Modular framework for similarity-based dataset discovery using external knowledge

Martin Nečaský and Petr Škoda
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czechia
David Bernhauer
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czechia and Department of Software Engineering, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czechia, and
Jakub Klímek and Tomáš Skopal
Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague, Czechia

Received 27 September 2021
Revised 3 December 2021
Accepted 17 December 2021

Abstract

Purpose – Semantic retrieval and discovery of datasets published as open data remains a challenging task. The datasets inherently originate in the globally distributed web jungle, lacking the luxury of centralized database administration, database schemes, shared attributes, vocabulary, structure and semantics. The existing dataset catalogs provide basic search functionality relying on keyword search in brief, incomplete or misleading textual metadata attached to the datasets. The search results are thus often insufficient. However, there exist many ways of improving the dataset discovery by employing content-based retrieval, machine learning tools, third-party (external) knowledge bases, countless feature extraction methods and description models and so forth.

Design/methodology/approach – In this paper, the authors propose a modular framework for rapid experimentation with methods for similarity-based dataset discovery. The framework consists of an extensible catalog of components prepared to form custom pipelines for dataset representation and discovery.

Findings – The study proposes several proof-of-concept pipelines including experimental evaluation, which showcase the usage of the framework.

Originality/value – To the best of authors' knowledge, there is no similar formal framework for experimentation with various similarity methods in the context of dataset discovery. The framework has the ambition to establish a platform for reproducible and comparable research in the area of dataset discovery. The prototype implementation of the framework is available on GitHub.

Keywords Dataset, Discovery, Search, Framework, Similarity, Knowledge graph

Paper type Research paper

1. Introduction

The number of datasets available on the web increases tremendously. For example, the number of datasets published by public authorities in European countries increased from 880k datasets in August 2019 [1] to 1140k datasets in November 2021 [2]. Also Google *observed an explosive growth in the number of available datasets in recent years* according to Benjelloun *et al.* (2020).

This work was supported by the Czech Science Foundation (GAČR), Grant Number 19-01641S.

Although there exist dataset catalogs providing search for datasets, their retrieval features are restricted to simple keyword search based on textual metadata recorded in the catalog. These simple search methods presume that their users, the data consumers, know exactly what they are searching for and which search query leads to the expected results. However, this assumption is usually not valid, and, in principle, it neglects the very purpose of the catalogs. When users know which datasets they are searching for, they usually also know who publishes a dataset and how the publisher titles the dataset. With this knowledge, it is quite straightforward to locate a dataset on the publisher's website using a generic search engine. The genuine purpose of data catalogs emerges when users do not exactly know which datasets they are searching for and how to find them. This is a usual situation where the users know only a few keywords and topics that roughly characterize the needed data. The problem of missing information about data is inherently related to the big data phenomenon and is generally discussed as the problem of data findability by Zezula (2015). In their studies, Gregory *et al.* (2020a), Koesten (2018) and Degbelo (2020) show that users typically need to search for more than a single, isolated dataset. Typically, the users wish to find multiple datasets similar to each other in some way, and this is where the pure metadata-based search methods come up short. The studies also show that dataset discovery depends on the context of the user's needs and discovery tasks. Various works such as Fernandez *et al.* (2018), Zhang and Balog (2018) and Mountantonakis and Tzitzikas (2018) also show that dataset content can be important for building dataset discovery services. Therefore, it is not easy to construct a dataset discovery service on top of a single similarity discovery method. It is necessary to be able to experiment with various combinations of different methods and compare them. This leads us to the following research questions we try to solve in this paper:

RQ1. How can we support such experiments with different similarity dataset discovery methods?

RQ2. How can we support combining the methods to more complex pipelines for computing dataset similarities?

RQ3. How can we evaluate and compare different pipelines?

In this paper, we introduce a modular framework for rapid experimentation with methods for similarity-based dataset discovery, using the perspective of software engineering. We are aware that the development of an ultimate and universal method for dataset discovery would be an infeasible effort. This is based on our previous work – Škoda *et al.* (2019), Skopal *et al.* (2021) – where we already experimented with various similarity discovery methods. We have measured them on various real search scenarios, and we showed that none of the evaluated methods performs best on all the scenarios. In this paper, we do not propose yet another method. Instead, we focus on answering the research question above by proposing a framework for experiments with various dataset similarity methods.

Therefore, the framework is not proposed as a complete solution to particular dataset discovery problems, but it should rather act as an extensible modular toolbox for experimentation with various dataset discovery pipelines, including future ones. It supports experimentation by providing a predefined and extensible set of compatible components which can be combined to more complex pipelines which can then be measured, evaluated and compared. Although the framework is designed as generic and extensible, its retrieval model is based on the similarity search paradigm that proved to be an effective general mechanism for retrieval of complex data. Another feature of the framework is its presumption of external knowledge in the process of dataset discovery, which is essential to retrieval using different dataset contexts. In Škoda *et al.* (2020), we proposed a framework for evaluation of dataset discovery methods. This paper focuses not only on the evaluation but also on the experimentation with the methods and their combinations.

1.1 Paper contributions

We identify four contributions in this paper:

(1) The modular and extensive architecture of the framework provides a formal meta-model solution under which particular steps or methods can be easily connected together into dataset discovery pipelines.

(2) The framework includes a proof-of-concept component catalog that could be used as a predefined source for assembling discovery pipelines.

(3) We demonstrate the framework benefits using a proof-of-concept implementation and evaluation of several pipelines constructed within the framework.

(4) The extensible architecture of the framework could provide other scientists both formal and system platforms for reproducible and repeatable research in the area of dataset discovery.

The rest of the paper is organized as follows. In Section 2, we give an overview of the dataset discovery area and various discovery methods with the main focus on similarity-based methods. We also show that there is a lack of a framework introduced in this paper. In Section 3, we introduce the framework. We present its layered architecture and describe its layers in detail. In Section 4, we prove the viability of our approach using a proof-of-concept where we construct concrete pipelines for measuring dataset similarity and its presentation to users willing to discover datasets. We conclude in Section 5.

2. Related work

2.1 Importance of dataset discovery

Finding related datasets, or shortly dataset discovery, is one of the important tasks in data integration as shown by Miller *et al.* (2018). Chapman *et al.* (2020) recognize dataset discovery as a research field with its unique technical challenges and open questions. Large companies such as Google develop their own dataset search techniques and solutions (Brickley *et al.*, 2019). New solutions for dataset search in specific domains started to appear recently. For example, Chen *et al.* (2018) introduced *Datamed*, which is an open source discovery index for finding biomedical datasets. The field of dataset discovery is not studied only from the technical point of view but also from a more social point of view. Gregory *et al.* (2020a) investigate how researchers discover data they need for their research projects on the base of the large survey among researchers (1,677 respondents from 105 countries). Gregory *et al.* (2020b) investigate the same problems by analyzing existing research literature and interviewing scientists who need to discover datasets for their work. Koesten (2018) interviewed 20 data professionals asking them questions on how they search for datasets. These recent studies and surveys show that dataset discovery is an important problem which needs further research. Gregory *et al.* (2020a, b) conclude that dataset search engines could help searchers looking for data outside their domain to better identify new possible sources of data. Gregory *et al.* (2020a) moreover show that data needed for a research task can be diverse data from different sources of various types. Degbelo (2020) formulate 27 open data user needs as a synthesis of current findings from recent literature focusing on smart city data. They structure the statements to 10 categories. One of the categories is called *serendipitous resource discovery (SRD)* which involves user questions such as “Are there datasets that I never thought of, that could also be relevant to my tasks?.” Degbelo (2020) emphasize the need of cross-linking to other datasets related to the dataset being looked at by the user.

2.2 Dataset discovery techniques

All the studies emphasize the role of quality metadata for dataset findability, while Chapman *et al.* (2020) point out that available metadata do not always describe what is actually in a dataset and whether a described dataset fits for a given task. Gregory *et al.* (2020a, b) and Koesten (2018) confirm that dataset discovery is highly contextual depending on the current user's task. The studies show that this contextual dependency must be reflected by the dataset search engines. This makes the task of dataset discovery harder as it may not be sufficient to search for datasets only by classical keyword-based search. More sophisticated approaches which are able to search for similar or related datasets could be helpful in these scenarios. As shown by Chapman *et al.* (2020) and Miller *et al.* (2018), many existing dataset discovery solutions are based on simple keyword query search. This is what is typically implemented in open data portals such as NODC or EDP. There are also open data portal mash-ups. For example, EDP collects metadata records about open datasets from national portals of individual member states and provides search features across the whole European Union. Recent works further extend these basic approaches. Brickley *et al.* (2019) describe Google Dataset Search. The authors explain in the paper how dataset metadata is crawled from the web and cleansed. The metadata is then mapped to the Google's knowledge graph, which is then used for dataset duplicates detection and for dataset discovery. Chen *et al.* (2020) enrich metadata records with labels based on the dataset content. Chapman *et al.* (2020) describe the whole dataset discovery process comprising querying for datasets, query processing resulting in a list of datasets, result handling and its presentation. It also surveys recent techniques for these individual steps.

2.3 Similarity-based dataset discovery techniques

In this paper, we are interested in similarity-based dataset discovery techniques. Dataset similarity can be used either during query processing where the query result can be enriched with similar datasets or for result handling and presentation where similarity of retrieved datasets can be used to group datasets in the presentation or to enable exploration of retrieved datasets in case of large results. For example, EDP offers a discovery feature based on dataset similarity besides the basic keyword query search. For a dataset found by the keyword query search, similar datasets are also offered to the user. According to source code published at GitLab [3], the portal uses TLSH presented by Oliver *et al.* (2013). Firstly, they concatenate the title and description of the dataset. The locally sensitive hash is constructed from the concatenated string, which should produce a similar hash for a similar dataset, and these hashes are compared. Originally, the method was introduced by Dutkowski and Schramm (2015). It was implemented as a technique for searching for duplicate or almost equal datasets, ignoring typing errors.

Similarity-based dataset discovery is discussed in the recent survey by Chapman *et al.* (2020). It discusses techniques to extend a table by discovering tables through table similarity based on tabular schema similarity (e.g. Das Sarma *et al.*, 2012, Yakout *et al.*, 2012) and semantic similarity using embedding approaches (e.g. Zhang and Balog, 2018). These can be considered also as dataset discovery techniques because a table is just a special case of a dataset. Several novel techniques for similarity dataset discovery have been proposed in literature in the last few years. Fernandez *et al.* (2018) propose Aurum. It is a system to build, maintain and query an enterprise knowledge graph (EKG) which represents datasets and their structural elements, for example, table columns, as nodes and relationships between them as edges. A relationship between two structural elements may represent content similarity, schema similarity, for example, similarity of names of the columns, or key/foreign key pairs defined in the dataset schemas. The paper introduces an efficient model which exploits EKG. Moreover, the introduced technique requires only a linear passage through datasets to build EKG. Dataset discovery is then performed on top of EKG. When a user selects a dataset, the tool offers other relevant datasets through the relationships in EKG. Bogatu *et al.* (2020) propose a technique based on content and schema similarity. For schema similarity, the approach considers similarity of column names. For content similarity, the approach considers various similarity models, for example, based on value embedding. Mountantonakis and Tzitzikas (2020) propose union and complement metrics between RDF datasets. The metrics are content-based and computed directly on the RDF triples forming the datasets. Several papers propose dataset similarity techniques based on metadata similarity. Altaf *et al.* (2019) describe a method which enables to measure similarity between datasets on the base of papers citing the datasets and a citation network between datasets. Degbelo and Teka (2019) evaluate four different metadata-based models for searching spatially related datasets, that is, datasets which are related because of the same or similar spatial area covered. The first model is a full-text search model. The second one parses and geocodes user's query. The other two models map user's query to knowledge graphs, WordNet (Fellbaum, 2005) and ConceptNet (Speer *et al.*, 2017), enrich the query with the neighborhoods from these knowledge graphs and use the result for the full-text search.

There are also works which focus on methods useful for datasets published as Linked Data (Berners-Lee, 2006). For example, Mountantonakis and Tzitzikas (2018) introduce content-based metrics for measuring connectivity between datasets using links and shared entities between the datasets. Wagner *et al.* (2014) also use shared entities to define similarity between datasets and extend this approach also to clusters of similar entities. The metrics can be then used also for measuring similarity between datasets. Ellefi *et al.* (2016) present a dataset recommendation approach which identifies linking candidates based on the presence of schema overlap between datasets. Ellefi *et al.* (2016) introduce a dataset recommendation method based on cosine similarity of sets of concepts present in datasets. Similarly, Martins *et al.* (2016) recommend datasets based on the similarity of resource labels present in the datasets. Then they rank the recommended datasets based on their TF-IDF score and their coverage of labels present in the original dataset. Leme *et al.* (2013) present a probabilistic Bayesian classifier for dataset recommendation. Such recommendation techniques can also be used for dataset discovery services as they recommend similar or related datasets.

2.4 Data catalogs and metadata

Any dataset discovery method depends on the ability to find and access available datasets. To make a dataset available and accessible, the current practice is to describe it with metadata which is published in a data catalog. A typical data catalog consists of a database of dataset metadata records and a pair of query interfaces, one for machines, for example, a SPARQL endpoint, and one for humans, for example, a user interface, the latter typically using the former. Examples of such data catalogs are the official portal for European data (EDP) [4], the Czech National Open Data Catalog (NODC) [5] described by Klímek (2019), or the US government's open data portal [6]. In addition, there are many more data catalogs within enterprises. Those typically contain non-open data, but otherwise, they work in an identical way as the open data catalogs.

For dataset metadata, there is the DCAT W3C Recommendation (Browning *et al.*, 2020), a vocabulary specifying the metadata fields and their representation based on RDF (Cyganiak *et al.*, 2014). For data portals within the European Union, there is an application profile of DCAT called DCAT-AP [7], further specifying the metadata fields, improving metadata interoperability. Furthermore, there are additional application profiles of DCAT-AP for individual countries and use cases.

According to DCAT, a dataset is "A collection of data, published or curated by a single agent, and available for access or download in one or more representations [8]." The dataset metadata record according to DCAT contains various kinds of fields. Some fields are textual, such as dataset title, dataset description and keywords describing a dataset. Other fields, such as, in the case of DCAT-AP, update frequency, file format or language used, contain values from code lists from the EU Vocabularies [9].

2.5 Modularity and reusability of existing solutions

Based on the dataset metadata, data catalogs offer dataset search and retrieval functionalities for their users. As the studies mentioned above show, the users typically need to search for more than a single, isolated dataset. Typically, the users wish to find multiple datasets related to each other in some way, and this is where the pure metadata-based search methods present in today's data catalogs come up short. The studies also show that dataset discovery depends on the context of the user's needs and discovery tasks. Therefore, it is not easy to construct a dataset discovery service on top of a single similarity discovery technique. It is necessary to be able to experiment with various combinations of various techniques and compare them. A framework supporting such experimentation, possibly providing predefined components which can be combined and compared, would be helpful for anyone who needs to find a viable solution for their domain and contexts. The framework shall support extracting metadata from data catalogs, processing the extracted metadata and computing similarities between described datasets, and presenting the resulting similarities to human users. The framework shall be modular. It shall provide a set of various components for metadata extraction, metadata processing, similarity measuring and also various human interfaces for presenting the similarities to human users. One could say that any ETL framework shall be sufficient. An ETL framework, as explained by El-Sappagh *et al.* (2011), is a framework for defining data manipulation processes, each starting with extracting data from its sources, its various transformations and loading the result to a database. The database is then used for various purposes, including presentation of the data to human users. However, what is important is not a particular ETL framework but a set of ETL components prepared for a certain task. In our case, this task is dataset discovery and measuring similarity of datasets in particular. From this point of view, a given technique for metadata extraction or similarity computation shall be packed as a component. Moreover, the components need to have standardized and compatible outputs and inputs so that it is possible to combine them and introduce new components for experimentation.

Unfortunately, the only standardization in the existing solutions are the DCAT and DCAT-AP metadata standards described above. The existing techniques for metadata extraction and similarity computation surveyed in the subsections above are isolated and incompatible solutions without open and modular architecture. This makes them hard to combine and compare. To the best of our knowledge, the existing works in the field of dataset discovery do not address this problem, and our paper is the first which addresses the problem of defining such open and modular architecture for dataset discovery. For example, Brickley *et al.* (2019) describe the architecture of the Google dataset search solution. The architecture comprises components for metadata extraction; metadata manipulation including its normalization, cleansing and deduplication; and a user interface for searching datasets using the cleansed metadata. The EDP architecture comprises components for harvesting metadata from national data catalogs in EU countries, components for metadata manipulation including a component for computing dataset similarities using TLSH. Fernandez *et al.* (2018) describe the architecture of Aurum comprising a component for extracting dataset profiles of tabular datasets, a component for building relationships between datasets and a component for querying the resulting search index by human users. We can see that all these works somehow describe a dataset discovery architecture, but none of them defines it as open and modular architecture which enables to reuse and combine various solutions.

3. Architecture of dataset discovery framework

In this section, we introduce our dataset discovery framework. We present its basic concepts, its architecture and conceptual model of its components. As can be seen from the overview of dataset similarity techniques in Section 2, the framework needs to accommodate various solutions, from simple metadata similarity techniques to techniques employing knowledge graphs. The framework must be modular so that it is possible to combine the techniques. To achieve modularity, it is necessary to encapsulate various techniques as framework components with well-defined inputs and outputs.

The basic concepts of our framework are *similarity production pipelines* and *similarity presentations*. A similarity production pipeline is a data processing pipeline which takes a data catalog as an input together with other possible inputs and produces similarity of datasets in the catalog. It comprises various components which either extract data from external sources, process data or present data to end users. A similarity presentation then presents similarity to human users through a user interface. It comprises a presentation component which takes an output of one or more similarity production pipelines as an input. The framework supports experts in building their own similarity production pipelines and similarity presentations. The experts can experiment with various combinations of components suitable for their domain and use cases and compare them.

The architecture is depicted in Figure 1. It comprises three layers of abstraction: pipeline (yellow), catalog (orange) and conceptual (green). The pipeline layer comprises concrete assembled similarity production pipelines, each complemented with a similarity presentation. A similarity production pipeline connects various components together, defines in which order the components are executed and how the components pass their outputs. A similarity presentation then takes the outputs of one or more pipelines and presents them to human users using a presentation component. Two sample similarity production pipelines are outlined in Figure 1. Their components are depicted as blue boxes with rounded corners. Component inputs and outputs are depicted as squares in different colors. Each pipeline ends with a similarity presentation.

The catalog layer comprises a *catalog* of supported components which can be placed in pipelines and used for presentations. A component specifies expected input data, how the input data is processed and what output is provided. The sample catalog depicted in Figure 1 comprises eight different components, each depicted as a blue box with rounded corners and with their expected inputs and their outputs as colored squares. The gray dashed arrows

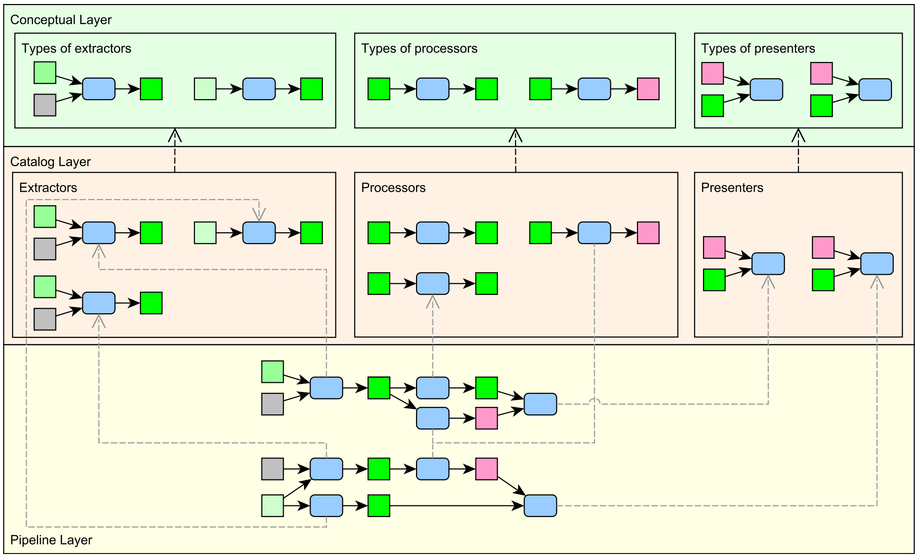

Figure 1.
The architecture of the dataset similarity discovery framework

oriented from the pipeline layer to the catalog layer depict how the individual components chosen from the catalog layer are placed in the similarity pipelines or used for similarity presentation.

The conceptual layer is an abstract layer which defines the *conceptual model* of similarity production pipelines and similarity presentations. The conceptual model defines supported *component types*. Each component in the catalog is an instance of a component type from the conceptual model. A component type specifies an action but it is abstracted from a specific method and specific form of input and output data. The input and output data specification is abstracted to conceptual *entity types* which describe possible inputs and outputs at the conceptual level without technical details. Figure 1 outlines the conceptual model. It shows that the conceptual model defines component types with their input and output entity types. From the left to the right, each component type depicted as a blue box has one or more instances at the catalog layer. The first component type depicted on the left of the conceptual layer has two components as its instances in the catalog layer, the second has one component as an instance and so forth. Each component type instance, that is, a specific component, adheres to its component type. It means that it provides a specific implementation of the action defined by the component type. To perform the action, it takes inputs and provides outputs which adhere to the input and output types, respectively, predefined by the component type.

In this section, we describe each layer in detail. We describe the conceptual model as a fixed set of component types, with fixed set of entity types as their inputs and outputs. We then define how components in a catalog shall be defined. However, we do not define a fixed catalog. A definition of a specific catalog of components is up to a specific implementation of the framework and a given catalog can be arbitrarily extended with new components. However, any component must be an instance of one of the types from the conceptual level which is fixed. This section ends with a formal definition of similarity pipelines and presentations which shows how components from a catalog can be combined together. A specific component catalog, specific similarity pipelines and presentations and how they can be executed are presented in Section 4.

3.1 Conceptual layer

The conceptual layer defines the conceptual model for similarity production pipelines and similarity applications. It ensures compatibility of components in pipelines as it defines supported entity and component types. Each component type represents some action with a given input and output. This defines not only supported actions but also possible ordering of components in pipelines because it is necessary that their inputs and outputs are compatible.

3.1.1 Entity types. An entity type represents types of data entities handled during similarity production pipelines execution. The conceptual model of possible entity types is depicted in Figure 2. We distinguish the following entity types in this paper:

(1) `Dataset` represents a dataset D in a collection of datasets $\{D_1, \ldots, D_n\}$ or shortly $\{D_i\}_1^n$ or $\{D_i\}$ when n is not important. The goal of a similarity production pipeline is to produce similarities of datasets in $\{D_i\}$ *among each other*.

(2) `Knowledge` represents external knowledge which is used in a similarity production pipeline. We distinguish three subtypes in this paper:

- `KnowledgeGraph` represents external knowledge structured as a graph $G = (N, E)$ where N is a set of nodes, $E \subseteq N \times P \times (N \cup L)$ is a set of edges, P is a set of all possible node properties and L is a set of all possible literal values. A literal value is simply a string. For an edge (n, p, o), n is called subject, p is called predicate and o is called object. An edge (n, p, o) s.t. $o \in N$ is called object edge.

An edge (n, p, o) s.t. o ∈ L is called literal edge. A reader may notice that the knowledge graph model corresponds to the RDF data model (Cyganiak et al., 2014), but this is not important at the conceptual layer.

- `NodeEmbedding` represents external knowledge structured as a node embedding $\mathscr{N}_G : N \rightarrow \mathbb{R}^n$ of a knowledge graph G where N is the set of nodes of G.
- `WordEmbedding` represents external knowledge structured as a word embedding $\mathscr{W} : L \rightarrow \mathbb{R}^n$ where L is the set of all possible literal values.

(3) `Descriptor` represents a descriptor of a dataset D. A descriptor of D is any collection of data which describes D, for example, vector, a histogram, time-series, a set of descriptors or a combination of other dataset descriptors. We distinguish three subtypes:

- `Metadata` represents a descriptor which describes D using metadata. Currently, we consider only one possible kind of metadata descriptors which correspond to the DCAT standard (see Section 2.4).
- `Mapping` represents a descriptor which describes D by mapping its representation $\{d^1, \ldots, d^m\}$ to an external knowledge. A representation $\{d^1, \ldots, d^m\}$ of D is any set of elements which characterize D. It is a generic concept which may comprise elements of the data schema of D, all or chosen data elements from the content of D, and it can also be D itself. A mapping of D to an external knowledge comprises particular mappings of the elements of the representation.

  If the external knowledge is a knowledge graph *G,* then the mapping is a set $\{(D, d^i, \{n^i_1, \ldots, n^i_{k_i}\})\}^m_{i=1}$ which maps each element d^i of a representation $\{d^1, \ldots, d^m\}$ of *D* to a set of nodes $\{n^i_1, \ldots, n^i_{k_i}\}$ from *G*.

  If the external knowledge is an embedding *E,* then the mapping is a set $\{(D, d^i, \{v^i_1, \ldots, v^i_{k_i}\})\}^m_{i=1}$ which maps each element d^i of a representation $\{d^1, \ldots, d^m\}$ of *D* to vectors $\{v^i_1, \ldots, v^i_{k_i}\}$ where each vector is a result of applying the embedding *E* and some mapping logic represented by a concrete mapping component.
- `SimilarityRanking` represents a descriptor which describes D in a collection $\{D_i\}^n_1$ using similarity of D with other datasets in $\{D_i\} \setminus \{D\}$ as a pair $(D, \{(D_1, s_1), \ldots, (D_n, s_n)\})$ where s_j = similarity(D, D_j) for some function measuring similarity of datasets.

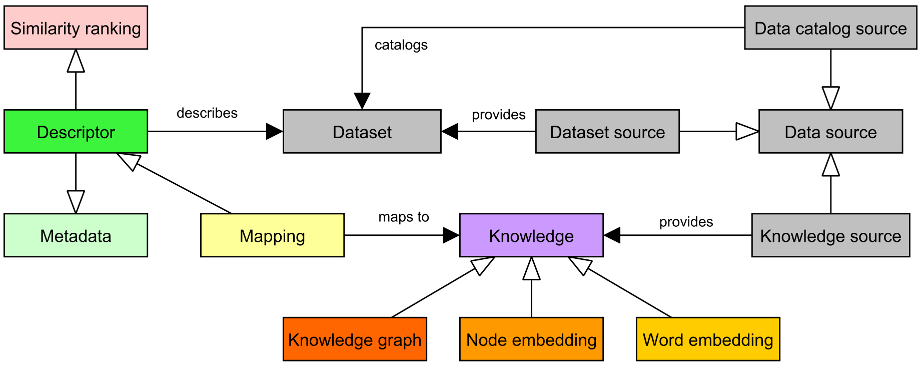

Figure 2.
The conceptual model of possible entity types for similarity production pipelines

(4) `DataSource` represents an external source of a certain kind of data. An instance of a data source type is a concrete data source which can be accessed through an API to extract data. We distinguish the following data source types.

- `DataCatalogSource` represents a data catalog source $\mathrm{DS}_{\mathrm{cat}}$ which provides an API for extracting metadata descriptors. Currently, we consider only data catalog sources providing DCAT metadata descriptors through a SPARQL endpoint or for a bulk download.
- `KnowledgeSource` represents an external knowledge source DS_{k} which provides an API for extracting external knowledge. It can be a SPARQL endpoint for extracting an RDF knowledge graph, a bulk download of a word embedding or a set of documents on which a word embedding can be trained.
- `DatasetSource` represents a datasets source DS_{d} which enables downloading a dataset described by a metadata descriptor. It can simply be the web where the metadata descriptor provides a URL for downloading the content of the dataset.

3.1.2 Component types. A component type represents an operation. It is abstracted from a concrete algorithm for that operation and its implementation. The concrete algorithm and implementation is provided by a concrete component which is an instance of the component type.

Formally, an operation $\mathit{operation}(\mathtt{in}_1, \ldots, \mathtt{in}_m) : \mathtt{out}$ takes input parameters $\{\mathtt{in}_1, \ldots, \mathtt{in}_m\}$ and produces an output `out`. An input $\mathtt{in}_j$ is a regular expression $\mathtt{T}$, $\mathtt{T}^+$, $\mathtt{T}^*$ or $\mathtt{T}^?$ meaning that one instance, one or more instances, zero or more instances, or zero or one instance of an entity type `T` is provided on the input. The same is for the operation output.

The top-level component type is `Component`. It represents all possible components. There are three more specific subtypes: `Extractor`, `Processor` and `Presenter`. The full list of supported component types with their inheritance hierarchy is shown in Figure 3.

3.1.2.1 Extractors. `Extractor` represents an extractor. An extractor is a component which performs an operation of extracting data from a data source. The conceptual model of extractors is shown in Figure 4. We distinguish the following subtypes of `Extractor`:

(1) `ExtractExternalKnowledge` represents components performing

$$\mathit{extract}_k(\mathtt{KnowledgeSource}) : \mathtt{Knowledge}$$

which extracts an external knowledge from a given external knowledge source.

(2) `ExtractMetadataDescriptor` represents components performing

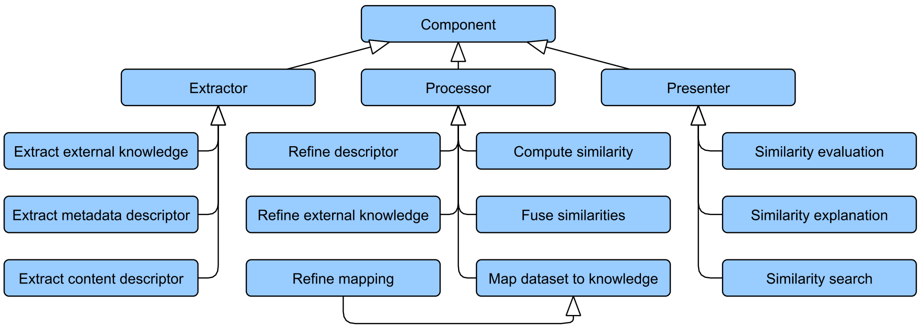

Figure 3.
The conceptual model of possible components types for similarity production pipelines

$$\textit{extract}_{\textit{meta}}(\texttt{DataCatalogSource}):\texttt{Metadata}^{+}$$
which extracts metadata about datasets from a given data catalog source.

(3) `ExtractContentDescriptor` represents components performing

$$\textit{extract}_{\textit{desc}}(\texttt{DatasetSource},\texttt{Metadata}^{+}):\texttt{Descriptor}^{+}$$
which extracts from a dataset source a descriptor for each dataset with a provided metadata descriptor.

3.1.2.2 Processors. `Processor` represents a processor. A processor is a component which performs an operation for processing input entities to output entities. The conceptual model of processors is shown in Figure 5.

(1) `RefineDescriptor` represents components performing

$$\textit{process}_{\textit{refd}}(\texttt{Descriptor}^{+}):\texttt{Descriptor}^{+}$$
which transforms input descriptors to output descriptors.

(2) `RefineExternalKnowledge` represents components performing

$$\textit{process}_{\textit{refk}}(\texttt{Knowledge}):\texttt{Knowledge}$$
which transforms an external knowledge to another external knowledge, for example, transforming literal objects of literal edges to other literal objects in a knowledge graph, removing some object edges from a knowledge graph or applying some vector operation on the vectors in an embedding.

(3) `MapDatasetToKnowledge` represents components performing

$$\textit{process}_{\textit{map}}(\texttt{Descriptor}^{+},\texttt{Knowledge}):\texttt{Mapping}^{+}$$
which maps a set of datasets described by the input descriptors to an input external knowledge. The resulting output mapping is created on the base of the input descriptors, and it maps each dataset to the external knowledge.

(4) `RefineMapping` represents components performing

$$\textit{process}_{\textit{refm}}(\texttt{Mapping}^{+},\texttt{Descriptor}^{*},\texttt{Knowledge}^{?}):\texttt{Mapping}^{+}$$
which transforms an input mapping of datasets to an external knowledge to another mapping of the same datasets to the same external knowledge or another external knowledge specified as an optional input. The new mapping is created using the optional input descriptors of the datasets.

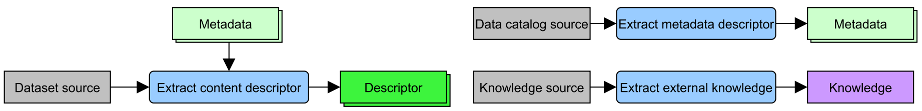

Figure 4. The conceptual model of possible extractor component types for similarity production pipelines

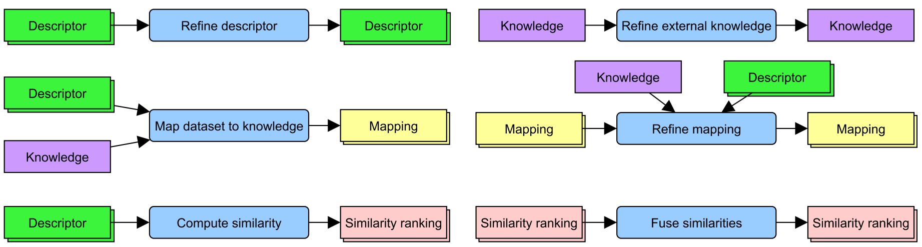

Figure 5. The conceptual model of possible processor component types for similarity production pipelines

(5) `ComputeSimilarity` represents components performing

$$process_{sim}(\texttt{Descriptor}^{+}) : \texttt{SimilarityRanking}^{+}$$

which computes a similarity of each dataset described by an input descriptor with other datasets described by the other input descriptors. The component uses the input descriptors to compute the similarity.

(6) `FuseSimilarities` represents components performing

$$process_{fuse}(\texttt{SimilarityRanking}^{+}) : \texttt{SimilarityRanking}^{+}$$

which performs multimodal fusion of two or more input similarities for a dataset to a single similarity. For each dataset with specified similarities on the input, an output similarity is produced.

3.1.2.3 Presenters. `Presenter` represents a presenter. The conceptual model of presenters is shown in Figure 6. A presenter is a component which uses the products of dataset similarity pipelines to present them to human users and provide them with some functionality.

(1) `SimilarityEvaluation` represents components performing

$$present_{eval}(\texttt{SimilarityRanking}^{+}, \texttt{Descriptor}^{+})$$

which presents dataset similarities to a human user who evaluates the similarities. It uses input descriptors of the datasets for presenting the datasets.

(2) `SimilaritySearch` represents components performing

$$present_{sear}(\texttt{SimilarityRanking}^{+}, \texttt{Descriptor}^{+})$$

which enables a human user to choose a dataset and then shows datasets similar to the chosen one on the base of the input similarities. It uses input descriptors of the datasets for presenting the datasets.

(3) `SimilarityExploration` represents components performing

$$present_{expl}(\texttt{SimilarityRanking}^{+}, \texttt{Descriptor}^{+}, \texttt{Knowledge}^{*})$$

which explains similarities between datasets to a human user. For the explanation, it uses the input similarity, dataset descriptors and optionally also external knowledge of different kinds. Among the input descriptors, there are also mappings of the datasets to the external knowledge if provided on the input. Descriptors may be used for presentation as well as explanation purposes.

3.2 Catalog layer

The catalog layer contains the catalog of components experts may use to build their similarity production pipelines and similarity presentations. Each component `c` in the catalog is an instance of a component type `T` from the conceptual level which we write as `c` ∈ `T`. A component `c` is described in the catalog with a record using the following structure:

type: specifies the type of `c` from the conceptual layer

input: specifies the inputs of `c` from the conceptual layer using the entity types from the conceptual layer

output: specifies the outputs of `c` from the conceptual layer using the entity types from the conceptual layer

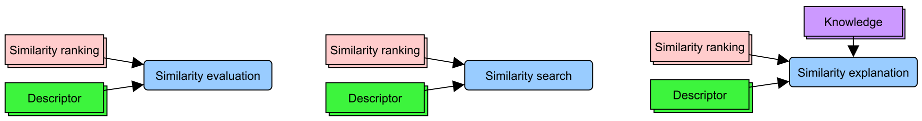

Figure 6. The conceptual model of possible presenter component types for similarity production pipelines

description: describes verbosely the behavior of `c`

implementation: specifies a link or links to the source code and documentation of one or more implementations of `c`

configurable: specifies whether and how the behavior of `c` can be configured

configurations: if `c` is configurable there is a list of predefined configurations; for each predefined configuration, a name and configuration specification is provided

As can be seen from the structure, a component has one or more executable implementations. The implementations consume the same conceptual entities on the input, produce the same conceptual entities on the output but they technically differ in data formats used to express the inputs and outputs. Furthermore, a component's behavior can be further specified by configuring it. In case of a configurable component, some concrete configurations may be predefined in the catalog. These configurations are then used by their name in pipeline specifications in the next section which makes pipeline specifications clearer.

3.3 Pipeline layer

The components in the catalog layer represent concrete operations with their concrete implementations. The pipeline layer contains concrete similarity production pipelines and similarity presentations which put components together. In this section, we introduce a formal algebraic model of similarity production pipelines and similarity presentations. A pipeline expressed using the algebraic model cannot be directly executed. The algebraic expression must be interpreted and translated to an executable script as described in Section 4.4. The algebraic model enables one to define a pipeline without implementation details so that it is more suitable for comparing different pipelines. Using the algebraic model, it is easier to see what are the conceptual differences between given pipelines.

A *pipeline fragment* F is an expression.

(1) $\mathtt{c}(F_1, \ldots, F_m)$ where $\mathtt{c} \in \mathtt{T}$ is a component of type T performing an operation $operation(\mathtt{in}_1, \ldots, \mathtt{in}_m) : \mathtt{out}$ and $F_1, \ldots, F_m$ are pipeline fragments s.t. $\forall j \in \{1, \ldots, m\}$ F_j is compatible with the input parameter $\mathtt{in}_j$ of *operation*. F_j is *compatible* with an input parameter in_j of a component type T iff the type of the result of execution of F_j is the same as the type of in_j.

(2) $F_1 \cup F_2$ where F_1, F_2 are pipeline fragments with their outputs being of the same entity type, or

(3) `s` where $\mathtt{s} \in \mathtt{DataSource}$.

An *execution* of $F = \mathtt{c}(F_1, \ldots, F_m)$ means executing the operation defined by `c` with the results of executing $F_1, \ldots, F_m$ passed as parameters. An execution of $F = F_1 \cup F_2$ means creating the set union of the results of executing F_1 and F_2. An execution of $F = \mathtt{s}$ where $\mathtt{s} \in \mathtt{DataSource}$ is undefined.

A pipeline fragment F is a *similarity production pipeline* iff $F = \mathtt{c}(F_1, \ldots, F_m)$ where $\mathtt{c} \in \mathtt{ComputeSimilarity} \cup \mathtt{FuseSimilarity}$. A pipeline fragment F is a *similarity presentation* iff $F = \mathtt{c}(F_1, \ldots, F_m)$ where $\mathtt{c} \in \mathtt{Presenter}$.

The introduced algebraic model does not allow for specification of component configurations. If a component described in the catalog is configurable, its configurations must be defined and named in the catalog, and only these named configurations can be used in pipeline specifications.

4. Dataset discovery framework proof-of-concept

In this section, we demonstrate the dataset discovery framework on a concrete component catalog and similarity production pipelines constructed using the components from the catalog. To demonstrate the framework, we apply the pipelines on a specific open data catalog and external knowledge. We also show examples of similarity presenters which we used for user evaluation of the resulting similarities, for example in Škoda *et al.* (2020).

4.1 Data and external knowledge used for proof-of-concept

For the framework demonstration purposes, we work with metadata from the Czech National Open Data Catalog (NODC) [10] described in Klímek (2019), which is regularly harvested by the European Data Portal (EDP). This metadata describes open datasets published by public institutions in Czechia, such as the Czech Statistical Office, the city of Prague or the Czech Social Security Administration. For illustration of how the framework can be used, we use the most basic, textual metadata fields: *title*, *description* and *keywords* – which contain textual descriptions of the datasets in Czech. The collection we work with contains approximately 6,600 datasets from 39 publishers.

Besides dataset metadata, we utilize multiple types of external knowledge. We choose to employ Wikidata (Vrandečić and Krötzsch, 2014), a collaboratively edited knowledge-base with free access, as a knowledge graph source. In general, the Wikidata model is built around the entities and their relations. The entities may represent concepts or real objects or people. The relations are of various types, but for the framework demonstration we utilize only instance of [11] and subclass of [12] relations. We have also used this model for computing Node2Vec (Grover and Leskovec, 2016) text (labels) and concept (nodes) embeddings with different hyperparameters. For comparison, we utilize two standard word embeddings: Word2Vec (Mikolov *et al.*, 2013) model trained on Czech Wikipedia articles and BERT (Devlin *et al.*, 2018) using the *BERT-base, multilingual cased* model.

Formally, we define the following entities (see Section 3.1.1), which will be used in the proof of concept by components in pipelines:

(1) `NODC`

 type: `CatalogDataSource`

 description: Snapshot of the DCAT-AP metadata dump from the Czech National Open Data Catalog in the RDF TriG (Carothers and Seaborne, 2014) file format from 2020–04–20. For practical reasons, datasets of the State Administration of Land Surveying and Cadastre were omitted. It was approximately 120,000 datasets with very similar metadata, which did not add any value for the purpose of the framework demonstration, but unnecessarily increased the time and hardware requirements of the demonstration.

 download: `2020.04.20-data.gov.cz-no-cuzk.trig` [13] (Klímek and Škoda, 2021a).

(2) `W2V [CSWiki]`

 type: `KnowledgeSource`

 description: Word2Vec model for word embedding trained on Czech Wikipedia articles.

 download: cswiki-latest-pages-articles.word2vec [14] (Bernhauer and Skopal, 2020).

(3) `N2V [Wikidata Labels/80/40]`

 type: `KnowledgeSource`

 description: Node2Vec model for word embedding trained on Czech labels of items from the Wikidata knowledge graph, using the `instance of` and `subclass of` directed edges. For this model, length of random walk is 80 nodes and number of random walks per node is 40.

download: `labels.80.40.d` [15] (Bernhauer and Skopal, 2021d).

(4) `N2V [Wikidata Labels/160/40]`

type: `KnowledgeSource`

description: Node2Vec model for word embedding trained on Czech labels of items from the Wikidata knowledge graph, using the `instance of` and `subclass of` directed edges. For this model, the length of the random walk is 160 nodes and the number of random walks per node is 40.

download: `labels.160.40.d` [16] (Bernhauer and Skopal, 2021c).

(5) `N2V [Wikidata KG/80/40]`

type: `KnowledgeSource`

description: Node2Vec model for node (concept) embedding trained on the Wikidata knowledge graph using the `instance of` and `subclass of` directed edges. For this model, the length of the random walk is 80 nodes and the number of random walks per node is 40.

download: `concepts.80.40.d` [17] (Bernhauer and Skopal, 2021b).

(6) `N2V [Wikidata KG/40/10]`

type: `KnowledgeSource`

description: Node2Vec model for node (concept) embedding trained on the Wikidata knowledge graph using the `instance of` and `subclass of` directed edges. For this model, the length of the random walk is 40 nodes and the number of random walks per node is 10.

download: `concepts.40.10.d` [18] (Bernhauer and Skopal, 2021a).

(7) `BERT`

type: `KnowledgeSource`

description: BERT is pretrained model for NLP tasks presented in Devlin *et al.* (2018). All pretrained models are officially available at `https://github.com/google-research/bert`. In our experiments, we have used BERT-base, multilingual cased model.

download: `multi_cased_L-12_H-768_A-12.zip` [19] (Devlin et al., 2018).

(8) `Wikidata knowledge graph`

type: `KnowledgeSource`

description: Dump of Wikidata as available at `https://dumps.wikimedia.org/other/wikidata/`. The dump contains information about Wikidata entities and their relations.

download: 20181 217.json.gz [20] (Klímek and Škoda, 2021b).

4.2 Proof-of-concept component catalog

The full list of available components and links to their implementations can be found in Appendix A. In this section, we list a subset of the components, which is used later by the selected similarity production pipelines in Section 4.3. The components are color-coded according to the type of their outputs: knowledge (violet), SimilarityRanking (pink), descriptor (green) and mapping (orange).

4.2.1 Extractors.

- `extractMetadata`

  type: `ExtractMetadataDescriptor`

  input: `catalog` $\in$ `DataCatalogSource`

  output: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  description: Extracts metadata descriptors for all datasets from a given catalog. An extracted descriptor contains a title, description and keywords of a dataset if provided by the catalog.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/extractors/extract-metadata-descriptor/dcat-ap-extractor`

- `getVectorEmbeddingModel`

  type: `ExtractExternalKnowledge`

  input: `knowledgesource` $\in$ `KnowledgeSource`

  output: `knowledge` $\subset$ `Knowledge`

  description: Extracts external knowledge as an embedding model (i.e. Word2Vec, Node2Vec, BERT, …).

  implementation: There is no implementation as these models can be just downloaded.

  configurable: Kind of embedding used.

  configurations:

  `getWord2VecModel` - Word2Vec embedding in Gensim format

  `getNode2VecModel` - Node2Vec embedding in Gensim format

- `wikidataHierarchyExtractor`

  type: `ExtractExternalKnowledge`

  input: `knowledgesource` $\in$ `KnowledgeSource`

  output: `knowledge` $\subset$ `Knowledge`

  description: Extracts hierarchy, made of `instance of` and `subclass of` edges, from the Wikidata knowledge graph.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/extractors/extract-external-knowledge/wikidata-hierarchy-extractor`

- `wikidataLabelsExtractor`

  type: `ExtractExternalKnowledge`

  input: `knowledgesource` $\in$ `KnowledgeSource`

  output: `knowledge` $\subset$ `Knowledge`

  description: Extracts item labels and aliases from the Wikidata knowledge graph.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/extractors/extract-external-knowledge/wikidata-labels-extractor`

4.2.2 Processors.

- `projectDescriptor`

  type: `RefineDescriptor`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  output: $\{\texttt{descriptor'}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  description: Performs property projection on input descriptors.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/refine-descriptor/json-to-csv`

  configurable: Projected metadata property.

  configurations:

  `projectToTitle` - `title` property

  `projectToDescription` - `description` property

  `projectToKeywords` - `keywords` property

- `concatenate`

  type: `RefineDescriptor`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  output: $\{\texttt{descriptor'}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  description: Refines two textual descriptors into one concatenated descriptor.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/refine-descriptor/join`

- `doLemmatization`

  type: `RefineDescriptor`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  output: $\{\texttt{descriptor'}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$

  description: Refines a textual descriptor by lemmatisation, i.e. transforms each word into its lemma and eliminates stop-words. In this case, the lemmatisation and optional stop-word elimination could be separated, but it is more convenient work with NLP processor just once.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/refine-descriptor/udpipe`

- `mapToAverageVector`

  type: `MapDatasetToKnowledge`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$, $\texttt{knowledge} \subset \texttt{Knowledge}$

  output: $\{\texttt{mapping}_i\}_{i=1}^{n} \subset \texttt{Mapping}^{+}$

description: Maps textual metadata to provided embedding using the average vector over all words.

implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/map-dataset-to-knowledge/vectorize`

configurable: Kind of descriptor used.

configurations:

`mapTextToAverageVector` - textual descriptor

`mapConceptsToAverageVector` - Wikidata concepts

- `instanceToClass`

  type: `RefineMapping`

  input: $\{\texttt{mapping}_i\}_{i=1}^{n} \subset \texttt{Mapping}^{+}$, $\texttt{knowledge} \subset \texttt{Knowledge}$

  output: $\{\texttt{mapping}_i\}_{i=1}^{n} \subset \texttt{Mapping}^{+}$

  description: Map entities to their `instance of` ancestors.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/refine-mapping/instance-to-class`

- `bagOfWordsMapper`

  type: `MapDatasetToKnowledge`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Metadata}^{+}$, $\texttt{knowledge} \subset \texttt{Knowledge}$

  output: $\{\texttt{mapping}_i\}_{i=1}^{n} \subset \texttt{Mapping}^{+}$

  description: Maps textual metadata to provided textual entities.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/map-dataset-to-knowledge/bag-of-words-mapper`

- `computeSimilarity`

  type: `ComputeSimilarity`

  input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Descriptor}^{+}$

  output: $\{\texttt{similarityRanking}_j\}_{j=1}^{n} \subset \texttt{SimilarityRanking}^{+}$

  description: Computes similarity for scalar-based descriptors such as texts, sets of words or vectors.

  implementation: `https://github.com/mff-uk/simpipes-components/tree/main/processors/compute-similarity/basic`

  configurable: Kind of similarity used.

  configurations:

  `computeJaccardSimilarity` - Jaccard distance, usable for texts and sets of words

  `computeCosineSimilarity` - Cosine distance, usable for texts and vectors

  `computeTLSHSimilarity` TLSH distance, usable for texts

4.2.3 Presenters.

(1) `evaluateExactSize`

type: `SimilarityEvaluation`

input: $\{\texttt{similarityRanking}_j\}_{j=1}^{n} \subset \texttt{SimilarityRanking}^{+}$, $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Descriptor}^{+}$

description: Evaluates similarity using the provided baseline. For every baseline's dataset, the k`NN` query (Papadias, 2009) similarity search is performed. k is the expected number of similarity datasets. Average ratio of observed and expected is presented to the user.

implementation: https://github.com/mff-uk/simpipes-components/tree/main/presenters/similarity-evaluation/exact-size

(2) `evaluateTopK`

type: `SimilarityEvaluation`

input: $\{\texttt{similarityRanking}_j\}_{j=1}^{n} \subset \texttt{SimilarityRanking}^{+}$, $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Descriptor}^{+}$

description: Evaluates similarity using the provided baseline. For every baseline's dataset, the k`NN` query similarity search is performed. k is specified by user through parameter, and it is a constant for each test case. Average ratio of observed and expected is presented to the user.

Implementation: https://github.com/mff-uk/simpipes-components/tree/main/presenters/similarity-evaluation/top-k

(3) `evaluatePRCurve`

type: `SimilarityEvaluation`

Input: $\{\texttt{similarityRanking}_j\}_{j=1}^{n} \subset \texttt{SimilarityRanking}^{+}$, $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Descriptor}^{+}$

Description: Evaluates similarity using the provided baseline. For every baseline's dataset, the 11-point PR curve (Zhang and Zhang, 2009) is computed and presented to the user.

implementation: https://github.com/mff-uk/simpipes-components/tree/main/presenters/similarity-evaluation/pr-curve

(4) `OpenDataInspectorEvaluation`

type: `SimilarityEvaluation`

input: $\{\texttt{descriptor}_i\}_{i=1}^{n} \subset \texttt{Descriptor}^{+}$, $\{\texttt{similarityRanking}_j\}_{j=1}^{n} \subset \texttt{SimilarityRanking}^{+}$

description: OpenDataInspector is a standalone tool. The evaluation module (see Škoda *et al.*, 2020) allows domain experts to evaluate similarity production pipeline results.

implementation: https://github.com/mff-uk/simpipes-components/tree/main/presenters/similarity-evaluation/odin-similarity

4.3 Proof-of-concept pipelines

In this section, we present four similarity production pipelines as examples. The full list of currently available pipelines can be found in Appendix B. The pipelines demonstrate the practical usability of our framework. For example, the first pipeline presented in Section 4.3.1 is an implementation of the solution employed by the EDP (see Section 2). In Section 2, we also discussed discovery solutions which employed an external knowledge in a form of a knowledge graph, for example, Brickley *et al.* (2019), The fourth pipeline presented in Section 4.3.4 is an implementation of a dataset discovery solution employing an external knowledge in a form of a knowledge graph. In our case, the external knowledge is the Wikidata knowledge graph. A similar solution is employed by Google in their dataset search architecture with their own knowledge graph as described by Brickley *et al.* (2019). Various pipelines implementing existing as well as novel solutions can be constructed in our framework. The results of similarity production pipelines can be used, for instance, for their evaluation, both automatically and with assistance of domain experts. Later, in Section 4.5, we show the usage of appropriate presenters.

4.3.1 TLSH similarity production pipeline. TLSH similarity production pipeline corresponds to the similarity feature implemented by the EDP. Note that `computeTLSHSimilarity` can be split into hash computation and similarity function computation, but in this case, these two parts are interconnected.

```
computeTLSHSimilarity(
  concatenate(
    projectToTitle( extractMetadata(NODC) ),
    projectToDescription( extractMetadata(NODC) )
  )
)
```

4.3.2 Metadata-based similarity production pipeline. The metadata-based similarity production pipeline is one of the most straightforward pipelines. It relies on suitable usage of dataset metadata such as title, keywords and description. The metadata is pre-processed by lemmatization and by removing stop-words. Then, it is compared by the Jaccard similarity function.

```
computeJaccardSimilarity(
  doLemmatization(
    projectToTitle( extractMetadata(NODC) )
  )
)
```

4.3.3 Text-based similarity production pipeline using Word2Vec embedding. This similarity production pipeline is an example of text embedding used as external knowledge. The lemmatization phase can be followed by various embedding mappings. In this case, we use a mapping to a Word2Vec model trained on Czech Wikipedia articles. It helps to deal with synonyms and words with similar meaning.

```
computeCosineSimilarity(
  mapTextToAverageVector(
    doLemmatization(
      projectToTitle( extractMetadata(NODC) )
    ),
    getWord2VecModel(W2V [CSWiki])
  )
)
```

4.3.4 Concept-based similarity production pipeline with additional external knowledge. In some cases, it can be useful to enhance a similarity model with additional external knowledge. For example, one part of external knowledge can provide information about mapping words to concepts. Another part can embed the concepts into a vector space. In our case, both come from the same database (i.e. Wikidata), but they are processed in different ways. Firstly, we have mapped words and phrases to Wikidata concepts. Secondly, we have trained a Node2Vec model using the Wikidata knowledge graph and applied a concept-to-vector embedding. In contrast with the Word2Vec embedding, the external knowledge in this pipeline is built using the Wikidata concept hierarchy instead of the natural language processing of the Wikipedia articles.

```
computeCosineSimilarity(
  mapConceptsToAverageVector(
    instanceToClass(
      bagOfWordsMapper(
        projectToTitle( extractMetadata(NODC) ),
        wikidataLabelsExtractor(Wikidata knowledge graph)
      ),
      wikidataHierarchyExtractor(Wikidata knowledge graph)
    ) ∪
    instanceToClass(
      bagOfWordsMapper(
        projectToDescription( extractMetadata(NODC) ),
        wikidataLabelsExtractor(Wikidata knowledge graph)
      ),
      wikidataHierarchyExtractor(Wikidata knowledge graph)
    ) ∪
    instanceToClass(
      bagOfWordsMapper(
        projectToKeywords( extractMetadata(NODC) ),
        wikidataLabelsExtractor(Wikidata knowledge graph)
      ),
      wikidataHierarchyExtractor(Wikidata knowledge graph)
    ),
    getNode2VecModel(N2V [Wikidata KG/40/10])
  )
)
```

4.4 Pipeline implementation

In the previous section, we showcased a few selected pipeline definitions. The definitions capture all important information that is relevant in order to compare different pipelines. However, as the pipelines are described using the catalog layer model (see Section 3.2), additional steps must be carried out to get an executable pipeline implementation. In the rest of this section, we describe what a user needs to do in order to obtain an executable version of a pipeline in a step-by-step fashion. In addition, we provide an example of each step motivated by the pipeline described in Section 4.3.3.

The first step is to employ the component catalog (see Section 4.2), resolve component configurations and obtain links to component implementations in our GitHub repository: `https://github.com/mff-uk/SimPipes-Components`. In the component repository, each component has an implementation, input and output data sample and a README.md file with details on how to use the component. The component description consists of: textual description, requirements, input description, output description, configuration and example execution script.

For example, the `projectDescriptor` component performs a projection of a given property as described in the initial part of the `README.md`. However, the component implementation also changes the data format from JSON to CSV, as can be seen from the following input and output descriptions:

```
Format: Directory of JSON files.
Contents: Dataset descriptor.
Sample: Input sample

Format: CSV file.
Contents: CSV with iri and required property.
Sample: Output sample
```

Each input or output description consists of format specification, human readable description of the content and link to a data sample.

Another important part of the description is the component implementation configuration. The configuration is used not only to set the inputs and outputs of components, but also to provide additional parameters to the implemented algorithms.

```
input - Path to datasets descriptor files.
output - Path to output file.
property - Name of property to project.
linePerValue - For each value create a new line.
```

The last part of the component description is an example of the execution script. Note that the name of the component entry script, `json-to-csv.py`, does not have to correspond to the component name, `JSON To CSV`, nor to the component type name, `refine-descriptor`.

```
python3 json-to-csv.py \
    --datasets ./input-sample/datasets \
    --output ./output/output.csv \
    --property title
```

The next step is to create a component instance configuration. Most of the time, this needs to be done manually, as the user needs to understand the configuration description in the component catalog Section 4.2 and create a corresponding configuration for the component instance based on the `README.md` file in the component repository.

For example, `projectToTitle` is a named configuration of the above-mentioned `projectDescriptor` component. The configuration description of `projectToTitle` states that the projected property is set to `title`. From the description configuration, the user should be able to figure out that this can be archived by setting the `property` configuration to `title`. As `title` is already in the example, the `property` argument remains the same here. The `input` and `output` arguments of the script should be changed to match the rest of the pipeline.

Once the configurations are ready, the user can use them to obtain commands that can be used to execute the given components, and put those commands into a script that forms a backbone of the pipeline implementation. However, as the pipelines algebra captures the pipeline at the conceptual level, it leaves out some implementation details like utility components.

Utility components are stored in the `processors/utilities` directory in the repository. They do not change the contents or meaning of the data passed among the individual components in the pipeline, but they might be necessary for operations like data format changes, which make the inputs and outputs of the individual components in the pipeline compatible. Therefore, the user now needs to take a look at the implementation pipeline backbone and possibly insert necessary utility components.

A good example of the necessity to use a utility component is the pipeline union described in Section 3.3 and used in Section 4.3.4. The pipeline produces several different descriptors by using the `instanceToClass` component on descriptors computed from title, description and keywords. In the next step, all the data should be put together by union. While the union has no counterpart in the component catalog, there is a component in the component repository that can carry on this operation called `json-union`.

With the complete script to run all the components, the last step is to make sure that requirements of each component are fulfilled. An example of such a script can be found in Appendix C.

Some components may require external data or installationn of additional libraries, which are described in the component's `README.md` file. As the components are written in Python, their dependencies are provided as `requirements.txt` files. The installation of the dependencies is then as simple as running `pip3 install-r requirements.txt` for the used components. Once all the requirements are met and the script is ready, the pipeline can be executed simply by running the script. While the process is relatively straightforward, the user needs to have good knowledge of the available components and their configurations as well as basic knowledge of data processing techniques.

4.5 Proof-of-concept similarity presentation

The possibilities of usage of the results of similarity production pipelines is out of scope of this paper. Nevertheless, we present at least two usage examples: manual and automatic evaluation of results of various pipelines.

4.5.1 Manual evaluation. The manual evaluation process is fully described in Škoda *et al.* (2020), showcasing, among others, the first three similarity production pipelines from Section 4.3. Note that the results of this evaluation led us to later replace the first pipeline (Section 4.3.1) with the last pipeline (Section 4.3.4) in further evaluation, which is used for illustration in this paper.

The core idea of the manual evaluation protocol is to employ a set of predefined use cases. Each use case defines a textual description of the user's intent, which is used to set up a scenario in which a user performs the search. In addition, the use case contains a collection of the so-called *starting datasets* – datasets already verified to be relevant to the user and their scenario. The user scenario definition is important as different scenarios may lead to different results even with the same set of starting datasets. For the purpose of the evaluation, all team members followed the same protocol as described in Škoda *et al.* (2020).

We carried out the evaluation on selected pipelines from Section 4.3 using the `OpenDataInspectorEvaluation` presenter:

```
OpenDataInspectorEvaluation(
  extractMetadata(NODC),
  computeJaccardSimilarity(
    doLemmatisation(
      projectToTitle( extractMetadata(NODC) )
    )
  )
)
```

The presenter provides not only an environment to run the evaluation but also the scripts that can be used to gain basics insight into the evaluation and plot the results. In Figure 7, we can see an example of one of those outputs. The values S1, S2 . . . S12 on the *x*-axis represent the different use-cases as defined in Škoda *et al.* (2020). There are results for 10 methods, whose algebraic definitions can be found in Appendix B. To briefly summarize the results, it is clear that there is no one method that outperforms all others in all use-cases, which shows the need for further research into the various types of use cases, various similarity production pipelines and into the problem of how to select the best similarity for a given use case. These findings correspond with those in Škoda *et al.* (2020).

4.5.2 Automatic evaluation. During our previous experiments (Škoda *et al.*, 2020), we quickly found out that the time required for manual evaluation increases rapidly with the increasing number of similarity production pipelines to be evaluated and their configurations such as different embeddings and projections. This led us to investigate the possibilities of

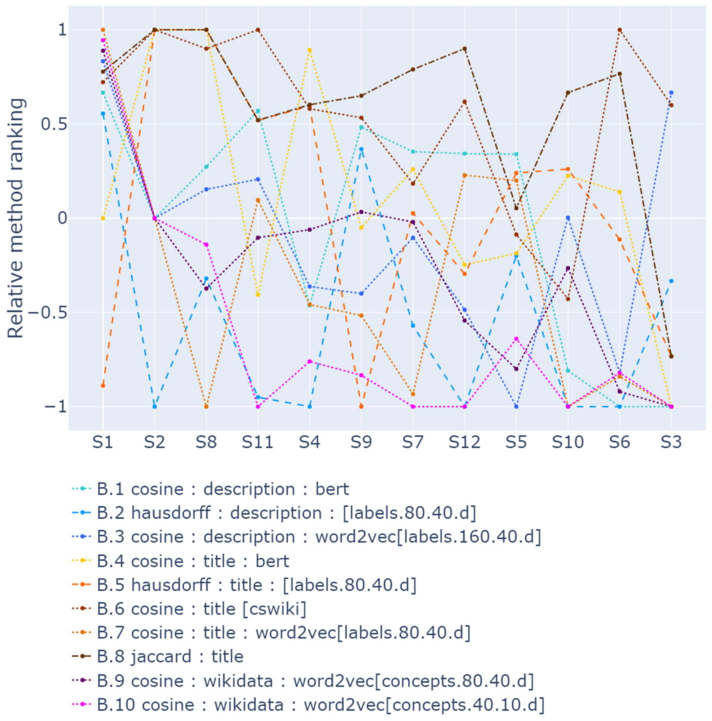

Figure 7.
Normalized average performance per model

automating at least a part of the evaluation process. We devised an automated similarity production pipeline screening process – an automatic evaluation of the success potential of different similarity production pipelines in the form of presenters. These presenters can provide a quick estimate of the pipeline efficiency so that we can focus on a much lower number of candidates for the manual evaluation.

However, the automatic evaluation relies on a user-defined baseline. To obtain the baseline, we manually added the list of all relevant datasets from the given catalog for each use case in Škoda *et al.* (2020).

The `evaluateExactSize` and `evaluateTopK` presenters perform a kNN query search, where k is the number of expected answers, or the specified constant, respectively. For each use-case, the accuracy is computed as the ratio of expected datasets in the result. The presenter `evaluateTopK` reflects more precisely the user-based evaluation as it is not essential to have an exact match. For the users, it is a success if the expected datasets are found in the first few results. The `evaluatePRCurve` presenter provides a more comprehensive view of the issue. Instead of one number, the PR curve says how much effort the users have to make to achieve the required precision.

In our experiments, these methods provided a good estimate for comparing similar pipelines, especially with different kinds of external knowledge. They are also useful for filtering out models with a great number of settings of hyperparameters. Example of results for similarity production pipelines in Appendix B is shown in Table 1. The accuracy found in the automatic evaluation correspondeds very well with manual evaluation. As expected, methods based on word embeddings with a large dictionary (e.g. Word2Vec or BERT based on Wikipedia) have been very successful. Worse, due to the specific domain, were approaches based on smaller databases (e.g. Node2Vec).

Since we already know that different similarity production pipelines are successful for different use cases and the results of the automatic evaluation represent an aggregation of accuracy over all use cases, the results are not conclusive, and therefore we use them only for the initial screening process.

4.5.3 Run-time evaluation. During the testing of various similarity models (pipelines), we can measure pipeline run-time in addition to precision, recall, accuracy and other statistical metrics. We can measure the run-time of entire pipelines, but we can also measure the run-time of individual components due to the modularity of the proposed framework. This can be very useful when comparing two implementations of the same component, where we replace one implementation with another in an identical pipeline.

Table 1 shows the approximate run-time of each pipeline. Individual runs do not use precomputed parts from previous runs but can use parallel processing. The computations were run on a virtual machine running 8 cores with 32 GB of RAM running Ubuntu 21.10. Testing on a virtual machine of course means that the measurements were influenced by

Table 1.
Results of evaluateTopK with $K = 20$ for pipelines presented in Appendix B

Pipeline	Accuracy	Run-time
B.1 cosine: description: bert	10.7%	$\sim 20m$
B.2 hausdorff: description: [labels.80.40.d]	4.0%	$\sim 8m$
B.3 cosine: description: word2vec[labels.160.40.d]	19.4%	$\sim 6m$
B.4 cosine: title: bert	21.2%	$\sim 20m$
B.5 hausdorff: title: [labels.80.40.d]	7.1%	$\sim 5m$
B.6 cosine: title [cswiki]	34.9%	$\sim 4m$
B.7 cosine: title: word2vec[labels.80.40.d]	12.7%	$\sim 3m$
B.8 jaccard: title	32.4%	$\sim 2m$
B.9 cosine: wikidata: word2vec[concepts.80.40.d]	14.3%	$\sim 3h$
B.10 cosine: wikidata: word2vec[concepts.40.10.d]	3.0%	$\sim 7h$

other virtual machines running on the same hardware, and are therefore only very approximate. Since our server does not support GPU acceleration, parts of pipelines B.1 and B.4 were run externally using GPU acceleration within Google Colab service [21].

The longer run-time of pipelines B.9 and B.10 is caused by long-running knowledge extraction components. Those components execution time alone accounts for roughly 3 and 3.5 *h,* respectively. The reasons for such long execution times are the size of the Wikidata knowledge graph dump and the proof-of-concept single threaded implementation. As a result, there is plenty of room for improvement that could significantly speed up the execution.

One of the indisputable advantages of our framework is the ability to use existing outputs from common parts of pipelines. This allows us to test different pipelines that share similar properties efficiently. At the same time, if pipelines are designed appropriately, it is also possible to use parallel processing and, therefore, experimental evaluation of several pipelines simultaneously. As a result, the framework allows efficient experimental evaluation despite the inefficient implementation of components, which is very common in the experimentation phase. Although our proof-of-concept implementation is not scalable in terms of production deployment, scalability of experiments is achieved by reusing already computed parts, which is one of the goals of our framework.

5. Conclusions

In this paper, we have introduced a modular framework for experimentation with dataset discovery methods. We have presented the framework from a software-engineering perspective, providing:

(1) formal conceptual definitions of framework components,

(2) a catalog of ready-to-use component implementations,

(3) production pipelines focusing on similarity-based methods and utilization of external knowledge, such as knowledge graphs and embedding models and

(4) publication of the framework on GitHub.

An interested reader can dive from the conceptual level to the more detailed implementation level, where the algebraic definitions of pipelines are translated into scripts that could be directly executed as a piece of experimental software. The framework, including the implementation, was published on GitHub and is ready to be freely used and extended.

Notes

1. https://data.europa.eu/catalogue-statistics/Evolution
2. Measured by a SPARQL query for selecting the number of distinct datasets on https://data.europa.eu/data/sparql
3. https://gitlab.com/european-data-portal/metrics/edp-metrics-dataset-similarities/-/blob/master/src/main/java/io/piveau/metrics/similarities/SimilarityVerticle.java
4. https://data.europa.eu
5. https://data.gov.cz
6. https://www.data.gov/
7. https://joinup.ec.europa.eu/collection/semantic-interoperability-community-semic/solution/dcat-application-profile-data-portals-europe
8. https://www.w3.org/TR/vocab-dcat-2/#Class:Dataset
9. https://op.europa.eu/en/web/eu-vocabularies/authority-tables

10. https://data.gov.cz
11. https://www.wikidata.org/wiki/Q21503252
12. https://www.wikidata.org/wiki/Q21514624
13. https://zenodo.org/record/4433 464/files/2020.04.20-data.gov.cz-no-cuzk.trig?download=1
14. https://zenodo.org/record/3975 038
15. https://zenodo.org/record/4433 699
16. https://zenodo.org/record/4433 737
17. https://zenodo.org/record/4433 778
18. https://zenodo.org/record/4433 795
19. https://storage.googleapis.com/bert_models/2018_11_23/multi_cased_L-12_H-768_A-12.zip
20. https://zenodo.org/record/4436 356/files/20 181 217.json.gz?download=1
21. https://colab.research.google.com/

References

Altaf, B., Akujuobi, U., Yu, L. and Zhang, X. (2019), "Dataset recommendation via variational graph autoencoder", *2019 IEEE International Conference on Data Mining (ICDM)*, pp. 11-20, doi: 10.1109/ICDM.2019.00011.

Benjelloun, O., Chen, S. and Noy, N.F. (2020), "Google dataset search by the numbers", *Proceedings, Part II. Vol. 12507 of Lecture Notes in Computer Science, The Semantic Web - ISWC 2020-19th International Semantic Web Conference*, Springer, Athens, Greece, November 2-6, 2020, pp. 667-682, doi: 10.1007/978-3-030-62466-8_41.

Berners-Lee, T. (2006), "Linked data", available at: https://www.w3.org/DesignIssues/LinkedData.html.

Bernhauer, D. and Skopal, T. (2020), *Word2Vec Model - Czech Wikipedia*, Zenodo, Geneva, doi: 10.5281/zenodo.3975038.

Bernhauer, D. and Skopal, T. (2021a), *Node2Vec Model - Czech Wikidata (Knowledge Graph/Concepts/L40/Rw10)*, Zenodo, Geneva, doi: 10.5281/zenodo.4433795.

Bernhauer, D. and Skopal, T. (2021b), *Node2Vec Model - Czech Wikidata (Knowledge Graph/Concepts/L80/Rw40)*, Zenodo, Geneva, doi: 10.5281/zenodo.4433778.

Bernhauer, D. and Skopal, T. (2021c), *Node2Vec Model - Czech Wikidata (Knowledge Graph/Labels/L160/Rw40)*, Zenodo, Geneva, doi: 10.5281/zenodo.4433737.

Bernhauer, D. and Skopal, T. (2021d), *Node2Vec Model - Czech Wikidata (Knowledge Graph/Labels/L80/Rw40)*, doi: 10.5281/zenodo.4433699.

Bogatu, A., Fernandes, A.A.A., Paton, N.W. and Konstantinou, N. (2020), "Dataset discovery in data lakes", *36th IEEE International Conference on Data Engineering, ICDE 2020*, IEEE, Dallas, TX, USA, April 20-24, 2020, pp. 709-720, doi: 10.1109/ICDE48307.2020.00067.

Brickley, D., Burgess, M. and Noy, N.F. (2019), in Liu, L., White, R.W., Mantrach, A., Silvestri, F., McAuley, J.J., Baeza-Yates, R. and Zia, L. (Eds), "Google dataset search: building a search engine for datasets in an open web ecosystem", *The World Wide Web Conference, WWW 2019*, ACM, San Francisco, CA, USA, May 13-17, 2019, pp. 1365-1375, doi: 10.1145/3308558.3313685.

Browning, D., Beltran, A.G., Perego, A., Winstanley, P., Albertoni, R. and Cox, S. (2020), *Data Catalog Vocabulary (DCAT) - Version 2. W3C Recommendation*, W3C, available at: https://www.w3.org/TR/2020/REC-vocab-dcat-2-20200204/.

Carothers, G. and Seaborne, A. (2014), *RDF 1.1 TriG. W3C Recommendation*, W3C, available at: https://www.w3.org/TR/2014/REC-trig-20140225/.

Chapman, A., Simperl, E., Koesten, L., Konstantinidis, G., Ibáñez, L.D., Kacprzak, E. and Groth, P. (2020), "Dataset search: a survey", *VLDB Journal*, Vol. 29 No. 1, pp. 251-272, doi: 10.1007/s00778-019-00564-x.

Chen, X., Gururaj, A.E., Ozyurt, B., Liu, R., Soysal, E., Cohen, T., Tiryaki, F., Li, Y., Zong, N., Jiang, M., Rogith, D., Salimi, M., Kim, H.-e., Rocca-Serra, P., Gonzalez-Beltran, A., Farcas, C., Johnson, T., Margolis, R., Alter, G., Sansone, S.-A., Fore, I.M., Ohno-Machado, L., Grethe, J.S. and Xu, H. (2018), "DataMed – an open source discovery index for finding biomedical datasets", *Journal of the American Medical Informatics Association*, Vol. 25 No. 3, pp. 300-308, doi: 10.1093/jamia/ocx121.

Chen, Z., Jia, H., Heflin, J. and Davison, B.D. (2020), "Leveraging schema labels to enhance dataset search", in Jose, J.M., Yilmaz, E., Magalhães, J., Castells, P., Ferro, N., Silva, M.J. and Martins, F. (Eds), *Advances in Information Retrieval*, Springer International Publishing, Cham, pp. 267-280.

Cyganiak, R., Lanthaler, M. and Wood, D. (2014), *RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation*, W3C, available at: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

Das Sarma, A., Fang, L., Gupta, N., Halevy, A., Lee, H., Wu, F., Xin, R. and Yu, C. (2012), "Finding related tables", *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. SIGMOD'12*, Association for Computing Machinery, New York, NY, pp. 817-828, doi: 10.1145/2213836.2213962.

Degbelo, A. (2020), "Open data user needs: a preliminary synthesis", *Companion Proceedings of the Web Conference 2020*, WWW'20. Association for Computing Machinery, New York, NY, pp. 834-839, doi: 10.1145/3366424.3386586.

Degbelo, A. and Teka, B.B. (2019), "Spatial search strategies for open government data: a systematic comparison", CoRR abs/1911.01097, available at: https://arxiv.org/abs/1911.01097.

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2018), "Bert: pre-training of deep bidirectional transformers for language understanding", arXiv preprint arXiv:1810.04805, available at: https://arxiv.org/abs/1810.04805.

Dutkowski, S. and Schramm, A. (2015), "Duplicate evaluation - position paper by fraunhofer FOKUS", *Tech. rep., Fraunhofer FOKUS*, available at: https://www.w3.org/2016/11/sdsvoc/SDSVoc16_paper_24.

El-Sappagh, S., Hendawi, A. and El-Bastawissy, A. (2011), "A proposed model for data warehouse etl processes", *Journal of King Saud University - Computer and Information Sciences*, Vol. 23, pp. 91-104.

Ellefi, M.B., Bellahsene, Z., Dietze, S. and Todorov, K. (2016), "Dataset recommendation for data linking: an intensional approach", in Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P. and Lange, C. (Eds.), *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Proceedings*, Springer, Heraklion, Crete, Greece, May 29 - June 2, 2016, pp. 36-51, doi: 10.1007/978-3-319-34129-3_3.

Fellbaum, C. (2005), "WordNet and wordnets", in Brown, K. (Ed.), *Encyclopedia of Language and Linguistics*, 2nd ed., Elsevier, Oxford, pp. 665-670.

Fernandez, R.C., Abedjan, Z., Koko, F., Yuan, G., Madden, S. and Stonebraker, M. (2018), "Aurum: a data discovery system", *34th IEEE International Conference on Data Engineering, ICDE 2018*, IEEE Computer Society, Paris, France, April 16-19, 2018, pp. 1001-1012, doi: 10.1109/ICDE.2018.00094.

Gregory, K., Groth, P., Scharnhorst, A. and Wyatt, S. (2020a), "Lost or found? Discovering data needed for research", *Harvard Data Science Review*, Vol. 2 No. 2, available at: https://hdsr.mitpress.mit.edu/pub/gw3r97ht.

Gregory, K.M., Cousijn, H., Groth, P., Scharnhorst, A. and Wyatt, S. (2020b), "Understanding data search as a socio-technical practice", *Journal of Information Science*, Vol. 46 No. 4, pp. 459-475, doi: 10.1177/0165551519837182.

Grover, A. and Leskovec, J. (2016), "node2vec: scalable feature learning for networks", *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Klímek, J. (2019), "DCAT-AP representation of Czech national open data catalog and its impact", *Journal of Web Semantics*, Vol. 55, pp. 69-85, doi: 10.1016/j.websem.2018.11.001.

Klímek, J. and Škoda, P. (2021a), *Dump of Metadata from the Czech National Open Data Catalog, 2020-04-20, State Administration of Land Surveying and Cadastre Datasets Removed*, Zenodo, Geneva, doi: 10.5281/zenodo.4433464.

Klímek, J. and Škoda, P. (2021b), *Wikidata Dump from 2018-12-17 in JSON*, Zenodo, Geneva, doi: 10.5281/zenodo.4436356.

Koesten, L. (2018), "A user centred perspective on structured data discovery", *Companion Proceedings of the The Web Conference 2018. WWW'18. International World Wide Web Conferences Steering Committee*, CHE, Republic and Canton of Geneva, pp. 849-853, doi: 10.1145/3184558.3186574.

Leme, L.A.P.P., Lopes, G.R., Nunes, B.P., Casanova, M.A. and Dietze, S. (2013), "Identifying candidate datasets for data interlinking", in Daniel, F., Dolog, P. and Li, Q. (Eds), *Web Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 354-366.

Martins, Y.C., da Mota, F.F. and Cavalcanti, M.C. (2016), *DSCrank: A Method for Selection and Ranking of Datasets*, Springer International Publishing, Cham, pp. 333-344, doi: 10.1007/978-3-319-49157-8_29.

Mikolov, T., Chen, K., Corrado, G.S. and Dean, J. (2013), "Efficient estimation of word representations in vector space", available at: http://arxiv.org/abs/1301.3781.

Miller, R.J., Nargesian, F., Zhu, E., Christodoulakis, C., Pu, K.Q. and Andritsos, P. (2018), "Making open data transparent: data discovery on open data", *IEEE Database Engineering Bulletin*, Vol. 41 No. 2, pp. 59-70, available at: http://sites.computer.org/debull/A18june/p59.pdf.

Mountantonakis, M. and Tzitzikas, Y. (2018), "Scalable methods for measuring the connectivity and quality of large numbers of linked datasets", *Journal of Data and Information Quality*, Vol. 9 No. 3, doi: 10.1145/3165713.

Mountantonakis, M. and Tzitzikas, Y. (2020), "Content-based union and complement metrics for dataset search over RDF knowledge graphs", *Journal of Data and Information Quality*, Vol. 12 No.2, doi: 10.1145/3372750.

Oliver, J., Cheng, C. and Chen, Y. (2013), "TLSH – a locality sensitive hash", *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pp. 7-13.

Papadias, D. (2009), *Nearest Neighbor Query*, Springer, Boston, MA, pp. 1890-1890, doi: 10.1007/978-0-387-39940-9_245.

Škoda, P., Klímek, J., Nečaský, M. and Skopal, T. (2019), "Explainable similarity of datasets using knowledge graph", *Similarity Search and Applications - 12th International Conference, SISAP 2019, Proceedings*, Springer, Newark, NJ, USA, October 2-4, 2019, pp. 103-110, doi: 10.1007/978-3-030-32047-8_10.

Škoda, P., Bernhauer, D., Nečaský, M., Klímek, J. and Skopal, T. (2020), "Evaluation framework for search methods focused on dataset findability in open data catalogs", *Proceedings of The 22nd International Conference on Information Integration and Web-based Applications and Services (iiWAS2020)*, Chiang Mai, Thailand, November 2020, pp. 200-209, available at: http://www.iiwas.org/conferences/iiwas2020/proceedings/proceedings_iiwas_2020.pdf.

Skopal, T., Bernhauer, D., Skoda, P., Klímek, J. and Nečaský, M. (2021), "Similarity vs relevance: from simple searches to complex discovery", *Similarity Search and Applications - 14th International Conference, SISAP 2021, Proceedings*, Springer, Dortmund, Germany, September 29 - October 1, 2021, pp. 104-117, doi: 10.1007/978-3-030-89657-7_9.

Speer, R., Chin, J. and Havasi, C. (2017), "Conceptnet 5.5: an open multilingual graph of general knowledge", in Singh, S.P. and Markovitch, S. (Eds), *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI Press, San Francisco, California, USA, February 4-9, 2017, pp. 4444-4451, available at: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14972.

Straka, M. and Straková, J. (2019), *Universal Dependencies 2.5 Models for UDPipe (2019-12-06)*, LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Charles University, Faculty of Mathematics and Physics, Institute of Formal and Applied Linguistics, Prague, available at: http://hdl.handle.net/11234/1-3131.

Vrandečić, D. and Krötzsch, M. (2014), "Wikidata: a free collaborative knowledgebase", *Communications of the ACM*, Vol. 57 No. 10, pp. 78-85, doi: 10.1145/2629489.

Wagner, A., Haase, P., Rettinger, A. and Lamm, H. (2014), "Entity-based data source contextualization for searching the web of data", *The Semantic Web: ESWC 2014 Satellite Events*, Springer International Publishing, Cham, pp. 25-41.

Yakout, M., Ganjam, K., Chakrabarti, K. and Chaudhuri, S. (2012), "Infogather: entity augmentation and attribute discovery by holistic matching with web tables", *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. SIGMOD'12*, Association for Computing Machinery, New York, NY, USA, pp. 97-108, doi: 10.1145/2213836.2213848.

Zezula, P. (2015), "Similarity searching for the big data - challenges and research objectives", *Mobile Networks and Applications*, Vol. 20 No. 4, pp. 487-496, doi: 10.1007/s11036-014-0547-2.

Zhang, S. and Balog, K. (2018), "Ad hoc table retrieval using semantic similarity", *Proceedings of the 2018 World Wide Web Conference. WWW'18. International World Wide Web Conferences Steering Committee*, CHE, Republic and Canton of Geneva, pp. 1553-1562, doi: 10.1145/3178876.3186067.

Zhang, E. and Zhang, Y. (2009), *Eleven Point Precision-Recall Curve*, Springer, Boston, MA, pp. 981-982, doi: 10.1007/978-0-387-39940-9_481.

Appendix

The Appendix is available online for this article.

Corresponding author
Jakub Klímek can be contacted at: jakub.klimek@matfyz.cuni.cz