Prediction

Prediction workflows.

After training, the interaction model (e.g., TransE, ConvE, RotatE) can assign a score to an arbitrary triple, whether it appeared during training, testing, or not. In PyKEEN, each is implemented such that the higher the score (or less negative the score), the more likely a triple is to be true.

However, for most models, these scores do not have obvious statistical interpretations. This has two main consequences:

1. The score for a triple from one model can not be compared to the score for that triple from another model
2. There is no *a priori* minimum score for a triple to be labeled as true, so predictions must be given as a prioritization by sorting a set of triples by their respective scores.

For the remainder of this part of the documentation, we assume that we have trained a model, e.g. via

```
>>> from pykeen.pipeline import pipeline
>>> result = pipeline(dataset="nations", model="pairre", training_kwargs=dict(num_epochs=0))
```

High-Level

The prediction workflow offers three high-level methods to perform predictions

- `pykeen.predict.predict_triples()` can be used to calculate scores for a given set of triples.
- `pykeen.predict.predict_target()` can be used to score choices for a given prediction target, i.e. calculate scores for head entities, relations, or tail entities given the other two.
- `pykeen.predict.predict_all()` can be used to calculate scores for all possible triples. Scientifically, `pykeen.predict.predict_all()` is the most interesting in a scenario where predictions could be tested and validated experimentally.

Warning

Please note that not all models automatically have interpretable scores, and their calibration may be poor. Thus, exercise caution when interpreting the results.

Triple Scoring

When scoring triples with `pykeen.predict.predict_triples()`, we obtain a score for each of the given triples. As an example, we will calculate scores for all validation triples from the dataset we trained the model upon.

```
>>> from pykeen.datasets import get_dataset
>>> from pykeen.predict import predict_triples
>>> dataset = get_dataset(dataset="nations")
>>> pack = predict_triples(model=result.model, triples=dataset.validation)
```

The variable `pack` now contains a `pykeen.predict.ScorePack`, which essentially is a pair of ID-based triples with their predicted scores. For interpretation, it can be helpful to add their corresponding labels, which the *"nations"* dataset offers, and convert them to a pandas dataframe:

```
>>> df = pack.process(factory=result.training).df
```

Since we now have a dataframe, we can utilize the full power of pandas for our subsequent analysis, e.g., showing the triples which received the highest score

```
>>> df.nlargest(n=5, columns="score")
```

or investigate whether certain entities generally receive larger scores

```
>>> df.groupby(by=["head_id", "head_label"]).agg({"score": ["mean", "std", "count"]})
```

Target Scoring

`pykeen.predict.predict_target()`'s primary usecase is link prediction or relation prediction. For instance, we could use our models to score all possible tail entities for the query *("uk", "conferences", ?)* via

```
>>> from pykeen.datasets import get_dataset
>>> from pykeen.predict import predict_target
>>> dataset = get_dataset(dataset="nations")
>>> pred = predict_target(
...     model=result.model,
...     head="uk",
...     relation="conferences",
...     triples_factory=result.training,
... )
```

Notice that the result stored into *pred* is a `pykeen.predict.Predictions` object, which offers some post-processing options. For instance, we can remove all targets which are already know from the training set

```
>>> pred_filtered = pred.filter_triples(dataset.training)
```

or add additional columns to the dataframe proving the information whether the target is contained in another set, e.g., the validation or testing set.

```
>>> pred_annotated = pred_filtered.add_membership_columns(validation=dataset.validation,
testing=dataset.testing)
```

The predictions object also exposes filtered / annotated dataframe through its *df* attribute

```
>>> pred_annotated.df
```

Full Scoring

Finally, we can use `pykeen.predict.predict()` to calculate scores for *all* possible triples. Notice that this operation can be prohibitively expensive for reasonably sized knowledge graphs, and the model may produce additional ill-calibrated scores for entity/relation combinations it has never seen paired before during training. The next line calculates *and* stores all triples and scores

```
>>> from pykeen.predict import predict_all
>>> pack = predict_all(model=result.model)
```

In addition to the expensive calculations, this additionally requires us to have sufficient memory available to store all scores. A computationally equally expensive option with reduced, fixed memory requirement is to store only the triples with the top k scores. This can be done through the optional parameter *k*

```
>>> pack = predict_all(model=result.model, k=10)
```

We can again convert the score pack to a predictions object for further filtering, e.g., adding a column indicating whether the triple has been seen during training

```
>>> pred = pack.process(factory=result.training)
>>> pred_annotated = pred.add_membership_columns(training=result.training)
>>> pred_annotated.df
```

Low-Level

The following section outlines some details about the implementation of operations which require calculating scores for all triples. The algorithm works are follows:

```
for batch in DataLoader(dataset, batch_size=batch_size):
  scores = model.predict(batch)
  for consumer in consumers:
    consumer(batch, scores)
```

Here, *dataset* is a `pykeen.predict.PredictionDataset`, which breaks the score calculation down into individual target predictions (e.g., tail predictions). Implementations include `pykeen.predict.AllPredictionDataset` and `pykeen.predict.PartiallyRestrictedPredictionDataset`. Notice that the prediction tasks are built lazily, i.e., only instantiating the prediction tasks when accessed. Moreover, the `torch_max_mem` package is used to automatically tune the batch size to maximize the memory utilization of the hardware at hand.

For each batch, the scores of the prediction task are calculated once. Afterwards, multiple *consumers* can process these scores. A consumer extends `pykeen.predict.ScoreConsumer` and receives the batch, i.e., input to the predict method, as well as the tensor of predicted scores. Examples include

- `pykeen.predict.CountScoreConsumer`: a simple consumer which only counts how many scores it has seen. Mostly used for debugging or testing purposes
- `pykeen.predict.AllScoreConsumer`: accumulates all scores into a single huge tensor. This incurs massive memory requirements for reasonably sized datasets, and often can be avoided by interleaving the processing of the scores with calculation of individual batches.
- `pykeen.predict.TopKScoreConsumer`: keeps only the top k scores as well as the inputs leading to them. This is a memory-efficient variant of first accumulating all scores, then sorting by score and keeping only the top entries.

Potential Caveats

The model is trained on a particular link prediction task, e.g. to predict the appropriate tail for a given head/relation pair. This means that while the model can technically also predict other links, e.g., relations between a given head/tail pair, it must be done with the caveat that it was not trained for this task, and thus its scores may behave unexpectedly.

Migration Guide

Until version 1.9, the model itself provided wrappers which would delegate to the corresponding method in *pykeen.models.predict*

- *model.get_all_prediction_df*
- *model.get_prediction_df*
- *model.get_head_prediction_df*
- *model.get_relation_prediction_df*
- *model.get_tail_prediction_df*

These methods were already deprecated and could be replaced by providing the model as explicit parameter to the stand-alone functions from the prediction module. Thus, we will focus on the migrating the stand-alone functions.

In the *pykeen.models.predict* module, the prediction methods were organized differently. There were

- *get_prediction_df*
- *get_head_prediction_df*
- *get_relation_prediction_df*
- *get_tail_prediction_df*
- *get_all_prediction_df*
- *predict_triples_df*

where *get_head_prediction_df*, *get_relation_prediction_df* and *get_tail_prediction_df* were deprecated in favour of directly using *get_prediction_df* with all but the prediction target being provided, i.e., e.g.,

```
>>> from pykeen.models import predict
>>> prediction.get_tail_prediction_df(
...     model=model,
...     head_label="belgium",
...     relation_label="locatedin",
...     triples_factory=result.training,
... )
```

was deprecated in favour of

```
>>> from pykeen.models import predict
>>> predict.get_prediction_df(
...     model=model,
...     head_label="brazil",
...     relation_label="intergovorgs",
...     triples_factory=result.training,
... )
```

get_prediction_df

The old use of

```
>>> from pykeen.models import predict
>>> predict.get_prediction_df(
...     model=model,
...     head_label="brazil",
...     relation_label="intergovorgs",
...     triples_factory=result.training,
... )
```

can be replaced by

```
>>> from pykeen import predict
>>> predict.predict_target(
...     model=model,
...     head="brazil",
...     relation="intergovorgs",
...     triples_factory=result.training,
... ).df
```

Notice the trailing *.df*.

get_all_prediction_df

The old use of

```
>>> from pykeen.models import predict
>>> predictions_df = predict.get_all_prediction_df(model, triples_factory=result.training)
```

can be replaced by

```
>>> from pykeen import predict
>>> predict.predict_all(model=model, triples_factory=result.training).process().df
```

predict_triples_df

The old use of

```
>>> from pykeen.models import predict
>>> score_df = predict.predict_triples_df(
...     model=model,
...     triples=[("brazil", "conferences", "uk"), ("brazil", "intergovorgs", "uk")],
...     triples_factory=result.training,
... )
```

can be replaced by

```
>>> from pykeen import predict
>>> score_df = predict.predict_triples(
...     model=model,
...     triples=[("brazil", "conferences", "uk"), ("brazil", "intergovorgs", "uk")],
...     triples_factory=result.training,
... )
```

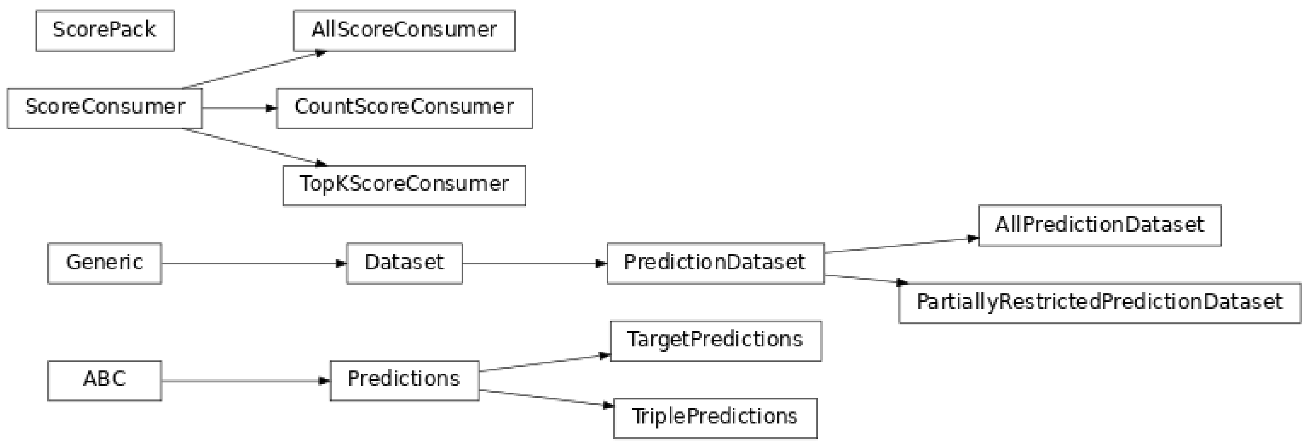
Functions

`predict_all` (model, *[, k, batch_size, mode, ...])	Calculate scores for all triples, and either keep al
`predict_triples` (model, *, triples[, ...])	Predict on labeled or mapped triples.
`predict_target` (model, *[, head, relation, ...])	Get predictions for the head, relation, and/or tai
`consume_scores` (model, dataset, *consumers[, ...])	Batch-wise calculation of all triple scores and co

Classes

`ScoreConsumer` ()	A consumer of scores for visitor pattern.
`CountScoreConsumer` ()	A simple consumer which counts the number of ba
`TopKScoreConsumer` ([k, device])	Collect top-k triples & scores.
`AllScoreConsumer` (num_entities, num_relations)	Collect scores for all triples.
`CountScoreConsumer` ()	A simple consumer which counts the number of ba
`ScorePack` (result, scores)	A pair of result triples and scores.
`Predictions` (df, factory)	Base class for predictions.
`TriplePredictions` (df, factory)	Triples with their predicted scores.
`TargetPredictions` (df, factory, target, ...)	Targets with their predicted scores.
`PredictionDataset` ([target])	A base class for prediction datasets.
`AllPredictionDataset` (num_entities, ...)	A dataset for predicting all possible triples.
`PartiallyRestrictedPredictionDataset` (*[, ...])	A dataset for scoring some links.

Class Inheritance Diagram


ScorePack
AllScoreConsumer
ScoreConsumer
CountScoreConsumer
TopKScoreConsumer
AllPredictionDataset
Generic
Dataset
PredictionDataset
PartiallyRestrictedPredictionDataset
TargetPredictions
ABC
Predictions
TriplePredictions