Linked Data Science powered by Knowledge Graphs [Extended]

Essam Mansour*, Mossad Helali*, Shubham Vashisth*, Philippe Carrier*, Katja Hose§
*Concordia University, Canada
{first}.{last}@concordia.ca
§Aalborg University, Denmark
khose@cs.aau.dk

ABSTRACT

In recent years, we have witnessed a growing interest in data science not only from academia but particularly from companies investing in data science platforms to analyze large amounts of data. In this process, a myriad of data science artifacts, such as datasets and pipeline scripts, are created. Yet, there has so far been no systematic attempt to holistically exploit the collected knowledge and experiences that are implicitly contained in the specification of these pipelines, e.g., compatible datasets, cleansing steps, ML algorithms, parameters, etc. Instead, data scientists still spend a considerable amount of their time trying to recover relevant information and experiences from colleagues, trial and error, lengthy exploration, etc. In this paper, we therefore propose a novel system (KGLiDS) that employs machine learning to extract the semantics of data science pipelines and captures them in a knowledge graph, which can then be exploited to assist data scientists in various ways. This abstraction is the key to enable Linked Data Science since it allows us to share the essence of pipelines between platforms, companies, and institutions without revealing critical internal information and instead focusing on the semantics of what is being processed and how. Our comprehensive evaluation uses thousands of datasets and more than thirteen thousand pipeline scripts extracted from data discovery benchmarks and the Kaggle portal, and show that KGLiDS significantly outperforms state-of-the-art systems on related tasks, such as dataset recommendation and pipeline classification.

ACM Reference Format:
Essam Mansour*, Mossad Helali*, Shubham Vashisth*, Philippe Carrier*, Katja Hose§. 2023. Linked Data Science powered by Knowledge Graphs [Extended]. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD '23).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Data science is the process of collecting, cleaning, and analyzing structured and unstructured data to derive insights or to build models for predicting particular tasks. To achieve these goals, data science is founded on datasets and the pipelines built on top of them. In recent years, we have witnessed a growing interest in data science not only from academia but particularly from companies owning colossal amounts of data and investing in data science platforms to help develop and enhance pipelines analyzing these datasets for various tasks. Furthermore, open data science and collaborative portals, such as Kaggle [29] and OpenML [33], are expanding in popularity as well; the Kaggle portal, for instance, contains tens of thousands of open datasets and hundreds of thousands of associated data science pipelines[1]. Despite the importance of efforts and their large-scale nature, existing systems do not offer a holistic approach but instead consider data science artifacts, e.g., datasets and pipeline scripts, only in isolation from each other; there is little or no support to help users learn from the experiences and best practices connecting datasets and pipelines.

A data scientist building a pipeline, for instance, is generally interested in datasets relevant to the task at hand, whether on open data portals or within their enterprise, as well as in previously built pipelines using these datasets. Integrating related datasets builds upon traditional approaches for data enrichment (e.g. by having more rows or columns) and data integrity checks, and allows not only to fulfill the initial task but can also enable a broader range of analyses by introducing additional dimensions and perspectives. The associated data science pipelines then allow the data scientist to benefit from the accumulated (public or enterprise) domain knowledge about the datasets, which helps accelerating the development and discovery of new solutions and possibilities. Unfortunately, existing portals and platforms have no or little support to assist data scientists in easily exploring and systematically discovering data science artifacts. Instead, existing platforms offer only isolated and limited support for dataset search or pipeline recommendation.

To overcome this lack of support and to allow for the understanding and maintenance of large repositories of pipelines, a number of techniques [1, 4, 18] has been proposed to provide machine-readable semantic abstractions of software code. However, the majority of these systems targets statically-typed languages like Java, where accurate static code analysis is feasible. However, the vast majority of data science pipelines is not written in Java but in Python, which makes these approaches inapplicable. For example, over 91% (~553,000) of pipelines on Kaggle are written in Python according to the Meta Kaggle dataset. Moreover, even existing approaches for Python code [1] resort to using general-purpose static code analysis techniques although for a dynamic language, such as Python, accurate static analysis is challenging or even infeasible in some cases [30]. Hence, there is a need for approaches tailored specifically to the requirements of data science pipelines in Python and their particularities.

Existing platforms, such as Google Dataset Search, or enterprise data catalog tools, such as Amundsen[2], offer a rudimentary form of searching for relevant datasets based on keyword matching. Hence, these systems do not allow discovering datasets based on certain characteristics or similar datasets given a sample or a complete dataset [5, 11, 24]. Other state-of-the-art data discovery approaches,

SIGMOD '23, June 18–23, 2023, Seattle, WA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] https://www.kaggle.com/kaggle/meta-kaggle
[2] http://www.amundsen.io

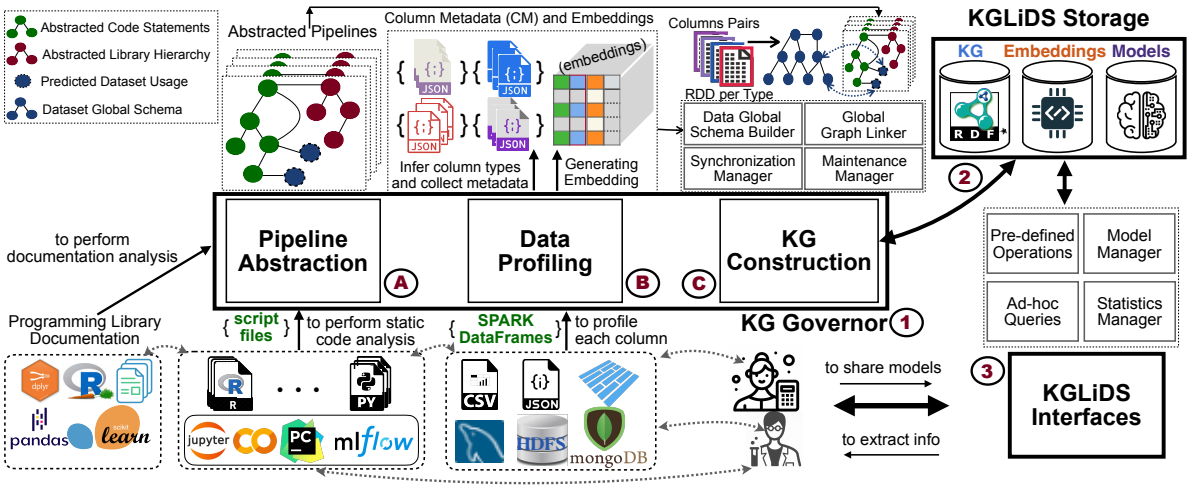

Figure 1: An overview of KGLiDS's main components and the interaction with data sources and pipeline scripts managed by different data science platforms. Data science pipelines and the associated datasets are analyzed via the Pipeline Abstraction and Data Profiling components, respectively. The KG Builder consumes their outputs to construct the LiDS graph and maintains the graph and associated embeddings into an RDF-star and embedding stores. Different users can interact with KGLiDS via the KGLiDS Interfaces to extract information or to share with the community their models developed on top of the LiDS graph.

e.g., for measuring table relatedness [36], identifying joinable [38] or unionable [24] tables, work in isolation from each other and do not support discovery based on a combination of diverse features.

In conclusion, although some techniques have been proposed for dataset and pipeline discovery, they (i) have been studied in isolation, (ii) come with several shortcomings, and (iii) have not been integrated in existing platforms that can be used in practice. In this paper, we therefore propose KGLiDS, a fully-fledged platform powered by knowledge graphs capturing the semantics of data science artifacts, incl. both datasets and pipelines as well as their interconnections. To the best of our knowledge, KGLiDS is the first system to encapsulate both aspects of data science in a single representation. In summary, the contributions of this paper are:

- First fully-fledged platform to enable Linked Data Science[3].
- Scalable deep learning-based data profiling techniques and fine-grained type inference methods to construct a global representation of datasets.
- Capturing the semantics related to data science from pipeline scripts at scale with specific support for Python.
- A novel approach for exploring data science artifacts in consideration of datasets, pipelines, and their connections.
- A comprehensive evaluation using thousands of datasets and more than thirteen thousand pipeline scripts extracted from data discovery benchmarks and Kaggle – showing the superiority of KGLiDS over the state of the art.

The remainder of this paper is organized as follows. Section 2 provides an architecture overview of KGLiDS. Section 3 describes our knowledge graph construction process. Section 4 highlights the KGLiDS interfaces. Section 5 presents the results of our comprehensive evaluation. Sections 6 and 7 are related work and conclusion.

[3]The KGLiDS repository is private and can be accessed at https://gitfront.io/r/CoDS-GCS/Ma9Uz7qkpyqu/kglids/. We will publish the code of KGLiDS as well as the ontology and datasets as open source upon acceptance.

2 KGLIDS AND LINKED DATA SCIENCE

The architecture of KGLiDS is illustrated in Figure 1. The main components are: (1) *KG Governor*, which is responsible for creating, maintaining, and synchronizing the KGLiDS knowledge graph with the covered data science artifacts, pipelines, and datasets, (2) *KGLiDS Storage*, which stores the constructed knowledge graphs as well as embeddings and ML models, (3) *KGLiDS Interfaces*, which allows a diverse range of users to interact with KGLiDS to extract information or share their findings with the community.

2.1 The KG Governor and Data Science Artifacts

KGLiDS captures semantics of data science artifacts, i.e., pipelines and their associated datasets, by applying novel methods for pipeline abstraction and data profiling. During bootstrapping, KGLiDS is deployed by enabling the KG Govenor to profile the local datasets and abstract pipeline scripts to construct a knowledge graph as illustrated in Figure 1. The main subcomponents of the *KG Governor* are: (A) *Pipeline Abstraction*, which captures semantics of pipelines by analyzing pipeline scripts, programming libraries documentation, and usage of datasets, (B) *Data Profiling*, which collects metadata and learns representations of datasets including columns and tables, and (C) *Knowledge Graph Construction*, which builds and maintains the LiDS graph and embeddings.

KGLiDS does not only rely on extracted information and relationships between nodes but can also predict nodes and edges in the knowledge graph using the *Global Graph Linker*, which is part of the *KG Construction* subcomponent. As discussed in more detail in Section 3, KGLiDS adopts embedding-based methods to predict relationships among data items and utilizes MapReduce algorithms to guarantee a scalable approach for constructing the graph from large growing datasets and pipelines. KGLiDS is not a static platform; as more datasets and pipelines are added, KGLiDS continuously and incrementally maintains the LiDS graph. To avoid having to run the prediction every time a user wants to use the

system, predictions are materialized and the resulting nodes and edges annotated with a prediction score expressing the degree of confidence in the prediction.

The Linked Data Science (LiDS) ontology. To store the created knowledge graph in a standardized and well-structured way, we developed an ontology for linked data science: the *LiDS ontology*. Its main types of nodes (classes) are: datasets (with related nodes representing datasets, tables, and columns), libraries, and pipeline scripts (with related nodes describing statements). The *LiDS ontology* conceptualizes the data, pipeline, and library entities in data science platforms, as illustrated in Figure 2. The ontology is specified in the Web Ontology Language (OWL 2) and has 13 classes, 19 object properties, and 22 data properties. OWL was chosen as a standard because of its integral support of interoperability and sharing data on the Web and across platforms. The URIs of classes and properties (relationships) have the prefix *http://kglids.org/ontology/*, while data instances (resources) have the prefix *http://kglids.org/resource/*. For example, a column *Age* in a table *train.csv* of the *titanic* dataset has the URI *http://kglids.org/resource/kaggle/titanic/train.csv/Age*. Similarly, libraries are identified by unique URIs such as *http://kglids.org/resource/library/sklearn/svm/SVC*.

The Linked Data Science (LiDS) graph. We refer to the graph populating the ontology with instances of the classes as the *LiDS graph*. Using the RDF standard [20] , KGLiDS captures and describes the relationships among these entities and uses Uniform Resource Identifiers (URIs) for nodes and edges in the LiDS graph so that the graph can easily be published and shared on the Web. All entities are associated with an RDF label and RDF type to facilitate RDF reasoning on top of the LiDS graph. Example object properties include column similarity relationships: *http://kglids.org/ontology/data/hasSemanticSimilarity*, and data flow between pipeline statements: *http://kglids.org/ontology/pipeline/hasDataFlowTo*, while data properties include statement parameters: *http://kglids.org/ontology/pipeline/hasParameter*.

2.2 The KGLiDS Storage and Interfaces

KGLiDS maintains different types of information, namely the LiDS graph, the generated embeddings for columns and tables, and the machine learning models for different use cases. The current implementation of KGLiDS adopts the RDF model to manage the LiDS graph and uses Stardog as storage engine [10]. KGLiDS uses RDF-based knowledge graph technology because (i) it already includes the formalization of rules and metadata using a controlled vocabulary for the labels in the graphs ensuring interoperability [12, 34], (ii) it has built-in notions of modularity in the form of named graphs, for instance, each pipeline is abstracted in its own named graph [8], and (iii) it is schema-agnostic, allowing the platform to support reasoning and semantic manipulation, e.g., adding new labeled edges between equivalent artifacts, as the platform evolves [7, 13, 37], and (iv) it has a powerful query language (SPARQL) to support federated query processing [2, 22, 31]. In extension, KGLiDS uses the RDF-star [15] model, which supports annotating edges between nodes with metadata, which enables us, for instance, to capture the similarity scores for similarity edges between column nodes of datasets.

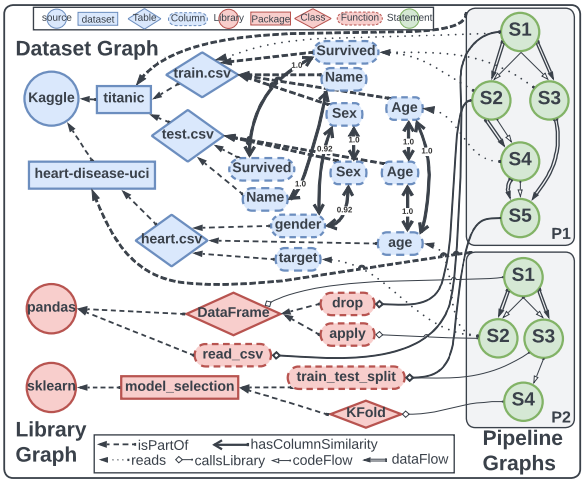

Figure 2: An overview of the LiDS graph, which consists of the dataset, library, and pipeline graphs. Each pipeline is isolated in a named graph.

KGLiDS uses an embedding store, i.e., Faiss [19], to index the generated embeddings and enable several methods for similarity search based on approximate nearest neighbor operations on both CPU and GPU. Thus, KGLiDS enables efficient APIs to interact with the embeddings. This allows users to query the LiDS graph based on the embeddings associated with graph nodes. In Section 4, we discuss KGLiDS's APIs and Embedding Similarity Search. Finally, KGLiDS also stores the ML models developed on top of the LiDS graph.

The KGLiDS portal supports access restrictions to prevent unauthorized users or could be public for anyone. Authorized users have access to query the LiDS graph or embeddings. However, accessing the actual data files in an enterprise may need another level of authorization. A broad range of users with different technical backgrounds can use and benefit from KGLiDS, which provides a simplified interface for non-technical users where pre-configured operations are available, as well as an interface for experts that allows, for instance, editing parameters or formulating ad-hoc queries against the LiDS graph to discover data items and pipelines. Interoperability with other tools was one of KGLiDS's design goals. Hence, KGLiDS, for instance, exports query results as Pandas DataFrame, a widely used format in data science [28].

3 THE KGLIDS GOVERNOR

The core component of the KGLiDS platform is the KG Governor, which captures the semantics of datasets, associated data science pipelines, and programming libraries to construct (i) pipeline graphs, e.g., each pipeline script is abstracted and stored in a separate named graph, (ii) a library graph for all programming libraries used by the abstracted pipelines, and (iii) a dataset graph, i.e., a global schema for datasets accessible by KGLiDS. For pipeline and library graphs, KGLiDS performs static code analysis combined with documentation and dataset usage analysis to naturally interlink a pipeline graph into the dataset and library graphs. For the dataset graph, KGLiDS starts by profiling the datasets at the granularity of columns and then predicts links between columns.

Algorithm 1: Pipeline Abstraction

Input: Pipeline Scripts $\mathcal{S}$, Pipeline Metadata $\mathcal{MD}$, Programming Library Documentation $\mathcal{LD}$

1 **Main Node:**
2 library_hierarchy ← build_library_hierarchy_subgraph($\mathcal{LD}$)
3 pipeline_metadata ←build_pipeline_metadata_subgraph($\mathcal{MD}$)
4 json.dump(library_hierarchy, pipeline_metadata)
5 $\mathcal{S}_{rdd}.map(analyze\_pipeline\_script)$
6 **Worker (Parallel)** *analyze_pipeline_script(s)*:
7 g ← static_code_analysis(s) ▷ Control and data flow graph
8 **for** *node* ∈ *g* ▷ Documentation Analysis
9 **if** *node calls library* `lib` ∈ $\mathcal{LD}$
10 node.return_type ← $\mathcal{LD}$ [lib].return_type
11 node.parameter_names← $\mathcal{LD}$[lib].parameters.names
12 **for** p ∈ $\mathcal{LD}$ *[lib].parameters and* p ∉ *node.parameters*
13 node.default_parameters += (p.name, p.value)
14 **for** *node* ∈ *g* ▷ Dataset Usage Analysis
15 **if** *node calls* `pandas.read_csv(dataset_name.csv)`
16 node.detected_dataset_read ← `dataset_name`
17 **if** *node calls* `pandas.DataFrame[column_name]`
18 node.detected_column_reads += `column_name`
19 json.dump(g) ▷ Save abstracted pipeline graph

3.1 Pipeline Abstraction

A data science pipeline, i.e., code or script, performs one or more data science tasks, e.g., data analysis, visualization, or modelling. The pipeline could be abstracted as a control flow graph, in which code statements are nodes and edges are a flow of instruction or data. The objective of pipeline abstraction is to have a language-independent representation of the pipeline semantics, such as code and data flow within a pipeline, invocations of built-in or third-party libraries, and parameters used in such invocations. This information can be obtained using dynamic or static program analysis.

Dynamic vs. Static Code Analysis. Dynamic analysis involves the execution of a pipeline and examining the memory traces of each statement at run time. Thus, it is more detailed and accurate but does not scale. This is because executing a pipeline is costly in terms of time and memory. It also involves setting up the runtime environment, which is not always possible due to, for example, deprecated libraries. In contrast, static code analysis is less accurate, especially for dynamic languages, such as Python and R. However, it scales well to thousands of pipelines as no pipeline execution is required. To overcome these limitations, KGLiDS combines static code analysis with library documentation and dataset usage analyses to have a rich semantic abstraction that captures the essential concepts in data science pipelines. Algorithm 1 is the pseudocode of our Pipeline Abstraction algorithm. The main inputs of Algorithm 1 are a dataset $S = \{s\}$, where s is a pipeline script, MD metadata of pipelines, such as information of datasets used in the pipeline, pipeline author, and its score, and LD documentations of programming libraries. The algorithm also maintains the library graph for the used libraries and a named graph for each pipeline (lines 2 to 3). Our algorithm decomposes the pipeline abstraction into a set of independent jobs (line 5), where each job is to generate an abstracted graph (named graph) per a pipeline script (line 19).

Static Code Analysis. Algorithm1 applies static code analysis per s (line 7). KGLiDS utilizes the lightweight static code analysis tools, which are natively supported by several programming languages, such as Python (via, for instance `ast` and `astor`) or R (via, for instance, CodeDepends). Each statement corresponds to a variable assignment or a method call. If there are multiple calls in a single line, they will correspond to multiple statements. For each statement, we store the following: i) *code flow*, i.e., the order of execution of statements, ii) *data flow*, i.e., subsequent statements that read or manipulate the same data variable, and iii) Control flow type, i.e., whether the statement occurs in a loop, a conditional, an import, or a user-defined function block, which captures the semantics of how a statement is executed, and iv) statement text, i.e., the raw text of the statement as it appears in the pipeline. We discard from our analysis statements that have no significance in the pipeline semantics, such as `print()`, `DataFrame.head()`, and `summary()`.

Documentation Analysis. Static analysis is not sufficient to capture all semantics of a pipeline. For instance, it cannot detect that `pd.read_csv()` in line 6 in Figure 3 returns a Pandas DataFrame object. In Algorithm 1, we enrich the static program analysis using the documentation of data science libraries (lines 8 to 13). Each statement from static program analysis is enriched by the library it calls, names and values of parameters, including implicit and default ones, and data types of return variables. For each class and method in the documentation, we build a JSON document containing the names, values, and data types of input parameters, including default parameters, as well as their return data types. This analysis enables accurate data type detection for library calls. It also allows the inference of names of implicit call parameters, such as `n_estimators`, the first parameter in line 13 in Figure 3. A useful by-product of documentation analysis is the library graph, indicating methods belonging to classes, sub-packages, etc. (shown in red in Figure 2). This is useful for deriving exciting insights related to data science programming languages. For example, it helps find which libraries are used more frequently than others. KGLiDS, enable retrieving this kind of insight via queries against the LiDS graph.

Predicting Dataset Usage. The critical aspect of our LiDS graph is the realization of connections between pipeline statements and the tables or columns used by the pipeline. These connections enable the novel use-cases of linked data science platforms described in Section 4. In KGLiDS, we build these links in two phases. First, Algorithm 1 applies dataset usage analysis to predict such cases and adds a node of the *Predicted Dataset Usage* (lines 14 to 18). If statement reads a table via `pandas.read_csv` (e.g. line 7 in Figure 3) or a column via string indices over DataFrames (e.g. line 8 in Figure 3), such tables or columns are predicted as potential reads of actual data. Second, when constructing the knowledge graph, these predicted nodes are later verified by the Graph Linker against the `Data Global Schema` of the corresponding dataset.

3.2 Data Profiling

KGLiDS profiles datasets to learn representations (embeddings) of columns and tables, then generates fixed-size and dense embeddings based on their content, e.g., column values, and semantics,

```python
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy score
# Read the dataset
df = pd.read_csv('titanic/train.csv')
X,y = df.drop('Survived', axis=1), df['Survived']
X['NomralizedAge'] = (X['Age'] - X['Age'].mean()) / X['Age'].std()
# Split to train and test
X_train,y_train,X_test,y_test = train_test_split(X,y,test_size=0.2)
print(X_train.shape)
# Train an RF classifier
clf = RandomForestClassifier(50, max_depth=10)
clf.fit(X_train, y_train)
# Evaluate the classifier
print(accuracy_score(y_test, clf.predict(X_test))
```

Figure 3: A running example to demonstrate the KGLiDS Pipeline Abstraction. In this pipeline script, a dataset is loaded using Pandas and split into training and testing portions. Then, a random forest classifier is fitted and evaluated.

e.g., table or column names. Moreover, our profiler collects useful statistics and classifies columns into fine-grained types using our data type inference model. Inspired by [23], we developed a deep learning model to generate embeddings for columns and tables based on their content. For embedding based on semantics, we developed a method based on Word Embeddings [**?**]. KGLiDS analyzes datasets at the level of individual columns and utilizes PySpark DataFrame, enabling scalability via distributed computation and using fixed-size embeddings regardless of the variant sizes of columns. Moreover, KGLiDS inherits the PySpark DataFrame support to handle files of different formats, such as CSV, JSON, and Parquet, and connect to relational DB and NoSQL systems. The pseudocode of our Data Profiling algorithm is shown in Algorithm 2. It receives a dataset $D = \{t\}$, where t is a table, and our models to generate the embeddings and predict fine-grained types. In Algorithm 2, tables are broken down into a set of columns. Then, KGLiDS generates a column profile (JSON) per column.

Data Type Inference. KGLiDS predicts the link between columns by performing a pairwise comparison between embeddings of columns belonging to the same fine-grained type – this help reduce the cost of constructing the dataset graph. Algorithm 2 starts by retrieving a set of tables as Spark DataFrames (lines 2 and 3). A fine-grained type is inferred for each column in a Spark DataFrame (line 7). Unlike other systems, KGLiDS breaks down the main data types (numerical, textual, and dates) into fine-grained types based on pattern matching and deep learning methods. For example, Booleans, emails, or postal codes are identified using predefined regular expressions, while person, organization, or location names are predicted using named entity recognition (NER) models [27]. Having fine-grained types dramatically reduces the number of false positives in column similarity prediction, as these similarities are predicted only between columns of the same fine-grained type.

Dataset Embeddings. The Data Profiling component generates embedding for each column based on a sample of its actual values using our deep dataset embedding (*DeDE*) model. We train the DeDE model to capture similarities between columns. DeDE provides three main advantages to KGLiDS. First, higher accuracy of predicted column content similarities in contrast with standard statistical measures [23]. Second, enabling data discovery without exposing datasets' raw content is invaluable in an enterprise setting, where access to the raw data might be restricted. Third, a compact

Algorithm 2: Data Profiling

Input: Datasets $\mathcal{D}$, NER Model f_σ, Type Matching Patterns $\mathcal{P}$, Dataset Embedding Model h_θ

1 **Main Node:**
2 $\quad$ tables$_{rdd} \leftarrow \{t; t \in \mathcal{D}\}$ ▷ Set of all tables
3 $\quad$ tables$_{rdd}.map(profile_table)$
4 **Worker (Parallel)** *profile_table(t)*:
5 $\quad$ df $\leftarrow$ spark.read(t) ▷ Read the table into a Spark DataFrame
6 $\quad$ **for** $c \in df.columns$ **do**
7 $\quad\quad$ $\mathcal{T} \leftarrow$ infer_fine_grained_type$(c, f_\sigma, \mathcal{P})$
8 $\quad\quad$ $E \leftarrow [0]^{300\times1}$ ▷ 300-Dimensional column embedding
9 $\quad\quad$ **for** $v \in c$ **do**
10 $\quad\quad\quad$ E = E + $\frac{1}{|c|} h_\theta(v)$
11 $\quad\quad$ $\mathcal{M} \leftarrow$ collect_column_metadata$(c, \mathcal{T})$
12 $\quad\quad$ $\mathcal{CP} \leftarrow \{\mathcal{T}, E, \mathcal{M}\}$
13 $\quad\quad$ json.dump$(\mathcal{CP})$ ▷ Save column profile

representation of fixed-size embeddings, regardless of the actual dataset size, greatly reduces the storage requirements.

Two columns have similar embeddings if their raw values have high overlap, have similar distributions, or measure the same variable – even with different distributions (e.g. `temp_in_celsius` is similar to `temp_in_fahrenheit`). To generate a column embedding, a neural network h_θ is applied to each value and averaged for the entire column (lines 9 and 10). We trained h_θ on a collection of 600 OpenML [33] datasets, where the input is a pair of columns and whether they are similar (binary target variable) using the binary cross-entropy loss. In KGLiDS, the embedding of a table is an aggregation of its individual columns:

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

where $|\mathcal{D}|$ is the number of columns in $\mathcal{D}$. 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 2 generates a column's profile containing the predicted fine-grained type $\mathcal{T}$, the generated embeddings E, and the column metadata $\mathcal{M}$ and dumps it as a JSON document (lines 12 and 13). Algorithm 2 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.

3.3 The KG Construction

This section highlights the dataset graph construction, interlinks it to pipeline graphs, and maintains the LiDS graph as a Web-accessible graph based on our LiDS ontology.

Data Global Schema. Algorithm 3 illustrates the pseudocode of our algorithm to construct the dataset graph. The algorithm receives a set $\mathcal{CP} = \{cp\}$ and a set of similarity thresholds. Each cp contains the predicted fine-grained type $\mathcal{T}$, the generated embeddings E, and the column metadata $\mathcal{M}$, which are generated by Algorithm 2. First, Algorithm 3 constructs a metadata subgraph, which contains the hierarchy structure of the datasets and statistics collected for each

Algorithm 3: Data Global Schema Builder

Input: Column Profiles $\mathcal{CP}$, Inferred Column Types: $\mathcal{T}$, Graph $G = \phi$, Similarity Thresholds: $\alpha, \beta, \theta, \gamma$

1 **Main Node:**
2 $\quad \mathcal{CP}_{rdd}$.mapPartitions(column_metadata_worker)
3 **Worker (Parallel)** *column_metadata_worker(cp)*:
4 $\quad$ g = create_metadata_subgraph(cp)
5 $\quad$ json.dump(g) ▷ Save metadata subgraph
6 **Main Node:**
7 $\quad \mathcal{P} = \{(cp_i, cp_j) \mid cp_i, cp_j \in \mathcal{CP}; i \neq j; \mathcal{T}_{cp_i} = \mathcal{T}_{cp_j}\}$
8 $\quad \mathcal{P}_{rdd}$.mapPartitions(column_similarity_worker)
9 **Worker (Parallel)** *column_similarity_worker(cp_i, cp_j)*:
10 $\quad g$ =create_semantic_sim_subgraph($cp_i.name, cp_j.name, \alpha$)
11 $\quad g \leftarrow$create_content_sim_subgraph($cp_i.embed, cp_j.embed, \beta$)
12 $\quad g \leftarrow$create_inclusion_dep_subgraph($cp_i.distr, cp_j.distr, \theta$)
13 $\quad g \leftarrow$ create_pkfk_sim_subgraph($g, cp_i.distr, cp_j.distr, \gamma$)
14 $\quad$ json.dump(g) ▷ Save column similarity subgraph
15 **Main Node:**
16 $\quad$ **for** *subgraph g* **do**
17 $\quad\quad G \leftarrow G \cup g$
18 $\quad$ **return** G ▷ Data global schema graph

column. Next, the similarity relationships are checked between all possible column pairs having the same fine-grained data type and exist in different tables. Algorithm 3 distributes the processing of the pairwise comparisons in a MapReduce fashion (lines 9 to 14).

Each worker takes a pair of cp and generates the similarity edges, i.e., predicates, between them. Two columns have similarity relationships if they have higher similarity scores than the predefined thresholds for the following similarities: (i) semantic similarity: exists between columns that have similar column names based on GloVe Word embeddings [26] and a semantic similarity technique [14], (ii) content similarity: exists between columns that have similar raw values based on the cosine distance between their column embeddings, (iii) inclusion dependency: exists between numerical columns whose range of values are included in one another based on [9], and (iv) primary key foreign key similarity: exists between columns that can be used to join their respective tables. The primary key foreign key relationship predicted by KGLiDS builds upon two major constraints. The primary key must have a high uniqueness ratio, where uniqueness of a column c can be defined as:

$$uniqueness_c = \frac{\mid distinct\_values_c \mid}{\mid total\_values_c \mid} \tag{2}$$

The foreign column must have content similarity relation with the primary column. KGLiDS handles columns based on their type. For columns of type string or string-subsets, our system relies on the embeddings for computing the content similarity. Whereas, in-case of columns of type numeric, a more constrictive approach is followed that further requires an inclusion dependency relationship which is built on a heuristic to comply with the definition that all values of a column are a subset of the other one. Thus, to generate a primary key foreign key edge, for each column type, pairs p are generated where $p = (potential\_primary\_column, potential\_foreign\_column)$. Afterwards, only the pairs where uniqueness $p[1] = 1$ are considered and the rest are filtered out. This results in a filtered set of pairs $p\prime$ where $p\prime = (primary\_column, potential\_foreign\_column)$. Finally, KGLiDS verifies if $p\prime[2] \subseteq p\prime[1]$ as explained. For string columns, a pair is considered valid based on this ratio

$$\frac{size(p'[1] \cap p'[2])}{size(p'[2])} \geq 1 - 10^{-4} \tag{3}$$

Whereas for the numerical columns, it is adequate to examine if the range of the $p\prime[2] \subseteq p\prime[1]$ and the measure of overlap $\geq 1-10^{-4}$. Finally, the dataset graph is constructed at (lines 15 to 18). 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 semantic 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.

Graph Linker. In pipeline abstraction, the dataset usage analysis predicts potential columns or tables read by statements based on predefined heuristics. However, not all predicted columns or tables exist in the dataset. For instance, the column `NormalizedAge` in line 8 in Figure 3 is added by the user and does not exist in the data global schema. The graph linker verifies the predicted tables and columns against the global schema and creates an edge between the statement and the predicted data item if it exists in the data global schema. Therefore, pipelines that read the same table or column are linked together in the global graph.

4 THE KGLIDS INTERFACES

The primary users of KGLiDS are data scientists in an open or enterprise setting whose objective is to derive insights from their datasets, construct pipelines, and share their results with other users. To achieve this goal, KGLiDS offers a number of interfaces (see component 3 in Figure 1). Our interfaces include a set of *Pre-defined Operations*, *Ad-hoc Queries* enabling users to query the raw LiDS graph directly. In addition, the *Statistics Manager* helps collect and manage statistics about the system and the LiDS graph. Finally, the *Model Manager* enables data scientists to run analyses and train models directly on the LiDS graph to derive insights. Examples of these models are presented in Section 5.

We developed the KGLiDS Interfaces library as a Python package that provides simple API interfaces, allowing users to directly access the KGLiDS storage. We designed these APIs to formulate the query results as a Pandas Dataframe, which Python libraries widely support. Thus, data scientists can use our APIs interactively or programmatically while writing their pipeline scripts via Jupyter Notebook or any Python-based data science platform. Due to limited space, the remainder of this section focuses on pre-defined operations.

Pre-defined Operations

Let us consider a scenario where a data scientist is interested in predicting heart failure in patients and illustrate how KGLiDS's pre-defined operations can help achieve this goal.

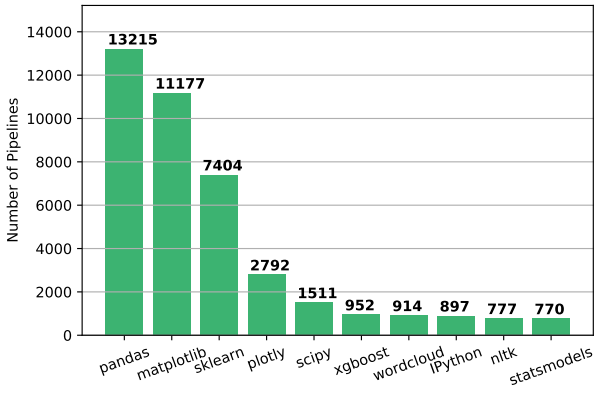

Figure 4: Top-k libraries used (k=10) in 13k kaggle pipelines

Search Tables Based on Specific Columns. To get started, the data scientist would like to find relevant datasets using keyword search, the following operation supports this:

```
table_info = search_keywords([['heart', 'disease'],
                              ['patients']])
```

KGLiDS will then perform a search in LiDS using the conditions that are passed by the user who has the possibility to express conjunctive (AND) and disjunctive conditions (OR) using nested lists. In the example above 'heart' and 'disease' are conjunctive and 'patients' is a disjunctive condition. Let us assume the data scientist has found the following two datasets of interest: `heart-failure-prediction`, `heart-failure-clinical-data`.

Discover Unionable Columns. In the next step, let us assume that the data scientist would like to combine the two tables into one. Since it is very unlikely that the two tables in our example come with exactly the same schema, the data scientist is seeking assistance to identify matching (unionable) columns expressing the same information. KGLiDS provides support to automatically recommend a schema for the merged table containing all columns from both input tables that can be matched (unionable columns). The output will be a Pandas DataFrame. This can be expressed as:

```
find_unionable_columns(table_info.iloc[0],
                       table_info.iloc[1])
```

Join Path Discovery. Let us now assume that the data scientist would like to join the obtained table with a table from another dataset to obtain a richer set of features for downstream tasks. Let us further assume that the table is not directly joinable. Hence, KGLiDS suggests intermediate tables (restricted to a join path with 2 hops in our example) and displays the potential join paths. This is done by computing an embedding of the given DataFrame (`df`), finding the most similar table in the LiDS graph, and determining potential join paths to the given target table. This could be expressed as follows:

```
get_path_to_table(table_info.iloc[0], hops=2)
```

KGLiDS also supports more challenging variations, such as identifying the shortest path between two given tables.

Library Discovery. Before switching to the machine learning phase of their pipeline, the data scientist would like to have a look at the most used libraries by their fellow users. The library graph of KGLiDS can be utilized to retrieve the number of unique pipelines calling a specific library. This can be expressed as:

```
get_top_k_library_used(k: int)
```

where k is the number of libraries. KGLiDS also plots the corresponding bar chart with the top-k used libraries in all the pipelines (see Figure 4). Data scientists can quickly get statistics about different data science artifacts, i.e., used libraries. As the data scientist wishes to predict heart failure and is not an expert in building ML pipelines, KGLiDS assists the data scientist by providing a way to become familiar with the most used libraries in ML pipelines for a specific task, classification in this case. This can be expressed as follows:

```
get_top_used_libraries(k=10, task='classification')
```

Pipeline Discovery. After reading more about some of the most used libraries (namely, Pandas, Scikit-learn, and XGBoost), the data scientist is interested to see example pipelines where the following components are used: `pandas.read_csv`, `XGBoost.XGBClassifier`, and `sklearn.f1_score`. This can be expressed as:

```
get_pipelines_calling_libraries('pandas.read_csv',
    'xgboost.XGBClassifier', 'sklearn.metrics.f1_score')
```

KGLiDS returns a DataFrame containing a list of pipelines matching the criteria along with other important metadata.

Transformation Recommendation. Before training a model, its important to perform the necessary pre-processing and transformations. For the same KGLiDS offers a function to recommend transformations for a given dataset. The data scientist in our example can make use of this function as follows:

```
recommend_transformations(dataset=
  'heart-failure-prediction')
```

KGLiDS returns the set of transformations that could be applied on the given dataset such as sklearn's MinMaxScaler, OneHotEncoder etc. Based on these recommendations the data scientist can then easily transform their data to generate a much more representative dataset for model building.

Classifier Recommendation. Afterwards, the data scientist needs to decide which classification model to use and would like to retrieve suggestions:

```
model_info = recommend_ml_models(
    dataset='heart-failure-prediction',
    task='classification')
```

KGLiDS returns a dataframe with the list of all classifiers that have been used for the given dataset along with their score to assist the data scientist in finalizing the model of their choice.

Hyperparameter Recommendation. In the final step, the data scientist would like KGLiDS to provide help with finding a promising configuration for the hyperparameters of the chosen classifier.

This can be expressed as follows:

```
recommend_hyperparameters(model_info.iloc[0])
```

The data scientist can then perform hyperparameter optimization with ease and use these configuration values to train a model.

5 EXPERIMENTAL EVALUATION

To empirically evaluate KGLiDS, and because there is no holistic system combining programming scripts and data discovery, we compared against related systems that capture the semantics of code or support data discovery. We compare the performance of KGLiDS's components to these systems and analyze the quality of modelling different tasks on top of our LiDS graph.

Compared Systems. We evaluated KGLiDS in terms of pipelines abstraction against GraphGen4Code [1], a toolkit to generate a knowledge graph for code [4]. For data discovery, we evaluated KGLiDS against D^3L [5] and Aurum [11]. Both systems construct a navigational data structure for datasets of tables and CSV files.

Pipeline Scripts and Datasets. From Kaggle [5] we collected 13,800 data science pipeline scripts used in the top 1000 datasets, which includes *3,775* tables and *141,704* columns. We selected these pipelines and datasets based on the number of users' votes on the Kaggle platform. For each dataset, we select up to 20 most voted Python pipelines. The datasets are related to various supervised and unsupervised tasks in different domains, such as health, economics, games, and product reviews. The pipeline scripts have different categories of pipelines, such as pure data exploration and analysis (EDA), and machine learning modelling. We also used the benchmarks provided by D^3L, namely `SmallerReal` and *Synthetic*. `SmallerReal` is a collection of 654 tables. It is collected from the UK Open Data Portal covering different domains, such as business, health, transport, etc.[6] *Synthetic* is a collection of 5,044 tables[7].

RDF Engines and Computing Infrastructure. We used Stardog version *8.0.0* as SPARQL engine to store the LiDS graph. We deployed the endpoint, KGLiDS, and GrapGen4Code on the same local machine. We used two different settings for our experiments. In the first setting, we used a Linux machine with 16 cores and 90GB of RAM. In the second setting, we used a SPARK cluster of 32 machines; each with 64 GB RAM and 16 cores to construct our dataset graphs. Unlike KGLiDS, both D^3L and Aurum are not distributed systems, i.e., cannot use threads across multiple machines. Thus, we compared them in terms of Precision (P) and Recall (R) and reported their results in [5]. KGLiDS scales well as it is based on SPARK.

The KGLiDS Setup. KGLiDS uses Stardog version *8.0.0* as SPARQL engine to store the LiDS graph. We deployed the Stardog, KGLiDS, and GrapGen4Code on the same machine. We analyzed Python pipeline scripts (including Jupyter notebooks) and analyzed the documentation of the top three Python data science libraries, namely, `Pandas`, `Sklearn`, and `Numpy`. For our dataset graph, we set the thresholds for semantic similarity, content similarity, inclusion dependency, and primary-key-foreign-key similarity to 0.75, 0.95, 0.90, and 0.60, respectively.

[4]GrapGen4Code is obtained from https://github.com/wala/graph4code
[5]The script to download these datasets is available at KGLiDS's repository
[6]https://github.com/alex-bogatu/DataSpiders.git
[7]https://github.com/RJMillerLab/table-union-search-benchmark.git

Table 1: A comparative analysis between KGLiDS and GraphGen4Code in terms of pipeline abstraction. For each category in each graph g of T triples, we reported $t(r\%)$, where t is number of triples and r is the ration of t to T.

Modelled Aspect	KGLiDS	GraphGen4Code
Library call	500.5k (3.8%)	15,272k (15.6%)
Code flow	2,110.7k (15.9%)	20,304k (20.8%)
Data flow	1,272.4k (9.6%)	13,295k (13.6%)
RDF node types	2,249.2k (17.0%)	-
Control flow type	809.6k (6.1%)	1,124k (1.2%)
Column reads	212.5k (1.6%)	1,997k (2.0%)
Dataset reads	26.1k (0.2%)	-
Function parameters	3,699.3k (28.0%)	7,511k (7.7%)
Library hierarchy	9.2k (0.1%)	-
Statement text	2,124.5k (16.0%)	7,944k (8.1%)
Statement location	-	3,972k (4.1%)
Variable names	-	987k (1.0%)
Function parameter order	-	25,133k (25.8%)
Total	13,221.9k	97,538k

5.1 Pipeline Graphs: Abstraction and Quality

We compared KGLiDS to GraphGen4Code [1] using the 13,800 Kaggle pipelines. Our comparison focused on pipeline abstraction and accuracy of models trained on the generated graphs.

Semantic Pipeline Abstraction. For this set of pipelines, KGLiDS consumed 113.7 minutes to generate our LiDS graph, which contains 13M triples. GraphGen4Code consumed 2,255.4 minutes to generate a graph of 97M triples for the same set of pipelines. In GraphGen4Code, the focus is abstracting semantics from general code of programs. Thus, GraphGen4Code captures unnecessary semantics from pipeline scripts. Unlike GraphGen4Code, KGLiDS captures semantics related only to data science. Comparing with GraphGen4Code, KGLiDS achieved a graph reduction of more than 86% and even captured semantics, which GraphGen4Code did not manage to capture, as shown in Table 1. For example, capturing semantics related to *statement location*, *variable names*, or *function's parameter order* is irrelevant to pipeline automation or discovery. In contrast, capturing semantics related to datasets, such as *dataset reads*, or libraries, such as *library hierarchy*, is essential for automating data science pipelines, such as discovering pipelines for similar datasets and highly effective libraries.

Modelling Accuracy. We evaluated the quality of the generated graphs for pipelines using four graph classification tasks (two binary and two multi-class), as summarized in Table 2. Task *T1* classifies pipelines into pure EDA or ML Modelling. Task *T2* classifies pipelines into classical ML-based or DL-based pipelines. In task *T3*, ML pipelines are classified by the ML task as either regression, classification, or clustering. In task *T4*, the pipelines are classified based on the application domain of the pipeline. We selected the four most popular domains: finance, education, health, and entertainment. The ground truth for these tasks is based on the tags associated with each pipeline. These tags are provided by the authors who uploaded the pipelines to Kaggle, i.e., human labelled data. We split the pipelines into training, validation, and test sets for each task with ratios 80%, 10%, and 10%, respectively.

Table 2: A comparison between KGLiDS and GraphGen4Code in terms of modelling. For each task, we report $a(t)$, where a is the classification accuracy and t is the training time in min.

System	Binary Tasks		Multi-class Tasks	
	T1	T2	T3	T4
KGLiDS	59.8 (14.6)	**63.5** (8.8)	**45.1** (3.2)	**36.7** (6.5)
GraphGen4Code	**66.1** (69.5)	60.9 (39.8)	38.7 (13.8)	34.8 (19.4)
Baseline Accuracy	50.4	50.5	36.4	27.6
No. Pipelines	1974	1188	551	543

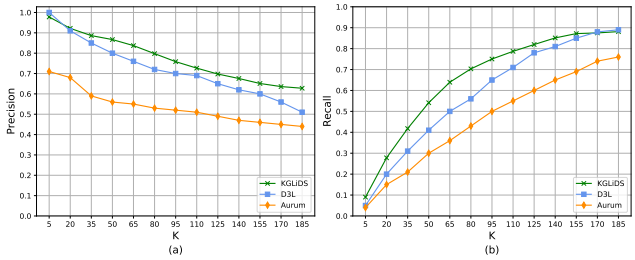

Figure 5: Average precision (a) and recall (b) of table relatedness for KGLiDS, D^3L, and Aurum on `SmallerReal` dataset as k increases from 5 to 185.

We report use the models with the highest validation scores to report the scores on the test set. Further, we down-sample the majority class, if any, to account for model bias.

We trained a Deep Graph Convolutional Neural Network (DGCNN) [35] classification model, which takes a pipeline graph and its class/label as input. Additionally, the model requires for each node the associated node embedding. To this end, we train a structural node embedding model, namely TransE [6], for each pipeline graph (both KGLiDS and GraphGen4Code) separately and feed the trained node embeddings to the DGCNN classifier. We use the same training configuration and hyperparameters when training on KGLiDS or GraphGen4Code. KGLiDS outperformed GraphGen4Code in most tasks. KGLiDS achieved 86% graph reduction and more than 90% acceleration in graph construction. This graph reduction helps prune noisy nodes and edges and generate a graph of high quality. Moreover, our LiDS graph helps automate different aspects of pipelines in faster time as these models are dominated by the number of vertices in the graph, which is significantly more in the graphs generated by GraphGen4Code.

5.2 The Datasets Graph and Data Discovery

We compared KGLiDS to D^3L [5] and Aurum [11] in terms of different data discovery tasks, such as table relatedness and joinable paths, using the same benchmarks used by D^3L. We used both `SmallerReal` and *Synthetic* datasets of D^3L. Both datasets have an average number of 10.7 columns per table. The average number of rows per table is 12,207 in `SmallerReal` and 1,908 in *Synthetic*. Thus, `SmallerReal` is more challenging as each table contains 84% more rows of real data. We got similar results in both datasets.

Table Relatedness. In an extensive collection of datasets, such as a data lake, one of the data discovery tasks is to find related tables. As defined in [5], for a certain table T in a dataset D, (i)

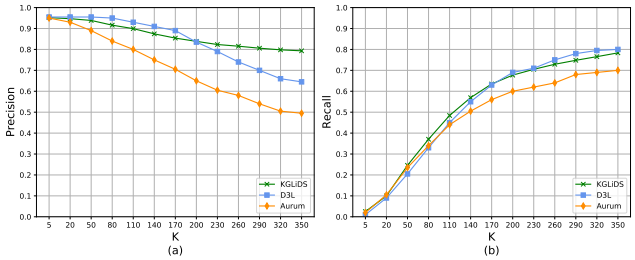

Figure 6: Average precision (a) and recall (b) of table relatedness for KGLiDS, D^3L, and Aurum on `Synthetic` dataset as k increases from 5 to 350.

table relatedness is calculated in terms of Precision@k and Recall@k for different values of k, where k is the number of related tables to T, and (ii) average precision and recall is calculated per 100 target tables sampled at random without replacement for each value of k. KGLiDS consistently outperforms D^3L and Aurum for both precision and recall on the `SmallerReal` dataset, as shown in Figure 5. These results demonstrate the effectiveness of our table unionability relationships by breaking down column data types into sub-types using both a named entity recognition model and pattern matching. This decreases the number of false-positive relationships in these majority string benchmark datasets. The results also show the accuracy of our table similarity score, which considers the number of similar columns and their similarity scores. For the `Synthetic` dataset shown in Figure 6, KGLiDS significantly outperforms Aurum, and performs comparably to D^3L. The different between the systems is less pronounced in `Synthetic` than `SmallerReal` because the latter is a more challenging dataset.

Join Paths. This experiment evaluates the quality of predicted join paths between tables, which is based on *attribute precision*. A join path starting from a related table R is a chain of tables that can be joined on one or more columns to populate R with more columns. Therefore, for a target table T, tables $\mathcal{J} = \{J_1, J_2, ..., J_n\}$ on join paths starting from either of the top-k related tables R are also promising candidates for being related to T – if table joinability is predicted accurately. We evaluate the attribute precision when considering the set $\{R, \mathcal{J}\}$ as related tables vs. when considering R only. Similar to the previous experiment, we calculate attribute precision at different values of k, where we average the results for 100 randomly sampled target tables for each value of k. For both settings i.e. with and without join paths, KGLiDS outperforms Aurum for all values of k, as shown in Figure 7. Attribute precision with join paths is comparable between KGLiDS and D^3L for lower values of k. However, KGLiDS outperforms D^3L for higher values of k with a significantly high margin. Our DeDE embeddings generates more accuare `primary-key-foreign-key` relationships, which leads to this improvement, as shown in Figure 7.

6 RELATED WORK

While some existing systems capture the semantics of code and others support data discovery, KGLiDS is the first platform to combine these aspects in a single system and implement our vision of linked data science on federated datasets [21]. Our platform is powered by knowledge graph technologies to enable the data science community to explore, exchange, and automatically learn from data science artifacts, namely pipeline scripts and datasets. Based on our LiDS

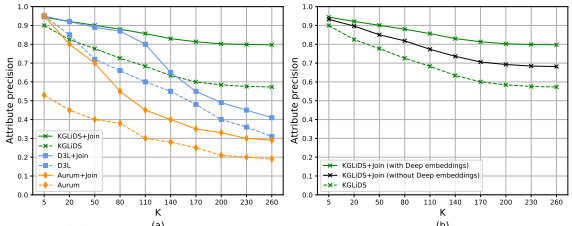

Figure 7: (a) The impact of considering join paths for attribute precision on `SmallerReal` dataset for $k \in [5, 260]$. (b) Using deep dataset embeddings to detect join paths improves the attribute precision in KGLiDS.

graph, we can automate different aspects of data science pipelines, such as pipeline automation or data preparation. For pipeline automation, we formalized the AutoML problem as a graph generation problem [17] to benefit from the graph representation of pipelines.

Capture Pipeline Semantics. The use of knowledge graphs to support applications on top of source code has seen much traction in recent years. Several approaches [1, 3, 4, 18] have been proposed to provide a semantic representation of source code with knowledge graphs. These techniques cannot be generalized to data science pipelines due to their heavy reliance on the detailed static analysis of Java, where information such as method input and return types is straightforward to determine. GraphGen4Code [1] is a toolkit utilizing a tool called WALA [32] for general-purpose static code analysis. Unlike GraphGen4Code, KGLiDS combines static code analysis with library documentation and dataset usage analyses to have a rich semantic abstraction that captures the essential concepts in data science pipelines.

Data Discovery Systems design metadata collected from datasets as different navigational data structures. Consider for example, (*i*) RONIN [25], which builds a hierarchical structure, (*ii*) D^3L [5] that constructs hash-based indices, and (*iii*) Aurum [11], which creates an in-memory linkage graph. These systems do rarely use open standards. We are the first to design navigational data structure 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 [24, 36, 38], KGLiDS also enables users to query datasets based on embeddings and combined easily different operations in one pipeline script. We demonstrated these capabilities of KGLiDS's data discovery support, and we called it KGLac [16].

7 CONCLUSION

This paper describes KGLiDS, a fully-fledged platform to enable Linked Data Science powered by knowledge graphs and machine learning. KGLiDS constructs a highly interconnected knowledge graph capturing the semantics of datasets, pipeline scripts, and code libraries. KGLiDS implements specialized static code analysis that infers information not otherwise obtainable with general-purpose static analysis, resulting in a richer, more accurate, and more compact abstraction of data science pipelines. KGLiDS further utilizes an advanced data profiler empowered by machine learning to analyze data items, including datasets, tables, and columns. In combination, these components enable novel use cases for discovery, exploration, reuse, and automation in data science platforms.

REFERENCES

[1] Ibrahim Abdelaziz, Julian Dolby, James McCusker, and Kavitha Srinivas. 2021. A Toolkit for Generating Code Knowledge Graphs. *Proceedings of Knowledge Capture Conference (K-CAP)* (2021). https://doi.org/10.1145/3460210.3493578

[2] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. 2017. Lusail: A System for Querying Linked Data at Scale. *Proceedings of the VLDB Endowment, (PVLDB)* (2017), 485–498. http://www.vldb.org/pvldb/vol11/p485-abdelaziz.pdf

[3] Awny Alnusair and Tian Zhao. 2010. Component search and reuse: An ontology-based approach. In *Proceedings of IEEE International Conference on Information Reuse & Integration, (IRI)*. 258–261. https://doi.org/10.1109/IRI.2010.5558931

[4] Mattia Atzeni and Maurizio Atzori. 2017. CodeOntology: RDF-ization of Source Code. In *Proceedings of International Semantic Web Conference, (ISWC)*. 20–28. https://doi.org/10.1007/978-3-319-68204-4_2

[5] Alex Bogatu, Alvaro Fernandes, Norman Paton, and Nikolaos Konstantinou. 2020. Dataset Discovery in Data Lakes. In *Proceedings of International Conference on Data Engineering (ICDE)*. 709–720. https://doi.org/10.1109/ICDE48307.2020.00067

[6] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf

[7] Damian Bursztyn, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. 2015. Reasoning on web data: Algorithms and performance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1541–1544. https://doi.org/10.1109/ICDE.2015.7113422

[8] Jeremy Carroll, Christian Bizer, Patrick Hayes, and Patrick Stickler. 2005. Named graphs, provenance and trust. In *Proceedings of the international conference on World Wide Web (WWW)*. 613–622. https://doi.org/10.1145/1060745.1060835

[9] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, and et al. 2019. Inclusion dependency discovery: An experimental evaluation of thirteen algorithms. In *Proceedings of the ACM International Conference on Information and Knowledge Management, (CIKM)*. 219–228. https://doi.org/10.1145/3357384.3357916

[10] The Stardog RDF Engine. 2022. http://stardog.com. http://stardog.com

[11] Raul Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and et al. 2018. Aurum: A Data Discovery System. In *Proceedings of International Conference on Data Engineering (ICDE)*. 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[12] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. 2013. AMIE: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the International World Wide Web Conference (WWW)*. 413–422. https://doi.org/10.1145/2488388.2488425

[13] Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. 2015. Fast rule mining in ontological knowledge bases with AMIE+. *Proceedings of the VLDB J. Endowment, (VLDB)* (2015), 707–730. https://doi.org/10.1007/s00778-015-0394-1

[14] Josu Goikoetxea, Eneko Agirre, and Aitor Soroa. 2016. Single or Multiple? Combining Word Representations Independently Learned from Text and WordNet. In *Proceedings of the Thirtieth Conference on Artificial Intelligence (AAAI)*. 2608–2614. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11777

[15] Olaf Hartig. 2019. 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)*. http://ceur-ws.org/Vol-2447/paper3.pdf

[16] Ahmed Helal, Mossad Helali, Khaled Ammar, and Essam Mansour. 2021. A Demonstration of KGLac: A Data Discovery and Enrichment Platform for Data Science. *Proceedings of VLDB Endowment, (PVLDB)* 12, 2075–2089. http://www.vldb.org/pvldb/vol13/p2075-christodoulakis.pdf

[17] Mossad Helali, Essam Mansour, Ibrahim Abdelaziz, Julian Dolby, and Kavitha Srinivas. 2022. A Scalable AutoML Approach Based on Graph Neural Networks. *In Proceedings of the VLDB Endowment (accepted), (PVLDB)* (2022). https://arxiv.org/abs/2111.00083

[18] Azanzi Jiomekong, Gaoussou Camara, and Maurice Tchuente. 2019. Extracting ontological knowledge from Java source code using Hidden Markov Models. *Open Computer Science* (2019), 181–199. https://doi.org/10.1515/comp-2019-0013

[19] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-scale similarity search with GPUs. *Proceedings of IEEE Transactions on Big Data* 3 (2021), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

[20] Graham Klyne, Jeremy Carrol, and Brian McBride. 2014. RDF 1.1 Concepts and Abstract Syntax. World-Wide Web Consortium. https://www.w3.org/TR/rdf11-concepts/

[21] Essam Mansour, Kavitha Srinivas, and Katja Hose. 2021. Federated Data Science to Break Down Silos [Vision]. *SIGMOD Record* 50, 4 (2021), 16–22. https://doi.org/10.1145/3516431.3516435

[22] Gabriela Montoya, Hala Skaf-Molli, and Katja Hose. 2017. The Odyssey Approach for Optimizing Federated SPARQL Queries. In *Proceedings of the International Semantic Web Conference (ISWC)*, Vol. 10587. 471–489. https://doi.org/10.1007/978-3-319-68288-4_28

[23] Jonas Mueller and Alex Smola. 2019. Recognizing Variables from Their Data via Deep Embeddings of Distributions. In *International Conference on Data Mining (ICDM)*. 1264–1269. https://doi.org/10.1109/ICDM.2019.00158

[24] Fatemeh Nargesian, Erkang Zhu, Ken Pu, and Renée Miller. 2018. Table Union Search on Open Data. *Proceedings of the VLDB Endowment, (PVLDB)* (2018), 813–825. http://www.vldb.org/pvldb/vol11/p813-nargesian.pdf

[25] Paul Ouellette, Aidan Sciortino, Fatemeh Nargesian, Bahar Bashardoost, Erkang Zhu, and et al. 2021. RONIN: Data Lake Exploration. *Proceedings of the VLDB Endowment, (PVLDB)* (2021), 2863–2866. http://www.vldb.org/pvldb/vol14/p2863-nargesian.pdf

[26] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. https://doi.org/10.3115/v1/D14-1162

[27] Matthew Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. 2017. Semi-supervised sequence tagging with bidirectional language models. In *Proceedings of the Association for Computational Linguistics (ACL)*, Vol. 1. 1756–1765. https://doi.org/10.18653/v1/P17-1161

[28] Devin Petersohn. 2021. Scaling Data Science does not mean Scaling Machines. In *Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2021/papers/cidr2021_abstract11.pdf

[29] Kaggle Portal. 2022. Accessed: 2022-07-15. https://www.kaggle.com/

[30] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. *International Conference on Software Engineering (ICSE)* (2021), 1646–1657. https://doi.org/10.1109/ICSE43902.2021.00146

[31] Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. 2011. FedX: A Federation Layer for Distributed Query Processing on Linked Open Data. In *Proceedings of The Semanic Web: Research and Applications (ESWC) (Lecture Notes in Computer Science)*. Springer, 481–486. https://doi.org/10.1007/978-3-642-21064-8_39

[32] WALA Tool. 2022. Accessed: 2022-07-15. https://wala.github.io

[33] Joaquin Vanschoren, Jan Rijn, Bernd Bischl, and Luis Torgo. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations* 15 (2013), 49–60. https://doi.org/10.1145/2641190.2641198

[34] Boris Villazón-Terrazas, Nuria García-Santa, Yuan Ren, Kavitha Srinivas, Mariano Rodriguez-Muro, and et al. 2017. Construction of Enterprise Knowledge Graphs (I). In *Proceedings of Exploiting Linked Data and Knowledge Graphs in Large Organisations*. 87–116. https://doi.org/10.1007/978-3-319-45654-6_4

[35] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the Symposium on Educational Advances in Artificial Intelligence (EAAI)*. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17146

[36] Yi Zhang and Zachary Ives. 2020. Finding Related Tables in Data Lakes for Interactive Data Science. In *Proceedings of The International Conference on Management of Data, (SIGMOD)*. 1951–1966. https://doi.org/10.1145/3318464.3389726

[37] Weiguo Zheng, Lei Zou, Wei Peng, Xifeng Yan, Shaoxu Song, and et al. 2016. Semantic SPARQL Similarity Search Over RDF Knowledge Graphs. *Proceedings of the VLDB Endowment, (PVLDB)* (2016), 840–851. http://www.vldb.org/pvldb/vol9/p840-zheng.pdf

[38] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 847–864. https://doi.org/10.1145/3299869.3300065