Enhancing PyKEEN with Multiple Negative Sampling Solutions for Knowledge Graph Embedding Models

Claudia d'Amato[1,2][0000−0002−3385−987X], Ivan Diliso[1][0009−0007−2942−202X], Nicola Fanizzi[1,2][0000−0001−5319−7933], and Zafar Saeed[1][0000−0002−1282−5478]

[1] Dipartimento di Informatica – University of Bari Aldo Moro, Italy
i.diliso1@phd.uniba.it, claudia.damato@uniba.it, nicola.fanizzi@uniba.it
zafar.saeed@uniba.it
[2] CILA – University of Bari Aldo Moro, Italy

Abstract. *Embedding methods* have become popular due to their scalability on link prediction and/or triple classification tasks on *Knowledge Graphs*. Embedding models are trained relying on both positive and negative samples of triples. However, in the absence of negative assertions, these must be usually artificially generated using various *negative sampling* strategies, ranging from random corruption to more sophisticated techniques which have an impact on the overall performance. Most of the popular libraries for knowledge graph embedding, support only basic such strategies and lack advanced solutions. To address this gap, we deliver an extension for the popular KGE framework *PyKEEN* that integrates a suite of several advanced negative samplers (including both static and dynamic corruption strategies), within a consistent modular architecture, to generate meaningful negative samples, while remaining compatible with existing *PyKEEN*-based workflows and pipelines. The developed extension not only enhances *PyKEEN* itself but also allows for easier and comprehensive development of embedding methods and/or for their customization. As a proof of concept, we present a comprehensive empirical study of the developed extensions and their impact on the performance (link prediction tasks) of different embedding methods, which also provides useful insights for the design of more effective strategies.

Keywords: Knowledge Graphs · Graph Embedding · Graph Representation Learning · Negative Sampling · Corruption Techniques

1 Introduction

Knowledge Graphs (KGs) represent data in graph structure as factual statements in the form of triples. They provide a complex data representation solution that is effectively used for numerous knowledge-intensive applications [13, 21]. Despite their usefulness, KGs result incomplete because of their inherently distributed nature. To address this problem, automated completion tasks have gained great importance, which has led to the development of efficient *Knowledge Graph Embedding* (KGE) models, resulting from the application of *Representation Learning* to KGs. KGE models encode entities and relations described by a KG through a low-dimensional vector space. Training such models relies on a contrastive learning approach, where observed positive triples must be distinguished from artificially generated *negative samples* (NS) [23]. In these models NS is essential for discriminative purposes. However, generating accurate negative samples is known to be a critical and challenging task, as KGs consist exclusively of positive statements (*triples*). NS is typically performed by randomly corrupting observed triples, i.e. by replacing either their subject or object with an entity randomly sampled from the KG, on the ground of the *local closed-world assumption* (LCWA), which posits that the collection is locally complete [26]. This random method may generate poor (trivial or false) negative samples. Therefore, NS requires careful differentiation, as the quality of negative samples greatly affects the performance of downstream tasks such as *link prediction* that are performed via KGE models. Low quality negatives lead to suboptimal embeddings and degraded performance. After the foundational work on learning KGE models [6] many advances have been proposed that leverage NS to generate high-quality negatives, focusing on entity similarity, relational semantics [18, 12, 2], network structure [34], type-constrained [19], probability distribution of entities over relations [35], entity aware [15], and adversarial sampling [9, 38]. Current methods attempt to exploit the latent semantics in KGs to generate negative samples, but finding a balance between generating meaningful negative samples and avoiding false ones remains a critical challenge. The semantics of KGs should be fully and exclusively exploited to get explicit, correct negative statements. In principle, the approach to generate correct negative statements should be deeply rooted in the semantics of the schema axioms. However, despite its high value and availability, little use is made of schema-level knowledge.

In order to devise and experiment advanced approaches to learning quality embedding models, it appears essential to investigate the interplay between orthogonal components: the representation model and the sampling approach adopted for its fitting. In fact, given state-of-the-art models, new approaches can be devised by integrating them with alternative NS solutions, so to gain improved effectiveness. Most of KGE software libraries support only basic NS strategies and lack advanced solutions. In addition, ad hoc sampling techniques are tightly coupled to the embedding methods provided. In order to tackle such issues, targeting *PyKEEN*, one of the most popular libraries for building KGE models, we have designed and implemented a modular extension that provides a seamless way to integrate advanced NS techniques with the many existing (and even future) KGE models available, thus multiplying the number of possible combined solutions that can be comparatively investigated. Based on this extension, the exploration of the impact of NS on existing and novel KGE methods (specified in *PyKEEN*) is facilitated. As a proof-of-concept, we performed an empirical comparative study on NS, analyzing state-of-the-art techniques, and several KGE models available in *PyKEEN*.

In the following, after reviewing, in Sect. 2, the fundamental principles behind negative sampling, the basic and more advanced NS approaches that we implemented within *PyKEEN*, we proceed with illustrating, in Sect. 3, the design and development of provided resource, that is a modular implementation of multiple NS techniques as an extension of the popular *PyKEEN* framework for KGE. For showcasing the utility of the developed *PyKEEN* extesion, in Sect. 4, we present a comprehensive experiment design to compare and evaluate the impact of the developed NS solutions, assessing their performance on link prediction tasks.

2 Basics on Negative Sampling Methods

The concept of *negative sampling* was first introduced in the context of probabilistic language modeling [24]. The inclusion of negative samples simplifies the task by transforming maximum likelihood estimation into a binary classification problem and learning to discriminate between positive and negative samples. Training KGE models using negative sampling involves treating (head) entities as analogous to words and their neighboring (tail) entities as their context [23]. KGE models are trained by distinguishing between positive and negative instances. Accurate negative samples enhance the model capacity of grasping the underlying semantics: the quality of negative samples is crucial for optimizing training effectiveness and the models' performance on KG completion tasks.

Since negative triples are not included in the KG, most approaches leverage some form of *Close-World Assumption* (CWA) and systematically corrupt the positive triples to generate negative samples. By imposing various forms of CWA, most negative sampling methods treat non-occurring triples as negatives [10]. Following this formulation, given the set $\mathcal{E}$ of the KG entities and a positive triple $p = \langle h, r, t \rangle$, with $h, t \in \mathcal{E}$, $p \in \mathcal{T}$ the set of positive triples, one can generate negative triples by replacing either h or t with any entity e' sampled from $\mathcal{E}$, yielding the negative triples $n_t = \langle h, r, e' \rangle$ (*tail corruption*) and $n_h = \langle e', r, t \rangle$ (*head corruption*) ensuring that $n_h, n_t \notin \mathcal{T}$. A notion that is key to formulating specific negative samplers is the *negative pool*, the set of all available entities that can be used to form a correct negative triple when replacing a head or tail; formally:

$$\begin{aligned} \mathcal{P}_{head}(\langle h, r, t \rangle) &= \{h' \mid h' \in \mathcal{E}, \langle h', r, t \rangle \notin \mathcal{T}\} \\ \mathcal{P}_{tail}(\langle h, r, t \rangle) &= \{t' \mid t' \in \mathcal{E}, \langle h, r, t' \rangle \notin \mathcal{T}\} \end{aligned}$$

Given head and tail negative pools, one can randomly sample entities from these sets, corrupt the relative target and produce the negative triples. This formulation of negative pool generation is the key difference between the various implemented negative samplers, as it changes the corruption logic for producing ad hoc negative sets for each triple.

2.1 Static Corruption

In static corruption, a negative triple is created by defining a criterion for selecting a subset of candidate entities to be used as a negative pool from which to sample the entities. The selection of the negative pool can follow specific criteria [30, 18], such as probability distribution over entities [18], types, and relations. The main difference between these approaches is the logic used to compute the negative pool.

Random Sampling. This is the most commonly used, efficient, and interpretable negative sampler, which defines the negative pool as the entire entity set and samples entities at random. It follows exactly the same formulation as the general one, where the entire entity set is used as a negative candidate pool.

$$\mathcal{P}_{head}^{random}(\langle h, r, t\rangle) = \mathcal{P}_{tail}^{random}(\langle h, r, t\rangle) = \mathcal{E}$$

Bernoulli Sampling [6]. Bernoulli shares structural similarities with random sampling in that the negative pool consists of the full set of entities. However, the key difference lies in the sampling probability distribution. While random sampling treats all entities uniformly (assigning equal probability to each candidate) Bernoulli sampling introduces an asymmetric corruption strategy. Specifically, it adjusts the probability of corrupting the head or tail entity by analyzing how the relation typically connects entities, for example, whether it tends to link one entity to many others (1-to-N) or vice versa (N-to-1). By analyzing the negative pool we can say that:

$$\mathcal{P}_{head}^{bernoulli}(\langle h, r, t\rangle) = \mathcal{P}_{tail}^{bernoulli}(\langle h, r, t\rangle) = \mathcal{E}$$

Corrupt Sampling [31]. The corruption strategy is based on the entities that appear as heads or tails for each relation in the positive instance set, leveraging the relation structural information to define the pool of available candidate negative triples. Given a triple $\langle h, r, t\rangle$ we can define the negative pools as:

$$\begin{aligned}\mathcal{P}_{head}^{corrupt}(\langle h, r, t\rangle) &= \{h' \mid h' \in \mathcal{E}, \langle h', r, *\rangle \in \mathcal{T}\}\\ \mathcal{P}_{tail}^{corrupt}(\langle h, r, t\rangle) &= \ \{t' \mid t' \in \mathcal{E}, \langle *, r, t'\rangle \in \mathcal{T}\}\end{aligned}$$

Typed Sampling [19]. This approach takes advantage of the strongly typed relations and entities present in KGs such as DBpedia [20] and YAGO [29]. Given a triple $\langle h, r, t\rangle$, we define D_r, R_r as the domain and range classes of the relation r, assuming the dataset has type information in the form of $\langle h, instanceOf, c\rangle$, we can define the negative pools for each triple as:

$$\begin{aligned}\mathcal{P}_{head}^{typed}(\langle h, r, t\rangle) &= \{h' \mid h' \in \mathcal{E}, \langle h', \mathit{instanceOf}, D_r\rangle \in \mathcal{T}\}\\ \mathcal{P}_{tail}^{typed}(\langle h, r, t\rangle) &= \ \{t' \mid t' \in \mathcal{E}, \langle t', \mathit{instanceOf}, R_r\rangle \in \mathcal{T}\}\end{aligned}$$

Basically, the negative pool for a triple head (tail) corruption is the set of all entities that belong to the domain (range) of its relation. If domain and range properties are not available, a variant of this implementation is defined. In these cases when corrupting head (tail), we use as a negative pool all the entities belonging to the same class of the head (tail) entity:

$$\begin{aligned}\mathcal{P}_{head}^{typed}(\langle h,r,t\rangle) &= \{h' \mid h' \in \mathcal{E}, \langle h', \mathit{instanceOf}, C\rangle, \langle h, \mathit{instanceOf}, C\rangle \in \mathcal{T}\}\\ \mathcal{P}_{tail}^{typed}(\langle h,r,t\rangle) &= \{t' \mid t' \in \mathcal{E}, \langle t', \mathit{instanceOf}, C\rangle, \langle t, \mathit{instanceOf}, C\rangle \in \mathcal{T}\}\end{aligned}$$

Relational Sampling [18]. This approach makes the assumption that each head-tail pair participates only in one relation. Given a triple $\langle h, r, t\rangle$ the negative pools are computed as:

$$\begin{aligned}\mathcal{P}_{head}^{relational}(\langle h,r,t\rangle) &= \{h' \mid h' \in \mathcal{E}, \langle h', r', t\rangle \in \mathcal{T}, r' \neq r\}\\ \mathcal{P}_{tail}^{relational}(\langle h,r,t\rangle) &= \{t' \mid t' \in \mathcal{E}, \langle h, r', t'\rangle \in \mathcal{T}, r' \neq r\}\end{aligned}$$

2.2 Dynamic Corruption

In contrast to static sampling strategies where the negative pool can be precomputed based on fixed structural or semantic information, dynamic samplers adopt a different paradigm. These methods leverage a pre-trained auxiliary model to guide the selection of informative, harder negative samples. Specifically, instead of defining the negative pool through schema-based constraints or relational assumptions, the implemented dynamic samplers leverage the entities and predicted entities vector space representation to find more informative negatives.

In our implementation, we incorporate two dynamic sampling strategies following the approach proposed in [18]. These variants differ in how the auxiliary model's learned embeddings are utilized to construct negative pools, reflecting different assumptions about what constitutes an informative negative. Let n be the total number of entities, d the dimensionality of the embedding space, $\mathbf{E} \in \mathbb{R}^{n\times d}$ the entity embeddings learned by the auxiliary model and $\mathrm{KNN}(\boldsymbol{e}, \mathbf{E}, k) \rightarrow \mathbb{N}$ the function that returns the indices of the top k entities closest to $\boldsymbol{e}$ from the embedding $\mathbf{E}$ using a (dis)similarity measure of choice.

Nearest Neighbor [18]. In this approach, we use an auxiliary pre-trained model to produce vector representations of entities. Given a triple $\langle h, r, t\rangle$, when corrupting on heads (tails) one selects hard negatives by retrieving the k entities nearest to either the head (tail) vector representation. This results in a context-aware negative pool, tailored to the embedding geometry. Let $\mathrm{Model}_{\mathrm{AUX}} : \mathbb{N} \rightarrow \mathbb{R}^d$ be the function that returns the embedding vector for entity e using the auxiliary model learned embeddings then, the negative pools for head and tail corruption are defined as follows:

$$\begin{aligned}\mathcal{P}_{head}^{knn}(\langle h,r,t\rangle) &= \mathrm{KNN}(\mathrm{Model}_{\mathrm{AUX}}(h), \mathbf{E}, k)\\ \mathcal{P}_{tail}^{knn}(\langle h,r,t\rangle) &= \mathrm{KNN}(\mathrm{Model}_{\mathrm{AUX}}(t), \mathbf{E}, k)\end{aligned}$$

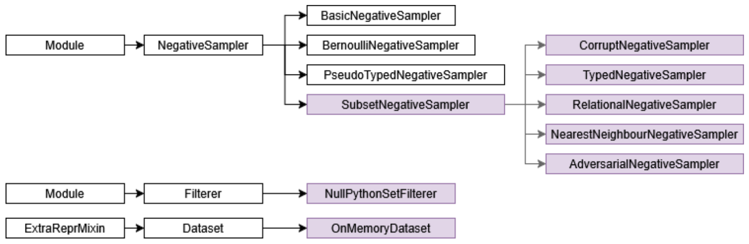

Fig. 1. Class inheritance diagram based on *PyKEEN*'s class structure. Custom classes introduced by our extension are highlighted in color.

Adversarial Sampling [18]. This sampling method follows exactly the same formulation as the previous dynamic sampler, but uses the model predictions in the vector space instead of the entity embeddings. Let $\text{Model}_{\text{AUX}}(e, r, target) \rightarrow \mathbb{R}^d$ be the function that returns the embedding vector of the prediction on target $target$, given entity e and relation r (for example, in *TransE*, the prediction on tail would be $h+r$ and the prediction on head would be $t-r$). Then the negative pools for head and tail corruption are defined as follows:

$$\mathcal{P}_{head}^{adversarial}(\langle h, r, t\rangle) = \text{KNN}(\text{Model}_{\text{AUX}}(h, r, \text{"head"}), \mathbf{E}, k)$$
$$\mathcal{P}_{tail}^{adversarial}(\langle h, r, t\rangle) \ = \text{KNN}(\text{Model}_{\text{AUX}}(t, r\text{"tail"}), \mathbf{E}, k)$$

3 Resource Description and Technical Details

The proposed resource is an extension to the *PyKEEN* framework, adding a suite of advanced negative sampling strategies for KGE models that are unified under a standardized architecture. The resource is designed to be modular and extensible, and is fully compatible with existing *PyKEEN* workflows. This is achieved by extending the *PyKEEN* abstract classes and adhering to its internal design principles to provide new functionalities within the same standardization. The extension integrates seamlessly with core *PyKEEN* functionalities, such as triple filtering, training, evaluation, and hyperparameter optimisation loops.

3.1 Core Extensions: Static Sampling

Static negative sampling refers to strategies in which a predefined pool of candidate entities is used to corrupt triples. Our work builds on this concept by introducing a significant refinement: the negative pool is dynamically adjusted for each triple, enabling context-aware sampling. This means allowing each triple to define its own tailored set of candidates, based on types or structural constraints. The extension is structured around the core class '`SubsetNegativeSampler`', which inherits from the abstract base class '`NegativeSampler`' in *PyKEEN*. This provides a generic, reusable, abstract implementation designed to handle the low-level functionalities required to support ad hoc negative pools for each triple. The extension has been developed with extensibility and developer usability in mind. In particular, it isolates the core sampling logic in two abstract methods that custom samplers must implement:

- `generate_subset()`: This method handles any pre-computation required for sampling, such as filtering by type, frequency, or other criteria. It is designed to avoid redundant computation by preparing intermediate structures beforehand instead of recalculating them for each batch, thereby improving efficiency.
- `strategy_negative_pool()`: This method specifies how to calculate the set of actual candidate negatives for a given triple, corruption target and precomputed subset.

By requiring the implementation of only these two methods, the framework abstracts all the 'under-the-hood' complexity, such as batching and tensor operations. Five new negative samplers have been developed following this implementation: `"Corrupt"`, `"Typed"`, and `"Relational"`. These samplers employ various strategies that explore structural and semantic criteria for entity selection, offering an efficient and user-friendly solution within the *PyKEEN* ecosystem. Fig. 1 provides an overview of the implemented samplers in the *PyKEEN* class diagram. Additionally, several functionalities have been implemented in the abstract class, namely:

- `choose_from_pool()`: A default implementation is provided for randomly sampling entities from the given negative pool. However, developers can override this method and implement custom sampling logic tailored to specific strategies.
- `get_positive_pool()`: This returns the true positive pool of entities for a given triple and corruption target. This can be used to further refine or filter the negative pool during subset generation.
- `average_pool_size()`: It computes the average size of the negative pool across all triples. This metric offers valuable insights into the effectiveness and selectivity of the sampling strategy, enabling more informed decisions about sampler selection.
- `integrate`: This parameter specifies whether to supplement the negative pool of the sampler with random entities when the specific corruption criterion is unable to produce a negative pool of the desired size. More details and an in-depth analysis of this topic is provided in Sect. 4.

3.2 Core Extensions: Dynamic Sampling

These strategies adhere to the same architectural design and logic as static samplers, and are implemented using the `SubsetNegativeSampler` base class. Unlike static samplers, however, which pre-compute entity subsets, dynamic samplers need to compute both the subset and the negative pool on the fly at runtime, for each input triple. This design choice is necessary because the negative pool is derived from the predictions of an auxiliary model, which makes pre-computation

Table 1. Available negative samplers in SOTA KGE libraries

Library	**Samplers**
scikit-kge	Random, Corrupt
OpenKE [14]	Random
Ampligraph [11]	Random
GraphVite [40]	Random
LibKGE [8]	Random, Bernoulli
TorchKGE [7]	Random, Bernoulli, Corrupt
DGL-KE [39]	Random
PyKEEN [3]	Random, Bernoulli, Corrupt
Ours	Random, Bernoulli, Corrupt, Relational, Typed (with domain and range), Typed (with only entity classes), NearestNeighbour, Adversarial

infeasible. Therefore, model predictions must be performed dynamically for each triple, based on the current context and model state. To support flexibility and generalization, the dynamic samplers are designed to accept any pre-trained model that implements:*PyKEEN*'s `ERModel` abstract class. This decouples the auxiliary model from the core sampling logic, enabling developers to define their own training regimes and scoring strategies. The negative sampler requires two inputs:

- a pretrained `ERModel` instance to serve as the auxiliary scoring model;
- a user-defined prediction function, which specifies how to compute prediction with the defined `ERModel`, keeping the predicted entity representation in the vector space.

This approach offers maximum flexibility and extensibility, allowing users to define their own logic for computing model predictions. Following this standard, two samplers have been implemented, namely "NearestNeighbour" and "Adversarial".

3.3 Comparison with SOTA libraries

Tab. 1 presents a detailed comparison of negative sampling support across major open-source KGE libraries. As shown, most libraries limit negative sampling to uniform or Bernoulli corruption, with little or no built-in support for more sophisticated strategies. None provide metadata-driven techniques, and none integrate a standardized fallback mechanism to supplement sparse pools with random samples. Although advanced samplers implementations can be found scattered in separate repositories, our work is the first to offer a fully standardized, modular, negative sampling framework within a widely used library.

3.4 Documentation, Usability and Compatibility

Thorough documentation has been provided for the code base to support reuse and encourage community adoption, namely: Python Docstrings following the standard Google style format to ensure clarity and consistency for developers, dedicated READMEs with usage instructions and configuration options, easy-to-use Bash scripts and examples, and consistent code formatting following the 'black' standard, used to promote readability and adherence to Python best practices. This structured documentation ensures that the resource is technically robust, accessible, maintainable, and easy to extend. It also aligns with best practices for reusable scientific software.

To further demonstrate compatibility with the broader *PyKEEN* ecosystem and facilitate user adoption, we provide working examples of each sampler in three usage scenarios: *Training pipelines*, which integrate the samplers into full training scripts using common KGE models; *hyperparameter optimization*, which configure samplers as part of an automated search space; *standalone use and analysis*, which provides minimal examples that showcase how to instantiate each sampler for inspection, experimentation, and statical analysis.

To demonstrate the ease of integrating our extension into a standard *PyKEEN* pipeline, we provide a minimal example that shows how to register and use a custom negative sampler. The following code snippet shows how a user can use the RelationalNegativeSampler and NullPythonSetFilterer within a *PyKEEN* workflow to integrate the sampler without altering the standard training interface (imports and other details are omitted for brevity):

```
negative_sampler_resolver.register(element=RelationalNegativeSampler)
filterer_resolver.register(element=NullPythonSetFilterer)
pipeline_result = pipeline(
    dataset = "Nations"
    model="TransE",
    negative_sampler="typed"
    negative_sampler_kwargs = dict(
        filtered=True,
        filterer="nullpythonset",
        num_negs_per_pos=params.num_neg_per_pos,
        local_file=/home/me/file.cache
    )
)
```

3.5 Additional Extensions

Filtering Handling triples for which no valid negative sample can be generated is a non-trivial issue. For example, this can happen when a strategy relies on type constraints and type information is unavailable for a subset of entities. To ensure compatibility with *PyKEEN*'s internal computation pipelines, these triples are assigned a entity index value of -1 to serve as a placeholder indicating that they cannot be corrupted by the current strategy. To properly filter out such invalid triples during training, we implemented the custom filtering class `NullPythonSetFilterer`, which extends *PyKEEN*'s set-based filter behavior and excludes negatives containing a negative entity index.

Dataset Negative sampling strategies, such as Typed, require additional semantic information. Specifically, they require domain and range properties for relations, and class membership for entities. To enable the use of these strategies within the *PyKEEN* ecosystem, we have developed a custom data loader that extends the functionality of the *PyKEEN* core dataset. This enhanced loader enables users to readily provide and load external semantic metadata alongside standard triple sets. It enables the seamless integration of semantic metadata with standard knowledge graph triples, giving users the ability to supply extra context for negative sampling with minimal setup. The extended loader supports the ingestion of the following external metadata:

- Domain and Range: A JSON file specifying the domain and range classes associated with each relation;
- Class Membership: A JSON file mapping each entity to its associated semantic classes;
- Precomputed IDs Mappings: JSON files that map entities and relations to internal identifiers, ensuring consistency.

Additionally, four datasets that adhere to the aforementioned standards have been provided in the repository. Each dataset comes with precomputed ID mapping for entities and relations, as well as triples splits (training, testing and validation) in plain text tab-separated value files. Optional metadata files are also included. This standardized formatting promotes consistent usage and reproducibility across experiments. Specifically, data and metadata (if available) are provided for `YAGO4-20` [29], and its subset used in [4], `DBPedia50K` [20], `WN18` [25] and `FB15K` [5].

4 Experimental Design

This section presents the experimental framework used to validate the proposed extension. Rather than performing an exhaustive benchmarking of KGE models, the objective is to demonstrate the functionality, integration, and practical applicability of the developed components through a proof of concept. Our experiments aim to verify the samplers' seamless compatibility with *PyKEEN*'s training, evaluation and hyperparameter optimization pipelines, and to leverage the new proposed implementation to gain insight into the applicability of negative sampling strategies to a wider range of data.

4.1 Analysis of the Number of Available Negatives

The number of negatives generated for each positive instance is a critical parameter in the evaluation of KGE models. In standard random-based approaches, a corrupted entity is sampled from the set of all available entities. However, this assumption poses challenges for methods that employ dynamic or context-specific sampling, since the pool of candidate negatives varies for each triple. In these cases, some triples may lack a sufficient number of valid negative entities. To mitigate this issue, a strategy commonly adopted in the literature [18] involves

Table 2. Statistics of the datasets

Dataset	#Ent	#Rel	Train	Valid	Test	Avg.Degree
FB15K	14951	1345	483142	50000	59071	32.31
WN18	40943	18	141442	5000	5000	3.45

augmenting the negative pool with randomly sampled entities. While this solution ensures a minimum number of negative samples it introduces a compromise by overriding the intended behavior of the negative sampling strategy. This is particularly problematic for samplers where most triples do not have a negative pool of the desired size. Ultimately, this results in the majority of the training is influenced by random corruption, diminishing the effectiveness of the custom strategy. To examine this issue, we conducted an empirical analysis of negative pool sizes across four datasets: YAGO4-20, WN18, DBpedia50K and FB15K. We evaluated the ability of each negative sampling method to generate distinct true negatives as the desired number of negatives increased. This highlighted the operational limits of each approach in different datasets. In contrast, dynamic samplers behave similarly to random sampling in terms of pool size. However, they differ in that they apply a learned, triple-specific probability distribution to guide the sampling process based on embedding similarity. For this reason, they were excluded from this analysis.

4.2 Link Prediction Evaluation

Benchmark Datasets. Our experiments and analysis focus on two datasets: one schema-less WN18 [25] and one with additional schema information FB15K [6]. Their main statistics are reported in Tab. 2. WN18 is extracted from WordNet[3], an English-language lexical database where words are interlinked with conceptual semantics. FB15K is a subset of Freebase, an extensive database of general human knowledge organized into tuples. These datasets are canonical benchmarks widely adopted in the KGE literature. Additionally, their smaller scale wrt to other avialble KGs enables faster and more efficient experimentation for demonstrating the functionality of our extension.

Knowledge Graph Embedding Methods. We have selected twelve KGE methods from three different categories: (1) *translation-based* TransE [6], TransH, [35], TransR, [22] and TransD [16], (2) *semantic similarity-based* DistMult [36], RESCAL [28], ComplEx [33], SimplE [17], and (3) *geometric-based* RotatE [32], QuatE [37], BoxE [1], HolE [27]. This selection of models, which are already available in the *PyKEEN* ecosystem, ensures that our negative sampling implementations are tested across a diverse range of approaches. Thereby validating their general applicability and integration with different types of KGE architecture.

[3] `https://wordnet.princeton.edu/`

Negative Samplers. All of the negative samplers introduced in Sect. 2 have been included in the experimental evaluation, except for the nearest-neighbor-based strategy. This method was excluded due to its significantly higher computational demands, as it requires dense similarity computation for each triple, making it not well suited to the limited scope of this proof-of-concept study. Given our objective, we prioritized strategies that align with our goal of broadly applicable experimentation. To make the results comparable to those of other resources and studies, the sampling procedure was performed using the integration of random negatives when a strategy could not provide a large enough pool. To align our experiments with the proposed experimental design in [18], we used *RESCAL* as the auxiliary model.

Hyper-parameter Optimization. We conducted hyperparameter tuning to ensure fair and optimal conditions for all experiments. For each KGE model, and for each configuration of the number of negatives per positive triple, hyperparameters were optimized over 20 trials, with a time limit of 4 hours per trial. All models were trained using mini-batches and an L_2 regularization term, with the regularization weight λ searched in log scale within the range $[10^{-5}, 10^{-2}]$. Optimization was performed using the *Adam* optimizer, with the learning rate also searched in log scale over the range $[10^{-6}, 10^{-2}]$. A margin-based ranking loss was employed, selecting the margin γ from the set $1, 2, 5, 10$. The number of training epochs, batch size, and embedding dimensionality were fixed at 100, 500, and 100, respectively. This optimization procedure was repeated independently for each setting in the range of negatives per positive: $1, 2, 5, 20, 50, 100$.

Evaluation Protocol. Following the standard ranking-based evaluation protocol, we use the hit ratio with a cutoff value of 10 (Hit@10) as an evaluation metric. Hit@K measures the proportion of correct entities among the first n entities. The performance of KGE models was evaluated using filtered criteria to prevent known triples from influencing the ranking list.

5 Discussion of the Results and Comparative Analysis

5.1 Negative Pools Statistics

Tab. 3 reports the calculated statistics on the size of negative pools generated by each strategy across four datasets. The percentage values represent the proportion of triples with a negative pool size below a certain threshold, while 'Avg.' Pool column shows the average number of distinct negative entities available per triple (averaged over head and tail corruptions). This analysis is essential for assessing the practical feasibility and scalability of different sampling strategies. As shown in the 'Avg. Pool column, it is evident that all strategies produce significantly smaller candidate sets compared to random sampling. For example, in YAGO4-20, the Corrupt strategy reduces the entity pool from 96,910 to just 4,679 entities – approximately 4% of the total. This limitation is further highlighted in the 'Typed' strategy, where missing metadata lead to an

Table 3. Statistics of Negative Sampling Strategies

Dataset	**Sampling**	**Avg. Pool**	< 100	< 40	< 10	< 2	**No Pool**
YAGO4-20	Corrupt	4679	6.40	1.18	1.05	0.01	0.01
	Typed	1125	31.38	31.38	16.36	16.35	16.01
	Relational	7	99.76	99.33	76.25	34.13	10.82
DBpedia50	Corrupt	297	44.91	26.41	7.43	2.71	1.35
	Typed	316	69.13	54.52	37.96	34.41	21.03
	Relational	2	99.52	99.12	97.69	89.58	38.64
FB15K	Corrupt	439	45.01	30.81	20.73	10.87	3.45
	Typed	1295	44.50	26.12	12.17	4.45	1.49
	Relational	39	96.27	76.27	16.67	2.65	0.56
WN18	Corrupt	11367	0.78	0.70	0.00	0.00	0.00
	Typed	–	–	–	–	–	–
	Relational	2	99.90	99.64	96.98	61.54	5.32

even smaller negative pool. Relational approaches perform the worst, with up to 99% of triples having fewer than 100 different negatives across all datasets. This outcome is motivated by its core assumption that each source-target pair participates in only one relation. However, this assumption is too restrictive for datasets with 1-to-N, N-to-N, and N-to-1 relational patterns.

5.2 Analysis of Link Prediction Results

This work presents a proof-of-concept study validating the design and functionality of the proposed extension. Rather than exhaustively benchmarking all samplers, we aim to demonstrate how easily our strategies integrate into the standard *PyKEEN* link prediction pipeline and their empirical viability. To this end, we present the results of link prediction on FB15K and WN18, reporting the Hits@10 ranking metric across multiple values of negative triples generated per positive triple as shown in Figs. 2,3

Results confirm that sampling strategies affect model performance differently depending on dataset structure and sampling design. These effects reflect both the structural properties of the graph and the design of the sampling algorithm. As well as verifying the correctness and usability of our library, the experiments emphasize that negative sampling choices can have a significant impact on model performance and should therefore be made with an awareness of the characteristics of the dataset and the evaluation objectives.

In FB15K, strategies such as Bernoulli sampling produce strong results across various embedding models, while adversarial remains the least effective. This can be attributed to RESCAL, the auxiliary model used for prediction in this dynamic corruption strategy. According to our evaluation, RESCAL was one of the lowest performing models, which significantly affected the performance of the adversarial sampler.

Taking into account the overall evaluation on negative pool sizes and link prediction metrics, we observe that increasing the number of negative samples

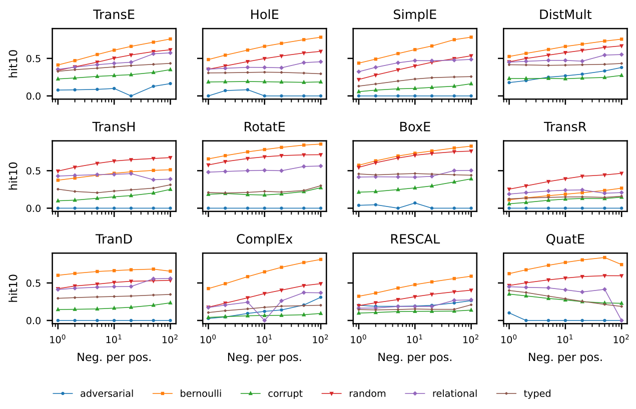

Fig. 2. Hits@10 evaluation on FB15K test set

per positive instance has a negligible impact on performance, with similar metrics across all models and negative samplers. This can be explained by the use of integration of random negative in the corruption strategy when the negative sampler failed to produce enough candidates. As the analysis shows, many strategies struggle to maintain a meaningful negative pool at higher negative sample sizes. Consequently, as the required number of negatives increases, the majority of training triples become corrupted through random sampling, which reduces the effectiveness of the intended sampling strategy. This results in the sampling behavior becoming increasingly similar to the random corruption scheme as the number of negatives increases.

These results highlight the limitations of static samplers and suggest practical considerations for their application in research. One key takeaway from this evaluation is the importance of understanding the operational boundaries and dataset compatibility of different negative sampling strategies. It also highlights the need for more in-depth investigations into how negative samplers interact with graph structure and influence model behavior.

6 Conclusions and Future Works

This work introduced and validated a modular extension to the PyKEEN framework. The extension has been designed for proving a wide coverage of standardised negative sampler implementations when adopting KGE methods thus filling an important gap of the availability of more advanced implemented negative samplers. We specifically provided a fully compatible implementation of five negative samplers with static and dynamic corruption strategies. We demonstrated the practical utility through a range of experiments showcasing how the extension can be seamlessly integrated into the training, evaluation and hyperparameter

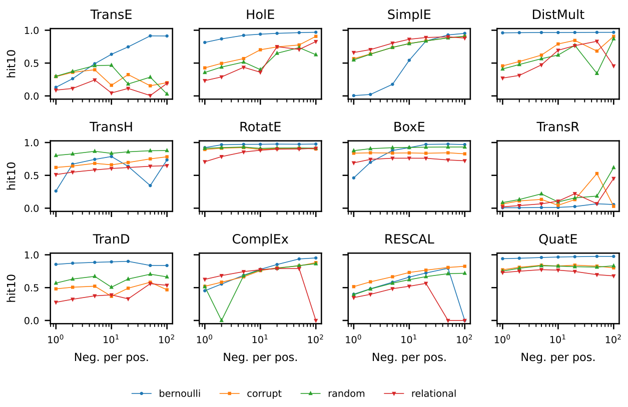

Fig. 3. Hits@10 evaluation on WN18 test set

optimisation pipelines of KGE. Furthermore, we presented negative pool statistics for four commonly used datasets, highlighting the operational constraints and behaviour of various sampling strategies in different conditions. Adhering to PyKEEN architecture and design standards, the proposed extension supports reproducibility, modularity, and ease of experimentation. By reducing the barriers to developing and evaluating new sampling strategies, our goal is to foster a more unified and efficient research ecosystem.

Future developments will focus on additional negative sampling methods and a broader experimental evaluation across more diverse datasets. An important enhancement lies in performance optimisation, specifically through caching strategies and algorithms to accelerate computation. We also foresee the potential for parallel and distributed implementation.

Resource Availability Statement: Source code, pretrained model weights, documentation and datasets (YAGO4-20, DB50K, FB15K, WN18) with their relevant metadata are available from GitHub [4] and Zenodo [5] (the most up to date information is on GitHub). Readthedocs style documentation can be found on GitHub Pages [6]

Acknowledgments. This work was partially supported by project *FAIR - Future AI Research* (PE00000013), spoke 6 - Symbiotic AI (`https://future-ai-research.it/`) under the PNRR MUR program funded by the European Union - NextGenerationEU, and by PRIN project *HypeKG - Hybrid Prediction and Explanation with Knowledge Graphs* (Prot. 2022Y34XNM, CUP H53D23003700006) under the PNRR MUR program funded by the European Union - NextGenerationEU

[4] https://github.com/ivandiliso/refactor-negative-sampler/
[5] https://doi.org/10.5281/zenodo.15413075
[6] https://ivandiliso.github.io/refactor-negative-sampler

References

1. Abboud, R., Ceylan, I., Lukasiewicz, T., Salvatori, T.: Boxe: A box embedding model for knowledge base completion. Advances in Neural Information Processing Systems **33**, 9649–9661 (2020)
2. Alam, M.M., Jabeen, H., Ali, M., Mohiuddin, K., Lehmann, J.: Affinity dependent negative sampling for knowledge graph embeddings. (2020)
3. Ali, M., Berrendorf, M., Hoyt, C.T., Vermue, L., Sharifzadeh, S., Tresp, V., Lehmann, J.: PyKEEN 1.0: A Python Library for Training and Evaluating Knowledge Graph Embeddings. Journal of Machine Learning Research **22**(82), 1–6 (2021), `http://jmlr.org/papers/v22/20-825.html`
4. Barile, R., d'Amato, C., Fanizzi, N.: Explanation of link predictions on knowledge graphs via levelwise filtering and graph summarization. In: European Semantic Web Conference. pp. 180–198. Springer (2024)
5. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., Taylor, J.: Freebase: a collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 1247–1250 (2008)
6. Bordes, A., Usunier, N., Garcia-Duran, A., Weston, J., Yakhnenko, O.: Translating embeddings for modeling multi-relational data. Advances in neural information processing systems **26** (2013)
7. Boschin, A.: Torchkge: Knowledge graph embedding in python and pytorch. In: International Workshop on Knowledge Graph: Mining Knowledge Graph for Deep Insights (Aug 2020)
8. Broscheit, S., Ruffinelli, D., Kochsiek, A., Betz, P., Gemulla, R.: LibKGE - A knowledge graph embedding library for reproducible research. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 165–174 (2020), `https://www.aclweb.org/anthology/2020.emnlp-demos.22`
9. Cai, L., Wang, W.Y.: KBGAN: Adversarial learning for knowledge graph embeddings. In: Walker, M., Ji, H., Stent, A. (eds.) Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers). pp. 1470–1480. Association for Computational Linguistics, New Orleans, Louisiana (Jun 2018). https://doi.org/10.18653/v1/N18-1133, `https://aclanthology.org/N18-1133`
10. Chen, X., Zhang, W., Yao, Z., Chen, M., Tang, S.: Negative sampling with adaptive denoising mixup for knowledge graph embedding. In: International Semantic Web Conference. pp. 253–270. Springer (2023)
11. Costabello, L., Bernardi, A., Janik, A., Creo, A., Pai, S., Van, C.L., McGrath, R., McCarthy, N., Tabacof, P.: AmpliGraph: a Library for Representation Learning on Knowledge Graphs (Mar 2019). https://doi.org/10.5281/zenodo.2595043, `https://doi.org/10.5281/zenodo.2595043`
12. Dash, S., Gliozzo, A.: Distributional negative sampling for knowledge base completion. arXiv preprint arXiv:1908.06178 (2019)
13. Gashteovski, K., Gemulla, R., Kotnis, B., Hertling, S., Meilicke, C.: On aligning openie extractions with knowledge bases: A case study. Association for Computational Linguistics (ACL) (2020)
14. Han, X., Cao, S., Xin, L., Lin, Y., Liu, Z., Sun, M., Li, J.: Openke: An open toolkit for knowledge embedding. In: Proceedings of EMNLP (2018)

15. Je, S.H.: Entity aware negative sampling with auxiliary loss of false negative prediction for knowledge graph embedding. arXiv preprint arXiv:2210.06242 (2022)
16. Ji, G., He, S., Xu, L., Liu, K., Zhao, J.: Knowledge graph embedding via dynamic mapping matrix. In: Proceedings of the 53rd annual meeting of the association for computational linguistics and the 7th international joint conference on natural language processing (volume 1: Long papers). pp. 687–696 (2015)
17. Kazemi, S.M., Poole, D.: Simple embedding for link prediction in knowledge graphs. Advances in neural information processing systems **31** (2018)
18. Kotnis, B., Nastase, V.: Analysis of the impact of negative sampling on link prediction in knowledge graphs. arXiv preprint arXiv:1708.06816 (2017)
19. Krompaß, D., Baier, S., Tresp, V.: Type-constrained representation learning in knowledge graphs. In: The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part I 14. pp. 640–655. Springer (2015)
20. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., et al.: Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic web **6**(2), 167–195 (2015)
21. Li, D., Qu, H., Wang, J.: A survey on knowledge graph-based recommender systems. In: 2023 China Automation Congress (CAC). pp. 2925–2930. IEEE (2023)
22. Lin, Y., Liu, Z., Sun, M., Liu, Y., Zhu, X.: Learning entity and relation embeddings for knowledge graph completion. In: Proceedings of the AAAI conference on artificial intelligence. vol. 29 (2015)
23. Madushanka, T., Ichise, R.: Negative sampling in knowledge graph representation learning: A review. arXiv preprint arXiv:2402.19195 (2024)
24. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems **26** (2013)
25. Miller, G.A.: Wordnet: a lexical database for english. Communications of the ACM **38**(11), 39–41 (1995)
26. Nickel, M., Murphy, K., Tresp, V., Gabrilovich, E.: A review of relational machine learning for knowledge graphs. Proceedings of the IEEE **104**(1), 11–33 (2016). https://doi.org/10.1109/JPROC.2015.2483592
27. Nickel, M., Rosasco, L., Poggio, T.: Holographic embeddings of knowledge graphs. In: Proceedings of the AAAI conference on artificial intelligence. vol. 30 (2016)
28. Nickel, M., Tresp, V., Kriegel, H.P., et al.: A three-way model for collective learning on multi-relational data. In: Icml. vol. 11, pp. 3104482–3104584 (2011)
29. Pellissier Tanon, T., Weikum, G., Suchanek, F.: Yago 4: A reason-able knowledge base. In: The Semantic Web: 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31–June 4, 2020, Proceedings 17. pp. 583–596. Springer (2020)
30. Senaratne, A., Omran, P.G., Christen, P., Williams, G.: Tric: A triples corrupter for knowledge graphs. In: European Semantic Web Conference. pp. 117–122. Springer (2023)
31. Socher, R., Chen, D., Manning, C.D., Ng, A.: Reasoning with neural tensor networks for knowledge base completion. Advances in neural information processing systems **26** (2013)
32. Sun, Z., Deng, Z.H., Nie, J.Y., Tang, J.: Rotate: Knowledge graph embedding by relational rotation in complex space. arXiv preprint arXiv:1902.10197 (2019)

33. Trouillon, T., Welbl, J., Riedel, S., Gaussier, É., Bouchard, G.: Complex embeddings for simple link prediction. In: International conference on machine learning. pp. 2071–2080. PMLR (2016)
34. Wang, C., Zhu, Z., Meng, P., Qiu, Y.: Leveraging network structure for efficient dynamic negative sampling in network embedding. Information Sciences **606**, 853–863 (2022)
35. Wang, Z., Zhang, J., Feng, J., Chen, Z.: Knowledge graph embedding by translating on hyperplanes. In: Proceedings of the AAAI conference on artificial intelligence. vol. 28 (2014)
36. Yang, B., Yih, W.t., He, X., Gao, J., Deng, L.: Embedding entities and relations for learning and inference in knowledge bases. arXiv preprint arXiv:1412.6575 (2014)
37. Zhang, S., Tay, Y., Yao, L., Liu, Q.: Quaternion knowledge graph embeddings. Advances in neural information processing systems **32** (2019)
38. Zhang, Y., Yao, Q., Shao, Y., Chen, L.: Nscaching: simple and efficient negative sampling for knowledge graph embedding. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE). pp. 614–625. IEEE (2019)
39. Zheng, D., Song, X., Ma, C., Tan, Z., Ye, Z., Dong, J., Xiong, H., Zhang, Z., Karypis, G.: Dgl-ke: Training knowledge graph embeddings at scale. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. p. 739–748. SIGIR '20, Association for Computing Machinery, New York, NY, USA (2020)
40. Zhu, Z., Xu, S., Qu, M., Tang, J.: Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In: The World Wide Web Conference. pp. 2494–2504. ACM (2019)