Understanding the Evaluation

This part of the tutorial is aimed to help you understand the evaluation of knowledge graph embeddings. In particular it explains rank-based evaluation metrics reported in `pykeen.evaluation.RankBasedMetricResults`.

Knowledge graph embedding are usually evaluated on the task of link prediction. To this end, an evaluation set of triples $\mathcal{T}_{eval} \subset \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ is provided, and for each triple $(h, r, t) \in \mathcal{T}_{eval}$ in this set, two tasks are solved:

- **Right-Side** In the *right-side* prediction task, a pair of head entity and relation are given and aim to predict the tail, i.e. $(h, r, ?)$. To this end, the knowledge graph embedding model is used to *score* each of the possible choices (h, r, e) for $e \in \mathcal{E}$. Higher scores indicate higher plausibility.
- **Left-Side** Analogously, in the *left-side* prediction task, a pair of relation and tail entity are provided and aim to predict the head, i.e. $(?, r, t)$. Again, each possible choice (e, r, t) for $e \in \mathcal{E}$ is scored according to the knowledge graph embedding model.

> **Note**
>
> Practically, many embedding models allow fast computation of all scores (e, r, t) for all $e \in \mathcal{E}$, than just passing the triples through the model's score function. As an example, consider DistMult with the score function $score(h, r, t) = \sum_{i=1}^{d} \mathbf{h}_i \cdot \mathbf{r}_i \cdot \mathbf{t}_i$. Here, all entities can be scored as candidate heads for a given tail and relation by first computing the element-wise product of tail and relation, and then performing a matrix multiplication with the matrix of all entity embeddings. # TODO: Link to section explaining this concept.

In the rank-based evaluation protocol, the scores are used to sort the list of possible choices by decreasing score, and determine the *rank* of the true choice, i.e. the index in the sorted list. Smaller ranks indicate better performance. Based on these individual ranks, which are obtained for each evaluation triple and each side of the prediction (left/right), there exist several aggregation measures to quantify the performance of a model in a single number.

> **Note**
>
> There are theoretical implications based on whether the indexing is 0-based or 1-based (natural). PyKEEN uses 1-based indexing to conform with related work.

As an example, consider we trained a KGEM on the countries dataset, e.g., using

```
from pykeen.datasets import get_dataset
from pykeen.pipeline import pipeline
dataset = get_dataset(dataset="countries")
result = pipeline(dataset=dataset, model="mure")
```

During evaluation time, we now evaluate head and tail prediction, i.e., whether we can correct the correct head/tail entity from the remainder of a triple. The first triple in the test split of this dataset is *['belgium', 'locatedin', 'europe']*. Thus, for tail prediction, we aim to answer *['belgium', 'locatedin', ?]*. We can see the results using the prediction workflow:

```
from pykeen.models.predict import get_tail_prediction_df

df = get_tail_prediction_df(
    model=result.model,
    head_label="belgium",
    relation_label="locatedin",
    triples_factory=result.training,
    add_novelties=False,
)
```

which returns a dataframe of all tail candidate entities sorted according to the predicted score. The index in this sorted list is essentially the *rank* of the correct answer.

Rank-Based Metrics

Given the set of individual rank scores for each head/tail entity from evaluation triples, there are various aggregation metrics which summarize different aspects of the set of ranks into a single-figure number. For more details, please refer to their documentation.

Ranking Types

While the aforementioned definition of the rank as "the index in the sorted list" is intuitive, it does not specify what happens when there are multiple choices with exactly the same score. Therefore, in previous work, different variants have been implemented, which yield different results in the presence of equal scores.

- The *optimistic* rank assumes that the true choice is on the first position of all those with equal score.
- The *pessimistic* rank assumes that the true choice is on the last position of all those with equal score.
- The *realistic* rank is the mean of the optimistic and the pessimistic rank, and moreover the expected value over all permutations respecting the sort order.
- The *non-deterministic* rank delegates the decision to the sort algorithm. Thus, the result depends on the internal tie breaking mechanism of the sort algorithm's implementation.

PyKEEN supports the first three: optimistic, pessimistic and realistic. When only using a single score, the realistic score should be reported. The pessimistic and optimistic rank, or more specific the deviation between both, can be used to detect whether a model predicts exactly equal scores for many choices. There are a few causes such as:

- finite-precision arithmetic in conjunction with explicitly using sigmoid activation
- clamping of scores, e.g. by using a ReLU activation or similar.

Ranking Sidedness

Besides the different rank definitions, PyKEEN also report scores for the individual side predictions.

Side	Explanation
head	The rank-based metric evaluated only for the head / left-side prediction.
tail	The rank-based metric evaluated only for the tail / right-side prediction.
both	The rank-based metric evaluated on both predictions.

By default, “both” is often used in publications. The side-specific scores can however often give access to interesting insights, such as the difference in difficulty of predicting a head/tail given the rest, or the model’s incapability to solve of one the tasks.

Ranking Aggregation Scope

Real graphs often are scale-free, i.e., there are a few nodes / entities which have a high degree, often called hub, while the majority of nodes has only a few neighbors. This also impacts the evaluation triples: since the hub nodes occur in a large number of triples, they are also more likely to be part of evaluation triples. Thus, performing well on triples containing hub entities contributes strongly to the overall performance.

As an example, we can inspect the `pykeen.datasets.WD50KT` dataset, where a single (relation, tail)-combination, (“instance of”, “human”), is present in 699 evaluation triples.

```
from pykeen.datasets import get_dataset
ds = get_dataset(dataset="wd50kt")
unique_relation_tail, counts = dataset.testing.mapped_triples[:, 1:].unique(return_counts=True,
dim=0)
# c = 699
c = counts.max()
r, t = unique_relation_tail[counts.argmax()]
# https://www.wikidata.org/wiki/Q5 -> "human"
t = dataset.testing.entity_id_to_label[t.item()]
# https://www.wikidata.org/wiki/Property:P31 -> "instance of"
r = dataset.testing.relation_id_to_label[r.item()]
```

There are arguments that we want these entities to have a strong effect on evaluation: since they occur often, they are seemingly important, and thus evaluation should reflect that. However, sometimes we also do *not* want to have this effect, but rather measure the performance evenly across nodes. A similar phenomenon also exists in multi-class classification with imbalanced classes, where frequent classes can dominate performance measures. In similar vein to the macro F_1-score (cf. `sklearn.metrics.f1_score()`) known from this area, PyKEEN implements a `pykeen.evaluation.MacroRankBasedEvaluator`, which ensure that triples are weighted such that each unique ranking task, e.g., a (head, relation)-pair for tail prediction, contributes evenly.

Technically, we solve the task by implemented variants of existing rank-based metrics which support weighting individual ranks differently. Moreover, the evaluator computes weights inversely proportional to the "query" part of the ranking task, i.e., e.g., (head, relation) for tail prediction.

Filtering

The rank-based evaluation allows using the "filtered setting", proposed by [bordes2013], which is enabled by default. When evaluating the tail prediction for a triple (h, r, t), i.e. scoring all triples (h, r, e), there may be additional known triples (h, r, t') for $t \neq t'$. If the model predicts a higher score for (h, r, t'), the rank will increase, and hence the measured model performance will decrease. However, giving (h, r, t') a high score (and thus a low rank) is desirable since it is a true triple as well. Thus, the filtered evaluation setting ignores for a given triple (h, r, t) the scores of all other *known* true triples (h, r, t').

Below, we present the philosophy from [bordes2013] and how it is implemented in PyKEEN:

HPO Scenario

During training/optimization with `pykeen.hpo.hpo_pipeline()`, the set of known positive triples comprises the training and validation sets. After optimization is finished and the final evaluation is done, the set of known positive triples comprises the training, validation, and testing set. PyKEEN explicitly does not use test triples for filtering during HPO to avoid any test leakage.

Early Stopper Scenario

When early stopping is used during training, it periodically uses the validation set for calculating the loss and evaluation metrics. During this evaluation, the set of known positive triples comprises the training and validation sets. When final evaluation is done with the testing set, the set of known positive triples comprises the training, validation, and testing set. PyKEEN explicitly does not use test triples for filtering when early stopping is being used to avoid any test leakage.

Pipeline Scenario

During vanilla training with the `pykeen.pipeline.pipeline()` that has no optimization, no early stopping, nor any *post-hoc* choices using the validation set, the set of known positive triples comprises the training and testing sets. This scenario is very atypical, and regardless, should be augmented with the validation triples to make more comparable to other published results that do not consider this scenario.

Custom Training Loops

In case the validation triples should *not* be filtered when evaluating the test dataset, the argument `filter_validation_when_testing=False` can be passed to either the `pykeen.hpo.hpo_pipeline()` or `pykeen.pipeline.pipeline()`.

If you're rolling your own pipeline, you should keep the following in mind: the `pykeen.evaluation.Evaluator` when in the filtered setting with `filtered=True` will always use the evaluation set (regardless of whether it is the testing set or validation set) for filtering. Any other triples that should be filtered should be passed to `additional_filter_triples` in `pykeen.evaluation.Evaluator.evaluate()`. Typically, this minimally includes the training triples. With the [bordes2013] technique where the testing set is used for evaluation, the `additional_filter_triples` should include both the training triples and validation triples as in the following example:

```python
from pykeen.datasets import FB15k237
from pykeen.evaluation import RankBasedEvaluator
from pykeen.models import TransE

# Get FB15k-237 dataset
dataset = FB15k237()

# Define model
model = TransE(
    triples_factory=dataset.training,
)

# Train your model (code is omitted for brevity)
...

# Define evaluator
evaluator = RankBasedEvaluator(
    filtered=True,  # Note: this is True by default; we're just being explicit
)

# Evaluate your model with not only testing triples,
# but also filter on validation triples
results = evaluator.evaluate(
    model=model,
    mapped_triples=dataset.testing.mapped_triples,
    additional_filter_triples=[
        dataset.training.mapped_triples,
        dataset.validation.mapped_triples,
    ],
)
```

Entity and Relation Restriction

Sometimes, we are only interested in a certain set of entities and/or relations, $\mathcal{E}_I \subset \mathcal{E}$ and $\mathcal{R}_I \subset \mathcal{R}$ respectively, but have additional information available in form of triples with other entities/relations. As example, we would like to predict whether an actor stars in a movie. Thus, we are only interested in the relation *stars_in* between entities which are actors/movies. However, we may have additional information available, e.g. who directed the movie, or the movie's language, which may help in the prediction task. Thus, we would like to train the model on the full dataset including all available relations and entities, but restrict the evaluation to the task we are aiming at.

In order to restrict the evaluation, we proceed as follows:

1. We filter the evaluation triples $\mathcal{T}_{eval}$ to contain only those triples which are of interest, i.e. $\mathcal{T}'_{eval} = \{(h, r, t) \in \mathcal{T}_{eval} \mid h, t \in \mathcal{E}_I, r \in \mathcal{R}_I\}$
2. During tail prediction/evaluation for a triple (h, r, t), we restrict the candidate tail entity t' to $t' \in \mathcal{E}_{eval}$. Similarly for head prediction/evaluation, we restrict the candidate head entity h' to $h' \in \mathcal{E}_{eval}$

Example

The `pykeen.datasets.Hetionet` is a biomedical knowledge graph containing drugs, genes, diseases, other biological entities, and their interrelations. It was described by Himmelstein *et al.* in Systematic integration of biomedical knowledge prioritizes drugs for repurposing to support drug repositioning, which translates to the link prediction task between drug and disease nodes.

The edges in the graph are listed here, but we will focus on only the compound treat disease (CtD) and compound palliates disease (CpD) relations during evaluation. This can be done with the following:

```
from pykeen.pipeline import pipeline

evaluation_relation_whitelist = {'CtD', 'CpD'}
pipeline_result = pipeline(
    dataset='Hetionet',
    model='RotatE',
    evaluation_relation_whitelist=evaluation_relation_whitelist,
)
```

By restricting evaluation to the edges of interest, models more appropriate for drug repositioning can be identified during hyper-parameter optimization instead of models that are good at predicting all types of relations. The HPO pipeline accepts the same arguments:

```
from pykeen.hpo import hpo_pipeline

evaluation_relation_whitelist = {'CtD', 'CpD'}
hpo_pipeline_result = hpo_pipeline(
    n_trials=30,
    dataset='Hetionet',
    model='RotatE',
    evaluation_relation_whitelist=evaluation_relation_whitelist,
)
```