This is a preprint of the article: Michalis Mountantonakis, Yannis Tzitzikas, "Applying Cross-Dataset Identity Reasoning for Producing URI Embeddings over Hundreds of RDF Datasets", Accepted for Publication in *International Journal of Metadata, Semantics and Ontologies*, December 2020

Applying Cross-Dataset Identity Reasoning for Producing URI Embeddings over Hundreds of RDF Datasets

Michalis Mountantonakis[1,2] and Yannis Tzitzikas[1,2]
[1]Institute of Computer Science - FORTH-ICS, Greece and
[2]Computer Science Department - University of Crete, Greece
e-mail: {mountant|tzitzik}@ics.forth.gr

Abstract

There is a proliferation of approaches that exploit RDF datasets for creating URI embeddings, i.e., embeddings that are produced by taking as input URI sequences (instead of simple words or phrases), since they can be of primary importance for several tasks (e.g., machine learning tasks). However, existing techniques exploit either a single or a few datasets for creating URI embeddings. For this reason, we introduce a prototype, called `LODVec`, which exploits `LODsyndesis` for enabling the creation of URI embeddings by using hundreds of datasets simultaneously, after enriching them with the results of cross-dataset identity reasoning. By using `LODVec`, it is feasible to produce URI sequences by following paths of any length (according to a given configuration), and the produced URI sequences are used as input for creating embeddings through *word2vec* model. We provide comparative results for evaluating the gain of using several datasets for creating URI embeddings, for the tasks of classification and regression, and for finding the most similar entities to a given one.

1 Introduction

There is an increasing trend of exploiting knowledge graphs (e.g., [32, 10]) for creating embeddings which can be exploitable for a number of tasks. Indicatively, they can be used for i) machine learning-based tasks (e.g., [29]), such as classification and regression, ii) similarity-based tasks (e.g., [16]), for answering queries like "Give me the top-K related entities to a given one", iii) link prediction purposes (e.g., [23]), iv) fact validation (e.g., [1]), v) language translation (e.g., [22]) and others (e.g., [32, 10]). There is also a proliferation of novel methods (e.g., [29, 30]), that exploit LOD (Linked Open Data) and RDF (Resource Description Frameworks) knowledge graphs (or datasets) for creating embeddings for URIs (Uniform Resource Identifiers). The key difference of URI

embeddings comparing to word embeddings is that they are produced by taking as input a sequence of URIs (instead of simple words or phrases). In particular, such methods produce URI sequences for a set of given entities, i.e., URI sequences which start from a focused entity, and contain a path of URIs which are reachable from this entity. These URI sequences are given as input for producing URI embeddings through neural networks models, such as *word2vec* [15], and *GloVe* [25], where each URI is mapped to a vector of real numbers. The produced embeddings can be used in any of the above tasks.

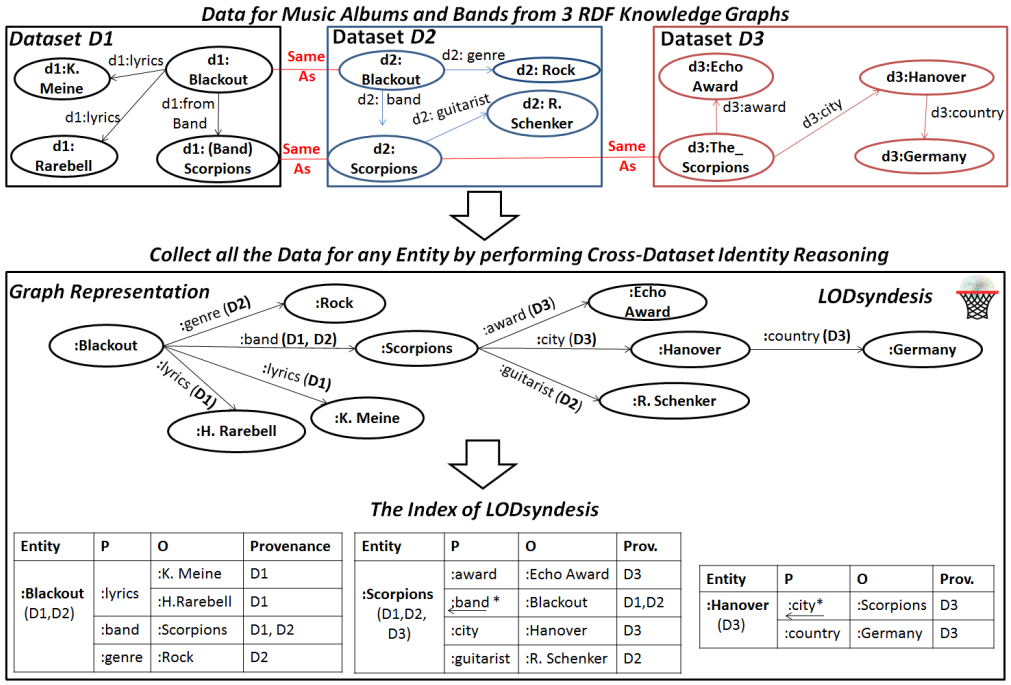
However, current approaches exploit usually a single RDF dataset for creating URI embeddings for one or more entities. Moreover, many approaches are difficult to be configured by non-experts, since they do not provide an interactive service. Our objective is to make it feasible to create URI sequences and URI embeddings for any given entity (i.e., a URI), by combining data from hundreds of RDF datasets simultaneously.

As a motivating example, suppose that we desire to predict the exact user rating of one or more music albums, like "Blackout" (see Fig. 1), which is an album from the German rock band "Scorpions". For this reason we plan to create embeddings from URI sequences, for using them as features in such a machine learning task. In this example, we can see that there exists three available knowledge graphs (or datasets), two of them contain information about the music album "Blackout", and the last one information about the music band "Scorpions". Through this example, we desire to show the importance of using multiple datasets, for creating a) complementary URI sequences, and b) "mixed-path" URI sequences, for the desired entities (e.g., music albums).

Regarding a), by selecting to use only a single dataset, say D_1 in Fig. 1, we can find valuable information such as the people that written the lyrics of each music album. However, D_1 does not contain information about the genre of each album, e.g., in Fig. 1 such data occur in dataset D_2. Therefore, only by using both datasets D_1 and D_2, we will be able to create URI sequences including both the lyrics and the genres of each album, i.e., by using complementary data.

Regarding b), suppose that we desire to create larger URI Sequences, i.e., sequences including a path of n edges (or n triples). By following such paths, it is feasible to discover more information for the neighbours of a given entity, and we assume that such information can be of primary importance for improving the accuracy of predictions. In Fig. 1, suppose that we desire to create URI sequences that include more information about the band of each music album, e.g., URI sequences containing "a music album, its band, and the awards won by this band". In our example, there is not a single dataset (or graph) that contains such a path. In particular, from datasets D_1 and D_2 we can find the band of a music album, and from dataset D_2 information about the guitarist of the band. However, it does not contain any additional information about bands. On the contrary, dataset D_3 contains more data about bands, such as data about the awards won by a band. Therefore, for creating the desired URI sequence, it is mandatory to follow a "mixed-path", i.e., a path that includes data from at least two datasets, e.g., D_2 and D_3.

However, it is not trivial to collect and integrate all the available information for any given entity from several datasets, which is essential for creating either complementary or "mixed-path" URI sequences. The major integration difficulties that are related to our desired task (i.e., creating URI sequences from multiple datasets) are the following: (Diff. 1) datasets use different URIs and models for representing the same real world objects, and (Diff. 2) data are scattered in different places [21]. For instance, in Fig. 1,



Entity	P	O	Provenance
:Blackout (D1,D2)	:lyrics	:K. Meine	D1
		:H.Rarebell	D1
	:band	:Scorpions	D1, D2
	:genre	:Rock	D2

Entity	P	O	Prov.
:Scorpions (D1,D2, D3)	:award	:Echo Award	D3
	:band *	:Blackout	D1,D2
	:city	:Hanover	D3
	:guitarist	:R. Schenker	D2

Entity	P	O	Prov.
:Hanover (D3)	:city*	:Scorpions	D3
	:country	:Germany	D3

Figure 1: Running example containing 3 knowledge graphs and `LODsyndesis`

the three datasets use three different URIs for representing the same band "Scorpions", whereas datasets D_1 and D_2 use different URIs for referring to the same schema element, e.g., see the two URIs for the property "band".

For making it feasible to collect all the data for any entity, e.g., for creating URI sequences by using all the datasets of Fig. 1, it is a prerequisite to identify all the equivalent URIs of each URI for the entities, e.g., d1:(Band) Scorpions `owl:sameAs` d2:Scorpions `owl:sameAs` d3:The_Scorpions, and for the schema elements, e.g., d1:fromBand `owl:equivalentProperty` d2:band. The major problem is that these relationships (i.e., `owl:sameAs` and `owl:equivalentProperty` relationships) model an equivalence relation, and therefore we should compute their transitive closure. However, it presupposes knowledge of all datasets and such a computation can be quite expensive [18].

Based on the above analysis, the major research questions are the following: a) how to overcome the problem of having different URIs for the same entities (and schema elements), for being able to create URI sequences (of any length) from multiple datasets, b) whether a single RDF knowledge graph can outperform all the others for any possible task, and c) whether the accuracy of predictions for several tasks (such as machine learning based tasks) can be increased by creating embeddings from multiple datasets (instead of using one or few datasets), and by following larger paths.

Concerning our contribution, we introduce a research prototype, i.e., `LODVec`, which is accessible through `https://demos.isl.ics.forth.gr/LODVec/`. `LODVec` (i) takes as input one or more entities (e.g., their URIs) and (ii) offers several configurable options for creating URI sequences for the input entities. Afterwards, (iii) it exploits

LODsyndesis knowledge graph for enabling the production of URI sequences and embeddings from hundreds of datasets simultaneously, and (iv) produces URI sequences (of any length) based on a given configuration and by using the notion of basic graph patterns. Moreover, LODVec (v) converts the produced URI sequences into vector representations (i.e., embeddings) by using *word2vec* approach [15] through *dl4j* API (https://deeplearning4j.org/). Finally, it can (vi) exploit the produced vectors for several purposes, e.g., for performing classification and regression tasks by using *WEKA* API [35].

For testing the proposed approach, we report experimental results for machine learning classification and regression tasks by using three evaluation datasets containing thousands of movies, music albums and basketball players. Furthermore, we have created a dataset (by using a Google service) for evaluating the effectiveness of LODVec for identifying similar entities, e.g., finding the most similar basketball players to a given player. We introduce experiments and measurements for evaluating the impact of using multiple datasets and cross-dataset identity reasoning in terms of effectiveness, we compare the performance of different configurations, and we discuss the efficiency of the proposed approach.

This paper is an extended version of the paper [20]. In comparison to that work, this paper is more self-contained. Moreover, in this paper: a) we enrich the related work section, b) we extend LODVec for creating URI sequences of any length by following paths containing two or more edges, whereas in [20] we created URI sequences by using only the direct neighborhood of each entity (i.e., by following single-edge paths), and c) we perform more experiments (by using more evaluation datasets), for evaluating the impact of using multiple datasets for several machine learning tasks, and the efficiency of the proposed approach.

The rest of this paper is organized as follows: §2 introduces the background and related work. §3 provides the problem statement and describes the context, while §4 introduces the steps and the algorithms for creating URI sequences and embeddings. §5 includes the experimental evaluation about the effectiveness and the efficiency of the proposed approach, whereas §6 discusses the results of the evaluation. Finally, §7 concludes the paper and outlines directions for future work.

2 Background and Related Work

2.1 Background

Linked Data. Resource Description Framework (RDF) [2] is a model that can be represented as a graph, and uses Uniform Resource Identifiers (URIs), or anonymous nodes to denote resources, and literals to denote constants. Every statement in RDF can be represented as a triple. A triple is a statement of the form subject-predicate-object $\langle s, p, o \rangle$, and it is any element of $T = (U \cup B) \times (U) \times (U \cup B \cup L)$ where U, B and L are the sets of URIs, blank nodes and literals, respectively. Any finite subset of T corresponds to an RDF graph (or dataset). We divide the URIs in three disjoint sets, entities E (e.g., :Blackout), properties P (e.g., :band) and RDF classes C (e.g., Rock Music Album). In this paper, we focus on triples that contain URIs in all the positions (i.e., subject, predicate and object). Specifically, these triples contain an entity

(i.e., URI) as subject, and the URI of an RDF class or the URI of an entity as object. Therefore we consider triples in $T_U = U \times P \times (E \cup C) \subseteq T$. Finally, we denote as $D = \{D_1, ..., D_n\}$ a set of datasets, and as $T_U(D_i)$ the set of triples of a given dataset $D_i \in D$, that contain only URIs ($T_U(D_i) \subseteq T_U$).

Word2vec. It is a shallow two-layer neural network model for producing word embeddings [15]. It takes as input a text, and it produces a vector with several (usually hundreds of) dimensions for each unique word appearing in the text. The target of *word2vec* is to group the vectors of similar words closely in the vector space. In this paper, we will exploit this model for creating vectors for entities, by using the skip-gram model, which is a method that uses a specific word for predicting a target context, since "it produces more accurate results for large datasets" (`https://deeplearning4j.org/docs/latest/deeplearning4j-nlp-word2vec`). Our target is to use this model for placing similar entities (e.g., similar music albums) to a close position in the vector space.

2.2 Related work

RDF Knowledge Graph Embeddings. There have been proposed several approaches and applications for producing knowledge graphs embeddings, e.g., see two recent surveys in [32] and in [10]. Regarding approaches that exploit RDF knowledge graphs, *RDF2Vec* [29] is an approach that takes as input an RDF knowledge graph, produces URI sequences based on several strategies, such as random graph walks, and uses *word2vec* for creating vectors. They have also proposed strategies for performing biased graph walks [6], which are based on a number of metrics and statistics, such as the frequency of properties, objects, pagerank and others. They have tested these strategies for multiple tasks, such as classification and regression, by using two RDF datasets; *Wikidata* [31] and *DBpedia* [3], whereas they have used the *GloVe* model [25] for creating RDF embeddings by exploiting global patterns. Moreover, [30] have proposed a metric, called specificity, which can be used for identifying the most relevant nodes and edges in the neighborhood of an entity. This metric is exploited for creating URI sequences by performing biased random walks, and the approach was evaluated by using *DBpedia*. Furthermore, [1] used the *RDF2Vec* model for producing embeddings, and these embeddings were used for validating the facts of *DrugBank* dataset. Moreover, *THOTH* approach [22] extracts bilingual alignments from two datasets and enriches them with knowledge graph embeddings. Their target was to translate the source dataset to a target dataset, and they evaluated their approach by using the German and the English version of *DBpedia*.

[11] used several bibliographic RDF datasets and *word2vec* for enriching the data of scientific publications with information from multiple data sources, while [13] exploited enriched ontology structures for producing RDF embeddings which were used for the task of *Entity Linking*. Moreover, [23] combined embeddings from *DBpedia* and social network datasets for performing link prediction, whereas [16] exploited Wikipedia knowledge graph for finding the most similar entities to a given one for a specific time period. Furthermore, [24] created joint embeddings for words and entities for improving the task of entity search in knowledge graphs, and they tested their approach by using *DBpedia*, while the target of [8] was to cluster similar entities by using embeddings from two RDF datasets. Finally, *KGvec2go* [26] is an online service which contains already trained embeddings (by using RDF2Vec approach) from four RDF datasets. The user

can download the vectors, can find the n closest concepts to a given one, and others.

Concerning other graph-based models, such as TransH [33] and TransR [14], they use algorithms for creating entity and relation graph embeddings, i.e., the relationships between two entities are represented as translations in the embedding space. Finally, there have been proposed several methods that construct embeddings from RDF datasets by taking also into account the literals (and not only URIs), i.e., see a recent survey from [9].

Feature extraction approaches combining data from several datasets. [17] proposed a tool that can send SPARQL queries in several endpoints for creating features. However, it does not produce embeddings and it cannot collect all the data for a given entity (i.e., cross-dataset reasoning is required). Moreover, RapidMiner Semantic Web Extension [27] creates features by integrating data from a lot of datasets. However, it performs the integration task by traversing owl:sameAs paths on-the-fly (through SPARQL queries, which can be time-consuming), and not by exploiting pre-constructed indexes.

Novelty & Comparison with other approaches. To the best of our knowledge, this is the first work providing an interactive approach which can easily create URI sequences and embeddings for any set of entities. Moreover, since current approaches do not take into account the equivalences in schema and instance level, they have been mainly tested on a single or a few datasets, whereas `LODVec` produces embeddings by leveraging hundreds of datasets simultaneously, after enriching them with the results of cross-dataset identity reasoning. At this point our objective is a) to offer a simple way for creating URI sequences for multiple datasets, and b) to investigate whether the creation of even simple URI sequences and embeddings from different datasets can improve the effectiveness of several tasks (e.g., machine-learning tasks). Concerning the limitations of `LODVec`, for the time being the user decides which paths will be followed for creating the URI sequences. Therefore, we do not support automatic methods for estimating which paths are more important to be followed (e.g., through biased random walks such as [5, 30]). Moreover, we do not create sequences containing literals (e.g., see [9]), and we have not used algorithms that have been successfully applied to knowledge graphs (e.g., [14, 33]).

3 Problem Statement & Context

In this section, we introduce the problem statement (in §3.1) and the context (in §3.2).

3.1 Problem Statement

3.1.1 URI Sequences

The input is a set of selected entities $E_{sel} \subseteq E$, and the first target is to create URI sequences, i.e., $Seq_U(e)$, for each $e \in E_{sel}$. Each URI sequence, i.e., $seq \in Seq_U(e)$, corresponds to a sequence of n triples ($n \geq 1$), where each of these triples contains only URIs. A sequence containing a single triple ($n = 1$) is of the form $\langle e, p, o \rangle$, where $\langle e, p, o \rangle \in T_U$, e.g., $\langle$:Blackout,:band,:Scorpions$\rangle$. These URI sequences correspond to the direct neighborhood of an entity e, i.e., a single-edge path that starts from e. On the contrary, a sequence having n triples (i.e., a path of n-edges) is of the form

$\langle e, p_1, o_1\rangle, ..., \langle s_n, p_n, o_n\rangle$, where $e \in E_{sel}$, and for any given i ($1 \leq i \leq n$), it holds that $\langle s_i, p_i, o_i\rangle \in T_U$. Finally, for any pair of adjacent triples, i.e., say the i-th and the $(i+1)$-th triple ($i+1 \leq n$), it holds that $o_i = s_{i+1}$. Therefore, the object of the i-th triple is always the subject of the $(i+1)$-th triple. For instance, in our running example, a sequence of length $n = 3$ is the following one: $\langle$:Blackout,:band,:Scorpions$\rangle$, $\langle$:Scorpions,:city,:Hanover$\rangle$, $\langle$:Hanover,:country,:Germany$\rangle$. Our target is to construct the set $Seq_U E_{sel} = \bigcup_{e \in E_{sel}} Seq_U(e)$, where $Seq_U(e)$ corresponds to the URI sequences of each $e \in E_{sel}$.

3.1.2 Using Basic Graph Patterns for Creating the URI Sequences

For creating the desired URI sequences, we use the notion of Basic Graph Patterns, which are widely used for answering SPARQL queries [12]. In particular, a basic graph pattern is a set of triple patterns, where each triple pattern tp is similar to a triple $\langle s, p, o\rangle$. However, the subject, predicate (i.e., property), or object can be a variable. In our case, for any sequence of length n, we already know the URIs of the properties (they are given as input), however, any subject or object is a variable. In this way, we define the set BGP, where $bgp \subseteq BGP$ is a set of n triple patterns ($n \geq 1$), i.e., $bgp = tp(1), ..., tp(n)$. Each $tp(i)$ is of the form $tp(i) = \langle ?s_i, p_i, ?o_i\rangle$ and it is mandatory a) the subject of the first triple pattern, i.e., $tp(1)$, to be replaced by an entity $e \in E_{sel}$, and b) the object and the subject of two adjacent triples to be replaced by the same URI.

For example, suppose that the user wants to create a URI sequence for each music album, which contains the band of each album, the city where the band founded, and the country where that city is located in. The resulted sequence of triple patterns will be the following: $\langle ?s_1$,:band,$?o_1\rangle, \langle ?s_2$,:city,$?o_2\rangle, \langle ?s_3$,:country,$?o_3\rangle$. Since $?o_1$ should be the same as $?s_2$, and $?o_2$ the same as $?s_3$ we can write the previous sequence of triple patterns as follows: $\langle ?s_1$,:band,$?o_1\rangle, \langle ?o_1$,:city,$?o_2\rangle, \langle ?o_2$,:country,$?o_3\rangle$. Therefore, the target is to create all the sequences of triples that match each set of triple patterns for each entity $e \in E_{sel}$, i.e., for constructing the set $Seq_U(e)$.

3.1.3 From URI Sequences to URI Embeddings

The target is to use the produced set of URI sequences, i.e., $Seq_U E_{sel}$, for mapping each URI to a vector of real numbers, through a neural network model, such as *word2vec*. By using the skip-gram model of *word2vec*, we map each entity e to a vector $v(e)$. We expect that if two entities e and e' are similar, then their produced vectors, $v(e)$ and $v(e')$, will be close in the vector space, too.

3.1.4 Output Exploitation

The target is to use the produced vectors in several machine-learning based tasks, such as classification and regression, for finding the top-K similar entities to a given entity e, and others. Concerning classification and regression, one should also provide as input the corresponding categorical or continuous variable $Y(e)$ for each entity e. On the contrary, there is no need for additional input, when the desired task is to find the top-K similar entities to a given entity.

Table 1: Notations required for creating the URI sequences

ID	Notation	Formula
1	Provenance of a triple t	$prov(t) = \{D_i \in D \mid t = \langle s,p,o\rangle, t \in T_U(D_i)\}$
2	Provenance of an entity u	$dsets(u) = \{D_i \in D \mid \exists \langle u,p,o\rangle \in T_U(D_i)\}$
3	Provenance of a set of entities E' ($E' \subseteq E$)	$dsets_{E'} = \bigcup_{u \in E'} dsets(u)$
4	Coverage of a Dataset given a set of entities E' ($E' \subseteq E$)	$covD(D_i, E') = \lvert\{u \in E' \mid D_i \in dsets(u)\}\rvert$
5	All Objects (URIs) of an entity-property pair	$Objects(u,p) = \{o \mid \langle u,p,o\rangle \in T_U\}$
6	Number of Triples containing an entity u	$freq(u) = \lvert\{t \in T_U \mid t = \langle s,p,o\rangle, s = u \text{ or } o = u\}\rvert$
7	Properties of an Entity u	$Prop(u) = \{p \in P \mid \langle u,p,o\rangle \in T_U\}$
8	Properties of a set of Entities E' ($E' \subseteq E$)	$Prop_{E'} = \bigcup_{u \in E'} Prop(u)$
9	Coverage of Properties given E' ($E' \subseteq E$)	$covP(p, E') = \lvert\{u \in E' \mid \exists \langle u,p,o\rangle \in T_U\}\rvert$

3.2 Context: `LODsyndesis` Knowledge Graph

For tackling the integration difficulties (Diff 1) and (Diff 2) and for creating the desired URI sequences, we use the `LODsyndesis` knowledge graph. `LODsyndesis` [18] has pre-collected 2 billion triples and 412 million URIs from 400 RDF datasets of 9 different domains. Concerning (Diff 1), `LODsyndesis` has computed the cross-dataset identity reasoning of 44 million equivalence relationships (i.e., the transitive and symmetric closure of `owl:sameAs`, `owl:equivalentProperty`, and `owl:equivalentClass` relationships). `LODsyndesis` exploits the results of cross-dataset identity reasoning for assigning a unique identifier for each real world entity and schema element (i.e., for keeping a single representation). Finally, it collects in its index all the available triples (by preserving the provenance) for any given entity (e.g., :Blackout).

In the lower side of Fig. 1, we can see the index which is produced from `LODsyndesis` by using as input the three knowledge graphs of Fig. 1, whereas in the middle side of the same Figure, we can see the corresponding graph representation of `LODsyndesis`. In particular, this graph contains a single node for each entity (e.g., for the band "Scorpions"), and stores the provenance of each triple, i.e., see the text of each property (or edge) inside the parentheses. Regarding the index, there is a single index entry for each entity, e.g., see indicatively the entries for :Blackout, :Scorpions and :Hanover. For each entity e, the index stores information about its provenance (e.g., "Scorpions" exist in all the three datasets), and all its triples (and their provenance). Moreover, all the values for each entity-property pair are placed together, e.g., see the pair :Blackout-:lyrics. Finally, the index stores the direction of each property (or edge), i.e., a character "*" is added after a property in case it corresponds to an inverse edge, e.g., see the property :band in the index entry of :Scorpions.

The index of `LODsyndesis` is stored on disk and it is accessed through a random access file mechanism. In particular, for each entity e, a pointer is also stored (a long number), i.e., the pointer corresponds to the position of the file where the index entry of e starts. As we shall explain in §4, by using the aforementioned index, it is feasible to create URI sequences (including complementary and "mixed-path" URI sequences) for the same entity from several datasets by following paths of any length.

4 The Steps & Algorithms for Creating URI Sequences and Embeddings

Here, in §4.1 we provide some required notations, in §4.2-§4.4, we describe the functionality and all the steps of `LODVec`, and in §4.5 we provide more details about the web application of `LODVec`. Finally, the steps of `LODVec` for our running example are depicted in Fig. 2.

4.1 Notations

Table 1 represents notations that are required for the algorithm that creates the URI sequences (in §4.3.2), and metadata for aiding the user to select the desired datasets and the basic graph patterns. The first one denotes the datasets containing a triple t, while the second one indicates the provenance of an entity u, i.e., which datasets contain at least one triple, that includes u. The third one denotes the datasets that contain at least one entity $u \in E'$. The fourth formula indicates the number of entities E' that can be found in a single dataset D_i, whereas the fifth formula shows all the objects (or values) of a given entity-property pair (e.g., all the values of :Blackout-:lyrics). The sixth formula shows the frequency (popularity) of a URI in the whole graph, i.e., the number of triples containing a URI u, either as a subject or as an object. The seventh formula denotes the set of object properties (or edges) for an entity e, whereas the eighth formula corresponds to the union of all properties of entities E'. Finally, the last formula denotes the number of entities for which there is at least one triple that contains a property p.

4.2 Step A. Input & Configuration

Below, we present the sub-steps of Step A which are followed for creating the desired configuration.

Step A1. The first step is to select the desired entities E_{sel}. They can be given in three different formats a) as a list of URIs (e.g., dbp:Blackout), b) as a list of entities in plain text (e.g., "Blackout"), or c) just a URI that represents an RDF class or a category (e.g., `http://dbpedia.org/ontology/BasketballPlayer`). In the latter case, `LODVec` retrieves automatically the desired URIs, by sending a query to DBpedia SPARQL endpoint. In the running example of Fig. 2 the input is the URIs of several music albums.

Step A2. The next step is to choose the datasets D_{sel}. In particular, `LODVec` shows the list of datasets that contain triples for the input entities (i.e., the datasets $dsets_{E_{sel}}$), in descending order according to their $covD(D_i, E_{sel})$. Therefore, the first dataset in this list contains data for the maximum number of entities that belong to E_{sel} (comparing to any other dataset). The user can select to use either all the datasets $dsets_{E_{sel}}$, or any subset $D_{sel} \subseteq dsets_{E_{sel}}$. In Fig. 2 we selected to use all the available datasets (D_1,D_2,D_3) of our example (see Fig. 1).

Step A3. `LODVec` also provides an interactive way for aiding the user to select the desired basic graph patterns, i.e., BGP_{sel}. Concerning patterns of length $n = 1$, i.e., $\langle ?e, p, ?o1 \rangle$, it shows each property $p \in Prop_{E_{sel}}$ (that belongs also to datasets D_{sel}), in descending order with respect to their $covP(p, E_{sel})$. For each $p \in Prop_{E_{sel}}$, the user has three possible options. The first one is to use the property p, i.e., for creating the

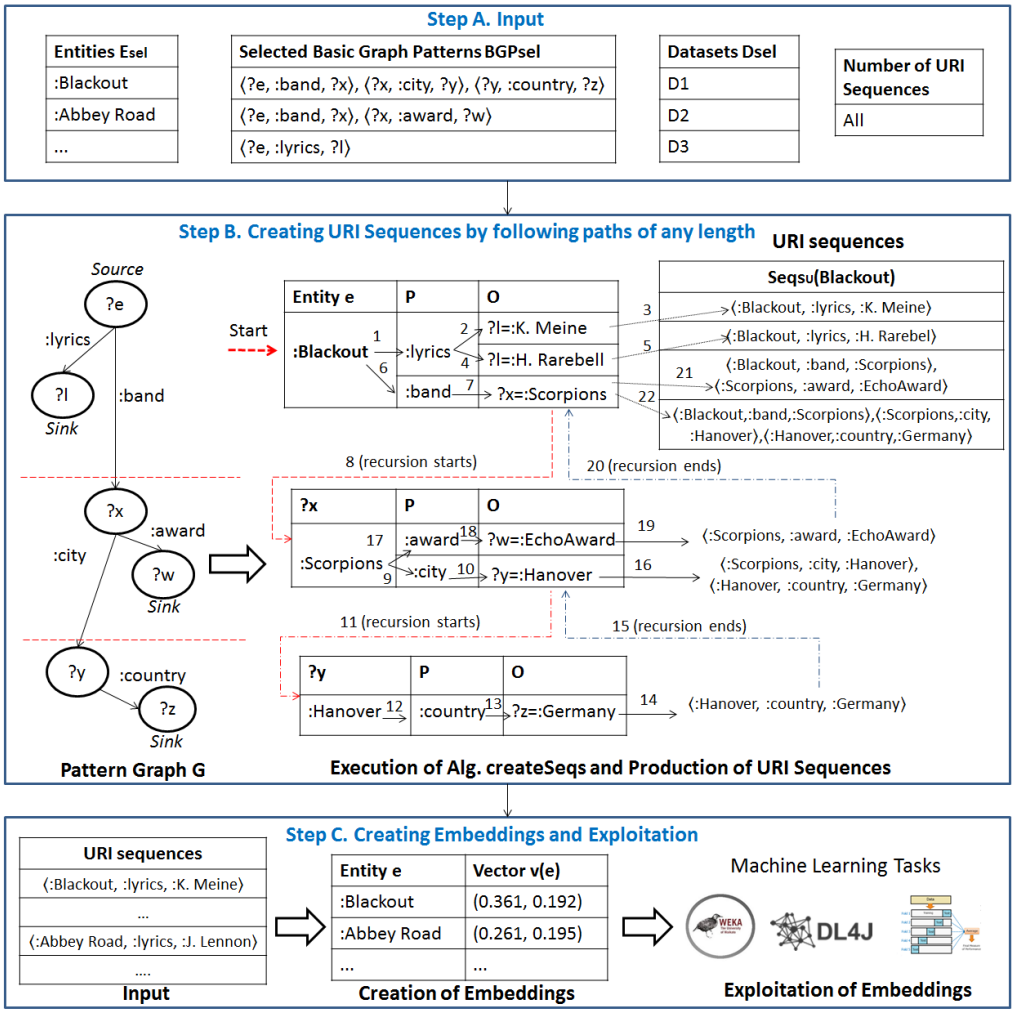


Figure 2: The steps of LODVec approach for our running example

basic graph pattern $\langle ?e, p, ?o1 \rangle$. The second option is to explore the sub-paths that pass from this property, i.e., for creating larger URI sequences, whereas the third option is to ignore the property.

By selecting the second option, `LODVec` follows larger paths for creating patterns of length $n > 1$. For instance, suppose that the user selects to explore the sub-paths of length $n = 2$, for a property, say $p1$. In this case, our target is to create patterns like $\langle ?e, p1, ?o1 \rangle$, $\langle ?o1, p2, ?o2 \rangle$. For this reason, `LODVec` selects randomly a small sample of entities, say $E_{sample} \subset E_{sel}$, and for each $e \in E_{sample}$, it finds all the objects $u \in Objects(e, p1)$. For example, if $p1$ corresponds to :band, it finds the corresponding band for each entity $e \in E_{sample}$. Afterwards for each such object u (e.g., a band), `LODVec` finds its outgoing properties, i.e., the set $Prop(u)$, and then it shows to the user the properties for these objects. The user should again select either to ignore or to use a property, say $p2$, for creating the patterns. For instance, suppose that a user selects to use a property $p2 =: guitarist$. In this case, `LODVec` will create all the URI sequences of the pattern $\langle ?e, : band, ?o1 \rangle$, $\langle ?o1, : guitarist, ?o2 \rangle$, i.e., URI sequences including the guitarist of a band of a music album. On the contrary, for a property $p2$, one has also the option to explore its sub-paths, i.e., for creating URI sequences for $n > 2$, and so forth.

In Fig. 2 (see the corresponding table in the upper middle side), we selected to create URI sequences based on three basic graph patterns. In particular, one basic graph pattern of length $n = 3$, i.e., "the band of each album, the city where the band founded, and the country where that city is located in", one pattern of length $n = 2$, i.e., "the band of each album and the awards won by that band", and finally a pattern of length $n = 1$, i.e., "the persons that have written the lyrics of each album".

Step A4. The fourth step is to select the number of URI sequences that will be created, i.e., all the possible URIs sequences according to the given configuration or the top-K ones (see more details in §4.3.4). In Fig. 2, we selected to create all the possible URI sequences (by using the selected datasets and basic graph patterns).

Step A5. In this step, `LODVec` shows to the user the configuration of Steps A1-A4, and the user can click on a button for creating the desired URI sequences (by using the algorithms that are presented in §4.3).

4.3 Step B. Creating URI Sequences of any length

This step consists of two sub-steps, a) the conversion of the selected Basic Graph Patterns (given as input from the user) to a *Pattern Graph* G (i.e., see §4.3.1), and b) the creation of URI sequences, by using the desired configuration and the *Pattern Graph* G (i.e., see §4.3.2).

4.3.1 Creation of Pattern Graph G

Rationale. Two or more basic graph patterns can start with the same triple patterns, e.g., in our running example of Fig. 2, the first two selected basic graph patterns start with the triple pattern ⟨?e, :band, ?x⟩. For avoiding to follow the same path multiple times for creating the URI sequences (since it can be time-consuming), we create and use a pattern graph G, where $G \subseteq BGP$.

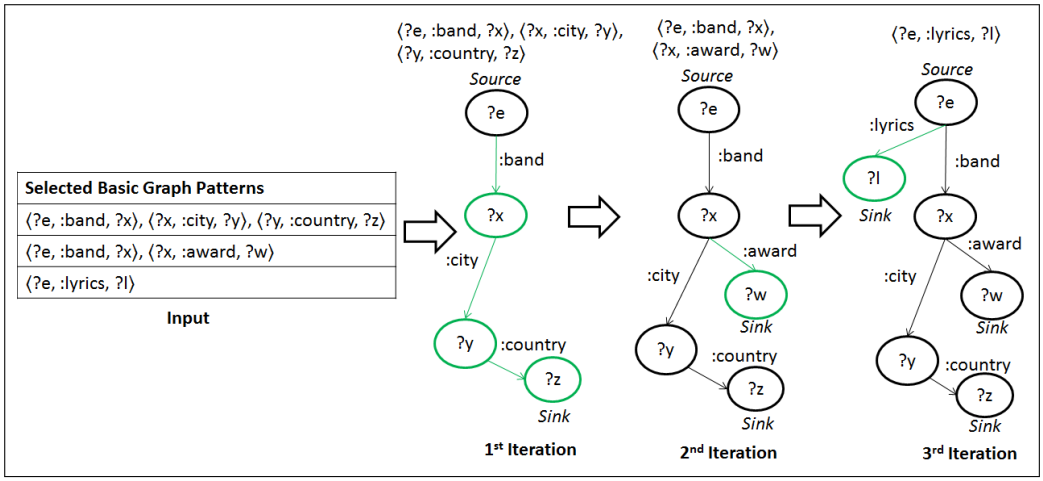


Figure 3: Execution of Alg. 1 for the basic graph patterns of our example

Input. The input is the set of Basic Graph Patterns which are selected by the user, i.e., BGP_{sel}.

Algorithm. Alg. 1 shows how to exploit the selected Basic Graph Patterns, i.e., BGP_{sel}, for creating the corresponding *Pattern Graph* $G = \{V, Ed\}$, where V are its nodes (or vertices) and Ed denote its edges. The produced pattern graph of our example is shown in the right side of Fig. 3 (and in the middle left side of Fig. 2).

This graph corresponds to a directed acyclic graph, i.e., it has not directed cycles. Graph G has a single source node, i.e., a node without incoming edges. As we shall see, the source node will be strictly replaced by an entity $e \in E_{sel}$. Any remaining node has exactly one incoming edge and corresponds to a variable (e.g., $?x$). Furthermore, it has m sink nodes (i.e., nodes without outgoing edges), where m equals the number of basic graph patterns that are given as input, and each sink node corresponds to the last variable of each basic graph pattern (e.g., to the variables $?z$, $?w$ and $?l$ of the basic graph patterns of Fig. 3). Finally, an edge corresponds to a property that connects two variables (or URIs).

For the graph G, we use a variable for storing all its nodes, and a variable for storing the unique source node (see line 1 of Alg. 1). For each node of G (see line 2), we use a boolean variable, i.e., $isSink$, which becomes true for the sink nodes. Moreover, we use a map called $edges$ for each node v, i.e., it is a binary relationship: $edges : P \rightarrow V$, that has in its left side a property $p \in P$ (e.g., the property :band), and in its right side the node $v \in V$ (i.e., a URI), where that edge ends up.

Concerning the function $createPatternGraph$, it initializes the graph G, and a new node $srcNode$, which corresponds to the single source node of G (see lines 5-7). Afterwards, Alg. 1 reads each basic graph pattern bgp and sets as the current node, i.e., $cNode$, the source node (see lines 8-9). Then, it reads all the triple patterns of bgp, i.e., the function $bgp.tpSize$ denotes the number of triple patterns of each bgp (line 10).

For each triple pattern of bgp, Alg. 1 stores its property in a variable p (line 11). Afterwards, it checks if the property p exists in the left side of the $edges$ of the current node (i.e., in Alg. 1 $Left$ denotes the left side of a binary relationship). In case it is false,

Algorithm 1: Creating the Pattern Graph G by using the selected basic graph patterns BGP_{sel}

Input: Basic Graph Patterns BGP_{sel}
Output: PatternGraph G

```
1  Graph {nodes ← ∅, sourceNode ← nil }
2  Node {edges ← ∅, isSink ← false }
3
4  function createPatternGraph(BGP_sel)
5      Graph G ← new Graph()
6      Node srcNode ← new Node()
7      G.sourceNode ← srcNode
8      forall bgp ∈ BGP do
9          cNode ← G.sourceNode
10         foreach i ← 1 to bgp.tpSize do
11             p ← bgp.tp(i).p
12             if p ∉ Left(cNode.edges) then
13                 Node tmpNode ← new Node()
14                 if i ≡ bgp.tpSize then
15                     tmpNode.isSink ← true
16                 G.nodes ← G.nodes ∪ {tmpNode}
17                 cNode.edges ← cNode.edges ∪ {(p, tmpNode)}
18                 cNode ← tmpNode
19             else
20                 cNode ← cNode.edges(p)
21     Return G
```

it creates a new node, i.e., *tmpNode*. For that node, it sets the value of *isSink* variable to "true", when it corresponds to the last variable of a *bgp* (see lines 13-16). Finally, it adds to the *edges* of the current node, a key-value pair, where the key is the property p and the value is the *tmpNode*, it assigns as the current node the *tmpNode* (see lines 17-18), and it continues with the next triple pattern. On the contrary, in case we have already created an outgoing edge for property p, Alg. 1 sets as the current node, the node where this edge ends up (see lines 19-20), before continuing with the next triple pattern. The output is the produced graph G (see line 21).

Example. Fig. 3 shows the graph G of our running example after each iteration of lines 8-20 (see Alg. 1). By using the green color, we represent the edges and nodes that are created in each iteration. At first, we read the triple patterns of the first *bgp*, starting with ⟨?e, :band, ?x⟩. We create the edge :band and a node for variable ?x, which becomes the current node. For the second triple pattern, we create the edge :city, which connects the current node ?x with the new node ?y, whereas for the last triple pattern, the edge :country is created, which connects the current node ?y and the new sink node ?z. For the second basic graph pattern, we read its first triple pattern, i.e., ⟨?e, :band, ?x⟩, however, we have already created an edge that connects these two variables. Therefore, we set ?x as the current node (i.e., we use lines 19-20). Regarding its second (and last) triple pattern, we create the edge :award, that connects ?x with a new sink node ?w. Finally, we read the third *bgp*, where a single edge is created between the source node and a newly created (sink) node $?l$.

Time and Space Complexity. The time complexity of Alg. 1 is $\mathcal{O}(|BGP_{sel}|*|tpSize_{avg}|)$, where $tpSize_{avg}$ is the average number of triple patterns for all the basic graph patterns. The space complexity is $\mathcal{O}(|V|+|Ed|)$, since we keep in memory all the nodes and edges of *Pattern Graph G*.

4.3.2 Algorithm for Creating URI sequences

Rationale. Alg. 2 creates all the possible URI sequences according to a given configuration, by exploiting the index of `LODsyndesis`. Since that index stores all the triples (and their provenance) for a given entity from hundreds of datasets, we can easily create both complementary and "mixed-path" URI sequences.

Input. Alg. 2 uses as input the selected entities E_{sel}, the pattern graph G and the selected datasets, i.e., D_{sel}.

Algorithm. Alg. 2 traverses the pattern graph G in a depth first search way, for creating the URI sequences for each entity $e \in E_{sel}$ separately, through the recursive function *createSeqs*. This function takes as input a URI u, and the corresponding node in the graph G, and produces all the URI sequences starting with u, with respect to the pattern graph G.

Each time we select an entity $e \in E_{sel}$, we set the source node of G as the current node, and we call the function *createSeqs*, for creating the desired URI sequences for e (lines 1-3). Concerning the depth-first traversal that is followed, we start from the source node of G, and we continue forward by exploring the nodes of a specific path, until we reach a sink node. In the latter case, we move backwards to the same path, for finding other nodes to traverse (i.e., by backtracking). Moreover, we assume that the left edges are always chosen before the right edges, e.g., for the pattern graph of Fig. 2, we will visit the nodes in the following order: $?e, ?l, ?x, ?y, ?z, ?w$.

Algorithm 2: Creating *URI sequences* of any length for a set of entities E_{sel}

Input: Entities E_{sel}, PatternGraph G, Datasets D_{sel}
Output: URI Sequences $Seq_U E_{sel}$ for all the entities E_{sel}

```
1  forall e ∈ E_sel do                                   // For each desired entity e
2      Node cNode ← G.sourceNode                         // Set the source node of G as current node
3      Seq_U(e) ← createSeqs(e, cNode)                   // Create URI sequences starting with entity e
4      Seq_U E_sel ← Seq_U E_sel ∪ Seq_U(e)              // Add Seq_U(e) to the set of all sequences Seq_U E_sel
5  Return Seq_U E_sel
6
7  function createSeqs(URI u, Node cNode)               // Create URI sequences starting with URI u
8      Seq_U(u) ← ∅
9      forall p ∈ Left(cNode.edges), p ∈ Prop(u) do      // For each property (or edge) of cNode
10         forall o ∈ Objects(u, p) do                   // For all the objects of the pair u, p
11             if prov(⟨u, p, o⟩) ∩ D_sel ≠ ∅ then       // Check the provenance of the triple ⟨u, p, o⟩
12                 nextNode ← cNode.edges(p)             // The next node is the node where p ends up
13                 if nextNode.isSink ≡ true then        // If next node is sink, add ⟨u, p, o⟩ to Seq_U(u)
14                     Seq_U(u) ← Seq_U(u) ∪ {⟨u, p, o⟩}
15                 else
16                     Seq_U(o) ← createSeqs(o, nextNode)   // Visit sub-paths starting with o, recursively
17                     forall seq ∈ Seq_U(o) do          // For each sub-path that starts with URI o
18                         Seq_U(u) ← Seq_U(u) ∪ {⟨u, p, o⟩+',' +seq}   // Concatenate sub-path with ⟨u, p, o⟩
19     Return Seq_U(u)
```

Regarding the steps of *createSeqs*, it iterates over the direct properties (or edges) of the current node of graph G (which corresponds to the input URI u). For each edge (or property) p of the current node it reads the index entry of u (i.e., by using the index of `LODsyndesis`), for retrieving all the objects of the entity-property pair (u, p). Afterwards, for each $o \in Objects(u, p)$, it checks the provenance of the triple $\langle u, p, o \rangle$. Specifically, the lines 12-18 are executed only if the triple $\langle u, p, o \rangle$ occurs at least in one of the selected datasets D_{sel}. In case it holds, it assigns the node where the property p ends up as the next node (line 12), i.e., the node which corresponds to the object o. Afterwards, it checks if the next node corresponds to a sink node. In case it is true, it adds to the sequences of u the triple $\langle u, p, o \rangle$ (lines 13-14).

On the contrary, if the current node is not a sink node, it calls the function *createSeqs* by giving as input the object o, and the *nextNode*. In this way, it visits in a depth-first search way the URI o (lines 15-16), for creating all the sub-paths that start with the URI o. After the end of each recursion, it concatenates all the produced sub-paths that start with the URI o with the triple $\langle u, p, o \rangle$ (lines 17-18). Finally, the produced URI sequences are returned as output (lines 19).

The same process is followed for each entity $e \in E_{sel}$, and the produced URI sequences are added to $Seq_U E_{sel}$ (line 4), i.e., the set of URI sequences of all the entities E_{sel}. The set $Seq_U E_{sel}$, i.e., the output of the whole process (line 5), will be used as input for creating the embeddings for the entities E_{sel}.

Example. In the middle part of Fig. 2, we show the order (see the numbers near to each arrow) that is followed for creating the URI sequences for the entity $?e$ =:Blackout, according to the graph G (see the graph in the middle left side of Fig. 2). A red arrow indicates that a recursion starts (i.e., a new index entry is accessed), and a blue arrow indicates the end of a recursion. For the entity :Blackout, Alg. 2 calls the function *createSeqs*(:Blackout,$G.sourceNode$), i.e., :Blackout replaces the source node of G. Moreover, in this recursion, the algorithm will read the index entry of the input entity, i.e., :Blackout (see the arrow Start).

By traversing the pattern graph G in a depth first search way, it visits the node $?l$ through the property p =:lyrics. Therefore, it retrieves (from the index entry of :Blackout) all the objects of the entity-property pair :Blackout-:lyrics, and it checks the provenance of each triple. Since we have selected to use all the datasets, it creates the first two URI sequences (see arrows 1-5 in Fig. 2), which correspond to all the results of the first basic graph pattern $\langle ?e$, lyrics, $?l \rangle$. Then, it visits the node $?x$ =:Scorpions through the property p =":band" (see arrows 6-7). However, since $?x$ is not a sink node in graph G, we call the function *createSeqs* for the URI u =:Scorpions (see the red dotted arrow 8 in Fig. 2). In this way, we will create all the sub-paths that start with the URI u, i.e., we read the index entry of the URI :Scorpions. In particular, we iterate over the values of the pair :Scorpions-:city for finding the values of $?y$ variable, i.e., $?y$ =:Hannover. Since $?y$ does not correspond to a sink node, we move forward (in a depth-first search way), i.e., we call the function *createSeqs* for u =:Hannover.

Therefore, a new recursion starts (see arrows 9-11), where we visit the next node $?z$, which is replaced by the URI :Germany (see arrows 12-14). Since $?z$ is a sink node, the recursion ends (see the blue dotted arrow 15), and we return the URI sequence of this sub-path, i.e., ⟨:Hanover,:country,:Germany⟩. Afterwards, we return to the previous node, i.e., $?x$=:Scorpions, where we concatenate the aforementioned URI sequence with the triple ⟨:Scorpions,:city,:Hanover⟩ (arrow 16). Then, we visit the node $?w$ through

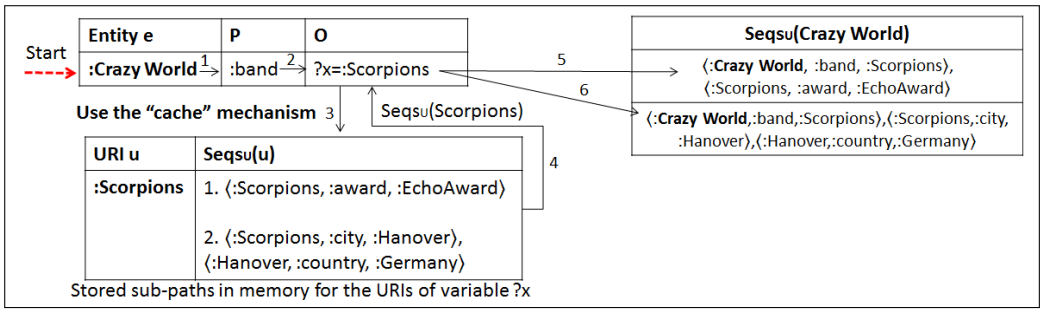


Figure 4: Storing already explored sub-paths in memory

the property p =:award, and we add to the URI sequences of :Scorpions the triple ⟨:Scorpions,:award,:EchoAward⟩ (arrows 17-19).

Finally, we return back to the recursive call of the desired entity (arrow 20), i.e., :Blackout, where we concatenate the two produced URI sequences that start with the URI :Scorpions, with the sequence ⟨:Blackout,:band,:Scorpions⟩. In this way, we create the last two URI sequences for :Blackout (arrows 21-22). These two sequences correspond to "mixed-path" URI sequences, since they contain triples derived from at least two datasets (see the provenance of triples in Fig. 1).

Time and Space Complexity. The time complexity of Alg. 2 is $\mathcal{O}(|E_{sel}|*(|V|+|Ed|))$, i.e., for each entity $e \in E_{sel}$, we visit all the nodes and edges of the pattern graph G. The space complexity is $\mathcal{O}(|Seq_U E_{sel}| + |V| + |Ed|)$, since we keep in memory all the produced URI sequences, and the pattern graph G.

4.3.3 Avoiding to visit the same sub-paths multiple times

Limitation. A major limitation of Alg. 2 is that we can follow the same "sub-paths" multiple times for creating the desired URI sequences for two or more given entities. In this way, the number of index entries that we read and the number of recursions will be increased. For instance, suppose that we want to create URI sequences for two (or more) music albums from the same band, e.g., say the music albums "Blackout" and "Crazy World" of the band "Scorpions". In this case, for the first or/and the second basic graph pattern (see Fig. 2), we will read the index entries of "Scorpions" and "Hanover" twice (i.e., we will need three recursions for each album), whereas the corresponding URI sequences for these two albums will differ only on the subject of the first triple.

Cache Mechanism - Storing sub-paths in memory. For avoiding to visit one or more "sub-paths" multiple times, for each node v of the pattern graph G, we propose to store the URIs which have replaced this node (variable) previously, and also all the URI sequences that have been produced for each URI. In particular, we can use a function for each node v of G, $rep_U : U \rightarrow Seq_U$, where for a $u \in U$, $v.rep_U(u) = Seq_U(u)$. Therefore, it is a map having as a key a URI u and as a value the sub-paths starting with u. For example, for the variable (or node) $?x$, $?x.rep_U(: Scorpions)$ contains as value the following two sequences starting with the URI :Scorpions, i) ⟨:Scorpions,:city, :Hanover⟩, ⟨:Hanover,:country,:Germany⟩ and ii) ⟨:Scorpions,:award,:EchoAward⟩, whereas for the node $?y$, $?y.rep_U(: Hanover)$ contains

Algorithm 3: Replace lines 13-18 of Alg. 2 for enabling cache mechanism, i.e., for storing already explored sub-paths in memory

```
1  if nextNode.isSink ≡ true then
2      Seq_U(u) ← Seq_U(u) ∪ {⟨u, p, o⟩}
3  else
4      if o ∈ Left(nextNode.rep_U) then
           // Retrieve sub-paths from cache
5          Seq_U(o) ← nextNode.rep_U(o)
6      else
7          Seq_U(o) ← createSeqs(o, nextNode)
           // Store sub-paths to cache
8          nextNode.rep_U ← nextNode.rep_U ∪ {(o, Seq_U(o))}
9      forall seq ∈ Seq_U(o) do
10         Seq_U(u) ← Seq_U(u) ∪ {⟨u, p, o⟩+',' +seq}
```

a single URI sequence, i.e., ⟨:Hanover,:country,:Germany⟩.

Algorithm. For enabling the "cache" mechanism, we can replace the "if-else" statement of lines 13-18 of Alg. 2, with the lines 1-10 of Alg. 3. In particular, Alg. 3 keeps in memory the produced URI sequences for the sub-paths that start with o (see line 8), and each time it checks if the next node has been already replaced by the URI o. In such a case, it retrieves all the URI sequences for the already explored "sub-paths" through the map rep_U (and not by calling the function *createSeqs* in a recursive way), i.e., see lines 4-5 in Alg. 3.

For instance, in Fig. 4, suppose that we have created the URI sequences for the album :Blackout, therefore, we have stored in memory all the sub-paths that start with the entity :Scorpions (see the table in the lower left side of Fig. 4). Therefore, for the second album, i.e., "Crazy World" of :Scorpions, we can retrieve from the map rep_U, all the sub-paths starting with the URI :Scorpions. In this way, in Fig. 4 we need only a single recursion for creating the URI sequences for the album :Crazy World (instead of three recursions that we needed for the :Blackout), since we do not need to read the index entries of :Scorpions and :Hanover. In §5, we provide experiments showing the efficiency of this "cache" mechanism.

4.3.4 Additional Functionality of `LODVec`

`LODVec` can produce K URI sequences, instead of all the possible ones according to the given configuration (i.e., according to the selected basic graph patterns and datasets). In particular, `LODVec` enables the production of K random URI sequences for each entity e. Moreover, it can produce the top-K URI sequences with respect to the frequency $freq(u)$ of the last URI of each sequence, either in ascending or in descending order, e.g., the descending order means that `LODVec` will create the K URI sequences whose last URI is a very "popular" URI. For retrieving the frequency $freq(u)$ of a URI u, we send a request to the REST API of `LODsyndesis` [19].

4.4 Step C. Creation & Exploitation of Embeddings

The final step (see Step C of Fig. 2) is to exploit the produced set of URI sequences $Seq_U E_{sel}$ (see the table in the lower left side of Fig. 2) for creating one vector for each entity e (see the table in the lower middle side of Fig. 2). For this reason, we use indicatively the *word2vec* implementation of *dl4j* library, which produces as output a single vector $v(e)$ for each $e \in E_{sel}$. The produced vectors can be exploited for several tasks, such as (i) machine-learning tasks, and (ii) similarity tasks.

Regarding task (i), the user should also provide as input, in a ".tsv" (tab separated values) file, the corresponding categorical or continuous variable $Y(e)$, for each entity e. The vectors which are produced through this process, can be exported in ".arff" format for performing either classification or regression through *WEKA* API. Moreover, the produced ".arff" file can be uploaded to `LODVec`, for retrieving immediately the classification or regression predictions (by exploiting *WEKA* API).

Concerning task (ii), the vectors can be downloaded in ".txt" format, which is directly accessible from *dl4j* API. The aforementioned API is also used from `LODVec` for producing the top-K related entities to a given entity.

4.5 Application and Demo

The online application of `LODVec` is accessible at `https://demos.isl.ics.forth.gr/LODVec/`, whereas a demo video which presents an example for a set of music albums is available at `https://youtu.be/w8Cnuz4knxE`. In particular, this video shows how to create the URI sequences and the embeddings, and how to use them for a classification task, and for finding the top-10 similar music albums to a given album.

5 Experimental Evaluation

Here, we evaluate the impact of using multiple datasets for creating URI embeddings in several machine-learning tasks, whereas we evaluate the efficiency of the proposed approach. First, in §5.1, we present the evaluation datasets, whereas in §5.2, we provide details about the hardware and the parameters of *word2vec*. In §5.3, we discuss the efficiency of `LODVec`, whereas in §5.4 we introduce experiments for classification and regression tasks. Moreover, in §5.5, we show experimental results for the task of finding similar entities. Finally, all the evaluation datasets and the experimental results can be downloaded from `http://islcatalog.ics.forth.gr/dataset/lodvec`.

5.1 Evaluation Datasets

Table 2 introduces for each evaluation dataset the number of entities (URIs) that it contains, the configurations that we have selected for creating the URI embeddings, and the tasks where we use each evaluation dataset. Specifically, we perform experiments that concern the efficiency (EFF) of `LODVec`, whereas we also evaluate our approach for that tasks of Classification (CF) and regression (REG) and for finding similar entities (SIM). Below, we provide more details for each evaluation dataset.

Table 2: Configuration (Basic Graph Patterns) that we selected for each evaluation dataset

Evaluation Dataset	URIs	Config. ID	Basic Graph Patterns of this Configuration	Used for Tasks
Movies	2,000	*Conf. I*	All Single Edge Paths	EFF, CF, REG, SIM
Movies	2,000	*Conf. II*	*Conf. I* and $\langle e, actor, ?o_1\rangle, \langle ?o_1, award, ?o_2\rangle$ and $\langle e, actor, ?o_1\rangle, \langle ?o_1, type, ?o_3\rangle$ and $\langle e, actor, ?o_1\rangle, \langle ?o_1, subject, ?o_4\rangle$	EFF, CF, REG
Movies	2,000	*Conf. III*	*Conf. II* and $\langle e, actor, ?o_1\rangle, \langle ?o_1, hometown, ?o_5\rangle, \langle ?o_5, country, ?o_6\rangle$	EFF
Music Albums	1,600	*Conf. IV*	All Single Edge Paths	EFF, CF, REG
Music Albums	1,600	*Conf. V*	*Conf. IV* and $\langle e, artist, ?o_1\rangle, \langle ?o_1, p, ?o_2\rangle$	EFF, CF, REG
Music Albums	1,600	*Conf. VI*	*Conf. V* and $\langle e, artist, ?o_1\rangle$, $\langle ?o_1, recordLabel, ?o_3\rangle$, $\langle ?o_3, country, ?o_4\rangle$	EFF
Top NBA Players	450	*Conf. VII*	All Single Edge Paths	EFF, CF, REG
Basketball Players	12,250	*Conf. VIII*	All Single Edge Paths	EFF, SIM

5.1.1 Movies Dataset

We use the Metacritic Movies evaluation dataset, derived from [28]. It contains the DBpedia URIs of 2,000 movies, and an average rating of all time reviews for each movie. Concerning the ratings, 1,000 of them have high rating (> 60), and the remaining 1,000 ones have low rating (< 40). We use three configurations (i.e., *Conf. I* to *Conf. III* in Table 2). In particular, we create URI embeddings by following all the possible single-edge paths (i.e., *Conf. I*). Moreover, we combine *Conf. I* by creating also larger URI sequences that include "for each actor of a movie, its rdf:type, categories and awards" (i.e., *Conf. II*), and finally, we enrich *Conf. II* with URI sequences that are created by following three-edges paths, i.e., they also include "for each actor of a movie, its hometown, and the country where that town is located in" (see *Conf. III*).

All these configurations are used for evaluating the efficiency of `LODVec` (in §5.3), whereas the embeddings of *Conf. I* and *Conf. II* are also used in §5.4.1, for a binary classification (i.e., to predict if a movie has a high or a low rating) and for a regression task (i.e.,, to predict the exact rating of each movie). Finally, the produced embeddings of *Conf. I* are also used for finding the most similar movies for a given movie §5.5.

5.1.2 Music Albums Dataset

We use the Metacritic Music Albums dataset [28], which contains the DBpedia URIs of 1,600 music albums, and for each album its average user rating. Regarding the ratings, 800 of them have high rating (> 79), and the other 800 ones have low rating (< 63). For these albums, we create URI embeddings for three configurations (*Conf. IV* to *Conf. VI* in Table 2). Regarding *Conf. IV*, we create all the possible single-edge URI sequences. Concerning *Conf. V*, we use the URI sequences of *Conf. IV*, whereas we also create URI sequences by following two edges paths, i.e., "the artist of each album and all the triples for each artist". Finally, in *Conf. VI*, we use all the URI sequences of the previous configuration, and we also create sequences including paths of three edges, i.e., "the artist of each music album, the record label of each artist and the country where that record label is located in".

First, we evaluate the efficiency by using these three configurations (in §5.3). Moreover, in §5.4.2, we evaluate the produced embeddings of *Conf. IV* and *Conf. V* in a

binary classification task, i.e., for predicting if a music album has high or low rating, and in a regression task, i.e., for predicting the exact rating of each album.

5.1.3 Basketball Datasets

We use the list of 500 Greatest NBA Players of All Time (until 2011) according to SLAM magazine (`https://www.basketball-reference.com/awards/slam_500_greatest.html`). We have divided the aforementioned players into the following two groups: the players in positions 1-225, and the players in positions 276-500. Therefore, we use 450 players (out of 500), i.e., see *Conf. VII* in Table 2. For each player, we have collected its DBpedia URI and its corresponding position in the list of the top NBA players. For these players, we create URI sequences and embeddings by using only their direct neighborhood, i.e., we follow all the single-edge paths. The resulted embeddings are used for classification and regression (in §5.4.3). The target of the binary classification is to predict if a player in that list belongs in a position in the range [1,225] or in the range [276,500]. On the other hand, the target of the regression task is to predict the exact position of each NBA player in the list of the top-500 players of all time.

Moreover, we have also downloaded the DBpedia URIs of 12,250 basketball players (all the URIs of the class `http://dbpedia.org/ontology/BasketballPlayer`), i.e., (see *Conf. VIII* in Table 2). Again, we create all the URI sequences of single edge paths for each player, and the produced embeddings are used for evaluating the effectiveness of finding the most similar basketball players to a given player (in §5.5).

5.2 Setup

Here, we introduce the hardware setup and the parameters that we used for creating the embeddings through *word2vec*.

5.2.1 Hardware Setup

All the experiments were performed on a single machine with an *i5 core*, *8GB RAM*, and *1 TB disc space*. Moreover, the indexes and services of `LODsyndesis`, which are used in our approach, are hosted in a single machine of *okeanos* cloud computing service (`https://okeanos.grnet.gr/`) with *8 cores*, *8GB RAM*, and *60 GB disc space*.

5.2.2 Word2vec Parameters

For building our *word2vec* model, we use the skip-gram model of *dl4j* library, we exclude URIs existing < 5 times in the produced sequences ($minWordFrequency = 5$), and we use 10 iterations. We selected the window size parameter to be 2 ($windowSize = 2$) for the configurations including URI sequences with only single-edge paths (*Conf. I*, *IV*, *VII* and *VIII*), and 4 ($windowSize = 4$) for the remaining configurations, i.e., they also contain URI sequences produced by following larger paths. For each entity e, we produce a single vector $v(e)$ with 100 dimensions ($layerSize = 100$). We expect that similar entities (e.g., albums with similar rating) will be placed in a close position in the vector space.

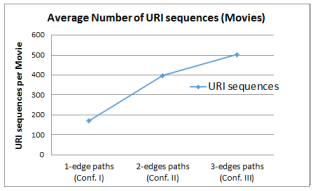


Figure 5: Average Number of URI sequences per movie for each configuration

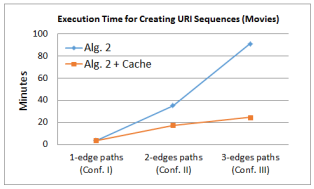


Figure 6: Total Execution Time for Movies dataset for each configuration

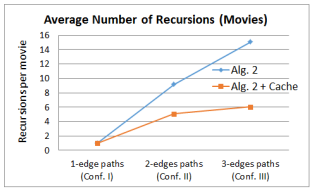


Figure 7: Average Number of Alg. 2 recursions per movie for each configuration

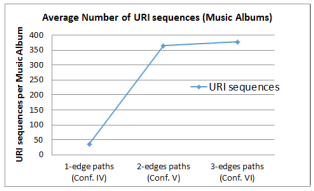


Figure 8: Average Number of URI sequences per album for each configuration

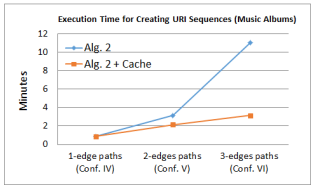


Figure 9: Total Execution Time for Music Albums dataset for each configuration

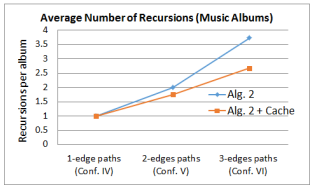


Figure 10: Average Number of Alg. 2 recursions per music album for each configuration

Table 3: Execution Time for Different Configurations

Evaluation Dataset	Configuration	$\|Seq_U(e)\|$ Average	Pattern Graph Creation Time	URI sequences Creation Time	Embeddings Creation Time	Total Time
Movies	*Conf. I* (1-edge paths)	170.1	0.001 min	3.5 min	1.7 min	5.2 min
Movies	*Conf. II* (2-edges paths)	397.9	0.002 min	17.4 min	4.0 min	21.4 min
Movies	*Conf. III* (3-edges paths)	501.8	0.002 min	24.5 min	6.8 min	31.3 min
Music albums	*Conf. IV* (1-edge paths)	35.9	0.001 min	0.9 min	0.3 min	1.2 min
Music albums	*Conf. V* (2-edges paths)	365.0	0.002 min	2.2 min	3.3 min	5.5 min
Music albums	*Conf. VI* (3-edges paths)	377.8	0.003 min	3.1 min	3.6 min	6.7 min
Top NBA players	*Conf. VII* (1-edge paths)	163.2	0.001 min	0.4 min	0.3 min	0.7 min
Basket players	*Conf. VIII* (1-edge paths)	54.8	0.001 min	6.1 min	4.1 min	10.2 min

5.3 Efficiency of Creating URI Sequences and Embeddings

For the efficiency experiments, we create URI sequences by using all the RDF datasets of `LODsyndesis`, for all the configurations of Table 2. The objective is to evaluate the gain of using the "cache" mechanism (see §4.3.3), i.e., to evaluate the decrease a) in the execution time, and b) in the number of index entries that we need to read (number of recursions). Moreover, we provide measurements about the efficiency of the whole process.

Movies. Fig. 5 shows the number of URI sequences that were produced for each configuration, i.e., by following larger paths the number of sequences increases (see also the third column of Table 3), whereas as we can see in Fig. 6, the execution time increases, too. However, by using the "cache" mechanism (see Fig. 6), we achieved 2.02× speedup for *Conf. II*, i.e., URI sequences including also two-edges paths, and 3.7× speedup for *Conf. III*, i.e., by following also three-edges paths. Indicatively, for creating URI sequences for both the movie and its actors (i.e., *Conf. II*), we needed 17.4 minutes. Concerning the number of recursions (see Fig. 7), by using the "cache" mechanism, we needed on average 5.1 recursions per movie (instead of 9.18) for *Conf. II*, and 6 recursions per movie (instead of 15) for *Conf. III*. Regarding the total execution time of these configurations, we can see in Table 3 (see the rows for movies) that the creation time of Pattern Graph G (see the fourth column of Table 3) is very fast in all cases, whereas the execution time of creating the embeddings (see the sixth column of Table 3) increases as the number of URI sequences grows.

Music Albums. Figures 8, 9 and 10 show the number of URI sequences, the execution time and the number of recursions for each configuration (i.e., *Conf. IV* to *Conf. VI*). Similarly to movies dataset, the execution time (see Fig. 9) and the number of recursions (see Fig. 10) decreased by using the "cache" mechanism, i.e., for *Conf. V* we needed on average 1.75 recursions per album (instead of 2) and we achieved 1.46× speedup, whereas for *Conf. VI*, we performed 2.68 recursions (instead of 3.74) and we achieved 3.48× speedup. Indicatively, we needed 2.16 minutes for creating URI sequences for each music album and its artist. Finally, we can observe in Table 3 (see the rows for music albums), that the time for creating the embeddings is increased as the number of URI sequences grows.

Basketball Datasets. In the last two rows of Table 3, we can see the execution time for the two datasets for basketball. For the first case, where the input is 450 entities (see the row for top NBA players in Table 3), we managed to complete the whole process very fast, i.e., in less than 1 minute. On the contrary, in the second case (see the last row of Table 3)), we needed over 10 minutes, since the input is much larger, i.e., 12,250 entities.

More Efficiency Experiments. In §5.4, we provide more efficiency experiments for each evaluation dataset, by using different subsets of RDF datasets, e.g., by using a single or pairs of RDF datasets, instead of all the available RDF datasets of `LODsyndesis`.

5.4 Task A. Classification and Regression

In this subsection, we introduce the results for the tasks of (binary) classification and regression.

Machine Learning Models & Metrics. For performing classification and regres-

Table 4: Classification and regression experiments on Movies dataset

Row ID	RDF Datasets Used	Config. Name	$\|Seq_U(e)\|$ Average	Total Time	RF (CF)	SVM (CF)	RF (REG)	LR (REG)
1	DBpedia (DB)	*Conf. I* (1-edge paths)	23.8	2.7 min	71.00%	71.14%	20.66	20.02
2	Freebase (FR)	*Conf. I* (1-edge paths)	112.0	3.6 min	81.78%	82.02%	16.47	16.37
3	Wikidata (WK)	*Conf. I* (1-edge paths)	22.5	2.5 min	71.80%	70.28%	20.19	20.81
4	DB,FR	*Conf. I* (1-edge paths)	38.3	3.7 min	82.41%	82.51%	16.45	16.11
5	DB,WK	*Conf. I* (1-edge paths)	132.0	2.8 min	74.90%	74.92%	16.76	19.18
6	FR,WK	*Conf. I* (1-edge paths)	129.0	3.8 min	82.48%	83.32%	16.53	16.25
7	DB,FR,WK	*Conf. I* (1-edge paths)	144.7	4.2 min	82.54%	84.10%	16.45	16.01
8	All 14 datasets	*Conf. I* (1-edge paths)	170.1	5.2 min	**82.61%**	**84.70%**	**16.40**	**15.57**
9	All 14 datasets	*Conf. I* (top-30 desc)	30.0	121.1 min	75.10%	75.86%	19.23	19.07
10	All 14 datasets	*Conf. I* (top-30 asc)	30.0	121.1 min	72.70%	72.50%	19.73	19.58
11	All 14 datasets	*Conf. I* (top-30 rand)	30.0	5.0 min	71.93%	73.10%	20.44	19.86
12	DBpedia (DB)	*Conf. II* (2-edges paths)	106.9	17.6 min	68.27%	70.92%	21.00	20.46
13	Freebase (FR)	*Conf. II* (2-edges paths)	152.9	17.9 min	77.00%	78.74%	18.70	17.20
14	Wikidata (WK)	*Conf. II* (2-edges paths)	54.2	16.2 min	71.90%	71.74%	20.46	19.99
15	DB,FR	*Conf. II* (2-edges paths)	256.3	19.2 min	74.10%	76.51%	19.59	17.95
16	DB,WK	*Conf. II* (2-edges paths)	125.3	17.6 min	68.92%	72.71%	20.71	19.97
17	FR,WK	*Conf. II* (2-edges paths)	172.0	17.8 min	76.08%	77.82%	19.08	17.59
18	DB,FR,WK	*Conf. II* (2-edges paths)	272.5	19.8 min	74.87%	77.16%	19.48	17.81
19	All 14 datasets	*Conf. II* (2-edges paths)	397.9	21.4 min	72.94%	75.63%	19.81	18.36

sion, the vectors produced by `LODVec` are given as input in WEKA API, by using a 10-fold cross validation [35].

Regarding Classification (CF), we use the default implementation of *Random Forest* (RF) and *Support Vector Machine* (SMO) of WEKA. For each model, we measure the accuracy percentage (percentage of correct predictions), i.e., the goal is to maximize that percentage. Concerning Regression (Reg), we use the default implementation of *Linear Regression* (LR) and *Random Forest* (RF) of WEKA, and we measure the root mean squared error (RMSE), i.e., the target is to minimize the RMSE value. Finally, we measure the accuracy and the RMSE for the trivial *Vote* method, that selects randomly a class (in classification) and a rating (in regression) for each entity.

Results for the evaluation datasets. Table 4, Table 5 and Table 6 show several statistics and experiments for movies, music albums and top NBA players, respectively. In each table, the first column indicates the ID of each row, whereas the second and the third one show the subset of RDF datasets and the configuration, which were used for creating the embeddings, respectively. We should note that for each configuration, we use eight different subsets (i.e., combinations) of RDF datasets for creating the embeddings. For each of these subsets, we present in the fourth column the average produced URI sequences per entity, and in the fifth column the total execution time of creating both the URI sequences and the embeddings. The columns 6 and 7 show the accuracy of the classification task (CF) and the last two columns show the RMSE value for the regression task (REG), by using different models. Below, we present the results for each evaluation dataset.

5.4.1 Results for Movies

In rows with ID 1-11 of Table 4, we show the results by following only single-edge paths, i.e., by using *Conf. I*, whereas the rows with ID 12-19 present the results by following also larger paths, i.e., by using *Conf. II*. We can see that the average number of URI

sequences and the execution time increases as we add more datasets (see the fourth and fifth column of Table 4). For example, for *Conf. I*, by using *DBpedia* (see row with ID 1) we needed 2.7 minutes and we created on average 23.8 URI sequences per movie, whereas by using all the available RDF datasets of `LODsyndesis` (see row with ID 8), the execution time was 5.5 minutes and the number of average URI sequences was 170.1.

Classification and Regression Results. First, for the trivial *Vote* method, we obtained 50% accuracy (CF), and the *RMSE* value was 23.1 (REG). Concerning *Conf. I*, the RDF dataset with the highest accuracy and the lowest RMSE is *FreeBase* [4] (see row with ID 2), i.e., we obtained 82% accuracy through *SVM* model, whereas its RMSE value was 16.37 (through *LR* model). The corresponding percentage for *DBpedia* and *Wikidata* were much smaller. However, by taking each pair of these 3 datasets (see rows with ID 4-6), the accuracy increased, and the RMSE value decreased in all cases (versus using only one dataset from each pair). Certainly, by using only *FreeBase*, we achieved better results comparing to use both *DBpedia* and *Wikidata*, which seems rational, since from *Freebase* we created a larger number of sequences. However, by combining *Freebase* with either *DBpedia* or *Wikidata*, or by using all 3 datasets (these combinations are feasible due to cross-dataset identity reasoning), the results improved. By using all the 14 RDF datasets (out of 400 datasets) containing data about these movies, we achieved the highest accuracy (84.7%) and the lowest RMSE (15.57).

Regarding the creation of the top-K URI sequences, we show some indicative experiments for *Conf. I* in rows with ID 9-11 of Table 4. By creating only the top-30 URI sequences according to objects frequency for each movie in descending order (i.e., triples with the 30 most popular objects per movie), we achieved higher accuracy lower RMSE, comparing to a random or an ascending order. However, the creation time of *desc* and *asc* is slower versus the other cases, since we send several requests to `LODsyndesis` REST API [19] for retrieving the frequency of each URI which occurs as the last object in each URI sequence.

Concerning the creation of larger URI sequences (for including more information for the actors of each movie), we obtained worse results comparing to the case of following only single-edge paths (i.e., see rows with ID 12-19 of Table 4). Only for the case of *Wikidata* we obtained higher accuracy and lower RMSE, by following also two-edges paths (see rows with ID 3 and 14).

Finally, for the classification task *SVM* outperformed *RF* in most cases, while for the regression task, *LR* was generally more effective than *RF*.

5.4.2 Results for Music Albums

In rows with ID 1-8 of Table 5, we can see experiments by creating embeddings only by following single-edge paths (see *Conf. IV* in Table 2), whereas in rows with ID 9-16 we present experiments by following larger paths (see *Conf. V* in Table 2). Concerning *Conf. IV*, by using only *DBpedia* (see row with ID 1), we created on average 16.3 URI sequences per album, whereas by using all the datasets, we created 35.9. Concerning the creation time of URI sequences, it is very low for *Conf. IV*, and it increases as we explore larger paths (i.e., for *Conf. V*).

Classification and Regression Results. First, for the trivial *Vote* method, we obtained 50% accuracy, whereas the *RMSE value* was 13.95. Concerning *Conf. IV* (i.e., only single-edge paths), the RDF dataset having the best performance for both classification

Table 5: Classification and regression experiments on Music Albums dataset

Row ID	RDF Datasets Used	Config. Name	$\|Seq_U(e)\|$ Average	Total Time	RF (CF)	SVM (CF)	RF (REG)	LR (REG)
1	DBpedia (DB)	*Conf. IV* (1-edge paths)	16.3	0.9 min	68.15%	68.21%	12.69	12.75
2	Freebase (FR)	*Conf. IV* (1-edge paths)	9.2	0.7 min	57.95%	56.57%	13.66	13.74
3	Wikidata (WK)	*Conf. IV* (1-edge paths)	6.7	0.6 min	59.74%	61.37%	13.85	13.89
4	DB,FR	*Conf. IV* (1-edge paths)	25.5	0.9 min	69.30%	68.64%	12.53	12.60
5	DB,WK	*Conf. IV* (1-edge paths)	20.5	0.9 min	67.61%	67.61%	12.76	12.85
6	FR,WK	*Conf. IV* (1-edge paths)	16.0	0.7 min	56.28%	57.65%	13.70	14.00
7	DB,FR,WK	*Conf. IV* (1-edge paths)	29.8	1.0 min	69.40%	69.02%	12.66	12.55
8	All 6 datasets	*Conf. IV* (1-edge paths)	35.9	1.1 min	70.81%	71.31%	12.55	12.41
9	DBpedia (DB)	*Conf. V* (2-edges paths)	91.4	2.5 min	68.50%	68.84%	12.77	12.62
10	Freebase (FR)	*Conf. V* (2-edges paths)	120.4	2.8 min	65.12%	62.78%	13.15	13.36
11	Wikidata (WK)	*Conf. V* (2-edges paths)	46.2	2.4 min	62.23%	61.45%	13.51	13.88
12	DB,FR	*Conf. V* (2-edges paths)	211.8	3.0 min	69.08%	69.2%	12.74	12.56
13	DB,WK	*Conf. V* (2-edges paths)	135.9	2.7 min	67.36%	69.58%	12.86	12.88
14	FR,WK	*Conf. V* (2-edges paths)	166.7	2.8 min	64.41%	64.70%	13.14	13.14
15	DB,FR,WK	*Conf. V* (2-edges paths)	256.4	3.3 min	70.09%	70.10%	12.76	12.42
16	All 20 datasets	*Conf. V* (2-edges paths)	365.0	5.5 min	**70.93%**	**72.32%**	**12.54**	**12.21**

Table 6: Classification and regression experiments on dataset for top 500 NBA basketball players of all time

Row ID	RDF Datasets Used	Config. Name	$\|Seq_U(e)\|$ Average	Total Time	RF (CF)	SVM (CF)	RF (REG)	LR (REG)
1	DBpedia (DB)	*Conf. VII* (1-edge paths)	37.9	0.4 min	70.55%	72.51%	121.7	123.0
2	Freebase (FR)	*Conf. VII* (1-edge paths)	71.2	0.4 min	72.91%	71.13%	123.1	128.2
3	Wikidata (WK)	*Conf. VII* (1-edge paths)	18.6	0.3 min	67.01%	66.86%	130.1	141.1
4	DB,FR	*Conf. VII* (1-edge paths)	109.1	0.5 min	75.74%	74.02%	115.8	124.5
5	DB,WK	*Conf. VII* (1-edge paths)	55.6	0.4 min	71.53%	71.50%	120.0	132.2
6	FR,WK	*Conf. VII* (1-edge paths)	89.8	0.4 min	74.21%	74.25%	121.3	129.6
7	DB,FR,WK	*Conf. VII* (1-edge paths)	126.9	0.6 min	76.90%	76.62%	114.5	**120.1**
8	All 15 datasets	*Conf. VII* (1-edge paths)	163.2	0.7 min	**77.03%**	**76.65%**	**114.4**	122.1

and regression is *DBpedia* (see row with ID 1 in Table 5), whereas *Freebase* is not so accurate for this task (see row with ID 2). Therefore, even by selecting to use exactly one RDF dataset for movies and music albums (i.e., the same dataset in both cases), we will not be able to obtain the best results for both tasks. Similarly to movies, as we add more datasets, the results are better for both regression and classification, except for the pairs containing the dataset *Wikidata* (see rows with ID 5-6). By including all the 6 available datasets for music albums (see row with ID 8 in Table 5), we obtained the highest accuracy (i.e., 71.31%) and the lowest *RMSE* value (i.e., 12.41), by using the *SVM* and the *LR* model, respectively.

Concerning the embeddings of *Conf. V* (see rows with ID 9-16 in Table 5), in all cases we obtained better results comparing to *Conf. IV*. Therefore, for this evaluation

Table 7: Results of $cmnEntitiesPer(E, K)$ for evaluating the task of finding similar entities by using 25 movies and 25 basketball players.

Datasets (used for creating URI sequences and embeddings)	$K = 1$ (Movies)	$K = 3$ (Movies)	$K = 5$ (Movies)	$K = 1$ (Basket-ball Players)	$K = 3$ (Basket-ball Players)	$K = 5$ (Basket-ball Players)
DBpedia	68.0%	56.0%	45.6%	56.0%	48.0%	47.2%
DBpedia, Freebase	80.0%	64.0%	53.6%	60.0%	57.3%	51.2%
DBpedia, Freebase, Wikidata	80.0%	69.3%	61.6%	88.0%	70.6%	68.0%
All RDF Datasets	**92.0%**	**80.0%**	**66.4%**	**92.0%**	**78.6%**	**70.4%**

dataset, it was effective to create more URI sequences by following also larger paths, i.e., URI sequences containing also information about the artist of each album. Indicatively, by using *Freebase* we can clearly see a high increase in the accuracy (+7.17%), and a decrease in the RMSE value (-0.51), by creating larger URI sequences (see rows with ID 2 and 10 of Table 5). Similarly to *Conf. IV*, we obtained the best results by using all the available datasets containing information for both music albums and artists (see row with ID 16), i.e., 72.32% accuracy (CF) and 12.21 RMSE value (Reg). Finally it is worth noting that 20 RDF datasets (out of 400) contain information about the artists, whereas only 6 of them about music albums. Therefore, we created a multiple "mixed-path" URI sequences by using *Conf. V*.

5.4.3 Results for top NBA Players

Table 6 shows the results for the top NBA players by using *Conf. VII*. As we can see, in all cases we needed less than 0.7 minutes for creating URI sequences and embeddings for 450 NBA players.

Classification and Regression Results. First, by using the trivial *Vote* method of *WEKA*, we obtained 50% accuracy, whereas the *RMSE* value was 153.1. Concerning classification, the RDF dataset with the highest accuracy (72.91%) is *Freebase* (see row with ID 2). By adding more datasets, the accuracy increases in most cases, whereas the highest accuracy, i.e., 77.03%, obtained by producing embeddings from all the available RDF datasets, and by using the *Random Forest* model (see row with ID 8). Regarding regression, the RDF dataset with the best performance is *DBpedia*, i.e., the *RMSE* value was 121.7, whereas we obtained again the best result by using all the RDF datasets and the *Random Forest* model (i.e., the *RMSE* was 114.4).

5.5 Task B. Finding Similar Entities

In this subsection, we introduce experiments for evaluating the effectiveness of the proposed approach, for retrieving similar entities. In particular, we selected 25 movies (from the set of 2,000 movies), and 25 basketball players (from the set of 12,250 basketball players). Then, we typed in *Google Search Engine* the corresponding keywords for each movie (e.g.,"Inception") and each basketball player (e.g., "Magic Johnson"), and we retrieved manually a list of related movies and basketball players, respectively. Afterwards, we created two evaluation datasets, which can be downloaded from `http://islcatalog.ics.forth.gr/dataset/lodvec`. For retrieving the list of related entities for each entity, we used the "People also Searched for" service of Google, i.e., it

Table 8: Indicative Example - Top-5 related movies to "Wall-E" movie by using different datasets. The bold indicates that an entity identified as related from both LODVec and *Google Search*

RDF Datasets	Position 1	Position 2	Position 3	Position 4	Position 5
DB	Pink Panther 2	**Ratatouille**	Shrek 2	Rain Mant	**Space Chimps**
DB,FR	**Finding Nemo**	**Toy Story 2**	**The Incredibles**	**Toy Story**	The Princess and the Frog
DB,FR,WK	**Finding Nemo**	**Toy Story 2**	**The Incredibles**	**Toy Story**	The Princess and the Frog
All Datasets	**Finding Nemo**	**Ratatouille**	**The Incredibles**	**Toy Story 3**	**Toy Story**

returns entities that have been searched for by people, who also searched for the focused entity (e.g., the movie "Inception"). For each entity, the corresponding (Google) list contains on average 24 related entities.

For this experiment we used the embeddings for 2,000 movies and 12,250 basketball players (according to *Conf. I* and *Conf. VIII*, respectively), by using a) only *DBpedia*, b) both *DBpedia* and *Freebase*, c) *DBpedia*, *Freebase* and *Wikidata*, and d) all the available RDF datasets. For finding the K most similar entities for each entity (i.e., movies and basketball players), LODVec uses the function "wordsNearest" from *dl4j* API.

5.5.1 What we measure.

Let $E' \subset E_{sel}$ be the entities that we use. Moreover, let $GoogleRelEnt(e)$ be the set of related entities for an entity $e \in E'$, derived through Google, whereas let $LODvecRelEnt(e, K)$ be the set of the top-K related entities for each entity e, derived through LODVec.

We measure the percentage of entities which are retrieved as similar, from both "Google Search Engine" and LODVec, i.e., we measure $cmnEntities(e, K) = |GoogleRelEnt(e) \cap LODvecRelEnt(e, K)|$, where $1 \leq K \leq |GoogleRelEnt(e)|$, and the range of this formula is $[0, K]$. Afterwards, we measure the average percentage of common entities, for all the entities of this experiment, i.e., $cmnEntitiesPer(E', K) = \frac{\sum_{\forall e \in E'} cmnEntities(e,K)}{|E'| * K}$ (i.e., a value with range [0,1]). We perform experiments for three different values of K, i.e., $K = 1$ and $K = 3$ and $K = 5$.

5.5.2 Results for Movies

In the second to fourth columns of Table 7, we can see the results for all the different values of K for Movies dataset. The best results obtained by using all the available RDF datasets. For example, for $K = 1$ (i.e., the top identified related movie of each movie), we achieved the highest accuracy, i.e., 92%, by using all the datasets. It means that in 23 (out of 25) cases, the most related movie identified from LODVec for each of these 23 movies, was also included in the corresponding list retrieved through *Google Search*. On the contrary, for $K = 5$ we obtained a 66.4% accuracy, i.e., on average 3.32 (out of 5) related movies of each movie that identified through LODVec, were also included in the *Google Search* list. As the K increases, the accuracy decreases for all the configurations. However, for any given K, when we add more datasets the percentage increases. Table 8 provides an indicative example for one of these 25 movies, i.e., the movie "WALL-E" and for $K = 5$, by using only *DBpedia*, 2 (out of 5) movies were in the list of related movies from *Google* (the bold ones in Table 8), whereas by using either *Freebase* and *DBpedia*,

Table 9: Indicative Example - Top-5 related basketball players to "Magic Johnson" by using different datasets. The bold indicates that an entity identified as related from both LODVec and *Google Search*

RDF Datasets	Position 1	Position 2	Position 3	Position 4	Position 5
DB	**Abdul-Jabaar**	James Worthy	John Stockton	**Shaquille O'Neal**	**Larry Bird**
DB,FR	**Julius Erving**	**Shaquille O'Neal**	Willies Reed	Pat Ewing	**Dwyane Wade**
DB,FR,WK	**Shaquille O'Neal**	Reggie Miller	**Dennis Rodman**	**Michael Jordan**	**Abdul-Jabaar**
All Datasets	**Michael Jordan**	**Dwyane Wade**	**Julius Erving**	**LeBron James**	**Shaquille O'Neal**

or these two datasets along with *Wikidata*, 4 (out of 5 movies) were at that list. Finally, by exploiting all the available RDF datasets, all the 5 movies were part of the list.

5.5.3 Results for Basketball Players

We have created URI sequences and embeddings for 12,250 players, therefore, we try to find which are the top-K similar basketball players (out of 12,250 players) for a given player. For creating the URI sequences and embeddings we needed 8 minutes by using *DBpedia* (on average 19.1 URI sequences per player), 8.2 minutes by using *DBpedia* and *Freebase* (on average 28.5 URI sequences), 8.9 minutes by using these two datasets and *Wikidata* (on average 43.5 URI sequences), and finally 10 minutes by using all the datasets (on average 54.8 URI sequences). Regarding the results, they are shown in the fifth to seventh columns of Table 7. Similarly to the case of movies, we obtained the best results by using all the available RDF datasets. Moreover, in all the cases the percentage increased when we added more datasets, especially by adding *Wikidata* to *DBpedia* and *Freebase*. Finally, Table 9 provides an example for the player "Magic Johnson" and for $K = 5$. Particularly, by using either one or two datasets, 3 (out of 5) players were included in the list of the related players of "Magic Johnson", derived from *Google*, i.e., the bold ones in Table 9. On the contrary, by using three RDF datasets, 4 (out of 5 players) occur in this list, whereas by using all the available RDF datasets, all the five basketball players were part of the list derived from *Google*.

In `http://islcatalog.ics.forth.gr/dataset/lodvec`, one can download all the derived related entities (identified from both *Google* and LODVec) for each of these a) 25 basketball players and b) 25 movies.

6 Discussion about the experimental results

Here, we provide four major conclusions regarding the experiments, concerning a) the efficiency of LODVec, b) the performance of single RDF datasets in different tasks, c) the gain of using more datasets for creating complementary URI sequences, and d) the importance of following larger paths for creating URI sequences.

Concerning a), we have seen that LODVec can produce URI sequences and embeddings quite fast even by using multiple datasets of *LODsyndesis* for thousands of entities, e.g., it needs approximately 1 minute for 1,600 music albums. Although the execution time increases as we follow larger paths, by using the "cache" mechanism the execution time can greatly decreased. Regarding b), we have showed that there is not a single RDF dataset (or knowledge graph) that can outperform the others for any possible task, e.g., *DBpedia* was more accurate for classifying the music albums, whereas *Freebase* was

more efficient for classifying the movies and the NBA players. Concerning c), we have observed that for each evaluation dataset we obtained better results by using multiple RDF datasets, instead of a single one, for the tasks of classification and regression, and for finding the most similar entities. Moreover, for each task we obtained the best results by using all the available RDF datasets of `LODsyndesis`. Finally, as regards d), it was effective to follow larger paths (including "mixed-paths") for creating also URI sequences for the artists of each music album, whereas it was not effective for creating larger sequences for the actors of the Movies dataset. It means that it is not always valuable to follow larger paths for creating URI sequences. A corresponding limitation of this paper (and a key research direction) is that we do not estimate whether a path is worth to be followed, i.e., the user selects manually the paths that will be followed for creating the desired URI sequences.

7 Conclusion

There is a lack of approaches that create URI embeddings from multiple RDF datasets. For this reason, we introduced a prototype called `LODVec` that exploits the semantically enriched indexes of `LODsyndesis` knowledge graph, and offers configurable options for creating URI sequences of any length for 412 million entities from 400 RDF datasets. Moreover, it offers a "cache" mechanism for avoiding to explore the same paths multiple times for creating URI sequences, and it uses *word2vec* model for converting the URI sequences to embeddings. The produced embeddings can be exploited in several tasks. In our case, we created URI embeddings for three evaluation datasets (containing movies, music albums and basketball players), by using multiple RDF datasets (after enriching them with the results of cross-dataset identity reasoning). We evaluated the gain of using several RDF datasets for creating embeddings for the tasks of classification, regression and for finding the most similar entities to a desired entity.

Concerning efficiency, `LODVec` can produce URI sequences and embeddings for over 1 thousand entities approximately in 1 minute, by using multiple RDF datasets. Moreover, we identified even 3.7× speedup by using the previously mentioned "cache" mechanism. Regarding effectiveness, in all tasks we obtained the best results by creating embeddings from all the available RDF datasets. Indicatively, by creating URI sequences and embeddings from 14 RDF datasets instead of using only *DBpedia*, we identified even 13% increase in the accuracy of predicting if a movie has a high or a low user rating (binary classification). As a future work, we plan (a) to estimate whether a path is worth to be followed, for aiding the user to create the most valuable URI sequences, i.e., URI sequences that can improve the effectiveness of a given task, (b) to create sequences containing also literals (e.g., [9]) and not only URIs, (c) to create vectors through other models, like *GloVe* [25] and BERT [7], and (d) to apply graph-based techniques, such as [6] and [33]. Finally, it would be interesting to use novel graph database platforms, such as *Neo4j* ([34]), for tackling index limitations (such as those presented in §4.3.3).

References

[1] A. Ammar and R. Celebi. Fact validation with knowledge graph embeddings. In M. C. Suárez-Figueroa, G. Cheng, et al., editors, *Proceedings of the ISWC 2019*

Satellite Tracks, volume 2456 of *CEUR Workshop Proceedings*, pages 125–128. CEUR-WS.org, 2019.

[2] G. Antoniou and F. van Harmelen. *A Semantic Web Primer, 2nd Edition*. Cooperative information systems. MIT Press, 2008.

[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. DBpedia: A nucleus for a web of open data. In K. Aberer, K. Choi, et al., editors, *The Semantic Web, 6th International Semantic Web Conference*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[4] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1247–1250. ACM, 2008.

[5] M. Cochez, P. Ristoski, S. P. Ponzetto, and H. Paulheim. Biased graph walks for RDF graph embeddings. In R. Akerkar, A. Cuzzocrea, et al., editors, *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, WIMS*, pages 21:1–21:12. ACM, 2017.

[6] M. Cochez, P. Ristoski, S. P. Ponzetto, and H. Paulheim. Global RDF vector space embeddings. In C. d'Amato, M. Fernández, et al., editors, *The Semantic Web - 16th International Semantic Web Conference (ISWC)*, volume 10587 of *Lecture Notes in Computer Science*, pages 190–207. Springer, 2017.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019.

[8] S. Eddamiri, E. M. Zemmouri, and A. Benghabrit. Graph embeddings for linked data clustering. In M. Indrawan-Santiago, E. Pardede, et al., editors, *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, pages 122–128. ACM, 2018.

[9] G. A. Gesese, R. Biswas, and H. Sack. A comprehensive survey of knowledge graph embeddings with literals: Techniques and applications. In M. Alam, D. Buscaldi, et al., editors, *Proceedings of the Workshop on Deep Learning for Knowledge Graphs (DL4KG2019)*, volume 2377 of *CEUR Workshop Proceedings*, pages 31–40. CEUR-WS.org, 2019.

[10] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[11] A. Hajra and K. Tochtermann. Linking science: approaches for linking scientific publications across different LOD repositories. *International Journal of Metadata, Semantics and Ontologies*, 12(2/3):124–141, 2017.

[12] S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 query language. *W3C recommendation*, 21(10):778, 2013.

[13] E. Inan and O. Dikenelli. Effect of enriched ontology structures on RDF embedding-based entity linking. In E. Garoufallou, S. Virkus, et al., editors, *Metadata and Semantic Research - 11th International Conference, MTSR*, volume 755 of *Communications in Computer and Information Science*, pages 15–24. Springer, 2017.

[14] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu. Learning entity and relation embeddings for knowledge graph completion. In B. Bonet and S. Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2181–2187. AAAI Press, 2015.

[15] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. 2013.

[16] N. Mohapatra, V. Iosifidis, A. Ekbal, S. Dietze, and P. Fafalios. Time-aware and corpus-specific entity relatedness. In M. Cochez, T. Declerck, et al., editors, *Proceedings of the First Workshop on Deep Learning for Knowledge Graphs and Semantic Technologies (DL4KGS)*, volume 2106 of *CEUR Workshop Proceedings*, pages 33–39. CEUR-WS.org, 2018.

[17] M. Mountantonakis and Y. Tzitzikas. How linked data can aid machine learning-based tasks. In J. Kamps, G. Tsakonas, et al., editors, *21st International Conference on Theory and Practice of Digital Libraries, TPDL*, volume 10450 of *Lecture Notes in Computer Science*, pages 155–168. Springer, 2017.

[18] M. Mountantonakis and Y. Tzitzikas. High performance methods for linked open data connectivity analytics. *Information, MDPI*, 9(6):134, 2018.

[19] M. Mountantonakis and Y. Tzitzikas. LODsyndesis: Global scale knowledge services. *Heritage*, 1(2):335–348, 2018.

[20] M. Mountantonakis and Y. Tzitzikas. Knowledge graph embeddings over hundreds of linked datasets. In E. Garoufallou, F. Fallucchi, and E. W. D. Luca, editors, *Metadata and Semantic Research - 13th International Conference (MTSR)*, volume 1057 of *Communications in Computer and Information Science*, pages 150–162. Springer, Cham, 2019.

[21] M. Mountantonakis and Y. Tzitzikas. Large-scale semantic integration of linked data: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–40, 2019.

[22] D. Moussallem, T. Soru, and A. N. Ngomo. THOTH: neural translation and enrichment of knowledge graphs. In C. Ghidini, O. Hartig, et al., editors, *The Semantic Web - 18th International Semantic Web Conference*, volume 11778 of *Lecture Notes in Computer Science*, pages 505–522. Springer, 2019.

[23] Y. Nechaev, F. Corcoglioniti, and C. Giuliano. Type prediction combining linked open data and social media. In A. Cuzzocrea, J. Allan, et al., editors, *Proceedings of the 27th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1033–1042. ACM, 2018.

[24] F. Nikolaev and A. Kotov. Joint word and entity embeddings for entity retrieval from a knowledge graph. In J. M. Jose, E. Yilmaz, et al., editors, *42nd European Conference on IR Research, ECIR 2020*, volume 12035 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2020.

[25] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543. ACL, 2014.

[26] J. Portisch, M. Hladik, and H. Paulheim. Kgvec2go - knowledge graph embeddings as a service. In N. Calzolari, F. Béchet, et al., editors, *Proceedings of The 12th Language Resources and Evaluation Conference (LREC)*, pages 5641–5647. European Language Resources Association, 2020.

[27] P. Ristoski, C. Bizer, and H. Paulheim. Mining the web of linked data with RapidMiner. *Journal of Web Semantics*, 35:142–151, 2015.

[28] P. Ristoski, G. K. D. de Vries, and H. Paulheim. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In P. Groth, E. Simperl, et al., editors, *The Semantic Web - 15th International Semantic Web Conference*, volume 9982 of *Lecture Notes in Computer Science*, pages 186–194, 2016.

[29] P. Ristoski, J. Rosati, T. D. Noia, R. D. Leone, and H. Paulheim. RDF2Vec: RDF graph embeddings and their applications. *Semantic Web*, 10(4):721–752, 2019.

[30] M. R. Saeed, C. Chelmis, and V. K. Prasanna. Extracting entity-specific substructures for RDF graph embeddings. *Semantic Web*, 10(6):1087–1108, 2019.

[31] D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.

[32] Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Trans. Knowl. Data Eng.*, 29(12):2724–2743, 2017.

[33] Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph embedding by translating on hyperplanes. In C. E. Brodley and P. Stone, editors, *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 1112–1119. AAAI Press, 2014.

[34] J. Webber. A programmatic introduction to neo4j. In G. T. Leavens, editor, *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12*, pages 217–218. ACM, 2012.

[35] I. H. Witten, E. Frank, and M. A. Hall. *Data mining: practical machine learning tools and techniques, 3rd Edition*. Morgan Kaufmann, Elsevier, 2011.