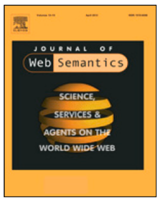
Web Semantics: Science, Services and Agents on the World Wide Web 75 (2023) 100741

Contents lists available at ScienceDirect

Web Semantics: Science, Services and Agents on the World Wide Web

journal homepage: www.elsevier.com/locate/websem



Towards the Web of Embeddings: Integrating multiple knowledge graph embedding spaces with FedCoder

Matthias Baumgartner[a,*], Daniele Dell'Aglio[b,a], Heiko Paulheim[c], Abraham Bernstein[a]

[a] *Department of Informatics, University of Zurich, Zurich, Switzerland*
[b] *Department of Computer Science, Aalborg University, Aalborg, Denmark*
[c] *Data and Web Science Group, University of Mannheim, Mannheim, Germany*

ARTICLE INFO

Article history:
Received 1 April 2022
Received in revised form 22 July 2022
Accepted 2 August 2022
Available online 8 August 2022

Keywords:
Knowledge graphs
Knowledge graph embedding
Embedding space integration
Web of Embeddings

ABSTRACT

The Semantic Web is distributed yet interoperable: Distributed since resources are created and published by a variety of producers, tailored to their specific needs and knowledge; Interoperable as entities are linked across resources, allowing to use resources from different providers in concord. Complementary to the explicit usage of Semantic Web resources, embedding methods made them applicable to machine learning tasks. Subsequently, embedding models for numerous tasks and structures have been developed, and embedding spaces for various resources have been published. The ecosystem of embedding spaces is distributed but not interoperable: Entity embeddings are not readily comparable across different spaces. To parallel the Web of Data with a Web of Embeddings, we must thus integrate available embedding spaces into a uniform space.

Current integration approaches are limited to two spaces and presume that both of them were embedded with the same method — both assumptions are unlikely to hold in the context of a Web of Embeddings. In this paper, we present FedCoder— an approach that integrates multiple embedding spaces via a latent space. We assert that linked entities have a similar representation in the latent space so that entities become comparable across embedding spaces. FedCoder employs an autoencoder to learn this latent space from linked as well as non-linked entities.

Our experiments show that FedCoder substantially outperforms state-of-the-art approaches when faced with different embedding models, that it scales better than previous methods in the number of embedding spaces, and that it improves with more graphs being integrated whilst performing comparably with current approaches that assumed joint learning of the embeddings and were, usually, limited to two sources. Our results demonstrate that FedCoder is well adapted to integrate the distributed, diverse, and large ecosystem of embeddings spaces into an interoperable Web of Embeddings.

Any problem in computer science can be solved with another level of indirection attributed to David Wheeler among others[1]

1. Introduction

Semantic Web research has not only had a major impact on the development and adoption of Knowledge Graphs (KGs) but also on making them interoperable. With the endorsement of Linked Open Data (LOD), a wide and diverse spectrum of graphs can nowadays be used in unison by having entities that represent the same real-world concept linked across KGs. In recent years, embedding methods furthered the versatility of KGs by making them accessible to machine learning methods. These methods learn a numerical representation of entities (and possibly relations) that reflect structural patterns in the graph and are typically used as a feature vector in downstream classification or regression applications. A variety of paradigms has been explored since, resulting in KG embedding models for different tasks or specific KG structures [1,2]. A knowledge graph embedding typically only provides representations for the entities in that KG. Hence its usability is limited to only those applications that agree with its set of entities. Since the Semantic Web, by definition, cannot have one graph that covers all possible entities, we envision an embedding space that spans all (or at least large parts) of the LOD cloud [3].

* Corresponding author.
E-mail addresses: baumgartner@ifi.uzh.ch (M. Baumgartner), dade@cs.aau.dk (D. Dell'Aglio), heiko@informatik.uni-mannheim.de (H. Paulheim), bernstein@ifi.uzh.ch (A. Bernstein).

[1] See https://en.wikipedia.org/wiki/Butler_Lampson#Quotes.

It is tempting to embed the whole LOD cloud directly. However, there are a number of conceptual, organizational, and technical challenges that render such an endeavor impractical. First and foremost, such an approach would require centralization in an otherwise purposely decentralized KG ecosystem. Second, it would require a single institution to finance an infrastructure that is able to collect, store, and embed billions of triples. Moreover, since a KG embedding is tied to the underlying graph, the substantial effort and cost of embedding incur again whenever any of the graphs, any alignment between graphs, or the composition of the LOD cloud changes. Third, with the diversity of graphs in the LOD cloud, it is challenging (or even impossible) to find a suitable one-size-fits-all embedding model, and searching for one would increase the cost of embedding further. One could potentially use joint embedding techniques to relax this last issue, however these approaches have so far only been studied for two KGs [4–7], and the LOD cloud is continuously growing, far exceeding this number.[2]

Given the impracticality of one big embedding space, the question arises: *how can we combine multiple KG embedding spaces into a uniform Web of Embeddings*? Embeddings for a variety of KGs have already been published by different providers, such as KGVec2Go,[3] PyTorch-BigGraph,[4] LibKGE,[5] and others. Each of them uses different embedding models and hyperparameters, adapted to the respective graphs and use-cases they intend to cover [8–10].

This situation reflects the distributed nature of the Semantic Web, where different providers share data about the same entities in different contexts. The challenge of building the Web of Embeddings is that entity embeddings are not readily comparable across KG embedding spaces, i.e., entities linked with the *owl:sameAs* relation do not have the same embedding vector despite representing the same real-world concept.

The Web of Embeddings mirrors the core idea of the Semantic Web that data is distributed yet interlinked: Each KG provider can choose the embedding method and hyperparameters that best fit their data, yet an application can combine embedding spaces from multiple graphs and use them in a uniform manner. Analogously to the Semantic Web, we must bridge multiple embedding spaces to solve this problem.

To integrate two KG embedding spaces, previous approaches learned a mapping function from one space to the other, given some linked entities as anchors [4–7,11,12]. However, such a pairwise mapping becomes inefficient for more than a few graphs. To address the challenge of a Web of Embeddings, we present FedCoder (Fig. 1), a method that learns a latent space from multiple given embedding spaces by finding mappings from each KG embedding into the latent space and vice-versa. So akin to the original vision of RDF as a joint representation that bridges the different knowledge sources [13, slide 15], we propose to compute a joint latent embedding space to bridge the various specialized embeddings. This principle retains the idea of mapping one KG embedding to another by expressing it as a composite of the two respective mappings via the latent space, i.e., the latent space can be seen as a learned hub to which we align KG embedding spaces. In this latent space, we assert that linked entities have a similar embedding. A key idea to enable this approach is that there should also be a mapping from each embedding space onto itself via the latent space. We formalize this insight by learning the latent space through autoencoders [14,15]. The benefit of this procedure is that autoencoders are trained in an unsupervised fashion, i.e., we can use all entities of a KG instead of only the linked ones to train the necessary mapping functions.

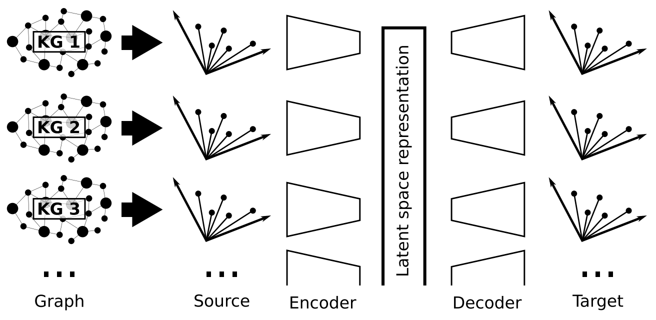


Fig. 1. FedCoder learns a mapping function (Encoder) from each source KG embedding into a latent space, and an analogous mapping from the latent space to each target KG embedding (Decoder), such that linked entities have the same representation in the latent space

The Semantic Web is large and diverse: As of May 2020, the Linked Open Data Cloud[2] covers over 1250 distinct resources and includes graphs from a variety of domains such as geography, linguistic resources, or scientific publications, as well as cross-domain resources like DBpedia [16]. Each of those resources can potentially be embedded, hence the Web of Embeddings is likewise confronted with heterogeneity in the embeddings and scalability in terms of graphs. In this paper, we, therefore, answer two research questions. First, we ask *how do different embedding space integration models compare in the face of heterogeneous embeddings*? In the Web of Embeddings, it is unlikely and, as argued, also undesirable that all embedding providers agree on the same embedding model since everyone optimizes towards their own data and goals. When empirically analyzing this question, we show that FedCoder delivers competitive integration performance between KGs when all are embedded with the same model and outperforms the state-of-the-art when different embedding models are employed.

Second, we ask *how do different embedding space integration models perform in the presence of multiple KGs*? As the Web of Embedding consists of more than two embedding spaces, it is essential to analyze the efficiency as well as the effectiveness of an integration approach, and to investigate how they can leverage/combine the knowledge from all KGs. Our results demonstrate that FedCoder scales better than a pairwise approach in the number of KGs and that the link prediction performance across graphs improves as more KGs are integrated simultaneously.

The remainder of this paper is structured as follows. First, we review related literature and contrast previous approaches with our work. Second, we introduce notations and give the necessary background. Third, we present our key ideas and discuss FedCoder in detail. Fourth, we contrast FedCoder against the state-of-the-art in their ability to integrate different embedding models (RQ1). Fifth, we demonstrate the scaling properties of FedCoder and the relevant baselines (RQ2). Finally, we conclude our main findings and discuss limitations and potential future work.

2. Related work

In this section, we review relevant KG embedding models as well as related embedding space integration methods that were developed for KG alignment, translation, or multi-modal embedding. In Table 1, we present an overview that distinguishes related works into four categories. First, we distinguish them by what source types they employ. We remark whether methods use embedding spaces from the same type of data source,

2 https://lod-cloud.net/
3 http://www.kgvec2go.org/
4 https://github.com/facebookresearch/PyTorch-BigGraph
5 https://github.com/uma-pi1/kge

Table 1
Literature overview.

Categorization				Methods	Problem setting
Source types	Learning framework	Integration type	Integration model		
Homogeneous	joint embedding	pairwise	identity	JE [4], MTransE [5], MGTransE [6], AliNet [7]	KG alignment
			translation	MTransE [5], MGTransE [6]	KG alignment
			projection	JEwP [4], MTransE [5], MGTransE [6]	KG alignment
			parameter-sharing	ITransE [11]	KG alignment
		latent space	replacement	Schwenk et al.[17]	Translation (sentences)
	mapping	pairwise	orthogonal projection	MUSE [18]	Translation (words)
Heterogeneous	joint embedding	pairwise	parameter-sharing, identity	StarSpace [19]	Multi-modal embedding
			replacement	Wang et al.[20]	Multi-modal embedding
			identity	KADE [21]	Multi-modal embedding
		latent space	orthogonal projection	MultiKE [22]	KG alignment

e.g., only KG embedding spaces (homogeneous), or if they mix different source types, e.g., use KG and word embeddings (heterogeneous). Second, we classify previous works by which learning framework they use. While most methods construct two embedding spaces simultaneously (joint embedding), others only learn a transformation from one pre-trained embedding space onto another (mapping). Third, we categorize approaches by their integration type. Specifically, we distinguish between pairwise approaches that combine two embedding spaces (pairwise) and latent space methods that combine embedding spaces in an intermediate, learned space (latent space). Fourth, we highlight various integration models, i.e., means to compare embedding vectors across spaces that have been employed by existing integration methods. In the remainder of this section, we first introduce KG embedding methods, then discuss previous works grouped by which problem setting they consider.

KG embedding models. A *KG embedding model* constructs a low-dimensional numerical representation, i.e., an embedding vector, for every entity and possibly every relation in the KG. The core idea is that these embeddings capture the structural information of the KG, such that the similarity of two entities' embeddings expresses how they are related. KG embedding methods mostly use this property to perform link prediction, i.e., estimating the most likely tail entity to a given head entity and a relation. The state-of-the-art can be coarsely separated into four categories: geometric, matrix factorization, deep learning, and language-based models [1,23]. Geometric models are based on the idea that the relation corresponds to a geometric operation between the head and tail entity in the embedding space. This was first realized in *TransE*, where the relation implements a translation from the head to the tail entities' embeddings [24]. The same idea was later extended in various ways, e.g., *TransH* or *TransR* use relation-specific projections of entity embeddings [1]. Matrix factorization models decompose the KG's adjacency tensor[6] into two separate components that characterize the entities or relations, respectively. The most prominent approaches are *RESCAL* and *DistMult*[25,26]. Both of them use a matrix to represent relations, but RESCAL uses a full-rank matrix while DistMult simplifies it to a diagonal matrix. Deep learning models employ a series of neural network layers with typically non-linear activation functions to capture patterns in the graph. For example, *ConvE* uses a two-dimensional convolutional layer on the head and relation embedding vectors that can learn elaborate interactions between them. They then use a fully connected layer with ReLU activation [27] to combine this with the tail entity embedding [28]. Language-based models such as *RDF2Vec* [29], *Owl2Vec** [30], *OPA2Vec* [31], or *Onto2Vec* [32] convert the graph into a set of entity sequences, then train a word embedding model [33] on these sequences.

For the geometric and matrix factorization categories, variants based on complex numbers were proposed. These models compose the embedding vector from a real and an imaginary part. Analogous to geometric models, *RotatE* implements the relation as rotation in the complex space [34]. For matrix factorization models, *ComplEx* factorizes the adjacency matrix in the complex space [35]. While numerous extensions and alternatives to the embedding models discussed here exist (see [1,23] for an overview), we focus on TransE, DistMult, RotatE, and ComplEx due to their conceptual clarity, conceptual relatedness, and widespread adoption.

Embedding-based KG alignment. Similar to FedCoder, embedding-based KG alignment methods establish a means to compare entity embeddings across KG embedding spaces. The goal of KG alignment is to find links between entities of two KGs. Embedding-based systems assume that linked entities are structurally similar in both KGs, and as a KG embedding captures a KG's structural information, they link entities with similar entity embeddings. To make entity embeddings comparable across KGs, several integration models have been proposed, which we review in the following.

Embedding-based KG alignment methods generally follow the joint embedding framework, which connects two embedding methods via a mapping function, then learns both KG embedding spaces simultaneously. The role of the two KG embeddings is to capture structural information of the respective graph, and the role of the integration model is to exchange information between them such that entity embeddings become comparable. Training a joint embedding requires the two KGs as well as a set of anchors, i.e., pairs of linked entities, along which the embedding spaces are joined. Approaches mostly differ in what integration model they employ.

In the most elementary view, embeddings of linked entities should be identical as they represent the same real-world concept. This is implemented by parameter-sharing, where linked entities share the same embedding by construction [11]. A relaxed version of this approach is to use a distance function instead of a shared embedding [4–7]. This ensures that linked entities have a similar but not necessarily identical embedding in both KG embedding spaces. The rationale is to have fewer constraints on the KG embedding model while still having a close coupling of the embedding spaces. The benefit of these two approaches is that the KG embedding integration introduces no additional parameters. The Semantic Web expresses links between KGs as relations, such as *owl:sameAs*. This concept can be mirrored in joint embedding by handling links between entities in the same

6 The three-way adjacency tensor is constructed by stacking the adjacency matrices of the KG per relation.

way as the KG embedding model handles relations. Several works build on TransE, which expresses relations as translations from subject to object, hence they likewise implement the mapping as translation between linked entities [5,6]. In contrast to a distance function, this approach introduces additional parameters into the integration model. Finally, multiple approaches suggest a projection matrix as the integration model [4–6]. Any integration model introduces additional constraints into the embedding model, which might have an adverse effect on the embedding itself. A projection matrix offers more flexibility than a distance or translation, so the embedding spaces remain loosely coupled, and different embedding dimensions can be chosen for the two KGs. This comes at the cost of an increase in parameters to be learned, which in turn increases the risk of overfitting. MultiKE [22] takes a different route by utilizing information from multiple modalities (triples, labels, and properties) about an entity. It uses TransE to embed triples, a custom convolutional neural network for properties, and an aggregate of pre-trained word embeddings for labels. Instead of comparing the KGs or modalities directly, it projects embeddings from all three modalities into a latent space via an orthogonal projection matrix. The latent space is trained such that an entity has the same representation across all modalities, and linked entities have the same representation across both KGs.

The discussed KG alignment methods differ from our work in three aspects: First, they all adopt the joint embedding framework, i.e., learn an embedding for both involved KGs. In contrast, we also aim at integrating pre-trained embedding spaces as training embedding spaces for multiple KGs is impractical. Second, most approaches use a pairwise integration and limit their discussion to two KGs. On the other hand, our problem setting requires the combination of multiple KG embeddings in a scalable way. Third, all discussed approaches assume the same KG embedding model and identical hyperparameters (mostly TransE) to embed the KGs. This assumption is unrealistic and unenforceable in the Web of Embeddings, hence we discuss the case where KGs are embedded with different embedding models and potentially different hyperparameters.

Language translation methods. Techniques to integrate multiple embedding spaces are also found in language translation methods. Given word or sentence embedding methods, their goal is to map representations between languages.

Schwenk et al. [17] propose a method that learns a common representation over sentences in multiple languages at once. Their approach is similar to ours in that they map every source language into a latent space and vice-versa. In contrast, they do not constrain the latent space, instead, they train by translating from every language into every other.

MUSE [18] is a word translation method that maps two word embedding spaces onto each other via an orthogonal projection. Similar to our problem setting, they assume pre-trained embedding spaces and only learn the mapping between them. However, they integrate embedding spaces pairwise while we take a latent space approach. Furthermore, they use the same word embedding model on both languages, while we are faced with different KG embedding models.

Multi-modal embedding methods. We aim at integrating multiple heterogeneous embedding spaces, a situation that also naturally occurs in multi-modal embedding. There, the idea is that different modalities convey different aspects of the same entity, and embedding multiple modalities jointly will result in better entity embeddings. These methods also typically follow the joint embedding paradigm by embedding all modalities with their own respective embedding model while connecting the embeddings through an integration model. To train the embeddings jointly, they also use a set of cross-modal anchors that are pairs of items from either modality.

Wang et al. [20] learn an embedding from KG triples and entity labels. Their KG embedding model is a probabilistic variant of TransE, for labels, they adapt the skip-gram word embedding model [33]. To join the two embedding spaces, they use the entity embedding in place of the respective word's embedding in the word embedding model and vice-versa. KADE [21] trains an embedding over a KG and sentences. They use off-the-shelf methods, TransE and its derivatives on the graph, and par2vec [36] for entity descriptions. The embedding spaces of both modalities are trained in an alternating fashion, where information between the two is exchanged in the form of a regularization term that minimizes the Euclidean distance between embeddings of corresponding entities and sentences. StarSpace [19] embeds multiple modalities with a single embedding model. They build a vocabulary of objects that includes items from all modalities, then generate a set of object pairs that are supposed to have similar embeddings. An object can be an item from any modality, or an aggregation thereof. They hereby do not distinguish whether or not items of a pair originate from the same modality. Effectively, they compare object embeddings via the cosine distance function. In part, it is also a parameter-sharing approach because it uses the same embedding if an object occurs in multiple modalities by construction.

In contrast to our work, these methods learn embedding spaces for all modalities, whereas we focus on a mapping between pre-trained embedding spaces. Furthermore, they use custom or selected embedding models for the involved modalities, while we cannot choose the embedding models freely but have to admit any KG embedding model in principle. Lastly, they predominantly use a pairwise, distance-based integration approach which is not suited for our purpose.

3. Preliminaries

In this section, we introduce the fundamental concepts and notations used throughout this paper (also consult Table 2 for a brief summary of symbols). We then describe a uniform framework to highlight commonalities and differences of state-of-the-art methods in the context of embedding space integration.

3.1. Knowledge graph

We define a knowledge graph as a structure of three sets $\mathcal{G} := (\mathcal{V}, \mathcal{R}, \mathcal{E})$, whereas $\mathcal{V}$ is a set of entities (such as :*turing* or :*mathematician*), $\mathcal{R}$ is a set of relations (e.g., :*occupation*), and $\mathcal{E}$ is a set of triples. In a KG $\mathcal{G}$, a triple $\tau = (h, r, t) \in \mathcal{E}$ represents a statement (e.g. (:*turing*, :*occupation*, :*mathematician*)), where $h \in \mathcal{V}$ is the head entity, $t \in \mathcal{V}$ is the tail entity, and $r \in \mathcal{R}$ is the relation. We interchangeably refer to entities as e, h, or t depending on whether or not their position in a triple is relevant and use the bold notation of these symbols (i.e., $\mathbf{e}$, $\mathbf{h}$, or $\mathbf{t}$) for their respective embeddings.

Given two KGs, $\mathcal{G}_i := (\mathcal{V}_i, \mathcal{R}_i, \mathcal{E}_i)$ and $\mathcal{G}_j := (\mathcal{V}_j, \mathcal{R}_j, \mathcal{E}_j)$, we define the set of anchors $\mathcal{A}$ as pairs of entities that are linked across graphs i and j with the *owl:sameAs* predicate, i.e., a set of entity pairs that represent the same real-world concept: $\mathcal{A} = \{(e_i, e_j) \in \mathcal{V}_i \times \mathcal{V}_j\}$. We assume anchors to be unambiguous so that an entity from $\mathcal{G}_i$ is linked to *at most* one entity from $\mathcal{G}_j$ and vice-versa. An entity from one graph can, however, still be linked to multiple other graphs. We further assume that at least a portion of these anchors $\mathcal{A}$ are known a priori to train the integration model.

3.2. Pairwise learning frameworks

To integrate embedding spaces, we can either embed the respective KGs jointly or learn pairwise mappings between

Table 2
Symbols overview.

Symbol	Description
$\mathcal{V}_i$	Set of entities in KG *i*.
$\mathcal{R}_i$	Set of relations in KG *i*.
$\mathcal{E}_i$	Set of edges in KG *i*.
$\mathcal{G}_i$	The KG *i*.
h, r, t	head, relation, and tail of a triple.
τ	A triple (h, r, t).
$\mathcal{A}$	Set of anchors between KGs.
N	Number of KGs or embedding spaces.
k_i	The dimension of space *i*.
Z	The latent space.
$m_{i,j}$	A mapping from space *i* to space *j*.
$\mathbf{h}, \mathbf{r}, \mathbf{t}$	Embedding vectors of triple components.
$\mathbf{e}_i$	Embedding vector from space *i*.
$\mathbf{z}$	A vector in the latent space Z.
$\mathcal{L}$	Overall loss.
$\mathcal{L}^I$	Integration loss.
$\mathcal{L}_i^{\mathcal{G}}$	KG embedding model loss for graph *i*.
$\mathcal{L}_i$	Autoencoder loss for space *i*.
$\mathcal{L}^R$	Orthogonal regularizer loss.
γ	Margin between positive and negative samples.
λ	Regularization coefficient.
$\mathbf{P}$	A projection matrix.
g	An arbitrary transformation function.
h	An activation function.
$\mathbf{W}$	A weight matrix.
$\mathbf{b}$	A bias vector.

pre-trained embedding spaces. In both cases, we formulate an optimization problem, expressed by a loss $\mathcal{L}$. Joint KG embedding is the most widespread approach to combining KG embedding spaces in a uniform space. In this framework, we learn the KG embedding spaces and connect them along anchors across the graphs, i.e., linked entities. The loss, therefore, consists of two components: One term for the respective KG embedding model $\mathcal{L}_i^{\mathcal{G}}$, and one to integrate embeddings across the graphs $\mathcal{L}^I$:

$$\mathcal{L} \sim \sum_{i=1}^{N} \mathcal{L}_i^{\mathcal{G}} + \mathcal{L}^I \tag{1}$$

The advantage of joint embedding is that both embedding spaces can be adapted to one another.

In the case where KG embedding spaces are pre-trained, we learn a mapping between each pair of spaces but not the embedding spaces themselves. The loss hence only includes the mapping:

$$\mathcal{L} \sim \mathcal{L}^I \tag{2}$$

The benefit of this scenario is that it is more efficient than joint embedding since we take advantage of computations that were already performed.

3.3. KG embedding models

A KG embedding model constructs a low-dimensional numerical representation, i.e., an embedding vector, for every entity and possibly every relation in the KG. This is achieved by defining a triple score function $f(\tau) : (\mathcal{V} \times \mathcal{R} \times \mathcal{V}) \mapsto \mathbb{R}$ which assesses the inverse plausibility of a triple $\tau = (h, r, t)$. The embedding vectors $\mathbf{h}$, $\mathbf{r}$, and $\mathbf{t}$ appear as latent vectors of size k (the embedding dimension) in f. Table 3 summarizes the triple score functions of the embedding models considered throughout this paper.

Table 3
Overview of different KG embedding models' triple score functions f. $\mathbf{h}, \mathbf{r}, \mathbf{t}$ denote embedding vectors of a triple's head h, relation r, or tail t. $\bar{\cdot}$ denotes the complex conjugate, $\odot$ is the Hadamard product. Re(·) is a function that returns the real part of a complex vector, and diag(·) returns the diagonal elements of a square matrix. The matrix factorization models express the triple score as similarity rather than a distance measure, hence the negative sign.

Model	Triple score function f
TransE	$\lVert\mathbf{h}+\mathbf{r}-\mathbf{t}\rVert_2$
DistMult	$-\left\lVert\mathbf{h}^T \mathrm{diag}(\mathbf{r})\mathbf{t}\right\rVert_2$
ComplEx	$-\left\lVert\mathrm{Re}\{\mathbf{h}^T \mathrm{diag}(\mathbf{r})\bar{\mathbf{t}}\}\right\rVert_2$
RotatE	$\lVert\mathbf{h}\odot\mathbf{r}-\mathbf{t}\rVert_2$

True statements are supposed to have substantially lower scores than false ones. Since KGs typically follow the open-world assumption, we cannot know whether a statement that is not part of the KG is false or not. To cope with this, KG embedding models typically generate negative triples by corrupting either the head or tail entity of a positive triple. We denote the positive triples as $\tau \in \mathcal{E}^+$ and negative triples as $\tau \in \mathcal{E}^-$.

Embedding vectors are learned by minimizing the overall loss $\mathcal{L}_{\mathcal{G}}$, which aggregates the scores of all positive and negative triples in the KG. Models predominantly use a margin-based loss, which tries to separate positive from negative triple scores by a margin of γ:

$$\mathcal{L}^{\mathcal{G}} \sim -\sum_{\tau\in\mathcal{E}^+} \log\sigma\left(\gamma - f(\tau)\right) - \sum_{\tau\in\mathcal{E}^-} \log\sigma\left(f(\tau) - \gamma\right), \tag{3}$$

where σ denotes the *sigmoid* function and γ is a hyperparameter.

3.4. Integration models

In an embedding space, entity similarity is expressed through the similarity of their respective embeddings. To integrate entity embeddings between two KGs, we likewise want linked entities to have similar embeddings. We achieve this by transforming one of the two spaces and by ensuring that transformed entity embeddings are close to the embedding of the respective linked entity in the other space.

Formally, we define a mapping m from embedding space i to space j as a function $m_{i,j}(\mathbf{e}_i) : \mathbb{R}^{k_i} \mapsto \mathbb{R}^{k_j}$. As this mapping should place an entity embedding from space i close to the respective entity embedding from space j (for all linked entities), we want to minimize the distance d between entity embeddings after mapping. We hence define the integration loss as:

$$\mathcal{L}^I \sim \sum_{(e_i,e_j)\in\mathcal{A}} d\left(m_{i,j}(\mathbf{e}_i), \mathbf{e}_j\right) \tag{4}$$

Typically, d is the Euclidean distance, i.e., $d(\mathbf{x}, \mathbf{y}) = \lVert\mathbf{x}-\mathbf{y}\rVert_2$, however other distance measures like the cosine distance can be used as well. Note that the mapping m can be applied on either embedding space, hence we omit the reciprocal form. All embedding space integration approaches use this notion but differ in the mapping function m they consider. In the following, we discuss the different options.

In the simplest case, we compare the entities directly to each other. Hence, the *Identity* mapping is defined as:

$$m_{i,j}(\mathbf{e}_i) = \mathbf{e}_i \tag{5}$$

Effectively, this means that linked entities should ideally have the same embedding in both spaces [4,11,19–21]. The advantages of this integration model are its conceptual simplicity and that it does not introduce new parameters that have to be learned. For the same reason, it is, however, only applicable in a joint embedding framework since it can only combine embedding spaces by changing the embedding vectors directly. Furthermore, it requires the embedding dimensions of the KG models to match, i.e., $k = k_i = k_j$, so it can only be applied if the embedding models' hyperparameters match. A high enough weight on $\mathcal{L}^I$ forces the respective entity embeddings across the graphs to become identical, hence we can also interpret this approach as a soft parameter-sharing model [11,19].

The *Translation* mapping is defined as:

$$m_{i,j}(\mathbf{e}_i) = \mathbf{e}_i + \mathbf{v} \tag{6}$$

Inspired by TransE's geometric interpretation of the embedding space, the mapping can be interpreted as another relation and, hence, be modeled by a translation vector $\mathbf{v}$ in the embedding space [5,11]. Evidently, this allows the KG embedding spaces to be offset by a constant, at the cost of additional $\mathcal{O}(k_i)$ parameters, and assuming matching embedding dimensions $k = k_i = k_j$. Although possible in principle, it is unlikely that pre-trained embedding spaces have the exact same shape besides an offset. Therefore, this model is only used in a joint embedding approach.

The *Projection* mapping is defined as:

$$m_{i,j}(\mathbf{e}_i) = \mathbf{P}_{ij}\mathbf{e}_i \tag{7}$$

By having a projection matrix as mapping, we can cope with geometric transformations such as a rotation between the two embedding spaces [4,5,11]. In contrast to the other mapping functions, it can also be applied if embedding dimensions do not match, i.e., $k_i \neq k_j$, at the cost of $\mathcal{O}(k_i k_j)$ additional parameters to be learned. At the same time, the additional parameters make this mapping more flexible than the previous ones, which has two benefits: First, the KG embedding models remain loosely coupled in a joint embedding setup so that they can potentially better optimize for their own objective. Second, it is a reasonable model when embedding spaces are pre-trained, as it can apply geometric transformations to warp one space onto another.

Building on the projection mapping, some approaches require the projection to be orthogonal [18,22]. This additional constraint on the projection matrix typically helps to prevent overfitting, especially when the number of anchors is much lower than the number of parameters in the mapping. The *Orthogonal Projection* is defined like the projection mapping in Eq. (7) but introduces an additional regularization on the projection matrices $\mathbf{P}_{ij}$:

$$\mathcal{L}^R = \sum_{i,j} \left\| \mathbf{P}_{ij}^T \mathbf{P}_{ij} - \mathbb{1} \right\|_F^2 \tag{8}$$

with $\mathbb{1}$ the identity matrix, $\|\cdot\|_F$ the Frobenius norm. This regularization term is added to the mapping loss in Eq. (4), i.e., we can rewrite the overall losses in Eqs. (1) and (2) as $\mathcal{L}' = \mathcal{L} + \lambda\mathcal{L}^R$, with the regularization coefficient λ as a hyperparameter.

4. FedCoder

FedCoder builds on two foundations: A latent space and autoencoding. The latent space serves as a hub to efficiently integrate multiple KG embedding spaces, circumventing the need for pairwise space integration. The autoencoder provides a means to learn the latent space from all entities instead of only linked ones, based on the idea that every entity embedding should map onto itself via the latent space. Fig. 2 visualizes both foundations. FedCoder can be employed in the joint embedding framework or to only compute a mapping between pre-trained embedding spaces.

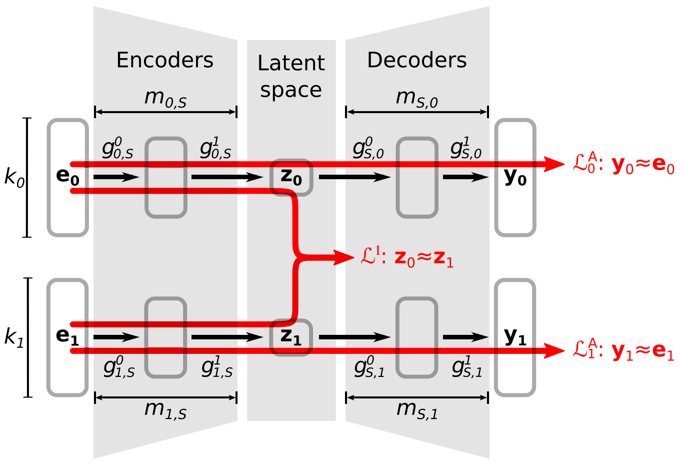


Fig. 2. Schematic view of FedCoder on two embedding spaces. The encoders realize the mapping from each space into the latent space ($m_{i,S}$), the decoders map latent space vectors into the embedding spaces ($m_{S,i}$). The encoders and decoders both have two layers. The black paths indicate the propagation of the embedding vectors through the layers, the red arrows depict the idea behind the integration and the autoencoder losses.

4.1. Latent space

Previously, integration approaches learned pairwise mappings between mostly two KG embedding spaces; Our goal, however, is to integrate $N \geq 2$ embedding spaces. With a pairwise integration approach, one would have to compute $\mathcal{O}(N^2)$ mappings — one from every graph to any other. Clearly, this does not scale well in the number of graphs. Instead, one may select one KG as a hub H and only learn mappings between each graph and the hub, and vice-versa. This reduces the computational effort to $\mathcal{O}(2N)$ mappings. We can still get a mapping between any two graphs by going through the hub H, i.e., $m_{i,j}(\mathbf{e}_i) := m_{H,j}(m_{i,H}(\mathbf{e}_i))$.

We build on this idea, but instead of selecting an actual KG as the hub, we propose a latent space Z with k_Z dimensions that takes its role. The benefit of this latent space is three-fold. First, a latent space can learn from entities linked between any two graphs. On the other hand, when using a KG as a hub, all graphs must be linked to the hub. Entities that are not part of the hub cannot be used to learn the mappings, therefore, valuable information is not exploited. Second, we do not integrate into a pre-determined KG embedding space but a learned latent space. This offers more versatility as the latent space can be adapted to the specific combinations of graphs to be integrated. Third, we learn the latent space from all linked entities simultaneously. In contrast, a KG hub approach learns pairwise mappings independently from each other.

When mapping embedding spaces onto one another, the goal is to have the same (or at least similar) embedding vectors for linked entities. We adopt the same idea, i.e., two linked entities should have a similar representation in the latent space Z, as depicted in Fig. 2. The FedCoder integration loss is hence formally expressed as:

$$\mathcal{L}^I \sim \sum_{(e_i,e_j)\in\mathcal{A}} d\left(m_{i,Z}(\mathbf{e}_i), m_{j,Z}(\mathbf{e}_j)\right) \tag{9}$$

Here, d is a distance function, typically the Euclidean or cosine distance, and the implied dimension k_Z of the latent space is a hyperparameter that can be chosen freely. This formulation is a generalization of Eq. (4), as here, we transform the embedding vectors from both spaces into the latent space Z instead of transforming one into the other (via the functions $m_{i,Z}$ and $m_{j,Z}$). Note that this constraint on the shared space applies to all anchors $\mathcal{A}$, i.e., entities linked between any two KGs. We neither require an entity to be linked between all involved graphs nor to a specific one. This is in contrast to the setting of using one KG as a hub, where one could only link to entities contained in this hub.

Integration loss with negative samples. In the latent space, embedding vectors of linked entities are supposed to be similar. Conversely, embedding vectors of distinct entities should be distant. We thus propose a negative sampling procedure inspired by the negative sampling in embedding methods: In each pair of linked entities, we replace one of the two with a randomly sampled entity from the respective KG. Besides minimizing the distance between latent-space representations of linked entities, we also maximize the distance between the latent-space representation of corrupted entity pairs.

We follow the concept presented in Eq. (3), which aims at separating positive from negative samples by a margin of γ. We here omit the input to the distance function (Eq. (9)) for readability:

$$\mathcal{L}^I \sim -\sum_{(e_i,e_j)\in\mathcal{A}} \log\sigma\left(\gamma - d(\cdot)\right) - \sum_{(e_i,e_j)\notin\mathcal{A}} \log\sigma\left(d(\cdot) - \gamma\right) \tag{10}$$

4.2. Autoencoding

So far, we discussed mapping an embedding vector $\mathbf{e}_i$ from space i into the latent space Z using a function $m_{i,Z}(\mathbf{e}_i)$, but this is only half of what we need: We also require a mapping from its latent space $\mathbf{z}$ to each KG embedding space, i.e., $m_{Z,j}(\mathbf{z})$ Only with both mappings can we transform an embedding vector between any two spaces as in $m_{i,j} := m_{Z,j}(m_{i,Z}(\mathbf{e}_i))$. We can interpret this in an encoder–decoder framework, where $m_{i,Z}(\cdot)$ encodes an embedding from a source embedding space into a latent representation, and $m_{Z,j}(\cdot)$ decodes that latent representation into a target space. Under this view, the latent representation of an entity can be decoded into any of the embedding spaces, including the one from which it was originally retrieved. In other words, any entity embedding from a KG embedding space can be mapped into the same space via the latent one, and naturally, we would expect to recover the same embedding vector: $m_{Z,i}(m_{i,Z}(\mathbf{e}_i)) = \mathbf{e}_i$. This is the core idea of autoencoding [14]. Fig. 2 visualizes this principle via the horizontal red arrows.

The benefit of autoencoding is that it can be trained in an unsupervised manner, as it only requires input embeddings $\mathbf{e}_i$. In practice, it is often desirable to have a bottleneck in the autoencoder that forces it to generalize rather than to learn mappings that copy the input to the output. In our case, this can be controlled by choosing the size of the latent space smaller than that of the input embedding space $k_S < k_i, \ \forall i \in [1, N]$. With $\|\cdot\|_F$ the Frobenius norm, we define the autoencoder loss for graph i as:

$$\mathcal{L}_i^A \sim \sum_{e_i\in\mathcal{V}_i} \left\| \mathbf{e}_i - m_{Z,i}(m_{i,Z}(\mathbf{e}_i)) \right\|_F \tag{11}$$

Like the latent space constraints from Eq. (9), this autoencoder loss adds constraints on the mappings m. But in contrast to Eq. (11), it sums over all entities of the KG, not only over the linked ones given by $\mathcal{A}$. The advantage of this is that with the autoencoder, we can learn the mappings m, and indirectly the latent space, using far more training data than only the given anchors. This is especially useful to prevent overfitting when there are more parameters in the mapping functions than the number of anchors we have.

4.3. Latent-space learning frameworks

The overall loss function of FedCoder is composed of the integration loss $\mathcal{L}^I$ and the autoencoder loss $\mathcal{L}_i^A$ and follows the structure of the frameworks presented in Section 3.2. Hence, in the scenario where embeddings are pre-computed and only the mapping is trained, the overall loss is

$$\mathcal{L} = \mathcal{L}^I + \lambda \sum_{i=1}^{N} \mathcal{L}_i^A \tag{12}$$

with $\mathcal{L}^I$ being the mapping loss with negative sampling (Eq. (9) and (10)), and $\mathcal{L}_i^A$ being the autoencoder loss for graph i (Eq. (11)).

In the joint embedding framework, the overall loss additionally includes the KG embedding loss $\mathcal{L}^{\mathcal{G}}$ (Eq. (3)):

$$\mathcal{L} = \sum_{i=1}^{N} \mathcal{L}_i^{\mathcal{G}} + \mathcal{L}^I + \lambda \sum_{i=1}^{N} \mathcal{L}_i^A \tag{13}$$

In this case, the optimization problem covers not only the parameters of the mapping functions but also the embedding vectors themselves, irrespective of where in these three terms they appear. The autoencoder coefficient λ can be chosen freely.

Mapping function. We have not yet specified the mapping functions m, as neither the integration loss (Eq. (9)) nor the autoencoder loss (Eq. (11)) assumes a specific one — they do not even require the same mapping function for different KGs, nor the same function for the encoder and decoder of one KG. We can hence assume that m is a multi-layer neural network, i.e., a composite of transformations g:

$$m_{i,j}(\mathbf{e}_i) = \left(g_{i,j}^0 \circ g_{i,j}^1 \circ \dots\right)(\mathbf{e}_i) \tag{14}$$

where each layer is specified by a weight matrix of appropriate dimensions $\mathbf{W}_{ij}$, a bias vector $\mathbf{b}_{ij}$, and an activation function $h(\cdot)$:

$$g_{i,j}^l(\mathbf{x}) = h\left(\mathbf{W}_{ij}^l \mathbf{x} + \mathbf{b}_{ij}^l\right) \tag{15}$$

The sizes of intermediate layers can be chosen freely, although the encoder and decoder for one KG are typically symmetrical, i.e., use the same number of layers and the same layer sizes in reverse.

FedCoder admits any kind of mapping function between the KG embedding spaces and the latent space. For special cases of mapping functions, however, efficient alternatives exist and have been used previously [22]. Specifically, if the mapping $m_{i,S}$ is an orthogonal projection matrix $\mathbf{P}_{iZ}$, the inverse mapping $m_{S,i}$ is the transposed of that projection matrix, i.e., $\mathbf{P}_{Zi} = \mathbf{P}_{iZ}^T$.

We hence define the *Latent Orthogonal Projection* integration model by using the projection mapping as in Eq. (7) in the latent-space integration loss (9) and adding the soft-orthogonal regularizer (8). Analogous to Eq. (12), the integration loss becomes:

$$\mathcal{L} \sim \sum_{(e_i,e_j)\in\mathcal{A}} d\left(\mathbf{P}_{iZ}\mathbf{e}_i, \mathbf{P}_{jZ}\mathbf{e}_j\right) + \lambda \sum_{i=1}^{N} \left\| \mathbf{P}_{iZ}^T \mathbf{P}_{iZ} - \mathbb{1} \right\|_F^2 \tag{16}$$

Structurally, we can interpret the autoencoding loss as a regularizer on the integration loss, hence the orthogonal regularizer takes the same role in this model as the autoencoder in FedCoder. The difference to FedCoder is that this approach is limited to the projection mapping, while FedCoder can incorporate any mapping function.

5. Evaluation #1: How do different embedding space integration models compare in the face of heterogeneous embeddings?

The Web of Embeddings is diverse: There exists a range of different embedding models adapted to different structures and use cases. In a distributed environment, independent parties each select an embedding model that fits the properties of the respective KG and that matches the tasks they intend to cover. The Web of Embeddings, therefore, consists of heterogeneous embedding spaces, as different graphs are likely embedded with different KG embedding models and/or hyperparameters. Yet, as of now, embedding space integration approaches implicitly assume that all KGs are embedded with the same model and the same embedding dimension. As we cannot make this assumption, we ask *how do different embedding space integration approaches compare in the face of heterogeneous embeddings*?

Procedure. We answer this question by conducting a series of experiments that integrate two KG embedding spaces. In this scenario, we can either embed both graphs with the same model (homogeneous case) or with different embedding models and/or embedding dimensions (heterogeneous case). The homogeneous case is analogous to related work and represents a best-case scenario, as we would expect spaces that are constructed in the same manner to compare well. The heterogeneous case mirrors the situation we face in the Web of Embeddings, and we focus on heterogeneity in the choice of the embedding models or the embedding dimensions across graphs. Specifically, we study four embedding models — TransE, RotatE, ComplEx, and DistMult— that relate to each other by either building on the same paradigm (geometric or matrix factorization) or the same numeric space (real or complex).

Evaluation metrics. In an integrated space, we expect the embedding vectors of each pair of linked entities to be similar. Specifically, they should be more similar to each other than to the remaining entity embeddings in the space, i.e., they should be nearest neighbors. To quantify the integration performance, we adopt the measures of embedding literature to assess the link prediction performance since this is also a nearest neighborhood problem. We thus measure the rank of an entity in a testing pair as the number of entities that are closer in the integrated space than its truly linked one. This typically produces a long-tail rank distribution that we characterize by the mean rank (MR) and the percentage of ranks below 10 (HITS@10). The mean rank (lower is better) gives an impression of how close embedding vectors are in the average case, HITS@10 (higher is better) states how skewed the rank distribution is towards the front.

Evaluation structure. FedCoder can learn the mappings between pre-trained embedding models, or it can embed the two KGs jointly. The former approach represents the situation of the Web of Embeddings, the latter one is more common in the literature. Next, we briefly present the datasets used in our experiments and then discuss both cases in Sections 5.2 and 5.3.

5.1. Datasets

Our experiments use up to five datasets, each consisting of two KGs and anchors between them. The five datasets differ in which KGs they include and the proportion of entities linked between them. We summarize their characteristics in Table 4 and discuss each of them in turn.[7]

Split15k-237. We construct two partially aligned KGs on the basis of FB15k-237, a dataset derived from freebase and commonly used to evaluate KG embedding methods [37]. We first randomly sample a set of about 20% of FB15k-237's entities that become part of both KGs and divide the remaining entities randomly between the two KGs. Each KG is then populated with triples from FB15k-237 whose head and tail are part of the respective graph. This construction procedure is similar to the one proposed in [4]. The resulting dataset becomes substantially smaller than FB15k-237 since it disregards triples with the head and tail in different KGs. By construction, entities are linked unambiguously and one-to-one, and only about a fourth of the entities are linked.

[7] The datasets and our implementation can both be found at https://gitlab.ifi.uzh.ch/DDIS-Public/fedcoder.

Table 4
Dataset characteristics.

Dataset	KG	Entities	Relations	Triples	Anchors
split15k-237	left	8724	232	110549	2793
	right	8725	227	98879	
WD15k-237	fb	14541	237	292581	14296
	wd	14295	322	137960	
$\text{DBP}_{\text{FR-EN}}$	fr	19661	903	105998	15000
	en	19993	1208	115722	
$\text{DBP}_{\text{JA-EN}}$	ja	19814	1299	77214	15000
	en	19780	1153	93484	
$\text{DBP}_{\text{ZH-EN}}$	zh	19388	1701	70414	15000
	en	19572	1323	95142	
DBP-YG	dbp	100000	302	386057	100000
	yg	100000	31	452307	

Wd15k-237. Since FB15k-237 is one of the most widespread datasets in embedding research, we also combine it with a second, distinct KG extracted from Wikidata. For this, we use a third-party alignment[8] between the two KGs to select the relevant Wikidata entities, then retrieve all Wikidata triples between them. The alignment is not quite complete, and Wikidata entities that correspond to more than one entity in FB15k-237 are discarded, leading to a slightly smaller entity set on the Wikidata side. This dataset features more triples on both KGs than split15k-237, whereas all entities from the Wikidata side and most entities from FB15k-237 are linked.

Dbp15k. We further use three datasets used by embedding-based KG alignment methods since they naturally come with two KGs and links between them. The DBP15k dataset collection was proposed in [38] and was built from different languages of DBpedia. It includes three pairs of KGs, each combining English with another language: Chinese ($\text{DBP}_{\text{ZH-EN}}$), Japanese ($\text{DBP}_{\text{JA-EN}}$), and French ($\text{DBP}_{\text{FR-EN}}$). Each pair of KGs includes 15000 anchors, which is almost three-quarters of the entities in the respective graphs.

Dbp-yg. Additionally, we evaluate on the DBP-YG dataset from [12], which includes subsets of DBpedia and YAGO. This is the largest dataset in terms of entities and triples and provides links for all its entities.

5.2. Pre-trained embedding space mapping

We first evaluate FedCoder in the setting where embedding spaces for the two KGs have already been trained, and only the mapping between them is learned (see Eqs. (2) and (12)). In this scenario, we only need the two embedding spaces and anchors, but not the KGs themselves, for the integration. We compare FedCoder to three previously proposed alternatives. First, the Projection model (*Proj*) according to Eq. (7) and introduced by [4–6]. Second, the Orthogonal Projection model (*Proj+Ortho*) given in Eq. (8) and proposed by [18]. Third, the Latent Orthogonal Projection model (*Latent+Ortho*) as defined by Eq. (16) and used in [22]. The two remaining integration models (Identity and Translation) are not suited for this scenario as they have little to

[8] https://github.com/villmow/datasets_knowledge_embedding

no means to learn a mapping between fixed embedding spaces. In the remainder of this section, we first describe the experimental setting, then discuss the homogeneous case, and finally show our results for the heterogeneous scenario.

5.2.1. Setup

We use LibKGE [9] to embed KGs from all datasets with TransE, RotatE, DistMult, and ComplEx. In a first step, we perform a hyperparameter search over the embedding dimension ($\{32, 64, 128, 256, 512\}$), the amount of negative samples ($\{32, 64, 128, 256, 512, 1024\}$), dropout ($[0.0, 0.5]$), and the learning rate ($[0.0003, 1.0]$). For each embedding model, we first select the embedding dimension that gives the best performance across both KGs of a dataset. This is to create a homogeneous setting where the two KGs are embedded with the same embedding model and identical embedding dimensions. We then select the optimal values for the remaining parameters per graph and embedding model. Having the hyperparameters, we then train the embedding for a maximum of 1000 epochs or until the performance starts to decrease (early stopping).

To evaluate the integration models, we randomly sample 20% of the datasets' known anchors $\mathcal{A}$ as a test set and use the remaining ones to train the models. We train and evaluate on five different, randomly sampled train-test splits and report the averaged results. To train the methods, we use the Adam optimizer with a learning rate of 0.01 (0.001 for DBP-YG) and train for a maximum of 5000 epochs or until the performance decreases six consecutive times, whereas we validate every 100 epochs. As mapping functions for FedCoder, we use single-layer networks with a linear activation function. We use the Euclidean distance in all integration models as it resulted in equal or better performance than the cosine distance across all cases. We initialize all weight and projection matrices with the identity matrix. For the two latent space methods (FedCoder and Latent Orthogonal Projection), we use 16 negative samples and a margin of $\gamma = 24$ and select the best performing latent space size k_Z out of $\{32, 64, 128, 256\}$. Typically, a size of 128 performed best for the Latent Orthogonal Projection, a size of 256 was ideal for FedCoder. For FedCoder and the two orthogonal methods (Orthogonal Projection and Latent Orthogonal Projection) we further search the optimal regularization coefficient $\lambda \in \{1.0, 0.1, 0.01, 0.001\}$. Consequently, we use $\lambda = 0.01$ in all cases except on DBP-YG, where FedCoder has shown the best performance for $\lambda = 0.001$. This hyperparameter optimization is performed as a grid search on a separate train-test split that is not used in further experiments.

5.2.2. Homogeneous models

In the homogeneous setting, we integrate KG embedding spaces that were produced with the same model and with the same embedding dimension.

Table 5 shows the performance of the different integration models per dataset and embedding model in terms of mean rank (MR) and HITS@10. We hypothesize that FedCoder performs better across all embedding models and datasets than the other integration approaches. More generally, we also presume that latent space methods perform better than pairwise methods. The rationale is that these methods have more trainable parameters and can hence potentially learn more complex functions.

In Table 5, we can observe that FedCoder has the lowest (i.e., best) mean rank in almost all cases and also achieves the best HITS score in the vast majority of cases. Compared to the Latent Orthogonal Projection model, it has an improvement in the MR of 46 ranks (22%) on average and shows an average increase in HITS of 5.4 (16%). Compared to the pairwise methods, FedCoder, on average, has a 45% lower MR and a 10% higher HITS. The benefit of FedCoder is substantial, with a decrease in MR of up to 1974 ranks (87%, ComplEx on WD15k-237) and an increase in HITS of up to 16pp (47%, DistMult on WD15k-237). In the worst case (DistMult on $\text{DBP}_{\text{JA-EN}}$), its HITS score is only 7pp (34%) below that of the there best-performing pairwise model. In particular, its mean rank is never significantly worse than that of the two baselines. Both comparisons demonstrate that FedCoder has a clear advantage over all other integration models.

Concerning latent space versus pairwise models, we observe that the Latent Orthogonal Projection model achieves a better mean rank than the two pairwise integration approaches in most cases. However, its HITS score is worse than the pairwise integration approaches in 14 out of 24 cases. For ComplEx and DistMult on $\text{DBP}_{\text{JA-EN}}$, the difference is substantial, however, this might be a case of overfitting. On average, even excluding the two worst cases, the HITS of the Latent Orthogonal Projection model is below that of the two pairwise models. Hence, the Latent Orthogonal Projection improves in the long tail of the rank distribution (better average rank) at the cost of losing precision among the best ranks (lower HITS).

Lastly, we want to point out that the two pairwise integration models do not have a significant difference, with few exceptions. This is unexpected since the orthogonalization has been shown to improve the integration of word embedding spaces [18].

5.2.3. Heterogeneous models

In the heterogeneous setting, we integrate KG embedding spaces that were produced by different embedding models and potentially different embedding dimensions.

Table 6 lists the integration performance for all embedding model combinations in terms of the mean rank (MR) and HITS@10 (HITS). As in the homogeneous case, we hypothesize that FedCoder achieves the overall best performance in the heterogeneous setting and that latent-space methods outperform the two pairwise methods. We would, however, not expect that the integration models reach a better performance in this scenario than in the homogeneous one since spaces created by the same model share their notion of embedding vector similarity. Furthermore, we presume that embedding model combinations that share a trait (category or numeric space) integrate better than models with no commonality.

In Table 6, we see that FedCoder outperforms the other integration models in all but a few cases. FedCoder typically achieves better MR and HITS scores than the Latent Orthogonal Projection model. On the $\text{DBP}_{\text{JA-EN}}$ and $\text{DBP}_{\text{ZH-EN}}$ datasets, the two methods perform comparably, as the difference between the two is not significant. On average, the FedCoder's mean rank is 85 ranks lower (25%), and the HITS score is 8.6pp higher (35%) than for the Latent Orthogonal Projection model, whereas we see improvements in MR of up to 671 ranks (53%, RotatE vs. ComplEx on DBP-YG), and in HITS of up to 35pp (138%, DistMult vs. ComplEx on WD15k-237). In contrast to the two pairwise baselines, FedCoder has a better MR by 661 ranks (56%) on average and an improvement in HITS of 7.7 (38%).

With the exception of HITS of the DistMult versus ComplEx combination on the $\text{DBP}_{\text{JA-EN}}$ and $\text{DBP}_{\text{ZH-EN}}$ datasets, the latent-space approaches perform better than the two pairwise integration models. This corresponds with our observations in the homogeneous case, where the pairwise models performed better on the same datasets and embedding models.

We next contrast FedCoder's performance in the heterogeneous case with the performance of the two respective embedding models in the homogeneous setting. The heterogeneous setting rarely outperforms both respective homogeneous ones (an example of this is RotatE and DistMult on WD15k-237). However, in most cases, it achieves a better performance than the worse

Table 5
Integration performance of homogeneous, pre-trained embedding spaces. Results marked with † are not significantly better than the runner-up (Wilcoxon test at $p = 0.05$). The best performance is marked in bold font, the worst one is italicized. Proj: Projection, Proj+Ortho: Orthogonal Projection, Latent+Ortho: Latent Orthogonal Projection.

Dataset	Model	MR				HITS@10			
		Proj	Proj+Ortho	Latent+Ortho	FedCoder	Proj	Proj+Ortho	Latent+Ortho	FedCoder
split15k-237	ComplEx	37.6	38.1	*59.7*	**15.0**	51.9	51.6	*32.6*	**66.5**
	DistMult	49.0	48.3	*60.3*	**21.7**	41.3	41.5	*23.7*	**54.4**
	RotatE	*31.2*	31.1	23.2	**12.8**	67.8	67.9	*62.0*	**75.5**
	TransE	17.3	*17.4*	10.9	**8.9**	78.1	*78.0*	**82.2**	81.0
WD15k-237	ComplEx	302.4	*546.5*	158.8	**69.2**	30.2	23.6	*22.1*	**33.2**
	DistMult	*226.7*	225.7	62.3	**48.2**	*33.7*	33.8	45.9	**49.8**
	RotatE	*122.5*	121.8	41.9	**34.9**	*52.1*	52.3	55.9	**63.2**
	TransE	*163.7*	163.4	82.3	**60.1**	44.0	44.2	*42.8*	**48.5**
DBP$_{\text{FR-EN}}$	ComplEx	467.9	*468.7*	347.5	†**327.4**	†**28.7**	†**28.7**	*16.0*	23.2
	DistMult	439.9	*546.7*	253.0	**151.0**	29.6	28.1	*27.9*	**34.7**
	RotatE	*198.8*	197.6	78.2	**69.4**	*28.0*	28.2	34.1	**37.3**
	TransE	72.6	72.6	*340.9*	†**70.3**	66.8	66.9	*31.6*	**69.9**
DBP$_{\text{JA-EN}}$	ComplEx	573.6	573.3	*1092.8*	†**554.3**	†**22.4**	22.3	*4.7*	17.0
	DistMult	594.7	594.9	*706.4*	**564.7**	†**20.2**	†**20.2**	*9.2*	13.3
	RotatE	*362.3*	361.0	172.7	**154.7**	*17.2*	17.3	20.8	**22.5**
	TransE	148.6	148.6	*152.6*	**139.8**	49.9	49.9	*34.0*	**53.7**
DBP$_{\text{ZH-EN}}$	ComplEx	561.6	*561.7*	519.1	**401.6**	†**25.7**	25.6	*14.7*	20.8
	DistMult	481.3	481.5	*526.8*	**432.2**	†**30.5**	†**30.5**	*17.4*	24.9
	RotatE	*320.8*	319.2	152.6	**136.2**	*21.4*	21.5	25.1	**27.6**
	TransE	138.1	138.1	**90.4**	*139.8*	50.1	50.2	*44.7*	†**53.1**
DBP-YG	ComplEx	*2512.8*	*2512.8*	710.8	**701.7**	*17.3*	*17.3*	19.5	**20.7**
	DistMult	*2826.1*	*2826.1*	845.5	**851.3**	*11.5*	*11.5*	13.9	†**14.1**
	RotatE	*1568.5*	1551.4	559.8	**332.4**	*21.1*	21.3	26.1	**27.4**
	TransE	*1345.2*	1345.1	427.6	**431.3**	*21.3*	*21.3*	26.7	†**26.7**

Table 6
Integration performance of heterogeneous, pre-trained embedding spaces. Results marked with † are not significantly better than the runner-up (Wilcoxon test at $p = 0.05$). The best performance is marked in bold font, the worst one is italicized. Proj: Projection, Proj+Ortho: Orthogonal Projection, Latent+Ortho: Latent Orthogonal Projection.

Model	split15k-237		WD15k-237		DBP$_{\text{FR-EN}}$		DBP$_{\text{JA-EN}}$		DBP$_{\text{ZH-EN}}$		DBP-YG	
	MR	HITS	MR	HITS	MR	HITS	MR	HITS	MR	HITS	MR	HITS
TransE vs. RotatE												
Proj	33.5	63.6	139.7	46.1	356.8	27.9	380.6	19.9	311.4	23.3	4491.5	8.4
Proj+Ortho	*33.7*	*63.4*	*141.9*	*45.9*	*454.5*	*24.1*	*473.8*	*17.6*	*400.3*	*20.4*	*6102.5*	*4.4*
Latent+Ortho	25.5	*63.4*	67.4	46.8	251.0	33.8	†**183.1**	†**27.6**	†**170.9**	†**30.9**	1144.7	15.9
FedCoder	**19.0**	**71.4**	**45.0**	**54.7**	**131.3**	**41.9**	236.0	25.7	205.9	30.4	**586.4**	**19.9**
TransE vs. DistMult												
Proj	57.7	43.4	242.7	34.5	322.8	38.6	*465.2*	25.9	419.8	30.9	*2817.1*	*11.5*
Proj+Ortho	*58.8*	42.8	*249.0*	*33.9*	*433.5*	*29.9*	465.0	25.9	*419.9*	31.0	*2817.1*	*11.5*
Latent+Ortho	49.8	*42.2*	94.8	35.8	246.7	31.2	450.1	*15.7*	385.6	*25.2*	909.0	†**12.8**
FedCoder	**23.6**	**57.9**	**57.2**	**44.0**	**128.9**	**51.0**	**331.3**	**31.2**	**292.7**	**36.5**	†**908.2**	†**12.8**
TransE vs. ComplEx												
Proj	58.3	49.0	268.6	32.2	331.3	37.2	458.7	26.8	458.9	28.5	*2665.5*	*14.8*
Proj+Ortho	*59.5*	*48.5*	*323.0*	30.9	331.4	37.2	458.5	26.8	*459.0*	28.4	2664.6	14.9
Latent+Ortho	40.2	50.7	111.8	*28.6*	*495.6*	*19.6*	*677.0*	*12.4*	409.2	*20.7*	†**820.9**	15.6
FedCoder	**21.9**	**64.5**	**78.6**	**34.5**	**170.3**	**46.3**	**340.2**	**31.6**	**338.7**	**32.8**	821.2	†**15.7**
DistMult vs. ComplEx												
Proj	43.4	48.9	260.0	31.9	457.6	29.4	606.8	†**20.4**	520.7	†**27.9**	*2667.3*	*14.3*
Proj+Ortho	43.8	48.6	*379.6*	*28.5*	*574.7*	*26.5*	606.8	†**20.4**	520.9	†**27.9**	2667.1	*14.3*
Latent+Ortho	*65.0*	*25.7*	108.9	34.2	261.2	28.8	*799.6*	*8.0*	*614.7*	*13.8*	†**774.4**	16.6
FedCoder	**19.4**	**61.4**	**58.9**	**41.8**	**168.7**	**32.7**	**587.5**	15.0	**495.8**	21.0	778.8	**17.3**
RotatE vs. ComplEx												
Proj	*54.7*	46.9	260.0	39.9	378.4	27.4	686.2	14.5	557.9	19.3	4257.3	9.0
Proj+Ortho	54.3	46.9	*282.0*	*37.7*	*454.8*	*26.3*	*734.3*	*13.9*	*610.6*	*18.8*	*5474.5*	*6.6*
Latent+Ortho	39.5	*43.8*	72.7	38.1	232.4	27.8	†**423.6**	†**18.7**	317.0	†**24.2**	1259.8	14.7
FedCoder	**19.3**	**63.8**	**53.8**	**48.0**	†**179.9**	†**28.5**	470.4	16.8	**274.3**	24.0	**588.8**	**21.7**
RotatE vs. DistMult												
Proj	48.1	47.2	*170.3*	*41.9*	373.2	*29.9*	631.1	14.6	543.0	20.9	4355.2	7.7
Proj+Ortho	48.2	47.5	169.7	42.2	*400.8*	30.1	*681.7*	*14.1*	*589.1*	*20.2*	*5644.3*	*5.4*
Latent+Ortho	*50.9*	*34.4*	49.2	49.8	134.8	34.7	†**334.9**	†**18.8**	†**280.4**	†**26.0**	1202.8	13.6
FedCoder	**24.0**	**58.9**	**39.8**	**57.1**	**114.3**	**37.3**	339.3	17.6	305.2	24.6	**613.4**	**19.7**

of the two homogeneous cases: e.g., on WD15k-237, FedCoder shows a HITS of 63 for TransE and a HITS of 48 for RotatE in Table 5, whereas mixing these two models produces a HITS of 54. Meaning that the heterogeneous setting produces an average but also robust performance.

There is no evidence that suggests that embedding model combinations that have some commonality perform better than those with no shared trait. The latter is represented by the TransE-ComplEx and RotatE-DistMult combinations, which perform worst in two out of six datasets. Since they also represent two out of six embedding model combinations, this observation gives no indication that they have a disadvantage over other embedding model combinations.

5.3. *Joint embedding*

We next evaluate FedCoder in the setting where two embedding spaces are learned jointly (see Eqs. (1) and (13)). This approach is popular for KG alignment approaches as we would expect the best integration results since both embedding models can incorporate information from the other while constructing the embedding space. However, it would not be feasible in a distributed or large-scale setting since we need to embed all KGs.

We contrast FedCoder to the three most widespread integration models for joint embedding: The Identity model (Eq. (5)) [4–6], the Translation model (Eq. (6)) [5,6], and the Projection model (Eq. (7)) [4–6]. In this setting, we separate the discussion of heterogeneous embedding models and embedding dimensions since the first two baselines assume matching dimensions. Furthermore, due to the extensive computational effort, we limit our experiments to the split15k-237, WD15k-237, and DBP-YG datasets.

In the following, we first present the experimental setup, then compare the integration models in the homogeneous case, and finally discuss heterogeneity in either the embedding model or the embedding dimension.

5.3.1. *Setup*

We perform a grid search on FB15k-237 to determine the ideal margin $\gamma \in \{6, 12, 24, 32, 64\}$, and in consequence employ $\gamma_1 = 24$. Furthermore, we use 16 negative samples, an embedding dimension $k = k_0 = k_1 = 100$, and a batch size of 2000.

For FedCoder, we use $k_Z = 50$ and an encoder with two layers of dimensions $(k_i \times 100), (100 \times 50)$ and a decoder with a single layer of size $(50 \times k_i)$. The encoders for both graphs share their second-layer weights in order to reduce the number of parameters and prevent overfitting. While we experimented with more layers, this has not shown a significant improvement. All layers use the linear activation function, and we include a bias on all layers and initialize their weights with random orthogonal matrices [39]. As distance function in FedCoder's integration loss, we use the Euclidean and cosine distance and present our results for both. Lastly, we conduct a search for the regularizer λ in $\{1, 10, 20, 30\}$, with $\lambda = 20$ producing the best results. We train each model five times for a maximum of 1000 epochs with the Adam optimizer, whereas we evaluate it every 100 epochs and stop the training when the performance decreases two consecutive times. In most cases, the maximum is not exceeded.

5.3.2. *Homogeneous models*

In the homogeneous setting, the same embedding model is applied to both KGs of a dataset. The results of our experiment in this setting are presented in Table 7. It summarizes the best alignment performances in terms of the mean rank (MR) and HITS@10 achieved by any of the baseline models (Identity, Translation, Projection) versus the two FedCoder variants with the Euclidean or cosine distance. We hypothesize that FedCoder reaches a higher alignment performance than the baselines. The intuition is that the performance of an alignment model increases with the number of parameters it contains, e.g., we would also expect that the Projection method ($\mathcal{O}(k_0 k_1)$ parameters) outperforms Identity (no parameters).

Table 7 shows that this intuition does not hold, as the Euclidean model outperforms the Projection model in most cases. However, there is no single alignment model that performs best. In terms of mean rank, FedCoder shows a stronger overall performance with the exception of RotatE. The HITS@10 measure gives mixed results, both in which alignment model performs best as well as in how big the difference between embedding models is. This can most prominently be seen in the split15k-237 dataset, where FedCoder outperforms the baselines by one order of magnitude with DistMult while reaching only about two-thirds of Identity's performance on RotatE. The difference in HITS@10 between the baselines and FedCoder decreases with the size of the dataset (DBP-YG includes more than ten times more entities than split15k-237). As we used an embedding dimension of $k = k_0 = k_1 = 100$ on all datasets, this might indicate that alignment models with more parameters like ours or the Projection model overfit. While we cannot claim that FedCoder strictly outperforms the baseline, our model delivers competitive and robust results across all embedding models and datasets, and it is important to note that the results in Table 7 clearly show that FedCoder's performance is never the worst one.

5.3.3. *Heterogeneous models*

The results of the heterogeneous setting, where KGs are embedded with different models, are shown in Table 8. It groups the KG embedding combinations into six blocks: The first row of blocks contains models that fall into the same category (geometric or matrix factorization), the model combinations in the second row use the same numeric space, and those in the third row or have no commonality. The table lists the link prediction performance as mean rank (MR) and HITS@10 (HITS) for all baselines as well as the two FedCoder variants. In the following, we first compare the alignment models on the hypothesis that FedCoder outperforms the baselines. Second, we contrast heterogeneous embedding models with the homogeneous setting. Last, we hypothesize that embedding model combinations perform better if they have some commonality, e.g., use the same numerical space.

Table 8 shows that FedCoder yields substantially higher MR and HITS@10 scores than the baselines in almost all settings and datasets, with the exception of the MR of TransE combined with RotatE on split15k-237 and DBP-YG. In particular, FedCoder with the cosine distance shows the best overall performance, outperforming baselines up to an order of magnitude and featuring the lowest variance across the results in both MR and HITS@10.

Comparing the heterogeneous setting to the homogeneous one makes it apparent that the former is more challenging, resulting in worse MR and HITS@10 scores. However, the difference between the two settings is lower for the FedCoder variants than the baselines. Hence, FedCoder is less affected by heterogeneity. Furthermore, in two-thirds of the cases, FedCoder's heterogeneous performance lies in between the performance of the respective embedding models in the homogeneous setting. For example, on DBP-YG, FedCoder achieves an MR value of 108.4 for the combination of TransE and DistMult, while in the homogeneous scenario, TransE achieves an MR of 58.3, but DistMult only produces an MR of 428.1. Meaning that with heterogeneous embedding models, the stronger model can compensate for the weaker.

Table 7
Integration performance of homogeneous, jointly learned embedding spaces. Results marked with † are not significantly better than the runner-up (Wilcoxon test at $p = 0.05$). The best performance is marked in bold font, the worst one is italicized.

Dataset	Model	MR					HITS@10				
		Identity	Translation	Projection	FedCoder (Euclid)	FedCoder (Cosine)	Identity	Translation	Projection	FedCoder (Euclid)	FedCoder (Cosine)
split15k-237	ComplEx	1308.5	*1702.0*	1570.5	985.2	**480.3**	4.3	*0.9*	2.7	19.7	**28.7**
	DistMult	1554.7	*1932.9*	1689.2	1100.0	**444.6**	3.1	*0.4*	2.1	17.7	**29.4**
	RotatE	†**149.6**	158.4	172.9	240.7	*248.9*	†**71.7**	70.4	57.1	*46.1*	47.3
	TransE	99.7	100.5	*101.0*	94.1	**51.0**	72.5	†**73.0**	71.3	*70.8*	71.2
WD15k-237	ComplEx	180.9	*375.2*	252.9	205.6	**142.6**	**54.4**	*20.1*	38.4	43.6	39.7
	DistMult	189.5	*455.2*	272.3	200.5	**157.9**	**53.4**	*15.1*	34.8	47.3	38.1
	RotatE	56.4	56.0	*59.7*	**46.9**	54.3	81.9	†**82.1**	79.1	75.3	*71.2*
	TransE	40.9	41.0	*42.2*	41.9	**27.7**	81.4	81.5	*80.0*	80.1	†**82.6**
DBP-YG	ComplEx	739.5	*2063.1*	1986.8	656.6	**208.5**	53.9	32.8	*26.6*	55.5	**65.5**
	DistMult	953.0	2274.0	*2630.3*	845.7	**428.1**	†**50.7**	28.8	*18.2*	50.0	49.2
	RotatE	44.8	**39.5**	44.7	*100.6*	66.3	82.2	†**83.1**	81.1	*63.8*	74.4
	TransE	†**58.3**	58.4	59.8	*287.8*	†**58.3**	68.1	68.1	66.3	*55.0*	†**70.8**

Table 8
Integration performance of jointly learned embedding spaces with heterogeneous embedding models. Results marked with † are not significantly better than the runner-up (Wilcoxon test at $p = 0.05$). The best performance is marked in bold font, the worst one is italicized.

Model	split15k-237		WD15k-237		DBP-YG		split15k-237		WD15k-237		DBP-YG	
	MR	HITS	MR	HITS	MR	HITS	MR	HITS	MR	HITS	MR	HITS
	TransE vs. RotatE						DistMult vs. ComplEx					
Identity	214.1	43.6	50.5	73.9	65.5	65.8	1727.5	2.2	241.2	41.0	1443.0	34.9
Translation	*250.0*	*39.7*	*53.1*	*73.6*	60.4	67.2	*1960.9*	*0.6*	*491.9*	*11.2*	*3145.2*	20.1
Projection	202.9	44.9	50.3	76.6	**51.4**	72.2	1726.4	1.8	348.7	26.3	3055.3	*13.7*
FedCoder (Euclid)	†**202.4**	**50.9**	40.6	78.9	*170.7*	*66.0*	1103.5	17.0	210.6	**44.6**	716.9	†**57.4**
FedCoder (Cosine)	243.1	48.1	**36.1**	†**79.7**	57.1	†**72.4**	**518.9**	**26.6**	**165.6**	35.8	**380.1**	56.0
	TransE vs. DistMult						RotatE vs. ComplEx					
Identity	*1574.4*	8.9	522.9	19.8	*3466.2*	32.6	*1231.8*	25.1	540.9	25.3	4212.6	*21.8*
Translation	1520.0	8.6	*675.7*	*10.9*	3391.9	*25.7*	1025.4	33.5	*616.3*	*19.5*	*2886.2*	32.4
Projection	1214.8	*7.8*	311.1	36.4	1226.0	43.7	885.1	*18.4*	265.4	45.9	1099.2	49.6
FedCoder (Euclid)	1039.6	22.9	308.6	33.4	332.7	59.9	748.2	29.8	272.9	41.0	1025.0	46.2
FedCoder (Cosine)	**367.4**	**42.5**	**91.4**	**63.7**	**108.4**	**70.1**	**280.9**	**47.6**	**93.5**	**64.3**	**172.6**	**66.7**
	TransE vs. ComplEx						RotatE vs. DistMult					
Identity	*1638.7*	7.2	569.8	16.5	*3422.5*	28.4	*1426.5*	19.1	534.9	26.0	3438.0	29.3
Translation	1584.2	6.7	*657.0*	*11.1*	2772.4	*28.1*	1275.7	24.1	*659.3*	*15.8*	*4444.3*	*19.4*
Projection	1399.2	*6.1*	364.8	30.4	2070.9	35.2	1053.1	*12.6*	307.4	37.1	1348.6	39.6
FedCoder (Euclid)	1211.1	18.5	359.1	27.6	373.4	56.9	725.2	30.8	242.5	42.8	884.3	52.2
FedCoder (Cosine)	**391.5**	**36.9**	**100.8**	**57.6**	**119.1**	**67.7**	**297.9**	**44.3**	**93.4**	**64.5**	**185.2**	**66.1**

Contrasting the three groups of embedding model combinations shows that combining a matrix factorization model (DistMult, ComplEx) with a geometric model (TransE, RotatE) produces a better result than combining the two matrix factorization models, however, this does not hold the other way round. Furthermore, mixing embedding models that use the same numerical space (real or complex) achieves better scores than if they have no commonality. Both observations hold for FedCoder as well as the baseline models, however, they are more pronounced for the latter.

5.3.4. Heterogeneous hyperparameters

Table 9 shows show the alignment performance in the case where different embedding dimensions are employed on the two KGs of each dataset. In this case, only the Projection baseline is applicable, but not the Identity or Translation baselines, as they require the embedding dimensions to match. For this experiment, we first train every embedding model on each KG of our datasets with the embedding dimension varying in $\{50, 100, 200, 500\}$ and identify which of those produces the highest link prediction performance. We then use the ideal dimensions, reported as k_0/k_1 in Table 9 to train the KG embedding models jointly in the alignment setting. We first hypothesize that FedCoder achieves a higher performance than the Projection baseline, and second that the alignment with heterogeneous embedding dimensions outperforms the homogeneous case.

It is apparent from Table 9 that FedCoder achieves a substantially better MR and HITS@10 than the Projection baseline in all but two cases except one (RotatE and TransE on DBP-YG). Comparing Tables 7 and 9 further shows that in most cases, FedCoder achieves a higher performance in MR and HITS@10 when the embedding dimension is tuned to the dataset. This is in contrast to the Projection approach, where typically a better performance is achieved in the homogeneous scenario, with the exception of TransE on DBP-YG.

6. Evaluation #2: How do different embedding space integration models perform in the presence of multiple KGs?

The Web of Embedding is large: The Linked Open Data Cloud currently includes more than 1300 datasets, and although not all of these resources have been embedded, we have seen continued interest in KG embedding and steady growth in the number of published embedding spaces. Contrarily, previous works on embedding space integration have either produced general-purpose methods to combine two spaces or tailored their models to a fixed number of specific sources. For the Web of Embeddings, however, we require generic methods that can integrate an arbitrary and potentially large number of embedding spaces. Hence, it is elementary to ask *how do different embedding space integration models perform in the presence of multiple KGs*?

Table 9
Integration performance of jointly learned embedding spaces with heterogeneous embedding dimensions, k_0 and k_1. Results marked with † are not significantly better than the runner-up (Wilcoxon test at $p = 0.05$). The best performance is marked in bold font, the worst one is italicized.

Dataset	Model	k_0/k_1	MR			HITS@10		
			Projection	FedCoder (Euclid)	FedCoder (Cosine)	Projection	FedCoder (Euclid)	FedCoder (Cosine)
split15k-237	ComplEx	100/100	*1570.5*	985.2	**480.3**	*2.7*	19.7	**28.7**
	DistMult	100/200	*1944.0*	923.8	**430.7**	*0.5*	22.4	**32.4**
	RotatE	200/100	*140.1*	108.7	**72.1**	*46.8*	57.9	**63.8**
	TransE	200/500	*227.9*	93.4	**49.9**	*36.0*	64.5	**72.9**
WD15k-237	ComplEx	50/500	*582.0*	169.1	**117.3**	*17.0*	**52.1**	47.2
	DistMult	50/500	*592.1*	174.5	**129.7**	*17.5*	**52.2**	38.1
	RotatE	100/200	*63.2*	**45.1**	55.1	*72.3*	72.8	**76.7**
	TransE	100/200	*51.2*	44.9	**33.1**	*70.9*	75.9	**80.4**
DBP-YG	ComplEx	100/500	*7544.6*	589.3	**157.8**	*3.1*	56.6	**62.7**
	DistMult	200/500	*5918.6*	575.5	**196.4**	*6.8*	57.2	**60.6**
	RotatE	100/100	**44.7**	*100.6*	66.3	**81.1**	*63.8*	74.4
	TransE	100/50	**49.4**	*329.6*	59.0	**73.0**	*54.1*	63.5

Procedure. We answer this question in two parts: First, we analyze the computational complexity of different integration models with respect to the number of involved embedding spaces. In the presence of numerous KGs, efficiency becomes paramount to not putting a limit on how many graphs can be integrated. Second, we evaluate the integration models on their ability to learn from all given embedding spaces. Integrating multiple embeddings simultaneously can, on the one hand, benefit from more comprehensive knowledge, on the other hand also poses a more difficult problem. For both parts, we contrast FedCoder with the pairwise projection integration model and the latent orthogonal projection approach.

Evaluation structure. In the following, we first briefly describe the datasets on which we conduct our experiment. We then discuss the computational complexity of the three approaches, both theoretically and experimentally. Finally, we present our experiment on the integration performance of the different methods on multiple embedding spaces.

6.1. Datasets

We consider two datasets for our experiments, summarized in Table 10. Both datasets are composed of several graphs and differ in their sizes and how entities are linked between graphs.

Fb5. We use FB15k-237, a graph popular in KG embedding literature, to construct a set of five partially linked KGs [37]. To create each sub-graph, we randomly sample 30% from all of FB15k-237's entities, then add all triples for which we sampled both their head and tail entity. As entities that do not occur in any triple are discarded, the effective number of entities in the sub-graph will be lower than 30%. Note that in this procedure, we do not control for the overlap between two graphs.

Dbp15k. We further use the DBP15k dataset collection as it already comes with three datasets, each including two linked KGs [38]. DBP15k was sampled from DBpedia in different languages and pairs a KG from the English version with a graph from either the Chinese, the French, or the Japanese DBpedia. The English KGs are all different from each other hence each constitutes a distinct KG, but they can be trivially aligned via the entity URIs.

6.2. Complexity analysis

We first analyze the complexity of integration models with respect to the number of embedding spaces to be integrated. Two integration paradigms were presented in this paper: The more common pairwise approach, where one embedding space is mapped onto another, and the latent space approach, where embedding spaces are transformed into a common representation. In the following, we first compare these integration methods theoretically and then present an empirical evaluation.

6.2.1. Theoretical analysis

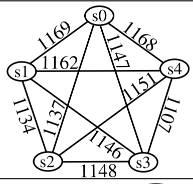
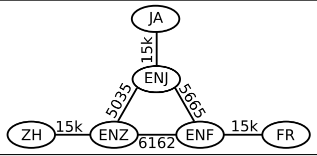
Typically, we face a scenario where we have N embedding spaces and a set of anchors $\mathcal{A}$ that link entity embeddings between them. Pairwise integration approaches (Eq. (4)) learn a mapping from a source to a target space, such that embeddings from both spaces can be compared in the target space. To compare embedding vectors between any two spaces, they hence have to compute a mapping between each pairwise space combination. Given N spaces, there are $\frac{N(N-1)}{2}$ possible combinations of spaces, assuming that no space is mapped onto itself and that we only need to compute a mapping in one direction for two given graphs, i.e., we only need to map the source onto the target but not vice-versa. The computational complexity of pairwise approaches is therefore $\mathcal{O}(N^2)$.

Latent space integration methods (Eq. (9)) learn a mapping from each embedding space into a latent space. For N graphs, this results in N mappings to be learned. To compare entity vectors in any given space, these methods may also compute a mapping from the latent space back into the embedding spaces, i.e., another N mappings. Hence, the computational complexity of latent-space approaches is $\mathcal{O}(2N)$.

To compute a mapping, we assume the availability of anchors, i.e., pairs of linked entities. All anchors are consumed at least once when integrating the embedding spaces. It can be expected that a larger number of graphs is accompanied by a larger number of anchors. However, this holds for both pairwise and latent-space approaches, in the same manner, thus does not lead to a difference in their complexity. Clearly, latent space methods scale better than pairwise methods.

Finally, we argue that learning an embedding space is more computationally expensive than learning a mapping between given embedding spaces. The complexity of computing KG embedding is proportional to the number of triples in the graph since an embedding model must consume the entire graph at least once, i.e., $\mathcal{O}(|\mathcal{E}|)$. The complexity of an integration model is proportional to the number of entities since every entity occurs at most once in the, i.e., $\mathcal{O}(|\mathcal{V}|)$. There are always more triples than entities because every entity must appear in at least one triple, i.e., $|\mathcal{V}| \leq |\mathcal{E}|$. Typically, the number of triples exceeds the number of entities, e.g., in the datasets we used throughout this paper, there are about 6.6 triples per entity. The complexity of an integration model is therefore substantially lower than that of an embedding model. The consequence of this is that if pre-trained embeddings are available, they should be integrated directly instead of jointly embedding the KGs a second time.

Table 10
Dataset characteristics, with the number of anchors specified for each pair of KGs.

Dataset	KG	Entities	Relations	Triples	Anchors
FB5	sub0	4169	222	28409	
	sub1	4086	219	28111	
	sub2	4130	220	28864	
	sub3	4096	201	27281	
	sub4	4154	221	28096	
DBP15k	ZH	19388	1701	70414	
	ENZ	19572	1323	95142	
	FR	19661	903	105998	
	ENF	19993	1208	115722	
	JA	19814	1299	77214	
	ENJ	19780	1153	93484	

6.2.2. Empirical analysis

We aim to confirm our theoretical results empirically by measuring the training time of different integration approaches on FB5. From the five graphs available in the dataset, we create all possible combinations that include exactly N graphs for all $N \in [2, 5]$. For each of these cases, we then integrate the respective TransE embedding spaces with the pairwise projection method, the latent space orthogonal projection approach, and FedCoder with a single layer and the linear activation function. The embedding dimension is 256, which we also use for the size of the latent space. We measure the time spent in training, disregarding time to set up, evaluate, or perform other tasks. The test is conducted with our PyTorch-based implementation on one Nvidia RTX 2080-TI GPU and one Intel Xeon 6230 CPU. We train for a maximum of 5000 epochs, repeat the experiment five times, and report averaged results.

Fig. 3 reports the training time in seconds, averaged over all combinations of spaces for a given N. It contrasts the pairwise projection method with the two latent space methods. We report the number of seconds to train for 1000 epochs, whereas one epoch consumes the entire training data once. Fig. 3 clearly mirrors the insights from the theoretical analysis: The two shared space methods scale linearly, while the pairwise method exhibits N-squared behavior. We also see that the latent orthogonal projection method is slightly more efficient than FedCoder. This is because FedCoder uses the entity embeddings of each space to train the latent space in the autoencoder fashion, while the orthogonal regularizer is independent of the number of entities in the graphs.

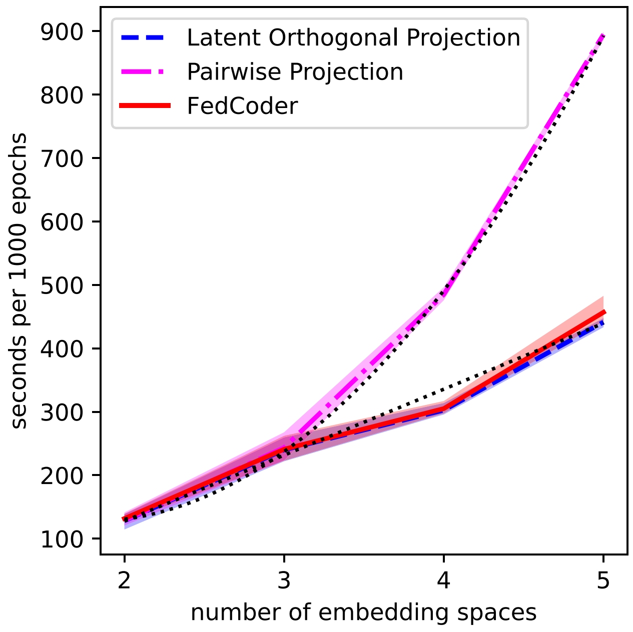


Fig. 3. Training time of pairwise and latent-space integration models, with respect to the number of integrated embedding spaces from FB5. The shading indicates the lowest and highest measured time of each method per N. The black dotted lines are linear and cubic functions fitted to the latent-space and pairwise methods, respectively.

6.3. Integration performance

Having seen that latent space methods can integrate multiple embedding spaces efficiently, we next evaluate their integration performance dependent on the number of spaces that are integrated simultaneously. One could reason that more KGs means more knowledge about an entity, which would lead to a better integration. On the other hand, additional spaces could also introduce more noise into the learning problem, decreasing the integration performance. In this section, we hence elaborate on the integration performance dependent on the number of graphs.

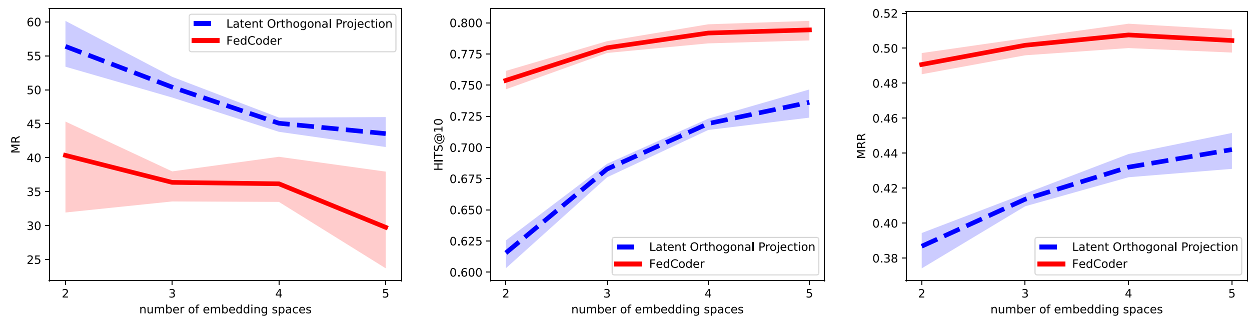
The goal of embedding space integration is to be able to compare embedding vectors of linked entities to one another, i.e., linked entities should have a similar representation in the latent space. As in Section 5, this notion is related to the link prediction task in embedding spaces, hence we, again, adopt their standard measures to quantify the integration performance: The mean rank (MR), HITS@10, and the mean reciprocal rank (MRR).

In the following, we first describe the experimental setup, then present our findings.

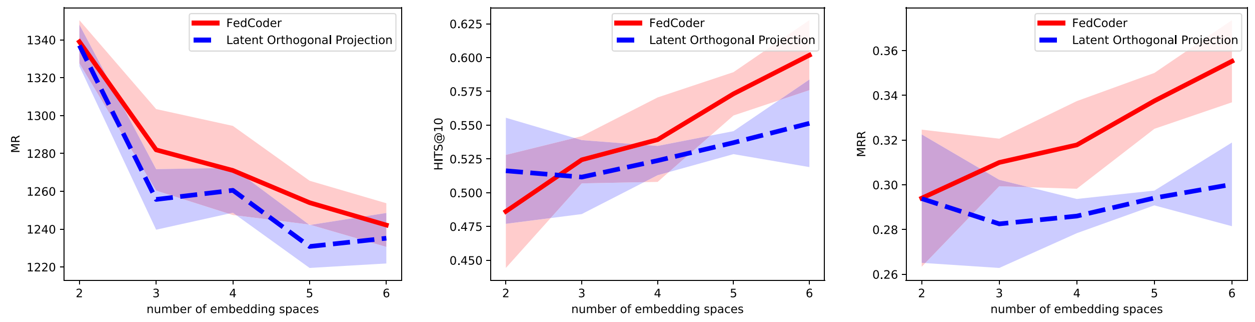
6.3.1. Setup

To evaluate the integration performance for an increasing number of embedding spaces, we build all possible combinations of N graphs from the respective dataset, whereas we increase N from two to the number of graphs in the dataset. For each combination of N graphs, we select two graphs on which we evaluate the integration performance. For those two, we randomly sample 20% of their anchors to be reserved for testing. The remaining anchors and all anchors between the other graphs can, in principle, be used for training. However, two testing entities may be linked implicitly via an intermediate graph, which, assuming that KG alignment is transitive, would leak information about the testing set. In such cases, we disregard the link to either of the testing graphs, i.e., we ensure that at most one of the two testing graphs is in the transitive closure of each testing entity. We select each pair out of the N graphs once for testing and create five random train-test splits in all of those cases.

To train the integration models, we re-use the TransE embeddings from Section 5.2 that were independently learned for each KG with LibKGE. We train each integration model with a learning



(a) Integration performance on the FB5 dataset



(b) Integration performance on the DBP15k dataset

Fig. 4. Integration performance by the number of jointly integrated embedding spaces in MR, HITS@10, and MRR (from left to right). The lines represent the average performance, the shading indicates the standard deviation.

rate of 0.01 for a maximum of 5000 epochs or until convergence. For both latent space methods, we set the margin to $\gamma = 32$ and use 16 negative samples. The mapping functions in FedCoder are single-layer networks with the linear activation function, and all projection and weight matrices are initialized with the identity matrix. We perform a hyperparameter search to determine the size of the latent space k_Z and the regularization coefficient λ for both datasets. On FB5, we thus set $k_Z = 64$ and $\lambda = 0.01$, and on DBP15k, we use $k_Z = 256$ and $\lambda = 0.001$.

6.3.2. Results

In this experiment, we evaluate the integration performance of the FedCoder and the Latent Orthogonal Projection model for an increasing number of integrated embedding spaces. We hypothesize that the integration performance increases with the number of involved embedding spaces. We further presume that FedCoder performs better in this setting than the Latent Orthogonal Projection baseline.

Fig. 4 shows the integration performance in terms of mean rank (MR), HITS@10, and the mean reciprocal rank (MRR), with respect to the number of graphs N that are jointly integrated. All measures are averaged over all combinations of graphs for a particular N, all of their respectively selected pair of testing graphs, and the five train-test splits of all those cases. The abscissa of all figures, indicating the number of integrated embedding spaces (N), starts at two since we at least need two spaces to integrate. Each plot contrasts the performance of FedCoder and the Latent Orthogonal Projection method.

Fig. 4 shows that the integration performance of both latent spaces improves with an increasing number of integrated spaces in the mean rank (MR) and HITS@10. Despite the general improvement with more spaces, the MRR of FedCoder slightly decreases with the last space on the FB5 dataset. The source of this effect can be found in a slight decrease of HITS@1 as the number of embedding spaces increases. So whilst the top 10 results are improving (see HITS@10), the very top one becomes less precise.

FedCoder typically performs better than the Latent Orthogonal Projection model, especially when many embedding spaces are integrated. While both models deliver comparable performance on the DBP15k dataset, FedCoder improves more constantly and to a higher performance with more spaces being integrated. Its higher HITS@10 and MRR show that it performs better towards the top of the rank distribution, while the slightly worse MR means that this comes at the cost of losing precision on the tail end of that distribution.

7. Discussion & limitations

In this section, we first summarize our findings and discuss them in the light of the Web of Embeddings, then highlight some limitations.

7.1. Summary and discussion of results

We identified two key challenges that we face in a Web of Embeddings: The heterogeneity of embedding models that we want to integrate and the need for integrating multiple embedding spaces. We addressed both challenges in this paper in Section 5 and Section 6, respectively.

Heterogeneous and homogeneous embeddings. Concerning heterogeneity, we discussed the scenario where embedding spaces are pre-trained and the scenario where embeddings are learned jointly. In both scenarios, we evaluated the integration performance of FedCoder and its viable baselines in the homogeneous and the heterogeneous case. We found that *in the heterogeneous setting, FedCoder outperforms the state-of-the-art in almost all cases*, and in many, its advantage is substantial. In a few cases, FedCoder ties with the Latent Orthogonal Projection model, indicating that latent-space approaches are preferable over pairwise methods in our setting. For the Web of Embeddings, this is a promising result since latent-space methods also scale better than pairwise ones.

In the homogeneous setting, FedCoder delivers competitive performance to the respective baselines. It outperforms the state-of-the-art in many scenarios, and while it may not always be the best choice, it is typically also not the worst one. In particular, even when FedCoder has a lower HITS score than other methods, it often achieves a better mean rank, meaning that it produces a better average performance at the cost of a reduced precision in the top ranks.

Our results hold in both the joint embedding and mapping-only scenarios. Contrasting these two settings, we find that *joint embedding reaches a better integration performance on the larger and sparser datasets, whereas mapping of pre-trained embedding spaces works better on smaller to mid-size graphs.* This behavior was expected since joint embedding methods can use the graph alignment as additional information when embedding, which can have a large impact, especially on entities with few neighbors in each graph. While re-using pre-trained embeddings is the more efficient — and for a true Web of Embeddings the only practical — approach, it shows that this comes at a price as it typically achieves a reasonable yet not ideal integration.

Multiple embedding spaces. Concerning the integration of multiple embedding spaces, we analyzed the computational complexity of latent-space methods and compared the integration performance of latent-space approaches with regard to an increasing number of integrated spaces. We theoretically and empirically demonstrated that *pairwise integration methods — unsurprisingly — have a complexity of $\mathcal{O}(N^2)$, while latent-space methods only have a complexity of $\mathcal{O}(2N)$.* We suspect that it will be difficult to decrease the complexity much further without additional assumptions, as an integration method has to consider each embedding space at least once. This is a promising result that suggests that we should shift from the predominantly used pairwise approaches to latent-space methods when creating the Web of Embeddings. We have also shown that *it can be beneficial to integrate multiple embedding spaces jointly, as both latent-space methods improve when faced with more spaces.* For the Web of Embeddings, this means that embedding space integration produces not only a more comprehensive space but also a more consistent one.

7.2. Limitations and future work

In the following, we present a number of limitations and outline possible areas for future work.

Source-specific mapping functions. While FedCoder would admit arbitrary mapping functions, we only experimented with relatively simple neural networks composed of one or two layers and a linear activation function. The benefit of such simple functions is that they can be trained efficiently. Over the past years, however, there has been tremendous progress in the development of novel and intricate neural architectures, such as graph convolutional networks. It remains to be seen in which contexts such methods will be useful. Moreover, we use the same mapping function on all embedding spaces, although this is not a requirement in FedCoder. Mapping functions that are tailored to the properties of each embedding space — using knowledge about the graph or embedding model used — could possibly lead to a substantial improvement in the integration performance. The challenge in both of these possible future directions is overfitting: We have observed that already our simple mapping functions may overfit, and the risk thereof greatly increases when more complex functions are used. It could thus also be interesting to regularize the mapping functions and the latent space stronger to prevent such an outcome.

Embedding models. We evaluated FedCoder and its baselines on four popular KG embedding models, but since numerous alternatives exist, one might ask if our findings generalize to other embedding model choices [1,23]. We assumed that a KG embedding model represents each entity by one embedding vector. Most KG embedding models satisfy this constraint, and we expect no conceptual complications in these cases. More challenging would be to incorporate models like BoxE [40] that represent entities by multiple embedding vectors. The straightforward approach for the baseline methods to cope with this is concatenating all vectors of one entity. However, this loses the semantics of the different vectors. For FedCoder, we could instead devise a mapping function that learns how to combine all embedding vectors of an entity into a single representation more effectively. In either case, an integration method needs not to be aware of how an embedding space was constructed, i.e., the embedding model's triple score function, as apparent in our experiments on pre-trained embedding spaces. However, the integration performance depends on how easily one embedding space can be mapped onto another. In particular, deep learning models like ConvE [28] that include non-linearity might produce an embedding space that is vastly different from spaces of linear methods like TransE. In this scenario, we would expect more expressive integration models like FedCoder and the projection-based ones to yield a substantially better performance than simpler models. In particular, as FedCoder can operate on arbitrary mapping functions, we could adjust them to the increased complexity, e.g., by using non-linear activation or deeper networks.

Multi-modal embedding spaces. While we focused on KG embedding spaces, it would be of great benefit to the Web of Embeddings to integrate not only KG embedding spaces but also embeddings of other modalities like text or images. Such an endeavor would be particularly interesting since we could combine these implicitly structured data sources with structured, curated, and semantic KGs.

Data evolution. The Web of Embeddings is affected by two types of data evolution: Updates within a KG and changes in the composition of the KG ecosystem. The first type of data evolution concerns the KG embedding providers, as many KG embedding models have to be re-computed whenever the KG changes. The maintainer of a quickly evolving graph can solve this issue of high reincurring computational costs by resorting to embedding models that were proposed for this particular scenario [41,42]. The second type of data evolution concerns the maintainer of the Web of Embeddings, who wants to integrate any newly emerged embedding space. In FedCoder, we can keep the integration costs low by taking advantage of pre-trained embedding spaces. However, even this approach might become impractical for many spaces or a fast-paced ecosystem. Instead, it would be preferable to integrate an additional embedding space into an existing, previously learned representation. A straightforward solution to this problem is to train only the new graph's mapping functions while keeping the previous latent representations fixed. Such an approach would be efficient, but the result may not optimal since the previous representations are not adapted to the new graph and would not profit from its added knowledge. To alleviate such effects, we could first train only the new graph's mappings, then fine-tune all latent representations and mapping functions together. Fundamentally, the required fine-tuning effort depends on whether or not FedCoder converges to a universal representation given enough KGs, which we leave for future work to explore.

Fine-tuning. We experimented with joint embedding and mapping of given embedding spaces, however, there is a middle ground: Pre-trained embedding spaces can be fine-tuned in the integration setting, i.e., trained for only a few epochs. As we have seen differences in performance between the two extremes, a fine-tuning approach could provide a reasonable compromise between efficiency and performance.

Evaluation datasets. Finally, as always in such investigations, the generalizability of our findings is limited by the underlying datasets. Whilst we did our utmost to collect a good set of datasets, going beyond two KGs — the usual number used in KG alignment competitions — proved to be very hard. We, therefore, think that a major building block for accelerating the research and practice of the Web of Embeddings would be to collect a large set of KGs and their alignments (e.g., by building on the work of [43,44]).

8. Conclusion

In this paper, we argue for building a Web of Embeddings: an ecosystem of KG embedding spaces as diverse and distributed as the Web of Data or the Semantic Web. From years of Semantic Web research, we know that it is desirable and also possible to achieve interoperability in such an environment. The main lessons from the Semantic Web are that such an effort will have to acknowledge local authority over the published content as well as have to be able to cope with an arbitrary number of KGs. Hence, any attempt to build the Web of Embeddings will have to accept that the published embedding spaces will follow whatever (embedding) procedure, use whatever hyperparameters, and exhibit whatever dimensionality that the publishers chose for their own goals. Furthermore, the Web of Embeddings will have to be able to integrate a large number of embedding spaces into a common representation or space.

Following the computer science 'mantra' that *"Any problem in computer science can be solved with another level of indirection"*,[9] we propose to map the individual KG embedding spaces into a joint Web of Embedding space. Specifically, addressing the above challenges, we presented FedCoder, which jointly integrates multiple heterogeneous KG embedding spaces via a latent space and autoencoders.

We established in a series of experiments that FedCoder substantially outperforms state-of-the-art baselines in a heterogeneous setting. Our results imply that each KG embedding provider can and should optimize for their own graphs. We further demonstrated that FedCoder scales linearly in the number of embedding spaces and improves with more integrated spaces, making it suitable to integrate even large numbers of embedding spaces. These results may also provide a first indication that the complexity of integrating many embedding spaces could actually benefit from doing so over an increasing number — an observation that seems to echo the findings of learning large-scale transformer models in the text and image processing domain.

In consequence, whilst many challenges remain, and the community needs to embark on an effort of large-scale experimentation, we believe that our highly flexible FedCoder approach can help break ground and serve as a foundation for a future Web of Embeddings that might significantly increase the usefulness of the Web of Data.

[9] See https://en.wikipedia.org/wiki/Butler_Lampson#Quotes.

CRediT authorship contribution statement

Matthias Baumgartner: Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Daniele Dell'Aglio:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Supervision. **Heiko Paulheim:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing. **Abraham Bernstein:** Conceptualization, Methodology, Resources, Writing – original draft, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the Swiss National Science Foundation for their partial support under contract number #407550_167177. We also thank Swiss Re's Advanced Analytics team for their support and practical exchange, Sven Hertling for a constructive discussion, and the anonymous reviewers for their insightful comments.

References

[1] Q. Wang, Z. Mao, B. Wang, L. Guo, Knowledge graph embedding: A survey of approaches and applications, IEEE Trans. Knowl. Data Eng. 29 (12) (2017) 2724–2743.
[2] M. Nickel, K. Murphy, V. Tresp, E. Gabrilovich, A review of relational machine learning for knowledge graphs, Proc. IEEE 104 (1) (2016) 11–33.
[3] M. Kejriwal, P.A. Szekely, Co-LOD: Continuous space linked open data, in: ISWC (Satellites), in: CEUR Workshop Proceedings, vol. 2456, CEUR-WS.org, 2019, pp. 333–337.
[4] Y. Hao, Y. Zhang, S. He, K. Liu, J. Zhao, A joint embedding method for entity alignment of knowledge bases, in: CCKS, in: Communications in Computer and Information Science, vol. 650, Springer, 2016, pp. 3–14.
[5] M. Chen, Y. Tian, M. Yang, C. Zaniolo, Multilingual knowledge graph embeddings for cross-lingual knowledge alignment, in: IJCAI, ijcai.org, 2017, pp. 1511–1517.
[6] W. Liu, J. Liu, M. Wu, S. Abbas, W. Hu, B. Wei, Q. Zheng, Representation learning over multiple knowledge graphs for knowledge graphs alignment, Neurocomputing 320 (2018) 12–24.
[7] Z. Sun, C. Wang, W. Hu, M. Chen, J. Dai, W. Zhang, Y. Qu, Knowledge graph alignment network with gated multi-hop neighborhood aggregation, in: AAAI, AAAI Press, 2020, pp. 222–229.
[8] J. Portisch, M. Hladik, H. Paulheim, Kgvec2go - knowledge graph embeddings as a service, in: LREC, European Language Resources Association, 2020, pp. 5641–5647.
[9] S. Broscheit, D. Ruffinelli, A. Kochsiek, P. Betz, R. Gemulla, LibKGE - a knowledge graph embedding library for reproducible research, in: EMNLP: System Demonstrations, 2020, pp. 165–174, URL https://www.aclweb.org/anthology/2020.emnlp-demos.22.
[10] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, A. Peysakhovich, Pytorch-BigGraph: A large scale graph embedding system, in: MLSys, mlsys.org, 2019.
[11] H. Zhu, R. Xie, Z. Liu, M. Sun, Iterative entity alignment via joint knowledge embeddings, in: IJCAI, ijcai.org, 2017, pp. 4258–4264.
[12] Z. Sun, W. Hu, Q. Zhang, Y. Qu, Bootstrapping entity alignment with knowledge graph embedding, in: IJCAI, ijcai.org, 2018, pp. 4396–4402.
[13] T. Berens-Lee, Semantic web concepts, 2015, URL http://www.w3.org/2005/Talks/0517-boit-tbl/.
[14] H. Bourlard, Y. Kamp, Auto-association by multilayer perceptrons and singular value decomposition, Biol. Cybernet. 59 (4) (1988) 291–294.
[15] I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, Deep Learning, Vol. 1, no. 2, MIT press Cambridge, 2016.
[16] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, Dbpedia - A crystallization point for the web of data, J. Web Semant. 7 (3) (2009) 154–165.
[17] H. Schwenk, M. Douze, Learning joint multilingual sentence representations with neural machine translation, in: Rep4NLP@ACL, Association for Computational Linguistics, 2017, pp. 157–167.

[18] A. Conneau, G. Lample, M. Ranzato, L. Denoyer, H. Jégou, Word translation without parallel data, 2017, CoRR abs/1710.04087.

[19] L.Y. Wu, A. Fisch, S. Chopra, K. Adams, A. Bordes, J. Weston, Starspace: Embed all the things!, in: AAAI, AAAI Press, 2018, pp. 5569–5577.

[20] Z. Wang, J. Zhang, J. Feng, Z. Chen, Knowledge graph and text jointly embedding, in: EMNLP, ACL, 2014, pp. 1591–1601.

[21] M. Baumgartner, W. Zhang, B. Paudel, D. Dell'Aglio, H. Chen, A. Bernstein, Aligning knowledge base and document embedding models using regularized multi-task learning, in: ISWC, in: Lecture Notes in Computer Science, vol. 11136, Springer, 2018, pp. 21–37.

[22] Q. Zhang, Z. Sun, W. Hu, M. Chen, L. Guo, Y. Qu, Multi-view knowledge graph embedding for entity alignment, in: IJCAI, ijcai.org, 2019, pp. 5429–5435.

[23] A. Rossi, D. Barbosa, D. Firmani, A. Matinata, P. Merialdo, Knowledge graph embedding for link prediction: A comparative analysis, ACM Trans. Knowl. Discov. Data 15 (2) (2021) 14:1–14:49.

[24] A. Bordes, N. Usunier, A. García-Durán, J. Weston, O. Yakhnenko, Translating embeddings for modeling multi-relational data, in: NIPS, 2013, pp. 2787–2795.

[25] B. Yang, W. Yih, X. He, J. Gao, L. Deng, Embedding entities and relations for learning and inference in knowledge bases, in: ICLR (Poster), 2015.

[26] M. Nickel, V. Tresp, Tensor factorization for multi-relational learning, in: ECML/PKDD (3), in: Lecture Notes in Computer Science, vol. 8190, Springer, 2013, pp. 617–621.

[27] V. Nair, G.E. Hinton, Rectified linear units improve restricted Boltzmann machines, in: ICML, Omni Press, 2010, pp. 807–814.

[28] T. Dettmers, P. Minervini, P. Stenetorp, S. Riedel, Convolutional 2D knowledge graph embeddings, in: AAAI, AAAI Press, 2018, pp. 1811–1818.

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

[30] J. Chen, P. Hu, E. Jiménez-Ruiz, O.M. Holter, D. Antonyrajah, I. Horrocks, OWL2Vec*: embedding of OWL ontologies, Mach. Learn. 110 (7) (2021) 1813–1845.

[31] F.Z. Smaili, X. Gao, R. Hoehndorf, OPA2Vec: combining formal and informal content of biomedical ontologies to improve similarity-based prediction, Bioinform. 35 (12) (2019) 2133–2140.

[32] F.Z. Smaili, X. Gao, R. Hoehndorf, Onto2Vec: joint vector-based representation of biological entities and their ontology-based annotations, Bioinform. 34 (13) (2018) i52–i60.

[33] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in: NIPS, 2013, pp. 3111–3119.

[34] Z. Sun, Z. Deng, J. Nie, J. Tang, Rotate: Knowledge graph embedding by relational rotation in complex space, 2019, CoRR abs/1902.10197.

[35] T. Trouillon, J. Welbl, S. Riedel, E. Gaussier, G. Bouchard, Complex embeddings for simple link prediction, in: ICML, in: JMLR Workshop and Conference Proceedings, vol. 48, JMLR.org, 2016, pp. 2071–2080.

[36] Q.V. Le, T. Mikolov, Distributed representations of sentences and documents, in: ICML, in: JMLR Workshop and Conference Proceedings, vol. 32, JMLR.org, 2014, pp. 1188–1196.

[37] K. Toutanova, D. Chen, Observed versus latent features for knowledge base and text inference, in: Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality, 2015, pp. 57–66.

[38] Z. Sun, W. Hu, C. Li, Cross-lingual entity alignment via joint attribute-preserving embedding, in: ISWC (1), in: Lecture Notes in Computer Science, vol. 10587, Springer, 2017, pp. 628–644.

[39] A.M. Saxe, J.L. McClelland, S. Ganguli, Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, in: ICLR, 2014.

[40] R. Abboud, I.I. Ceylan, T. Lukasiewicz, T. Salvatori, Boxe: A box embedding model for knowledge base completion, in: NeurIPS, 2020.

[41] T. Wu, A. Khan, H. Gao, C. Li, Efficiently embedding dynamic knowledge graphs, 2019, CoRR abs/1910.06708.

[42] Y. Tay, A.T. Luu, S.C. Hui, Non-parametric estimation of multiple embeddings for link prediction on dynamic knowledge graphs, in: AAAI, AAAI Press, 2017, pp. 1243–1249.

[43] A. Hofmann, S. Perchani, J. Portisch, S. Hertling, H. Paulheim, DBkWik: Towards knowledge graph creation from thousands of Wikis, in: ISWC (Posters, Demos & Industry Tracks), in: CEUR Workshop Proceedings, vol. 1963, CEUR-WS.org, 2017.

[44] S. Hertling, H. Paulheim, DBkWik: A consolidated knowledge graph from thousands of Wikis, in: ICBK, IEEE Computer Society, 2018, pp. 17–24.