pyRDF2Vec: A Python Implementation and Extension of RDF2Vec

Bram Steenwinckel ✉, Gilles Vandewiele, Terencio Agozzino, and Femke Ongenae

IDLab, Ghent University – imec, 9000 Gent, Belgium
bram.steenwinckel@ugent.be

Abstract. This paper introduces pyRDF2Vec, a Python software package that reimplements the well-known RDF2Vec algorithm along with several of its extensions. By making the algorithm available in the most popular data science language, and by bundling all extensions into a single place, the use of RDF2Vec is simplified for data scientists. The package is released under an MIT license and structured in such a way to foster further research into sampling, walking, and embedding strategies, which are vital components of the RDF2Vec algorithm. Several optimisations have been implemented in pyRDF2Vec that allow for more efficient walk extraction than the original algorithm. Furthermore, best practices in terms of code styling, testing, and documentation were applied such that the package is future-proof as well as to facilitate external contributions.

Keywords: RDF2Vec · walk-based embeddings · open source

Resource type: Software
License: MIT license
URL: https://github.com/IBCNServices/pyRDF2Vec

1 Introduction

Knowledge Graphs (KGs) are an ideal candidate to perform hybrid Machine Learning (ML) where both background and observational knowledge are taken into account to construct predictive models. However, since KGs are symbolic data structures, they cannot be fed to ML algorithms directly and first require a non-trivial transformation step in which symbolic substructures of the graph are converted into numerical representations. These transformation techniques can typically be classified as being *feature*-based or *embedding*-based [26]. Feature-based approaches are often interpretable, but require domain knowledge about the task at hand and are effort-intensive. Embedding-based approaches, on the other hand, are typically agnostic to the task and are usually able to outperform their feature-based counterparts. Resource Description Framework To Vector (RDF2Vec) [18] is an unsupervised, task-agnostic, and embedding-based approach that has gained significant popularity over the past few years. RDF2Vec

builds on the popular Natural Language Processing (NLP) technique Word2Vec. The latter generates embeddings for different tokens present in a corpus, by training a neural network in an unsupervised way that must predict either a token based on its context (Continuous Bag of Words) or the context based on a token (Skip-Gram). The corpus, fed to Word2Vec, is constructed by extracting a large number of walks from the KG. A walk is a sequence of entities obtained from the KG by starting at a certain entity and traversing the directed edges.

Since its initial publication, in 2017, many extensions to the algorithm have been proposed. However, each of these extensions are individual implementations, which complicates combining several of them. Moreover, the original code for RDF2Vec was written in Java, which is significantly less popular than Python for data science, according to the Kaggle Survey 2022[1]. In Figure 1, the answers to the question "What programming languages do you use on a regular basis?", where multiple answers were possible, are depicted. It should be noted that among the 3862 of the people who selected Java as being used regularly, only 461 did not pick Python. This makes it difficult to integrate the original RDF2Vec implementation into a data science pipeline, which is typically written in Python.

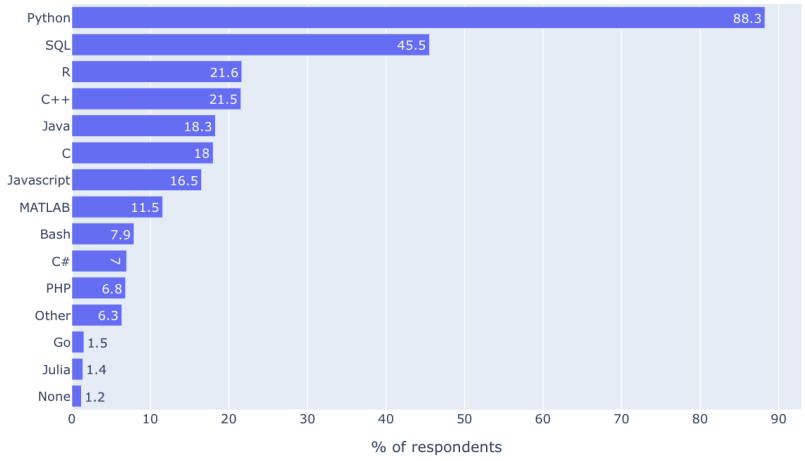

Fig. 1: Programming languages used by data scientists according to the Kaggle Survey 2021.

In this paper, we present `pyRDF2Vec`, a Python implementation of the original algorithm and many of its extensions. Moreover, various mechanisms are built, allowing to better handle large KGs. The code is released under an open-source license and is written in a way to facilitate further research into the different com-

[1] `https://www.kaggle.com/c/kaggle-survey-2022`

ponents of the RDF2Vec algorithm. The remainder of this paper is structured as follows. In Section 2, we provide background on representation learning for KGs, followed by an in-depth discussion of RDF2Vec and its extensions. Then, in Section 3, we present the architecture of our `pyRDF2Vec` package and the mechanisms set in-place to easily allow for contributions by others. In Section 4, we discuss some studies and other software packages that have already made use of `pyRDF2Vec`. Finally, we conclude our paper in Section 6. In Appendix A, we provide a code snippet that shows how `pyRDF2Vec` can be used.

2 Background

In this section, we describe the necessary background to elaborate upon `pyRDF2Vec`. First, we will discuss related work regarding the transformation of a KG into numerical representations. Afterwards, we outline an in-depth overview of how RDF2Vec works and its extensions released over the past few years.

2.1 Representation Learning

As mentioned in the introduction, a *feature*-based or *embedding*-based transformation step is required that converts the symbolic KGs into numerical vectors before they can be used in ML models. Especially embedding-based approaches, which make use of Deep Learning techniques, have gained increasing popularity over the past few years as these can be applied out-of-the-box and can run efficiently on Graphical Processing Units (GPUs), which are quite commonly available today. Moreover, the largest advantage of embedding-based techniques is that they are typically task-agnostic and as such do not require extensive domain knowledge and/or significant effort, as opposed to feature-based approaches. A further distinction can be made between embedding-based techniques. The first category consists of techniques that learn embeddings either through tensor factorisation or through negative sampling [15,3,26], e.g. TransE [1]. A second category consists of Deep Learning architectures that make use of parameterised transformations, based on information from the neighbourhood of a node that is collected through message passing [19], e.g. Relational Graph Convolutional Networks (R-GCN). The parameters of this transformation are learned through back-propagation in a supervised fashion. A third, and final, category adapts existing NLP techniques, such as Word2Vec [13], to work on graph structures. RDF2Vec belongs to this final category [18].

2.2 RDF2Vec

RDF2Vec is an unsupervised, task-agnostic algorithm that achieves state-of-the-art performances on many benchmark datasets [18]. It extends Word2Vec to work on graph structures by first extracting walks that serve as a corpus. Each walk can be seen as a sentence of a corpus and each hop within such walks corresponds to a token. Word2Vec will then learn embeddings for each of these tokens in an

unsupervised matter by learning to predict either a token based on its context (Continuous Bag of Words), or the context based on a token (Skip-Gram). Over the past few years, several extensions to RDF2Vec have been suggested, which we will discuss subsequently. A good up-to-date overview of how RDF2Vec works, which extensions have been proposed over the last few years, and of applications that make use of RDF2Vec can be found on a website hosted by the original authors[2].

The number of walks that can be extracted quickly grows, depending on the depth of those walks and the size of the KG. As such, exhaustively extracting every possible walk becomes infeasible rather quickly. As a solution, Cochez et al. [4] proposed several sampling, or biased walking, techniques which enable only extracting a subset of walks that still capture most of the information. Recently, more sampling strategies have been proposed: (i) utilising page transition probabilities [25], (ii) using Metropolis-Hastings sampling [27], or (iii) other forms of prior knowledge [14].

Originally, the RDF2Vec algorithm used random walking and the Weisfeiler-Lehman paradigm to extract the corpus of walks for Word2Vec. However, within the domain of graph-based ML, walking techniques that are more advanced than random sampling have been suggested over the past few years. In addition, it has been shown that the Weisfeiler-Lehman paradigm introduces little to no extra information in the extracted walks. As such, Vandewiele et al. evaluated different walking strategies on several benchmark datasets to show that there is no one-size-fits-all strategy, and that tuning the strategy for the task at hand can result in increased performances [23].

Finally, Portisch et al. [17] applied an order-aware variant of Word2Vec to the corpus extracted by the walking and sampling strategies, which resulted in significantly increased predictive performances on multiple benchmark datasets.

3 `pyRDF2Vec`

In this section, we elaborate upon our `pyRDF2Vec` package. We first present its architecture, then give an overview of all the extensions available today and finally discuss the different mechanisms implemented to facilitate external contributions.

3.1 Architecture

In Figure 2, an overview of the `pyRDF2Vec` workflow is provided. Seven main modules are used, which we now discuss subsequently.

[2] `www.rdf2vec.org`

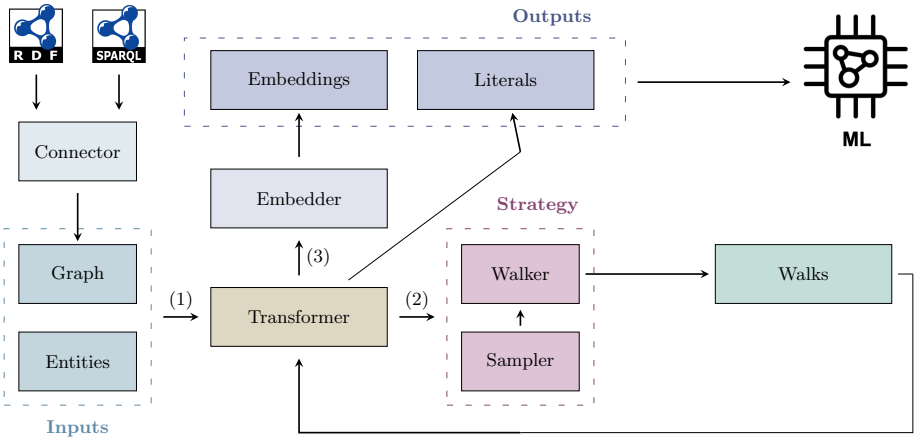

Fig. 2: Workflow of `pyRDF2Vec`. A `Graph` and collection of `Entities` are provided by the user to the `Transformer` (1), which is instantiated with a list of different strategies consisting of a `Walker` and `Sampler` (2). The latter are responsible for extracting walks from the `Graph` which are, in turn, fed to the embedder to calculate `Embeddings` (3). In addition, the `Transformer` also extracts `Literals` by following paths specified by the user.

1. **Connector**: coordinates the interaction with a local or remote graph. For KGs located on hard disk, `pyRDF2Vec` uses `rdflib` to load the graph into memory. If required, walk extraction from remote graphs is also possible through a SPARQL endpoint. Additional connectors can be implemented based on the provided `Connector` base class.
2. **Graph**: is the internal representation of the KG based on the representation of De Vries et al. [5]. It is used to efficiently traverse the graph and to store additional information regarding nodes and edges without being dependant upon other Python packages. As this representation removes the multi-relational aspect of the KG by transforming the edges to intermediate nodes, it enables `pyRDF2Vec` to create embeddings for predicates.
3. **Entities**: is the set of nodes within the graph for which we want to generate embeddings. These entities will serve as the starting points for the walk extraction and need to be provided by the user. It should however be noted that in fact all of the entities that appear in these extracted walks will have an associated embedding.
4. **Transformer**: the main interface for users that combines all other components.
5. **Sampler**: prioritises the use of some edges in the graph over others using a weight allocation strategy. The current `pyRDF2Vec` version implemented each of the sampling techniques described by Cochez et al. [4]. The currently supported sampling strategies are:
   - Uniform sampling: assigns a uniform weight to each edge.

- Object frequency: prioritizes walks containing edges with the highest degree objects. The degree of an object is defined by the number of predicates present in its neighbourhood.
- Predicate frequency: prioritizes walks containing edges with the highest degree predicates. The degree of a predicate is defined by the number of occurrences a predicate appears in the graph.
- Predicate-Object frequency: prioritizes walks containing edges with the highest degree of (predicate, object) relations. The degree of such relation is defined by the number of occurrences that a (predicate, object) relation appears in the graph.
- Wide: gives priority to walks containing edges with the highest degree of predicates and objects. The degree of a predicate and an object is defined by the number of predicates and objects present in its neighbourhood, but also by their number of occurrences in the graph.
- PageRank: prioritizes walks containing the most frequent objects. This frequency is defined by assigning a higher weight to the most frequent objects using the PageRank ranking.

Additional sampling techniques can easily be implemented, according to the provided `Sampler` base class.

6. **Walker**: responsible for extracting walks from the KG. Different walking strategies, proposed by Vandewiele et al. [23] are incorporated in the current `pyRDF2Vec` version. The currently supported walking strategies are:
   - Random: equal probability to select a hop within our walk
   - Weisfeiler-Lehman: selecting hops based on the Weisfeiler-Lehman kernel.
   - Walklets: walks of length two containing the root node and one of the hops.
   - Anonymous: random walks but neglecting the label information
   - HALK: hierarchical random walks, removing rare hops
   - N-Gram: one-to-many mapping within walks by introducing wild cards
   - Community: provide a probability to hop to important (community) nodes within the graph.

   New walking strategies can be implemented using the `Walker` base class.
7. **Embedder**: is in charge of transforming the extracted walks into embeddings, based on a trained model. By default, Word2Vec is used within this embedder code to generate these embeddings. A fastText [9] embedder is also made available in the current `pyRDF2Vec` version and additional embedding techniques can be added by using the `Embedder` base class.

It is important to `Connector`, `Sampler`, `Walker`, and `Embedder` expose interfaces that can be implemented by users. That way, we hope to both facilitate and stimulate further research into these components of the RDF2Vec algorithm.

3.2 Optimizations and Extensions

The `pyRDF2Vec` implementation has several extensions, that speed up walk extraction and provide information in addition to the embeddings based on walks.

First, the `Transformer` takes a list of `Walker` strategies, with optionally associated `Sampler` strategies, which enables the combination of several strategies. This allows for further research into techniques similar to ensembling, where the information obtained from several strategies is combined. This combination can be done either (i) on corpus-level, by concatenating the walks extracted by the different strategies together before feeding them to the `Embedder`, (ii) on the embedding level, where embeddings are learned on the corpora of each strategy individually and then aggregated, or (iii) on prediction level, where the embeddings learned on each corpus are fed to a classifier to make predictions for the downstream task and then aggregated. The combination of different strategies is illustrated in the example code provided in Appendix A.

A second extension in the `pyRDF2Vec` allows the extraction of literal information in addition to the embeddings learned, based on the graph structure surrounding entities of interest. To achieve this, the user can specify a set of paths, starting from the nodes provided in `Entities`, for which literal information can be found. `pyRDF2Vec` will then traverse these paths and return (i) `NaN` if the literal cannot be found, (ii) a scalar in case exactly one literal can be found, and (iii) a list of literals in case the path to a literal can be found multiple times. From then on, the user can process this information and concatenate this to the provided embeddings. The usage of literal information is illustrated in the example code provided in Appendix A.

`pyRDF2Vec` enables reverse walking by traversing across incoming edges as opposed to outgoing edges. This is due to the fact that the direction of certain predicates is chosen rather arbitrarily [e.g., (`Brussels`, `isCapitalOf`, `Belgium`) vs. (`Belgium`, `hasCapital`, `Brussels`)]. This also allows for nodes from `Entities` to be in positions different from the starting position within walks.

Several mechanisms are implemented to speed up the extraction: (i) SPARQL requests to find the next hop in walks can be bundled together to reduce overhead introduced by HTTP when a remote KG is used, (ii) multi-threading is enabled to parallelize the extraction of walks, and (iii) caching is implemented to avoid redundant requests. To show the effect of these mechanisms, a benchmark evaluation on three well-known datasets from the original RDF2Vec paper was performed. In this benchmarking approach, a comparison was made between the fully optimized `pyRDF2Vec` library and a version resembling the original RDF2Vec approach. The results, for a varying amount of entities for which embeddings were created, are provided in Table 1. For large datasets, the reduction in time is more than 50%. For smaller datasets such as the MUTAG datasets, the optimized `pyRDF2Vec` package can be up to 10 times faster.

3.3 CI/CT/CD and Documentation

To facilitate contributions by the open-source community to our code repository, multiple mechanisms have been set up. First, Continuous Integration (CI),

Table 1: Evaluation of the SPARQL bundling, multi-threading and cache optimisations for different datasets in function of the number of entities. The time measurements are averages and their standard deviations over 10 different runs. The last column shows the relative speedup of `pyRDF2Vec` compared to the original, non-optimized, RDF2Vec implementation in Python.

Dataset	**Entities**	**Depth**	**#Walks**	**Time** (s)		**Speedup**
				`(py)RDF2Vec`	`pyRDF2Vec`	
MUTAG	25	4	500	74.85 ± 13.88	7.30 ± 0.34	10.25
	50			132.99 ± 24.02	14.75 ± 0.75	9.02
	100			255.83 ± 35.86	28.20 ± 1.06	9.07
AM	25	4	500	87.92 ± 17.87	9.83 ± 0.51	8.94
	50			207.97 ± 30.11	82.84 ± 4.61	2.51
	100			339.50 ± 39.52	87.68 ± 3.70	3.87
DBP:Cities	25	4	500	541.89 ± 29.63	218.43 ± 16.41	2.48
	50			1037.23 ± 84.15	401.44 ± 48.54	2.58
	100			1950.79 ± 132.02	764.23 ± 115.90	2.55

through the use of GitHub Actions[3], is implemented which makes sure that the merge of the work of several developers does not impact the release of a project. With each push to one of the branches, several checks are performed, such as checking whether any styling guidelines have been violated. Second, Continuous Delivery (CD) is guaranteed as the `main` is always supposed to be the stable branch for which the checks performed by the CI pass. Added to that, the use of `poetry` [4] as dependency manager helps to facilitate future releases of `pyRDF2Vec` to the PyPI platform. Finally, a Continuous Testing (CT) mechanism executes a battery of unit tests, using `pytest` [5], for every push to the code repository. Afterwards, a coverage report is generated. With the help of these continuous methods, `pyRDF2Vec` has been able to release several new features and fix bugs to increase its stability, popularity, and notoriety.

Having an up-to-date and clear documentation is essential for the proper use of a library and its evolution. Good documentation will make it easier to use and contribute to a library. To improve the clarity of the documentation in Python, `mypy` [6], an optional static type checker, can also be used in addition to PyDoc. While Python is natively a dynamically typed language, the use of such a static type checker requires that consistent types are filled in, which improved

[3] `https://github.com/features/actions`
[4] `https://python-poetry.org/`
[5] `www.pytest.org`
[6] `http://mypy-lang.org/`

documentation. Finally, this documentation generation is done with Sphinx [7] and is automatically updated on the online website hosted by Read the Docs, at each commit on the `main` branch.

4 Package Usage

At the time of writing, `pyRDF2Vec` has amassed 180 stars on Github and 24700 downloads according to PePy[8]. An overview of the number of downloads for the latest six months can be found in Figure 3.

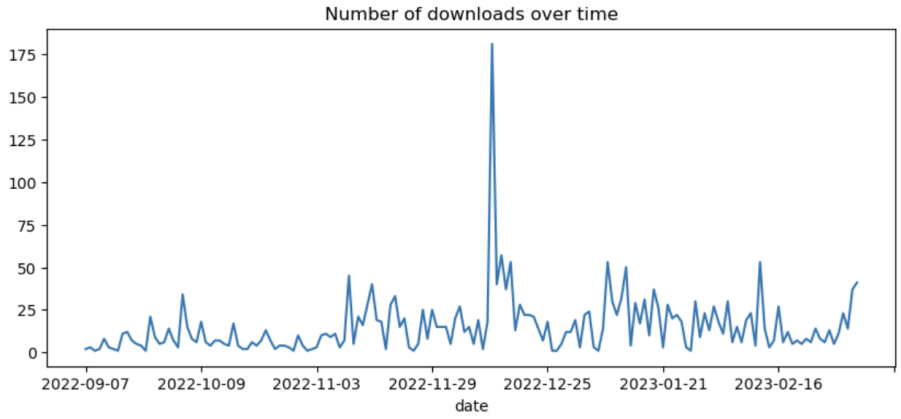

Fig. 3: The number of downloads of the last 180 days of our `pyRDF2Vec` package.

`pyRDF2Vec` has been used in several research projects and practical use cases. As of today, `pyRDF2Vec` appears in 40 studies published on Google Scholar[9]. We now give a brief overview of these studies. Ontowalk2vec [8] and Owl2Vec* [2] extend `pyRDF2Vec` to embed concepts by extracting walks from ontology information. Iana et al. [11] showed that applying reasoning to infer extra information in the KG before extracting walks results in little to no increased predictive performance. Portisch et al. [16] compared embedding techniques suited for link prediction and suited for data mining on both link prediction and data mining tasks. `pyRDF2Vec` was used as one of the data mining techniques during evaluation. In [12], `pyRDF2Vec` among many other embedding techniques, has been compared to non-embedding methods to better understand their semantic capabilities. In Sousa et al. [22] `pyRDF2Vec` is used to tailor aspect-oriented semantic similarity measures to fit a particular view on biological similarity or relatedness

[7] `https://www.sphinx-doc.org/`
[8] `https://pepy.tech/project/pyRDF2Vec`
[9] `https://scholar.google.com/scholar?q="pyRDF2Vec"`

in protein-protein, protein function similarity, protein sequence similarity and phenotype-based gene similarity tasks. Engleitner et al. [7] compare `pyRDF2Vec` with other embedding techniques for news article tag recommendation. Shi et al. [21,20] use `pyRDF2Vec` to calculate semantic similarity between concepts in several datasets. Gurbuz et al. [10] evaluate many different techniques, including `pyRDF2Vec`, for explainable target-disease link prediction. Steenwinckel et al. [24] compare their newly proposed technique, INK, to state-of-the-art techniques such as `pyRDF2Vec`. Finally, Degraeve et al. [6] qualitatively compare embeddings produced by `pyRDF2Vec` with embeddings produced by their proposed RR-GCN through a t-SNE plot.

5 Discussion

In the previous sections, we showed how and why we designed `pyRDF2Vec`. The number of downloads or the number of stars shows the interest in the created package, but it does not directly show its research impact. Many researchers already depend upon this resource as shown in the previous section. They use embeddings in a wide research field, far beyond the scopes of the semantic web community. This is also reflected in the questions asked as GitHub issues, where the authors of this paper frequently have to explain some key concepts within our community (such as Literals, SPARQL, remote endpoints, etc.).

Besides its popularity, the `pyRDF2Vec` package is created to be extended and used in many application domains. The original RDF2Vec package had some limitations regarding extendability. To make sure new research ideas could be implemented based on the original RDF2Vec idea, a redesign of the Graph-Transformer-Walker-Embedder was, to our knowledge, needed. Separating all these key components in a new architecture benefits both the maintenance of this package and it resulted in the optimizations to deal with larger and more complex KGs.

6 Conclusion and Future Work

This paper presented the `pyRDF2Vec` software package. It reimplements the well-known RDF2Vec algorithm in Python, as this language is several significantly more popular in the data science community than Java, in which RDF2Vec was originally implemented. This reimplementation allows data scientists to integrate RDF2Vec immediately into their pipeline. Many optimisations regarding the walking algorithm were added to ensure this package can extract embeddings fast while handling large knowledge graphs. In addition to the original algorithm, `pyRDF2Vec` implements many extensions that have already been published, provides additional information and can handle literals. The fact that these extensions are bundled in a single place could facilitate future research.

The pyRDF2Vec architecture is set up in such a way, in combination with automatic styling, testing, and documentation to foster future external contributions. Several research projects and use cases have already used pyRDF2Vec in their experimentation or as a basis for their code, which we discuss in this paper.

Resource Availability Statement: pyRDF2Vec is available under a MIT license on Github[10].

References

1. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. Advances in neural information processing systems **26** (2013)
2. Chen, J., Hu, P., Jimenez-Ruiz, E., Holter, O.M., Antonyrajah, D., Horrocks, I.: Owl2vec*: Embedding of owl ontologies. Machine Learning **110**(7), 1813–1845 (2021)
3. Choudhary, S., Luthra, T., Mittal, A., Singh, R.: A survey of knowledge graph embedding and their applications. arXiv preprint arXiv:2107.07842 (2021)
4. Cochez, M., Ristoski, P., Ponzetto, S.P., Paulheim, H.: Biased graph walks for rdf graph embeddings. In: Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics. pp. 1–12 (2017)
5. De Vries, G.K.D., De Rooij, S.: Substructure counting graph kernels for machine learning from rdf data. Journal of Web Semantics **35**, 71–84 (2015)
6. Degraeve, V., Vandewiele, G., Ongenae, F., Van Hoecke, S.: R-gcn: The r could stand for random. arXiv preprint arXiv:2203.02424 (2022)
7. Engleitner, N., Kreiner, W., Schwarz, N., Kopetzky, T., Ehrlinger, L.: Knowledge graph embeddings for news article tag recommendation. In: Joint Proceedings of the Semantics co-located events: Poster\&Demo track and Workshop on Ontology-Driven Conceptual Modelling of Digital Twins co-located with Semantics 2021, Amsterdam and Online, September 6-9, 2021. CEUR-WS. org (2021)
8. Gkotse, B., Jouvelot, P., Ravotti, F.: Ontology Embeddings with ontowalk2vec: an Application to UI Personalisation. Ph.D. thesis, MINES ParisTech-PSL Research University; CERN-Suisse (2022)
9. Grave, E., Bojanowski, P., Gupta, P., Joulin, A., Mikolov, T.: Learning word vectors for 157 languages. In: Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018) (2018)
10. Gurbuz, O., Alanis-Lobato, G., Picart-Armada, S., Sun, M., Haslinger, C., Lawless, N., Fernandez-Albert, F.: Knowledge graphs for indication expansion: An explainable target-disease prediction method. Frontiers in genetics **13**, 814093–814093 (2022)
11. Iana, A., Paulheim, H.: More is not always better: The negative impact of a-box materialization on rdf2vec knowledge graph embeddings. arXiv preprint arXiv:2009.00318 (2020)
12. Jain, N., Kalo, J.C., Balke, W.T., Krestel, R.: Do embeddings actually capture knowledge graph semantics? In: European Semantic Web Conference. pp. 143–159. Springer (2021)

[10] https://github.com/IBCNServices/pyRDF2Vec

13. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space (2013)
14. Mukherjee, S., Oates, T., Wright, R.: Graph node embeddings using domain-aware biased random walks. arXiv preprint arXiv:1908.02947 (2019)
15. Nickel, M., Murphy, K., Tresp, V., Gabrilovich, E.: A review of relational machine learning for knowledge graphs. Proceedings of the IEEE **104**(1), 11–33 (2015)
16. Portisch, J., Heist, N., Paulheim, H.: Knowledge graph embedding for data mining vs. knowledge graph embedding for link prediction–two sides of the same coin? Semantic Web **13**(3), 399–422 (2022)
17. Portisch, J., Paulheim, H.: Putting rdf2vec in order. arXiv preprint arXiv:2108.05280 (2021)
18. Ristoski, P., Rosati, J., Di Noia, T., De Leone, R., Paulheim, H.: Rdf2vec: Rdf graph embeddings and their applications. Semantic Web **10**(4), 721–752 (2019)
19. Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: European semantic web conference. pp. 593–607. Springer (2018)
20. Shi, Y., Cheng, G., Tran, T.K., Kharlamov, E., Shen, Y.: Efficient computation of semantically cohesive subgraphs for keyword-based knowledge graph exploration. In: Proceedings of the Web Conference 2021. pp. 1410–1421 (2021)
21. Shi, Y., Cheng, G., Tran, T.K., Tang, J., Kharlamov, E.: Keyword-based knowledge graph exploration based on quadratic group steiner trees. In: IJCAI. vol. 2021, pp. 1555–1562 (2021)
22. Sousa, R.T., Silva, S., Pesquita, C.: Supervised semantic similarity. bioRxiv (2021)
23. Steenwinckel, B., Vandewiele, G., Bonte, P., Weyns, M., Paulheim, H., Ristoski, P., De Turck, F., Ongenae, F.: Walk extraction strategies for node embeddings with rdf2vec in knowledge graphs. In: Database and Expert Systems Applications-DEXA 2021 Workshops: BIOKDD, IWCFS, MLKgraphs, AI-CARES, ProTime, AISys 2021, Virtual Event, September 27–30, 2021, Proceedings 32. pp. 70–80. Springer (2021)
24. Steenwinckel, B., Vandewiele, G., Weyns, M., Agozzino, T., Turck, F.D., Ongenae, F.: Ink: knowledge graph embeddings for node classification. Data Mining and Knowledge Discovery pp. 1–48 (2022)
25. Taweel, A.A., Paulheim, H.: Towards exploiting implicit human feedback for improving rdf2vec embeddings. arXiv preprint arXiv:2004.04423 (2020)
26. Wang, Q., Mao, Z., Wang, B., Guo, L.: Knowledge graph embedding: A survey of approaches and applications. IEEE Transactions on Knowledge and Data Engineering **29**(12), 2724–2743 (2017)
27. Zhang, S., Lin, X., Zhang, X.: Discovering dti and ddi by knowledge graph with mhrw and improved neural network. In: 2021 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). pp. 588–593. IEEE (2021)

A Appendix: Example Usage

We now provide a simple code snippet in Listing 1 that demonstrates how a user can generate embeddings for nodes of interest in his/her KG with just a few lines of code.

Listing 1: Example usage of pyRDF2Vec

```
# entities is a list of URIs which we want to embed.
entities = [ ... ]

# Loads a KG object from hard disk, removes triples with
# "dl#isMutagenic" as predicate, and specifies the paths
# where literals can be found.
dl = "http://dl-learner.org/carcinogenesis"
kg = KG(
    "mutag.owl",
    skip_predicates={dl + "#isMutagenic"},
    literals=[
        [
            dl + "#hasBond",
            dl + "#inBond",
        ],
        [
            dl + "#hasAtom",
            dl + "#charge",
        ],
    ]
)

# Create a Word2Vec embedder that trains for ten epochs.
embedder = Word2Vec(workers=1, epochs=10)

# Create a Sampler that uses PageRank (damping 0.85).
sampler = PageRankSampler(alpha=0.85)

# Use HALK strategy to extract all walks of depth 2.
walker1 = HALKWalker(2, None, n_jobs=4, sampler=None)

# Create walker that samples 100 walks per entity.
walker2 = RandomWalker(2, 100, n_jobs=4, sampler=sampler)

# Create our transformer object.
transformer = RDF2VecTransformer(
    embedder,
    walkers=[walker1, walker2]
)

# Extract the embeddings and literals.
embeddings, literals = transformer.fit_transform(kg, entities)
```