Abstra: Toward Generic Abstractions for Data of Any Model

Nelly Barret, Ioana Manolescu, Prajna Upadhyay

Abstra: Toward Generic Abstractions for Data of Any Model

Nelly Barret, Ioana Manolescu, Prajna Upadhyay
Inria & Institut Polytechnique de Paris
nelly.barret@inria.fr,ioana.manolescu@inria.fr,prajna-devi.upadhyay@inria.fr

ABSTRACT

Digital data sharing leads to unprecedented opportunities to develop data-driven systems for supporting economic activities (e.g., e-commerce or maps for tourism), the social and political life, and science. Many open-access datasets are RDF graphs, but others are CSV files, Neo4J property graphs, JSON or XML documents, etc.

Potential users need to *understand* a dataset in order to decide if it is useful for their goal. While some datasets come with a schema and/or documentation, this is not always the case. Data summaries or schema can be derived from the data, but their technical features may be hard to understand for non-IT specialist users, or they may overwhelm users with information.

We propose to demonstrate Abstra, a *dataset abstraction* system, which (*i*) applies on a large variety of data models; (*ii*) computes a *description* meant for humans (as opposed to a schema meant for a parser), akin to an Entity-Relationship diagram; (*iii*) integrates Information Extraction data profiling to *classify* dataset content among a set of categories of interest to the user.

1 INTRODUCTION

Open-access data being shared over the Internet has enormous positive impact. It enables the development of new businesses, economic opportunities and applications; it also leads to circulating knowledge on a variety of topics, from health to education, environment, the arts, science, news, etc.

The World Wide Web Consortium's recommended data sharing format is RDF graphs, and many datasets are shared this way. However, **in practice, other formats are also widely used**. For instance, CSV files are shared on portals such as Kaggle or the French public portal data.gouv.fr; hundreds of millions of bibliographic notices on PubMed, a leading medical scientific site, are available in XML; JSON is increasingly used, e.g., to document the activity of the French parliament on the websites NosDeputes.fr and NosSenateurs.fr, on Twitter, etc. Relational databases are sometimes shared as dumps, including schema constraints such as primary and foreign keys, etc., or as CSV files; property graphs (PGs, in short, such as pioneered by Neo4J) are used to share Offshore leaks, a journalistic database of offshore companies.

Users who must decide whether to use a dataset in an application need a basic **understanding of its content and the suitability to their need**.

Towards this goal, *schemas* may be available to describe the data structure, yet they have some limitations: (*i*) schemas are *often unavailable* for semistructured datasets (XML, JSON, RDF, PGs). Even when a schema is supplied with or extracted from the data, e.g., [4, 9, 16, 19]: (*ii*) schema *syntactic details*, such as regular expressions, etc., are *hard to interpret for non-expert users*; (*iii*) a schema focuses primarily on the dataset structure, not on its *content*. It does not exploit the linguistic information encoded in node names, in the string values the dataset may contain, etc.; (*iv*) schemas employ *the data producer's terminology*, not the categories of interest to users; (*v*) schemas *do not quantitatively reflect the dataset*, whereas knowing "what is the main content of a dataset" can be very helpful for a first acquaintance with it. *Data summarization* has been studied for semistructured data models, e.g., [7, 11]. In the particular case when the dataset is RDF, it may come with an *ontology* describing the semantics of the dataset, which is a step toward lifting limitation (*iii*) above; however, all the others still apply. *Mining for patterns in the dataset* [13] allows to find popular motifs, e.g., items often purchased together, or small groups of strongly connected nodes in a graph, etc. This avoids shortcomings (*i*) and (*v*), but not the others. *Dataset documentation*, when well-written, is most helpful for users. However, it still suffers from limitations (*i*) and (*iv*) above: it is often lacking, and it reflects the producer's view.

We propose to demonstrate Abstra, **a all-in-one system for abstracting any relational, CSV, XML, JSON, RDF or PG dataset**. Abstra is based on the idea that any dataset comprises some *records*, typically grouped in *collections* (which we view as sets). Records describe *entities* or *relationships* in the classical conceptual database design sense [17]; Abstra entities can have deeply nested structure. When several collection of entities co-exist in a dataset, relationships typically connect them. To identify the entities and relationships, Abstra proceeds as follows.

(1). Given any dataset, Abstra **models it as a graph**, and identifies **collections of equivalent nodes**, leveraging graph structural summarization, as we describe in Section 2.

(2). Among the collections, Abstra detects the **main** ones, that is: a small number of collections, each comprising records that may be simple or very complex (i.e., with deeply nested structure), and such that these collections, together, hold a large part of the dataset contents. The challenge here is to detect, in the data graph, the nodes and edges that are "part of" each main collection record, and to do so efficiently even if the graph has complex, cyclic structure. This is addressed by introducing a notion of *data weight* and exploiting it as we describe in Section 3.

(3). Abstra attempts to **classify each collection of entities** into a given **semantic category**, such as Person, Product, GeographicalPosition, etc., using a set of *semantic properties*, some of which we collect from well-known knowledge bases, while others can be elicited from users. The classification leverages Information Extraction to detect the presence of entities in the text fields of the data, then exploit them in our *semantic properties* (Section 4). It also uses language models to detect proximity between the dataset and the target categories.

Abstra outputs a **natural-language, compact description** of the main, classified collections of entities, together with the possible relationships in which they participate; this description is free of any data model-specific details. For instance, given an XMark [18] XML document describing an online auctions site, containing 2.3M nodes, which, together, have 80 different labels, and are organized

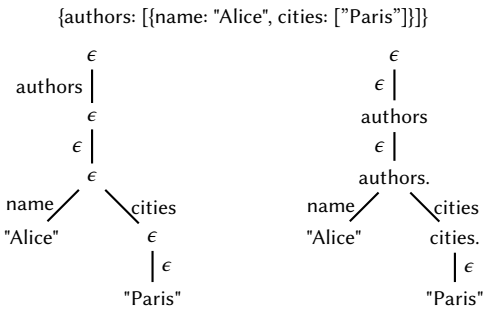

Figure 1: JSON fragment (top), its direct tree model (bottom left), and the model used in Abstra (bottom right).

on 124 labeled paths, Abstra returns: "A collection of Person entities, a collection of Product, and a collection of category" (the latter are used to describe the items for sale). Users can explore them and inspect their internal structure through an interactive GUI (see Section 5).

Below, we describe the abstraction steps and outline the demonstration scenarios, before concluding. Abstra examples and a video can be found at: https://team.inria.fr/cedar/projects/abstra/.

2 BUILDING A COLLECTION GRAPH

We first explain how any dataset is converted in a graph representation (Section 2.1), before partitioning it and constructing the central tool of our method, the collection graph (Section 2.2).

2.1 Graph representation of any dataset

The graph representation we start from has been introduced in ConnectionLens [2, 8], a graph-based heterogeneous data integration system, to which we bring some modifications. Any relational, XML, JSON, RDF, or PG dataset is turned into a **directed graph** $G_0 = (N_0, E_0, \lambda_0)$ where $E_0 \subseteq N_0 \times N_0$ is a set of directed edges, and λ_0 is a function labeling each node and edge with a string label, that could in particular be ϵ (the empty label).

XML trees and RDF graphs naturally map into this modeling.

JSON documents are modeled as trees. A common model, also used in ConnectionLens, turns maps and arrays into unlabeled nodes. Figure 1 shows a sample JSON fragment and, at left, this tree model: the ϵ-labeled nodes, from the top down, correspond respectively to the outermost map, the outermost array, the innermost map, and the innermost array. In Abstra, we are interested in recognizing *groups of nodes that play similar roles* in the dataset (see Section 2.2), and we facilitate that by attaching them more meaningful node names. As illustrated in Figure 1 at the bottom right, we (*i*) move the labels of edges which connect a map parent to its children, on the child nodes; (*ii*) label the children of an array node with label of their parent, to which we concatenate . (a dot). In general, a node's label is computed from the closest non-empty label among its ancestors nodes, and concatenating a dot whenever going from an array to one of its children. This process ensures that every node but the root has a non-empty label.

A CSV file leads to a tree, whose root has an edge going to a node for each tuple; in turn, such a node has edges (labeled with the possible CSV attribute names) going toward each attribute value.

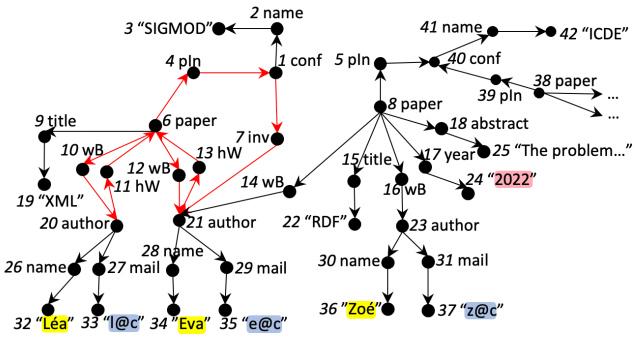

Figure 2: Sample normalized graph.

A relational database is similarly modeled; in the presence of a primary key-foreign key constraint of the form "$R.a$ is a foreign key referencing $S.b$", each node n_r corresponding to an R tuple has an outgoing edge labeled a pointing to the respective S tuple node.

From a PG, we create a node for each PG node and for each of its attributes, with labeled edges connecting them. We also create a node for each PG edge, having one child node for each edge attribute (if any). Whenever the PG contains an edge e from np_1 to np_2, our graph has an edge from np_1 to the node ne representing e, and one from ne to np_2.

In G_0, some edges have empty (ϵ) labels, e.g., parent-child edges in XML, while other edges are labeled. For uniformity, Abstra transforms G_0 into a **normalized graph** G, copying all the nodes of G_0 and all its ϵ-label edges, and replacing each G_0 edge of the form $n_1 \xrightarrow{l} n_2$ where $l \neq \epsilon$ by two unlabeled edges $n_1 \rightarrow x_l, x_l \rightarrow n_2$ where x_l is a new intermediary node labeled l. All subsequent Abstra steps apply on the normalized graph G.

Figure 2 shows a sample bibliographic data graph G. It depicts three papers (one partially shown), which are published in (pIn) conferences. The papers are written by (wb) authors, described by their name and email. Note the inverse "has written" (hW) edges going from authors to their papers. Author *21* is invited (inv) by the conference organizers. As Figure 2 shows, the graph may contain: (*i*) nodes such as papers, whose information content is deeply nested, and (*ii*) several cycles (in-cycle edges are shown in red). Within each leaf (value) node, ConnectionLens **extracts named entities** such as: persons (highlighted in yellow), dates (pink highlight), emails (light blue), etc. Abstra leverages these in order to classify the main entity collections (Section 4).

2.2 Partitioning nodes into collections

To leverage structural information present in G, we build a partition $\mathcal{P} = \{C_i\}_i$ of the graph nodes N, such that $\bigcup_i C_i = N$ and the C_i are pairwise disjoint. We say the nodes from a given set C_i are *equivalent*, and call C_i an *equivalence class*. Many node partitioning schemes, a.k.a. quotient summaries, exist [7]. We need a method that is robust to heterogeneity, i.e., it can recognize the various papers in Figure 2 even though they have heterogeneous structure, and efficiently computed (ideally in linear time in the size of E). For RDF graphs, we use Type Strong summarization [10], which satisfies these requirements; it leverages RDF types when available, but can also identify interesting equivalence classes without them. We extend it also to PGs and graphs derived from CSV files and

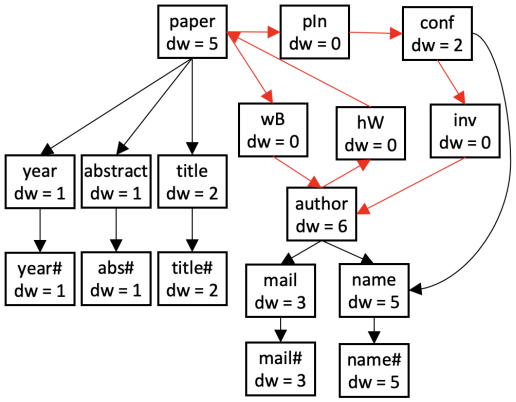

Figure 3: Sample collection graph corresponding to Figure 2.

relational databases. For graphs derived from XML or JSON as discussed in Section 2.1, we simply partition the nodes by their labels. On the graph in Figure 2, the equivalence classes are: {*1*, 40}; {*6*, *8*, *38*}; {*20*, *21*, *23*}, etc.; there is one class for each distinct label of non-leaf nodes, and one class for each set of leaf nodes whose parents are equivalent.

We call *collection graph* the graph whose nodes are the collections C_i, and having an edge $C_i \rightarrow C_k$ if and only if for some nodes $n_i \in C_i, n_j \in C_j$, $n_i \rightarrow n_j \in E$. Figure 3 shows the collection graph corresponding to the graph in Figure 2. Here, in each collection, all nodes have the same label, shown in the collection; this does not hold in general, e.g., in a collection of RDF nodes, each node has a different label. The label year# is used to denote the collection of text children of the nodes from the collection with the label year and similarly for the others whose label end in #. The dw attributes will be discussed in Section 3.

3 IDENTIFYING THE MAIN ENTITIES TO REPORT AND THEIR RELATIONSHIPS

Among the collections C, some are clearly better abstractions of the dataset than others, e.g., in Figure 3, paper seems a better candidate than its child collection year. However, one cannot simply return "the parent (or root) collection(s)": the collection graph may have no root at all, if it is cyclic as in Figure 3 (red edges are part of cycles). Even if a root collection exists, it may not be the best choice. For instance, consider XHTML search results grouped in pages, of the form ⟨top⟩ ⟨page⟩⟨result⟩...⟨/result⟩⟨result⟩... ⟨/result⟩⟨/page⟩ ⟨page⟩... ⟨/page⟩ ... ⟨/top⟩. Here, the top collection is that of pages, but the actual data is in the results, thus, "a collection of results" is a better abstraction.

3.1 Overview of the method

A high-level view of our method is the following (concrete details will be provided below):

(1) **Selecting the main entities** (Section 3.2):
   (a) We assign to each collection a *weight*, and to each edge in the collection graph, a *transfer factor*.
   (b) We *propagate* weights in the collection graph, based on the weights and transfer factors, to assign to each collection a *score* that reflects not only its own weight, but also its position in the graph.
   (c) In a *greedy* fashion, we select *the main entities* by repeating the following steps:
      (i) Select the collection node C_E currently having the highest score, as a *root of a main entity*;
      (ii) Determine *the boundary* of the entity C_E: this is a connected subgraph of the collection graph, containing C_E. We consider all this subgraph as part of C_E, which will be reported to users including all its boundary;
      (iii) *Update the collection graph* to reflect the selection of C_E and its boundaries, and recompute the collection scores;

   until a certain maximum number E_{max} of entities have been selected, or these entities together cover a sufficient fraction cov_{min} of the data.

(2) **Selecting relationships between the main entity collections**. These relationships will also be reported as part of the abstraction (Section 3.3).

3.2 Main entity selection

We assign to each leaf node in G an **own data weight (ow)** equal to **the number of edges incoming that node**. In tree data formats, ow is 1; in RDF, for instance, a literal that is the value of many triples may have $ow > 1$. We leverage this to define the ow **of a leaf collection** as the sum of the ow of its nodes, e.g., in Figure 3, $ow(\text{title\#}) = 2$, $ow(\text{name\#}) = 5$ etc.

For each edge $C_i \rightarrow C_j$ in the collection graph, we define the **edge transfer factor** $f_{j,i}$ as the fraction of nodes in C_j having a parent node in C_i; $0 < f_{j,i} \leq 1$. Intuitively, $f_{i,j}$ of C_j's weight can also be seen as belonging to its parent C_i. For instance, there are 5 name nodes, but two belong to conferences, thus the transfer factor from name to conf is $f_{name,conf} = 2/5$.

We experiment with **two weight propagation methods**.

- We run the PageRank [6] algorithm on the collection graph *with the edge direction inverted*, so that each child node transfers $f_{i,j}$ of its weight to the parent. Initially, only leaf collections have non-zero ow, but successive PageRank iterations spread their weights across the graph. We denote this method $\mathbf{PR}_{ow}$.
- Our second method propagates weights still backwards, but *only outside of the collection graph cycles*. Specifically, we assign to each collection a **data weight** dw, which on leaf collection is initialized to ow, and on others, to 0. Then, for each non-leaf C_i, and non cyclic path from C_i to a leaf collection C_k, we increase $dw(C_i)$ by $f_{k,i} \cdot ow(C_k)$. We denote this method $\mathbf{prop}_{dw}$.

For instance, using the second method, the collection author in Figure 3 obtains $dw = 6$, corresponding to 3 transferred from mail, and 3 transferred from name. The intuition behind prop_{dw} is that edges that are part of cycles may have a meaning closer to "symmetric relationships between entities", than to "including a collection in another collection's boundary".

To **determine entity boundaries**, we proceed as follows:

- When using prop_{dw}, we consider part of the boundary of an entity C_i, any entity C_k that transferred some weight to C_i, and all the edges along which such transfers took place. For instance, in Figure 3, mail and name are within the boundary of author.

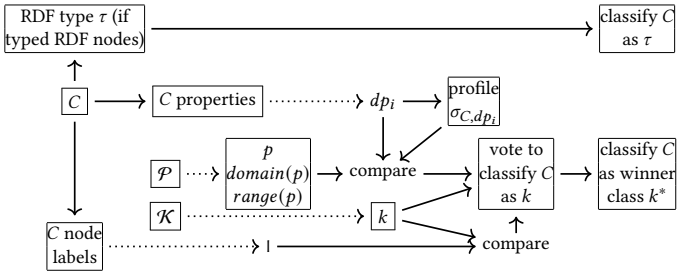

Figure 4: Entity classification outline.

- When using PR_{ow}, to determine the boundary of C_i, we traverse the graph edges starting from C_i and include its neighbor node C_j if and only if (*i*) the edge $C_i \rightarrow C_j$ has a transfer factor of at least f_{min}, or (*ii*) each node from C_i has at most one child in C_j. The intuition for (*ii*) is that such a child C_j "can be assimilated to an attribute of C_i", rather than being "independent of it".

Finally, to **update the graph** after selecting one main entity C_E, each leaf collection in the boundary of C_E *subtracts from its own weight ow* the fraction (at most 1.0) that it propagated to C_E. For instance, once author is selected with name in its boundary, the *ow* of name decreases to 2. Then, the scores of all graph collections (*dw*, respectively, PageRank score based on *ow*) are recomputed.

3.3 Relationship selection

Having selected the main entities $\{C_E^1, \ldots, C_E^{max_E}\}$ and their boundaries, every oriented path in the collection graph that goes from a given C_E^i to another C_E^j is reported as a relationship. For instance, in Figure 3, if the main entities are author (with mail and name in its boundary) and paper (with year, title and abstract in its boundary), the relationships reported are: paper $\xrightarrow{\text{wB}}$ author, author $\xrightarrow{\text{hW}}$ paper, and paper $\xrightarrow{\text{pIn.conf.inv}}$ author.

If the scores lead to reporting three main entities, the two above entities and also conf (with name and inv in its boundary), the relationships are: paper $\xrightarrow{\text{wB}}$ author, author $\xrightarrow{\text{hW}}$ paper, paper $\xrightarrow{\text{pIn}}$ conf, and conf $\xrightarrow{\text{inv}}$ author.

3.4 Discussion

Abstra may return different results on a given dataset, depending on the scoring method used ($prop_{dw}$ or PR_{ow}), as well as the parameters: E_{max} and cov_{min} (Section 3.1), and f_{min} (Section 3.2). Empirically, we have used $E_{max} \in \{3, 5\}$, $cov_{min} = 0.8$ and $f_{min} = 0.3$. More generally, classical Entity-Relationship (E-R) modeling is known to include a subjective factor, and for a given database, several E-R models may be correct. Our focus is on *not missing any essential component of the dataset*, while allowing users to *limit the amount of information through* E_{max}, and *classifying the main entities into semantic categories*, to make them as informative as possible, as we explain below.

4 MAIN ENTITY CLASSIFICATION

To each main entity C thus identified, we want to associate a category from a predefined set $\mathcal{K}$ of semantic classes, and using also a set $\mathcal{P}$ of semantic properties, which have known domain and range constraints connecting them to the classes in $\mathcal{K}$.

We bootstrapped our $\mathcal{K}$ and $\mathcal{P}$ by selecting six common classes (`Person`, `Place`, `Organization`, `Creative Work`, `Event` and `Product`), and associating them, based on WikiData and YAGO, properties they are likely to have, or to be values of. For instance, a `Person` has property `birthPlace`, while the value of `birthPlace` is a `Place`). Next, we rely on GitTables [14], a repository of 1.5M tables extracted from Github. For each attribute name encountered in a table, it provides candidate properties from DBPedia [3] and/or schema.org[1]; it also provides the domain and range triples corresponding to these properties. For instance, GitTable's entry for gender is:

"id":"schema:gender", "label":"gender",
"description":"Gender of something, typically a Person,
"domain":["schema:Person","schema:SportsTeam"],
"range": ["schema:GenderType","schema:Text"]

GitTables have been populated using SHERLOCK [15], a state-of-the-art deep learning semantic annotation technique. From GitTables, we derive 4.187 $\mathcal{P}$ properties; 3.687 among them have domain information, and 3.898 have range statements.

The overall classification process is outlined in Figure 4; solid arrows connect associated data items and trace the classification process, while dotted arrows go from a set to one of its elements.

At the top of the figure, if the collection contains RDF resources considered equivalent by RDFQuotient due to a common type τ, we return that type, considering it is the most precise.

Otherwise, we exploit two kinds of information attached to C:

(1) We consider each *data property* dp_i that C has, such as mail for the author collection in Figure 3. Out of all the values that dp_i takes on a node from C, we compute an *entity profile* σ_{C,dp_i}, reflecting the entities extracted from these values. For instance, $\sigma_{\text{author, mail}}$ states that each value of the property mail contains an email (blue highlight in Figure 2), and that overall, 100% of the length of these values is part of an email entity. In general, a profile may reflect the presence of entities of several types, which may span over only a small part of the property values.
We compare dp_i and σ_{C,dp_i} to each property $p \in \mathcal{P}$ and the classes in the range of p. If the property name dp_i is sufficiently similar (through word embeddings) with some property $p \in \mathcal{P}$, and that σ_{C,dp_i} is similarly sufficiently similar to a class $k_i \in \mathcal{K}$, e.g., `EmailAddress`, that is in the range of p, this leads to a *vote* of dp_i for classifying C, in every classes $k \in \mathcal{K}$ such that k is in the domain of p. Each property dp_i may "vote" in favor of several classes, via different domain constraints; the higher the similarity between dp_i and p, the more frequent dp_i is on C nodes, the fewer domain and range constraints p has, the stronger the vote is.

(2) The *labels of C nodes* may also "vote" toward classifying collection C. All C nodes may have the same label, e.g., author, which may resemble the name of a class, e.g., `Author`. Or, C nodes may all have different names, e.g., RDF URIs of the form http://ns.com/Author123, from which we extract the component after the last /, eliminate all but alphabet letters,

[1] https://schema.org/

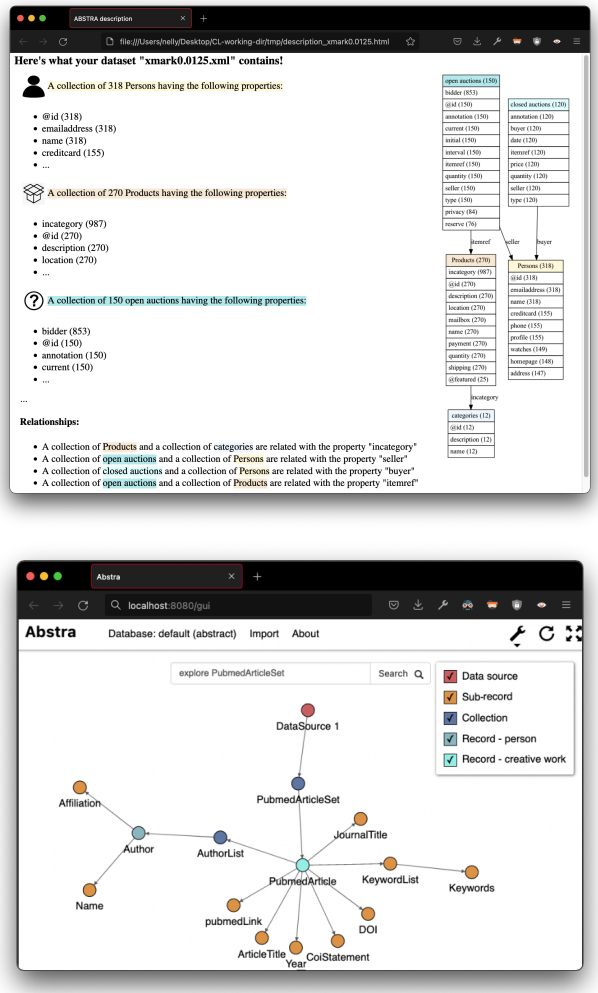

Figure 5: Sample screenshots of the Abstra GUI.

and use the result(s), weighted by their support among C nodes, in a similar fashion.

Finally, C is classified with the class $k^* \in \mathcal{K}$ having received the highest sum of votes. The process resembles domain inference using RDF Schema ontology constraints, with the difference that our "votes" are quantified by similarity and support, and, to keep things simple for the users, we select a single class, the one having the strongest support.

5 SYSTEM AND SCENARIOS

Abstra is implemented in Java, leveraging the graph creation (including entity extraction) and Postgres-based store of ConnectionLens [2]. All Abstra steps scale up linearly in the data size, which we experimentally verified on datasets of up to tens of millions of edges. The main memory needs are in TypedStrong partitioning, namely $O(|N|)$; other operations are implemented in SQL and benefit from Postgres' optimizations.

We have computed abstractions of dozens of synthetic benchmark datasets used in the data management literature, such as XMark [18], BSBM [5], LUBM [12], and real-life datasets about cooking, NASA flights, clean energy production, Nobel prizes, CSV datasets from Kaggle, etc. varying the data model, the number of collections and entity complexity, the presence of relations, etc. During the demonstration, users will be able to: (*i*) change the system parameters, and see the impact on the classification results; (*ii*) edit the semantic information $\mathcal{K}$ and $\mathcal{P}$ to influence the entity classification; (*iii*) edit a set of small RDF, JSON and XML examples, then abstract them to see the impact.

Abstractions are shown both as **HTML text** and as a **lightweight E-R** diagram of the main entity collections and their relationships. Clicking on a collection launches a **node-link GUI** (JavaScript) enabling users to see records from the respective collection, navigate to neighbors, etc. (see Figure 5).

6 CONCLUSION

Abstra extracts the main entities and relationships from heterogeneous-structure datasets, leveraging Information Extraction, language models, knowledge bases, and user input to classify the main collections, so as to get close to the user's interest. The goal of our tool is to generate, as automatically as possible, compact dataset descriptions. Abstra complements schema extraction [4, 9, 16, 19] or data profiling [1], aimed at more technical uses and users; Abstra aims to help novice, first-time users discover and start interacting with the data, through compact, graphical abstractions.

REFERENCES

[1] Z. Abedjan, L. Golab, F. Naumann, and T. Papenbrock. *Data Profiling*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[2] A. C. Anadiotis, O. Balalau, C. Conceicao, H. Galhardas, M. Y. Haddad, I. Manolescu, T. Merabti, and J. You. Graph integration of structured, semistructured and unstructured data for data journalism. *Information Systems*, July 2021.

[3] S. Auer et al. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] M. A. Baazizi, C. Berti, D. Colazzo, G. Ghelli, and C. Sartiani. Human-in-the-loop schema inference for massive JSON datasets. In *EDBT*, 2020.

[5] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *IJSWIS*, 5(2):1–24, 2009.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Networks*, 30(1-7):107–117, 1998.

[7] S. Cebiric, F. Goasdoué, H. Kondylakis, D. Kotzinos, I. Manolescu, G. Troullinou, and M. Zneika. Summarizing Semantic Graphs: A Survey. *The VLDB Journal*, 28(3), June 2019.

[8] C. Chanial, R. Dziri, H. Galhardas, J. Leblay, M. L. Nguyen, and I. Manolescu. ConnectionLens: Finding connections across heterogeneous data sources (demonstration). *PVLDB*, 11(12), 2018.

[9] D. Colazzo, G. Ghelli, and C. Sartiani. Schemas for safe and efficient XML processing. In *ICDE*. IEEE Computer Society, 2011.

[10] F. Goasdoué, P. Guzewicz, and I. Manolescu. RDF graph summarization for first-sight structure discovery. *The VLDB Journal*, 29(5), Apr. 2020.

[11] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[12] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

[13] J. Han, M. Kamber, and J. Pei. *Data mining concepts and techniques, third edition*. Morgan Kaufmann Publishers, 2012.

[14] M. Hulsebos, Ç. Demiralp, and P. Groth. Gittables: A large-scale corpus of relational tables. *CoRR*, abs/2106.07258, 2021.

[15] M. Hulsebos, K. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, Ç. Demiralp, and C. A. Hidalgo. Sherlock: A Deep Learning Approach to Semantic Data Type Detection. *SIGKDD explorations : newsletter of the Special Interest Group (SIG) on Knowledge Discovery & Data Mining*, May 2019.

[16] H. Lbath, A. Bonifati, and R. Harmer. Schema inference for property graphs. In *EDBT*. OpenProceedings.org, 2021.

[17] R. Ramakhrishnan and J. Gehrke. *Database Management Systems (3rd edition)*. McGraw-Hill, 2003.

[18] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *PVLDB*, 2002.

[19] W. Spoth, O. A. Kennedy, Y. Lu, B. Hammerschmidt, and Z. H. Liu. Reducing ambiguity in JSON schema discovery. In *SIGMOD*, 2021.