Running an Ablation Study

You want to find out which loss function and training approach is best-suited for your interaction model (model architecture)? Then performing an ablation study is the way to go!

In general, an ablation study is a set of experiments in which components of a machine learning system are removed/replaced in order to measure the impact of these components on the performance of the system. In the context of knowledge graph embedding models, typical ablation studies involve investigating different loss functions, training approaches, negative samplers, and the explicit modeling of inverse relations. For a specific model composition based on these components, the best set of hyper-parameter values, e.g., embedding dimension, learning rate, batch size, loss function-specific hyper-parameters such as the margin value in the margin ranking loss need to be determined. This is accomplished by a process called hyper-parameter optimization. Different approaches have been proposed, of which random search and grid search are very popular.

In PyKEEN, we can define execute an ablation study within our own program or from the command line interface using a configuration file (`file_name.json`).

First, we show how to run an ablation study within your program. For this purpose, we provide the function `pykeen.ablation.ablation_pipeline()` that requires the `datasets`, `models`, `losses`, `optimizers`, `training_loops`, and `directory` arguments to define the datasets, models, loss functions, optimizers (e.g., Adam), training approaches for our ablation study, and the output directory in which the experimental artifacts should be saved. In the following, we define an ablation study for `pykeen.models.ComplEx` over the `pykeen.datasets.Nations` dataset in order to assess the effect of different loss functions (in our example, the binary cross entropy loss and the margin ranking loss) and the effect of explicitly modeling inverse relations.

Now, let's start with defining the minimal requirements, i.e., the dataset(s), interaction model(s), the loss function(s), training approach(es), and the optimizer(s) in order to run the ablation study.

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex01_minimal"
>>> ablation_pipeline(
...     directory=directory,
...     models=["ComplEx"],
...     datasets=["Nations"],
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops=["LCWA"],
...     optimizers=["Adam"],
...     # The following are not part of minimal configuration, but are necessary
...     # for demonstration/doctests. You should make these numbers bigger when
...     # you're using PyKEEN's ablation framework
...     epochs=1,
...     n_trials=1,
... )
```

We can provide arbitrary additional information about our study with the `metadata` keyword. Some keys, such as `title` are special and used by PyKEEN and `optuna`.

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex02_metadata"
>>> ablation_pipeline(
...     directory=directory,
...     models=["ComplEx"],
...     datasets=["Nations"],
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops=["LCWA"],
...     optimizers=["Adam"],
...     # Add metadata with:
...     metadata=dict(
...         title="Ablation Study Over Nations for ComplEx.",
...     ),
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     n_trials=1,
... )
```

As mentioned above, we also want to measure the effect of explicitly modeling inverse relations on the model's performance. Therefore, we extend the ablation study by including the `create_inverse_triples` argument:

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex03_inverse"
>>> ablation_pipeline(
...     directory=directory,
...     models=["ComplEx"],
...     datasets=["Nations"],
...     losses=["BCEAfterSigmoidLoss"],
...     training_loops=["LCWA"],
...     optimizers=["Adam"],
...     # Add inverse triples with
...     create_inverse_triples=[True, False],
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     n_trials=1,
... )
```

Note

Unlike `models`, `datasets`, `losses`, `training_loops`, and `optimizers`, `create_inverse_triples` has a default value, which is `False`.

If there is only one value for either the `models`, `datasets`, `losses`, `training_loops`, `optimizers`, or `create_inverse_triples` argument, it can be given as a single value instead of the list.

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex04_terse_kwargs"
>>> ablation_pipeline(
...     directory=directory,
...     models="ComplEx",
...     datasets="Nations",
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops="LCWA",
...     optimizers="Adam",
...     create_inverse_triples=[True, False],
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     n_trials=1,
... )
```

Note

It doesn't make sense to run an ablation study if all of these values are fixed.

For each of the components of a knowledge graph embedding model (KGEM) that requires hyper-parameters, i.e., interaction model, loss function, and the training approach, we provide default hyper-parameter optimization (HPO) ranges within PyKEEN. Therefore, the definition of our ablation study would be complete at this stage. Because hyper-parameter ranges are

dataset-dependent, users can/should define their own HPO ranges. We will show later how to accomplish this. To finalize the ablation study, we recommend defining early stopping for your ablation study, which is done as follows:

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex05_stopper"
>>> ablation_pipeline(
...     directory=directory,
...     models=["ComplEx"],
...     datasets=["Nations"],
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops=["LCWA"],
...     optimizers=["Adam"],
...     stopper = "early",
...     stopper_kwargs = {
...         "frequency": 5,
...         "patience": 20,
...         "relative_delta": 0.002,
...         "metric": "hits@10",
...     },
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     n_trials=1,
... )
```

We define the early stopper using the argument `stopper`, and through `stopper_kwargs`, we provide instantiation arguments to the early stopper. We define that the early stopper should evaluate every 5 epochs with a patience of 20 epochs on the validation set. In order to continue training, we expect the model to obtain an improvement > 0.2% in Hits@10.

After defining the ablation study, we need to define the HPO settings for each experiment within our ablation study. Remember that for each ablation-experiment we perform an HPO in order to determine the best hyper-parameters for the currently investigated model. In PyKEEN, we use Optuna as HPO framework. Again, we provide default values for the Optuna related arguments. However, they define a very limited HPO search which is meant for testing purposes. Therefore, we define the arguments required by Optuna by ourselves:

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex06_optuna_kwargs"
>>> ablation_pipeline(
...     directory=directory,
...     models="ComplEx",
...     datasets="Nations",
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops="LCWA",
...     optimizers="Adam",
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     # Optuna-related arguments
...     n_trials=2,
...     timeout=300,
...     metric="hits@10",
...     direction="maximize",
...     sampler="random",
...     pruner= "nop",
... )
```

We set the number of HPO iterations for each experiment to 2 using the argument `n_trials`, set a `timeout` of 300 seconds (the HPO will be terminated after `n_trials` or `timeout` seconds depending on what occurs first), the `metric` to optimize, define whether the metric should be maximized or minimized using the argument `direction`, define random search as HPO algorithm using the argument `sampler`, and finally define that we do not use a pruner for pruning unpromising trials (note that we use early stopping instead).

To measure the variance in performance, we can additionally define how often we want to re-train and re-evaluate the best model of each ablation-experiment using the argument `best_replicates`:

```
>>> from pykeen.ablation import ablation_pipeline
>>> directory = "doctests/ablation/ex5"
>>> ablation_pipeline(
...     directory=directory,
...     models=["ComplEx"],
...     datasets=["Nations"],
...     losses=["BCEAfterSigmoidLoss", "MarginRankingLoss"],
...     training_loops=["LCWA"],
...     optimizers=["Adam"],
...     create_inverse_triples=[True, False],
...     stopper="early",
...     stopper_kwargs={
...         "frequency": 5,
...         "patience": 20,
...         "relative_delta": 0.002,
...         "metric": "hits@10",
...     },
...     # Fast testing configuration, make bigger in prod
...     epochs=1,
...     # Optuna-related arguments
...     n_trials=2,
...     timeout=300,
...     metric="hits@10",
...     direction="maximize",
...     sampler="random",
...     pruner= "nop",
...     best_replicates=5,
... )
```

Eager to check out the results? Then navigate to your output directory `path/to/output/directory`. Within your output directory, you will find subdirectories, e.g., `0000_nations_complex` which contains all experimental artifacts of one specific ablation experiment of the defined ablation study. The most relevant subdirectory is `best_pipeline` which comprises the artifacts of the best performing experiment, including its definition in `pipeline_config.json`, the obtained results, and the trained model(s) in the sub-directory `replicates`. The number of replicates in `replicates` corresponds to the number provided through the argument `-r`. Additionally, you are provided with further information about the ablation study in the root directory: `study.json` describes the ablation experiment, `hpo_config.json` describes the HPO setting of the ablation experiment, `trials.tsv` provides an overview of each HPO experiment.

Define Your Own HPO Ranges

As mentioned above, we provide default hyper-parameters/hyper-parameter ranges for each hyper-parameter. However, these default values/ranges do not ensure good performance. Therefore, it is time that you define your own ranges, and we show you how to do it! For the definition of hyper-parameter values/ranges, two dictionaries are essential, `kwargs` that is used to assign the hyper-parameters fixed values, and `kwargs_ranges` to define ranges of values from which to sample from.

Let's start with assigning HPO ranges to hyper-parameters belonging to the interaction model. This can be achieved by using the dictionary `model_to_model_kwargs_ranges`:

```
...

# Define HPO ranges
>>> model_to_model_kwargs_ranges = {
...     "ComplEx": {
...         "embedding_dim": {
...             "type": "int",
...             "low": 4,
...             "high": 6,
...             "scale": "power_two"
...         }
...     }
... }

...
```

We defined an HPO range for the embedding dimension. Because the `scale` is `power_two`, the lower bound (`low`) equals to 4, the upper bound `high` to 6, the embedding dimension is sampled from the set $\{2^4, 2^5, 2^6\}$.

Next, we fix the number of training epochs to 50 using the argument `model_to_training_loop_to_training_kwargs` and define a range for the batch size using `model_to_training_loop_to_training_kwargs_ranges`. We use these two dictionaries because the defined hyper-parameters are hyper-parameters of the training function (that is a function of the `training_loop`):

```
...
>>> model_to_model_kwargs_ranges = {
...     "ComplEx": {
...         "embedding_dim": {
...             "type": "int",
...             "low": 4,
...             "high": 6,
...             "scale": "power_two"
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs = {
...     "ComplEx": {
...         "lcwa": {
...             "num_epochs": 50
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs_ranges= {
...     "ComplEx": {
...         "lcwa": {
...             "label_smoothing": {
...                 "type": "float",
...                 "low": 0.001,
...                "high": 1.0,
...                 "scale": "log"
...             },
...             "batch_size": {
...                 "type": "int",
...                 "low": 7,
...                 "high": 9,
...                 "scale": "power_two"
...             }
...         }
...     }
... }

...
```

Finally, we define a range for the learning rate which is a hyper-parameter of the optimizer:

```
...

>>> model_to_model_kwargs_ranges = {
...     "ComplEx": {
...         "embedding_dim": {
...             "type": "int",
...             "low": 4,
...             "high": 6,
...             "scale": "power_two"
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs = {
...     "ComplEx": {
...         "lcwa": {
...             "num_epochs": 50
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs_ranges= {
...     "ComplEx": {
...         "lcwa": {
...             "label_smoothing": {
...                 "type": "float",
...                 "low": 0.001,
...                "high": 1.0,
...                 "scale": "log"
...             },
...             "batch_size": {
...                 "type": "int",
...                 "low": 7,
...                 "high": 9,
...                 "scale": "power_two"
...             }
...         }
...      }
... }

>>> model_to_optimizer_to_optimizer_kwargs_ranges= {
...     "ComplEx": {
...         "adam": {
...             "lr": {
...                 "type": "float",
...                 "low": 0.001,
...                 "high": 0.1,
...                 "scale": "log"
...             }
...         }
...     }
... }

...
```

We decided to use Adam as an optimizer, and defined a `log` `scale` for the learning rate, i.e., the learning rate is sampled from the interval $[0.001, 0.1)$.

Now that we defined our own hyper-parameter values/ranges, let's have a look at the overall configuration:

```
>>> from pykeen.ablation import ablation_pipeline
>>> metadata = dict(title="Ablation Study Over Nations for ComplEx.")
>>> models = ["ComplEx"]
>>> datasets = ["Nations"]
>>> losses = ["BCEAfterSigmoidLoss"]
>>> training_loops = ["lcwa"]
>>> optimizers = ["adam"]
>>> create_inverse_triples= [True, False]
>>> stopper = "early"
>>> stopper_kwargs = {
...     "frequency": 5,
...     "patience": 20,
...     "relative_delta": 0.002,
...     "metric": "hits@10",
... }

# Define HPO ranges
>>> model_to_model_kwargs_ranges = {
...     "ComplEx": {
...         "embedding_dim": {
...             "type": "int",
...             "low": 4,
...             "high": 6,
...             "scale": "power_two"
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs = {
...     "ComplEx": {
...         "lcwa": {
...             "num_epochs": 50
...         }
...     }
... }

>>> model_to_training_loop_to_training_kwargs_ranges= {
...     "ComplEx": {
...         "lcwa": {
...             "label_smoothing": {
...                 "type": "float",
...                 "low": 0.001,
...                "high": 1.0,
...                 "scale": "log"
...             },
...             "batch_size": {
...                 "type": "int",
...                 "low": 7,
...                 "high": 9,
...                 "scale": "power_two"
...             }
...         }
...     }
... }


>>> model_to_optimizer_to_optimizer_kwargs_ranges= {
...     "ComplEx": {
...         "adam": {
...             "lr": {
...                 "type": "float",
...                 "low": 0.001,
...                 "high": 0.1,
```

```
...                 "scale": "log"
...             }
...         }
...     }
... }

# Run ablation experiment
>>> ablation_pipeline(
...     models=models,
...     datasets=datasets,
...     losses=losses,
...     training_loops=training_loops,
...     optimizers=optimizers,
...     model_to_model_kwargs_ranges=model_to_model_kwargs_ranges,
...     model_to_training_loop_to_training_kwargs=model_to_training_loop_to_training_kwargs,
...
model_to_optimizer_to_optimizer_kwargs_ranges=model_to_optimizer_to_optimizer_kwargs_ranges,
...     directory="doctests/ablation/ex6",
...     best_replicates=5,
...     n_trials=2,
...     timeout=300,
...     metric="hits@10",
...     direction="maximize",
...     sampler="random",
...     pruner="nop",
... )
```

We are expected to provide the arguments `datasets`, `models`, `losses`, `optimizers`, and `training_loops` to **`pykeen.ablation.ablation_pipeline()`**. For all other components and hype-parameters, PyKEEN provides default values/ranges. However, for achieving optimal performance, we should carefully define the hyper-parameter values/ranges ourselves, as explained above. Note that there are many more ranges to configure such hyper-parameters for the loss functions or the negative samplers. Check out the examples provided in *tests/resources/hpo_complex_nations.json`* how to define the ranges for other components.

Run an Ablation Study With Your Own Data

We showed how to run an ablation study with a PyKEEN integrated dataset. Now you are asking yourself, whether you can run ablations studies with your own data? Yes, you can! It requires a minimal change compared to the previous configuration:

```
>>> datasets = [
...     {
...         "training": "/path/to/your/train.txt",
...         "validation": "/path/to/your/validation.txt",
...         "testing": "/path/to/your/test.txt"
...     }
... ]
```

In the dataset field, you don't provide a list of dataset names but dictionaries containing the paths to your train-validation-test splits.

Run an Ablation Study From The Command Line Interface

If you want to start an ablation study from the command line interface, we provide the function `pykeen.experiments.cli.ablation()`, which expects as an argument the path to a JSON configuration file. The configuration file consists of a dictionary with the sub-dictionaries `ablation` and `optuna` in which the ablation study and the Optuna related configuration are defined. Besides, similar to the programmatic interface, the `metadata` dictionary can be provided. The configuration file corresponding to the ablation study that we previously defined within our program would look as follows:

```
{
    "metadata": {
        "title": "Ablation Study Over Nations for ComplEx."
    },
    "ablation": {
        "datasets": ["nations"],
        "models":   ["ComplEx"],
        "losses": ["BCEAfterSigmoidLoss", "CrossEntropyLoss"]
        "training_loops": ["lcwa"],
        "optimizers": ["adam"],
        "create_inverse_triples": [true,false],
        "stopper": "early"
        "stopper_kwargs": {
            "frequency": 5,
            "patience": 20,
            "relative_delta": 0.002,
            "metric": "hits@10"
        },
        "model_to_model_kwargs_ranges":{
            "ComplEx": {
                "embedding_dim": {
                    "type": "int",
                    "low": 4,
                    "high": 6,
                    "scale": "power_two"
                }
            }
        },
        "model_to_training_loop_to_training_kwargs": {
            "ComplEx": {
                "lcwa": {
                    "num_epochs": 50
                }
            }
        },
        "model_to_training_loop_to_training_kwargs_ranges": {
            "ComplEx": {
                "lcwa": {
                    "label_smoothing": {
                        "type": "float",
                        "low": 0.001,
                        "high": 1.0,
                        "scale": "log"
                    },
                    "batch_size": {
                        "type": "int",
                        "low": 7,
                        "high": 9,
                        "scale": "power_two"
                    }
                }
            }
        },
        "model_to_optimizer_to_optimizer_kwargs_ranges": {
            "ComplEx": {
                "adam": {
                    "lr": {
                        "type": "float",
                        "low": 0.001,
                        "high": 0.1,
                        "scale": "log"
                    }
                }
```

```
            }
        }
    "optuna": {
        "n_trials": 2,
        "timeout": 300,
        "metric": "hits@10",
        "direction": "maximize",
        "sampler": "random",
        "pruner": "nop"
        }
    }
}
```

The ablation study can be started as follows:

```
$ pykeen experiments ablation path/to/complex_nation.json -d path/to/output/directory
```

To re-train and re-evaluate the best model of each ablation-experiment `n` times in order to measure the variance in performance the option `-r` / `--best-replicates` should be used:

```
$ pykeen experiments ablation path/to/complex_nation.json -d path/to/output/directory -r 5
```

In this tutorial, we showed how to define and start an ablation study within your program, how to execute it from the command line interface. Furthermore, we showed how you can define your ablation study using your own data.