or IDs.

An analysis of different real-world and synthetic data lakes (shown in Section 3.3) shows these categories cover the vast majority of columnar data found in practice. Furthermore, each type is

semantically distinct, implying different requirements for processing and analysis. For example, a Named Entity column suggests different analytical tasks (e.g., aggregation by location) than a generic String column (e.g., counting unique IDs). This partitioning is therefore a foundational step that aligns the data with relevant downstream data science operations.

Type inference is performed on a 10% random sample of a column’s values. Each value in the sample is classified, and the column is assigned the type that represents the majority of the sampled values.

3.1.2 Column Metadata and Statistics

In addition to the fine-grained type, KGLiDS enriches each column profile with a comprehensive set of metadata and summary statistics. This collection serves a dual purpose: it provides a concise, human-readable summary of each column, and it enables efficient pre-filtering of columns when querying the knowledge graph.

The collected metadata includes the column’s name, its parent table and dataset, and structural information such as file paths. The summary statistics are a mix of common and type-dependent metrics. Statistics common across all data types include the total number of rows, the count of null values, and the number of unique values. For numerical columns like Integer and Float, the system also computes type-specific statistics, including mean, minimum, maximum, standard deviation, and interquartile range (IQR).

This collection of metadata and statistics is then associated with the corresponding column representation within the knowledge graph, making these attributes accessible for queries.

3.1.3 Column and Table Embeddings

Unlike previous data discovery approaches that rely on techniques like minhashing [13] or knowledge base concept matching [65], KGLiDS utilizes column embeddings for discovering similar columns and datasets in a data lake. An embedding is a fixed-size, dense vector that summarizes a column’s semantics. In this vector space, columns with similar content or meaning have embeddings that are close to each other, while dissimilar columns have embeddings that are far apart. This embedding-based approach offers three main advantages for data discovery. First, it yields

higher accuracy in predicting column similarities compared to systems that use hand-crafted meta-features, which can fail when a column’s data distribution does not match the feature design [80, 53]. Second, it enables data discovery without exposing the raw data, which is invaluable in enterprise settings with strict data privacy requirements. Third, the compact, fixed-size nature of embeddings regardless of the column size greatly reduces storage and memory needs of the system compared to methods that analyze raw values.

To comprehensively capture column semantics, KGLIDS utilizes two distinct types of embeddings: one derived from the column’s name and another from its values. These embeddings provide two complementary semantic signals. The name embedding captures the intended meaning from human-provided labels, while the value embedding captures the empirical semantics from the data itself. By storing both in the column profile, KGLiDS allows discovery queries to flexibly leverage either or both signals for a more powerful and nuanced search capability.

3.1.3.1 Column Name Embedding

In many data lakes, column names are strong indicators of the column’s content. To capture this signal, KGLiDS embeds column names using pre-trained word embeddings. This process converts each column name into a dense vector, such that columns with semantically similar names (e.g., "location" and "place") will have close embeddings, while columns with dissimilar names will have distant ones. Several word embedding techniques exist such as GloVe [90], FastText [14], and embeddings from language models like GPT [87]. In our work we utilize GloVe embeddings due to its relative simplicity and scalability. However, KGLiDS supports any word embedding technique to generate column name embeddings.

3.1.3.2 Column Value Embedding

To capture the semantics of a column’s underlying data distribution, KGLiDS generates an embedding from its raw values. While several methods for this task exist [8, 68, 10], our Column Learned Representation (CoLR) models adapt the deep learning approach proposed by Mueller and Smola [80], which embeds columns using pretrained deep neural networks. These networks are trained on a binary classification task to predict if two columns are semantically similar, using

a binary cross-entropy loss function [39]. To generate an embedding for a column, its values are first preprocessed based on its type. Numerical values are converted to 32-bit representation, natural language values are converted to the average word embedding with FastText [14], and generic string values are preprocessed using a character-level embedding neural network. After preprocessing, the deep neural network is applied to each value in a column and the embedding is obtained from the last hidden layer. The column embedding is the average of its value embeddings.

The baseline approach of Mueller and Smola trains three separate deep neural networks for three generic column types (numerical, textual, and string), creating a distinct embedding space for each. However, when applied directly to our data discovery framework, this method yields low accuracy due to two significant issues. The first is that the three generic types are insufficient to capture the differences between a diversity of data types in data lakes. The second is the misrepresentation of specific data types. For example, boolean columns containing 1s and 0s are incorrectly treated as numerical, while those with textual "True"/"False" values are treated as textual columns. This leads to similar columns being embedded in separate embedding spaces. Another example is date columns, which are labeled as generic strings and preprocessed using a character-level neural network. Consequently, similarity between date columns is determined primarily by the utilized date time format rather than their actual temporal proximity.

KGLiDS overcomes these limitations with a refined solution. First, it replaces the three generic types with the seven fine-grained types detailed previously. Second, it ensures a more accurate data representation before embedding. For instance, it resolves the semantic mismatch for dates by converting all date formats to a canonical epoch time integer and converting all boolean columns to 1/0 format. Third, instead of training three generic models, KGLiDS uses six new, fine-grained Column Learned Representation (CoLR) models, one for each non-boolean data type (int, float, date, named entity, natural language, and string). For boolean columns, a complex deep learning model is unnecessary and computationally wasteful. The column’s complete semantics can be captured losslessly by a single statistic: the ratio of true values, making it the most efficient and equally effective representation. The six CoLR models were trained on a corpus of 5,500 datasets from Kaggle and OpenML, following the original architecture proposed by Mueller and Smola. To generate a column embedding, a 10% random sample is taken from the column, and a CoLR model h_θ is applied to

each value and averaged for the entire sample (lines 9-11 in Algorithm 1).

The final embedding for a table is generated by aggregating the embeddings of its columns for each fine-grained type and concatenating the results, as shown in the following formula:

$$h_{\theta}(\mathcal{D}) = \left\| \frac{1}{|c_{fgt}|} \sum_{c_{fgt} \in \mathcal{D}} h_{\theta,fgt}(c_{fgt}) \right\| \quad (4)$$

where $|c_{fgt}|$ is the number of columns in $\mathcal{D}$ with the fine-grained type fgt . Similarly, an embedding of a dataset is an aggregation of its tables' embeddings. The Data Profiling stores the generated embeddings in the embedding store. For simplicity, we do not show the generation of the table's embeddings. Algorithm 1 generates a column's profile containing the predicted fine-grained type fgt , the generated embeddings E , the column statistics S , and metadata $\mathcal{M}$ and dumps it as a JSON document (lines 11 and 12). Algorithm 1 is designed to work with independent tasks at two levels. First, the dataset is decomposed into independent tables. Second, each table is decomposed into a set of columns, where most computations are done. This design profiles datasets at scale.

3.2 Knowledge Graph Construction

The column profiles, containing fine-grained type, statistical, and embedding information, serve as the foundational elements for the knowledge graph. This section details the construction process, which transforms this collection of isolated profiles into a single, connected semantic representation of the entire data lake, which we call the Data Global Schema. The KG Builder component materializes this graph and stores it alongside the generated embeddings for efficient querying.

3.2.1 Data Global Schema Structure

The Data Global Schema is a graph-based semantic representation of the datasets within a data lake and the relationships between them. The schema consists of four types of nodes: data sources (e.g., Kaggle, OpenML), datasets (collections of tables), tables, and columns.

These nodes are connected by two types of edges. Hierarchical edges represent structural containment, linking columns to their parent tables, tables to their parent datasets, and datasets to

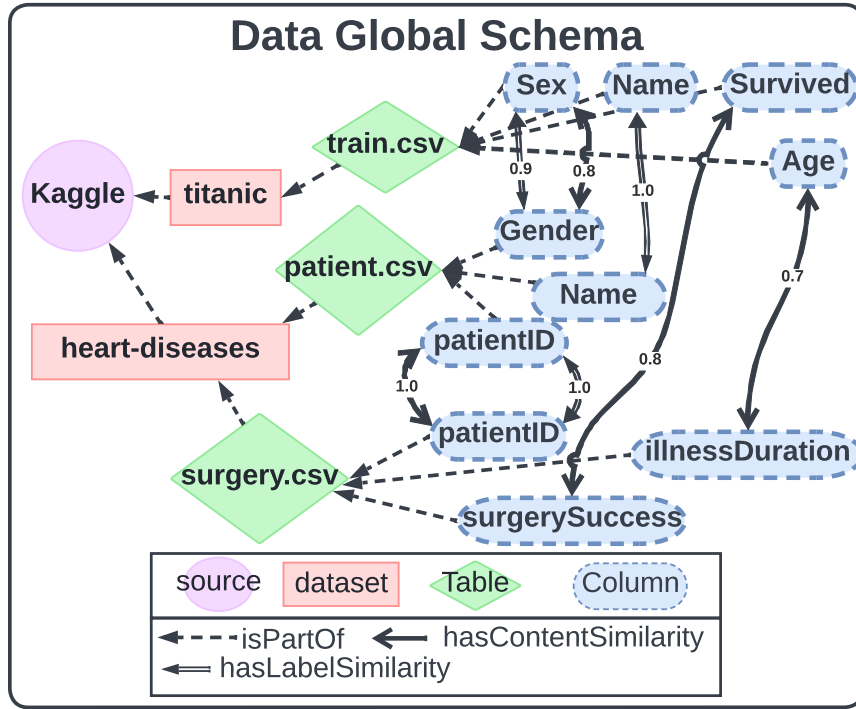


Figure 3.2: A sample data global schema from two Kaggle datasets showing the different node and edge types in our knowledge graph.

their respective data sources. Similarity edges represent semantic relationships and are the primary mechanism for data discovery. These edges primarily occur between columns that share the same fine-grained type but belong to different tables. We recognize two kinds of similarity:

- **Label Similarity:** Derived from column name embeddings, this relationship indicates that two columns have semantically similar names.
- **Content Similarity:** Derived from column value embeddings, this relationship indicates that two columns contain similar raw values or describe the same concept.

Similarity edges in the data global schema are annotated with a similarity score, a decimal value between 0 and 1 indicating the strength of the similarity relationship.

3.2.2 Data Global Schema Construction Process

The Data Global Schema is built by systematically comparing columns and materializing similarity edges. Algorithm 2 illustrates the pseudocode of our algorithm to construct the data global

Algorithm 2: Data Global Schema Builder

Input: Column Profiles $\mathcal{CP}$, Similarity Thresholds: α, β, θ

1 Main Node:

2 $\lfloor \mathcal{CP}_{rdd}.mapPartitions(column_metadata_worker)$

3 Worker (Parallel) $column_metadata_worker(cp)$:

4 $\lfloor g = create_metadata_subgraph(cp)$

5 $\lfloor json.dump(g)$ ▷ Save metadata subgraph

6 Main Node:

7 $\lfloor \triangleright$ column pairs with the same fine-grained type

8 $\lfloor \mathcal{P} = \{(cp_i, cp_j) \mid cp_i, cp_j \in \mathcal{CP}; i \neq j; \mathcal{T}_{cp_i} = \mathcal{T}_{cp_j}\}$

9 $\lfloor \mathcal{P}_{rdd}.map(column_similarity_worker)$

9 Worker (Parallel) $column_similarity_worker(cp_i, cp_j)$:

10 $\lfloor g = \phi$

11 **if** $word_embed_sim(cp_i.label, cp_j.label) \geq \alpha$:

12 $\lfloor \lfloor add_edge(g, cp_i, cp_j, "LabelSimilarity", \alpha)$

13 **if** $\mathcal{T}_{cp_i} == "boolean"$:

14 $\lfloor \lfloor \mathbf{if}$ $(1 - |cp_i.true_ratio - cp_j.true_ratio|) \geq \beta$:

15 $\lfloor \lfloor \lfloor add_edge(g, cp_i, cp_j, "ContentSimilarity", \beta)$

16 **else:**

17 $\lfloor \lfloor \mathbf{if}$ $colr_embed_sim(cp_i.embed, cp_j.embed) \geq \theta$:

18 $\lfloor \lfloor \lfloor add_edge(g, cp_i, cp_j, "ContentSimilarity", \theta)$

19 $\lfloor json.dump(g)$ ▷ Save column similarity subgraph

20 Main Node:

21 $\lfloor G = \phi$ ▷ Data global schema graph

22 **for** $subgraph\ g$ **do**

23 $\lfloor G \leftarrow G \cup g$

24 **return** G

schema. The algorithm receives a set of column profiles $\mathcal{CP}$ and a set of similarity thresholds. Each profile contains the predicted fine-grained type $\mathcal{T}$, the generated embeddings E , the column statistics S , and metadata $\mathcal{M}$, which are generated by Algorithm 1. First, a metadata subgraph is constructed, which contains the hierarchy structure of the datasets and statistics collected for each column. Next is a pairwise comparison of all columns belonging to the same fine-grained type and belonging to different tables. This is distributed in a MapReduce fashion. Each worker takes a pair of profiles and generates the similarity edges, i.e., predicates, between them. For most data types,

the similarity score is computed in a two-step process. First, the cosine distance is calculated between the columns' respective content or label embeddings. Second, this distance is normalized to a similarity score between 0 and 1 using the formula:

$$sim = 1 - \tanh(distance) \quad (5)$$

We chose $\tanh(x)$ as it is a non-decreasing function with range $[0, 1]$ for domain values $\in [0, \infty)$. Therefore, the similarity score is in the range $\in [0, 1)$ for distance values in the domain $\in [0, \infty)$.

For `boolean` columns, content similarity is calculated more directly using the difference in the columns' true value ratios as follows:

$$sim_{boolean} = 1 - |true_ratio_1 - true_ratio_2| \quad (6)$$

The thresholds, α , β , and θ are user-defined and control the similarity thresholds above which column similarity relationships are materialized in the data global schema. High similarity thresholds result in less but more accurate edges, i.e. high precision and low recall. Conversely, lower similarity thresholds might be used when high recall is desirable. This configurable threshold allows the system to be adapted for different tasks. A data scientist performing a broad, exploratory search might use a lower threshold to maximize recall, while a user performing automated data integration would likely prefer a higher threshold to ensure high-precision matches. Finally, the dataset graph is constructed. KGLiDS utilizes the predicted relationships between columns to identify unionable and joinable tables. Two tables are unionable or joinable if one or more of their columns have high label or content similarity relationships, respectively. The similarity score between two tables is based on both the number of similar columns and the similarity scores between them.

Algorithm 2 summarizes the construction process of the Data Global Schema. The algorithm takes as input the set of all column profiles generated by Algorithm 1 and a series of user-defined similarity thresholds (α, β, θ) that control the materialization of semantic relationships. The construction is executed in three main phases using a distributed framework.

The first phase involves creating the structural backbone of the graph (lines 1-5). The algorithm

begins on the main node by distributing the column profiles to parallel workers (line 1), each executing the `column_metadata_worker` function. This worker processes its set of profiles to construct a metadata subgraph containing the nodes for columns, tables, and datasets, along with their hierarchical `isPartOf` relationships and associated statistics (line 4). These individual subgraphs, representing the basic structure and properties of the data artifacts, are then saved (line 5).

The second phase is the creation of similarity edges (lines 6-19). Back on the main node, the algorithm generates all unique pairs of columns that share the same fine-grained data type but belong to different tables (line 7). These pairs are distributed to workers running in parallel the function `column_similarity_worker` (line 8). For each pair of columns, the worker first calculates the similarity between their name embeddings; if this score exceeds the threshold α , a `LabelSimilarity` edge is added to a new subgraph (lines 11-12). The worker then calculates content similarity, using a method dependent on the column type. For boolean columns, similarity is calculated from the difference in their true value ratios, and a `ContentSimilarity` edge is added if the score meets the threshold β (lines 13-15). For all other data types, similarity is computed from the cosine distance of their value embeddings. If this score surpasses the user-defined threshold θ , a `ContentSimilarity` edge is materialized (lines 17-18). The resulting subgraphs containing these new similarity edges are then saved (line 19).

In the final phase (lines 20-24), the algorithm aggregates all the independently generated subgraphs. On the main node, the final Data Global Schema graph is initialized as an empty graph (line 21). The algorithm then iterates through all the metadata and similarity subgraphs created by the workers and merges them into this single, unified graph (lines 22-23). The completed Data Global Schema is then returned (line 24), representing a fully connected semantic map of the entire data lake.

3.2.3 Storage and Querying

The KGLiDS architecture is designed to be independent of any single storage technology. The current implementation manages two distinct types of information—the semantic knowledge graph and the numerical embeddings—using a separate, specialized store for each. This separation is a

practical choice for the current implementation, although other storage solutions that combine graph and vector indexing could also be used.

For the knowledge graph, the system adopts the RDF model and uses GraphDB [100] as its storage engine. RDF-based technologies were chosen for several key reasons: (i) they formalize meta-data using controlled vocabularies, ensuring interoperability [106, 34]; (ii) their schema-agnostic nature supports reasoning and allows the platform to evolve easily [121, 16, 35]; (iii) RDF stores are mature and scalable due to powerful indexing techniques; and (iv) they feature the powerful SPARQL query language for efficient query processing [79, 98]. Specifically, KGLiDS leverages RDF-star [42], which allows edges to be annotated with metadata, enabling us to directly attach similarity scores to the similarity edges between column nodes.

For the vector data, KGLiDS uses a dedicated embedding store to index the generated embeddings. This store enables efficient similarity search based on approximate nearest neighbor operations. This design empowers data scientists with a flexible query interface, allowing them to execute structured graph traversals via SPARQL and also to initiate powerful similarity-based discovery searches that begin with an embedding and seamlessly link back to the rich semantic context of the knowledge graph.

3.3 Experimental Evaluation

We evaluate the performance of KGLiDS’s components against the state-of-the-art (SOTA) systems in data discovery, semantic abstraction of code, data cleaning, data transformation, and AutoML. Unlike KGLiDS, none of these systems provides simultaneous support for all these functionalities.

3.3.1 Experimental Setup.

We used GraphDB¹ version *10.4.1* as an RDF engine to store the LiDS graph. We used two different hardware settings for our data discovery experiments. For comparison with the baselines, we used Ubuntu 22.04, a 64-core CPU at 2.45 GHz, 1TB of RAM, and an Nvidia A100 GPU. In the

¹<https://www.ontotext.com/products/graphdb/>

Table 3.1: Data Discovery Benchmarks. The breakdown of column data types is obtained using our data profiler.

Statistic	D ³ L Small	TUS Small	SANTOS Small	SANTOS Large
Size (GB)	1.3	1.2	0.4	11.5
No. tables	654	1,530	550	11,090
No. query tables	50	150	50	80
Avg. No. unionable tables	110	163	14	-
Avg. No. rows per table	12,207	4,457	6,921	7,718
Total columns	8,767	14,810	6,336	121,796
int cols.	1,885	1,222	1,267	25,618
float cols.	513	288	271	5,702
boolean cols.	8	111	110	1,173
date cols.	661	884	331	6,891
named_entity cols.	516	1,766	1,053	18,897
natural_language cols.	4,241	9,345	2,908	53,502
string cols.	957	1,194	396	10,013

second setting, we used a SPARK cluster of 16 machines, each with 64 GB RAM and 16 cores to construct our dataset graphs. Unlike KGLiDS, none of the other systems are distributed, i.e., do not scale up with multiple machines. Therefore, we used this configuration to demonstrate KGLiDS’s strong scalability (Figure 3.5).

3.3.2 Data Discovery Systems

We evaluate KGLiDS against two SOTA data discovery systems: Starmie [29] and SANTOS [65]. Starmie discovers unionable tables via column embeddings from pre-trained language models. SANTOS uses open and synthesized knowledge bases to match column relationships within tables. We evaluate the effectiveness of these systems using three benchmarks: *D³L Small*² [13], *TUS Small*³ [84], and *SANTOS Small*⁴ [65]. *D³L Small* is a collection of 654 real-world tables, where each table in the data lake is manually annotated with all other related tables. *TUS Small* and *SANTOS Small* are collections of 1,539 and 550 synthetic tables, respectively, generated using random horizontal and vertical partitioning from real-world tables. The statistics of these

²<https://github.com/alex-bogatu/DataSpiders>

³<https://github.com/RJMillerLab/table-union-search-benchmark>

⁴<https://github.com/northeastern-datalab/santos>

Table 3.2: Preprocessing and average query time for all benchmarks. KGLiDS outperforms Starmie and SANTOS in both.

Benchmark	Time	SANTOS	Starmie	KGLiDS
D³L	Preprocessing	2.96 hr	0.88 hr	0.37 hr
Small	Avg. Query	18.8 s	0.71 s	0.02 s
TUS	Preprocessing	4.66 hr	0.70 hr	0.69 hr
Small	Avg. Query	13.35 s	0.04 s	0.02 s
SANTOS	Preprocessing	1.89 hr	0.33 hr	0.26 hr
Small	Avg. Query	17.12 s	0.11 s	0.01 s
SANTOS	Preprocessing	30.40 hr	7.67 hr	4.15 hr
Large	Avg. Query	12.28 s	0.79 s	0.24 s

benchmarks are shown in Table 3.1. Note that the D^3L Small is the most challenging because it was manually annotated rather than synthetically generated, has the highest number of rows per table on average, and has a high number of unionable tables of 110 on average. Furthermore, because SANTOS requires additional manual annotation of intent columns for all tables, we followed the authors’ recommendation of using the first columns as intent columns for D^3L Small.

3.3.3 Accuracy of Data Discovery

These benchmarks perform a data discovery query for table unionability for a query table T . The accuracy of the system in performing this query is measured in terms of Precision@ k and Recall@ k for different values of k , such that k is the number of desired unionable tables to T [13, 84, 65]. The precision and recall are calculated for each k as the averages over N query tables. In addition, we evaluated KGLiDS’s scalability using SANTOS Large [65], which contains 11,090 tables without ground truth. In our evaluation, we used the same values of (N, k) as in previous evaluations. More specifically, for D^3L Small we used (50, 185), for TUS Small we used (150, 60), and for SANTOS Small we used (50, 10).

Figure 3.3 illustrates the average Precision@ k and Recall@ k across various systems and benchmarks. KGLiDS significantly outperforms Starmie and SANTOS in precision and recall on D^3L Small and TUS Small. Unlike KGLiDS, Starmie uses column embeddings from pre-trained language models, which work well with textual columns but do not achieve the same accuracy for

numerical columns. We analyzed Starmie’s individual column matches for D^3L Small, which had 52.2 precision for numerical columns compared to 63.4 for textual columns. Our CoLR models are trained to predict column similarity even with different value distributions, which enhances similarity prediction for real-world benchmarks like D^3L Small. KGLiDS still achieves high accuracy for synthetically generated benchmarks, where unionable tables have the same distribution. The SANTOS Small benchmark used tables of relatively smaller size and a low value of k . Hence, the three systems archive comparable precision and recall.

3.3.4 System Scalability Analysis

These experiments analyze the preprocessing overhead and execution time of the query tables. The preprocessing overhead is the offline time required to profile tables in a data lake. The query execution time is reported as the average query time of all query tables (see Table 3.1 for the number of query tables in each benchmark). Table 3.2 summarizes our results. KGLiDS consistently outperforms SANTOS and Starmie in preprocessing and average query times for small and large benchmarks. We focus on the SANTOS Large benchmark as it has the largest number of tables. KGLiDS is 7.3x faster than SANTOS in preprocessing time. This is because SANTOS analyzes data lake tables on the granularity of column values, matching each value against two knowledge bases (KB): an open KB (YAGO [89]) and a synthesized KB generated during preprocessing. SANTOS then iterates over *all value pairs* of matching columns per table to determine their semantic relationships.

KGLiDS analyzes data lakes on the granularity of columns and applies our CoLR models to column samples. Compared to Starmie, KGLiDS achieves a 1.8x speed up in preprocessing time. The preprocessing phase of Starmie includes training a language model (LM) to generate the column embeddings for a data lake. Starmie uses data augmentation on data lake columns to generate enough samples to train the LM. Further, Starmie needs to train the LM for a sufficient number of epochs to achieve a decent score (we use ten epochs as recommended by the authors of Starmie). Unlike Starmie, KGLiDS does not require training our embedding models per data lake as our models are independently pre-trained on open datasets.

Furthermore, KGLiDS converts data discovery queries into SPARQL queries against our LiDS

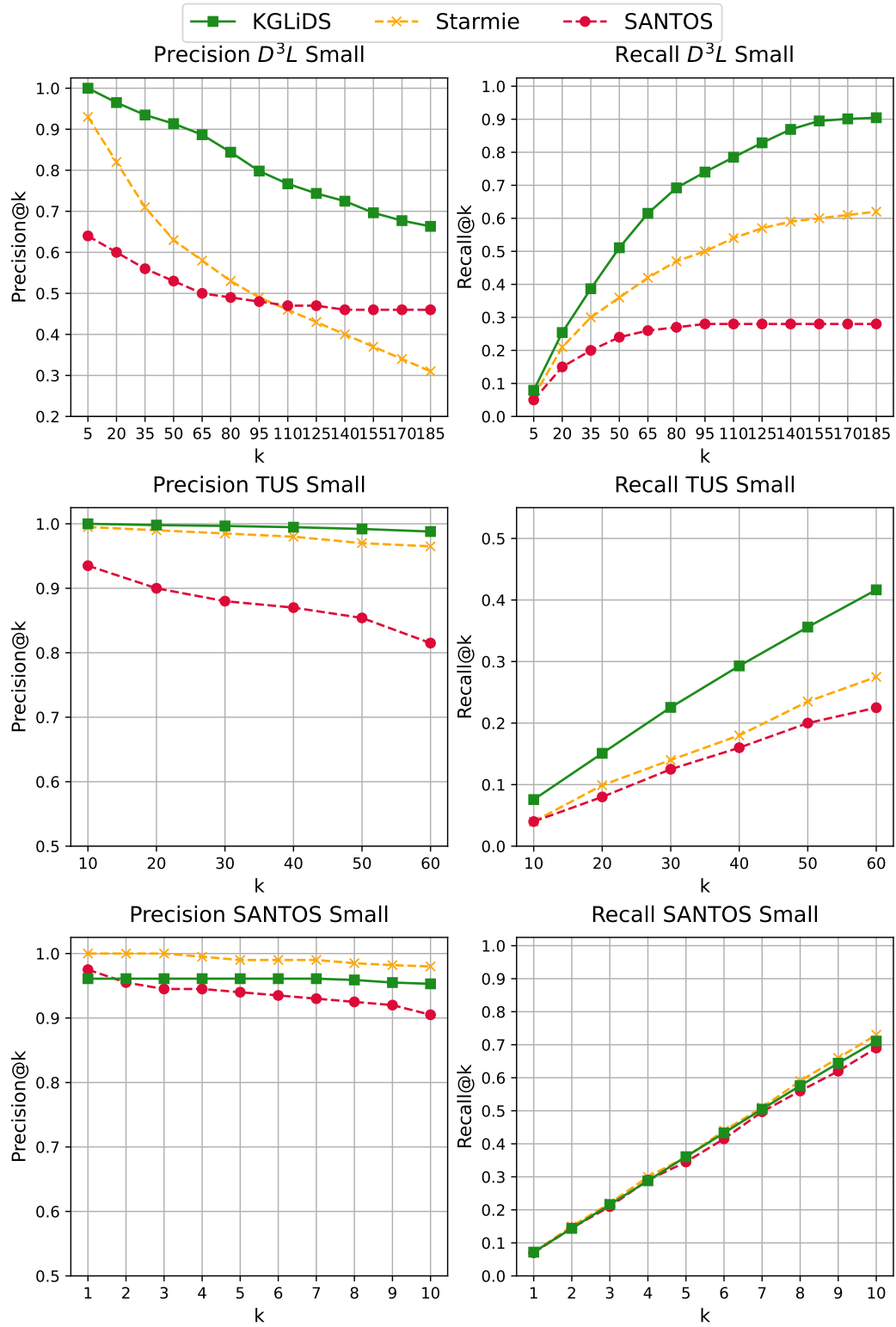


Figure 3.3: Average precision and recall of unionable table discovery on all benchmark datasets.

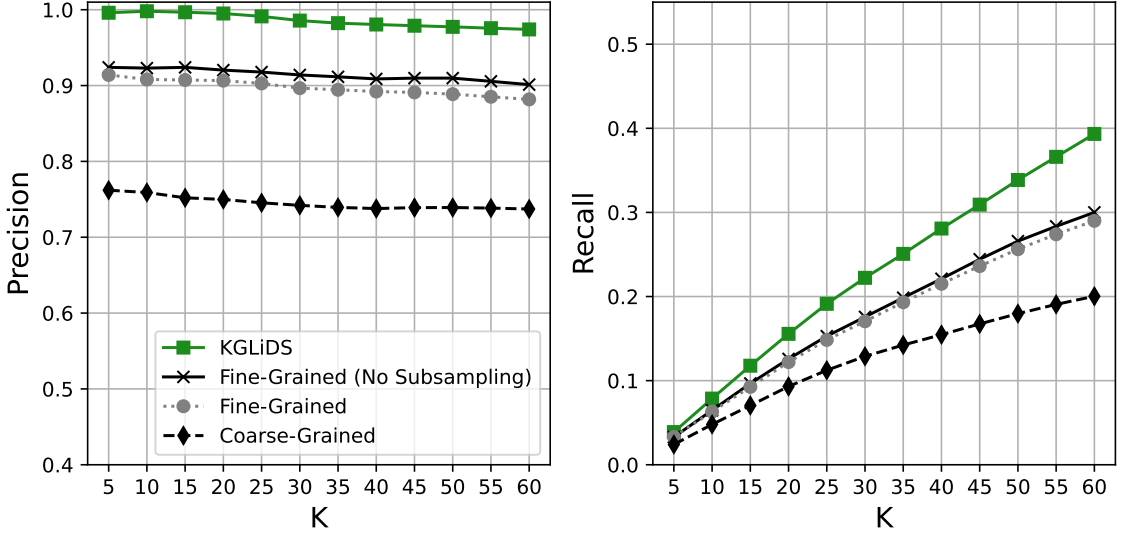


Figure 3.4: Ablation study for the effectiveness of KGLiDS for table union search on the TUS Small benchmark.

graph. We designed these queries to leverage the built-in indices in RDF engines. Hence, KGLiDS is 51.2x faster than SANTOS in the `SANTOS Large` benchmark, as shown in Table 3.2. For a table unionability query, SANTOS retrieves a list of candidate unionable tables by looking up two indices (one for each KB) for column relationships similar to those found between pairs of columns in the query table. Then, the query table is matched against candidate tables at different granularities to determine the unionability score for each candidate. This results in many comparisons and higher query times for SANTOS. Similarly, KGLiDS is 3.3x faster than Starmie in average query time. While Starmie uses an efficient vector index (Hierarchical Navigable Small World (HNSW) [75]) to retrieve similar columns based on embeddings, it requires calculating an approximate distance between embeddings of size 768.

3.3.5 Ablation Study

We conducted an ablation study to analyze our models for generating column and label embeddings.

3.3.5.1 Impact of column names and values on data discovery

Figure 3.4 shows the precision and recall of different KGLiDS configurations on the TUS Small benchmark. The highest scores were achieved by utilizing both the CoLR and label embeddings (indicated by the green line). In existing benchmarks, column names are a strong predictor of table unionability. However, raw value-based unionability is useful for data lakes without column names (e.g. anonymized). Figure 3.4 illustrates that KGLiDS still performs competitively when using only CoLR embeddings w.r.t Starmie and SANTOS, whose TUS results are shown in Figure 3.3. This predicts unionability based only on raw values (indicated as “Fine-Grained”) in Figure 3.4.

3.3.5.2 Impact of fine-grained vs. coarse-grained CoLR models

We developed coarse-grained embedding models inspired by [80], which introduced an embedding-based method through three coarse-grained models. We evaluated these coarse-grained embedding models and our fine-grained CoLR models. The results indicate that our fine-grained CoLR models significantly improved precision and recall, with precision reaching up to 17% higher and recall up to 50% higher at $k = 60$, as illustrated in Figure 3.4. We analyzed the impact of subsampling columns when generating column embeddings using our CoLR models, as outlined in Section 3.1. Using the CoLR models on column samples representing 10% of the original column sizes demonstrates comparable precision and recall to using them on entire columns. Our subsampling approach reduces the profiling time for the TUS Small benchmark by 14%.

3.3.5.3 Strong scalability test of Data Profiler and KG Builder

We perform a strong scalability test, where we fix the data lake (SANTOS Large) and vary the number of Spark workers between 2 and 16. Figure 3.5 shows the time taken by both the data profiling (Algorithm 1) and KG Builder (Algorithm 2). As shown, KGLiDS scales up as more Spark workers are added with an average scalability ratio of 0.86, i.e. KGLiDS utilizes 86% of each added worker on average to improve the runtime.

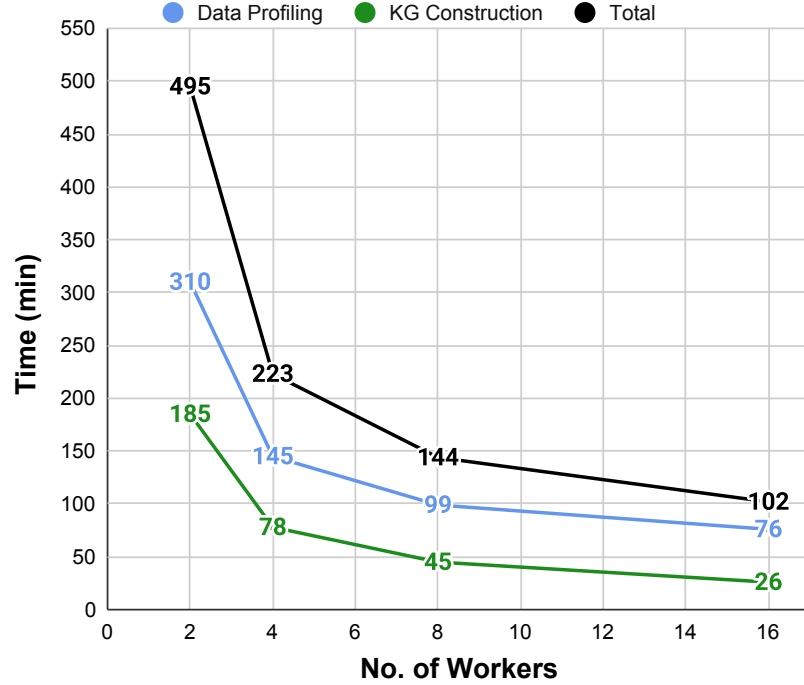


Figure 3.5: Strong scalability tests of KGLiDS data profiling and KG Builder on SANTOS Large benchmark with a Spark cluster of different numbers of workers.

3.4 Related Work

3.4.1 Data Discovery Systems

profile datasets to construct different navigational data structures. Starmie [29] uses embeddings from language models with an HNSW [75] index. SANTOS [65] builds column relationship trees by matching column values against open and synthetic knowledge bases. D^3L [13] constructs a variety of feature vectors for each column and matches them using hash-based indices. These systems rarely use open standards. We are the first to design navigational data structures capturing the semantics of datasets and pipelines based on knowledge graph technologies. Furthermore, we enable several data discovery operations, such as unionable tables, joinable tables, and join path discovery. Unlike [120, 122, 84], KGLiDS also enables users to query datasets based on embeddings and combined easily different operations in one pipeline script. Our KGLac [44] demonstrated the capabilities of KGLiDS’s data discovery support.

3.4.2 Table representation learning (TRL)

aims to construct fixed-size, low-dimensional representations for tables. Traditional approaches construct table or column embeddings using feature-based ML models [117] or deep neural networks [80]. The approach of [80] was originally proposed for variable recognition. KGLiDS adapts this approach and improves it by training fine-grained CoLR models and using them to predict relationships between data items. More recent approaches such as [18, 22, 52, 54] generate TRL using language models. These approaches are orthogonal to KGLiDS, which is modular for integrating new models seamlessly.

3.5 Chapter Summary

This chapter introduced the KGLiDS data discovery system, a novel approach designed to overcome the accuracy and scalability limitations of existing data discovery methods. While prior systems often rely on hand-crafted features or language models ill-suited for non-textual data, KGLiDS implements a more robust, two-stage process centered on fine-grained semantic profiling and knowledge graph construction.

The first stage, Embedding-Based Data Profiling, creates a detailed semantic profile for every column in a data lake. A key innovation in this stage is the use of seven fine-grained data types, which are empirically derived and operationally significant for data science tasks. This fine-grained typing allows for more precise comparisons and serves as the foundation for training six specialized Column Learned Representation (CoLR) models. These models, adapted from prior work, are refined to handle the nuances of each data type, resolving semantic mismatches present in baseline approaches and ultimately producing more accurate column embeddings for both names and values. The second stage, Knowledge Graph Construction, transforms the collection of individual column profiles into a connected Data Global Schema. This knowledge graph represents the entire data lake, with nodes for data sources, datasets, tables, and columns, linked by both hierarchical and semantic similarity edges.

A comprehensive evaluation on four standard data discovery benchmarks demonstrates the effectiveness of our approach. When compared against state-of-the-art systems, KGLiDS achieves

significantly higher accuracy, with at least 1.5x greater precision and recall on challenging, real-world benchmarks. Furthermore, the system proves to be more scalable, running 1.8x faster in preprocessing and 3.3x faster in query time, validating the superiority of our approach.

Chapter 4

AutoML via Graph Neural Networks

AutoML is the process by which machine learning models are built automatically for a new dataset. Given a dataset, AutoML systems perform a search over valid data transformations and learners, along with hyper-parameter optimization for each learner [69]. Choosing the transformations and learners over which to search is our focus. A significant number of systems mine from prior runs of pipelines over a set of datasets to choose transformers and learners that are effective with different types of datasets (e.g. [33], [113], [32]). Thus, they build a database by actually running different pipelines with a diverse set of datasets to estimate the accuracy of potential pipelines. Hence, they can be used to effectively reduce the search space. A new dataset, based on a set of features (meta-features) is then matched to this database to find the most plausible candidates for both learner selection and hyper-parameter tuning. This process of choosing starting points in the search space is called meta-learning for the cold start problem.

Other meta-learning approaches include mining existing data science code and their associated datasets to learn from human expertise. The AL [17] system mined existing Kaggle notebooks using dynamic analysis, i.e., actually running the scripts, and showed that such a system has promise. However, this meta-learning approach does not scale because it is onerous to execute a large number of pipeline scripts on datasets, preprocessing datasets is never trivial, and older scripts cease to run at all as software evolves. It is not surprising that AL therefore performed dynamic analysis on just nine datasets.

Our system, KGpip, provides a scalable meta-learning approach to leverage human expertise,

using static analysis to mine pipelines from large repositories of scripts. Static analysis has the advantage of scaling to thousands or millions of scripts [1] easily, but lacks the performance data gathered by dynamic analysis. The KGpip meta-learning approach guides the learning process by a scalable dataset similarity search, based on dataset embeddings, to find the most similar datasets and the semantics of ML pipelines applied on them. Many existing systems, such as Auto-Sklearn [32] and AL [17], compute a set of meta-features for each dataset. We developed a deep neural network model to generate embeddings at the granularity of a dataset, e.g., a table or CSV file, to capture similarity at the level of an entire dataset rather than relying on a set of meta-features.

Because we use static analysis to capture the semantics of the meta-learning process, we have no mechanism to choose the **best** pipeline from many seen pipelines, unlike the dynamic execution case where one can rely on runtime to choose the best performing pipeline. Observing that pipelines are basically workflow graphs, we use graph generator neural models to succinctly capture the statically-observed pipelines for a single dataset. In KGpip, we formulate learner selection as a graph generation problem to predict optimized pipelines based on pipelines seen in actual notebooks.

KGpip does learner and transformation selection, and hence is a component of an AutoML systems. To evaluate this component, we integrated it into two existing AutoML systems, FLAML [108] and Auto-Sklearn [32]. We chose FLAML because it does not yet have any meta-learning component for the cold start problem and instead allows user selection of learners and transformers. The authors of FLAML explicitly pointed to the fact that FLAML might benefit from a meta-learning component and pointed to it as a possibility for future work. For FLAML, if mining historical pipelines provides an advantage, we should improve its performance. We also picked Auto-Sklearn as it does have a learner selection component based on meta-features, as described earlier [31]. For Auto-Sklearn, we should at least match performance if our static mining of pipelines can match their extensive database. For context, we also compared KGpip with the recent VolcanoML [69], which provides an efficient decomposition and execution strategy for the AutoML search space. In contrast, KGpip prunes the search space using our meta-learning model to perform hyperparameter optimization only for the most promising candidates.

The contributions of this chapter are the following:

- Section 4.1 defines a scalable meta-learning approach based on representation learning of mined ML pipeline semantics and datasets for over 100 datasets and 11K Python scripts.
- Section 4.2 formulates AutoML pipeline generation as a graph generation problem. KGpip efficiently predicts optimized ML pipelines for unseen datasets using our meta-learning model. To the best of our knowledge, we are the first to formulate AutoML pipeline generation this way.
- Section 4.3 presents a comprehensive evaluation using a large collection of 121 datasets from major AutoML benchmarks and Kaggle. Our experimental results show that KGpip outperforms all existing AutoML systems and achieves state-of-the-art results on the majority of these datasets. KGpip significantly improves the performance of both FLAML and AutoSklearn in classification and regression tasks. We also outperformed AL in 75 out of 77 datasets and VolcanoML in 75 out of 121 datasets, including 44 datasets used only by VolcanoML [69]. On average, KGpip achieves scores that are statistically better than the means of all other systems.

4.1 The KGpip Scalable Meta-Learning

Our meta-learning approach is based on mining large databases of ML pipelines associated with the used datasets, as illustrated in Figure 4.1. The mining process uses static program analysis instead of executing the actual pipeline scripts or preparing the actual raw data. The KGpip meta-learning component enhances the search strategy of existing AutoML systems, such as AutoSklearn and FLAML, and allows these systems to handle ad-hoc datasets, i.e., unseen ones. To retain a maximal degree of flexibility, KGpip captures metadata and semantics in a flexible graph format, and relies on graph generator models as the database of pipelines.

Unlike existing meta-learning approaches, our approach is designed to learn from a large-scale database and achieve high degree of coverage and diversity. Several ML portals, such as Kaggle or OpenML [103], provide access to thousands of datasets associated with hundreds of thousands of public notebooks, i.e., ML pipelines/code. KGpip mines these large databases of datasets and

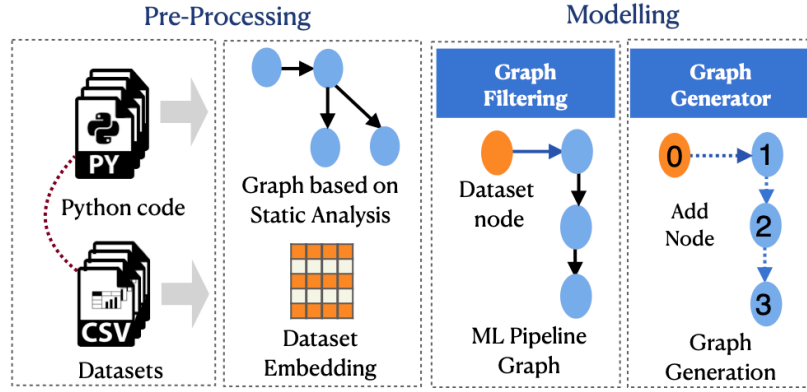


Figure 4.1: An overview of KGpip’s meta-learning approach for mining a database of ML pipelines to train a graph generator model to predict ML pipeline skeletons in the form of graphs.

pipelines using static analysis and filters them into ML pipelines customized for the learner selection problem. The KGpip meta-learning approach leverages [1] for code understanding via static analysis of scripts/code of ML pipelines. It extracts the semantics of these scripts as code and form an initial graph for each script.

KGpip cleans the graphs generated by [1] to keep the semantic required for the ML meta-learning process. Furthermore, our approach introduces dataset nodes and interlinks the relevant pipeline semantic to them. So, our meta-learning approach produces MetaPip, a highly interconnected graph of seen datasets and pipelines applied to them. We also developed a deep embedding model to find the closest datasets to an unseen one, i.e., to effectively prune MetaPip. We then train a deep graph generator model [70] using MetaPip. This model is the core of our meta-learning component as illustrated in Figure 4.1 and discussed in the next section.

4.1.1 Graph Representation of Code Semantics

Static and dynamic program analysis techniques could be used to abstract the semantics of programs and extract language-independent representations of code. A program source code is examined in the static analysis without running the program. In contrast, dynamic analysis examines the source code during runtime to collect memory traces and more detailed statistics specific to the analysis technique. Unlike static analysis, dynamic analysis helps in capturing more rich semantics from programs with the high cost of execution and storing massive memory traces. ML portals,

```
df = pd.read_csv('example.csv')
df_train, df_test = train_test_split(df)
X = df_train['X']
model = svm.SVC()
model.fit(X, df_train['Y'])
```

Figure 4.2: An example from a data science notebook.

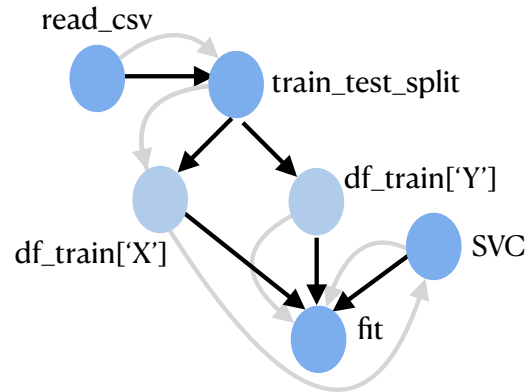


Figure 4.3: Code graph corresponding to Figure 4.2 obtained with GraphGen4Code. The graph shows control flow with gray edges and data flow with black edges. Numerous other nodes and edges are not shown for simplicity.

such as Kaggle, have hundreds of thousands of ML pipelines with no instructions for running or managing the environments of these pipelines. KGpip combines dataset embedding with static code analysis tools, such as GraphGen4Code [1], to enrich the collected semantics of ML pipelines while avoiding the need to run them.

GraphGen4Code is optimized to efficiently process millions of Python programs, performing interprocedural data flow and control flow analysis to examine for instance, what happens to data that is read from a Pandas dataframe, how it gets manipulated and transformed, and what transformers or estimators get called on the dataframe. GraphGen4Code’s graphs make it explicit what APIs and functions are invoked on objects without the need to model the used libraries themselves; hence GraphGen4Code can scale static analysis to millions of programs. Figures 4.2 and 4.3 show a small code snippet and its corresponding static analysis graph from GraphGen4Code, respectively. As shown in Figure 4.3, the graph captures control flow (gray edges), data flow (black edges), as well as numerous other nodes and edges that are not shown in the figure. Examples of these nodes and

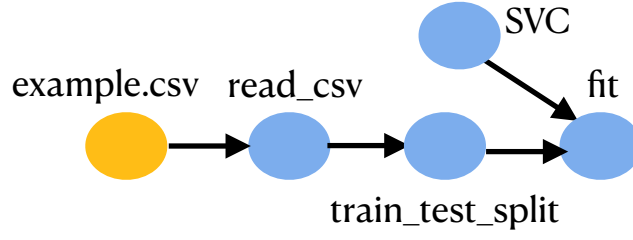


Figure 4.4: Our MetaPip graph of the graph from Figure 4.3, where the abstracted ML pipeline is linked to a dataset node (highlighted in Orange). MetaPip contains at least 96% less nodes and edges than the original graph while enhancing the overall quality of the graph generation process, as experimented in Section 4.3.5.

edges include those capturing location of calls inside a script file and function call parameters. For example, GraphGen4Code generates a graph of roughly 1600 nodes and 3700 edges for a Kaggle ML pipeline script of 72 lines of code. The number of nodes and edges dominate the complexity of training a graph generator model.

4.1.2 MetaPip: from Code to Pipeline Semantics

For AutoML systems, a pipeline is a set of data transformations, learner selection, and hyperparameter optimization for each model that is selected. Mined data science notebooks often contain data analysis, data visualization, and model evaluation. Moreover, each notebook is associated with one or more datasets. Thus, it is essential for our meta-learning model to distinguish between different types of pipelines and realize this association with datasets. Existing systems for static code analysis extract general semantics of code and cannot link pipeline scripts to the used datasets. Thus, the generated graphs by systems, such as GraphGen4Code, are scattered and unlinked, i.e., a graph per an ML pipeline script. Moreover, each graph will have nodes and edges that are not relevant for the meta-learning process. These irrelevant nodes and edges, i.e., triples, will add noise to the training data. Hence, a meta-learning model will not be able to learn from the abstracted graph pipelines generated by such tools, as shown in Table 4.4. We developed a method to filter out this kind of triples from GraphGen4Code’s graph and analyze ML pipelines to prepare a training set interconnecting repositories of ML pipeline scripts with their associated datasets. Moreover, our method cleans the noisy nodes and edges and calls to modules outside the target ML libraries. For example, our method will extract triples related to libraries, such as Scikit-learn, XGBoost, and

LGBM. These libraries are the most popular among the top-scoring ML pipelines in ML portals. The code for the cleaning method is available at the KGpip’s repository.

Our meta-learning component aims to pick learners and transformer for unseen datasets. Thus, KGpip links the filtered ML pipelines with the used datasets. The result of adding these dataset nodes is a highly interconnected graph for ML pipelines, we refer to it as *MetaPip*. Our MetaPip graph captures both the code and data aspects of ML pipelines. Hence, we can populate the MetaPip graph with datasets from different sources, such as OpenML and Kaggle, and pipelines applied on these datasets. Figure 4.4 shows the MetaPip graph corresponding to the code snippet in Figure 4.2. KGpip utilizes MetaPip to train a model based on a large set of pipelines associated with similar datasets. For example, a *pandas.read_csv* node will be linked to the used table node, i.e., csv file. In some cases, the code, which reads a csv file, does not explicitly mention the dataset name. The pipelines are usually associated with datasets, such as Kaggle pipelines and datasets, as shown in Figure 4.1.

4.1.3 Dataset Representation Learning

Our approach efficiently guides the meta-learning process by linking the extracted semantics of pipelines to dataset nodes representing the used datasets. There is a sheer amount of datasets of variable sizes and we need to develop a scalable method for finding the most similar datasets for an unseen one. The pairwise comparison based on the actual content of datasets, i.e., tuples in CSV files, does not scale. Thus, we developed a dataset representation learning method to generate a fixed-size and dense embedding at the granularity of a dataset, e.g., a table or CSV file. The embedding of a dataset $\mathcal{D}$ is the average of its column embeddings, i.e.:

$$h_{\theta}(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{c \in \mathcal{D}} h_{\theta}(c) \quad (7)$$

where $|\mathcal{D}|$ is the number of columns in $\mathcal{D}$. Our work generalizes the approach outlined in [81] for individual column embeddings, where column embeddings are obtained by training a neural network on a binary classification task. The model learns when two columns represent the same concept, but with different values, as opposed to columns representing different concepts. Embeddings for

an unseen dataset are produced by the last layer of the neural net.

KGpip reads datasets only once and leverages PySpark DataFrame to achieve high task and data parallelism. We use the embeddings of datasets to measure their similarity. With these embeddings, we build an index of vector embeddings for all the datasets in our training set. We utilize efficient libraries [60] for similarity search of dense vectors to retrieve the most similar dataset to a new input dataset based on its embeddings. Thus, our method scales well and leads to accurate results in capturing similarities between datasets.

4.2 The KGpip Pipeline Automation

The KGpip workflow for pipeline automation is based on our meta-learning model, as illustrated in Figure 4.5. KGpip predicts the top-K pipeline skeletons, i.e., a specific set $\{P, E\}$ of Preprocessor (P) and Estimators (E), for an unseen dataset (D) based on the most similar seen dataset (SD), i.e., the nearest neighbour dataset. KGpip starts by finding SD based on the embedding of the unseen dataset. Then, KGpip generates the top-K validated ML pipeline graphs VG and converts them into ML pipeline skeletons $\{P, E\}$. Then, it performs hyperparameter optimization using systems such as FLAML [108] and Auto-Sklearn [32] to find the optimum hyperparameters for each pipeline skeleton within a specific time budget.

4.2.1 Graph Generation for ML Pipelines

KGpip formulates the generation of ML pipelines as a graph generation problem. The intuition behind this idea is that a neural graph generator might capture more succinctly multiple pipelines seen in practice for a given dataset, and might also capture statistical similarities between different pipelines more effectively. To effectively use such a network, we add a single dataset node as the starting point for the filtered pipelines we generate from Python notebooks. The node is assumed to flow into a `read_csv` call which is often the starting point for the pipelines. For generating an ML pipeline, we simply pass in a dataset node for the nearest neighbour of the unseen dataset, i.e., the most similar dataset based on content similarity, as shown in Figure 4.5.

Our meta-learning model generates ML pipeline graphs in a sequential node-by-node fashion.

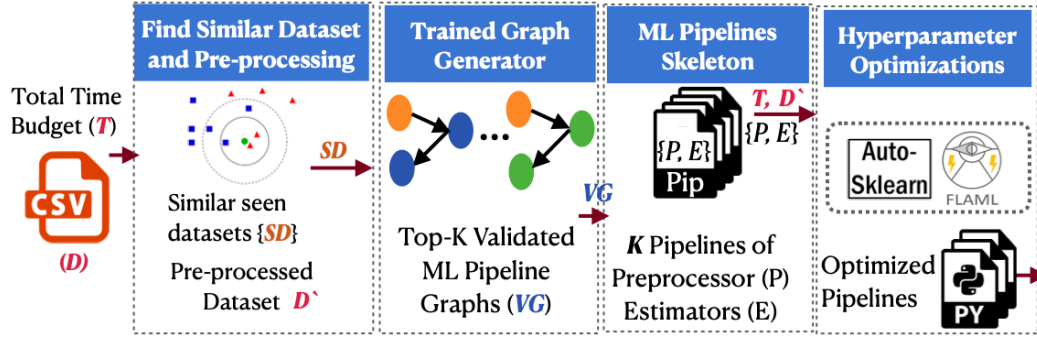


Figure 4.5: An overview of KGpip’s workflows of ML pipeline generation for a given unseen dataset and certain time budget. KGpip utilizes systems for hyperparameter optimization, such as FLAML or Auto-Sklearn, to optimize KGpip’s top-K predicted pipelines (VG), i.e., pruning the search space.

Algorithm 3 illustrates the implementation of the graph generation model. For an empty graph G and the most similar dataset SD , the algorithm starts by adding an edge between SD and `pandas.read_csv`. Then, the graph neural network $f_{AddNode}$ decides whether to add a new node of a certain type. The network $f_{AddEdge}$ decides whether to add an edge to the newly added node. Then, the network $f_{ChooseNode}$ decides the existing node to which the edge is to be added. The While loop at line 7 is repeated until no more edges to be added. The While loop at line 4 is repeated until no more nodes to be added. The three neural networks, namely $f_{AddNode}$, $f_{AddEdge}$, and $f_{ChooseNode}$, utilize node embeddings that are learned throughout training via graph propagation rounds. These embeddings capture the structure of ML pipeline graphs.

The generated graph G is not guaranteed to be a valid ML pipeline. Thus, Algorithm 3 at line 14 checks that G is a valid ML pipeline graph. In KGpip, a graph G is valid if 1) it contains at least one estimator matching the task, i.e., regression or classification, and 2) the estimator is supported by the hyperparameter optimizer (AutoSklearn or FLAML in our case). With these modifications, it is possible to generate ML pipelines for unseen datasets using the closest seen dataset node – more specifically, its content embedding obtained from the dataset embedding module. We built Algorithm 3 on top of the system proposed in [70]. This system does not support conditional graph generation at test time by default, i.e., building a graph on top of a provided dataset node. We extended this system to generate valid ML pipeline graphs, as illustrated in Algorithm 3.

Algorithm 3: Graph Generation Process

Input: Graph G : ($E = \phi$, $V = \phi$), Similar Dataset Node: SD ,
Neural Networks: $f_{AddNode}$, $f_{AddEdge}$, $f_{ChooseNode}$

- 1 $V \leftarrow V \cup \{SD, pandas.read_csv\}$
- 2 $E \leftarrow E \cup \{(SD, pandas.read_csv)\}$
- 3 $nodeToAdd = f_{AddNode}(V, E)$
- 4 **while** $nodeToAdd \neq Null$ **do**
- 5 $V \leftarrow V \cup \{nodeToAdd\}$
- 6 $addEdge = f_{AddEdge}(V, E)$
- 7 **while** $addEdge$ **do**
- 8 $nodeToLink = f_{ChooseNode}(V, E)$
- 9 $E \leftarrow E \cup \{(nodeToAdd, nodeToLink)\}$
- 10 $addEdge \leftarrow f_{AddEdge}(V, E)$
- 11 $nodeToAdd \leftarrow f_{AddNode}(V, E)$
- 12 $VG = validate_pipeline_graph(G)$
- 13 **return** VG

4.2.2 Hyperparameter Optimization

KGpip maps the valid graphs into ML pipeline skeletons, where each skeleton is a set of pre-processors and an estimator with place holders for the optimal parameters. In KGpip, the hyperparameter optimizer is responsible for finding the optimal parameters for the pre-processors and learners on the target dataset. Then, KGpip replaces the place holders with these parameters. Finally, KGpip creates a python script using the pre-processors and estimator achieving the highest scores. KGpip is well designed to support both numerical and non-numerical datasets. Thus, KGpip applies different pre-processing techniques on the given dataset (D) and produces a pre-processed dataset (D'). Our pre-processing includes 1) detecting task type (i.e. regression or classification) automatically based on the distribution of the target column 2) automatically inferring accurate data types of columns, 3) vectorizing textual columns using word embeddings [19], and 4) imputing missing values in the dataset. In KGpip, the hyperparameter optimizer uses D' .

Similar to hyperparameter optimizers implemented in AutoML systems, such as FLAML or Auto-Sklearn, KGpip works within a provided time budget per dataset. We note here that the majority of the allotted time budget for ML pipeline generation is spent on the hyperparameter optimization; that is, if the user desires only to know what learners would work best for their dataset, KGpip can do that almost instantaneously. Given a time budget (T), KGpip calculates t , the time

consumed in generating and validating the graphs. KGpip then divides the rest of the time budget between the K graphs. Hence, the hyperparameter optimizer has a time limit of $((T - t)/K)$ to optimize each graph independently.

The hyperparameter optimizer repeatedly applies the learners and pre-processors with different configurations while monitoring the target score metric throughout. KGpip keeps updating its output with the best pipeline skeleton, i.e., learners and pre-processors, and its score. For example, if the predicted learner is LogisticRegression, it searches for the best combination of regularization type (L1 or L2) and regularization parameter. The difference between hyperparameter optimizers is the search strategy followed to arrive at the best hyperparameters within the allotted time budget. A naive approach would be to perform an exhaustive grid search over all combinations, while a more advanced approach would be to start with promising configurations first. We integrate KGpip with the hyperparameter optimizers of both FLAML [108] and Auto-Sklearn [32] to demonstrate the generality of KGpip. The integration of a hyperparameter optimizer into KGpip needs a JSON document of the particular preprocessors and estimators supported by the hyperparameter optimizer. Thus, the integration is relatively easy. Finally, our neural graph generation produces a diverse set of pipelines across runs, allowing for exploration and exploitation.

4.3 Experiments

4.3.1 Benchmarks

We evaluate KGpip as well as the other baselines on four benchmark datasets: 1) *Open AutoML Benchmark* [38], a collection of 39 binary and multi-class *classification* datasets (used by FLAML [108]). The datasets are selected such that they are representative of the real world from a diversity of problem domains and of enough difficulty for the learning algorithms. 2) *Penn Machine Learning Benchmark* (PMLB) [86]: Since Open AutoML Benchmark is limited to classification datasets, the authors of FLAML [108] evaluated their system on 14 more *regression* datasets selected from PMLB, such that the number of samples is more than 10,000. To demonstrate the generality of our approach, we include those datasets in our evaluation as well. 3) *AL's datasets*: We also evaluate on the datasets used for AL's [17] evaluation which include 6 Kaggle datasets (2 regression and

Table 4.1: Breakdown of all 121 datasets used in our evaluation, indicating those used by FLAML^{*}, AL[†], and VolcanoML[§].

Task	Source			
	AutoML	PMLB	OpenML	Kaggle
Binary	22 (18 [*] +1 [†] +3 [§])	5 (4 [†] +1 [§])	27 (3 ^{†§} +3 [†] +21 [§])	2 [†]
Multi-class	17 (15 [*] +1 [†] +1 [§])	4 [†]	7 (2 ^{†§} +1 [†] + 4 [§])	2 [†]
Regression	0	14 [*]	19 [§]	2 [†]
Total	39	23	53	6

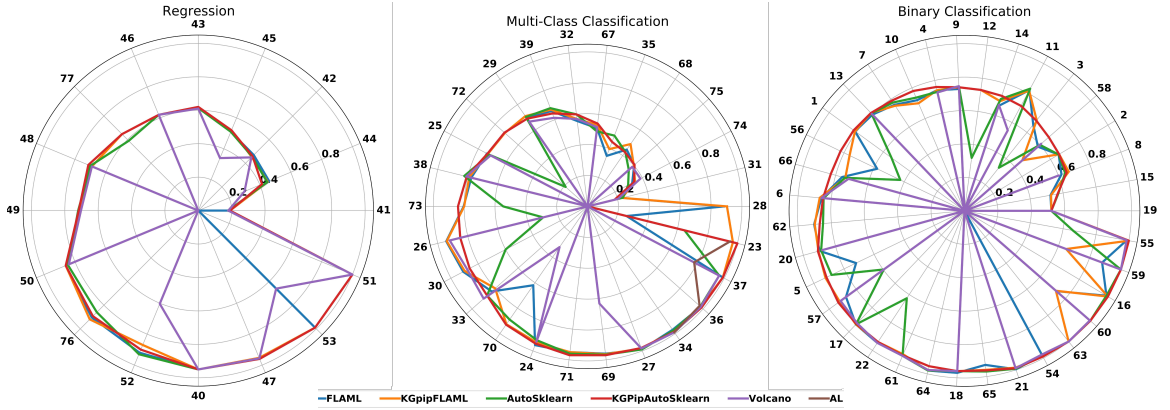


Figure 4.6: A radar diagram of the performance of KGpip vs. existing systems on multiple tasks (77 datasets) with a time budget of 1 hour for all systems. The outer numbers indicate different dataset IDs and the ticks inside the figure denote performance ranges of respective metrics; e.g., 0.2, 0.4, ..., etc. for F1 in binary classification. For any dataset, the system with the outermost curve has the best performance. As an example, KGpipAutoSklearn and KGpipFLAML achieved 100% and 97% F1 on dataset #23 (multi-class classification) compared to 65% and 26% for AutoSklearn and FLAML, respectively.

4 classification) and another 18 classification datasets (9 from PMLB and 9 from OpenML). Unlike other benchmarks, the Kaggle datasets include datasets with textual features. 4) VolcanoML’s datasets: finally, we evaluate KGpip on 44 more datasets used by VolcanoML [69]. The authors of VolcanoML evaluate their system on a total of 66 datasets from OpenML and Kaggle, from which 11 datasets are not specified, 10 datasets overlap with ours, and 1 dataset consists of image samples. Table 4.1 includes a summary of all 121 benchmark datasets. The detailed statistics of all datasets are shown in the appendix of [46]. These statistics include names, number of rows and columns, number of numerical, categorical, and textual features, number of classes, sizes, sources, and papers that evaluated on them.

4.3.2 Baselines

We empirically validate KGpip against three AutoML systems: (1) Auto-Sklearn (v0.14.0) [32] which is the overall winner of multiple challenges in the ChaLearn AutoML competition [40], and one of the top 4 competitors reported in the Open AutoML Benchmark [38]. (2) FLAML (v0.6.6) [108]: an AutoML library designed with both accuracy and computational cost in mind. FLAML outperforms Auto-Sklearn among other systems on two AutoML benchmarks using a low computational budget, (3) AL [17]: a meta-learning-based AutoML approach that utilizes dynamic analysis of Kaggle notebooks, an approach that has similarities to ours, and (4) VolcanoML (v0.5.0) [69], a recent AutoML approach which proposes efficient decomposition strategies for the large AutoML search spaces. In all our experiments, we used the latest code provided by the authors for existing systems, the same exact hardware, time budget, and the parameters recommended by the authors of these systems.

4.3.3 Training Setup

Because our approach to mining historical pipelines from scripts is relatively cheap, we can apply it more easily on a wider variety of datasets to form a better base as more and more scripts get generated by domain experts on Kaggle competitions. In this work, we performed program analysis on 11.7K scripts associated with 142 datasets, and then selected those with estimators from `sklearn`, `XGBoost` and `LightGBM` since those were the estimators supported by the most AutoML systems for classification and regression. This resulted in the selection of 2,046 notebooks for 104 datasets; a vast portion of the 11.7K programs were about exploratory data analysis, or involved libraries that were not supported by Auto-Sklearn [32] or FLAML (e.g., PyTorch and Keras) [108]. We used Macro F1 for classification tasks to account for data imbalance, if any, and use R^2 for regression tasks, as in FLAML [108]. We also varied the time budget given to each system between 1 hour and 30 minutes, to measure how fast can KGpip find an efficient pipeline compared to other approaches. The time budget is end-to-end, from loading the dataset till producing the best AutoML pipeline. In all experiments, we report averages over 3 runs.

Table 4.2: Average scores (mean and standard deviation) of KGpip compared to FLAML, Auto-Sklearn, and VolcanoML for binary classification (F1), multi-class classification (F1) and regression (R^2) tasks on 77 benchmark datasets. T-test values are for KGpip vs. FLAML and KGpip vs. Auto-Sklearn.

	Binary	Multi-class	Regression	T-Test
FLAML	0.74 (0.23)	0.70 (0.29)	0.65 (0.29)	0.0129
KGpipFLAML	0.81 (0.14)	0.76 (0.24)	0.72 (0.24)	-
Auto-Sklearn	0.76 (0.20)	0.65 (0.29)	0.71 (0.24)	0.0002
KGpipAutoSklearn	0.83 (0.14)	0.73 (0.28)	0.72 (0.24)	-
VolcanoML	0.55 (0.43)	0.51 (0.38)	0.56 (0.32)	-

4.3.4 Comparison with Existing Systems

In this section, we evaluate KGpip against state-of-the-art systems: FLAML [108] and Auto-Sklearn [32]. Figure 4.6 shows a radar graph of all systems when given a time budget of 1 hour. It shows the performance of all systems on the three tasks in all benchmarks, namely, binary classification, multi-class classification, and regression. For every dataset, the figure shows the actual performance metric (F1 for classification and R^2 for regression) obtained from every system ¹. Therefore, the outermost curve from the center of the radar graph has the best performance. In Figure 4.6, both variations of KGpip achieve the best performance across all tasks, outperforming both FLAML and Auto-Sklearn. We also performed a *two-tailed t-Test* between the performance obtained by KGpip compared to the other systems. The results show that KGpip achieves significantly better performance than both FLAML and Auto-Sklearn with a t-Test value of 0.01 and 0.0002, respectively (both have $p < 0.05$).

Table 4.2 also shows the average F1 and R^2 values for classification and regression tasks, respectively. The results show that both variations of KGpip achieve better performance compared to both FLAML and Auto-Sklearn over all tasks and datasets.

Scalability of KGpip’s meta-learning against existing systems: The AL meta-learning approach [17] mines pipelines using dynamic code analysis, which has high cost as discussed in Section 4.1.1. Thus, the authors of AL provided a pre-trained meta-learning model on 500 pipelines and 9 datasets, which does not scale to cover various cases. In contrast, we trained our meta-learning model using

¹The detailed scores for every system and dataset as well as the corresponding names of datasets are shown in the appendix of [46]

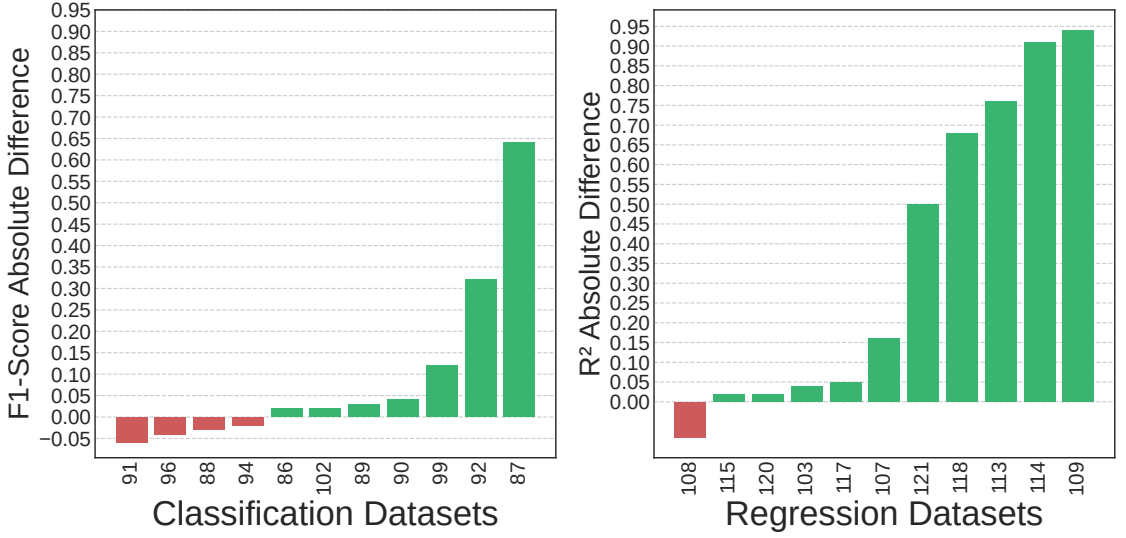


Figure 4.7: Score difference between KGpipFLAML and VolcanoML on the 44 classification and regression datasets from VolcanoML with a time budget of 1 hour. For brevity, we removed from this Figure the 22 datasets on which both systems perform comparably (within a difference of ≤ 0.01).

Table 4.3: Average scores (mean and standard deviation) of KGpipFLAML compared to VolcanoML on the 44 datasets from VolcanoML. Overall, KGpipFLAML achieves significantly better compared to VolcanoML, according to a statistical significance test of $p < 0.05$.

	Binary	Multi-class	Regression	T-Test
KGpipFLAML	0.82 (0.14)	0.86 (0.16)	0.83 (0.13)	-
VolcanoML	0.69 (0.23)	0.70 (0.31)	0.68 (0.25)	0.0001

2000 pipelines and 142 datasets. None of these datasets were included in the 77 datasets used in testing. AL failed in 22 and timed out in 38 datasets. This shows that the KGpip meta-learning approach, which is based on pipelines semantics and dataset representation learning, is more effective. AL failed on many of the datasets during the fitting process. As the figure shows, KGpip still outperforms all other approaches, including AL, significantly. On these datasets, AL achieved the lowest F1 score on binary and multi-class classification tasks with values of 0.36 and 0.36, respectively. This compares to 0.74 and 0.75 by FLAML, 0.73 and 0.68 by Auto-Sklearn, 0.79 and 0.79 by KGpipFLAML, and 0.79 and 0.74 by KGpipAuto-Sklearn.

VolcanoML Datasets: VolcanoML used a variety of datasets that are not included in our 77 datasets of Figure 4.6. Some of these datasets are quite large, which are meant to test the system’s

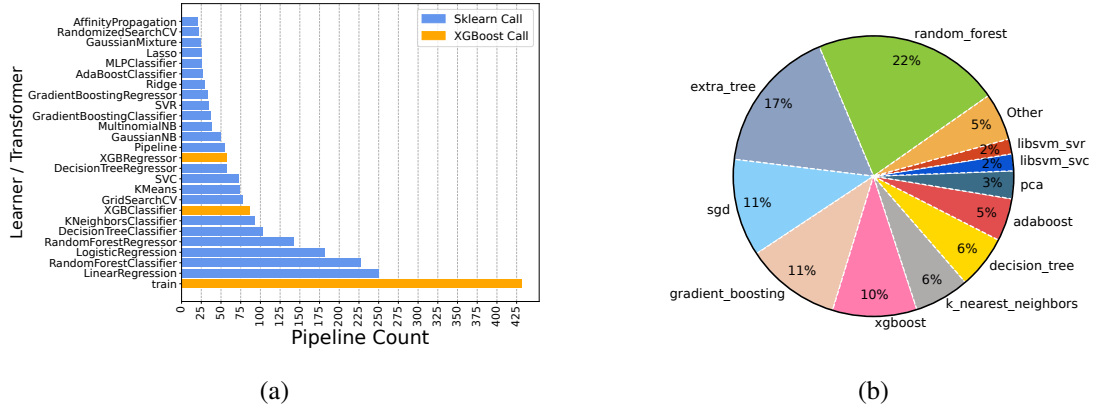


Figure 4.8: Top learner and transformers selected by KGpip (a) are with a wide range of coverage and diversity (b).

Table 4.4: Different aspects comparing a model trained on a set of code graphs vs a model trained on a set of MetaPip graphs. The model based on original code graphs fails in trivial datasets to generate valid pipelines and limits KGpip’s scalability to a larger set of ML pipelines scripts and KGpip’s learning by using a fewer number of epochs.

Dataset/Aspect	Code Graph	MetaPip Graph
kr-vs-kp	0 ₍₀₎	1.00 ₍₀₎
nomao	0 ₍₀₎	0.96 ₍₀₎
cnae-9	0 ₍₀₎	0.95 _(0.01)
mfeat-factors	0 ₍₀₎	0.98 ₍₀₎
segment	0 ₍₀₎	0.98 ₍₀₎
Avg. F1	0 ₍₀₎	0.97 _(0.02)
No. Nodes	29,139	974
No. Edges	252,486	1,052
Training Time	175 (min)	2 (min)

scalability. Therefore, we also collected all 49 datasets we could find in their paper and tested the best version of KGpip (KGpipFLAML) against VolcanoML on these datasets with a time budget of 1 hour. The performances of KGpipFLAML and VolcanoML are shown in Figure 4.7. For brevity, we omitted from the figure all datasets on which the performance difference between both systems is ≤ 0.01 and the 5 datasets overlapping with the ones shown in Figure 4.6. On those datasets, KGpipFLAML found a valid pipeline for all of them, sometimes with a decent absolute difference in F1 or R^2 scores of ≥ 0.90 . Across all the 44 datasets, KGpipFLAML achieved significantly better average scores compared to VolcanoML (statistical significance test of $p < 0.05$), see Table 4.3 for details.

4.3.5 Ablation Study

4.3.5.1 The effectiveness of MetaPip

Our MetaPip approach manages to reduce dramatically the number of nodes and edges in the code graph. Using the original graph obtained from static analysis, it produces the MetaPip graph that focuses on the core aspects needed to train a graph generation model for ML pipelines, such as data transformations, learner selection, and hyper-parameter selection. This experiment investigates the scalability of our graph generation model based on two different training sets, i.e., the sets of MetaPip graphs described in section 4.1.2 vs. the original set of code graphs from static analysis for the same ML pipeline scripts.

For this experiment, we use a small-scale training set of 82 pipeline graphs pertaining to one classification dataset. The original code graphs for these 82 pipelines include 29,139 nodes and 252,486 edges. Our MetaPip graph, however, includes 974 nodes and 1052 edges. This is a graph reduction rate of at least 96.6%, Figure 4.4 shows these detailed statistics. The main investigation here is whether this huge reduction ratio will help improve the accuracy and scalability of our graph generation model. We train one model on the original code graph and another on the MetaPip graphs. Both models are trained for 15 epochs with the same set of hyperparameters. It is worth noting that due to the huge time required to process the nodes and edges in the code graph, we had to reduce the number of epochs from 400 to 15.

We test the performance of KGpip when trained on both graphs on the most trivial binary and multi-class classification datasets in the AutoML benchmark. These are the datasets where the F1 score of all the reported systems in section 4.3.4 is above 0.9. The result is a total of 5 datasets (1 binary and 4 multi-class). Both models use Auto-Sklearn as the hyperparameter optimizer with a time budget of 15 minutes and 3 graphs. We take the average of three runs. The results are summarized in Table 4.4. For these trivial datasets, the model trained using code graphs did not manage to generate any valid ML pipeline. This means the model failed to capture the core aspects of ML pipelines, i.e., valid transformation or learners. Moreover, our MetaPip approach helps KGpip to reduce the training time by 99%, as shown in Table 4.4.

Table 4.5: Performance of KGpipFLAML (mean and standard deviation) as we vary the number of predicted pipeline graphs within 30 minutes time limit. We obtained similar results for KGpipAutoSklern, and hence omitted its results.

	Binary	Multi-Class	Regression
Top-3 graphs	0.80 (0.14)	0.70 (0.31)	0.71 (0.23)
Top-5 graphs	0.81 (0.14)	0.73 (0.26)	0.70 (0.23)
Top-7 graphs	0.81 (0.14)	0.75 (0.24)	0.71 (0.24)

4.3.5.2 The KGpip meta-learning quality

This experiment tests the quality of our meta-learning component. We test the performance as we vary the number of graphs selected from the graph generation phase before feeding it to the hyper-parameter optimization module. Table 4.5 shows the KGpipFLAML performance as we vary the number of predicted graphs between 3, 5 and 7.

With only 3 graphs, KGpip is still outperforming FLAML (second best system after KGpipFLAML), although the effect is weaker (t-Test value = 0.06). Compared to Auto-sklern (third best system after KGpipFLAML), all variations have similar or better performance, but the difference is insignificant. This experiment shows that even with three graphs, KGpip outperforms FLAML and KGpipAuto-Sklern, i.e., the correct pipelines often appear in the top 3. As another assessment of the quality of our predictions, we measure where in our ranked list of predicted pipelines the best pipeline turned out to be. Ideally, the top pipeline would always be first, and we use Mean Reciprocal Rank (MRR) to measure how close to that our predictions are. Across all runs, the MRR is 0.71, indicating that the top pipeline is typically very near the top.

4.3.5.3 The KGpip meta-learning diversity

One question we address is whether KGpip produced different pipelines for the *same dataset* across different runs. This gives us a sense of whether KGpip is deterministic, or whether it produces different pipelines to help with pruning the AutoML search space. We took different runs for the exact same dataset, and created a list of learners and transformers produced for each dataset across runs. The list was limited by the shortest number of learners and transformers produced across runs. We then computed correlations for datasets across runs 1, 2, and 3. The correlations ranged from

0.60 - 0.64, suggesting that the runs did not produce the same transformers and learners across runs. We also examined the types of learners selected by KGpip for consideration. Figure 4.8a shows the learners and transformers found at least 20 times in the training pipelines. One can see from the figure that KGpip does not blindly output learners and transformers by counts. Figure 4.8b shows more diversity in what was selected overall. So, a variety of methods are covered by KGpip.

4.4 Related Work

In this section, we summarize the related work and restrict our review to meta-learning approaches for AutoML, dataset embeddings, and processing tabular structured data.

Learner and preprocessing selection

In most AutoML systems, learner and pre-processing selection for the cold start problem is driven by a database of actual executions of pipelines and data; e.g., [17], [32], [33]. This database often drives both learner selection and hyper parameter optimization (HPO), so we focus here more on how the database is collected or applied to either problem, since the actual application to learner selection or HPO is less relevant. For HPO, some have cast the application of the database as a multi-task problem (see [99]), where the hyperparameters for cold start are chosen based on multiple related datasets. Others, for instance, [32, 95], compute a database of dataset meta-features on a variety of OpenML [103] datasets, including dataset properties such as the number of numerical attributes, the number of samples or skewness of the features in each dataset.

These systems measure similarity between datasets and use pipelines from the nearest datasets based on the distance between the datasets' feature vectors as we do, but the computation of these vectors is different, as we describe in detail below. Auto-Sklearn 2.0 [31] defines instead a static portfolio of pipelines that work across a variety of datasets, and use these to cold-start the learner selection component - i.e., every new dataset uses the same set of pipelines. Others have created large matrices documenting the performance of candidate pipelines for different datasets and viewed the selection of related pipelines as a collaborative filtering problem [33].

Dataset embeddings

The most used mechanism to capture dataset features rely on the use of meta-features for a dataset such as [32, 95]. These dataset properties vary from simple, such as number of classes (see, e.g. [27]), to complex and expensive, such as statistical features (see, e.g. [105]) or landmark features (see, e.g. [92]). As pointed out in Auto-Sklearn 2.0 [31], these meta-features are not defined with respect to certain column types such as categorical columns, and they are also expensive to compute, within limited budgets. The dataset embedding we adopt is builds individual column embeddings, and then pools these for a table level embedding. Similar to our approach, [26] use pretrained language models to get dataset embeddings based on available dataset textual information, e.g. title, description and keywords. Given these embeddings, their approach tries to find the most similar datasets and their associated baselines. Unlike [26], our approach relies on embedding the actual data inside the dataset and not just their overall textual description, which in many cases is not available. OBOE [114] uses the performance of a few inexpensive, informative models to compute features of a model.

Pipeline generation

There is a significant amount of work viewing the selection of learners as well as hyperparameters as a bayesian optimization problem like [99, 101]. Other systems have used evolutionary algorithms along with user defined templates or grammars for this purpose such as TPOT [66] or Recipe [96]. Still, others have viewed the problem of pipeline generation as a probabilistic matrix factorization [33], an AI planning problem when combined with a user specified grammar [63, 111], a bayesian optimization problem combined with Monte Carlo Tree Search [94], or an iterative alternating direction method of multipliers optimization (ADMM) problem [71]. Systems like VolcanoML focus on an efficient decomposition of the search space [69]. To the best of our knowledge, KGpip is the first system to cast the actual generation of pipelines as a neural graph generation problem.

Some recent AutoML systems have moved away from the fairly linear pipelines generated by most earlier systems to use ensembles or stacking extensively. H2O for instance uses fast random

search in combination with ensembling for the problem of generating pipelines [67]. Others rely on "stacking a bespoke set of models in a predefined order", where stacking and training is handled in a special manner to achieve strong performance [28]. Similarly, PIPER [77] uses a greedy best-first search algorithm to traverse the space of partial pipelines guided over a grammar that defines complex pipelines such as Directed Acyclic Graphs (DAGs). The pipelines produced by PIPER are more complex than the linear structures used in the current AutoML systems we use to test our ideas for historical pipeline modeling, and we do not use ensembling techniques yet in our approach. Neither is precluded, however, because KGpip meta-learning model can generate any type of structures, including complex structures that mined pipelines may have.

4.5 Chapter Summary

We proposed a novel formulation for the AutoML problem as a graph generation problem, where we can pose learner and pre-processing selection as a generation of different graphs representing ML pipelines. Hence, we developed the KGpip system based on mining large repositories of scripts, and leveraging recent techniques for static code analysis. KGpip utilized embeddings generated based on dataset contents to predict and optimize a set of ML pipelines based on the most similar seen datasets. KGpip is designed to work with AutoML systems, such as Auto-Sklearn and FLAML, to utilize their hyperparameter optimizers. We conducted the most comprehensive evaluation of 121 datasets, including the datasets used by FLAML, VolcanoML, and AL. Our comprehensive evaluation shows that KGpip significantly improves the performance of FLAML and Auto-Sklearn in classification and regression tasks. Moreover, KGpip outperformed AL, which is based on a more costly meta-learning process, in 97% of the datasets. This outstanding performance shows that the KGpip meta-learning approach is more effective and efficient. Finally, KGpip outperforms VolcanoML in 62% of the datasets and ties with it in 22%.

Chapter 5

Automated Data Exploration via RAG and AI Agents

Exploratory Data Analysis (EDA) is a foundational data science step for understanding datasets and uncovering patterns. Its significance is underscored by surveys indicating EDA constitutes a primary responsibility for 57% of data scientists [61] and consumes up to 40% of their time [4]. Consequently, automating EDA code generation is critical to accelerating data science workflows. However, this automation presents challenges distinct from general coding tasks. It requires not only reliably generating code for specific operations but, more critically, involves autonomously selecting appropriate EDA operations (e.g., chart types) and identifying the most relevant data columns for each operation directly from a raw dataset—navigating a vast combinatorial search space.

Current state-of-the-art approaches for automating EDA primarily leverage the capabilities of Large Language Models (LLMs) [23, 72]. While LLMs have shown promising results for general code generation tasks [59, 116, 49], their direct application to EDA automation reveals specific limitations. For instance, LLMs often perform well when generating EDA code for popular, well-studied public datasets, likely reflecting patterns seen in their vast training corpora. However, their accuracy (e.g., in selecting relevant EDA operations) and reliability (e.g., in generating consistently executable code) have been shown to significantly diminish when applied to private or enterprise datasets, which may feature unique structures or domain-specific conventions [64]. Furthermore, the

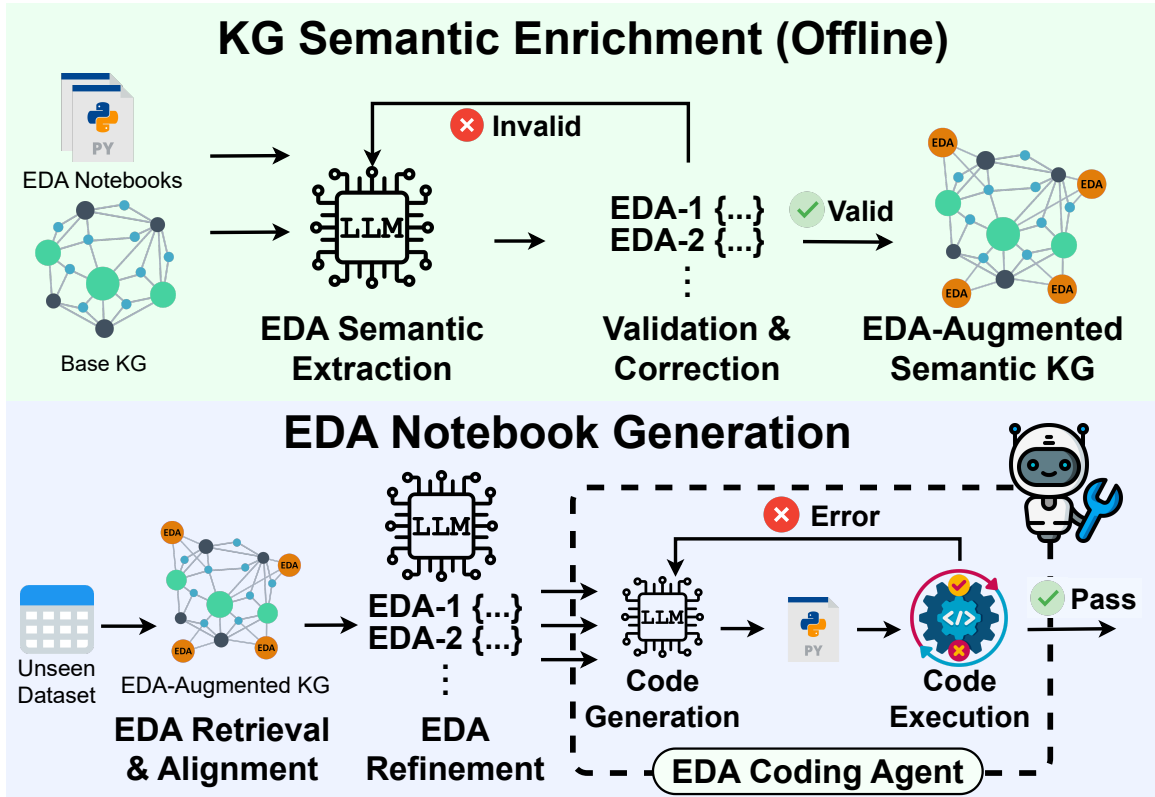


Figure 5.1: An overview of RAGvis’s KG semantic enrichment (Sec. 5.1) and EDA notebook generation (Sec. 5.2).

efficacy of these LLM-only methods is primarily tied to the use of large-scale, proprietary models such as Gemini 2.5 Pro [37] and GPT-4 [87]. However, such performance may not readily translate to smaller, open-source LLMs, which are often preferred in enterprise settings for privacy, control over local deployment, and cost. Therefore, the usability and scalability of LLM-only solutions is restricted in many practical scenarios.

To address these challenges, we introduce RAGvis, a novel Retrieval-Augmented Generation (RAG) framework designed for generating executable EDA code. RAGvis’ retrieval is based on a knowledge graph linking EDA operations code to their semantics and utilized data columns. Our approach operates in two stages as shown in Figure 5.1.

The first is offline **Knowledge Graph (KG) Semantic Enrichment**. Initially, a large corpus of Kaggle EDA notebooks is processed to generate a base KG capturing statement-level notebook semantics (e.g., code dependencies, library usage) and associated dataset column embeddings. This

base KG, however, lacks explicit EDA operations semantics (e.g., the specific chart type or columns an operation analyzes), a critical input for effective retrieval. To systematically define which semantics to collect, given the wide diversity of EDA operations, we first develop our structured EDA taxonomy (Figure 5.3). The core of the enrichment stage is then our novel KG-guided EDA semantic extraction: for each EDA operation in the notebooks, we provide an LLM with the operation’s code script contextualized by relevant parent statements derived from the base KG’s data flow dependencies. The LLM is then tasked to predict the operation’s semantic attributes according to our taxonomy. After validation and correction, these extracted EDA semantics are augmented to the base KG.

In the subsequent online **EDA notebook generation** stage, RAGvis takes an unseen dataset as input. First, it retrieves relevant EDA operations from the augmented KG and aligns their column specifications to the unseen dataset. Next, an LLM refines this set of candidate operations based on the unseen dataset’s characteristics. Finally, an EDA coding agent takes these refined operation specifications to generate and iteratively self-correct Python code through execution feedback, and assembles the verified snippets into a notebook. By leveraging the enriched KG to guide the selection of EDA operations tailored to the input dataset, and using an agent for robust code generation, RAGvis overcomes key limitations of relying solely on LLMs.

To validate our approach, we conducted extensive experiments on two benchmarks [20], comparing RAGvis to a state-of-the-art LLM-based baseline [23]. RAGvis demonstrates significant improvement on EDA retrieval accuracy (up to 0.35 higher recall), code reliability (near 100% pass rate), visual chart quality (LLM judge score), and cost (64% less LLM tokens consumed). Moreover, RAGvis enables small, open-source models such as Gemma 3 to achieve performance comparable to proprietary models such as GPT 4o-mini.

This work makes the following contributions:

- We introduce RAGvis¹, a novel RAG framework for EDA code generation, featuring: a) an empirically-driven structured taxonomy of EDA operations that categorizes visualizations by their semantic intent, b) a KG-guided semantic extraction method to guide an LLM in building an EDA-aware KG from notebooks, c) an online process that integrates EDA-specific

¹Code and datasets available at: <https://github.com/google/ragvis>

retrieval, alignment, and refinement to tailor EDA operations for unseen datasets, and d) a self-correcting EDA coding agent that iteratively generates and validates Python code to ensure reliability.

- Through comprehensive experiments on two benchmarks, we demonstrate that RAGvis significantly outperforms a state-of-the-art LLM-only baseline, achieving substantial improvements in semantic accuracy, code reliability, visual chart quality, and LLM token costs.
- We show that RAGvis empowers smaller, open-source models to achieve comparable performance to that of large, proprietary models, making automated EDA more feasible in low-resource and privacy-sensitive scenarios.

5.1 Knowledge Graph Semantic Enrichment

The retrieval in RAGvis relies on a Knowledge Graph (KG) that links EDA operations to their implementing code and the specific data columns they analyze. As such an EDA-specific KG is not available, we first utilize a base KG built from a large corpus of EDA notebooks and datasets. The base KG captures essential statement-level information about the code and its use of dataset columns but lacks EDA information. Therefore, we systematically augment it with EDA-specific semantics, yielding an EDA-aware KG crucial for retrieval.

5.1.1 Base Knowledge Graph Generation

To generate the base KG, we first collected a corpus comprising approximately 1600 EDA notebooks associated with 911 distinct datasets sourced from the Kaggle platform. To ensure quality, these notebooks were selected based on their high vote counts by the Kaggle community [78, 93].

To process this corpus into a structured KG, we built upon recent advances in building KGs from code repositories [104] and generating embeddings for tabular data [58]. We employed KGLiDS [47], a platform designed for generating KGs from data science pipelines and their associated datasets, integrating both statement-level code semantics and column embeddings. Nodes in the KG include code statements and dataset columns, while edges denote code dependencies and data

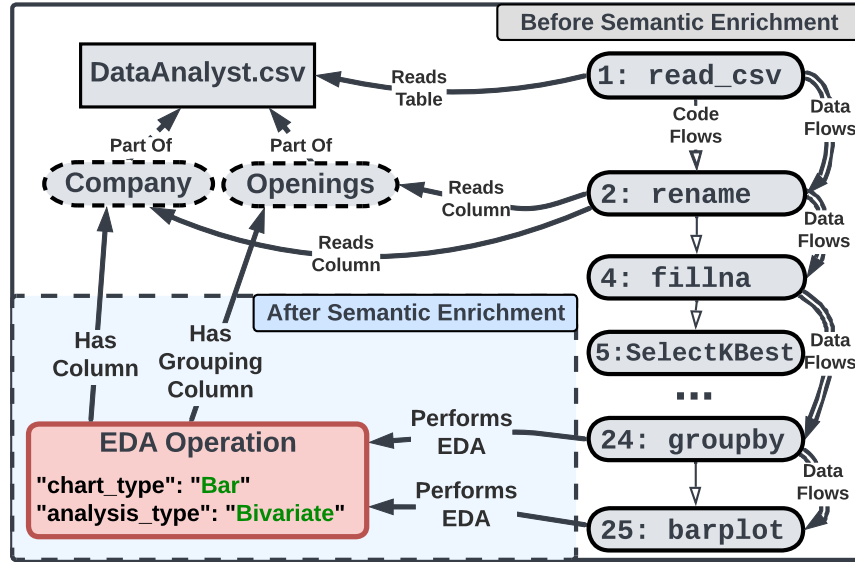


Figure 5.2: The base and enriched KG corresponding to the script in Figure 5.4.

reads by statements. For instance, considering the example in Figure 5.4, the base KG represents all code statements (lines 1-25) as code statement nodes. These are connected by code flow edges that capture both logical sequence (e.g., line 2 follows line 1) and data dependencies (e.g., the `df` variable from line 4 is used in line 24). Columns from the dataset, like "Openings" and "Company", are also represented as column nodes. These are linked to code statement nodes by column read edges whenever a statement uses them (e.g., line 2 is connected to the "Openings" and "Company" nodes). This base KG, however, does not identify EDA operations in a notebook and lacks explicit EDA-specific semantics required for our task such as the objective and type of an EDA operation. Figure 5.2 shows the base KG corresponding to the example in Figure 5.4 before and after enrichment.

The KGLiDS platform also analyzes datasets at the column level, generating embeddings based on the column name, raw value, and a concatenation of both. We utilize these embeddings for accurate EDA retrieval (Section 5.2.1).

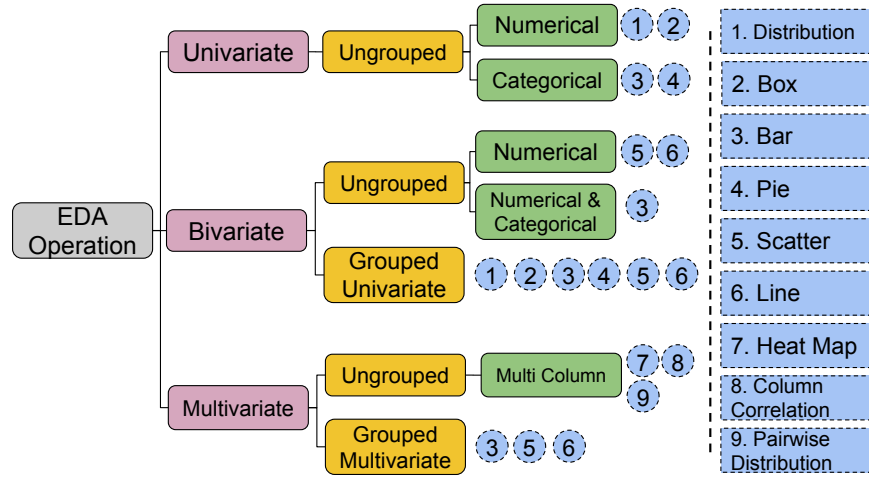


Figure 5.3: Our taxonomy of EDA operations based on a survey of EDA notebooks on the Kaggle platform.

5.1.2 Taxonomy of EDA Operations

To systematically enrich the KG with EDA-specific semantics, we first need a structured way to define and categorize the wide variety of EDA operations found in practice. Simply instructing a model to "extract EDA semantics" without a clear definition would lead to inconsistent and incomplete information. Therefore, we developed a comprehensive taxonomy of EDA operations derived from an extensive survey of notebooks on the Kaggle platform. While empirically derived, this taxonomy is designed to be broadly generalizable beyond Kaggle; its core structure (Figure 5.3) is based on fundamental analytical concepts, such as the number of variables analyzed, rather than specific implementation details like plotting libraries or cosmetic variations.

Our taxonomy is hierarchical, first categorizing operations by the number of primary columns involved: Univariate (one column), Bivariate (two columns), and Multivariate (three or more columns). Further distinctions consider grouping and analysis types (observed frequencies: Univariate 36.3%, Bivariate 44.9%, Multivariate 18.8%).

Univariate operations, in our taxonomy, are Ungrouped. They typically involve Distribution plots (1) or Box plots (2) for Numerical columns, and Bar charts (3) or Pie charts (4) for Categorical columns. **Bivariate** operations relate two variables. Ungrouped forms include Scatter plots (5) or Line charts (6) for two numerical columns, or Bar charts (3) for comparing a numerical column

```

1: df = pd.read_csv("DataAnalyst.csv")
2: df = df.rename(columns={"Openings":"Jobs",
3:                          "Company":"Employer"})
4: df["Jobs"] = df["Jobs"].fillna("missing")
5: feature_selector = SelectKBest(k=20)

...

24: data = df.groupby("Employer")["Jobs"].sum()
25: chart = sns.barplot(data,x="Employer",y="Jobs")
-----
{"chart_type": "Bar", "analysis_type": "Bivariate",
 "chart_columns": ["Openings"],
 "grouping_column": "Company"}

```

Figure 5.4: Top: EDA script, where a dataset is loaded, preprocessed (Lines 2-5), and used to visualize a Bar Chart showing the number of jobs per employer (Lines 24-25). Bottom: corresponding EDA semantics extracted using the taxonomy in Figure 5.3.

against a categorical one (e.g., 'Total Population' per 'City'). A key subtype is **Grouped Univariate** analysis. This involves performing a univariate analysis (using charts 1-6) on a value column for each segment defined by a *grouping column* (typically categorical). An example is grouping 'Employee' data by 'Department' and showing the average 'Income'. Notably, 23.6% of analyzed operations involved a grouping column, representing a common pattern.

Multivariate operations analyze three or more columns. Ungrouped examples include Heatmaps (7), Column Correlation (8) computations, and Pairwise Distributions (9). **Grouped Multivariate** operations use one column for grouping (e.g., for color or faceting) while visualizing relationships between two or more other columns (e.g., using faceted/colored Scatter (5) or Line (6) charts, or potentially grouped Bar charts (3)). This might explore how the relationships vary across data segments (e.g., average 'Income' vs. 'Happiness Index', colored by 'State').

It is important to note that functionally similar charts are grouped into categories; for instance, the "Box" category includes Box and Violin plots, while "Distribution" contains Histograms and KDE plots. Each level in the taxonomy corresponds to a semantic attribute of an EDA operation. Figure 5.4 shows a sample EDA operation and the corresponding EDA semantics per this taxonomy.

Algorithm 4: KG-Guided EDA Semantic Extraction

Input: EDA Notebook $\mathcal{N}$ consisting of n EDA blocks $\{b_1, \dots, b_n\}$,
Data Science Knowledge Graph $\mathcal{G}$, Large Language Model LLM

```
1 for  $b \in \mathcal{N}$ :
2   block_context = { }
3   for Statement:  $s \in b$ :
4     parents = {SELECT p FROM  $\mathcal{G}$  WHERE p  $\xrightarrow{\text{DataFlow}\{1...*\}}$  s}
5     block_context.add(parents)
6   block_context.sort()                                ▷ Sort by statement order
7   backtracked_block = block_context + b
8    $EDA_{Op_b} = LLM(\text{"Abstract EDA Code", backtracked\_block})$ 
9    $\mathcal{G} \leftarrow \mathcal{G} \cup EDA_{Op_b}$ 
10 return  $\mathcal{G}$ 
```

5.1.3 KG-Guided EDA Semantic Extraction

The base KG, while capturing statement-level details, initially lacks explicit EDA operation semantics, a critical input for effective retrieval. We enrich the KG by extracting the EDA-specific semantics from collected Kaggle notebooks. This extraction, as shown in Algorithm 4, occurs per EDA cell using an LLM with few-shot prompting, guided by our EDA taxonomy. The LLM prompt includes: 1) the target EDA code snippet (contextualized as detailed below); 2) a dataset sample; 3) expected semantic fields based on our taxonomy (as shown in Figure 5.4); and 4) few-shot (code snippet, dataset sample, EDA semantics) examples. The LLM then extracts chart columns, grouping column, chart type, and analysis type.

A key challenge is providing sufficient yet concise source code context for the LLM, as EDA notebooks often preprocess data (e.g., renaming "Openings" to "Jobs" in Figure 5.4, lines 1-5) before visualization (lines 24-25). Providing only the EDA cell (lines 24-25) to the LLM can lead to incorrect column identification (e.g., user-defined names like "Jobs" instead of the correct original name, "Openings"). In contrast, providing the full script as context is often too costly or might exceed the context size of some LLMs. Our solution is to provide the LLM with *relevant context* by querying the KG for data flow parent statements of the EDA cell. The data flow parents include all statements that define or modify data variables used within the EDA cell (e.g., lines 1-4 are parents for lines 24-25 in Figure 5.4). We prepend in order the data flow parents to the EDA

cell code within the LLM prompt. This KG-assisted contextualization ensures accurate semantic extraction with minimal added token consumption. Figure 5.4 (bottom) shows the extracted EDA semantics for the sample operation.

The format of the LLM-extracted semantics is validated by enforcing a JSON schema, while the semantic correctness is validated via heuristic rules (e.g. checking that column names exist in the original dataset); if an operation fails validation, the LLM is asked for regeneration, and operations remaining incorrect after set retries are discarded.

Algorithm 4 details the offline process for enriching a knowledge graph with structured information about EDA operations found in notebooks. The algorithm takes an EDA notebook ($\mathcal{N}$), the base Knowledge Graph ($\mathcal{G}$), and a Large Language Model (LLM) as input.

The process operates by iterating through each EDA block b within the notebook $\mathcal{N}$ (line 1). For each block, the goal is to provide the LLM with just enough context to understand the operation without providing the entire notebook script. To achieve this, the algorithm first initializes an empty `block_context` (line 2). It then traverses each statement s within the current EDA block b (line 3) and queries the knowledge graph $\mathcal{G}$ to find all data flow parent statements (parents) – that is, any preceding statements that defined or modified data used in s (line 4). These parent statements are collected into the `block_context` (line 5), which is then sorted to preserve the logical code order (line 6). This context is prepended to the original EDA block b to form a `backtracked_block` (line 7), which contains the visualization code and all relevant data preparation steps. This `backtracked_block` is then passed to the LLM in a few-shot prompt to extract the structured EDA semantics, which are returned as *EDA_Op* (line 8). Finally, this extracted information is used to augment the knowledge graph $\mathcal{G}$ by creating a new EDA operation node and linking it to the relevant code and column nodes (line 9). After all blocks in the notebook have been processed, the enriched graph $\mathcal{G}$ is returned (line 10).

5.1.4 Knowledge Graph Augmentation

To transform the base KG into an EDA-aware resource, we augment it with the extracted semantics. For each successfully analyzed EDA operation, we create a new EDA operation node and store its semantic attributes (e.g., `chart_type`, `analysis_type`) as node properties, as shown

at the bottom of Figure 5.4. This new node is then linked via edges to: 1) the code statement nodes that perform the analysis (lines 24-25), and 2) the nodes of the original dataset columns it analyzes ("Openings" and "Company"). This explicit linking between an EDA operation and its columns is vital, as it directly facilitates the targeted retrieval of EDA operations for a given column in the subsequent notebook generation phase.

5.2 EDA Notebook Generation

The objective of this stage is to generate a comprehensive EDA notebook given an unseen dataset.

5.2.1 Embedding-based EDA Retrieval

Our retrieval approach operates on the premise that similar datasets are often effectively analyzed using similar EDA operations. Retrieval in RAGvis (Algorithm 5, Lines 2-6) utilizes the EDA-augmented KG to fetch relevant EDA operations for an unseen dataset D_u . It does so by querying the KG for operations that were applied to columns in *seen* datasets similar to those in D_u . This utilizes the explicit links between EDA operations and data columns added in the KG enrichment.

However, measuring column similarity is challenging. Using column names alone is often insufficient as columns with identical names can have vastly different data types or value distributions. For example, a "rating" column could contain strings like "Good", integers on a 1-100 scale, or floats from 1-5. To address this, RAGvis utilizes column embeddings for similar column matching. These embeddings (generated in the base KG) are concatenations of column name embeddings and column raw value embeddings, resulting in accurate retrieval of EDA operations.

The retrieval process operates at the column granularity. For each column $C_u \in D_u$, we query the KG column embeddings using cosine similarity. The query targets seen columns that share C_u 's data type (e.g. our EDA-augmented KG recognizes 7 types: integers, floats, booleans, dates, natural language text, named entities, and generic strings). The set of top N most similar seen columns, $\mathcal{C}_{top-N}$, are retrieved, where N is a configurable parameter influencing recall. Subsequently, for

each retrieved seen column $C_s \in \mathcal{C}_{top-N}$, RAGvis queries the KG to fetch its associated EDA operations. Crucially, these operations are presented in the context of the *seen* datasets from which they were retrieved (i.e., referencing columns like C_s and its original neighbours). To render these operations directly executable on the unseen dataset D_u , a subsequent EDA Alignment step is necessary.

5.2.2 EDA Alignment

The goal of the alignment step is to find a suitable mapping for a retrieved EDA operation (applied on a seen dataset) to the unseen dataset D_u . EDA alignment in RAGvis (Lines 7-17 in Algorithm 5) uses a two-factor criterion: column data type compatibility and embedding similarity. For each seen column C_s used in an operation O , the system searches within the unseen dataset D_u for a potential target column C_{align} that has a matching data type with C_s , highest embedding similarity, and was not previously selected as an alignment target for the same operation (to ensure 1 : 1 mapping to D_u). Alignment for univariate operations applied to C_s is straightforward, mapping C_s directly to C_u . For bivariate and multivariate operations involving other columns than C_s , however, **all** participating columns from the seen dataset must also be aligned to suitable columns in the unseen dataset D_u using the same process. If, for a given operation O , a suitable alignment cannot be found for all seen columns to which O was historically applied, then O is deemed unalignable in the context of D_u and is discarded from the candidate set.

After this alignment process, all remaining EDA operations are now expressed in terms of columns within D_u and are considered directly applicable. This procedure, applied across all retrieved $\mathcal{C}_{top-N}$ columns for each C_u , may result in duplicate EDA operations being suggested. RAGvis regards the frequency of each unique operation as an estimation of the system confidence in recommending said operation. The frequency is utilized in the subsequent stage to inform the LLM about the relative importance of each EDA operation.

5.2.3 EDA Refinement

The EDA Refinement step leverages LLM reasoning to curate prior aligned operations. Its input includes aligned EDA operations, sorted by increasing frequency count (the LLM is explicitly

Algorithm 5: EDA Retrieval and Alignment

Input: Unseen Dataset: D_u , EDA-Augmented KG: $\mathcal{G}$
No. similar cols to retrieve: N

```
1  $EDA_{Aligned} = \{ \}$ 
2 for unseen column  $C_u \in D_u$ :
3    $C_{top-N} \leftarrow \{ \text{SELECT } C \text{ FROM } \mathcal{G} \text{ WHERE } C.type = C_u.type$ 
4     ORDER BY  $\mathbf{Emb}_{sim}(C, C_u)$  LIMIT  $N$   $\}$ 
5   for similar seen column  $C_s \in C_{top-N}$ :
6      $O_{C_s} \leftarrow \{ \text{SELECT EDA FROM } \mathcal{G} \text{ WHERE EDA} \leftrightarrow C_s \}$ 
7     for EDA operation  $O_{\{C_s\}}$   $\in O_{C_s}$ :
8        $aligned\_D_u\_cols \leftarrow \{ \}$ 
9       for seen column  $C_{seen}$  used in  $O_{\{C_s\}}$ :
10         $C_{align} \leftarrow \{ \text{SELECT } C \text{ FROM } D_u$ 
11          WHERE  $C.type = C_{seen}.type$ 
12          AND  $C \text{ NOT IN } aligned\_D_u\_cols$ 
13          ORDER BY  $\mathbf{Emb}_{sim}(C, C_{seen})$  LIMIT 1  $\}$ 
14         $aligned\_D_u\_cols.add(C_{align})$ 
15      if all columns in  $O_{\{C_s\}}$  are aligned:
16         $EDA_{Aligned}.add(O(aligned\_D_u\_cols))$ 
```

17 **return** Counter($EDA_{Aligned}$)

informed of this order), alongside key semantic details of the unseen dataset (column names, data types, and basic statistics). Using a one-shot prompt, the LLM is instructed to select k most suitable EDA operations, considering both the provided candidates and the dataset semantics, with the possibility of suggesting novel operations not present in the input list. To ensure reliable output structure, we utilize the LLM’s constrained generation (e.g., JSON mode), using our EDA taxonomy as a schema. This yields k refined EDA operations in JSON format, aligned to D_u and ready for code generation; k is user-defined. This refinement step is optional, particularly for smaller models with limited reasoning ability. If disabled, the top k operations ranked by frequency from the alignment step are passed directly to the EDA coding agent.

Algorithm 5 specifies the online procedure for retrieving and adapting relevant EDA operations from the knowledge graph for an unseen dataset. The algorithm’s inputs are the unseen dataset (D_u), the EDA-augmented knowledge graph ($\mathcal{G}$), and a parameter N defining the number of similar columns to retrieve for the search.

The process begins by initializing an empty list, $EDA_{Aligned}$, to store the final results (line 1).

The algorithm then enters its main loop, iterating through each column C_u in the unseen dataset D_u (line 2). For each of these columns, it performs a similarity search against the knowledge graph to find the top N most similar columns from the seen datasets, constrained by matching data types and ordered by embedding similarity (line 3). This retrieval step yields a set of similar seen columns, C_{top-N} .

The algorithm then proceeds to the alignment phase by iterating through each retrieved seen column C_s in C_{top-N} (line 4) and fetching all EDA operations historically associated with it from the graph (line 5). For each of these retrieved operations (line 6), the algorithm attempts to align it to the unseen dataset. It iterates through every source column (C_{seen}) that was part of the original operation (line 9) and searches for the best possible target column (C_{align}) within the unseen dataset D_u (lines 10-13). This mapping is determined by finding an unassigned column in D_u that has both a matching data type and the highest embedding similarity to C_{seen} . If a suitable alignment is found for every column involved in the original operation, the newly mapped operation is added to the $EDA_{Aligned}$ list (lines 15-16); otherwise, the operation is discarded.

After all columns in the unseen dataset and all their corresponding retrieved operations have been processed, the algorithm returns a frequency count (Counter) of all the successfully aligned EDA operations in the $EDA_{Aligned}$ list (line 17). This count serves as a confidence score, indicating which EDA operations were most frequently suggested through the retrieval and alignment process.

5.2.4 EDA Coding Agent

The EDA Coding Agent generates verified Python snippets for the k refined EDA operations applicable to the unseen dataset D_u . For each input operation (provided as JSON specification) paired with D_u 's summary, an LLM generates a corresponding Python code snippet designed to produce exactly one data visualization (a single chart). The agent then attempts to execute this generated snippet in a local environment containing D_u to verify it runs without compile-time or runtime errors. If execution fails, the agent captures the resulting error message and feeds it back into the original code generation session, prompting the LLM to regenerate the code specifically addressing the reported error. This initiates an automated, iterative debugging loop of generation, execution, and correction which continues until the code runs successfully or a predefined maximum

Table 5.1: A comparison between the utilized benchmarks. While VisEval has more tables, KaggleVisBench has higher variety and number of EDA operations. Median values shown between brackets.

	VisEval	KaggleVisBench
# Tables	239	50
Avg. # EDA Ops.	2.9 (2)	20.0 (16)
Supported Charts	Bar, Pie, Scatter, Line	Bar, Pie, Scatter, Line, Box, ColumnCorr., Histogram, HeatMap, PairwisePlot
Annotation	Manually-labelled	Automatically-labelled
Total Size (MB)	4.2	1,547.3
Avg. # Rows	291.6 (12)	16,514.9 (4,500)
Avg. # Columns	6.2 (6)	13.2 (12)

number of retries is reached. Any operation whose code cannot be fixed within the retry limit is ultimately excluded. Therefore, the output of this stage is a set of up to k Python code snippets, each successfully executed and verified error-free on D_u , corresponding to one of the refined EDA operations. These snippets can then be assembled into a single executable file (e.g. a Jupyter Notebook) using an LLM prompt to deduplicate redundant setup code.

5.3 Experimental Evaluation

We conduct a comprehensive evaluation between RAGvis and an LLM-based system on two independent data visualization benchmarks.

5.3.1 Benchmarks

We utilized two benchmarks in our evaluation:

KaggleVisBench: We collected a high-quality benchmark from 132 EDA notebooks associated with 50 Kaggle datasets. The notebooks are the most voted among Kaggle users for each dataset, with at least 20 votes. We extracted the set of EDA operations per dataset using the EDA semantic extraction described in Section 5.1.3. To ensure a fair evaluation and prevent information leakage, we verified that none of the 50 datasets in this benchmark were present in the corpus of notebooks

used to build our knowledge graph.

The benchmark contains an average of 20 EDA operations per dataset, totalling 999 operations. We make KaggleVisBench available for the research community.²

VisEval: To assess our system’s ability to generalize to EDA patterns beyond the Kaggle ecosystem, we also evaluate on VisEval [20]; This is a natural language to visualization (NL2VIS) benchmark consisting of high-quality datasets manually annotated by human experts, providing an out-of-domain test for our approach.

Original samples in VisEval consist of a visualization query for a dataset and the corresponding chart metadata, including chart type (Bar, Pie, etc.), chart columns, and query difficulty. We grouped the samples by datasets and considered the set of charts for each dataset as ground truth. In addition, we filter out the 462 multi-table operations as they are out of the scope of this study. This results in an average of 2.9 EDA operations per dataset, totalling 688 operations for 239 datasets.

5.3.2 Baselines

LIDA: We consider LIDA [23] as our primary baseline. LIDA is a multi-step prompt-based automated data visualization framework. LIDA takes a raw dataset as input and i) summarizes the dataset information, ii) explores k potential EDA operations specifications, iii) generates the corresponding Python code, and iv) executes each visualization code to generate k visualizations. Here, k is a parameter to control the number of charts. Steps i through iii are carried out using LLM prompting.

Data2Vis: To contextualize the performance of modern LLM-based approaches, we also evaluate against Data2Vis [74], a traditional, non-LLM AutoVis system. Data2Vis utilizes a sequence-to-sequence model to generate visualizations from raw data, outputting them in the Vega-Lite declarative specification language [97]. A full comparison is not feasible, as it does not generate source code. To enable a partial comparison, we mapped its outputs to our EDA taxonomy to compute semantic accuracy using the Recall@ k metric.

²Available at: <https://github.com/google/ragvis>

5.3.3 Evaluation Metrics

Recall@k: There is no standard metric to measure the “goodness” of a generated chart as data visualization is inherently subjective. In this work, we developed the EDA taxonomy described in Section 5.1.2. We utilize the taxonomy to measure the retrieval accuracy of the generated charts. To the best of our knowledge, our work is the first to provide an automated accuracy metric for data visualization. Recall is measured by matching the semantic attributes in our taxonomy shown in Figure 5.4. A true positive sample for a particular dataset is a predicted EDA operation that exists in the set of operations in the ground truth. We use exact matching for all attributes of the taxonomy; if, for instance, a correct chart type (e.g. Bar) is predicted but applied on or grouped by incorrect set of columns, the prediction is regarded as a false positive. The recall is averaged for all datasets in a benchmark. Moreover, we calculate Recall@ k , where k is the desired number of EDA operations. We selected practical values of k between 5-30 resembling real-world EDA notebooks of varying depths.

Pass Rate@k: Similar to previous works [20, 23], we measure source code reliability using pass rate, which is the percentage of generated EDA operations with source code executing without compile-time or run-time errors. We calculate Pass Rate@ k at varying k values similar to Recall@ k .

VLM Score: To assess the visual quality of the generated charts, we use a Vision Language Model (VLM) as a judge, following the same procedure reported by the authors of VisEval [20]. This automated metric, was shown to highly correlate with human expert ratings, providing a practical alternative to a user study. The score consists of two components: **Readability** evaluates the aesthetic and functional quality of a chart, including its layout, scales, ticks, and the clarity of titles and labels. The **Quality Score** then adjusts the readability score by penalizing any charts that are invalid or contain errors. We used Gemini 2.5 Pro as the judge given its advanced reasoning capabilities.

LLM Tokens: We calculate the number of LLM tokens used in the generation stage of both systems.

5.3.4 EDA Retrieval Accuracy

Table 5.2 shows the average Recall@ k of RAGvis compared to LIDA on both benchmarks. RAGvis significantly outperforms LIDA for all values of k and LLMs with a difference in Recall@30 reaching up to 0.33 on VisEval and 0.29 on KaggleVisBench. This demonstrates the effectiveness of our EDA retrieval and refinement compared to LLM prompting used in LIDA. By grounding the EDA selection process in the knowledge graph, RAGvis effectively navigates the large search space of the auto EDA problem. Prompt-based techniques like LIDA solely depend on LLM reasoning for EDA selection, which is not as effective for this task even with thinking models such as Gemini 2.5 Pro.

Furthermore, our comparison against Data2Vis highlights the significant performance gap between prior methods and LLM-based approaches for this task. As shown in Table 5.2, RAGvis substantially outperforms Data2Vis on the VisEval benchmark.

5.3.5 EDA Code Reliability

Table 5.3 shows the pass rates of RAGvis and LIDA on both benchmarks for different values of k and LLMs. RAGvis maintains an impressive $\sim 100\%$ pass rate across all values of k with Gemini 2.5 Pro and GPT 4o-mini. This illustrates the effectiveness of our EDA coding agent, which executes the generated source code and performs self correction if needed. While LIDA requires strong coding models such as Gemini 2.5 Pro to achieve acceptable pass rate of 95%+, RAGvis achieves competitive pass rates even with Gemma 3 12b.

5.3.6 EDA Visual Quality

To evaluate the visual quality of the generated charts, we conducted an analysis using the VLM Score metric on the VisEval benchmark with $k = 15$. We ran both RAGvis and LIDA using GPT-4o-mini and Gemma 3 12b, with Gemini 2.5 Pro as the VLM judge. The results, summarized in Table 5.4, show that RAGvis consistently achieves higher Readability and Quality scores. This indicates that RAGvis generates charts that are not only semantically correct but also more readable and aesthetically pleasing. A paired t-test confirms the results are statistically significant ($p < 5e-10$).

Table 5.2: Recall@k of RAGvis and LIDA on two benchmarks with open source and commercial LLMs.

	VisEval			KaggleVisBench		
Recall	@5	@15	@30	@5	@15	@30
Gemini 2.5 Pro (25/03)						
RAGvis	0.5	0.67	0.77	0.12	0.29	0.43
LIDA	0.15	0.36	0.49	0.01	0.06	0.14
Gemini 2.0 Flash						
RAGvis	0.46	0.66	0.72	0.12	0.23	0.34
LIDA	0.16	0.44	0.48	0.03	0.11	0.17
Claude Sonnet 3.7						
RAGvis	0.43	0.66	0.74	0.1	0.27	0.43
LIDA	0.12	0.28	0.38	0.02	0.06	0.16
GPT-4o mini						
RAGvis	0.33	0.59	0.68	0.07	0.18	0.3
LIDA	0.24	0.40	0.52	0.06	0.12	0.18
Gemma 3 12b						
RAGvis	0.34	0.63	0.7	0.07	0.18	0.22
LIDA	0.14	0.28	0.43	0.02	0.09	0.13
Gemma 3 4b						
RAGvis	0.22	0.46	0.61	0.05	0.1	0.15
LIDA	0.03	0.15	0.28	0.01	0.06	0.11
Data2Vis	0.02	0.5	0.12	-	-	-

5.3.7 Open vs. Commercial LLMs

A primary goal of our work is the enablement of effective and reliable EDA code generation with small, open-source models. While RAGvis is able to utilize the advanced reasoning of Gemini 2.5 Pro as shown in Tables 5.2 and 5.3, it manages to achieve comparable performance with Gemma 3 12b to GPT 4o-mini and a competitive performance with the small Gemma 3 4b. LIDA, on the other hand, is entirely dependent for its performance on the LLM reasoning capability, showing a significant drop in accuracy and pass rates with the Gemma models. This advantage also extends

Table 5.3: Pass Rate@k of RAGvis and LIDA on two benchmarks with open source and commercial LLMs.

	VisEval			KaggleVisBench		
Pass Rate	@5	@15	@30	@5	@15	@30
Gemini 2.5 Pro (25/03)						
RAGvis	99.9%	100%	100%	99.2%	99.9%	99.5%
LIDA	95.6%	90.3%	96.3%	89.2%	92.4%	93.5%
Gemini 2.0 Flash						
RAGvis	100%	99.9%	100%	100%	100%	99.8%
LIDA	92.5%	90.7%	91.0%	77.6%	81.6%	84.2%
Claude Sonnet 3.7						
RAGvis	100%	100%	100%	100%	100%	100%
LIDA	90.3%	91.6%	91.8%	78.0%	82.5%	88.0%
GPT-4o mini						
RAGvis	99.6%	100%	99.8%	99.8%	99.2%	99.1%
LIDA	93.2%	88.4%	89.3%	84.4%	81.6%	72.3%
Gemma 3 12b						
RAGvis	98.5%	97.7%	95.0%	96.8%	96.3%	95.7%
LIDA	88.9%	88.7%	91.1%	78.4%	82.9%	66.5%
Gemma 3 4b						
RAGvis	98.3%	98.6%	98.2%	92.0%	89.2%	87.3%
LIDA	86.6%	93.3%	90.3%	79.2%	77.9%	68.3%

to visual quality as shown in Table 5.4. RAGvis with Gemma 3 12b produces higher-quality visualizations than LIDA using the proprietary GPT-4o-mini. These results highlight the practicality of RAGvis in settings where open-source models are preferred.

5.3.8 LLM Token Usage

To assess the cost effectiveness of our approach, we compared the number of LLM tokens utilized by RAGvis and LIDA on VisEval. Table 5.5 shows the input, output, and thought tokens used. RAGvis significantly reduces token consumption, using 56%-64% less input tokens and 59%-68%

Table 5.4: Readability and Quality scores of RAGvis and LIDA with $k = 15$ using a VLM as a judge on VisEval.

Model	Method	Readability	Quality
GPT 4o-mini	RAGvis	2.93	2.93
	LIDA	2.55	2.34
Gemma 3 12b	RAGvis	2.72	2.72
	LIDA	2.42	2.22

Table 5.5: A comparison with LIDA in terms of number of tokens incurred to evaluate both systems on VisEval.

Model	Gemini 2.5 Pro		GPT 4o-mini	
Method	RAGvis	LIDA	RAGvis	LIDA
Input Tokens	14.46 M	33.13 M	10.43 M	30.19 M
Output Tokens	7.38 M	18.08 M	2.36 M	7.53 M
Thought Tokens	30.97 M	24.3 M	-	-

less output tokens. This efficiency stems largely from RAGvis’s RAG architecture offloading data summarization and EDA retrieval tasks to non-LLM components. Moreover, RAGvis avoids the high token overhead seen in LIDA, which often relies on long, detailed system prompts (increasing input tokens) and requires the generation of verbose rationale with each EDA specification (increasing output tokens). Our approach achieves competitive results without these token-intensive steps.

5.3.9 Ablation Studies

We perform ablation studies to analyze the impact of key design choices within RAGvis, focusing on the LLM refinement, EDA coding agent, and EDA retrieval strategies.

5.3.9.1 Impact of LLM Refinement

We first quantify the LLM refinement impact on RAGvis’s recall. Figure 5.5 shows three settings: full RAGvis pipeline with refinement of Gemini 2.5 Pro (Gemini-Refined), GPT 4o-mini (GPT-Refined), and the retrieved and aligned EDA operations without applying any LLM refinement (Retrieved). Crucially, the figure reveals the competitive recall achieved with retrieval only,

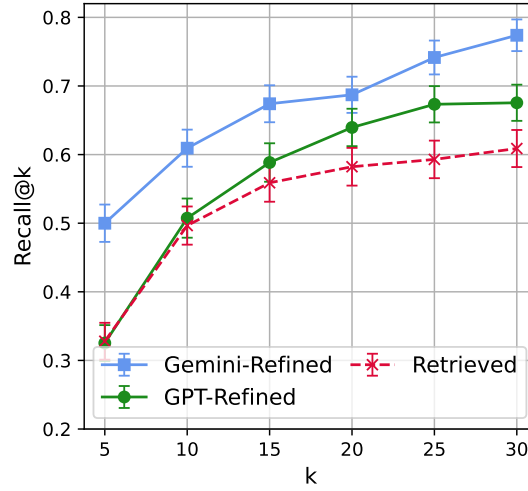


Figure 5.5: Impact of LLM refinement on Recall@k of RAGvis on the VisEval benchmark.

validating the effectiveness of our retrieval and alignment processes. Further, RAGvis takes advantage of the LLM-reasoning capabilities showing higher recall gains with Gemini 2.5 Pro’s advanced reasoning compared to GPT 4o.

5.3.9.2 Impact of EDA Coding Agent

To assess the contribution of the EDA coding agent, we conducted an ablation study where this component was replaced with a single LLM prompt without any execution feedback. The results, depicted in Figure 5.6, demonstrate the agent’s significant role in enhancing the pass rate of generated code. Notably, the EDA coding agent provides a considerable boost in pass rates for open-sourced models reaching approximately 6%. While strong coding models such as Gemini 2.5 Pro also benefit from the agent, the magnitude of this improvement is comparatively smaller (1.7%) than that observed for other models.

5.3.9.3 Impact of EDA Retrieval Methods

This study investigates how different strategies for retrieving similar columns—which subsequently determines the retrieved EDA operations—affect performance. We compare four settings:

- i) **LLM-Refined:** The full RAGvis pipeline output, using Gemini 2.5 Pro for the refinement step,
- ii) **Content-Label Retrieval:** Uses the direct output after retrieving and aligning EDA operations

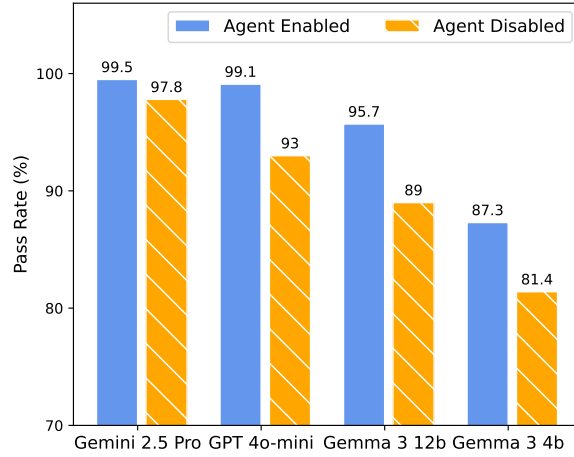


Figure 5.6: Impact of enabling the EDA coding agent on the PassRate@k of RAGvis on the VisEval benchmark.

based on column similarity calculated using both column name and raw values embeddings. This is the default retrieval strategy in RAGvis, presented here without LLM refinement, iii) **Content Retrieval**: Uses only raw value embeddings for column similarity (no LLM refinement), and iv) **Label Retrieval**: Uses only column name embeddings for column similarity (no LLM refinement).

Observing the results for the retrieval-only methods in Figure 5.7 (left). Comparing the three retrieval strategies on VisEval, using only Label embeddings performs slightly worse than using Content or Content-Label embeddings, indicating that cell values are better predictors of column similarity in the benchmark. Notably, our default Content-Label approach performs consistently well across both datasets among the retrieval-only methods.

5.3.9.4 Number of Retrieved Similar Columns

We also analyzed the sensitivity of RAGvis’s Recall to N , the number of similar columns retrieved per target column, testing values from 2 to 32 while fixing the retrieval method to Content-Label embeddings. Although larger N retrieves more initial EDA operation candidates, we select the final top-k based on frequency to have a fair comparison between the different settings, meaning N primarily impacts the confidence and ranking of the suggested EDA operations. Figure 5.7 (right) illustrates the results. On VisEval, Recall@k clearly benefits from larger N , plateauing around $N=8-32$. These results suggest $N=8$, used in other experiments, is a suitable default.

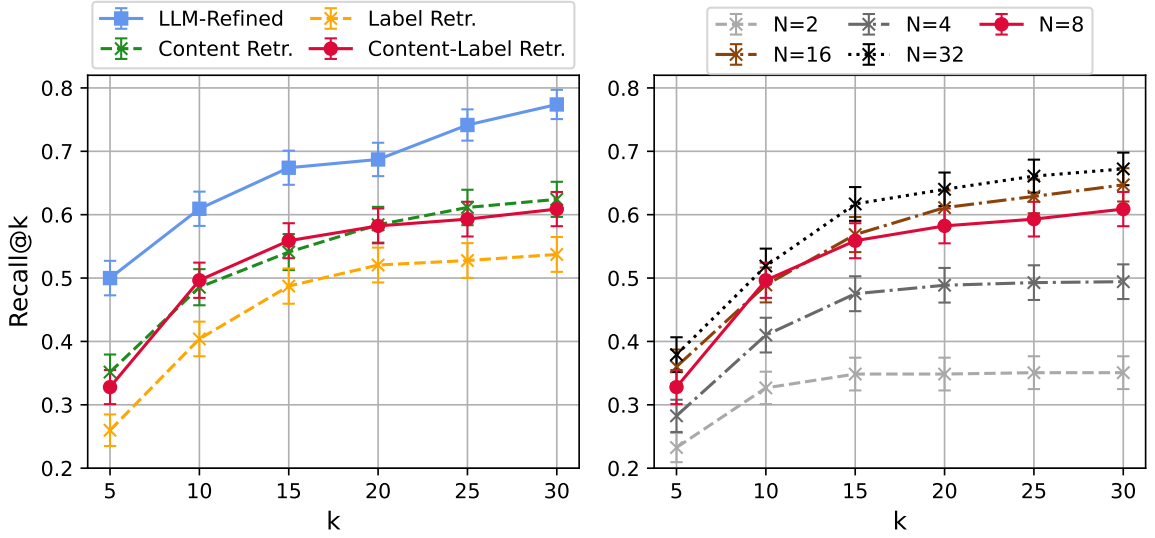


Figure 5.7: Ablation studies on VisEval [20] with different embedding retrieval methods (left) and different numbers of similar columns to retrieve, N (right).

5.4 Related Work

Automated Data Visualization (AutoVis) focuses on automatically generating appropriate data visualizations (charts) directly from raw datasets. Traditional AutoVis systems have explored various techniques, including reinforcement learning [9], sequence-to-sequence models [24], and heuristic-based recommendations [112, 73]. However, these approaches often face limitations: 1) their reliance on potentially brittle or incomplete heuristics can restrict the quality and scope of generated visualizations; 2) they typically support only a predefined, limited set of EDA operations; and 3) their output is typically an intermediate declarative specification (e.g., Vega-Lite [97]) rather than directly executable analysis code. Recently, the proficiency of LLMs in code generation tasks has inspired new methods for data visualization [72, 56]. For example, LIDA [23] leverages a multi-stage LLM-based approach including data summarization, goal (chart) exploration, and code generation. However, purely LLM-based approaches introduce their own challenges: 1) they typically lack explainability regarding why specific visualizations are recommended, and 2) they can suffer from hallucinations, potentially generating irrelevant charts. RAGvis employs a hybrid RAG approach to mitigate limitations of both paradigms: grounding recommendations in a knowledge

graph enhances accuracy and provides implicit explainability, while LLM capabilities enable flexible EDA refinement and code generation.

Natural Language to Visualization (NL2Vis) systems generate visualizations from a dataset guided by a natural language query specifying the desired analysis or target columns [50, 74, 115, 118], often using LLMs for interpretation and generation. This task is distinct from and largely orthogonal to ours. While NL2Vis receives the analytical specification via the query, a core challenge for RAGvis is the automatic selection and prioritization of relevant EDA operations (including charts and columns) directly from the raw dataset itself.

5.5 Chapter Summary

We introduced RAGvis, a novel RAG approach designed to automatically generate comprehensive and executable EDA notebooks directly from raw datasets. RAGvis utilizes an EDA-augmented knowledge graph built offline from real-world EDA notebooks. Given a new dataset, RAGvis performs novel EDA retrieval and alignment, LLM-based refinement, and generates the corresponding Python code via a self-correcting agent. Our comprehensive evaluation demonstrates that RAGvis significantly outperforms current LLM-based baselines with up to 0.35 higher recall, near 100% pass rate, and 64% less LLM tokens. Notably, our approach allows smaller, open-source models to achieve highly competitive results, highlighting the effectiveness and practical benefits of combining structured knowledge retrieval with LLM generation for automating complex data science tasks.

Chapter 6

Future Work Directions

This thesis lays a foundation for automating data science workflows by integrating knowledge graphs and table representation learning. The developed frameworks can be extended in several promising directions to further broaden their impact and capabilities.

6.1 Fully Agentic Systems

A promising direction is the investigation of new paradigms for utilizing the knowledge graph and table representation learning. While this thesis demonstrates an agent-assisted component for EDA, the current trend towards fully agentic and autonomous systems presents an important research frontier. Future work could explore the development of autonomous agents that leverage the knowledge graph as their primary world model for performing complex, multi-step data science tasks with minimal human intervention.

An agent could be tasked with a high-level goal, such as "build the best possible classification model for this new dataset." It could then use the knowledge graph to autonomously discover relevant datasets for feature enrichment, select promising pipelines based on similar past experiments, execute the code, evaluate the model, and even document its findings. This would represent a significant step beyond the current automated workflows, moving towards a system that can reason, plan, and act to solve end-to-end data science problems.

6.2 Diverse Data Modalities

Another significant extension is the incorporation of diverse data modalities into the knowledge graph. The current work focuses exclusively on structured, tabular data, yet real-world data lakes are rich with unstructured and semi-structured assets. Future work could aim to create a truly holistic data lake representation by integrating text documents, images, audio, video, and time-series data. This is a substantial research direction that would require careful consideration of both the knowledge graph ontology and the representation learning techniques employed.

Successfully integrating these sources would involve developing novel, modality-specific profiling techniques to extract relevant semantics and link them back to the tabular world. For example, this might involve using object recognition to link entities within an image to product tables, or using speech-to-text to link keywords from an audio file to customer feedback datasets. A unified, multi-modal Data Global Schema would empower data scientists to perform powerful cross-modal queries that are currently infeasible. For instance, a user could query the knowledge graph for all tables related to the products depicted in a set of images. This would transform the data global schema into a much more powerful and comprehensive tool for discovery and automation.

6.3 Integration of Broader Data Science Context

The current knowledge graph ontology is centered on data and code artifacts. A valuable extension would be to enrich this ontology with entities from the broader context of data science. This would involve adding new entity types to represent actors and activities, such as the data scientist, team, lab, experiment, model, and benchmark.

By linking these new entities to the existing data and code artifacts, the knowledge graph could capture a much richer picture of how data science is practiced. For example, it could model which teams have expertise with certain types of datasets, which experiments led to the development of a specific model, or how different models perform on standard benchmarks. Such a graph would move beyond simple data discovery to enable more complex queries related to expertise finding, project management, and the meta-analysis of research activities within an organization or a public platform like Kaggle.

6.4 Temporal Knowledge Graph for Data and Model Lineage

The static nature of the current Data Global Schema presents a compelling opportunity to explore the creation of a temporal knowledge graph for data and model lineage. Data lakes are highly dynamic environments where datasets are constantly added, schemas evolve, and data distributions drift over time. This research direction would focus on developing the foundations to manage this evolution for the entire data science lifecycle. A key area to investigate is the development of methods to efficiently track the full provenance of a machine learning model, from the specific version of the raw data used, through every transformation and feature engineering step, to the final trained artifact.

This work would involve investigating techniques for versioning all entities within the graph (datasets, code, and models) and linking them through explicit lineage edges. The resulting temporal knowledge graph would unlock powerful new capabilities essential for MLOps and responsible AI. It would enable full data and model provenance tracking, allowing users to understand how a model was created and reproduce its results. It would also support time-travel queries, giving data scientists the ability to query the state of the entire data science ecosystem at a specific point in the past.

Chapter 7

Conclusion

This thesis confronted a fundamental problem in modern data science: the persistent disconnection between datasets and code, which leads to siloed knowledge, redundant effort, and ad-hoc workflows. The central premise of this work was that by constructing a holistic, semantic representation of these artifacts, we could create a more intelligent, integrated, and automated data science ecosystem. To this end, this research proposed and validated a novel framework built upon two complementary technologies: Knowledge Graphs (KGs) to model the explicit, symbolic relationships between artifacts, and Column Learned Representations (CoLR) to capture the implicit, semantic similarity of their content. The primary contribution of this work was the creation of this representation as well as the demonstration of its power and versatility by exploring three distinct paradigms, each addressing a critical stage of the data science lifecycle.

The first paradigm addressed the foundational task of data discovery by leveraging direct, semantic queries against the knowledge graph. This work introduced a scalable data profiling pipeline that uses fine-grained data types and specialized CoLR models to produce high-quality embeddings. These profiles were used to construct a Data Global Schema, a comprehensive knowledge graph of the data lake. The experimental evaluation demonstrated that this approach significantly outperforms state-of-the-art systems, achieving superior accuracy and scalability by transforming data discovery from a simple keyword search into an expressive, semantic query process. This work successfully met the first research objective by proving that a KG and CoLR-based representation is a superior foundation for data discovery.

The second paradigm tackled the core modeling stage by introducing a data-driven, meta-learning approach to Automated Machine Learning (AutoML). This research demonstrated how the entire knowledge graph could serve as a training ground for Graph Neural Networks (GNNs) to learn from the collective experience encoded in thousands of prior data science projects. The developed system, KGpip, successfully learned the latent relationships between dataset characteristics and optimal pipeline architectures. By integrating these meta-learned recommendations into existing AutoML frameworks, this work showed significant performance improvements, thereby achieving the second research objective and validating the use of deep graph learning on the semantic representation to automate ML.

The third paradigm addressed the crucial exploratory stage by creating a more reliable and cost-effective system for Automated Exploratory Data Analysis (AutoEDA). This research introduced a Retrieval-Augmented Generation (RAG) framework that uses the knowledge graph as a factual, external memory to ground Large Language Models (LLMs). By providing LLMs with verifiable, structured context retrieved from the KG, the RAGvis system was able to generate high-quality, executable EDA code with greater reliability and lower cost than LLM-only approaches. This work achieved the third research objective, demonstrating that the semantic representation is a critical component for making modern AI tools safer, more accurate, and more accessible for data science tasks.

Ultimately, this thesis validates a powerful and versatile architectural pattern for the next generation of data science automation tools. The successful application of direct querying, deep graph learning, and retrieval-augmented generation from a single, unified semantic foundation demonstrates that this approach is not just a collection of separate solutions, but a cohesive and extensible framework. The findings presented herein provide a robust blueprint for building more collaborative, efficient, and intelligent data science ecosystems. The promising directions for future research, such as the development of fully agentic systems and the integration of diverse data modalities, will only build upon this strong foundation.

References

- [1] Ibrahim Abdelaziz et al. “A Toolkit for Generating Code Knowledge Graphs”. In: *Proceedings of Knowledge Capture Conference (K-CAP)* (2021). URL: <https://doi.org/10.1145/3460210.3493578>.
- [2] Ziawasch Abedjan et al. *Data Profiling*. Synthesis Lectures on Data Management. 2018. DOI: [10.2200/S00878ED1V01Y201810DTM052](https://doi.org/10.2200/S00878ED1V01Y201810DTM052).
- [3] Miltiadis Allamanis et al. “A Survey of Machine Learning for Big Code and Naturalness”. In: *ACM Comput. Surv.* 51.4 (2018). ISSN: 0360-0300. DOI: [10.1145/3212695](https://doi.org/10.1145/3212695). URL: <https://doi.org/10.1145/3212695>.
- [4] Anaconda. *Anaconda 2022 State of Data Science*. 2022. URL: <https://www.anaconda.com/resources/whitepapers/state-of-data-science-report-2022>.
- [5] *Artificial Analysis: AI Adoption Survey - H1 2025 - Highlights Report*. 2025. URL: <https://artificialanalysis.ai/downloads/ai-adoption-survey/2025/Artificial-Analysis-AI-Adoption-Survey-H1-2025.pdf>.
- [6] Mattia Atzeni and Maurizio Atzori. “CodeOntology: RDF-ization of Source Code”. In: *Proceedings of International Semantic Web Conference, (ISWC)*. Oct. 2017, pp. 20–28. ISBN: 978-3-319-68203-7. URL: https://doi.org/10.1007/978-3-319-68204-4_2.

- [7] Sören Auer et al. “DBpedia: A Nucleus for a Web of Open Data”. In: *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*. ISWC’07/ASWC’07. Busan, Korea: Springer-Verlag, 2007, pp. 722–735. ISBN: 3540762973.
- [8] Gilbert Badaro, Mohammed Saeed, and Paolo Papotti. “Transformers for Tabular Data Representation: A Survey of Models and Applications”. In: *Transactions of the Association for Computational Linguistics* 11 (2023), pp. 227–249. ISSN: 2307-387X. URL: https://doi.org/10.1162/tac1%5C_a%5C_00544.
- [9] Ori Bar El, Tova Milo, and Amit Somech. “Automatically Generating Data Exploration Sessions Using Deep Reinforcement Learning”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Association for Computing Machinery, 2020, pp. 1527–1537. ISBN: 9781450367356. DOI: [10.1145/3318464.3389779](https://doi.org/10.1145/3318464.3389779). URL: <https://doi.org/10.1145/3318464.3389779>.
- [10] Jan Henrik Bertrand et al. *Autoencoder-based General Purpose Representation Learning for Customer Embedding*. 2024. arXiv: [2402.18164](https://arxiv.org/abs/2402.18164) [cs.LG]. URL: <https://arxiv.org/abs/2402.18164>.
- [11] Stefano Bianchini, Moritz Müller, and Pierre Pelletier. “Drivers and barriers of AI adoption and use in scientific research”. In: *Technological Forecasting and Social Change* 220 (2025), p. 124303. ISSN: 0040-1625. DOI: <https://doi.org/10.1016/j.techfore.2025.124303>. URL: <https://www.sciencedirect.com/science/article/pii/S0040162525003348>.
- [12] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. “The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. 2022, pp. 2091–2103. ISBN: 9781450392211. DOI: [10.1145/3510003.3510057](https://doi.org/10.1145/3510003.3510057). URL: <https://doi.org/10.1145/3510003.3510057>.

- [13] Alex Bogatu et al. “Dataset Discovery in Data Lakes”. In: *Proceedings of International Conference on Data Engineering (ICDE)*. 2020, pp. 709–720. URL: <https://doi.org/10.1109/ICDE48307.2020.00067>.
- [14] Piotr Bojanowski et al. “Enriching Word Vectors with Subword Information”. In: *Transactions of the Association for Computational Linguistics* 5 (2017), pp. 135–146. ISSN: 2307-387X.
- [15] Pierre-Yves Bonnefoy et al. *The Definitive Guide to Data Integration: Unlock the power of data integration to efficiently manage, transform, and analyze data*. Packt Publishing, 2024.
- [16] Damian Bursztyn et al. “Reasoning on web data: Algorithms and performance”. In: *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2015, pp. 1541–1544. URL: <https://doi.org/10.1109/ICDE.2015.7113422>.
- [17] José P. Cambronero and Martin C. Rinard. “AL: Autogenerating Supervised Learning Programs”. In: *Proceedings of the ACM on Programming Languages*. Vol. 3. 2019. URL: <https://doi.org/10.1145/3360601>.
- [18] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. “Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1335–1349. ISBN: 9781450367356. URL: <https://doi.org/10.1145/3318464.3389742>.
- [19] Daniel Cer et al. “Universal Sentence Encoder”. In: *CoRR* abs/1803.11175 (2018). URL: <http://arxiv.org/abs/1803.11175>.
- [20] Nan Chen et al. “VisEval: A Benchmark for Data Visualization in the Era of Large Language Models”. In: *IEEE Transactions on Visualization and Computer Graphics* 31.1 (Jan. 2025), pp. 1301–1311. ISSN: 1077-2626. DOI: [10.1109/TVCG.2024.3456320](https://doi.org/10.1109/TVCG.2024.3456320). URL: <https://doi.org/10.1109/TVCG.2024.3456320>.

- [21] Philippe Cudré-Mauroux. “Leveraging Knowledge Graphs for Big Data Integration: the XI Pipeline”. In: *Semantic Web* 11.1 (2020), pp. 13–17. ISSN: 2210-4968. DOI: [10.3233/SW-190371](https://doi.org/10.3233/SW-190371).
- [22] Xiang Deng et al. “TURL: Table Understanding through Representation Learning”. In: *Proc. VLDB Endow.* 14.3 (2020), pp. 307–319. ISSN: 2150-8097. DOI: [10.14778/3430915.3430921](https://doi.org/10.14778/3430915.3430921). URL: <https://doi.org/10.14778/3430915.3430921>.
- [23] Victor Dibia. “LIDA: A Tool for Automatic Generation of Grammar-Agnostic Visualizations and Infographics using Large Language Models”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*. Ed. by Danushka Bollegala, Ruihong Huang, and Alan Ritter. Association for Computational Linguistics, 2023, pp. 113–126. URL: <https://aclanthology.org/2023.acl-demo.11/>.
- [24] Victor Dibia and Çağatay Demiralp. “Data2Vis: Automatic Generation of Data Visualizations Using Sequence-to-Sequence Recurrent Neural Networks”. In: *IEEE Computer Graphics and Applications* 39.5 (2019), pp. 33–46. DOI: [10.1109/MCG.2019.2924636](https://doi.org/10.1109/MCG.2019.2924636).
- [25] Xin Luna Dong et al. “AutoKnow: self-driving knowledge collection for products of thousands of types”. In: *KDD 2020*. 2020. URL: <https://www.amazon.science/publications/autoknow-self-driving-knowledge-collection-for-products-of-thousands-of-types>.
- [26] Iddo Drori et al. “AutoML using Metadata Language Embeddings”. In: *ArXiv* (2019). URL: <https://arxiv.org/abs/1910.03698>.
- [27] Robert Engels and Christiane Theusinger. “Using a Data Metric for Preprocessing Advice for Data Mining Applications”. In: *In Proceedings of the European Conference on Artificial Intelligence (ECAI)*. 1998, pp. 430–434. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.7414&rep=rep1&type=pdf>.
- [28] Nick Erickson et al. “AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data”. In: *ArXiv* (2020). URL: <https://arxiv.org/abs/2003.06505>.

- [29] Grace Fan et al. “Semantics-aware Dataset Discovery from Data Lakes with Contextualized Column-based Representation Learning”. In: *VLDB Endowment* 16.7 (2023), pp. 1726–1739. ISSN: 2150-8097. URL: <https://doi.org/10.14778/3587136.3587146>.
- [30] Wenqi Fan et al. “A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models”. In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. KDD '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 6491–6501. ISBN: 9798400704901. URL: <https://doi.org/10.1145/3637528.3671470>.
- [31] Matthias Feurer et al. “Auto-Sklearn 2.0: The Next Generation”. In: *arXiv* (2020). URL: <https://arxiv.org/abs/2007.04074>.
- [32] Matthias Feurer et al. “Efficient and Robust Automated Machine Learning”. In: *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 2015, pp. 2962–2970. URL: <https://dl.acm.org/doi/10.5555/2969442.2969547>.
- [33] Nicolo Fusi, Rishit Sheth, and Melih Elibol. “Probabilistic Matrix Factorization for Automated Machine Learning”. In: *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 2018, pp. 3352–3361. URL: <https://dl.acm.org/doi/10.5555/3327144.3327254>.
- [34] Luis Galárraga et al. “AMIE: association rule mining under incomplete evidence in ontological knowledge bases”. In: *Proceedings of the International World Wide Web Conference (WWW)*. 2013, pp. 413–422. URL: <https://doi.org/10.1145/2488388.2488425>.
- [35] Luis Galárraga et al. “Fast rule mining in ontological knowledge bases with AMIE+”. In: *Proceedings of the VLDB J. Endowment, (VLDB)* (2015), pp. 707–730. URL: <https://doi.org/10.1007/s00778-015-0394-1>.
- [36] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997. URL: <https://arxiv.org/abs/2312.10997>.

- [37] Gemini-Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. URL: <https://arxiv.org/abs/2312.11805>.
- [38] P. Gijsbers et al. “An Open Source AutoML Benchmark”. In: *AutoML Workshop at the International Conference on Machine Learning (ICML)*. 2019. URL: <https://arxiv.org/abs/1907.00909>.
- [39] I. J. Good. “Rational Decisions”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 14.1 (1952), pp. 107–114. ISSN: 00359246. URL: <http://www.jstor.org/stable/2984087> (visited on 02/23/2023).
- [40] Isabelle Guyon et al. “A brief Review of the ChaLearn AutoML Challenge: Any-time Any-dataset Learning without Human Intervention”. In: *Proceedings of Machine Learning Research*. Vol. 64. 2016, pp. 21–30. URL: https://proceedings.mlr.press/v64/guyon_review_2016.html.
- [41] Hong Hai Do. *Schema Matching and Mapping-based Data Integration: Architecture, Approaches and Evaluation*. VDM Verlag, 2007. ISBN: 3865509975.
- [42] Olaf Hartig. “Foundations to Query Labeled Property Graphs using SPARQL”. In: *Joint Proceedings of the 1st International Workshop On Semantics For Transport and the 1st International Workshop on Approaches for Making Data Interoperable co-located with 15th Semantics Conference (SEMANTiCS)*. 2019. URL: <http://ceur-ws.org/Vol-2447/paper3.pdf>.
- [43] Ahmed Helal. “Towards Empowering Data Lakes with Knowledge Graphs”. MA thesis. Concordia University, 2021.
- [44] Ahmed Helal et al. “A Demonstration of KGLac: A Data Discovery and Enrichment Platform for Data Science”. In: *Proceedings of VLDB Endowment, (PVLDB)* 12 (2021), pp. 2075–2089. URL: <http://www.vldb.org/pvldb/vol13/p2075-christodoulakis.pdf>.

- [45] Ahmed Helal et al. “A Demonstration of KGLac: A Data Discovery and Enrichment Platform for Data Science”. In: *Proceedings of the VLDB Endowment* 14.12 (2021), pp. 2675–2678. URL: <https://doi.org/10.14778/3476311.3476317>.
- [46] Mossad Helali et al. “A Scalable AutoML Approach Based on Graph Neural Networks”. In: *Proceedings of the VLDB Endowment* 15.11 (2022), pp. 2428–2436. DOI: [10.14778/3551793.3551804](https://doi.org/10.14778/3551793.3551804). URL: <https://arxiv.org/abs/2111.00083>.
- [47] Mossad Helali et al. “KGLiDS: A Platform for Semantic Abstraction, Linking, and Automation of Data Science”. In: *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 2024, pp. 179–192. DOI: [10.1109/ICDE60146.2024.00021](https://doi.org/10.1109/ICDE60146.2024.00021).
- [48] A. Hogan et al. *Knowledge Graphs*. Synthesis Lectures on Data, Semantics, and Knowledge. Morgan & Claypool Publishers, 2021. ISBN: 9781636392363. URL: <https://books.google.ca/books?id=hJlNEAAQBAJ>.
- [49] Xinyi Hou et al. “Large Language Models for Software Engineering: A Systematic Literature Review”. In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (2024). ISSN: 1049-331X. DOI: [10.1145/3695988](https://doi.org/10.1145/3695988). URL: <https://doi.org/10.1145/3695988>.
- [50] Xueyu Hu et al. “InfiAgent-DABench: Evaluating Agents on Data Analysis Tasks”. In: *Proceedings of the 41st International Conference on Machine Learning*. Ed. by Ruslan Salakhutdinov et al. Vol. 235. Proceedings of Machine Learning Research. PMLR, 2024, pp. 19544–19572. URL: <https://proceedings.mlr.press/v235/hu24s.html>.
- [51] Xiao Huang et al. “Knowledge Graph Embedding Based Question Answering”. In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. WSDM ’19. Melbourne VIC, Australia: Association for Computing Machinery, 2019, pp. 105–113. ISBN: 9781450359405. URL: <https://doi.org/10.1145/3289600.3290956>.
- [52] Xin Huang et al. *TabTransformer: Tabular Data Modeling Using Contextual Embeddings*. 2020. arXiv: [2012.06678](https://arxiv.org/abs/2012.06678).

- [53] Madelon Hulsebos et al. “Sherlock: A Deep Learning Approach to Semantic Data Type Detection”. In: *ACM Knowledge Discovery and Data Mining (KDD)*. 2019. DOI: [10.1145/3292500.3330993](https://doi.org/10.1145/3292500.3330993). URL: <http://doi.org/10.1145/3292500.3330993>.
- [54] Hiroshi Iida et al. “TABBIE: Pretrained Representations of Tabular Data”. In: *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Ed. by Kristina Toutanova et al. Association for Computational Linguistics, June 2021, pp. 3446–3456. URL: <https://doi.org/10.18653/v1/2021.naacl-main.270>.
- [55] Piotr Indyk and Rajeev Motwani. “Approximate nearest neighbors: towards removing the curse of dimensionality”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 604–613. URL: <https://doi.org/10.1145/276698.276876>.
- [56] Mohammed Saidul Islam et al. “DataNarrative: Automated Data-Driven Storytelling with Visualizations and Texts”. In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 2024, pp. 19253–19286. DOI: [10.18653/v1/2024.emnlp-main.1073](https://doi.org/10.18653/v1/2024.emnlp-main.1073). URL: <https://aclanthology.org/2024.emnlp-main.1073/>.
- [57] Shaoxiong Ji et al. “A Survey on Knowledge Graphs: Representation, Acquisition, and Applications”. In: *IEEE Transactions on Neural Networks and Learning Systems* 33.2 (2022), pp. 494–514. DOI: [10.1109/TNNLS.2021.3070843](https://doi.org/10.1109/TNNLS.2021.3070843).
- [58] Jun-Peng Jiang et al. *Representation Learning for Tabular Data: A Comprehensive Survey*. 2025. URL: <https://arxiv.org/abs/2504.16109>.
- [59] Juyong Jiang et al. “A Survey on Large Language Models for Code Generation”. In: *arXiv preprint arXiv:2406.00515* (2024).
- [60] Jeff Johnson, Matthijs Douze, and Hervé Jégou. “Billion-Scale Similarity Search with GPUs”. In: *IEEE Transactions on Big Data* 7.3 (2021), pp. 535–547. URL: <https://doi.org/10.1109/TBDATA.2019.2921572>.

- [61] *Kaggle ML Survey 2022*. Accessed: 2024-02-01. 2024. URL: <https://www.kaggle.com/kaggle-survey-2022>.
- [62] *Kaggle Portal*. Accessed: 2022-07-15. 2022. URL: <https://www.kaggle.com/>.
- [63] Michael Katz et al. “Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2020, pp. 403–411. URL: <https://ojs.aaai.org//index.php/ICAPS/article/view/6686>.
- [64] Moe Kayali et al. “Mind the Data Gap: Bridging LLMs to Enterprise Data Integration”. In: *Proceedings of the 2025 Conference on Innovative Data Systems Research*. 2025. URL: <https://vldb.org/cidrdb/papers/2025/p34-kayali.pdf>.
- [65] Aamod Khatiwada et al. “SANTOS: Relationship-based Semantic Table Union Search”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 2023. URL: <https://doi.org/10.1145/3588689>.
- [66] Trang T Le, Weixuan Fu, and Jason H Moore. “Scaling tree-based automated machine learning to biomedical big data with a feature set selector”. In: *Bioinformatics* 36.1 (2020), pp. 250–256. URL: <https://doi.org/10.1093/bioinformatics/btz470>.
- [67] Erin LeDell and Sebastien Poirier. “H2O AutoML: Scalable Automatic Machine Learning”. In: *AutoML Workshop at the International Conference on Machine Learning (ICML)*. 2020. URL: www.automl.org/wp-content/uploads/2020/07/AutoML_2020_paper_61.pdf.
- [68] Cheng-Te Li, Yu-Che Tsai, and Jay Chieh Liao. “Graph Neural Networks for Tabular Data Learning”. In: *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2023, pp. 3589–3592. DOI: [10.1109/ICDE55515.2023.00275](https://doi.org/10.1109/ICDE55515.2023.00275).
- [69] Yang Li et al. “VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition”. In: *Proceedings of the VLDB Endowment* 14.11 (2021), pp. 2167–2176. URL: <http://www.vldb.org/pvldb/vol14/p2167-li.pdf>.

- [70] Yujia Li et al. “Learning Deep Generative Models of Graphs”. In: *ArXiv* (2018). URL: <https://arxiv.org/abs/1803.03324>.
- [71] Sijia Liu et al. “An ADMM Based Framework for AutoML Pipeline Configuration”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. Apr. 2020, pp. 4892–4899. URL: <https://doi.org/10.1609/aaai.v34i04.5926>.
- [72] Pingchuan Ma et al. “InsightPilot: An LLM-Empowered Automated Data Exploration System”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2023, pp. 346–352. URL: <https://doi.org/10.18653/v1/2023.emnlp-demo.31>.
- [73] Pingchuan Ma et al. “MetaInsight: Automatic Discovery of Structured Knowledge for Exploratory Data Analysis”. In: *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2021, pp. 1262–1274. ISBN: 9781450383431. DOI: [10.1145/3448016.3457267](https://doi.org/10.1145/3448016.3457267).
- [74] Paula Maddigan and Teo Susnjak. “Chat2VIS: Generating Data Visualizations via Natural Language Using ChatGPT, Codex and GPT-3 Large Language Models”. In: *IEEE Access* 11 (2023), pp. 45181–45193. URL: <http://doi.org/10.1109/ACCESS.2023.3274199>.
- [75] Yu Malkov and Dmitry Yashunin. “Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.4 (2016), pp. 824–836. DOI: [10.1109/TPAMI.2018.2889473](https://doi.org/10.1109/TPAMI.2018.2889473). URL: <https://doi.org/10.1109/TPAMI.2018.2889473>.
- [76] Essam Mansour, Kavitha Srinivas, and Katja Hose. “Federated Data Science to Break Down Silos [Vision]”. In: *SIGMOD Rec.* 50.4 (2022), pp. 16–22. ISSN: 0163-5808. DOI: [10.1145/3516431.3516435](https://doi.org/10.1145/3516431.3516435). URL: <https://doi.org/10.1145/3516431.3516435>.
- [77] Radu Marinescu et al. “Searching for Machine Learning Pipelines Using a Context-Free Grammar”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2021, pp. 8902–

8911. URL: <https://ojs.aaai.org/index.php/AAAI/article/view/17077>.
- [78] meta-kaggle. <https://www.kaggle.com/kaggle/meta-kaggle>. 2022.
- [79] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. “The Odyssey Approach for Optimizing Federated SPARQL Queries”. In: *Proceedings of the International Semantic Web Conference (ISWC)*. Vol. 10587. 2017, pp. 471–489. URL: https://doi.org/10.1007/978-3-319-68288-4%5C_28.
- [80] Jonas Mueller and Alex Smola. “Recognizing Variables from Their Data via Deep Embeddings of Distributions”. In: *International Conference on Data Mining (ICDM)*. 2019, pp. 1264–1269. URL: <https://doi.org/10.1109/ICDM.2019.00158>.
- [81] Jonas Mueller and Alex Smola. “Recognizing Variables from Their Data via Deep Embeddings of Distributions”. In: *International Conference on Data Mining (ICDM)* (2019), pp. 1264–1269. URL: <https://doi.org/10.1109/ICDM.2019.00158>.
- [82] Aiswarya Raj Munappy, Jan Bosch, and Helena Homström Olsson. “Data Pipeline Management in Practice: Challenges and Opportunities”. In: *Product-Focused Software Process Improvement: 21st International Conference, PROFES 2020, Turin, Italy, November 25–27, 2020, Proceedings*. 2020, pp. 168–184. ISBN: 978-3-030-64147-4. DOI: [10.1007/978-3-030-64148-1_11](https://doi.org/10.1007/978-3-030-64148-1_11). URL: https://doi.org/10.1007/978-3-030-64148-1_11.
- [83] Zan Ahmad Naeem et al. “RetClean: Retrieval-Based Data Cleaning Using Foundation Models and Data Lakes”. In: *Proceedings of the VLDB Endowment*. Vol. 18. VLDB Endowment. 2023.
- [84] Fatemeh Nargesian et al. “Table Union Search on Open Data”. In: *Proceedings of the VLDB Endowment, (PVLDB)* (2018), pp. 813–825. URL: <http://www.vldb.org/pvldb/vol11/p813-nargesian.pdf>.

- [85] Natasha Noy, Matthew Burgess, and Dan Brickley. “Google Dataset Search: Building a search engine for datasets in an open Web ecosystem”. In: *The Web Conference (WebConf)*. 2019.
- [86] Randal S. Olson et al. “PMLB: a large benchmark suite for machine learning evaluation and comparison”. In: *BioData Mining* 10.1 (2017), p. 36. URL: <https://doi.org/10.1186/s13040-017-0154->.
- [87] OpenAI et al. *GPT-4 Technical Report*. 2024. URL: <https://arxiv.org/abs/2303.08774>.
- [88] Micaela S Parker, Arlyn E Burgess, and Philip E Bourne. “Ten simple rules for starting (and sustaining) an academic data science initiative”. In: *PLOS Computational Biology* 17.2 (2021), e1008628. DOI: [10.1371/journal.pcbi.1008628](https://doi.org/10.1371/journal.pcbi.1008628).
- [89] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian Suchanek. “YAGO 4: A Reasonable Knowledge Base”. In: *The Semantic Web*. Cham: Springer International Publishing, 2020, pp. 583–596. ISBN: 978-3-030-49461-2. URL: <https://doi.org/10.1145/1242572.1242667>.
- [90] Jeffrey Pennington, Richard Socher, and Christopher Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: <https://doi.org/10.3115/v1/D14-1162>.
- [91] Matthew Peters et al. “Semi-supervised sequence tagging with bidirectional language models”. In: *Proceedings of the Association for Computational Linguistics (ACL)*. Vol. 1. 2017, pp. 1756–1765. URL: <https://doi.org/10.18653/v1/P17-1161>.
- [92] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. “Meta-Learning by Landmarking Various Learning Algorithms”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. 2000, pp. 743–750. URL: <https://dl.acm.org/doi/10.5555/645529.658105>.

- [93] Jim Plotts and Megan Risdal. *Meta Kaggle Code*. 2023. DOI: [10.34740/KAGGLE/DS/3240808](https://doi.org/10.34740/KAGGLE/DS/3240808). URL: <https://www.kaggle.com/ds/3240808>.
- [94] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. “Automated Machine Learning with Monte-Carlo Tree Search”. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 2019, pp. 3296–3303. URL: <https://doi.org/10.24963/ijcai.2019/457>.
- [95] Matthias Reif, Faisal Shafait, and Andreas Dengel. “Meta-learning for evolutionary parameter optimization of classifiers”. In: *Machine Learning* 87.3 (2012), pp. 357–380. URL: <https://doi.org/10.1007/s10994-012-5286-7>.
- [96] Alex G. C. de Sá et al. “RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines”. In: *Genetic Programming*. Ed. by James McDermott et al. 2017, pp. 246–261. URL: https://doi.org/10.1007/978-3-319-55696-3_16.
- [97] Arvind Satyanarayan et al. “Vega-Lite: A Grammar of Interactive Graphics”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 341–350. DOI: [10.1109/TVCG.2016.2599030](https://doi.org/10.1109/TVCG.2016.2599030).
- [98] Andreas Schwarte et al. “FedX: A Federation Layer for Distributed Query Processing on Linked Open Data”. In: *Proceedings of The Semantic Web: Research and Applications (ESWC)*. Lecture Notes in Computer Science. Springer, 2011, pp. 481–486. URL: https://doi.org/10.1007/978-3-642-21064-8_5C_39.
- [99] Kevin Swersky, Jasper Snoek, and Ryan P. Adams. “Multi-Task Bayesian Optimization”. In: *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*. 2013, pp. 2004–2012. URL: <https://dl.acm.org/doi/10.5555/2999792.2999836>.
- [100] *The GraphDB RDF Engine*. Accessed: 2023-12-01. 2023. URL: <https://www.ontotext.com/products/graphdb/>.

- [101] Chris Thornton et al. “Auto-WEKA: Combined election and hyperparameter optimization of classification algorithms”. In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 2013, pp. 847–855. URL: <https://doi.org/10.1145/2487575.2487629>.
- [102] Alexander Tornede et al. *AutoML in the Age of Large Language Models: Current Challenges, Future Opportunities and Risks*. 2024. arXiv: 2306.08107. URL: <https://arxiv.org/abs/2306.08107>.
- [103] Joaquin Vanschoren et al. “OpenML: Networked Science in Machine Learning”. In: *SIGKDD Explorations* 15 (2013), pp. 49–60. URL: <https://doi.org/10.1145/2641190.2641198>.
- [104] Revathy Venkataramanan et al. “Constructing a metadata knowledge graph as an atlas for demystifying AI pipeline optimization”. In: *Frontiers in Big Data* Volume 7 - 2024 (2025). ISSN: 2624-909X. DOI: 10.3389/fdata.2024.1476506. URL: <https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2024.1476506>.
- [105] Ricardo Vilalta et al. “Using Meta-Learning to Support Data Mining”. In: *International Journal of Computer Science & Applications* 1 (2004). URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.1351&rep=rep1&type=pdf>.
- [106] Boris Villazón-Terrazas et al. “Construction of Enterprise Knowledge Graphs (I)”. In: *Proceedings of Exploiting Linked Data and Knowledge Graphs in Large Organisations*. 2017, pp. 87–116. URL: https://doi.org/10.1007/978-3-319-45654-6%5C_4.
- [107] Denny Vrandečić and Markus Krötzsch. “Wikidata: a free collaborative knowledgebase”. In: *Commun. ACM* 57.10 (Sept. 2014), pp. 78–85. ISSN: 0001-0782. URL: <https://doi.org/10.1145/2629489>.
- [108] Chi Wang et al. “FLAML: A Fast and Lightweight AutoML Library”. In: *Proceedings of Machine Learning and Systems (MLSys)*. Vol. 3. 2021, pp. 434–447. URL: <https://arxiv.org/abs/1911.04706>.

- [109] Yue Wang et al. *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. 2023. URL: <https://arxiv.org/abs/2305.07922>.
- [110] Ralph Weischedel et al. *OntoNotes Release 5.0 LDC2013T19*. Web Download. Philadelphia, 2013. URL: <https://doi.org/10.35111/xmhb-2b84>.
- [111] Marcel Wever, Felix Mohr, and Eyke Hüllermeier. “ML-Plan for Unlimited-Length Machine Learning Pipelines”. In: *AutoML Workshop at the International Conference on Machine Learning (ICML)*. 2018. URL: <https://ris.uni-paderborn.de/download/3852/3853/38.pdf>.
- [112] Kanit Wongsuphasawat et al. “Voyager 2: Augmenting Visual Analysis with Partial View Specifications”. In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017, pp. 2648–2659. ISBN: 9781450346559. DOI: [10.1145/3025453.3025768](https://doi.org/10.1145/3025453.3025768).
- [113] Anatoly Yakovlev et al. “Oracle AutoML: A Fast and Predictive AutoML Pipeline”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3166–3180. URL: <https://doi.org/10.14778/3415478.3415542>.
- [114] Chengrun Yang et al. “OBOE: Collaborative Filtering for AutoML Model Selection”. In: *Proceedings of the International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 2019. URL: <http://doi.org/10.1145/3292500.3330909>.
- [115] Zhiyu Yang et al. “MatPlotAgent: Method and Evaluation for LLM-Based Agentic Scientific Data Visualization”. In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku, Andre Martins, and Vivek Srikumar. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 11789–11804. DOI: [10.18653/v1/2024.findings-acl.701](https://doi.org/10.18653/v1/2024.findings-acl.701). URL: <https://aclanthology.org/2024.findings-acl.701/>.
- [116] Daoguang Zan et al. “Large Language Models Meet NL2Code: A Survey”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. 2023,

- pp. 7443–7464. DOI: [10.18653/v1/2023.acl-long.411](https://doi.org/10.18653/v1/2023.acl-long.411). URL: <https://aclanthology.org/2023.acl-long.411/>.
- [117] Dan Zhang et al. “Sato: Contextual Semantic Type Detection in Tables”. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 1835–1848. ISSN: 2150-8097. URL: <https://doi.org/10.14778/3407790.3407793>.
 - [118] Lei Zhang et al. “MLCopilot: Unleashing the Power of Large Language Models in Solving Machine Learning Tasks”. In: *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2024, pp. 2931–2959. URL: <https://aclanthology.org/2024.eacl-long.179/>.
 - [119] Shuo Zhang and Krisztian Balog. “Auto-completion for Data Cells in Relational Tables”. In: *CIKM ’19*. 2019, pp. 761–770. ISBN: 9781450369763. URL: <https://doi.org/10.1145/3357384.3357932>.
 - [120] Yi Zhang and Zachary Ives. “Finding Related Tables in Data Lakes for Interactive Data Science”. In: *Proceedings of The International Conference on Management of Data, (SIGMOD)*. 2020, pp. 1951–1966. URL: <https://doi.org/10.1145/3318464.3389726>.
 - [121] Weiguo Zheng et al. “Semantic SPARQL Similarity Search Over RDF Knowledge Graphs”. In: *Proceedings of the VLDB Endowment, (PVLDB)* (2016), pp. 840–851. URL: <http://www.vldb.org/pvldb/vol9/p840-zheng.pdf>.
 - [122] Erkang Zhu et al. “JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes”. In: *Proceedings of the International Conference on Management of Data (SIGMOD)*. 2019, pp. 847–864. URL: <https://doi.org/10.1145/3299869.3300065>.

Appendix A

A.1 RAGvis Prompt Templates

The following are the LLM prompt templates used in the RAGvis system.

EDA Semantic Extraction Prompt

Your task is to analyze an EDA code snippet and extract answers for the following questions:

1. Is the analysis in the code univariate, bivariate, or multivariate?
2. What is the high level chart category? Possible chart category types: Bar, Pie, Distribution (e.g. Histogram, KDEPlot), Box (e.g. BoxPlot, ViolinPlot), Scatter (e.g. ScatterPlot, RegressionPlot), Line, ColumnCorrelation, HeatMap, Pairwise.
3. Which columns in the original table are used in the chart axes?
4. Is the data grouped by a certain column before plotting (different from chart axes)?

The response must be in JSON in the following format:

```
{
  "analysis_type": "univariate | bivariate | multivariate",
  "chart_type": "Bar | Pie | Distribution | Box | Scatter | Line | ColumnCorrelation | HeatMap | Pairwise",
  "chart_axes_columns": ["<COLUMN_NAME>", ...],
  "grouping_column": "<COLUMN_NAME>", // null if no grouping column
}
```

If the code snippet contains multiple EDA operations, return a list of json objects, one for each operation. If the previous information is not available or the code snippet is not an EDA code, return an empty json object. return only the raw json object without any code.

Here is some example inputs and outputs:

EXAMPLE 1:

EXAMPLE 1 Input:

Table:

```
Country name,Regional indicator,Ladder score,upperwhisker,lowerwhisker,Log GDP per capita,Social support,Healthy life expectancy,
Finland,Western Europe,7.741,7.815,7.667,1.844,1.572,0.695,0.859,0.142,0.546,2.082
Denmark,Western Europe,7.583,7.665,7.500,1.908,1.520,0.699,0.823,0.204,0.548,1.881
Iceland,Western Europe,7.525,7.618,7.433,1.881,1.617,0.718,0.819,0.258,0.182,2.050
Sweden,Western Europe,7.344,7.422,7.267,1.878,1.501,0.724,0.838,0.221,0.524,1.658
```

Code:

```
...
df2024 = pd.read_csv("/kaggle/input/world-happiness-report-2024-yearly-updated/World-happiness-report-2024.csv", encoding='latin-1')

plt.figure(figsize = (15,8))
sns.kdeplot(x=df2024['Ladder score'], hue = df2024['Regional indicator'], fill = True, linewidth = 2)
plt.axvline(df2024['Ladder score'].mean(),c= 'black')
plt.title('Ladder Score Distribution by Regional Indicator')
plt.show()
...
```

EXAMPLE 1 Output:

```
{
  "analysis_type": "bivariate",
  "chart_type": "Distribution",
  "chart_axes_columns": ["Ladder score"],
  "grouping_column": "Regional indicator",
}
```

EXAMPLE 2:

EXAMPLE 2 Input:

Table:

```
year_film,year_award,ceremony,category,nominee,film,win
1943,1944,1,Best Performance by an Actress in a Supporting Role in any Motion Picture,Katina Paxinou,For Whom The Bell Tolls,True
1943,1944,1,Best Performance by an Actor in a Supporting Role in any Motion Picture,Akim Tamiroff,For Whom The Bell Tolls,True
1943,1944,1,Best Director - Motion Picture,Henry King,The Song Of Bernadette,True
1943,1944,1,Picture,The Song Of Bernadette,,True
```

Code:

```
...
df = pd.read_csv('/kaggle/input/golden-globe-awards/golden_globe_awards.csv')

film_awards_year = df.groupby(['film', 'year_award'])[win].sum()

film_award_df = pd.DataFrame(film_awards_year).reset_index()
film_four_awards = film_award_df[film_award_df[win] >= 4].sort_values(ascending = False, by='win')

film_four_awards = film_four_awards.rename({'film': 'movie', 'win': 'awards'})

plt.figure(figsize=(20,8))
sns.set_style('whitegrid')
sns.barplot(x='movie', y='awards', data=film_four_awards, palette='hls')
plt.title('Movies with atleast 4 awards in a single year', fontsize=12)
plt.xlabel('Movie', fontsize=12)
plt.ylabel('Award Count', fontsize=12)
plt.xticks(rotation=90, fontsize=12)
plt.yticks(rotation=90, fontsize=12)
plt.show()
...
```

EXAMPLE 2 Output:

```
{
  "analysis_type": "bivariate",
  "chart_type": "Bar",
  "chart_axes_columns": ["film", "win"],
  "grouping_column": null,
}
```

EXAMPLE 3:

EXAMPLE 3 Input:

Table:

```
#,Name,Type 1,Type 2,Total,HP,Attack,Defense,Sp. Atk,Sp. Def,Speed,Generation,Legendary
1,Bulbasaur,Grass,Poison,318,45,49,49,65,65,45,1,False
2,Ivysaur,Grass,Poison,405,60,62,63,80,80,60,1,False
3,Venusaur,Grass,Poison,525,80,82,83,100,100,80,1,False
3,VenusaurMega Venusaur,Grass,Poison,625,80,100,123,122,120,80,1,False
```

Code:

```
...
measurements = pd.read_csv('/kaggle/input/air-pollution-in-seoul/AirPollutionSeoul/Original Data/Measurement_info.csv')
measures = measurements.pivot_table(index=['Measurement date', 'Station code', 'Instrument status'], columns='Item code',
values='Average value').reset_index()

measures = measures.loc[measures['Status'] == 'Normal', :]
overview = measures.groupby('Date').mean().loc[:, 'SO2 (ppm)': 'PM2.5 (microgram/m3)']
for pol, func in evaluators.items():
    overview[pol.split()[0] + ' Level'] = overview[pol].map(func)

level_counts = pd.concat([overview[col].value_counts() for col in overview.loc[:, 'SO2 Level':]], axis=1, join='outer', sort=True).fillna(0.0)
level_counts = level_counts.loc[['Very bad', 'Bad', 'Normal', 'Good'], :]

level_counts.T.plot(kind='bar', stacked=True, figsize=(8,6), rot=0,
    colormap='coolwarm_r', legend='reverse')
plt.title('Levels of pollution in Seoul from 2017 to 2019', fontsize=16, fontweight='bold')
plt.show()
...
```

EXAMPLE 3 Output:

```
[]
```

Keeping in mind the previous examples, what is the corresponding output for the following table sample and code snippet?

Table:

{{TABLE_SAMPLE}}

Code:

```
...
{{SOURCE_CODE}}
...
```

EDA Refinement Prompt

I have a dataset in CSV format with the following information about its columns:

{{COLUMN_INFO}}

Your task is to predict a list of EDA operations that comprehensively analyze the dataset. Each EDA operation consists of the chart type and the columns to analyze.

The following are suggestions based on historical EDA operations for similar datasets, sorted from the least to the most frequent:

{{EDA_OPERATIONS}}

The previous are suggestions for EDA operations. They might have incorrect columns or chart types. You may remove EDA operations that are not suitable for this dataset. You may also add EDA operations that are not in the previous list.

Before predicting the EDA operations, think about the following:

1. Which are the most important columns in the dataset to analyze?
2. What univariate, bivariate and multivariate analyses are suitable for this dataset?
3. Which EDA operations in the previous list are applicable to this dataset?
4. Are there any relevant or informative EDA operations not in the previous list?
5. The EDA operations in the previous list are sorted from the least to the most frequent.
6. You must generate EXACTLY **{{NUM_EDA_OPERATIONS}}** DISTINCT EDA operations.

The generated EDA operations must be in the following JSON format:

```
{"chart_type": "<CHART_TYPE>",  
  "chart_columns": ["<COLUMN_NAME>", "<COLUMN_NAME>", ...]}
```

Possible values for <CHART_TYPE>: Bar | Pie | Distribution | Box | Scatter | Line | ColumnCorrelation | HeatMap | Pairwise

For example, a bar chart with columns 'name' and 'age' would have the following corresponding JSON:

```
{"chart_type": "Bar", "chart_columns": ["name", "age"]}
```

You must generate exactly **{{NUM_EDA_OPERATIONS}}** EDA operations. Return only the JSON list.

{{COLUMN_INFO}} Example:

Column 'Age' contains values of type integer. It has 30 unique values and 2 missing values.

Column 'Name' contains values of type natural language. It has 2300 unique values and 0 missing values.

...

{{EDA_OPERATIONS}} Example:

A Bar Chart using the column(s): {'Age', 'Gender', 'City'}.

A Scatter Plot using the column(s): {'Age', 'Salary'} grouped by the column 'City'.

...

EDA Coding Agent - Code Generation Prompt

I have a dataset in CSV format with the following information about its columns:

{{COLUMN_INFO}}

Your task is to write the Python code that generates the following chart:

A **{{CHART_TYPE}}** chart showing the column(s): **{{CHART_COLUMNS}}**

Instructions:

- import all necessary libraries
- read the dataset from the following file: `'{{DATASET_FILE_NAME}}'`
- perform only the requested analysis. The code should generate EXACTLY ONE chart that uses the specified columns.
- You may group by any of the given columns if suitable.
- DO NOT include `plt.show()` or any other blocking statement in the code.
- DO NOT save the figure to a file.
- DO NOT create user-defined functions. Have all the code in the top-level code.
- Write only the Python code without any explanation or comments.

{{COLUMN_INFO}} Example:

Column `Age` contains values of type integer. It has 30 unique values and 2 missing values.

Column `Name` contains values of type natural language. It has 2300 unique values and 0 missing values.

...

EDA Coding Agent - Code Repair Prompt

The code has produced the following error:

...

{{ERROR_STACK_TRACE}}

...

Change your code to fix the error. DO NOT include explanation of the error or fix.
Write only the fixed code.

EDA Notebook Assembly Prompt

I have a dataset in CSV format saved in the following file: **{{DATASET_FILE_NAME}}**

The following are code segments that perform EDA analysis on the dataset.
Your task to combine all the code segment into a single notebook.

Code Segments:

{{EDA_CODE_SNIPPETS}}

Instructions:

- Combine the previous code into a single notebook. Remove the redundant code.
- Add plt.show() after each plot.
- Include the necessary imports.
- You may add comments to explain the code.
- DO NOT change the EDA analysis code itself.

A.2 RAGvis VLM-as-Judge Prompt Template

The following is the prompt template for using a Vision Language Model (VLM) as a judge for evaluating the generated chart quality. This prompt is proposed by the authors of VisEval [20] and was shown to highly correlate with human expert ratings.

Vision Language Model Judge Prompt Proposed By VisEval (Chen et al., 2025)	
System Instructions	
Your task is to evaluate the readability of the visualization on a scale of 1 to 5, where 1 indicates very difficult to read and 5 indicates very easy to read. You will be given a visualization requirement and the corresponding visualization created based on that requirement. Additionally, reviews from others regarding this visualization will be provided for your reference. Please think carefully and provide your reasoning and score. ... <pre>{ "Rationale": "a brief reason", "Score": 1-5 }</pre> ... Examples: - If the visualization is clear and information can be easily interpreted, you might return: ... <pre>{ "Rationale": "The chart is well-organized, and the use of contrasting colors helps in distinguishing different data sets effectively. The labels are legible, and the key insights can be understood at a glance.", "Score": 5 }</pre> ... - Conversely, if the visualization is cluttered or confusing, you might return: ... <pre>{ "Rationale": "While there is no overflow or overlap, the unconventional inverted y-axis and the use of decimal numbers for months on the x-axis deviate from the standard interpretation of bar charts, confusing readers and significantly affecting the chart's readability.", "Score": 1 }</pre> ...	
Prompt	
Visualization Requirement: {{EDA_OPERATION}} Visualization image: {{CHART_IMAGE}} Please assess the readability, taking into account factors such as layout, scale and ticks, title and labels, colors, and ease of extracting information. Do not consider the correctness of the data and order in the visualizations, as they have already been verified. {{EDA_OPERATION}} Example: A Scatter Plot using the column(s): {'Age', 'Salary'} grouped by the column 'City'.	