LibKGE
A knowledge graph embedding library for reproducible research

Samuel Broscheit, Daniel Ruffinelli, Adrian Kochsiek, Patrick Betz, Rainer Gemulla
Data and Web Science Group
University of Mannheim, Germany
broscheit, daniel, adrian@informatik.uni-mannheim.de, pbetz@mail.uni-mannheim.de, rgemulla@uni-mannheim.de

Abstract

LibKGE\(^1\) is an open-source PyTorch-based library for training, hyperparameter optimization, and evaluation of knowledge graph embedding models for link prediction. The key goals of LibKGE are to enable *reproducible research*, to provide a framework for comprehensive experimental studies, and to facilitate analyzing the contributions of individual components of training methods, model architectures, and evaluation methods. LibKGE is highly configurable and every experiment can be fully reproduced with a single configuration file. Individual components are decoupled to the extent possible so that they can be mixed and matched with each other. Implementations in LibKGE aim to be as efficient as possible without leaving the scope of Python/Numpy/PyTorch. A comprehensive logging mechanism and tooling facilitates in-depth analysis. LibKGE provides implementations of common knowledge graph embedding models and training methods, and new ones can be easily added. A comparative study (Ruffinelli et al., 2020) showed that LibKGE reaches competitive to state-of-the-art performance for many models with a modest amount of automatic hyperparameter tuning.

1 Introduction

Knowledge graphs (KG) (Hayes-Roth, 1983) encode real-world facts as structured data. A knowledge graph can be represented as a set of (subject, relation, object)-triples, where the subject and object entities correspond to vertices, and relations to labeled edges in a graph.

KG embedding (KGE) models represent the KG’s entities and relations as dense vectors, termed embeddings. KGE models compute a score based on these embeddings and are trained with the objective of predicting high scores for true triples and low scores for false triples. Link prediction is the task of predicting edges missing in the KG (Nickel et al., 2015). Some uses of KGE models are: enhancing the knowledge representation in language models (Peters et al., 2019), drug discovery in biomedical KGs (Mohamed et al., 2019), as part of recommender systems (Wang et al., 2017), or for visual relationship detection (Baier et al., 2017).

KGE models for link prediction have seen a heightened interest in recent years. Many components of the KGE pipeline—i.e., KGE models, training methods, evaluation techniques, and hyperparameter optimization—have been studied in the literature, as well as the whole pipeline itself (Nickel et al., 2016; Wang et al., 2017; Ali et al., 2020). Ruffinelli et al. (2020) argued that it is difficult to reach a conclusion about the impact of each component based on the original publications. For example, multiple components may have been changed simultaneously without performing an ablation study, baselines may not have been trained with state-of-the-art methods, or the hyperparameter space may not have been sufficiently explored.

LibKGE is an open-source KGE library for reproducible research. It aims to facilitate meaningful experimental comparisons of all components of the KGE pipeline. To this end, LibKGE is faithful to the following principles:

- **Modularization and extensibility.** LibKGE is cleanly modularized. Individual components can be mixed and matched with each other, and new components can be easily added.

- **Configurability and reproducibility.** In LibKGE an experiment is entirely defined by a single configuration file with well-documented configuration options for every component. When an experiment is started, its current configuration is stored alongside the model to enable reproducibility and analysis.

\(^1\)https://github.com/uma-pi1/kge
Profiling and analysis. LIBKGE performs extensive logging during experiments and monitors performance metrics such as runtime, memory usage, training loss, and evaluation metrics. Additionally, specific monitoring of any part of the KGE pipeline can be added via a hook system. The logging is done in both human-readable form and in a machine-readable format.

Ease of use. LIBKGE is designed to support the workflow of researchers by convenient tooling and easy usage with single line commands. Each training job or hyperparameter search job can be interrupted and resumed at any time. For tuning of hyperparameters, LIBKGE supports grid search, quasi-random search and Bayesian Optimization. All implementations stay in the realm of Python/Torch/Numpy and aim to be as efficient as possible.

LIBKGE supports the needs of researchers who want to investigate new components or improvements of the KGE pipeline. The strengths of LIBKGE enabled a comprehensive study that provided new insights about training KGE models (Ruffinelli et al., 2020). For an overview about usage, pretrained models, and detailed documentation, please refer to LIBKGE’s project page. In this paper, we discuss the key principles of LIBKGE.

2 Modularization and extensibility

LIBKGE is highly modularized, which allows to mix and match training methods, models, and evaluation methods (see Figure 1). The modularization allows for simple and clean ways to extend the framework with new features that will be available for every model.

For example, LIBKGE decouples the RelationalScorer (the KGE scoring function) and KgeEmbedder (the way embeddings are obtained) as depicted in Figure 1. In other frameworks, the embedder function is hardcoded to the equivalent of LIBKGE’s LookupEmbedder, in which embeddings are explicitly stored for each entity. Due to LIBKGE’s decoupling, the embedder type can be freely specified independently of the scoring function, which enables users to train a KGE model with other types of embedders. For example, the embedding function could be an encoder that computes an entity or relation embedding from textual descriptions or pixels of an image (Pezeshkpour et al., 2018; Broscheit et al., 2020, inter alia).

3 Configurability and reproducibility

Reproducibility is important, which means that configuration is important. To enable reproducibility, it is key that the entire configuration of each experiment be persistently stored and accessible. While this sounds almost obvious, the crux is how this can be achieved. Typically, source code can and will change. Therefore, to make an experiment in a certain setting reproducible, the configuration for an experiment has to be decoupled from the code as much as possible.

In LIBKGE all settings are always retrieved from a configuration object that is initialized from configuration files and is used by all components of the pipeline. This leads to comprehensive configuration files that fully document an experiment and make it reproducible as well.

To make this comprehensive configurability feasible—while also remaining modular—LIBKGE includes a lightweight import functionality for configuration files. In Figure 2, we show an (almost) minimal configuration for an experiment for training a ComplEx KGE model (Trouillon et al., 2016). The main configuration file...
my_experiment.yaml

```yaml
job.type: train
dataset.name: fb15k-237
train:
  optimizer: Adagrad
  optimizer.args:
    lr: 0.2
valid:
  every: 5
  metric:
    mean_reciprocal_rank_filtered
model: complex
lookup_embedder:
  dim: 200
  regularize_weight: 0.8e-7
```

complex.yaml

```yaml
import: [lookup_embedder]
complex:
  class_name: ComplEx
  entity_embedder:
    type: lookup_embedder
  relation_embedder:
    type: lookup_embedder
import settings from
lookup_embedder.yaml
```

lookup_embedder.yaml

```yaml
lookup_embedder:
  class_name: LookupEmbedder
  dim: 100
  initialize: normal_
  dropout: 0.
  regularize: 'lp'
  regularize_weight: 0.0
  regularize_args:
    p: 2
```

token_pool_embedder:

```yaml
token_pool_embedder:
  class_name: TokenPoolEmbedder
  dim: 100
```

The resulting single configuration will be automatically merged and stored in one combined file. No default settings are defined in the code.

Figure 2: A minimal configuration `my_experiment.yaml` that defines 10 out of $\approx 100$ configurable settings. All settings from the main configuration file `my_experiment.yaml` and from the imported configurations are merged and stored in one combined file. No default settings are defined in the code.

As an example of how configurability also helps modularization, we come back to the example of switching the `LookupEmbedder` with an encoder that computes entity embeddings from string tokens. For this purpose, one may implement a `TokenPoolEmbedder`. The simple changes to the configuration that uses the new embedder type are demonstrated in Figure 3 (see line 12).

It is worth noting that while the default settings in LIBKGE’s main configuration file reflect the currently known best practices, LIBKGE also includes—and makes configurable—some settings that might not be considered best practice, e.g., different tie breaking schemes for ranking evaluations (Sun et al., 2020). Therefore, with regards to configurability, the goal is not only that the frame-
work reflects best practices, but also reflects popular practices that might influence ongoing research.

4 Hyperparameter optimization

Hyperparameter optimization is crucial in empirically investigating the impact of individual components of the KGE pipeline. LibKGE offers manual search, grid search, random search, and Bayesian Optimization; the latter two provided by the hyperparameter optimization framework Ax. In this context, LibKGE further benefits from its configurability because everything can be treated as a hyperparameter, even the choice of model, score function, or embedder. The example in Figure 4 shows a simple hyperparameter search with an initial quasi-random search, and a subsequent Bayesian Optimization phase over the learning rate, batch size and negative samples for the ComplEx model. The trials during the quasi-random search are independent, which can be exploited by parallelizing their runs over multiple devices. In this way, a comprehensive search over a large space of hyperparameters can be sped up significantly (also shown in the example; for more details, please refer to the documentation).

5 Profiling and metadata analysis

LibKGE provides extensive options for profiling, debugging, and analyzing the KGE pipeline. While most frameworks print the current training loss and some frameworks also record the validation metrics, LibKGE aims to make every moving part of the pipeline observable. Per default, LibKGE records during training things such as runtimes, training loss and penalties (e.g., from the entity and relation embedders), relevant meta data such as the PyTorch version and the current commit hash, and dependencies between various jobs. We show an example logging output during training one epoch in Appendix B. For more fine-grained logging, LibKGE also can log at the batch level.

During evaluation, the framework records many variations of the evaluation metrics, such as grouping relations by relation type, relation frequency, head or tail. Additionally, users can extract and add information by adding their custom function to one of multiple hooks that are executed before and after all relevant calls in the framework. In this way, users can interact with all components of the pipeline, without risking divergence from LibKGE’s master branch.

Finally, LibKGE provides convenience methods to export (subsets of) the logged meta data into plain CSV files.

6 Comparison to other KGE Projects

In this section, we compare LibKGE to other open source software (OSS) that provides functionality around training and evaluating KGE models for link prediction. The assessments are a snapshot taken at the end of May 2020. All model-specific comparisons have been evaluated w.r.t. the ComplEx model, which is supported by all projects.
In Table 1, we provide an overview of other KGE projects (full references in Appendix C) and compare them w.r.t. configurability and ease of use. We mainly included projects that could be considered as a basis for a researcher’s experiments because they are active, functional, and cover at least a few of the most common models. All projects can be extended with models, losses, or training methods. Large-scale projects and paper code projects—in comparison to more holistic frameworks—typically have a more narrow scope, e.g., they often do not feature hyperparameter optimization. Large-scale projects are typically tailored towards parallelizing training methods and models.

The focus on configurability and reproducibility in LibKGE is reflected by the large amount of configurable keys. For example, in contrast to other projects, LibKGE does not tie the regularization weights of the entity and relation embedder to be the same. For entity ranking evaluation, only LibKGE and PyKeen transparently implement different tie breaking schemes for equally ranked entities. This is important, because evaluation under different tie breaking schemes can result in differences of $\approx .40$ MRR in some models and can lead to misleading conclusions, as shown by Sun et al. (2020). OpenKE, for example, only supports the problematic tie breaking scheme named TOP by Sun et al. (2020). LibKGE and PyKeen are the only frameworks that provide machine-readable logging. Only LibKGE offers resuming from a checkpoint for training and hyperparameter search. LibKGE, Ampligraph, and PyKeen are the most active projects in terms of amount of commits during the past 12 months.

**Efficiency** In Table 2, we show a comparison of KGE frameworks in terms of time for one full training epoch. The configuration setting was chosen such that it was supported by all frameworks, and also facilitates to demonstrate behaviour under varying load. We translate the configurations faithfully to each framework, ensuring that total number of embedding parameters per batch are the same for each framework. Most projects, including LibKGE, can handle small numbers of negative samples efficiently, but LibKGE seems to scale...
Table 2: Runtime comparison between frameworks. The runtime is the time per epoch in seconds (averaged over 5 epochs executed on the same machine). The configuration is fixed to be similar for all frameworks (details in Appendix A). For negative samples, we show runtimes for random, i.e., sampling triples without checking if they are contained in the KG, and for pseudo-negative, which avoids sampling triples contained in the KG. The column parallel batch construction indicates whether the code in the training loop for generating the batches is parallelized; if yes, then we set the number of workers to 4. OOM stands for out-of-memory. OOT is short for out-of-time; we stopped the run when the first epoch did not finish within 30 minutes. (*) Graphvite is optimized for multi-gpu training with large batch sizes, therefore the chosen settings might not be optimal.

Table 3: The reported best performances (on the project’s homepage or the related publication as of May 2020) for ComplEx on the dataset FB15K-237 (Toutanova and Chen, 2015). The numbers are not comparable due to different amount of effort to find a good configuration3, but they reflect the performance that the framework authors achieved in their experiments. The results show that with LibKGE’s architecture and hyperparameter optimization a state-of-the-art result can be achieved. For more results obtained with LibKGE and an in-depth analysis of the impact of hyperparameters on model performance we refer to the study by Ruffinelli et al. (2020).

7 Conclusions

In this work, we presented LibKGE, a configurable, modular, and efficient framework for reproducible research on knowledge graph embedding models. We briefly described the internal structure of the framework and how it facilitates LibKGE’s goals. The framework is efficient and yields state-of-the-art performance. We hope that LibKGE is a helpful ingredient to gain new insights into knowledge graph embeddings, and that a lively community gathers around this project to improve and extend it further.

3We did not attempt to use our best configuration with other frameworks because they only partly support the settings, e.g., they do not offer dropout or independent regularization for entity and relation embeddings.
References


Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 2015. A review of relational machine learning for knowledge graphs. Proceedings of the IEEE.


A Runtime comparison experiment

The configuration was as follows: Dataset FB15K (Bordes et al., 2013), batch size 512, model ComplEx, effective parameter embedding size per entity/relation 128, optimizer Adagrad, negative sampling with negative log likelihood loss or sigmoid loss, no regularization. The hardware was a 8-core Intel Xeon E5-1630m v4.0, TitanXP GPU, dataset on SSD.
B  Logging

```json
{
  "entry_id": :84d75bf2-c3fe-4c6f-ac5e-001e1edb85de,
  "event": "job_created",
  "folder": /home/USER/kge/local/experiments/20200705-215353-toy-complex-train,
  "git_head": 7fad132,
  "hostname": USER-Workstation,
  "job": "eval",
  "job_id": 683d00bf-520d-4919-937e-d9b634c11d2e,
  "parent_job_id": dc960211-9cbe-4ba1-ad62-7ff41d2017e,
  "timestamp": 1593978837.304522,
  "torch_version": 1.5.0,
  "username": "USER"
}
{
  "entry_id": 418889f0-728b-486f-9977-48795f6ed5fa,
  "event": "job_created",
  "folder": /home/USER/kge/local/experiments/20200705-215353-toy-complex-train,
  "git_head": 7fad132,
  "hostname": USER-Workstation,
  "job": "train",
  "job_id": dc960211-9cbe-4ba1-ad62-7ff41d2017e,
  "timestamp": 1593978837.4033182,
  "torch_version": 1.5.0,
  "username": "USER"
}
{
  "avg_cost": 1.06542689547832,
  "avg_loss": 1.0650610147438764,
  "avg_penalties": {
    complex.entity_embedder.L2_penalty: 0.00031927969330354246,
    complex.relation_embedder.L2_penalty: 4.6601041140093004e-05
  },
  "avg_penalty": 0.000365880734436354,
  "backward_time": 0.0765678882598877,
  "batches": 20,
  "entry_id": 4b07adfa-3e2b-42f4-a994-2b4f02e1b3f4,
  "epoch": 1,
  "epoch_time": 1.161754846572876,
  "event": "epoch_completed",
  "forward_time": 0.7509596347808838,
  "job": "train",
  "job_id": dc960211-9cbe-4ba1-ad62-7ff41d2017e,
  "lr": [ 0.2 ],
  "optimizer_time": 0.015013933181762695,
  "other_time": 0.2690012454986572,
  "prepare_time": 0.05021214485168457,
  "scope": "epoch",
  "size": 1949,
  "split": "train",
  "timestamp": 1593978837.5940151,
  "type": "KvsAll"
}
```

Figure 5: Example for training logging output for one epoch. Evaluation logging output is too verbose to add an example here. Please see https://github.com/uma-pi1/kge/blob/master/docs/examples/train_and_valid_trace_after_one_epoch.yaml for an example for the output after one epoch of training and evaluation.
<table>
<thead>
<tr>
<th>KGE project</th>
<th>URL</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>LibKGE (Ours)</td>
<td><a href="https://github.com/uma-pil/kge">https://github.com/uma-pil/kge</a></td>
<td>Ruffinelli et al. (2020)</td>
</tr>
<tr>
<td>Ampligraph</td>
<td><a href="https://github.com/Accenture/AmpliGraph">https://github.com/Accenture/AmpliGraph</a></td>
<td>Costabello et al. (2019)</td>
</tr>
<tr>
<td>GraphVite</td>
<td><a href="https://github.com/DeepGraphLearning/graphvite">https://github.com/DeepGraphLearning/graphvite</a></td>
<td>Zhu et al. (2019)</td>
</tr>
<tr>
<td>Pytorch-Bigraph</td>
<td><a href="https://github.com/facebookresearch/PyTorch-BigGraph">https://github.com/facebookresearch/PyTorch-BigGraph</a></td>
<td>Lerer et al. (2019)</td>
</tr>
<tr>
<td>SK-libkge</td>
<td><a href="https://github.com/samehkamaleldin/libkge">https://github.com/samehkamaleldin/libkge</a></td>
<td></td>
</tr>
<tr>
<td>RotatE</td>
<td><a href="https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding">https://github.com/DeepGraphLearning/KnowledgeGraphEmbedding</a></td>
<td>Sun et al. (2019)</td>
</tr>
</tbody>
</table>

Table 4: Overview of related work