Global RDF Vector Space Embeddings

Michael Cochez[1,2,3(✉)], Petar Ristoski[4], Simone Paolo Ponzetto[4], and Heiko Paulheim[4]

[1] Fraunhofer FIT, 53754 Sankt Augustin, Germany
michael.cochez@fit.fraunhofer.de
[2] Informatik 5, RWTH University Aachen, Aachen, Germany
[3] Faculty of Information Technology, University of Jyvaskyla, Jyväskylä, Finland
[4] Data and Web Science Group, University of Mannheim, Mannheim, Germany
{petar.ristoski,simone,heiko}@informatik.uni-mannheim.de

Abstract. Vector space embeddings have been shown to perform well when using RDF data in data mining and machine learning tasks. Existing approaches, such as RDF2Vec, use *local* information, i.e., they rely on local sequences generated for nodes in the RDF graph. For word embeddings, global techniques, such as *GloVe*, have been proposed as an alternative. In this paper, we show how the idea of global embeddings can be transferred to RDF embeddings, and show that the results are competitive with traditional local techniques like RDF2Vec.

Keywords: Graph embeddings · Linked open data · Data mining

1 Introduction

While RDF data is graph shaped by nature, most traditional data mining and machine learning software expect data to be in propositional form. Hence, to be used in machine learning and data mining pipelines, RDF data needs to be transformed to propositional feature vectors.

Recently, vector space embeddings have been proposed as a means to create low-dimensional feature vector representations of nodes in an RDF graphs. Inspired by techniques from NLP, such as word2vec [14], they train neural networks for automatically learning the mapping of RDF nodes to feature vectors. Vector space embeddings have been shown to outperform traditional methods for creating propositional feature vectors from RDF [22], e.g., in tasks like content-based recommender systems [24].

Unlike the first models for RDF vector space embeddings, which are based on paths, walks, or kernels, and therefore rely on *local* patterns, in this paper we present an approach in that exploits *global* patterns for creating vector space embeddings, inspired by the Global Vectors (GloVe) [20] approach for learning vector space embeddings for words from a text corpus. We show that using the GloVe approach on the same data as the older RDF2Vec approach does not improve the created embeddings. However, this approach is able to incorporate larger portions of the graph, without substantially increasing the computational

C. d'Amato et al. (Eds.): ISWC 2017, Part I, LNCS 10587, pp. 190–207, 2017.
DOI: 10.1007/978-3-319-68288-4_12

time, leading to comparable results. The main contributions of this paper are this new embedding approach and an approach to approximate all-pairs Personalized PageRank (PPR) computation, which is used to efficiently compute such embeddings.

The rest of this paper is structured as follows. Section 2 presents an overview on related work. In Sect. 3, we explain the basic idea of GloVe embeddings, and show how we transfer that idea to RDF graphs. Section 4 discusses an evaluation in different scenarios. We close with a summary and an outlook on future work.

The source code used in this evaluation can be found from https://github.com/miselico/globalRDFEmbeddingsISWC. Possible further developments will also be on http://users.jyu.fi/~miselico/software/.

2 Related Work

RDF vector space embeddings, i.e., projections of an RDF graph into a low-dimensional, dense vector space, have recently been proposed as a means to make RDF data accessible for propositional machine learning techniques, and shown to outperform traditional feature generation techniques [22].

RDF2Vec [22] is one of the first approaches that uses language modeling approaches for unsupervised feature extraction from sequences of words, and adapts them to RDF graphs. The approach generates sequences by leveraging local information from graph sub-structures, harvested by Weisfeiler-Lehman Subtree RDF Graph Kernels and graph walks, and then learns latent numerical representations of entities in RDF graphs.

The RDF2Vec approach is closely related to the approaches DeepWalk [21] and Deep Graph Kernels [31]. DeepWalk uses language modeling approaches to learn social representations of vertices of graphs by modeling short random-walks on large social graphs, like BlogCatalog, Flickr, and YouTube. The Deep Graph Kernel approach extends the DeepWalk approach, by modeling graph substructures, like graphlets, instead of graph walks. In this paper, we pursue and deepen the idea of random and biased walks since those have proven to be scalable even to large RDF graphs, unlike other transformation approaches, such as graph kernels. Node2vec [7] is another approach very similar to DeepWalk, which uses second order random walks to preserve the network neighborhood of the nodes.

Furthermore, multiple approaches for knowledge graph embeddings for the task of link prediction have been proposed [16], which could also be considered as approaches for generating propositional features from graphs. RESCAL [17] is one of the earliest approaches, which is based on factorization of a three-way tensor. The approach is later extended into Neural Tensor Networks (NTN) [28], which can be used for the same purpose (optionally using multilingual information [10]). One of the most successful approaches is the model based on translating embeddings, TransE [2]. This model builds entity and relation embeddings by regarding a relation as translation from head entity to tail entity. This approach assumes that relationships between words could be computed by their vector

difference in the embedding space. However, this approach cannot deal with reflexive, one-to-many, many-to-one, and many-to-many relations. This problem was resolved in the TransH model [30], which models a relation as a hyperplane together with a translation operation on it. More precisely, each relation is characterized by two vectors, the norm vector of the hyperplane, and the translation vector on the hyperplane. While both TransE and TransH, embed the relations and the entities in the same semantic space, the TransR model [13] builds entity and relation embeddings in separate entity space and multiple relation spaces. This approach is able to model entities that have multiple aspects, and various relations that focus on different aspects of entities.

Unlike the first models for RDF vector space embeddings, which are based on paths, walks, or kernels, and therefore rely on *local* patterns, the approach in this paper exploits *global* patterns for creating vector space embeddings, inspired by the Global Vectors (GloVe) [20] approach for learning vector space embeddings for words from a text corpus.

3 Global Vectors from RDF Data

The embedding method which we propose borrows the optimization problem and approach from GloVe [20]. Glove training, however, is based on the creation of a global co-occurrence matrix from text. Consequently, in our approach we need to devise a way to build a co-occurrence matrix from graph data. To this end, we first weigh the edges of the graph and compute approximate personalized PageRank scores starting from each node. The PageRank score for the other nodes (i.e., context nodes) is then used as the absolute frequency in a matrix. This procedure is then repeated on the graph with all edges reversed and the result is added to the co-occurrence matrix. This combined matrix is then subsequently used for training the vectors with the original Glove approach.

3.1 The GloVe Model

GloVe was designed for creating dense word vectors (also known as *word embeddings*) from natural language texts, which have been recently used with much success in a plethora of Natural Language Processing tasks. GloVe follows a distributional semantic view of word meaning in context, which basically relies on the assumption that 'words which are similar in meaning occur in similar contexts' [25] – i.e., meaning can be derived from the context (i.e., the surrounding words) of the word in a large corpus of text.

Consequently, to build a GloVe model a word-word co-occurrence matrix is first built, which contains for each word how often other words occur in its context. Model parameters then include the size of the context window, whether to distinguish left context from right context, as well as a weighting functions to weight the contribution of each word co-occurrence – e.g., a decreasing weighting function, where word pairs that are d words apart contribute $1/d$ to the total co-occurrence count.

After obtaining a co-occurrence matrix, GloVe attempts to minimize the following cost function using Adagrad [5].

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2 \quad (1)$$

where $f\left(X_{ij}\right)$ is a weighting function on co-occurrence counts of word j in the context of word i (X_{ij}), w_i are word vectors, $\tilde{w_j}$ context vectors and b_i and b_j biases. The intuition behind this cost function is the following one. Each summand of the summation represents the amount of error attributed to a count X_{ij} in the co-occurrence matrix. The error consists of a weighing function f, to dampen the effect of very large co-occurrence counts, and a squared error factor. The squared error factor will become smaller when the dot product of word vectors becomes closer to the logarithm of the probability that the words co-occur. Or turned the other way, when two words co-occur often, their vectors' dot product will be relatively high, meaning that the vectors are more similar to make the error factor smaller. The logarithm also causes that ratios of co-occurrence probabilities are associated with differences of vectors. As a result, the embedding contains information useful for determining analogies.

3.2 Building a Co-occurrence Matrix from Graph Data

The co-occurrence matrix for textual data is obtained by linearly scanning through the text and counting the occurrence of context words in the context of each word. However, the graph which we use as input data does not have a linear structure. This problem has been worked around in the past by performing random walks starting from each of the nodes in the graph. Recording the paths of these walks results in a linear sequence of node (and optionally edge) labels, which can then, in turn, be used as a pseudo-text to train a model. This approach is, for example, used in node2vec [7] and RDF2Vec [22]. However, in these approaches, the trained model is different from the GloVe model and it does not use the co-occurrence counts, but rather trains a neural network on the individual context windows directly. In the case of GloVe, only the counts are needed and hence we are looking for a method to obtain these without generating the random walks explicitly.

A possible solution would be to perform a breadth-first search of a certain depth starting from each node in turn, and take all reachable nodes as the context of each start node. Given these kinds of contexts, one could then straightforwardly apply GloVe's co-occurrence weighting and assign a lower weight to co-occurrence counts of nodes which are further away from the focus node. However, this simple approach is problematic in that: (a) there could be nodes reachable through multiple paths at different levels, (b) there could be loops in the graph, making a walk pass through the same node multiple times, and (c) if there is a node with many context nodes at level d, but only few ones at level $d-1$, then the ones at level d will dominate the closer ones in the co-occurrence matrix as there are that many of them.

To solve this problem, we investigate the use of Personalized PageRank [18] to determine how important nodes are in the context of a focus node. In general, PageRank is used to find important nodes in a directed graph. Its first, well-known use is the ranking of web pages, but later PageRank has also been applied in other areas (e.g., peer-to-peer networks [9] and social network analysis [15], among others). At its heart, PageRank works by simulating random walkers over the graph and observing where these random walkers end up. A simplified model which we will elaborate below would be as follows. First, we denote the out degree of a node i as $deg(i)$. Then, if there are n nodes in the graph, construct an $n \times n$ matrix P filled with zeros except for positions i, j, for which there exists an arc $i \rightarrow j$. These positions contain $1/deg(i)$. Now, the simplified page rank problem is solved by finding the stationary solution to (notation from [1] – $p^{(i)}$ is the vector converging to the PageRank value for each page after i iterations.)

$$p^{(k+1)} = P^T p^{(k)}. \tag{2}$$

This simplified version of PageRank can run into a number of problems, namely some pages may have a zero out degree (so called dangling nodes) and there could be groups of pages which form closed cycles. In the first case, PageRank (i.e., random walkers) will get lost from the graph and any node linking directly or indirectly to a zero out-degree node will get a PageRank of zero. In the second case walkers will get trapped and the pages in the cycles will accumulate all PageRank. To amend these problems, the above equation is adapted to include parts which ensure that when a walk ends up in a dangling node, it will continue from another node selected from a distribution v, called the teleportation distribution. Further, to avoid ending in a cycle, a random jump is also performed with probability α to a node selected from the same distribution. Usually, v is chosen to be a uniform distribution, making each node equally likely to be the target of the jump. However, in the case of personalized page rank the distribution is degenerate as the target of these random jumps is always the node for which the rank vector is computed (which we called the focus node). In effect, the Personalized PageRank vector indicated the importance of nodes from the perspective of the focus node.

Computing PageRank (and also the PPR variant) is reasonably scalable. However, as we need to compute PPR for each individual node in turn, in order to build the co-occurrence matrix, the rapidly becomes too expensive. Moreover, the PageRank algorithm assigns a value to all nodes in the graph. If we computed the co-occurrence matrix this way, we would end up with a very large (in our experiments below this would become around 500 TB) dense matrix with many small values, which have little to no impact on the later training. Hence, we designed a faster, approximate all-pairs PPR computation method, which results in a sparse matrix. This algorithm is based on an approximate PPR method which we will introduce next.

3.3 BCA: A Fast Personalized PageRank Approximation

A method for faster computation of Personalized PageRank, called Bookmark-Coloring Algorithm (BCA) was presented by Berkhin [1]. The main idea behind this method is to create an approximation to the standard PPR such that the effort of the algorithm is only used for these nodes which will receive a significant rank. This requires fewer computations and since nodes with no significant PageRank are not assigned a value, a sparse representation is obtained.

An intuitive version of the BCA algorithm is as follows (for full details, see [1]). To compute the PPR vector $p^{(b)}$ for a focus node b, we start by injecting a unit amount of paint, representing the walkers in the standard personal PageRank computation, to b. From this paint an α-portion is retained and added to the value for b in $p^{(b)}$. The remaining $(1-\alpha)$-portion is distributed uniformly over the out-links. This retain-and-distribute process is then repeated recursively for all nodes which got paint injected. When a node has a zero out degree, the outgoing paint is discarded.

This basic algorithm can be improved by choosing the order in which nodes considered for the retain-and-distribute. It is more efficient to select nodes with a larger amount of paint fist. To achieve this, a max priority queue, with the amount of paint as priorities is maintained. In principle, the queue could contain an entry for each node involved in each distribute step. However, it is more efficient to merge the separate wet paint amounts into one entry. Hence, the queue must allow efficient finding and updating of elements. Finally, when the amount of paint to be distributed becomes negligible (i.e., less than the parameter ϵ) it gets discarded, making the resulting rank vector sparse. All these improvements are described in more detail in the BCA paper [1]. One more technique described in the same paper is reuse of Bookmark-Coloring Vectors (BCV – the equivalent to the PageRank vector) for the computation of other BCVs. This is analyzed further for the case of hubs (i.e., nodes which correspond to a subset of important pages). The BCV is precomputed for these pages and whenever the retain-and-distribute process forwards paint to a page $p^{(h)}$ in the hub, the amount is multiplied with the BCV corresponding to $p^{(h)}$ and added to $p^{(b)}$. This optimization makes sense when many BCVs have to be computed, which is also the case for the co-occurrence matrix. However, since we are interested in computing the BCV for *all* nodes, further enhancements are possible, as we will discuss in the following subsection.

3.4 A Fast All-Pairs PPR Algorithm

The method introduced in the previous subsection speeds up the computation of individual PPR computations. Now, the observation leading to reuse of BCVs for pages in a hub can be adapted to our setting. The main point is that the computation of the BCV of node b can reuse the BCV of nodes reachable through its out links. Especially, it is beneficial if the BCV of nodes one hop away have already been computed. Adopting this viewpoint, we say that computation of the BCV of the node b *depends* on the BCV computation of all one-hop reachable

Algorithm 1. Determining the BCV Computation Order

```
function BCVOrder(Graph & G_original)
    G ← G_original                    ▷ Copied because G will be modified in the function
    Initialize list Order             ▷ The list with the node ordering
    Initialize max priority queue Q_indeg   ▷ The nodes in ascending in-degree
    Add all nodes to Q_indeg
    repeat
        while G has a node n with out-degree 0 do
            Add n to Order, Remove n from G, Remove n from Q_indeg
        end while
        if G is not empty then        ▷ There is a cycle which needs to be broken
            n ← Q_indeg.pop()
            Add n to Order, Remove n from G
            for all d dependent on n do
                Update priority of d in Q_indeg
            end for
        end if
    until G is empty
    return Order
end function
```

nodes and hence b is a *dependent* of these nodes. Now, what we want to achieve is that we only compute the BCV for nodes once the BCVs of all its dependent nodes have been computed. However, this will not always be feasible as the graphs contains cycles. Hence, we want to quickly find an ordering of nodes, such that we can likely reuse as many BCV computations as possible. To achieve this we break cycles and in that case compute the BCV for the node at which we break without being able to count on all dependents being available. We choose the node for breaking the cycle to be the one with the highest in-degree as that one is likely to cause most reuse and break multiple cycles at once. The pseudocode of the Algorithm is shown in Algorithm 1, the actual implementation also includes a couple of indexes and bitmaps to speed up the computation. Now, with the order determined, we can compute each BCV, reusing many previously computed values.

3.5 Biasing the Random Walks

The default PageRank and BCA algorithm assume that a random walker will follow the out edges of a node with equal likelihood. However, one can also create a setup in which given out edges are more likely than others. For BCA, this possibility was already hinted in the original paper [1], but not elaborated much further. This so called biasing can be accomplished by taking into account the out edge weights when distributing the paint over them.

Following our previous work [3], we apply twelve different strategies for assigning these weights to the edges of the graph. These weights will then in turn bias the random walks on the graph. In particular, when a walk arrives in

a vertex v with out edges $v_{o1}, \ldots v_{od}$, then the walk will follow edge v_{ol} with a probability computed by

$$\Pr[\text{follow edge } v_{ol}] = \frac{weight(v_{ol})}{\sum_{i=1}^{d} weight(v_{oi})}$$

In other words, the normalized edge weights are directly interpreted as the probability to follow a particular edge. To obtain these edge weights, we make use of the following statistics computed from the RDF data:

Predicate Frequency for each predicate in the dataset, we count the number of times the predicate occurs (only occurrences as a predicate are counted).

Object Frequency for each resource in the dataset, we count the number of times it occurs as the object of a triple.

Predicate-Object frequency for each pair of a predicate and an object in the dataset, we count the number of times there is a statement with this predicate and object.

Besides these statistics, we also use PageRank [18] computed for the entities in the knowledge graph [29]. This PageRank is computed based on links between the Wikipedia articles representing the respective entities. When using the PageRank computed for DBpedia, not each node has a value assigned, as only entities which have a corresponding Wikipedia page are accounted for in the PageRank computation. Examples of nodes which do not have a PageRank include DBpedia types or categories, like http://dbpedia.org/ontology/Place and http://dbpedia.org/resource/Category:Central_Europe. Therefore, we assigned a fixed PageRank to all nodes which are not entities. We chose a value of 0.2, which is roughly the median PageRank in the non-normalized page rank values we used.

We have essentially two types of metrics, those assigned to nodes, and those assigned to edges. The predicate frequency and predicate-object frequency, as well as the inverses of these, can be directly used as weights for edges. Therefore, we call these weighting methods *edge-centric*. In the case of predicate frequency each predicate edge with that label is assigned the weight in question. In the case of predicate-object frequency, each predicate edge which ends in a given object gets assigned the predicate-object frequency. We also use the inverse metrics, where not the absolute frequency is assigned, but its multiplicative inverse.

In contrast, the object frequency, and also the used PageRank metric, assign a numeric score to each node in the graph. Therefore, we call weighting approaches based on them *node-centric*. To obtain a weight for the edges, we either *push* the weight down, meaning that the number assigned to a node is used as the weight of all in edges, or we *split* the number down, meaning that the weight is divided by the number of in edges and then assigned to all these edges. If *split* is not mentioned explicitly in node centric weighting strategies, then it is a push down strategy.

Note that uniform weights are equivalent to using object frequency with splitting the weights. To see why this holds true, we have to follow the steps

which will be taken. First, each node gets assigned the amount of times it is used as an object. This number is equal to the number of in edges to the node. Then, this number is split over the in edges, i.e., each in edge gets assigned the number 1. Finally, this weight is normalized, assigning to each out link a uniform weight. Hence, this strategy would result in the same walks as using unbiased random walks over the graph.

So, even if we add unbiased random walks to the list of weighting strategies, we retain 12 unique ones, each with their own characteristics. These strategies, which we further elaborated upon in our earlier work [3], are:

Uniform approach:

1. *Uniform = Object Frequency Split*

Edge-centric approaches:

2. *Predicate Frequency*
3. *Inverse Predicate Frequency*
4. *Predicate-Object Frequency*
5. *Inverse Predicate-Object Frequency*

Node-centric object freq. approaches (See also strategy 1):

6. *Object Frequency*
7. *Inverse Object Frequency*
8. *Inverse Object Frequency Split*

Node-centric PageRank approaches:

9. *PageRank*
10. *Inverse PageRank*
11. *PageRank Split*
12. *Inverse PageRank Split*

3.6 Combining the Pieces

In earlier work on RDF graph embeddings (specifically RDF2Vec [22]), symmetric windows were used on top of generated random walks, which include both node and edge labels. These symmetric windows have the focus word in the middle and the context of the word is both before and after it. This means that the context of a node b consists of the nodes it can reach by following edges, as well as the nodes which can reach b. What this means is that the result RDF2Vec would be the same, independently of whether the original walks would be performed forward or backward. Inspired by this, we investigated the effect of creating the co-occurence matrix as the sum of the normal PPR matrix as described above and the PPR matrix of the graph with all edges reversed. Since a positive effect on the embeddings was obtained (at least for the tasks we used in the evaluation) we chose to use this approach.

RDF2Vec also includes edge labels into the walks and the embedding procedure. We also noticed a positive effect including the edge labels whenever they are traversed by paint with a weight equal to the amount of paint. Because the summation and additions of the label weights might lead to a skew in the values, we normalize each BCV in the co-occurence matrix by removing the value on the diagonal and scaling the remaining values such that their sum is 1. This operation led to improvements in the results and hence we adopted this technique for the overall algorithm. The pseudo code of the Global RDF Vector Space Embedding algorithm can be found in Algorithm 2.

Algorithm 2. Global RDF Vector Space Embedding

```
function CreateEmbeddings(Graph & G, Weighting Strategy W)
    Weigh G according to W
    Order ← BCVOrder(G)
    Compute all BCV according to Order, reusing results
    G_r ← ReverseEdges(G)
    Weigh G_r according to W
    ReverseOrder ← BCVOrder(G_r)
    Compute all BCV according to ReverseOrder, reusing results
    Sum the BCVs obtained for the normal and reversed graph and normalize,
            forming the co-occurrence matrix.
    Execute Glove training for the co-occurrence matrix.
    return The resulting vectors
end function
```

The overall algorithm has several parameters. First, there is the weighting strategy; the options are described above. Second, there are the parameters α and ϵ for the BCA algorithm. We chose the α parameter to be 0.1 and $\epsilon = 0.00001$, which is within the ranges stated by Berkhin [1]. Third, there are the parameters for the GloVe training. There is the vector length, which we choose to be 200, which is in the middle of the sizes used in the original Glove experiments [20]. We use 20 training iterations as we noticed that more iterations did not significantly decrease the cost function. We used the default values for the Adagrad learning rate and damp function.

4 Evaluation

First, we evaluate the different weighting strategies on a number of classification and regression tasks, comparing the results of different feature extraction strategies combined with different learning algorithms. Second, we evaluate the weighting strategies on the task of computing document similarity. We evaluate our approach using DBpedia [12]. We use the English version of the 2016-04 DBpedia dataset, which contains 4,678,230 instances and 1,379 mapping-based properties. In our evaluation we only consider object properties, and ignore literals. All the experiments were run using a Linux machine using at most 300 GB RAM and 24 Intel Xeon 2.60 GHz CPUs. For all the weighing strategies the processes took between 6 h for the least demanding strategy, the Predicate Frequency strategy, and up to 48 h for the most demanding strategy, the Predicate-Object Frequency. The runtime for building the related work approaches, using the publicly available code,[1] was more than a week.

[1] https://github.com/thunlp/KB2E.

Table 1. Classification results. The best results for each dataset are marked in bold.

Strategy/Dataset	Cities				Metacritic movies				Metacritic albums				AAUP				Forbes			
	NB	KNN	SVM	C4.5	NB	KNN	SVM	C4.5	NB	KNN	SVM	C4.5	NB	KNN	SVM	C4.5	NB	KNN	SVM	C4.5
Uniform	57.32	63.89	67.47	58.32	68.41	68.66	70.65	66.11	62.05	60.44	64.12	58.68	83.64	89.42	29.54	89.98	94.08	79.74	74.51	94.64
Predicate frequency	60.00	59.32	66.39	54.79	58.31	56.22	58.92	58.06	61.73	58.30	61.40	59.27	83.65	89.42	27.75	88.68	91.93	79.74	74.51	94.37
Inverse predicate frequency	49.08	52.16	53.05	41.97	66.32	66.62	69.73	61.63	64.90	62.56	64.52	59.33	83.21	89.42	29.84	89.00	92.48	79.74	74.51	93.70
Predicate object frequency	57.39	61.84	67.89	52.79	64.28	62.85	64.73	64.12	58.11	56.49	60.17	56.68	83.65	89.42	29.51	90.97	93.14	79.74	74.51	93.83
Inv. predicate object freq.	54.37	63.47	60.26	47.53	62.50	65.55	67.34	61.88	61.27	64.38	62.83	59.14	82.45	89.42	29.39	89.87	93.28	79.74	74.51	93.96
Object frequency	61.89	56.32	68.42	46.16	65.65	62.85	65.04	63.97	57.72	55.91	59.20	59.59	84.08	89.42	29.42	90.96	92.87	79.74	74.51	93.43
Inverse object frequency	53.87	56.26	60.76	47.11	62.49	65.50	68.10	63.15	59.01	62.12	63.86	58.43	82.45	89.42	29.03	89.65	93.28	79.74	74.51	94.90
Inverse object freq. split	56.79	54.29	56.26	50.21	60.76	61.27	63.77	61.32	61.35	60.70	61.86	61.73	82.45	89.42	29.60	89.76	93.28	79.74	74.51	93.96
PageRank	63.37	64.95	66.89	59.34	73.56	78.26	77.79	75.09	76.39	78.20	79.69	71.66	81.58	89.42	29.91	93.31	93.93	79.74	74.51	**95.78**
Inverse PageRank	53.29	55.13	69.69	51.61	80.09	80.44	79.69	76.78	71.66	72.24	79.69	66.68	84.30	89.42	29.46	93.21	92.22	80.29	64.69	94.09
PageRank split	54.79	57.71	69.90	51.76	78.66	81.01	79.56	76.67	75.09	72.31	**80.99**	69.53	82.01	89.42	29.39	89.54	90.88	80.29	75.65	93.28
Inverse PageRank split	50.66	54.21	66.99	49.71	71.68	72.13	74.64	71.32	69.85	70.76	72.05	67.78	82.78	89.42	30.70	93.09	93.02	79.74	74.51	94.91
RDF2VecGloVe	64.84	48.18	67.26	53.34	64.25	67.20	69.61	62.69	63.75	66.10	65.21	59.13	73.89	85.66	27.49	92.45	86.98	81.07	74.92	95.42
Best baseline	72.71	60.00	71.70	75.29	78.50	66.90	79.30	70.80	74.25	64.69	77.94	64.50	63.44	91.04	93.44	92.81	67.09	76.49	76.97	76.47
DB_TransE	65.79	75.71	74.63	61.50	65.75	64.17	68.96	61.16	62.81	60.48	64.17	56.86	80.28	84.86	28.95	89.65	92.88	79.98	74.37	95.44
DB_TransH	64.39	72.66	76.66	60.89	63.51	63.25	67.43	60.96	63.97	63.13	65.07	60.23	80.39	84.86	27.55	89.21	93.82	79.98	74.37	93.68
DB_TransR	63.08	67.32	74.50	59.84	64.38	60.16	64.43	52.04	63.56	59.68	66.41	60.39	79.19	84.86	28.95	89.00	93.28	79.98	74.37	93.70
Best RDF2Vec	**89.73**	69.16	84.19	72.25	80.24	78.68	**82.80**	72.42	73.57	76.30	78.20	68.70	75.07	**94.48**	29.11	94.15	88.53	80.58	77.79	86.38

4.1 Machine Learning Tasks

We use the DBpedia entity embeddings on five different datasets from different domains, for the tasks of classification and regression, i.e., *Cities*[2], *Metacritic Movies*[3], *Metacritic Albums*[4], *AAUP*[5] and *Forbes*[6]. Details on the dataset can be found in [23]. We follow the same experimental setup as in our RDF2Vec paper [22], using Naive Bayes, k-Nearest Neighbors, C4.5, and Support Vector Machine for classification, and Linear Regression, M5Rules, and k-Nearest Neighbors for regression, measuring accuracy and root mean squared error (RMSE) in stratified 10-fold cross validation. The results on parameter settings for the algorithms can be found in [22].

Furthermore, from our original RDF2Vec paper [22], we report the *best baseline* and the *best RDF2Vec* performance. As an additional baseline, we use the same set of random walks used in [22] to build a simple GloVe model, and report the results under *RDF2VecGloVe*. Furthermore, we compare our results to the embedding approaches TransE, TransH, and TransR, which have shown to be scalable to large knowledge graphs.

Tables 1 and 3 depict the results for the classification and regression task. We determine the significance in ranking of the approaches using the approach introduced by Demšar [4], as discussed in [22]. The results are depicted in Tables 2 and 4.

We can observe that although RDF2Vec is a very strong competitor, the approach introduced in this paper is capable of producing embeddings which outperform the results achieved with RDF2Vec in specific cases. In particular for classification algorithms which yield inferior results with RDF2Vec. It is also remarkable that TransE, TransH, and TransR are often outperformed by the baselines. Furthermore, we can observe that a naive application of the GloVe approach to walks (RDF2VecGloVe) does not lead to convincing results.

4.2 Document Modeling

Calculating entity similarity lies at the heart of knowledge-rich approaches to computing semantic similarity, a fundamental task in Natural Language Processing and Information Retrieval [32]. As previously mentioned, in the feature embedding space semantically similar entities appear close to each other in the feature space. Therefore, the problem of calculating the similarity between two instances is a matter of calculating the distance between two instances in the given feature space. To do so, we use the standard cosine similarity measure, which is applied on the vectors of the entities.

[2] https://www.imercer.com/content/mobility/quality-of-living-city-rankings.html.
[3] http://www.metacritic.com/browse/movies/score/metascore/all.
[4] http://www.metacritic.com/browse/albums/score/metascore/all.
[5] http://www.amstat.org/publications/jse/jse_data_archive.htm.
[6] http://www.forbes.com/global2000/list/.

Table 2. Classification average rank results. The best ranked results for each method are marked in bold. The learning models for which the strategies were shown to have significant difference based on the Friedman test with $\alpha < 0.05$ are marked with *. The single values marked with $*$ mean that are significantly worse than the best strategy at significance level $q = 0.05$

Method	NB	KNN*	SVM*	C4.5*
Uniform weight	7.2	9.4	8.9	8.8
Predicate frequency weight	11.5	13	14.7	12.4
Inverse predicate frequency weight	10.2	11.4	9.9	14.2*
Predicate object frequency weight	10.1	12.1	11.5	10.8
Inverse predicate object frequency weight	11.7	9.2	12.6	11.9
Object frequency weight	9.8	13.1	11.7	11.8
Inverse object frequency weight	12.3	11.2	12.3	11.9
Inverse object frequency split weight	11.5	12.8	12.7	10.9
PageRank weight	**5.2**	6.2	6.6	**2.6**
Inverse PageRank weight	7.4	6.4	7.5	5.8
PageRank split weight	8.8	5.4	4.7	9
Inverse PageRank split weight	9	9.4	7.1	6.2
RDF2VecGloVe	12	9.4	10	8.6
Best baseline	9	8.8	**3.4**	7.2
DB_TransE	9.4	9.8	10.9	9.9
DB_TransH	8.6	9.4	11.2	11.6
DB_TransR	9.7	11.8	11.5	12
Best RDF2Vec	7.6	**2.2**	3.8	5.4

We use the entity similarity approach in the task of calculating semantic document similarity. We follow an approach similar to the one presented in [19], where two documents are considered to be similar if many entities of the one document are similar to at least one entity in the other document. More precisely, we try to identify the most similar pairs of entities in both documents, ignoring the similarity of all the other 1–1 similarities values. The similarity of two documents is then defined as the average maximum similarity for all entities in each document (see [3]).

We evaluate performance on document similarity approach using the LP50 dataset [11]. We follow standard practices and use Pearson's linear correlation coefficient and Spearman's rank correlation plus their harmonic mean as evaluation metrics. In addition to the baselines introduced above, we compare our approach to the following approaches:

Table 3. Regression results. The best results for each dataset are marked in bold.

Strategy/Dataset	Cities			Metacritic Movies			Metacritic Albums			AAUP			Forbes		
	LR	KNN	M5	LR	KNN	M5	LR	KNN	M5	LR	KNN	M5	LR	KNN	M5
Uniform	18.41	18.02	11.20	16.32	23.31	20.50	11.96	13.19	13.35	6.35	57.10	6.45	19.27	20.85	18.01
Predicate frequency	16.10	17.31	19.68	17.63	21.75	18.14	10.98	15.72	13.73	6.36	57.10	6.41	17.58	19.33	17.89
Inverse predicate frequency	21.71	16.42	14.40	20.69	21.88	18.44	12.67	13.44	12.94	6.35	57.10	6.37	18.93	20.58	18.04
Predicate object frequency	16.02	15.78	14.31	20.77	24.33	19.42	12.54	14.00	12.32	6.30	57.10	6.37	19.14	19.50	17.59
Inverse predicate object frequency	14.52	14.24	19.50	18.03	22.62	17.60	**10.79**	13.47	11.28	6.30	57.10	6.36	18.91	19.14	19.25
Object frequency	11.74	16.49	16.77	20.84	22.63	17.83	12.51	14.23	12.12	6.34	57.10	6.41	18.07	20.38	18.92
Inverse object frequency	15.87	18.31	15.40	18.49	22.00	18.49	13.37	14.60	13.38	6.37	57.10	6.44	17.86	19.07	17.00
Inverse object frequency split	15.96	14.01	20.52	21.40	23.32	18.94	11.61	13.20	12.63	6.40	57.10	6.43	19.62	20.31	19.87
PageRank	17.61	**9.50**	14.43	18.08	19.75	19.20	12.56	14.31	12.48	**6.**28	57.10	6.32	18.98	19.40	**16.27**
Inverse PageRank	13.41	13.33	10.47	17.91	20.52	16.63	13.17	13.73	12.72	6.37	57.10	6.46	18.79	18.99	18.93
PageRank split	19.70	20.51	12.44	17.22	19.86	18.84	12.58	12.46	10.93	6.31	57.10	6.32	17.61	20.90	19.22
Inverse PageRank split	17.22	18.63	12.65	17.76	23.09	19.82	12.04	14.17	11.90	6.36	57.10	6.39	17.42	18.93	20.10
RDF2VecGloVe	20.50	20.24	20.57	23.10	26.37	23.04	13.87	15.74	13.93	6.34	57.31	6.37	20.45	21.55	19.18
Best baseline	17.79	18.21	17.04	21.45	21.62	19.19	13.32	13.99	12.81	8.08	34.94	6.36	19.16	19.81	18.20
DB_TransE	14.22	14.45	14.46	20.66	23.61	20.71	13.20	14.71	13.23	6.34	57.27	6.43	20.00	21.55	17.73
DB_TransH	13.88	12.81	14.28	20.71	23.59	20.72	13.04	14.19	13.03	6.35	57.27	6.47	19.88	21.54	16.66
DB_TransR	14.50	13.24	14.57	20.10	23.37	20.04	13.87	15.74	13.93	6.34	57.31	6.37	20.45	21.55	17.18
Best RDF2Vec	11.92	12.67	10.19	**15.45**	17.80	15.50	10.89	11.72	10.97	**6.26**	56.95	6.29	18.35	21.04	16.61

Table 4. Regression average rank results. The best ranked results for each method are marked in bold. The learning models for which the strategies were shown to have significant difference based on the Friedman test with $\alpha < 0.05$ are marked with *. The single values marked with $*$ mean that are significantly worse than the best strategy at significance level $q = 0.05$

Method	LR*	KNN*	M5*
Uniform weight	9.2	9.7	11.4
Predicate frequency weight	6.7	9.5	11.3
Inverse predicate frequency weight	12.2	8.3	8.5
Predicate object frequency weight	9.3	10.1	7.7
Inverse predicate object frequency weight	5.3	6.9	8.4
Object frequency weight	7.1	10.3	9.1
Inverse object frequency weight	10.5	9.7	10.6
Inverse object frequency split weight	12	8.1	12.9
PageRank weight	8.4	6.1	6.3
Inverse PageRank weight	8.9	5.3	8.6
PageRank split weight	7.4	8.9	6.3
Inverse PageRank split weight	7.5	9.3	10
RDF2VecGloVe	15.5*	17.5*	15*
Best baseline	15.2*	7.2	9.2
DB_TransE	10.7	14.1	11.9
DB_TransH	11	11.9	11.2
DB_TransR	11.7	14.1	11
Best RDF2Vec	**2.4**	**4**	**1.6**

- TF-IDF: Distributional baseline algorithm.
- AnnOv: Similarity score based on annotation overlap that corresponds to traversal entity similarity with radius 0, as described in [19].
- Explicit Semantic Analysis (ESA) [6].
- GED: semantic similarity using a Graph Edit Distance based measure [27].
- Salient Semantic Analysis (SSA), Latent Semantic Analysis (LSA) [8].
- Graph-based Semantic Similarity (GBSS) [19].

The results for the related approaches were taken from the respective papers, except for ESA, which was taken from [19], where it is calculated via the public ESA REST endpoint[7]. All results are collected in Table 5. We can see that our approach, using inverse predicate object frequency weights, outperforms the state-of-the-art approaches, as well as the embeddings generated by RDF2Vec.

[7] http://vmdeb20.deri.ie:8890/esaservice.

Table 5. Document similarity results - Pearson's linear correlation coefficient (r) Spearman's rank correlation (ρ) and their harmonic mean μ

Approach	r	ρ	μ
Uniform weight	0.537	0.535	0.536
Predicate frequency weight	0.534	0.532	0.533
Inverse predicate frequency weight	0.632	**0.621**	**0.627**
Predicate object frequency weight	0.331	0.323	0.327
Inverse predicate object frequency weight	0.541	0.544	0.542
Object frequency weight	0.346	0.348	0.347
Inverse object frequency weight	0.523	0.547	0.534
Inverse object frequency split weight	0.504	0.513	0.509
PageRank weight	0.488	0.485	0.486
Inverse PageRank weight	0.429	0.481	0.454
PageRank split weight	0.539	0.528	0.533
Inverse PageRank split weight	0.512	0.511	0.512
RDF2VecGloVe	0.569	0.432	0.491
Best RDF2Vec	**0.708**	0.556	0.623
DB_TransE	0.565	0.432	0.490
DB_TransH	0.570	0.452	0.504
DB_TransR	0.578	0.461	0.513
TF-IDF	0.398	0.224	0.287
AnnOv	0.590	0.460	0.517
LSA	0.696	0.463	0.556
SSA	0.684	0.488	0.570
GED	0.630	\	\
ESA	0.656	0.510	0.574
GBSS	0.704	0.519	0.598

5 Conclusion and Outlook

In this paper, we have introduced a novel approach for generating embeddings of RDF graphs, which exploits global instead of local patterns. We have shown that it is possible to outperform local graph embeddings techniques, in particular on document similarity. For most other tasks similar performance can be obtained.

One key finding of this work is that weighting techniques are a crucial factor in the overall performance. In the future, we would like to investigate this point more thoroughly, and analyze the interplay of the dataset, the task, the learning algorithm, and the weighting technique more formally and with more exhaustive experimentation. One way to achieve this is by evaluating the embedding using intrinsic measures such as those suggested in [26]. Besides, we would

like to further investigate how the literals in the dataset can be incorporated while learning the embedding. Furthermore, as GloVe embeddings are known to work particularly well for finding analogies, we plan to adapt the approach for predicting missing links in RDF data sets.

Acknowledgements. The work presented in this paper has been partially funded by the Junior-professor funding programme of the Ministry of Science, Research and the Arts of the state of Baden-Württemberg (project "Deep semantic models for high-end NLP application"), and by the German Research Foundation (DFG) under grant number PA 2373/1-1 (Mine@LOD).

References

1. Berkhin, P.: Bookmark-coloring algorithm for personalized pagerank computing. Internet Math. **3**(1), 41–62 (2006)
2. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. In: NIPS, pp. 2787–2795 (2013)
3. Cochez, M., Ponzetto, S.P., Paulheim, H.: Biased graph walks for RDF graph embeddings. In: WIMS (2017)
4. Demšar, J.: Statistical comparisons of classifiers over multiple datasets. J. Mach. Learn. Res. **7**, 1–30 (2006)
5. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)
6. Gabrilovich, E., Markovitch, S.: Computing semantic relatedness using Wikipedia-based explicit semantic analysis. In: IJCAI, pp. 1606–1611 (2007)
7. Grover, A., Leskovec, J.: Node2vec: scalable feature learning for networks. In: KDD, pp. 855–864 (2016)
8. Hassan, S., Mihalcea, R.: Semantic relatedness using salient semantic analysis. In: AAAI, pp. 884–889 (2011)
9. Kamvar, S.D., Schlosser, M.T., Garcia-Molina, H.: The Eigentrust algorithm for reputation management in P2P networks. In: WWW, pp. 640–651 (2003)
10. Klein, P., Ponzetto, S.P., Glavaš, G.: Improving neural knowledge base completion with cross-lingual projections. In: EACL, vol. 2, pp. 516–522 (2017)
11. Lee, M., Pincombe, B., Welsh, M.: An empirical evaluation of models of text document similarity. Cogn. Sci. Soc. **27**, 1254–1259 (2005)
12. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., Bizer, C.: DBpedia - A large-scale. Multilingual knowledge base extracted from Wikipedia. Semant. Web J. **6**, 167–195 (2013)
13. Lin, Y., Liu, Z., Sun, M., Liu, Y., Zhu, X.: Learning entity and relation embeddings for knowledge graph completion. In: AAAI, pp. 2181–2187 (2015)
14. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781 (2013)
15. Mislove, A., Marcon, M., Gummadi, K.P., Druschel, P., Bhattacharjee, B.: Measurement and analysis of online social networks. In: IMC, pp. 29–42 (2007)
16. Nickel, M., Murphy, K., Tresp, V., Gabrilovich, E.: A review of relational machine learning for knowledge graphs. Proc. IEEE **104**(1), 11–33 (2016)
17. Nickel, M., Tresp, V., Kriegel, H.P.: A three-way model for collective learning on multi-relational data. In: ICML, pp. 809–816 (2011)

18. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. Technical report 1999–66, Stanford InfoLab, November 1999
19. Paul, C., Rettinger, A., Mogadala, A., Knoblock, C.A., Szekely, P.: Efficient graph-based document similarity. In: Sack, H., Blomqvist, E., d'Aquin, M., Ghidini, C., Ponzetto, S.P., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9678, pp. 334–349. Springer, Cham (2016). doi:10.1007/978-3-319-34129-3_21
20. Pennington, J., Socher, R., Manning, C.D.: Glove: global vectors for word representation. In: EMNLP 2014, pp. 1532–1543 (2014)
21. Perozzi, B., Al-Rfou, R., Skiena, S.: DeepWalk: online learning of social representations. In: KDD, pp. 701–710 (2014)
22. Ristoski, P., Paulheim, H.: RDF2Vec: RDF graph embeddings for data mining. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 498–514. Springer, Cham (2016). doi:10.1007/978-3-319-46523-4_30
23. Ristoski, P., de Vries, G.K.D., Paulheim, H.: A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In: Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., Gil, Y. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 186–194. Springer, Cham (2016). doi:10.1007/978-3-319-46547-0_20
24. Rosati, J., Ristoski, P., Di Noia, T., Leone, R.D., Paulheim, H.: RDF graph embeddings for content-based recommender systems. In: CEUR Workshop Proceedings, vol. 1673, pp. 23–30. RWTH (2016)
25. Rubenstein, H., Goodenough, J.B.: Contextual correlates of synonymy. Commun. ACM **8**(10), 627–633 (1965)
26. Schnabel, T., Labutov, I., Mimno, D.M., Joachims, T.: Evaluation methods for unsupervised word embeddings. In: EMNLP, pp. 298–307 (2015)
27. Schuhmacher, M., Ponzetto, S.P.: Knowledge-based graph document modeling. In: WSDM, pp. 543–552 (2014)
28. Socher, R., Chen, D., Manning, C.D., Ng, A.: Reasoning with neural tensor networks for knowledge base completion. In: NIPS, pp. 926–934 (2013)
29. Thalhammer, A., Rettinger, A.: PageRank on Wikipedia: towards general importance scores for entities. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 227–240. Springer, Cham (2016). doi:10.1007/978-3-319-47602-5_41
30. Wang, Z., Zhang, J., Feng, J., Chen, Z.: Knowledge graph embedding by translating on hyperplanes. In: AAAI, pp. 1112–1119 (2014)
31. Yanardag, P., Vishwanathan, S.: Deep graph kernels. In: KDD, pp. 1365–1374 (2015)
32. Zhang, Z., Gentile, A.L., Ciravegna, F.: Recent advances in methods of lexical semantic relatedness - a survey. Nat. Lang. Eng. **19**(4), 411–479 (2013)