FiLiPo: A Sample Driven Approach for Finding Linkage Points between RDF Knowledge Bases and Web APIs

Tobias Zeimetz
zeimetz@uni-trier.de
Trier University
Trier, Germany

Ralf Schenkel
schenkel@uni-trier.de
Trier University
Trier, Germany

ABSTRACT

Data integration is an important task in order to create comprehensive RDF Knowledge Bases. Many data sources are used to extend a given dataset or to correct errors in the data. Since several data providers make their data publicly available via Web APIs, they are used as a common external data source. The classical problems of data integration, i.e., how two different datasets can be mapped, remain. Furthermore, due to the heterogeneity of data structures, the integration of different datasets is a mainly manual task. In addition, Web APIs are often more restrictive than data dumps and of course slower to access due to latencies and other constraints. In this paper we present the FiLiPo (**Fi**nding **Li**nkage **Po**ints) system to automatically find connections (i.e., linkage points) between the data of Web APIs and local Knowledge Bases. FiLiPois a sample-driven schema matching system, which models Web API services as parameterized queries. These Web API services return single records which contain a view of the Web API data schema. Furthermore, our approach is able to find valid input values for Web API services automatically (e.g. IDs) and can determine combined linkage points (e.g. first and last name) despite different structures. Our results on seven real world API services with two local databases show that our linkage point detection algorithm performs well in terms of precision. FiLiPo was able to achieve a average f1 score of 0.88 while the chosen baseline only achieved 0.74.

KEYWORDS

Data Integration, Schema Mapping, Relation Alignment

1 INTRODUCTION

When it comes to data integration, Web APIs have become increasingly important in recent years across many domains (e.g., bibliographic or biological data). Accessing live data through an API has several advantages compared to the classic way of using data dumps. Normally, data dumps are not updated and redeployed on a daily basis. Data providers often update their dumps only every few months or once a year. In contrast, APIs provide a direct connection to the (usually) most recent data of the data provider. In addition, the number of potential data sources becomes much larger when using APIs for data integration, since most data providers tend to share their data via APIs. This seems to be a sweet-spot between making data openly accessible and protecting it[13].

Next to API's, RDF Knowledge Bases (KBs) have established themselves as an important database format and are used in many domains. RDF KBs are graph databases consisting of triples, where a triple is a statement of the form (subject, predicate, object). These KBs are usually openly accessible for everyone via SPARQL endpoints. SPARQL is the standard query language for RDF KBs and is similar in structure to SQL. Similar to APIs, SPARQL endpoints define an interface to access the data of the KB. In contrast to APIs, these endpoints are not restricted in their way to access data. Since databases generally come with the problem that they are potentially incomplete (considering how much new data is generated daily) it is highly desirable to integrate the data of an API into a KB.

The usual process of data integration is to download data dumps which contain potentially new information. Afterwards the schema of the local KB and the schema of the data dump are aligned. "Aligning" describes the process by which relations from the local KB are mapped to relations from external sources, thus creating a mapping between the local and the external data schema. The approach to use APIs instead of data dumps solves the issue to integrate the most recent data available but has also few disadvantages. APIs are often more restrictive and of course slower to access due to latencies and other constraints. The classical problems of data integration, i.e. how two different schemes can be mapped, remain. In the worst case, the schema of an external source is structured completely differently from that of the KB. For example, in case of a bibliographic KB, the author name may be modelled as a single property, whilst some external data sources use two properties to model the name, e.g. first and last name. Further problems could also be different spellings, different formatting and so forth. For this reason the data integration process remains a manual task for most parts [13].

Connecting KBs with data of APIs can significantly improve existing intelligent applications. For example, dblp[1] is a bibliographic database containing metadata about major scientific computer science articles. It contains different kinds of information about publications, e.g., titles, publisher names, author names and more. The dblp data is often used for work in the field of reviewer, venue or paper recommendation. By extending dblp with information from APIs, these applications can be improved. Furthermore, the information need of a user querying dblp can be satisfied by integrating the missing data. The aim in this case is to complete the missing information using external data sources. Therefore it is important that multiple APIs can be used and missing data can be integrated from many different sources. Additionally, the determined alignments can be used to identify erroneous data and correcting it if necessary.The computed alignment can be used to validate erroneous data in the KB by using data of APIs.

Problem Statement. When using an API it is not possible to access all data at once, contrary to data dumps. Most APIs have an input parameter that must be set with a valid input in order to retrieve information about a particular entity (see Figure 1). For example a bibliographic API may need as input parameter the

[1] https://dblp.uni-trier.de/

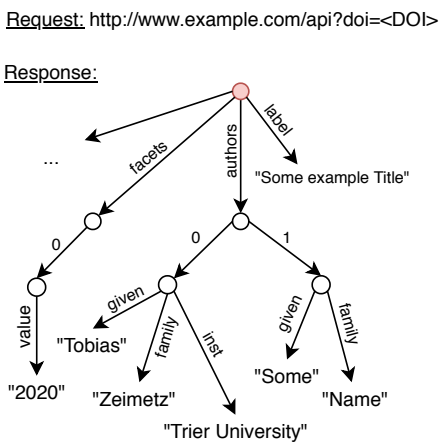

Figure 1: Example of a Web API Request and Response

DOI (Digital Object Identifier) of a paper and afterwards responds with all information about the requested entity. Since not all APIs have the same input parameter (e.g. DOIs) the first problem lays in finding the appropriate input values to call the API. A further problem is that some APIs behave like an information retrieval tool and respond with the entity closest to the requested entity. This behaviour is due to the fact that it is better to answer with any information than with none, since the user may have made a typo in the input. Therefore it is important to identify if the response of the API was a correct response (containing the desired data), a false response (containing data that does not belong the requested data) or an error code returned.

Additionally, the aligning of a KB and an API itself is challenging. As shown in Figure 1 some paths in the API response (usually sent in JSON format) do not give away any clear semantics (e.g. `facets`). For this reason, a name-based matching of the relations can only hardly be used. Furthermore, the question arises how the paths of API responses should be mapped with relations of a KB if the structure is completely different. Relations in the KB may correspond to paths of different length in the call result. Additionally, different spellings or other encodings could make a matching more difficult. For example, names are often abbreviated and exact matching is therefore difficult. Furthermore there is no such thing as one similarity metrics to compare all values. For some data types various similarity metrics are better suited than others. The last problem consists of the inference of the API's data schema/structure. In order to align the data schemas of an API and a KB, both schemas need to be known.

Contribution. We present FiLiPo (**Fi**nding **Li**nkage **Po**ints)[2], a system to discover alignments between a local KB and an API in an automatic manner. A user only needs knowledge about the local KB (e.g. class names) but no prior knowledge about the API data schema. FiLiPo automatically detects which kind of information has to be sent to an API in order to get valid responses. We use a sample driven approach coupled with the usage of fifteen different string similarity metrics in order to find valid matches between the schema of a local KB and that of an API. FiLiPo consists of two phases, the *probing phase* and the *aligning phase*. In the first phase FiLiPo sends various information (e.g. DOIs, titles, etc.) to an API to determine which information the API responds to. Then, the information returned is used to guess the schema of the API and to determine an alignment between the local and external data. In contrast to other systems we do not use just one string similarity method to determine an alignment rather several different ones. The reason for using more than one similarity method is that a single method is not suited to compare different data types (e.g. names, IDs, etc.). FiLiPo is able to automatically determine a similarity method that performs well for a given data type (e.g. DOI, names, titles, etc.) and determines afterwards an alignment. A user only needs to specify the number of samples sent to the API by the system in order to keep the approach usable even for non-technical users.

The *novel part* of FiLiPo is that it automatically detects which kind of information has to be sent to the Web API in order to get valid responses. In addition, FiLiPo is able to find not only one-to-one matches, like other systems, but also one-to-many matches, i.e. aligning a single relation (e.g. full name) with multiple other relations (e.g. first and last name). Another innovation is that the FiLiPo system is able to recognise what inputs of the Web API have in common. A Web API from a specific publisher such as Springer may only respond to books and publications that have been published by that publisher. FiLiPo is able to recognise this and provides this information to the user.

2 RELATED WORK

Web API Alignment. Some work has already been done regarding the aligning of different datasets. However, only the DORIS system [13, 14] has dealt with the alignment of KBs and APIs so far and is later used as baseline system for the evaluation of FiLiPo. The core idea of DORIS is to build upon the schema and structure of an existing KB containing instances and facts about these instances. First, the system sends some probing requests to a chosen API. DORIS uses label information of instances as its predefined input relation for APIs. However, this is not always the appropriate input parameter for the API. For example, some APIs expect DOIs or ISBNs as input parameters.

One key assumption of the DORIS system is that it is more likely to find information on well-known, popular or famous entities (e.g. famous actors, acclaimed books, or big cities) via APIs calls than it would be for unknown entities. Additionally, Koutraki et al. assume that a KB contains more facts (triples) for well-known entities than for lesser-known entities and therefore rank the entities of the input class by descending number of available facts. While this is a reasonable approach for open-topic knowledge bases like YAGO, it is likely to fail for domain-specific knowledge bases. For a publication, for example, the number of facts stored by a bibliographic KB is often determined by the meta data of that publication, not by its popularity (unless citations etc. are stored). In contrast to this approach FiLiPo picks randomly chosen entities of a KB.

DORIS normalises values by lower casing the string, ignoring punctuation, and reordering the words in alphabetical order. A benefit of this approach is that a pairwise comparison of KB and API values is no longer necessary. A relation and a path are considered as a match if the corresponding values are exactly equal after normalisation. Therefore, the system can use an efficient merge-join algorithm. Normalised values of the paths and relations are sorted in alphabetic order. Afterwards, the two lists can be merged to produce the relation-path matches. This stands in contrast to our approach since, as stated previously, a single method is not suited to compare all various data types. The

[2]https://github.com/dbis-trier-university/FiLiPo

limitations of DORIS become clear when examining, for example, author names or titles. Names are often abbreviated and the matching approach of DORIS will fail because DORIS performs an exact match on the normalised names. Similar problem will arise when examining titles.

DORIS uses two methods to determine the final alignment. The first and simplest method is the Overlap method, which involves counting the number of times that a relation has been matched to a given path and then dividing that number by the sample size. The sample size describes the number of requests sent to the Web API. If the overlap value is greater than or equal to a given threshold, the relation with the path is considered a match. This method works well in practice, but has the disadvantage that relations that occur rarely but are always matched with the same path are lost. Therefore a second method, PCA confidence, is used. This does not use the sample size as a benchmark, but how often the path was used in the API answers. The downside is that a path that was found only once in the API response only needs to match once in order to achieve a high confidence. In such cases it is risky to trust the match and therefore a re-probing is performed. The re-probing selects for this step entities for which the corresponding relation is defined in the facts of the entity. As previously, the entities are ranked by the descending number of facts which will result in an increasing runtime.

The problem of aligning the schema of KBs and APIs shares similarities with various fields [5, 13, 18] like schema matching, data warehouses, e-commerce, query discovery and Web service discovery. For this reason, insights and procedures of systems from other fields were also taken into consideration when developing the FiLiPo system. Aligning data of local KBs with that of APIs has similar problems as schema mapping or ontology alignment. The main difference lies in the online nature of APIs and that data first needs to be requested. Additionally, there are problems regarding latency and other restrictions (e.g., limited number of requests per day) set by the provider of the Web API.

Machine Learning. Machine Learning is experiencing a big boom in the past years. Approaches like BERT[7] or Flair[1] (both frameworks that learn contextual relations between words in a text) enable to grasp grasp the semantics of data and to compare them with others. A clear disadvantage of machine learning alignment approaches [12, 19, 20] is that they require training data, which is either provided by experts or learned from predefined mappings. Although this approach may be promising, it is not suitable for non-technical users due to the need for extensive prior knowledge. In contrast, DORIS and FiLiPo do not require the user to be a machine learning expert and assume no assistance through experts during the mapping process. Additionally, the results are often worse than with a rule-based program or classic approaches.

Schema Alignment. Classical approaches for schema or ontology alignment take a set of independently developed schemas as input and construct a global view [2, 6] over the data. Since the schemes are independently developed, they often have different structure and semantics. This is especially the case when the schemes are from different domains, of different granularity or simply developed by different people in different real-world contexts [18]. The first step when performing schema alignment is to identify and characterise relationships that are shared by the schemas. After identifying the shared relations they can be included in a global schema. Since APIs do not provide any schema information it is not possible to use classical schema or ontology alignment strategies.

Approaches like COMA++[2] represent customisable generic matching tools for both schemas and ontologies. COMA++ relies on using a taxonomy to determine an alignment and uses a so-called fragment based match approach (breaking down a large matching problem into smaller problems). Additionally, it enables various interaction possibilities to the user in order to influence the alignment process. Furthermore, COMA++ does not use any instance information of the KBs.

Systems like BLOOMS [10] use schema information if it is present in the Linked Open Data (LOD) cloud. This means that a schema no longer needs to be explicitly declared and the schema information of reused vocabularies (ontologies) can be accessed via the LOD cloud. However, since API responses (mostly) do not contain explicit schema information (of the LOD domain), the procedure does not work for such approaches. In addition, the collected answers of an API implicitly contain a schema which only needs to be extracted [4]. This is similar to KBs which contain implicit schema information even without the specification of a schema. Much user assistance and knowledge should not be required such that non-technical users are able to use such a system. Furthermore, our goal is a fully automatic determination of an alignment without any user assistance.

Systems like AgreementMaker [6] are based on the assumption that users of the system are domain experts and thus build on user assistance. The system uses a large set of matching methods and enables the user to define the types of components to be used in the aligning process, i.e. using the schema only or both schemas and instances.

Instance-Based Alignment. Instance-based alignment systems use the information bound to instances in KBs in order to find shared relations and instances between two KBs. These approaches can be divided into instance-based class alignment approaches and instance-based relation alignment approaches. The main difference between class and relation alignment lies in the fact that relations have a domain and range. Even if relations share the same value, they can have different semantics (e.g. `editor` and `author`).

A lot of works [8, 11, 15, 17] focus on instance-based relation alignment between two KBs. However, most of them focus on finding one-to-one matches, e.g. matching `publicationYear` to `year`. The iMAP system [8] semi-automatically determines one-to-one matches, but also considers the complex case of one-to-many matches. The system offers the user an explanation of the determined match containing information about why a particular match is or is not created, and why a certain match is ranked higher than another.

Similar to iMap, MWeaver [17] also needs user assistance. MWeaver realises a sample-driven schema mapping approach which automatically constructs schema mappings from sample target instances given by the user. The key idea behind this system is to allow the user to implicitly specify schema mappings by providing sample data.

Madhavan et al. [15] state that KBs often contain multiple schemas and data models to materialise similar concepts and hence build variations in entities and their relations. This makes purely schema-based matching inaccurate, which must therefore be supported by evidence in form of instances from the KB.

SOFYA [11] is an instance-based on-the-fly approach for relation alignment between two KBs. The main idea behind this approach is to use samples of data from both KBs in order to identify matching relations and to perform two types of alignments, subsumption and equivalence. The subsumption strategy is softer

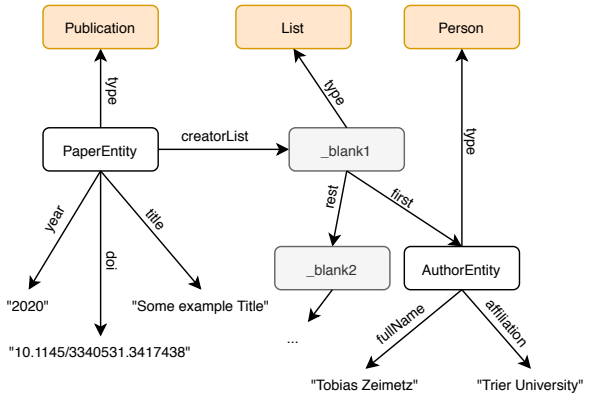

Figure 2: Fragment of an RDF Knowledge Base

than equivalence because it aligns two paths if the results of one are included in the results of the other.

Holistic Approaches. Some developed systems cannot be categorised easily since they use techniques of all fields. ILIADS [22] for example takes two OWL ontologies as input and determines afterwards an alignment based on lexical, structural and extensional similarity. They combine a flexible similarity matching algorithm with an incremental logical inference algorithm. Additionally, ILIADS uses a clustering algorithm which considers relationships among sets of equivalent entities, rather than individual pairs of entities.

PARIS [21] is an instance-based approach that aligns related entities and relationship instances, but also related classes and relations. The goal of PARIS is to discover and link identical entities automatically across ontologies. It was designed such that no training data or parameter tuning is needed. PARIS does not use any kind of heuristics on relation names and therefore it is able to align relations with completely different names. However, a downside of PARIS is that it is not able to deal with structural heterogeneity. This is a major problem, because most KBs are structured differently from each other, since they have been developed by different people [18].

Madhaven et. al. developed a system called Cupid [16] which is used to discover an alignment between KBs based on the names of the schema elements, data types, constraints and structure. It combines a broad set of techniques of various categories (e.g. instance-based, schema alignment, etc.). The system uses an linguistic and structural approach in order to find a valid alignment.

3 PRELIMINARIES

This section introduces some terms and notations used in this paper. The first part will introduce RDF knowledge bases and corresponding terms and the second part deals with APIs.

3.1 Knowledge Base

RDF Knowledge Bases. The RDF format has established itself as the standard for representing and storing KBs. An RDF KB is essentially a graph with labelled nodes and edges. The set of allowed nodes consists of three disjoint subsets: IRIs (I), literals (L) and blank nodes (B). In Figure 2 we present an exemplary snippet of a bibliographic KB to explain the three subsets. IRIs serve as global identifiers that can be used to identify resources, e.g. identifying a single paper in the exemplary KB (`PaperEntity`). Literals are a set of lexical values, e.g. titles or author names. Blank nodes serve as local identifiers for resources that are not otherwise named. Typically blank nodes are used to model lists or sequences (`creatorList` in the example).

The edges in an RDF KBs correspond to triples of the form (subject, predicate, object). Such an RDF triple represents a fact in the KB. The subject (an IRI from I or a blank node from B) describes the entity the fact is about, e.g. a paper entity. The predicate (an IRI from I) describes the relation between the subject and the object, e.g. `title`. The object (an IRI from I, a blank node from B or a literal from L) describes an entity, e.g. an author of the publication, or is a literal, e.g. the title of a publication. More formally, an RDF triple t is defined as $t = (s, p, o)$ where $s \in I \cup B$, $p \in I$ and $o \in I \cup B \cup L$.

A class in a KB is an entity that represents a group of entities. Every entity contained in a KB is assigned to at least one class. Entities assigned to a class are denoted as an instance of the assigned class. In the example, the entity `PaperEntity` is an instance of the class `Publication`, and the entity `_:blank1` is an instance of the class `List`.

Relations. Since this paper focuses on aligning relations, we introduce a formal definition for relations and relation paths in a KB. In the following we assume that we have given a KB K. If $(s, r, o) \in K$, we say that s and o are in relation r, or formally $r(s, o)$; in other words, there is a path from s to o with label r. Additionally, we write $r_1.r_2.....r_n(s, o)$ to denote that there exists a path of relations $r_1, r_2, ..., r_n$ in K from subject s to object o visiting every intermediate node only once. For example, in Figure 2 the relation `year(PaperEntity,"2020")` describes the path from the entity `PaperEntity` to the value `"2020"`. In the following we will refer to $r_1.r_2.....r_n(s, o)$ as relation-value triple.

Branching Points. A branching point represents a relation of an entity, which is used multiple times by this entity, but always points to other objects or literals. For example a generic relation like `identifier` could be used by an entity to link multiple different external keys. Additionally a relation that points to a list of entities is considered to be a branching point (e.g. `creatorList` in Figure 1). To indicate a branching point, we will use the symbol $ instead of the numeric index in paths; in the example, we will write `creatorList.$.fullName("Tobias Zeimetz")`.

Identifier Relations. As already mentioned, entities are represented in a KB by IRIs. Thus, each entity is unique and has its own identifier. However, some KBs also contain external identifiers or globally standardised identifiers such as DOIs, ISBNs or ISSNs. Such identifiers are only bound to a single entity and should therefore be unique. Relations to model identifier values are therefore "quasi-functions" and usually realised trough functional relations. Many works [9, 13, 21] have used the following definition of functionality for determining functional relations:

$$fun(r) := \frac{|\{x : \exists y : r(x, y)\}|}{|\{(x, y) : r(x, y)\}|} \tag{1}$$

Functional relations have functionality 1 by construction. Since real world KBs are designed and modeled by humans they are often error-prone and noisy. For this reason some identifier values may appear more than once. Therefore, we consider every relation r contained in K with $fun(r) \geq \theta_{id}$, where $\theta_{id} \in [0, 1]$ is a threshold, as *identifier relation*.

Knowledge Base Schema. In order to capture the rough structure of a KB a schema enriched with statistical information and more is needed. Ontologies only contain information about classes, sub-classes and relations. However, a KB contains

more information hidden in its instances than in an simple ontology, e.g. the number of instances of a specific class, the number of occurrences of a relation with a specific class, or what classes are connected to each other. For the rest of the paper we will refer to a schema $S = (C, R_s, R_{id})$ of a KB as a triple of three sets. The first component C defines a set of classes contained in the KB and how many instances (entities belonging to the corresponding class) for each class exist in the KB. The second component R_s tries to grasp the structure of the KB. It contains information about how classes and literals are connected to each other, how often a class is connected to another class and how often a relation is used by a specific class. The last component is the set R_{id} which contains all identifier relations contained in the KB.

3.2 Web Services

Web API. A Web Service can provide one or multiple APIs to access data. APIs are called via parameterised URLs responding with a call result in form of a (typically) JSON or XML file. The algorithm we describe in this paper works with JSON responses; existing frameworks [3] can be used to transform an XML result into a JSON result. As shown in Figure 1 the response of an API is an unordered and labelled tree. Inner nodes in the tree represent a JSON object (similar to an entity in a KB) or an array, leaf nodes represent values. The path to a node represents a relation between an instance (similar to an entity in a KB) and another instance or value. To avoid confusion we will describe the relations in a JSON response only as paths.

Path-Value-Pairs. In order to find a valid alignment between a KB and an API the information in the JSON response has to be compared with the values of the corresponding entity in a KB. It is easy to see that comparing JSON objects and arrays from the Web API response with entities from the KB to determine alignment is not promising. Therefore, only paths to leafs (literals) have to be considered. In the following we assume that we have given an API response res. We will write $p_1.p_2.....p_n(o)$ to denote that there exists a path $p_1, p_2, ..., p_n$ in res from the root of the JSON response to the leaf o with these labels. For example, in Figure 1 the path `title("Some example Title")` describes the path from the root of the JSON response (highlighted in red) to the leaf `"Some example Title"` via the path `title`.

Branching Points. Similar as in the case of KB relations, a branching point in a JSON repsonse indicates that there are several outgoing edges from one node, labelled by numeric index values 0 to n. These branching points represent arrays in the JSON response. For example the path `authors.0.given("Tobias")` in Figure 1 contains a branching point. To indicate a branching point, we will use the symbol `*` instead of the numeric index in paths; in the example, we will write `authors.*.given("Tobias")`. This indicates that all array entries carry the same information and should therefore be mapped to the same relation. Additionally, we write $P*$ to indicate a path $P * p$ that has P as prefix and p as suffix, with a branching point separating the two parts.

4 PROBLEM STATEMENT

FiLiPo is a system that discovers alignments between local KBs and APIs. Furthermore, it automatically detects which kind of information has to be sent as an input parameter to an API in order to retrieve valid responses. In the following we will discuss some of the major problems that FiLiPo encounters and solves in more detail.

[3]The json.org framework provides methods to convert an XML to a JSON file

Probing Phase Problems. The first major problem to be solved occurs in the probing phase. The goal of the probing phase is to determine which input values (e.g. DOI, ISBN, etc.) have to be sent to the API to retrieve a valid response. When a resource is requested that is unknown to the API, it can respond in several ways. The classic and simple case is that it returns a corresponding HTTP status code (e.g. `404 Not Found`). The more complicated case is when the API simply replies with a JSON response that contains an error message, or even returns information on a "similar" resource (e.g., with a similar DOI). This cannot be easily distinguished from a "real" response which contains data about the requested resource.

Alignment Phase Problems. The alignment phase is used in order to find valid matches between the data of an API and a KB. For this reason the data of the API response needs to be compared with the data of the KB. This is where the first problems of the alignment phase arise. Since the same value may be represented slightly differently in the KB and the API response (e.g., names with and without abbreviated first names, or names with typos), this comparison needs apply string similarity methods. The different existing similarity methods have different strengths and weaknesses. For example, Levenshtein distance is good for comparing the titles of a paper or movie, but when comparing names of authors or actors this method performs poorly. This is due to the fact that names are often abbreviated and that first and last names may be in different order. Therefore we can conclude that each "datatype" has its best performing method and more than one similarity method should be used. Additionally, for each data type the most appropriate metric should be chosen automatically.

As stated in Section 3.1 some KBs contain identifier relations (e.g. DOIs, ISBNs, etc.). Such identifiers should also be compared with a suitable metric, otherwise erroneous alignments will occur. For example, DOIs of a conference book and conference papers are very similar, in many cases only the last two digits of both DOIs will be different. Another example is an ISBN of a Book. Such identifiers can have different writing styles (e.g. 978-3-89318-084-4 or 9783893180844) but should be considered equal. For this reason a simple check for equality is not sufficient, otherwise possible alignments are lost.

Finding a match between the records of a KB and an API response can be particularly problematic if the API responds with a record similar to the requested one. As stated previously, some APIs behave like an information retrieval tool and respond with the closest entity to the requested entity. For example, a user requests all information for a book with with title "Some example Title" but the Web API responds with all information of a book with title "Some Title". Since the data of both records can overlap, especially for data with a low functionality[4], a system has to check if the API has responded with the requested record. Koutraki et. al [13] state that if the KB and the API share the same genre (e.g. bibliographic data) it is likely that the data of the requested KB entity and the API record overlaps. This means that if the information in the records overlaps sufficiently, the API has probably responded with the requested record and not with another record.

One observation we made during the development of FiLiPo is that some values, such as years, are included in the records

[4]Data such as years have a low functionality, because these values are shared with other entities. Therefore they can be mapped even if the response does not fit to the request.

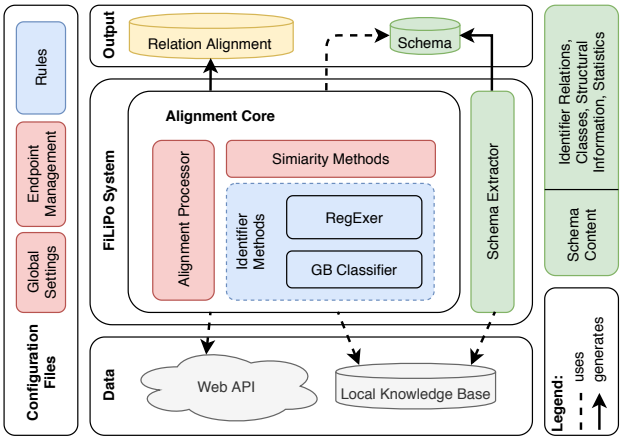

Figure 3: FiLiPo System Architecture

several times. This is because they represent different information. For example, some bibliographic APIs respond with data containing references and citations of a paper. These references often include the publication year. When a relation like the publication year of the KB is matched with the publication year of a reference (e.g. because the reference was published in the same year), the matching is no longer correct. Theoretically the semantics of the paths should be considered but as mentioned earlier, API responses do not always have a clear or a directly resulting semantics. The problem becomes even bigger when considering the modelling process of the API or KB schema. The value stored for the publication year of a paper often depends on the opinion of the data provider. Some data providers assume that the publication year is the year the conference proceedings are published. Other data providers model the year of the corresponding conference as publication year of the paper, since the paper was presented in this year at the conference.

5 SYSTEM OVERVIEW

In order to better understand how FiLiPo proceeds, we will first give an overview of the system components. The FiLiPo architecture shown in Figure 3 is divided into two main components, the *Alignment Core* and the *Schema Extractor*. Additionally, the system uses a set of configuration files to manage local KBs and APIs that will be used for the alignment process.

Configuration Files. There are three configuration files in FiLiPo. The first configuration file (`Global Settings`) stores all global settings like the paths to output files and a set of different thresholds values (more in in Section 6). The second configuration file (`Data Management`) stores information about KBs and APIs. It is used to manage multiple data sources and therefore stores the paths to registered KBs and the URLs under which the registered APIs can be accessed. Additionally, the input class (e.g. `Publication`) for every API that is used is specified. The input class describes the class of entities from the used KB whose facts are used to request information from the API. For example, the API presented in Figure 1 has as input class `Publication` and values of the input relation `doi` are used to request the API. The last configuration file (`Rule Set`) stores a set of rules how identifiers can be normalised (e.g. removing hyphens in ISBNs).

Schema Extractor. The *Schema Extractor* is used to derive a simple but useful overview of a KB. As explained previously, a schema $S = (C, R_s, R_{id})$ contains three components. In order to create a schema S, FiLiPo parses the RDF KB triple-wise. To determine identifier relations, the system computes the functionality (see Equation 1) of each relation. Since real world KBs may be noisy and contain errors, a perfect functional relation is unlikely. In case a relation has a functionality greater or equal to 0.99 (in case the KB contains erroneous data), the system assumes that the relation describes an identifier in the KB. It is important to determine the identifiers of a KB because the FiLiPo system uses several string similarity methods that may be too fuzzy to compare identifier values correctly.

Alignment Core. The *Alignment Core* consists of four components, namely the `Alignment Processor`, the `Similarity Processor`, the `RegExer`, and the `Classification` component. The Alignment Processor interacts with the other components to determine a correct alignment. The `Similarity Processor` is used by the `Alignment Processor` to compare values of a KB record and an API response. FiLiPo uses the string similarity library developed by Baltes et al. [3] in order to compare these values and to find valid matches. This similarity library uses three types of similarity methods: (1) equal, (2) edit and (3) set-based. String similarity methods of the equal category check for the equality of two strings; edit-based methods (e.g. Levenshtein) define the similarity of two strings based on the number of edit operations needed to transform one string into the other [3]. Set-based methods determine how large the intersection of two strings in terms of tokens (e.g. n-grams, n-shingles, etc.). We excluded the overlap method in the set-based category since this method is too fuzzy and would lead to an erroneous aligning (e.g. aligning a title and an abstract since they have in most cases many tokens in common). By using this library FiLiPo can use up to 15 different similarity methods with several variants.

Identifier Methods. In addition to the similarity methods, the `Alignment Processor` uses multiple methods to compare identifier values (`Identifier Methods`). For example, in order to match an ISBN with a value returned by an API the values need to be equal. As explained previously, identifiers can have several writing styles and therefore need to be normalised before comparing, e.g. removing hyphens in ISBNs. Using fuzzy similarity methods like Levenshtein is inaccurate but using *equals* is too strict. Therefore FiLiPo has additional components in order to compare identifier values, namely `RegEx` and `GB Classifier`. The first one is a simple regex based variant using predefined rules to compare two values by previous normalisation of those values. The second classifier variant is utilising a gradient boosting classifier working on Flair [1] embeddings of identifiers to determine whether two values are equal. We use Flair embeddings instead of other embeddings since this framework is character-based and therefore suits better for the comparison of two identifier strings. Both variants can be extended by an expert by adding new normalisation rules for the RegExer or by training the gradient boosting classifier for new identifiers. The user can specify in a configuration file whether to use the classifier or regular expression approach. A detailed explanation of the procedure of the Classifier and the RegExer can be found in Section 6.3.

6 SCHEMA MATCHING AND MAPPING

In this Section we will describe the aligning process in detail. A requirement for using FiLiPo is that the user has to describe every API (URL and input class) that is used by the system in the configuration file. The input class describes the class of entities of the local KB that will be used to request the API. After starting

the system the user has to chose a KB and API that subsequently will be aligned by the system. Additionally, the user has to specify how many samples (n_s) are sent to the API.

6.1 Schema Generation

The first step of FiLiPo is to check if a schema was already created for the chosen KB. If the corresponding schema does not exist, the first step of FiLiPo is to create a schema $S = (C, R_s, R_{id})$. The system parses every triple $t = (s, p, o)$ in KB K and divides the triples in *fact triples* and *type triples*. The type triple set (T) contains triples with a type information, i.e.

$$T = \{t : t = (s, \texttt{rdf:type}, \tau) \in K\},$$

where `rdf:type` denotes a standardised relation to denote the type τ of an entity s. The fact triple set (F) contains triples that actual represent a fact of the entity, i.e.

$$F = \{t : t = (s, p, o) \in K \text{ with p } \notin P_s\}$$

where P_s denotes the set of predicates that model structural information regarding classes and relations (e.g. classes, sub classes, property ranges, property domains, etc.)

In the next step of the schema generation process, all subjects of triples stored in F will be replaced by their type(s), based on information from T. Formally, we create a new multi set (that allows multiple instances for each of its elements) denoted as typed facts (F_T) where

$$F_T = \{t : t = (\tau, p, o) \text{ where } t_1 = (s, \texttt{rdf:type}, \tau) \in T \text{ and } t_2 = (s', p, o) \in F \text{ s.t. } s = s'\}.$$

After creating F_T, the final schema will be created by using F_T and T. The first schema element (C) that is created contains the classes used in K and how many instances of these classes exist. For this reason, for every type $\tau \in K$ the system counts how many triples with the corresponding class τ exist in T.

In the example fragment of a KB presented in Figure 2, three classes are used, namely `Publication`, `Person` and `List`. Additionally, the fragment contains four entities (`PaperEntity`, `AuthorEntity1`, `_blank1` and `_blank2`). For the sake of simplicity we assume that the entity `_blank2` is connected only to another author `AuthorEntity2`. The created class set of S is then $C = \{(\texttt{Publication}, 1), (\texttt{List}, 2), (\texttt{Person}, 2)\}$.

The second generated schema element (R_s) contains information about relations used by classes τ in K. To be precise, it stores information about which classes τ are connected to other classes and literals and which relations p are therefore used. Additionally it is stored how often entities of type τ use a relation p.

In order to give an example, see again Figure 2, assuming that the blank node `_blank2` is connected only to another author with a full name relation. We can see that `PaperEntity` is connected to the blank node `_blank1` of type `List` via the relation `creatorList`. Additionally, we can see that `PaperEntity` uses several other relations, e.g. `year`, `title` and `doi`. Therefore, R_s contains the following information:

$$R_s = \{(\texttt{Publication}, \texttt{creatorList}, \texttt{List}, 1), \ldots, (\texttt{Author}, \texttt{fullName}, \texttt{Literal}, 2)\}$$

In the example, entities of type `Publication` are connected to entities of type `List` only once, and two `Author` entities in the KB exist that use the relation `fullName` to point to a literal value. For the sake of simplicity all other information contained in R_s is omitted here.

The last generated schema element (R_{id}) contains all identifier relations of the KB K. In order to determine all identifier relations we use F_T and calculate for all relations p contained in triples of F_T their functionality by using $fun(p)$ (see Equation 1). If a relation p has a functionality $fun(p) \geq 0.99$ the systems assumes that the relation is used in order to model identifier relations (e.g. for DOIs, ISBNs, etc.).

The fragment of a KB presented in Figure 2 contains only one identifier relation, namely `doi`. Since the example is too small to determine whether a relation is an identifier relation with the help of the functionality function $fun()$ in a reasonable way, it would follow that also `title`, `year`, and `creatorList` are identifier relations. This is because there is only one paper in the example and therefore the relations `year`, `title` and `creatorList` are used only once. In a real KB with more than one publication, for example `year` would be used more often and would also point to the same year more often (more than one paper was published in 2020). This would result in a functionality lower than 0.99 and year would not be recognised as identifier relation anymore.

6.2 Probing Phase

After the system has successfully extracted the schema, the probing phase starts. This phase is used to find relations p_{in} of the input class that point to values which can be used to request the API successfully. This is done because a specific information from the entities (e.g. the DOI) is required to request the API. To illustrate this with an example, we assume that the input class of the API whose result is presented in Figure 1 is `Publication`. The illustrated fragment of a KB in Figure 2 has five relations to describe the metadata of a publication (`type`, `creatorList`, `title`, `doi` and `year`) but the API only responds to DOIs. First, all relations that are only connected to IRIs (e.g. `type` and `creatorList`) will be ignored. Afterwards n_p (initial) requests will be send to the API for each remaining relation of the input class in order to understand which relation provides a successful input values for the API. For example, the number of initial requests (n_p) is set to $n_p = 25$. In our tests it turned out that $n_p = 25$ initial requests are usually sufficient to achieve correctness while providing a reasonable speed. In our example, during the probing phase $3 \times 25 = 75$ requests are sent to the API.

The input values for each relation (e.g. DOIs) that are used to request the Web API are picked uniformly at random from entities of the input class in the KB with this relation. This is done to prevent the entities from being very similar to each other and thus increase the probability of an API response. For example, assume that the API in Figure 1 only responds to records with a specific publisher like Springer. If the entities are selected in any particular way, e.g. according to the amount of facts the KB contains about them, it is possible that entities of Springer are not included. This may be because the local KB is more likely to contain publications from other publishers. In this case, the API receives requests that cannot be answered since no data about other publishers is provided. As a consequence, no successful responses will be collected and no aligning can be done.

After sending a request to the API it can respond in several ways. In the best case, the API responds with the HTTP status code `200 OK` or with an HTTP error code (e.g. `404 Not Found`). In the worst case, the server responds with a JSON document containing an error message. This is done to show the requesting instance that the input does not produce a result. For example,

sending a year or ISBN to the API in Figure 1 would be unsuccessful because it only accepts DOIs. However, in this case the system cannot easily detect that some input values did not lead to a (successful) response and therefore will send further requests to the API (with the corresponding relation e.g. `year`) in the next step. Since this would result in a considerable increase of requests and runtime it is important to identify error messages to prevent the system from sending additional requests to the API.

In order to identify error responses, the system iterates over all answers and compares how similar they are to one another. This procedure is based on the observation that error messages are always very similar or even the same. For example, they usually contain all the same error message like "This is not a valid input" or consist of an generic error message in combination with the request value. In contrast, correct answers are different to one another since they contain information regarding various different entities. As a result, a (potential) error response is determined by counting how often a response was similar (by using Levenshtein) to other responses. The response that is most often similar (i.e. the similarity needs to be higher than θ_{err}) to other responses is considered an error message. Then all responses similar to this response will be deleted and all relations r_{in} which have not received enough (valid) answers will no longer be considered as valid input relations. In this way unnecessary requests to the API are prevented.

Next, the alignment phase begins by only using the relations $r \in R_{in}$ that have led to valid answers. The aligning phase itself is divided into two parts: (1) determining candidate alignments and (2) determining the final alignments.

6.3 Aligning Phase: Candidate Alignment

Algorithm 1 shows details of the candidate alignment procedure. The candidate alignment takes as input the set of valid input relations R_{in}, the chosen KB K and the corresponding schema $S = (C, R_c, R_{id})$. In the following we write $S.C$ to indicate that we access the C component of S (analogously for the other components). For each input relation $r_{in} \in R_{in}$, the algorithm sends further requests to the API using only the previously determined relations in R_{in} (e.g. `doi`) that lead to successful responses. The default value for additional requests (n_r) is set to $n_r = 75$ (which can be changed by an expert user via the configuration file), so that in combination with the 25 requests from the probing phase there are $n_s = n_p + n_r = 100$ requests for each input relation; n_s is called sample size.

A similar procedure to that in the probing phase is followed in the aligning phase. A random entity e in K is selected with the type of the input class (line 6). Note, the input class for each API is defined in the configuration files of FiLiPo by the user. This is done by selecting entities of the given input class from K via a SPARQL query, setting the query's `LIMIT` parameter to 1 and its `OFFSET` parameter to a random value between 0 and the number of instances of the input class (stored in $S.C$). Additionally, FiLiPo retrieves the set of all facts rec that K contains for the chosen entity e in the form of relation-value triple $r(e, l)$. Like Koutraki et al. [13] we take all facts into account up to depth three. This depth was chosen because all other facts usually do not make any more statements about the entity e. In the example of Figure 2 this would mean that for the entity `PaperEntity` the name of the author entity would also be considered as information about `PaperEntity`. To exclude the case that entities are connected to other entities in only one direction, inverse relations are also

output : Set of candidate alignments A_c
input : Input relations R_{in}, Knowledge Base K, Schema of Knowledge Base S

1 $A_c \leftarrow \emptyset$;
2 **foreach** $r_{in} \in R_{in}$ **do**
3 Multiset $A_{r_{in}} \leftarrow \emptyset$;
4 **for** $i \leftarrow 1$ **to** n_S **do**
5 $R \leftarrow \emptyset$;
6 $(e, rec, v_{req}) \leftarrow$ `GetRandRec`$(K, S.C, S.R_c)$;
7 res $\leftarrow$ `ApiCall`(v_{req});
8 **foreach** $(r(e, l)) \in rec$ **do**
9 **foreach** $p(v)) \in res$ **do**
10 **if** $l \in I \cup \mathbb{R}$ ***or*** $v \in I \cup \mathbb{R}$ **then**
11 $m \leftarrow equals()$
12 **else**
13 $m \leftarrow argmax_{m \in M_{sim}} m(l, v)$
14 **end**
15 **if** $r \in S.R_{id}$ **then**
16 $m \leftarrow$ `IdCompare`$(r(e, l), p(v))$
17 **end**
18 **if** $m(l, v) \geq \theta_{str}$ **then**
19 $R \leftarrow R \cup \{(r, p, m)\}$
20 **end**
21 **end**
22 **end**
23 **if** $\frac{|R|}{min(|rec|, |res|)} \geq \theta_{rec}$ **then**
24 $A_{r_{in}} \leftarrow A_{r_{in}} \cup R$
25 **end**
26 **end**
27 $A_c \leftarrow A_c \cup \{(r_{in}, A_{r_{in}})\}$;
28 **end**

Algorithm 1: Determining Candidate Alignment

considered. For example, `PaperEntity` is connection to an author via the relation `creatorList`. If the Web API expects a `Person` as input class, the papers written by the author would not be considered. This is because only the relation `creatorList` is contained in K but no corresponding inverse relation. For this reason FiLiPo uses $S.R_c$ to find out which relations exist for the input class e.g. for `Person` entities and how they are connected to other classes. Then FiLiPo uses the circumflex operator (to denote inverse relations) introduced in SPARQL 1.1 to access information like `PaperEntity` by using the inverse relation of `creatorList`. Afterwards FiLiPo calls the API with the value v_{req} of the currently considered input relation r_{in} of the randomly chosen entity e and stores the response in res (line 7).

The next step is to find all relation matches R between the fact set rec and the response res (lines 8-25). The set rec contains relation-value triples of the form $r(e, l)$ and res encodes information from the response as path-value pairs of the form $p(v)$ where p is the path in the response from the root to the value v. To realise this, all values l of the local record rec must be compared with all values v of the API response res. For each such pair $(r(e, l), (p(v))$, the best comparison method is now determined (lines 10-17), which will later be used to decide if $r(e, l)$ and $p(v)$ match (lines 18-19). If l or v is an IRI, it is important that they are compared with `equals`. IRIs are identifiers and therefore only

the same if they are identical. The same is true for numerical values. Since it is unclear how to check numbers for similarity the `equals` method is used here as well (lines 10-11). In all other cases FiLiPo uses a set M_{sim} of fifteen different similarity methods with several variants since one string similarity method is not sufficient to compare all different data types (lines 12-14). The method $m \in M_{sim}$ that returns the largest similarity of l and v is considered (temporarily) to be the best method to compare both values and is stored for the later process of FiLiPo.

As discussed before, fuzzy similarity measures are not appropriate for identifier relations, but comparing them for equality would be too strict; identifier relations are therefore handled in lines 15-17. If r is an identifier relation, it is included in the component $S.R_{id}$. In that case, the method `IdCompare()` is used to compare identifiers; details for this method are given below.

Once the best similarity function has been determined, and if the functions yields a similarity of at least θ_{str}, the triple (r, p, m) is created and added to the set of record matches R (lines 18-20).

If enough matches are found, it is assumed that the input entity and the API response overlap in their information and that the API has actually responded with information about the requested entity. We compute this overlap by dividing the number of matches $|R|$ by the number of pairs of the smallest record rec or res as we only count for which relations a mapping could be found. If this is greater than θ_{rec}, the overlap is considered sufficient and the matches will be added to $A_{r_{in}}$ (lines 23-25). This set represents the set of matches found for the input relation r_{in}. The matches are stored separately for each input relation, because the structure of the API response may depend on the input, e.g. requesting the API by using DOIs or titles. If not enough matches are found, it is assumed that the API has responded with information of a different entity; in this case, any matches found between the records must be ignored. In the last step all matching relations for each input relation are stored in a set A_c (line 27).

In summary, the sets $A_{r_{in}}$ contain the matches (r,p,m) for which there is an existing similarity method m which has determined a sufficiently large similarity. In the alignment phase, the relation matches r and p are determined which are most often matched with a certain similarity method m. This will filter out erroneous matches with unsuitable similarity methods.

Comparing Identifiers. As previously stated,`IdCompare()` (line 16) is used to compare identifiers. It provides two alternatives for the comparison that can be chosen in the configuration file. The first one is a regular expressions based approach and the second uses a pre-trained classifier.

The regular expression approach applies the $equals()$ method to normalised values of l and v, shortly $equals_\eta(l, v)$. A predefined rule set N, containing different identifier rules η in form of regular expressions, is used for this normalisation. Such a predefined rule could, for example, remove the hyphens from an ISBN. Since it is not known what type of identifier is modeled by the relation r, all normalisation rules are applied successively to l and v. Our assumption is that different notations of the same identifier are equal only with one specific normalisation because all other normalisations should change the values too much.

The second approach utilises a gradient boosting classifier working on Flair [1] embeddings of identifier values to determine whether two values are equal. We use Flair embeddings instead of others since this framework is character-based and therefore suits better for the comparison of two identifier values. The classifier is only used if there is a method $m \in M_{str}$ for which $m(l, v) \geq \theta_{sim}$. This is done because the comparison by using the classifier is expensive in terms of computation time. Thus, only values which are similar enough to be a possible match are compared.

6.4 Aligning Phase: Final Alignment

Matching paths without branching points, such as `label("Some example Title")`, is easy. Here we can simply match `label` with `title`. However, for paths with a branching point, we need to decide if all entries of the corresponding array provide the same type of information or different types. In the former case, which could be an array specifying the authors of a paper, we need to match *all paths* that are equal (with exception of index values) of the API response with the same relation. This is the case in the example in Figure 1 for the path `authors.*.given`. In the latter case, each different index value at the branching point should be mapped to one specific relation, possibly different relations for the different values. In the example, *facets.0.value("2020")*) always denotes the year of the publication, whereas *facets.1.value("Computer Science")*) (not shown in the example) could denote the genre of the publication. Therefore, either the year or the genre relation of K to `facets.*.value` should be prevented. As an orthogonal problem, some relations in the KB (like `fullName` in the example in Figure 2) have to be matched with multiple paths from the API response, e.g. `given` and `family` in Figure 1. We call such matches *one-to-many matches*.

In order to determine the final alignment and to solve the problems mentioned above, FiLiPo distinguishes three cases: (1) Determination of the one-to-many matches, (3) fixed path matches and (3) branching point matches. To create a valid alignment for each input relation r_{in} contained in A_c, the corresponding multi sets $A_{r_{in}}$ will be transformed to new multi sets $B_{r_{in}}$ that encode branching points explicitly. First, for each $(r, p, m) \in A_{r_{in}}$ we determine the number n of equal tuples (with exception of index values) contained in the multi set $A_{r_{in}}$. Afterwards, all path index values in p will be replaced by the branching point symbol and denoted as $P*$. Finally, for each $(r, p, m) \in A_{r_{in}}$ a tuple of the form $(r, P*, m, n)$ is added to $B_{r_{in}}$.

Afterwards, for every relation r for which at least one tuple exists in $B_{r_{in}}$ we determine the path $P*$ with maximal n (i.e., which was most frequently matched to r), regardless of which method m was used. Furthermore it is important that the relation r and the path $P*$ are matched sufficiently frequent. If such a path was found only a few times, e.g. once in 100 responses of the API, then the match is not very strong. We therefore calculate the confidence for the matching as $\frac{n}{n_v}$ where n_v represents the number of successful API responses. This confidence must be greater than θ_{rec}. We reuse θ_{rec} here based on the assumption that the overlapping of records is also reflected in the overlapping of relations. In practice and during the evaluation we were able to achieve good results with this assumption. More formally, we search for relations r and paths $P*$ for which it holds that

$$\exists (r, P*, m, n) \in B_{r_{in}} \text{ s.t. } \forall (r, P*, m', n') \in B_{r_{in}} : n \geq n'$$
$$\text{and } \frac{n}{n_p + n_a} \geq \theta_{rec}$$

To identify one-to-many matches, we now identify entries $(r, P'*, m', n') \in B_{r_{in}}$ with the same relation r and the same prefix in path $P*$ which could be matched similarly often as $(r, P*, m, n)$. The idea behind this is that branching point paths like `authors.*.family` and `authors.*.given` are matched similarly often (i.e. $n' \geq n \times \theta_{comb}$) with the corresponding relation `creatorList.$.fullName` and therefore are recognised as combined match. An additional condition is that one-to-many

matches must have the same prefix (up to the branching point), e.g. `authors*`. More formally

$$P_m = \{P'* : (r, P'*, m', n') \in B_{r_{in}} \text{ s.t. } \frac{min(n, n')}{max(n, n')} >= \theta_{comb} \text{ and } P = P' \text{ for } P* = P * p \text{ and } P'* = P' * p'\}$$

If this is the case, the identified paths in set P_m together with the original path $P*$ are considered as one-to-many matches for r. However, before adding these matches to the set of final alignments, a last step has to be done. For every path in $P_m \cup \{P*\}$ all equal paths $p \in A_{r_{in}}$ except for their index values are collected. Then the longest common prefix is identified in order to determine at which position the path can actually branch. Since a path of an API response can contain more than one array index as nodes, it is necessary to check where the path really branches off. An example is a path of the form $record.0.author.*.name$. The Web API in this example always responds with the array `record` containing only one element, which in turn contains the actual response. Therefore it would be inaccurate to take this array as branching point. All index values that appear after the longest common prefix will be taken as branching points again and added to the final set $A_{r_{in}}$.

If $|P_m \cup \{P*\}| < 2$ the relation r and path $P*$ are no longer considered as a one-to-many match. Therefore the next step is to check if the pair $(r, P*)$ is a branching point match or a fixed path match. Therefore the set multi set $B_{r,P*}$ is constructed as follows:

$$B_{r,P*} = \{(r', P*', m', n') : \forall (r', P'*, m', n') \in B_{r_{in}} \text{ s.t. } r' = r \text{ and } P'* = P*\}$$

$|B_{r,P*}|$ denotes the number of found paths $P*$ which can be matched with the relation r. Each path $P*$ had a different index value in the path before converting to a branching point path. If $|B_{r,P*}| < 2$ there was only one path to find the information of r in the API response. An example is the path `facets.0.value` in Figure 1. To match the relation `year` there is always only this fixed path, because the first place of the JSON array `facets` always describes the year of publication and never contains any other information. So if less than two paths are found, it is not a branching point match but a fixed path match. To ensure that it is a valid fixed path match, the number of times this path could be matched is checked. Therefore it has to yield $n \geq (n_v \times \theta_{rec})$ with $(r, P*, m, n) \in B_{r,P*}$. As previously, for all paths in $B_{r,P*}$ the correct branching points will be determined by searching the longest common prefix and replace only the index values that appear after that prefix by the branching point symbol. Afterwards the relation-path match is added to $A_{r_{in}}$.

Some relations and paths are dependent on the previous entity. In order to match the affiliation path for an author we have to include the whole author array of the JSON response because matching only one specific path (e.g. `authors.0.inst`) would not be sufficient. Therefore if $|B_{r,P*}| \geq 2$, it is possible that $(r, P*)$ is a branching point match. A match of a relation r and a branching point path $P*$ is considered valid if the following two conditions are satisfied: (1) if the relation r has led to a match often enough, i.e. number of triples containing r in $A_{r_{in}}$ has to be greater or euals to $n_v \times \theta_{rec}$, and (2) if the matched path $P*$ occurs frequently enough in all matches with the relation r, or more formally,

$$\frac{\sum_{(r',p',m',n') \in B_{r,P*}: r'=r} n'}{|\{(r', p', m') \in A_{r_{in}} : r' = r\}|} \geq \theta_{rec}$$

If both conditions are met, the match between r and $P*$ is considered a branching point match and added to $A_{r_{in}}$. Note that before adding to $A_{r_{in}}$, the most common prefix will be determined to only insert the "real" branching points. In all other cases, the match is discarded. The intuition behind this is that if a path does not occur frequently enough as a branching point path, it cannot occur frequently enough as a fixed path.

For the sake of simplicity, one aspect has not yet been considered in detail. Some relations can also potentially be matched with multiple paths in the JSON response. For example, the relation for the publication year could be incorrectly matched with the path to the publication years of the article's references. To mitigate such errors, a reciprocal discount is used, i.e. the number n of matches found for a possibly incorrect path p and a relation r is discounted by the length difference of the paths to $n/|(len(r) - len(p))|$. Thus paths with the same length as the KB are preferred.

Finally, the set $A_{r_{in}}$ contains all valid matches found for the input relation r_{in}. These can be one-to-many matches, branching point matches or fixed path matches.

6.5 Additional Statistics

FiLiPo is able to provide additional information and statistics about the API that may be useful, for example, if the API should be called during the execution of a SPARQL query on the original knowledge base (which is beyond the scope of this paper). One example is the frequency of various information items in the API response, e.g. titles may be more often available than affiliation information. In addition, the probability of receiving a valid response when the API is requested is also estimated. If several Web APIs are aligned with a local KB, the API that provides all or at least the largest share of the required information and at the same time has a high probability that the searched information is available can be selected for on-the-fly information integration.

Joint Feature Determination. FiLiPo also determines the joint feature determination. In this process it is determined which information (features) the input entities had in common that led to a valid answer of the API. As already mentioned, the API in Figure 1 only provides information about publications published by Springer. The joint feature determination will therefore find out that the API only yields valid results if there is a relation in the KB for the requested resource that indicates that the publisher is Springer.

In order to achieve this, FiLiPo follows a similar approach to that used for the analysis of shopping baskets. Namely using support and confidence metrics to filter joint features. The corresponding support and confidence functions are defined as follows:

$$supp(r(e, v), R_v) := \frac{|\{r(e, v') : r(e, v') \in R_v\}|}{n_v}$$

$$conf(r(e, v), R_v) := \frac{|\{r(e, v) : r(e, v) \in R_v\}|}{|\{r(e, v') : r(e, v') \in R_v\}|}$$

First, all information of the entities e with which the API was requested is divided into two sets. All relation-value triples $r(e, v)$ that led to a successful response (record-response overlap greater than θ_{rec}) are collected in R_v, all others are stored in R_n. In the following the two sets R_v and R_n are regarded as shopping baskets. Next for each $r(e, v) \in R_v$ the *support* is calculated and only those $r(e, v)$ with sufficient support (i.e. $supp(r(e, v), R_v) \geq \theta_{supp}$) are included in the set F_v.

Similarly the support for each $r(e, v) \in R_n$ is calculated and added to F_n. Then all non-selective relation-value triples are removed and the joint feature set F is created. Non-selective in this context means that the relation-value triples are included in both F_v and F_n. Therefore F is determined by $F = F_v \setminus F_n$.

The confidence describes the conditional probability that the information of an entity e contains the relation r which points to the value v. The confidence therefore is an indicator of how often the relation r was found in combination with the value v. For this reason, for all relation-value triples in F the confidence value will be computed as a last step and all relations-value triples with $conf(r(e, v), R_v) < \theta_{conf}$ will be removed from F.

7 EVALUATION

Used Datasets. We have evaluated FiLiPo and DORIS on two local KBs, six bibliographic APIs and one movie API. The first local KB is an RDF version of the dblp dataset[5] which contains bibliographic metadata about major computer science publications. The second local KB is the Linked Movie DB[6] which contains information on movies. The used APIs are SciGraph [7], CrossRef [8], Elsevier[9], ArXiv[10] and two APIs provided by Semantic Scholar[11] (one with DOIs and one with ArXiv keys as input parameters). All of these APIs respond with metadata about scientific articles and their authors. In order to align the Linked Movie DB with an API we used the Open Movie Database (OMDB)[12]. It responds with metadata about movies, e.g. movie director, known actors and movie genres.

FiLiPo Configuration. We assume that the user of the FiLiPo system is a non-technical user without programming knowledge or technical skills. Furthermore, the user has no in-depth knowledge of external data sources, but is familiar with the structure of the local KB. We assume that the user has domain knowledge and therefore can understand common data structures from the genre of the local database (e.g. bibliographic meta data). In addition, an expert with knowledge of the APIs (technical-user) can make further settings (e.g., changing string similarity thresholds) to fine-tune the system. Therefore, we divided the APIs into two sets: (1) non-technical evaluation set and (2) technical evaluation set. The non-technical set contains CrossRef, SciGraph, Semantic Scholar (with the two different APIs) and Elsevier. All APIs from the non-technical evaluation set were executed with the default settings of FiLiPo, i.e. 25 probing requests with 75 additional requests. The thresholds are set to $\theta_{str} = 0.5$ (string similarity threshold), $\theta_{rec} = 0.1$ (record overlap), and $\theta_{comb} = 0.3$ (for the determination of one-to-many matches).

Since Elsevier and dblp only have few publications in common it was ensures that only entities published by Elsevier are used. In this way it is possible to use the API to evaluate FiLiPo although the intersection of dblp and Elsevier is small. This was also done for DORIS in order to allow a sound comparison.

The technical-user set consists of ArXiv and OMDB. ArXiv was chosen for this set because it always responds with a list of the top most similar entities to the requested one. The only exception is when receiving an ArXiv key. Since FiLiPo is at this stage not able to pick the requested record from a list of top-k similar records the probing phase will not be successful. Because we assumed that a technical-user has knowledge of the used APIs, the user is able to provide a valid input relation (i.e. a relation modelling ArXiv keys) to FiLiPo. In this way the probing phase will be skipped and the aligning phase starts by using the provided relations. For the same reason we restricted OMDB to using titles only. The number of probing requests and the values of the thresholds are set to the same defaults used in the non-technical evaluation set.

Experiments. As a gold standard, we designed the correct path alignments manually for each API. Since FiLiPo pulls random records from the local KB and uses them to request the API, the alignments found may differ slightly between different runs. For this reason, the evaluation was performed three times for each combination of KB, API and identifier method (e.g. RegExer or Classifier). The runtime of FiLiPowas around 15 minutes for both, the regular expression approach and the classifier approach. If the input relations are known, as is the case with DORIS, then the system only needs not longer than 8 minutes due to the lack of a probing phase. The probing phase is expensive in runtime because a significant number of requests are sent to the API. This happens for each relation of the input class that points to literal values. With 25 probing requests as default this would be in the case of the dblp with 27 possible relations for Publications $25 \times 27 = 675$ requests during the probing phase.

FiLiPo was able to determine the correct input relations for all APIs contained in the non-technical evaluation set. Error messages such as those returned by SciGraph (a JSON response) were successfully identified in all cases. Thus the runtime and the number of requests to the API were kept relatively small.

For the evaluation we used the metrics Precision, Recall and F1 Score. FiLiPo was able to achieve a precision of 0.83 to 1.00 and a recall of 0.53 to 1.00. Values close to 1.0 were achieved mainly because there were only a few possible alignments. This means that outliers, caused by the random drawing of entities, have been extremely hit in the evaluation. The corresponding f1 scores for FiLiPo are between 0.68 and 1.00. The average f1 score for the regular expression approach is 0.88 and for the classifier 0.85. Both the classifier and the regular expression approach have performed equally well. However, the regular expression based approach has the advantage that it can be extended more easily as only appropriate normalisation rules (in the form of a regular expression) have to be specified. In contrast, a classifier has to be (re-)trained and strongly depends on the training data set. This requires considerably more expertise than the regular expression-based approach.

Comparison to Baseline. We re-implemented DORIS as a baseline system for our evaluation. DORIS uses label information of instances as its predefined input relation for APIs. However, this is not always the appropriate input parameter for the API. For example, some APIs expect DOIs or ISBNs as input parameters. In order to extend the set of APIs that can be used with DORIS, we modified DORIS such that the input relation can be specified by the user. This allows to use DORIS with more sources than its original specification would allow. Since DORIS is not random-based when selecting entities, unlike FiLiPo, there was no need for multiple test runs. DORIS has been configured in order to send 100 requests to the APIs. Furthermore, the threshold for the Overlap and PCA confidence has been set to 0.3.

As we can see in Table 1, our system performed better than DORIS in most cases. The strengths of DORIS are particularly evident in the alignment of dblp and Semantic Scholar with ArXiv

[5] provided by dblp: https://basilika.uni-trier.de/nextcloud/s/A92AbECHzmHiJRF
[6] http://www.cs.toronto.edu/~oktie/linkedmdb/linkedmdb-18-05-2009-dump.nt
[7] https://scigraph.springernature.com/explorer/api/
[8] https://www.crossref.org/services/metadata-delivery/rest-api/
[9] https://api.elsevier.com
[10] https://arxiv.org/help/api
[11] https://api.semanticscholar.org
[12] http://www.omdbapi.com

Table 1: Evaluation of Runtime, Precision and Recall of FiLiPo

Systems		FiLiPo									DORIS		
Data Sets	Mode	P			∅ P	R			∅ R	F1	∅P	∅R	F1
dblp ↔ CrossRef (DOI)	RegEx/Overlap	0.97	0.89	0.95	0.94	0.79	0.76	0.86	0.80	**0.86**	1.00	0.24	0.38
	GBC/PCA	0.86	0.97	0.94	0.92	0.90	0.69	0.76	0.79	**0.85**	0.67	0.29	0.40
dblp ↔ SciGraph (DOI)	RegEx/Overlap	1.0	1.0	1.0	1.0	0.95	0.74	0.95	0.88	**0.93**	n/a	n/a	n/a
	GBC/PCA	1.00	1.00	1.00	1.00	0.89	0.84	0.89	0.88	**0.93**	0.90	0.47	0.62
dblp ↔ S2 (DOI)	RegEx/Overlap	1.00	1.00	0.94	0.98	0.87	0.87	1.00	0.91	**0.94**	1.00	0.53	0.70
	GBC/PCA	0.94	1.00	1.00	0.98	1.00	0.67	0.87	0.84	**0.91**	1.00	0.67	0.80
dblp ↔ S2 (arXiv)	RegEx/Overlap	1.00	1.00	1.00	1.00	0.80	0.70	0.70	0.73	0.85	1.00	1.00	**1.00**
	GBC/PCA	1.00	1.00	1.00	1.00	0.90	0.70	0.70	0.77	0.87	1.00	0.67	**0.80**
dblp ↔ Elsevier (DOI)	RegEx/Overlap	0.88	0.93	1.00	0.94	0.79	0.74	0.79	0.77	**0.85**	n/a	n/a	n/a
	GBC/PCA	0.93	0.93	0.93	0.93	0.74	0.74	0.74	0.74	**0.82**	1.0	0.53	0.69
LMDB ↔ OMDB	RegEx/Overlap	0.92	0.91	0.91	0.91	0.92	0.77	0.77	0.82	**0.86**	0.86	0.46	0.60
	GBC/PCA	1.00	1.00	1.00	1.00	0.46	0.62	0.46	0.51	**0.68**	0.55	0.46	0.50
dblp ↔ ArXiv (arXiv)	RegEx/Overlap	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	**1.00**	1.00	1.00	1.00
	GBC/PCA	0.75	1.00	0.75	0.83	1.00	1.00	1.00	1.00	**0.91**	1.00	0.50	0.67

Keys as inputs. Here DORIS can achieve better results, especially in recall. This is due to the fact that DORIS sorts all entities of the input class in descending order of the number of their facts. Thus ArXiv papers are sorted upwards that also appear in a journal. In contrast to FiLiPo, DORIS is able to match journal-related relations. Since the dblp contains many preprints of ArXiv, but only few of them are published in a journal, these alignments are lost in FiLiPo.

The weaknesses of DORIS become apparent in the alignment of the dblp and SciGraph. While the PCA confidence method provides useful results in the area of precision, not a single relation can be aligned using the Overlap confidence. In contrast to FiLiPo, DORIS has not implemented any procedures to detect error responses. As the dblp and SciGraph overlap in one third of their data, this means that in the best case the overlap confidence is not higher than 0.33. If, for example, a title of a publication does not match a few times, the confidence is too low and relation alignments are lost. The reason for the low recall when using the PCA method is due to the used string similarity. The method is too strict and this leads to a loss in recall. Similar problems lead to the low recall when aligning dblp and Elsevier.

8 CONCLUSION

We presented a system to discover alignments between KBs and APIs in an automatic manner. A user only needs knowledge about the KB but no prior knowledge about the API data schema. Our evaluation shows that in summary FiLiPo has performed very well compared to DORIS. In contrast to DORIS, it was able to determine an alignment in all cases. Overall FiLiPo delivered better results in six out of seven cases. Only in one case, DORIS was able to achieve better results than FiLiPo. Nevertheless, FiLiPo is currently only able to work with APIs that return a single record.

REFERENCES

[1] A. Akbik, D. Blythe, and R. Vollgraf. 2018. Contextual String Embeddings for Sequence Labeling. In *COLING 2018*. 1638–1649.

[2] D. Aumueller, H. H. Do, S. Massmann, and E. Rahm. 2005. Schema and ontology matching with COMA++. In *SIGMOD 2005, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 906–908. https://doi.org/10.1145/1066157.1066283

[3] S. Baltes, L. Dumani, C. Treude, and S. Diehl. 2018. SOTorrent: reconstructing and analyzing the evolution of stack overflow posts. In *MSR 2018*. ACM, 319–330.

[4] F. Benedetti, S. Bergamaschi, and L. Po. 2014. Online Index Extraction from Linked Open Data Sources. In *LD4IE 2014, Riva del Garda, Italy, October 20, 2014 (CEUR Workshop Proceedings)*, Vol. 1267. CEUR-WS.org, 9–20.

[5] P. A. Bernstein, J. Madhavan, and E. Rahm. 2011. Generic Schema Matching, Ten Years Later. *Proc. VLDB Endow.* 4, 11 (2011), 695–701.

[6] I. F. Cruz, F. P. Antonelli, and C. Stroe. 2009. AgreementMaker: Efficient Matching for Large Real-World Schemas and Ontologies. *Proc. VLDB Endow.* 2, 2, 1586–1589. https://doi.org/10.14778/1687553.1687598

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. M. Domingos. 2004. iMAP: Discovering Complex Mappings between Database Schemas. In *SIGMOD 2004, Paris, France, June 13-18, 2004*. ACM, 383–394. https://doi.org/10.1145/1007568.1007612

[9] A. Hogan, A. Polleres, J. Umbrich, and A. Zimmermann. 2010. Some entities are more equal than others: statistical methods to consolidate linked data. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable Dynamic*.

[10] P. Jain, P. Hitzler, A. P. Sheth, K. Verma, and P. Z. Yeh. 2010. Ontology Alignment for Linked Open Data. In *ISWC 2010, Shanghai, China, November 7-11, 2010 (Lecture Notes in Computer Science)*, Vol. 6496. Springer, 402–417. https://doi.org/10.1007/978-3-642-17746-0_26

[11] M. Koutraki, N. Preda, and D. Vodislav. 2016. SOFYA: Semantic on-the-fly Relation Alignment. In *EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016*. OpenProceedings.org, 690–691. https://doi.org/10.5441/002/edbt.2016.89

[12] M. Koutraki, N. Preda, and D. Vodislav. 2017. Online Relation Alignment for Linked Datasets. In *ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 10249. 152–168. https://doi.org/10.1007/978-3-319-58068-5_10

[13] M. Koutraki, D. Vodislav, and N. Preda. 2015. Deriving Intensional Descriptions for Web Services. In *CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*. ACM, 971–980. https://doi.org/10.1145/2806416.2806447

[14] M. Koutraki, D. Vodislav, and N. Preda. 2015. DORIS: Discovering Ontological Relations In Services. In *ISWC 2015, PA, USA, October 11, 2015 (CEUR Workshop Proceedings)*, Vol. 1486. CEUR-WS.org.

[15] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. 2005. Corpus-based Schema Matching. In *ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 57–68. https://doi.org/10.1109/ICDE.2005.39

[16] J. Madhavan, P. A. Bernstein, and E. Rahm. 2001. Generic Schema Matching with Cupid. In *VLDB 2001, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 49–58.

[17] Li Qian, Michael J. Cafarella, and H. V. Jagadish. 2012. Sample-driven schema mapping. In *SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 73–84. https://doi.org/10.1145/2213836.2213846

[18] E. Rahm and P. A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB J.* 10, 4 (2001), 334–350. https://doi.org/10.1007/s007780100057

[19] T. Sahay, A. Mehta, and S. Jadon. 2019. Schema Matching using Machine Learning. *CoRR* abs/1911.11543.

[20] O. Schmidts, B. Kraft, I. Siebigteroth, and A. Zündorf. 2019. Schema Matching with Frequent Changes on Semi-Structured Input Files: A Machine Learning Approach on Biological Product Data. In *ICEIS 2019, Heraklion, Crete, Greece, May 3-5, 2019, Volume 1*. SciTePress, 208–215. https://doi.org/10.5220/0007723602080215

[21] F. M. Suchanek, S. Abiteboul, and P. Senellart. 2011. PARIS: Probabilistic Alignment of Relations, Instances, and Schema. *Proc. VLDB Endow.* 5, 3, 157–168. https://doi.org/10.14778/2078331.2078332

[22] O. Udrea, L. Getoor, and R. J. Miller. 2007. Leveraging data and structure in ontology integration. In *SIGMOD 2007, Beijing, China, June 12-14, 2007*. ACM, 449–460. https://doi.org/10.1145/1247480.1247531