pykg2vec Documentation

Release 0.0.52

Sujit Rokka Chhetri, Shih-Yuan Yuand, Ahmet Salih Aksakal, Palas

Apr 21, 2021

QUICK START TUTORIAL

Pykg2vec is a Pytorch-based library, currently in active development, for learning the representation of entities and relations in Knowledge Graphs. We have attempted to bring all the state-of-the-art knowledge graph embedding algorithms and the necessary building blocks including the whole pipeline into a single library.

- Previously, we built pykg2vec using TensorFlow. We switched to Pytorch as we found that more authors use Pytorch to implement their KGE models. Nevertheless, the TF version is still available in branch tf2-master.

CHAPTER ONE

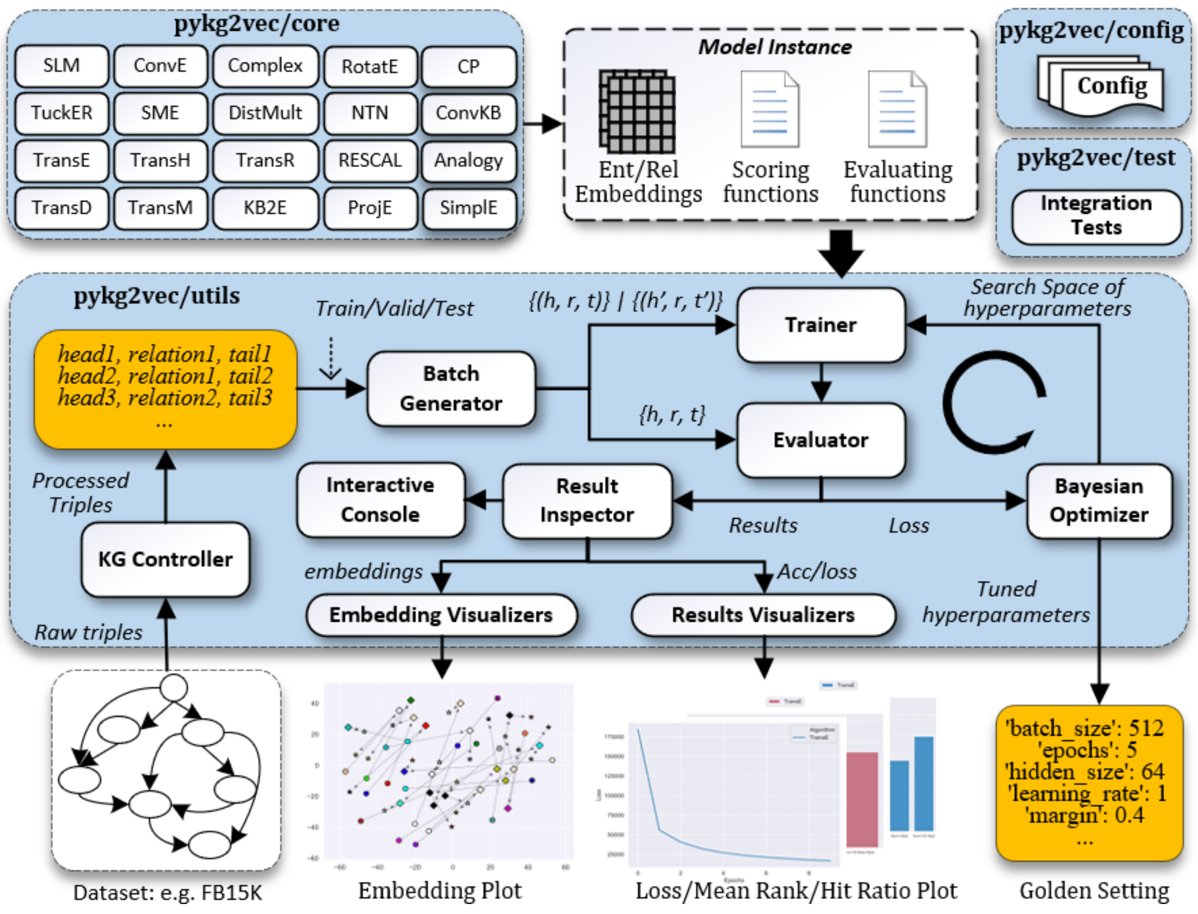
INTRODUCTION

Pykg2vec is built with PyTorch for learning the representation of entities and relations in Knowledge Graphs. In recent years, Knowledge Graph Embedding (KGE) methods have been applied in applications such as Fact Prediction, Question Answering, and Recommender Systems.

KGE is an active research field and many authors have provided reference software implementations. However, most of these are standalone reference implementations and therefore it is difficult and time-consuming to work with KGE methods. Therefore, we built this library, pykg2vec, hoping to contribute this community with:

1. A sheer amount of existing state-of-the-art knowledge graph embedding algorithms (TransE, TransH, TransR, TransD, TransM, KG2E, RESCAL, DistMult, ComplEX, ConvE, ProjE, RotatE, SME, SLM, NTN, TuckER, etc) is presented.
2. The module that supports automatic hyperparameter tuning using bayesian optimization.
3. A suite of visualization and summary tools to facilitate result inspection.

We hope Pykg2vec has both practical and educational values for users who hope to explore the related fields.


pykg2vec/core
SLM
ConvE
Complex
RotatE
CP
TuckER
SME
DistMult
NTN
ConvKB
TransE
TransH
TransR
RESCAL
Analogy
TransD
TransM
KB2E
ProjE
SimplE
Model Instance
Ent/Rel Embeddings
Scoring functions
Evaluating functions
pykg2vec/config
Config
pykg2vec/test
Integration Tests
pykg2vec/utils
Train/Valid/Test
{(h, r, t)} | {(h', r, t')}
Search Space of hyperparameters
head1, relation1, tail1
head2, relation1, tail2
head3, relation2, tail3
...
Batch Generator
Trainer
Evaluator
{h, r, t}
Processed Triples
Interactive Console
Result Inspector
Results
Loss
Bayesian Optimizer
KG Controller
embeddings
Acc/loss
Tuned hyperparameters
Embedding Visualizers
Results Visualizers
Raw triples
'batch_size': 512
'epochs': 5
'hidden_size': 64
'learning_rate': 1
'margin': 0.4
...
Dataset: e.g. FB15K
Embedding Plot
Loss/Mean Rank/Hit Ratio Plot
Golden Setting

CHAPTER

TWO

START WITH PYKG2VEC

In order to install pykg2vec, you will need setup the following libraries:

- python >=3.7 (recommended)
- pytorch>= 1.5

All dependent packages (requirements.txt) will be installed automatically when setting up pykg2vec.

- networkx>=2.2
- setuptools>=40.8.0
- matplotlib>=3.0.3
- numpy>=1.16.2
- seaborn>=0.9.0
- scikit_learn>=0.20.3
- hyperopt>=0.1.2
- progressbar2>=3.39.3
- pathlib>=1.0.1
- pandas>=0.24.2

Installation Guide

1. **Setup a Virtual Environment**: we encourage you to use anaconda to work with pykg2vec:

```
(base) $ conda create --name pykg2vec python=3.7
(base) $ conda activate pykg2vec
```

2. **Setup Pytorch**: we encourage to use pytorch with GPU support for good training performance. However, a CPU version also runs. The following sample commands are for setting up pytorch:

```
# if you have a GPU with CUDA 10.1 installed
(pykg2vec) $ conda install pytorch torchvision cudatoolkit=10.1 -c pytorch
# or cpu-only
(pykg2vec) $ conda install pytorch torchvision cpuonly -c pytorch
```

3. **Setup Pykg2vec**:

```
(pykg2vec) $ git clone https://github.com/Sujit-O/pykg2vec.git
(pykg2vec) $ cd pykg2vec
(pykg2vec) $ python setup.py install
```

4. **Validate the Installation**: try the examples under /examples folder.

```
# train TransE using benchmark dataset fb15k (use pykg2vec-train.exe on Windows)
(pykg2vec) $ pykg2vec-train -mn transe -ds fb15k
```

CHAPTER

THREE

PROGRAMMING EXAMPLES

We developed several programming examples for users to start working with pykg2vec. The examples are in ./examples folder.

```
(pykg2vec) $ cd ./examples
# train TransE using benchmark dataset fb15k
(pykg2vec) $ python train.py -mn transe -ds fb15k
# train and tune TransE using benchmark dataset fb15k
(pykg2vec) $ python tune_model.py -mn TransE -ds fb15k
```

Please go through the examples for more advanced usages:

- Work with one KGE method (train.py)
- Automatic Hyperparameter Discovery (tune_model.py)
- Inference task for one KGE method (inference.py)
- Train multiple algorithms: (experiment.py)
- Full pykg2vec pipeline: (kgpipeline.py)

Use Your Own Hyperparameters

- To experiment with your own hyperparameters, tweak the values inside ./examples/custom_hp.yaml or create your own files.

```
$ python train.py -exp True -mn TransE -ds fb15k -hpf ./examples/custom_hp.yaml
```

- YAML formatting example (The file is also provided in /examples folder named custom_hp.yaml):

```
model_name: "TransE"
datasets:
  - dataset: "freebase15k"
    parameters:
      learning_rate: 0.01
      l1_flag: True
      hidden_size: 50
      batch_size: 128
      epochs: 1000
      margin: 1.00
      optimizer: "sgd"
      sampling: "bern"
      neg_rate: 1
```

- NB: To make sure the loaded hyperparameters will be actually used for training, you need to pass in the same model_name value via -mn and the same dataset value via -ds as already specified in the YAML file from where those hyperparameters are originated.

Use Your Own Search Spaces

- To tune a model with your own search space, tweak the values inside ./examples/custom_ss.yaml or create your own files.

```
$ python tune_model.py -exp True -mn TransE -ds fb15k -ssf ./examples/custom_ss.
↪yaml
```

- YAML formatting example (THe file is also provided in /examples folder named custom_ss.yaml):

```
model_name: "TransE"
dataset: "freebase15k"
search_space:
    learning_rate:
        min: 0.00001
        max: 0.1
    l1_flag:
        - True
        - False
    hidden_size:
        min: 8
        max: 256
    batch_size:
        min: 8
        max: 4096
    margin:
        min: 0.0
        max: 10.0
    optimizer:
        - "adam"
        - "sgd"
        - "rms"
    epochs:
        - 10
```

- NB: To make sure the loaded search space will be actually used for tune_model.py, you need to pass in the same model_name value via -mn and the same dataset value via -ds as already specified in the YAML file that aligned with the parameters included in yaml.

Use Your Own Dataset

To create and use your own dataset, these steps are required:

1. Store all of triples in a text-format with each line as below, using tab space (“t”) to seperate entities and relations.:

```
head\trelation\ttail
```

2. For the text file, separate it into three files according to your reference give names as follows,

```
[name]-train.txt, [name]-valid.txt, [name]-test.txt
```

3. For those three files, create a folder [path_storing_text_files] to include them.

4. Create a new custom hyperparameter YAML file (detailed in "Use Your Own Hyperparameters"). For example,

```
model_name: "TransE"
datasets:
  - dataset: "[name]"
    parameters:
      learning_rate: 0.01
      l1_flag: True
      hidden_size: 50
      batch_size: 128
      epochs: 1000
      margin: 1.00
      optimizer: "sgd"
      sampling: "bern"
      neg_rate: 1
```

5. Once finished, you then can use your own dataset to train a KGE model or tune its hyperparameters using commands::

```
$ python train.py -mn TransE -ds [name] -dsp [path_storing_text_files] -hpf [path_
↪to_hyperparameter_yaml]
$ python tune_model.py -mn TransE -ds [name] -dsp [path_storing_text_files] -hpf␣
↪[path_to_hyperparameter_yaml]
```

3.1 Automatic Hyperparameter Discovery (tune_model.py)

With tune_model.py we can train and tune the existed model using command:

- check all tunnable parameters.

```
$ python tune_model.py -h
```

- We are still improving the interfaces to make them more convenient to use. For now, please refer to hyper-params.py to manually adjust the search space of hyperparameters.

```
# in hyperparams.py#xxxParams
self.search_space = {
    'learning_rate': hp.loguniform('learning_rate', np.log(0.00001), np.log(0.1)),
    'l1_flag': hp.choice('l1_flag', [True, False]),
    'hidden_size': scope.int(hp.qloguniform('hidden_size', np.log(8), np.log(256),
↪1)),
    'batch_size': scope.int(hp.qloguniform('batch_size', np.log(8), np.log(4096),
↪1)),
    'margin': hp.uniform('margin', 0.0, 2.0),
    'optimizer': hp.choice('optimizer', ["adam", "sgd", 'rms']),
    'epochs': hp.choice('epochs', [10])
}
```

- Tune TransE using the benchmark dataset fb15k.

```
$ python tune_model.py -mn TransE -ds fb15k
```

- Tune an algorithm using your own search space (algorithm_name and the algorithm name specified in YAML file should align):

```
$ python tune_model.py -mn [algorithm_name] -ds fb15k -ssf [path_to_file].yaml
```

- Please refer here for details of YAML format.

We also attached the source code of tune_model.py below for your reference.

```
# Author: Sujit Rokka Chhetri
# License: MIT

import sys


from pykg2vec.common import KGEArgParser
from pykg2vec.utils.bayesian_optimizer import BaysOptimizer


def main():
    # getting the customized configurations from the command-line arguments.
    args = KGEArgParser().get_args(sys.argv[1:])

    # initializing bayesian optimizer and prepare data.
    bays_opt = BaysOptimizer(args=args)

    # perform the golden hyperparameter tuning.
    bays_opt.optimize()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.2 Inference task for one KGE method (inference.py)

With inference.py, you can perform inference tasks with learned KGE model. Some available commands are:

```
$ python inference.py -mn TransE # train a model on FK15K dataset and enter␣
↪interactive CMD for manual inference tasks.
$ python inference.py -mn TransE -ld examples/pretrained/TransE # pykg2vec will load␣
↪the pretrained model from the specified directory.

# Once interactive mode is reached, you can execute instruction manually like
# Example 1: trainer.infer_tails(1,10,topk=5) => give the list of top-5 predicted␣
↪tails.
# Example 2: trainer.infer_heads(10,20,topk=5) => give the list of top-5 predicted␣
↪heads.
# Example 3: trainer.infer_rels(1,20,topk=5) => give the list of top-5 predicted␣
↪relations.
```

We also attached the source code of inference.py below for your reference.

```python
# Author: Sujit Rokka Chhetri and Shiy Yuan Yu
# License: MIT

import sys

from pykg2vec.common import Importer, KGEArgParser
from pykg2vec.utils.trainer import Trainer


def main():
    # getting the customized configurations from the command-line arguments.
    args = KGEArgParser().get_args(sys.argv[1:])

    # Extracting the corresponding model config and definition from Importer().
    config_def, model_def = Importer().import_model_config(args.model_name.lower())
    config = config_def(args)
    model = model_def(**config.__dict__)

    # Create the model and load the trained weights.
    trainer = Trainer(model, config)
    trainer.build_model()

    if config.load_from_data is None:
        trainer.train_model()

    trainer.infer_tails(1, 10, topk=5)
    trainer.infer_heads(10, 20, topk=5)
    trainer.infer_rels(1, 20, topk=5)


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3 Work with one KGE method (train.py)

You can train a single KGE algorithm with train.py by using the following commands:

- check all tunnable parameters:

```
$ python train.py -h
```

- Train TransE on FB15k benchmark dataset:

```
$ python train.py -mn TransE
```

- Train using different KGE methods. Check Implemented KGE Algorithms for more details:

```
$ python train.py -mn␣
↪[TransE|TransD|TransH|TransG|TransM|TransR|Complex|ComplexN3|
                    CP|RotatE|Analogy|DistMult|KG2E|KG2E_
↪EL|NTN|Rescal|SLM|SME|SME_BL|HoLE|
                    ConvE|ConvKB|Proje_
↪pointwise|MuRP|QuatE|OctonionE|InteractE|HypER]
```

- For KGE using projection-based loss function, use more processes for batch generation:

```
$ python train.py -mn [ConvE|ConvKB|Proje_pointwise] -npg [the number of␣
↪processes, 4 or 6]
```

- Train TransE model using different benchmark datasets:

```
$ python train.py -mn TransE -ds [fb15k|wn18|wn18_rr|yago3_10|fb15k_237|
                                ks|nations|umls|dl50a|nell_955]
```

- Train KGE method with the hyperparameters used in original papers: (FB15k supported only):

```
$ python train.py -mn␣
↪[TransE|TransD|TransH|TransG|TransM|TransR|Complex|ComplexN3|CP|RotatE|Analogy|
                    distmult|KG2E|KG2E_EL|NTN|Rescal|SLM|SME|SME_
↪BL|HoLE|ConvE|ConvKB|Proje_pointwise] -exp true -ds fb15k
```

- Train KGE method with your own set of hyperparameters stored (algorithm_name and the algorithm name specified in YAML file should align):

```
$ python train.py -mn [algorithm_name] -exp true -ds fb15k -hpf [path_to_file].
↪yaml
```

- Please refer here for details of YAML format.

We also attached the source code of train.py below for your reference.

```
# Author: Sujit Rokka Chhetri
# License: MIT

import sys

from pykg2vec.data.kgcontroller import KnowledgeGraph
from pykg2vec.common import Importer, KGEArgParser
from pykg2vec.utils.trainer import Trainer


def main():
    # getting the customized configurations from the command-line arguments.
    args = KGEArgParser().get_args(sys.argv[1:])

    # Preparing data and cache the data for later usage
    knowledge_graph = KnowledgeGraph(dataset=args.dataset_name, custom_dataset_
↪path=args.dataset_path)
    knowledge_graph.prepare_data()

    # Extracting the corresponding model config and definition from Importer().
    config_def, model_def = Importer().import_model_config(args.model_name.lower())
    config = config_def(args)
    model = model_def(**config.__dict__)

    # Create, Compile and Train the model. While training, several evaluation will be␣
↪performed.
    trainer = Trainer(model, config)
    trainer.build_model()
    trainer.train_model()
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.4 Train multiple Algorithms (experiment.py)

You can also design your own expriment plans. For example, we attached experiment.py (adjust it for your own usage) below for your reference, which trains multiple algorithms at once.

```
$ python experiment.py
```

```
# Author: Sujit Rokka Chhetri and Shih Yuan Yu
# License: MIT


from pykg2vec.data.kgcontroller import KnowledgeGraph
from pykg2vec.common import Importer, KGEArgParser
from pykg2vec.utils.trainer import Trainer


def experiment(model_name):
    args = KGEArgParser().get_args([])

    args.exp = True
    args.dataset_name = "freebase15k"

    # Preparing data and cache the data for later usage
    knowledge_graph = KnowledgeGraph(dataset=args.dataset_name, custom_dataset_
↪path=args.dataset_path)
    knowledge_graph.prepare_data()

    # Extracting the corresponding model config and definition from Importer().
    config_def, model_def = Importer().import_model_config(model_name)
    config = config_def(args)
    model = model_def(**config.__dict__)

    # Create, Compile and Train the model. While training, several evaluation will be␣
↪performed.
    trainer = Trainer(model, config)
    trainer.build_model()
    trainer.train_model()


if __name__ == "__main__":

    # examples of train an algorithm on a benchmark dataset.
    experiment("transe")
    experiment("transh")
    experiment("transr")
```

(continues on next page)

(continued from previous page)

```
    # other combination we are still working on them.
    # experiment("transe", "wn18_rr")
```

Total running time of the script: (0 minutes 0.000 seconds)

3.5 Full Pykg2vec pipeline (kgpipeline.py)

kgpipeline.py demonstrates the full pipeline of training KGE methods with pykg2vec. This pipeline first discover the best set of hyperparameters using training and validation set. Then it uses the discovered hyperparameters to evaluate the KGE algorithm on the testing set.

```
python kgpipeline.py
```

We also attached the source code of kgpipeline.py below for your reference. You can adjust to fit your usage.

```
# Author: Sujit Rokka Chhetri and Shih Yuan Yu
# License: MIT

from pykg2vec.common import Importer, KGEArgParser
from pykg2vec.data.kgcontroller import KnowledgeGraph
from pykg2vec.utils.bayesian_optimizer import BaysOptimizer
from pykg2vec.utils.trainer import Trainer


def main():
    model_name = "transe"
    dataset_name = "Freebase15k"
    # dataset_path = "path_to_dataset"

    # 1. Tune the hyper-parameters for the selected model and dataset.
    # p.s. this is using training and validation set.
    args = KGEArgParser().get_args(['-mn', model_name, '-ds', dataset_name])

    # initializing bayesian optimizer and prepare data.
    bays_opt = BaysOptimizer(args=args)

    # perform the golden hyperparameter tuning.
    bays_opt.optimize()
    best = bays_opt.return_best()

    # 2. Evaluate final model using the found best hyperparameters on testing set.
    args = KGEArgParser().get_args(['-mn', model_name, '-ds', dataset_name])

    # Preparing data and cache the data for later usage
    knowledge_graph = KnowledgeGraph(dataset=args.dataset_name)
    knowledge_graph.prepare_data()

    # Extracting the corresponding model config and definition from Importer().
    config_def, model_def = Importer().import_model_config(args.model_name.lower())
    config = config_def(args)

    # Update the config params with the golden hyperparameter
```

(continues on next page)

(continued from previous page)

```
    for k, v in best.items():
        config.__dict__[k] = v
    model = model_def(**config.__dict__)

    # Create, Compile and Train the model.
    trainer = Trainer(model, config)
    trainer.build_model()
    trainer.train_model()


if __name__ == "__main__":
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER

FOUR

KNOWLEDGE GRAPH EMBEDDING

4.1 Introduction for KGE

A knowledge graph contains a set of entities $\mathbb{E}$ and relations $\mathbb{R}$ between entities. The set of facts $\mathbb{D}^+$ in the knowledge graph are represented in the form of triples (h, r, t), where $h, t \in \mathbb{E}$ are referred to as the **head** (or *subject*) and the **tail** (or *object*) entities, and $r \in \mathbb{R}$ is referred to as the **relationship** (or *predicate*).

The problem of KGE is in finding a function that learns the embeddings of triples using low dimensional vectors such that it preserves structural information, $f : \mathbb{D}^+ \rightarrow \mathbb{R}^d$. To accomplish this, the general principle is to enforce the learning of entities and relationships to be compatible with the information in $\mathbb{D}^+$. The representation choices include deterministic point, multivariate Gaussian distribution, or complex number. Under the Open World Assumption (OWA), a set of unseen negative triplets, $\mathbb{D}^-$, are sampled from positive triples $\mathbb{D}^+$ by either corrupting the head or tail entity. Then, a scoring function, $f_r(h, t)$ is defined to reward the positive triples and penalize the negative triples. Finally, an optimization algorithm is used to minimize or maximize the scoring function.

KGE methods are often evaluated in terms of their capability of predicting the missing entities in negative triples $(?, r, t)$ or $(h, r, ?)$, or predicting whether an unseen fact is true or not. The evaluation metrics include the rank of the answer in the predicted list (mean rank), and the ratio of answers ranked top-k in the list (hit-k ratio).

4.2 Implemented KGE Algorithms

We aim to implement as many latest state-of-the-art knowledge graph embedding methods as possible. From our perspective, by so far the KGE methods can be categorized based on the ways that how the model is trained:

1. **Pairwise (margin) based Training KGE Models**: these models utilize a latent feature of either entities or relations to explain the triples of the Knowledge graph. The features are called latent as they are not directly observed. The interaction of the entities and the relations are captured through their latent space representation. These models either utilize a distance-based scoring function or similarity-based matching function to embed the knowledge graph triples. (please refer to pykg2vec.models.pairwise for more details)
2. **Pointwise based Training KGE Models**: (please refer to pykg2vec.models.pointwise for more details).
3. **Projection-Based (Multiclass) Training KGE Models**: (please refer to pykg2vec.models.projection for more details).

4.3 Supported Dataset

We support various known benchmark datasets in pykg2vec.

- FreebaseFB15k: Freebase dataset.
- WordNet18: WordNet18 dataset.
- WordNet18RR: WordNet18RR dataset.
- YAGO3_10: YAGO Dataset.
- DeepLearning50a: DeepLearning dataset.

We also support the use of your own dataset. Users can define their own datasets to be processed with the pykg2vec library.

4.4 Benchmarks

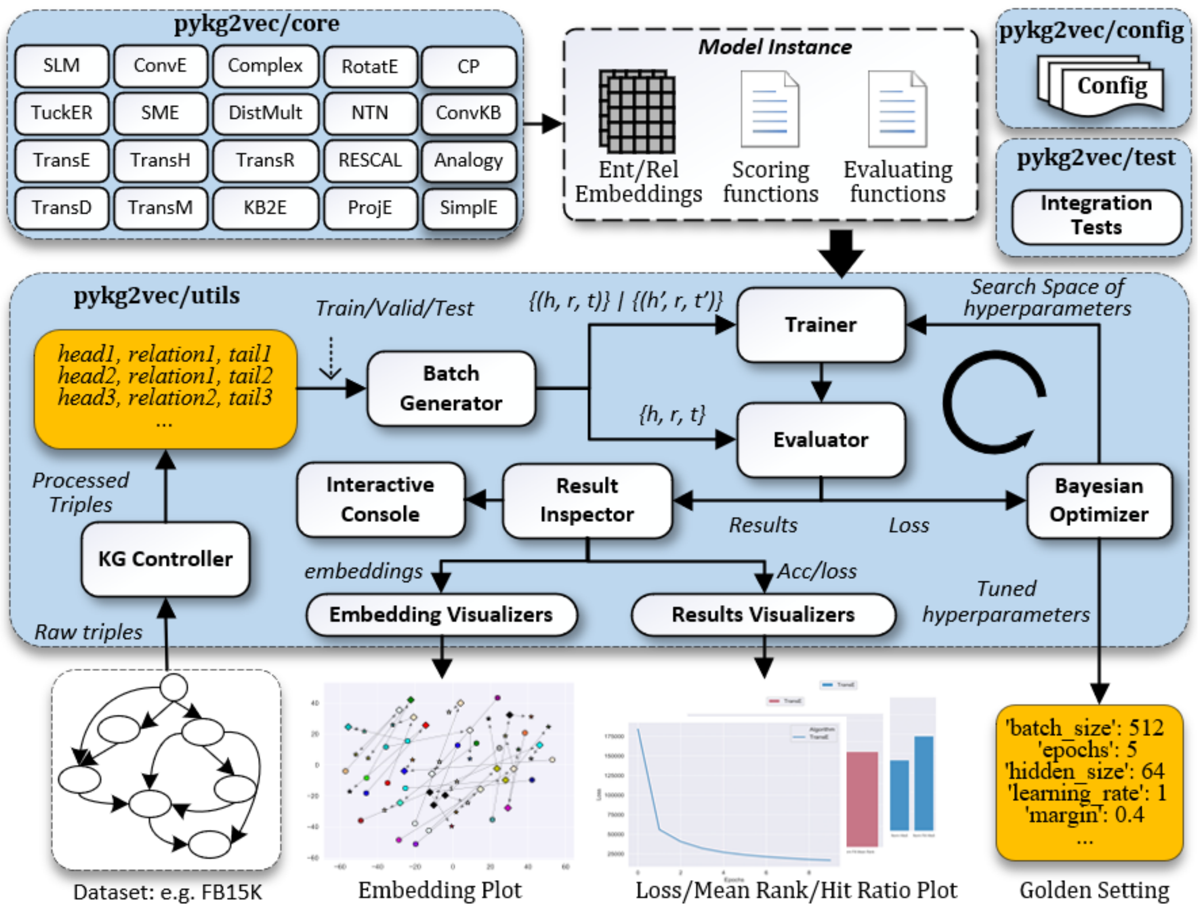
Some metrics running on benchmark dataset (FB15k) is shown below (all are filtered). We are still working on this table so it will be updated.

	MR	MRR	Hit1	Hit3	Hit5	Hit10
TransE	69.52	0.38	0.23	0.46	0.56	0.66
TransH	77.60	0.32	0.16	0.41	0.51	0.62
TransR	128.31	0.30	0.18	0.36	0.43	0.54
TransD	57.73	0.33	0.19	0.39	0.48	0.60
KG2E_EL	64.76	0.31	0.16	0.39	0.49	0.61
Complex	96.74	0.65	0.54	0.74	0.78	0.82
DistMult	128.78	0.45	0.32	0.53	0.61	0.70
RotatE	48.69	0.74	0.67	0.80	0.82	0.86
SME_L	86.3	0.32	0.20	0.35	0.43	0.54
SLM_BL	112.65	0.29	0.18	0.32	0.39	0.50

CHAPTER

FIVE

SOFTWARE ARCHITECTURE AND API DOCUMENTATION



The pykg2vec is built using Python and PyTorch. It allows the computations to be assigned on both GPU and CPU. In addition to the main model training process, pykg2vec utilizes multi-processing for generating mini-batches and performing an evaluation to reduce the total execution time. The various components of the library are as follows:

1) `KG Controller` - handles all the low-level parsing tasks such as finding the total unique set of entities and relations; creating ordinal encoding maps; generating training, testing and validation triples; and caching the dataset data on disk to optimize tasks that involve repetitive model testing.

2) `Batch Generator` - consists of multiple concurrent processes that manipulate and create mini-batches of data. These mini-batches are pushed to a queue to be processed by the models implemented in PyTorch or TensorFlow. The batch generator runs independently so that there is a low latency for feeding the data to the training module running on the GPU.

3) `Core Models` - consists of large number of state-of-the-art KGE algorithms implemented as Python modules in PyTorch and TensorFlow. Each module consists of a modular description of the inputs, outputs, loss function,and embedding operations. Each model is provided with configuration files that define its hyperparameters.
4) `Configuration` - provides the necessary configuration to parse the datasets and also consists of the baseline hyperparameters for the KGE algorithms as presented in the original research papers.
5) `Trainer and Evaluator` - the Trainer module is responsible for taking an instance of the KGE model, the respective hyperparameter configuration, and input from the batch generator to train the algorithms. The Evaluator module performs link prediction and provides the respective accuracy in terms of mean ranks and filtered mean ranks.
6) `Visualization` - plots training loss and common metrics used in KGE tasks. To facilitate model analysis, it also visualizes the latent representations of entities and relations on the 2D plane using t-SNE based dimensionality reduction.
7) `Bayesian Optimizer` - pykg2vec uses a Bayesian hyperparameter optimizer to find a golden hyperparameter set. This feature is more efficient than brute-force based approaches.

Contents

5.1 pykg2vec

5.1.1 config.py

This module consists of definition of the necessary configuration parameters for all the core algorithms. The parameters are seprated into global parameters which are common across all the algorithms, and local parameters which are specific to the algorithms.

class `pykg2vec.config.`**`Config`**(*args*)

The class defines the basic configuration for the pykg2vec.

Config consists of the necessary parameter description used by all the modules including the algorithms and utility functions.

Parameters

- **test_step** (*int*) – Testing is carried out every test_step.
- **test_num** (*int*) – Number of triples that will be tested during evaluation.
- **triple_num** (*int*) – Number of triples that will be used for plotting the embedding.
- **tmp** (*Path Object*) – Path where temporary model information is stored.
- **result** (*Path Object*) – Gives the path where the result will be saved.
- **figures** (*Path Object*) – Gives the path where the figures will be saved.
- **load_from_data** (*string*) – If set, loads the model parameters if available from disk.
- **save_model** (*True*) – If True, store the trained model parameters.
- **disp_summary** (*bool*) – If True, display the summary before and after training the algorithm.
- **disp_result** (*bool*) – If True, displays result while training.
- **plot_embedding** (*bool*) – If True, will plot the embedding after performing t-SNE based dimensionality reduction.
- **log_training_placement** (*bool*) – If True, allows us to find out which devices the operations and tensors are assigned to.
- **plot_training_result** (*bool*) – If True, plots the loss values stored during training.

- **plot_testing_result** (*bool*) – If True, it will plot all the testing result such as mean rank, hit ratio, etc.
- **plot_entity_only** (*bool*) – If True, plots the t-SNE reduced embdding of the entities in a figure.
- **hits** (*List*) – Gives the list of integer for calculating hits.
- **knowledge_graph** (*Object*) – It prepares and holds the instance of the knowledge graph dataset.
- **kg_meta** (*object*) – Stores the statistics metadata of the knowledge graph.

summary()
Function to print the summary.

5.2 pykg2vec.data

5.2.1 pykg2vec.data.kgcontroller

This module is for controlling knowledge graph

class pykg2vec.data.kgcontroller.**KGMetaData**(*tot_entity=None, tot_relation=None, tot_triple=None, tot_train_triples=None, tot_test_triples=None, tot_valid_triples=None*)

The class store the metadata of the knowledge graph.

Instance of KGMetaData is used later to build the algorithms based of number of entities and relation.

Parameters

- **tot_entity** (*int*) – Total number of combined head and tail entities present in knowledge graph.
- **tot_relation** (*int*) – Total number of relations present in knowlege graph.
- **tot_triple** (*int*) – Total number of head, relation and tail (triples) present in knowledge graph.
- **tot_train_triples** (*int*) – Total number of training triples
- **tot_test_triples** (*int*) – Total number of testing triple
- **tot_valid_triples** (*int*) – Total number of validation triples

Examples

```
>>> from pykg2vec.data.kgcontroller import KGMetaData
>>> kg_meta = KGMetaData(tot_triple =1000)
```

class pykg2vec.data.kgcontroller.**KnowledgeGraph**(*dataset='Freebase15k', custom_dataset_path=None*)

The class is the main module that handles the knowledge graph.

KnowledgeGraph is responsible for downloading, parsing, processing and preparing the training, testing and validation dataset.

Parameters

- **dataset_name** (*str*) – Name of the datasets
- **custom_dataset_path** (*str*) – The path to custom dataset.

dataset_name
The name of the dataset.

Type str

dataset
The dataset object isntance.

Type object

triplets
dictionary with three list of training, testing and validation triples.

Type dict

relations
list of all the relations.

Type list

entities
List of all the entities.

Type list

entity2idx
Dictionary for mapping string name of entities to unique numerical id.

Type dict

idx2entity
Dictionary for mapping the id to string.

Type dict

relation2idx
Dictionary for mapping the id to string.

Type dict

idx2relation
Dictionary for mapping the id to string.

Type dict

hr_t
Dictionary with set as a default key and list as values.

Type dict

tr_h
Dictionary with set as a default key and list as values.

Type dict

hr_t_train
Dictionary with set as a default key and list as values.

Type dict

tr_h_train
Dictionary with set as a default key and list as values.

Type dict

relation_property
list storing the entities tied to a specific relation.

Type list

kg_meta
Object storing the statistics metadata of the dataset.

Type object

Examples

```
>>> from pykg2vec.data.kgcontroller import KnowledgeGraph
>>> knowledge_graph = KnowledgeGraph(dataset='Freebase15k')
>>> knowledge_graph.prepare_data()
```

dump()
Function to dump statistic information of a dataset

is_cache_exists()
Function to check if the dataset is cached in the memory

prepare_data()
Function to prepare the dataset

read_cache_data(*key*)
Function to read the cached dataset from the memory

read_entities()
Function to read the entities.

read_hr_t()
Function to read the list of tails for the given head and relation pair.

read_hr_t_train()
Function to read the list of tails for the given head and relation pair for the training set.

read_hr_t_valid()
Function to read the list of tails for the given head and relation pair for the valid set.

read_mappings()
Function to generate the mapping from string name to integer ids.

read_relation_property()
Function to read the relation property.

Returns Returns the list of relation property.

Return type list

read_relations()
Function to read the relations.

read_tr_h()
Function to read the list of heads for the given tail and relation pair.

read_tr_h_train()
Function to read the list of heads for the given tail and relation pair for the training set.

read_tr_h_valid()
Function to read the list of heads for the given tail and relation pair for the valid set.

read_triple_ids(*set_type*)
Function to read the triple idx.

Parameters **set_type** (*str*) – Type of data, eithe train, test or valid.

read_triplets(*set_type*)
read triplets from txt files in dataset folder. (in string format)

class pykg2vec.data.kgcontroller.**Triple**(*h*, *r*, *t*)
The class defines the datastructure of the knowledge graph triples.

Triple class is used to store the head, tail and relation triple in both its numerical id and string form. It also stores the dictonary of (head, relation)=[tail1, tail2,..] and (tail, relation)=[head1, head2, . . .]

Parameters

- **h** (*str or int*) – String or integer head entity.
- **r** (*str or int*) – String or integer relation entity.
- **t** (*str or int*) – String or integer tail entity.

Examples

```
>>> from pykg2vec.data.kgcontroller import Triple
>>> trip1 = Triple(2,3,5)
>>> trip2 = Triple('Tokyo','isCapitalof','Japan')
```

set_ids(*h*, *r*, *t*)
This function assigns the head, relation and tail.

Parameters

- **h** (*int*) – Integer head entity.
- **r** (*int*) – Integer relation entity.
- **t** (*int*) – Integer tail entity.

5.2.2 pykg2vec.data.generator

This module is for generating the batch data for training and testing.

class pykg2vec.data.generator.**Generator**(*model*, *config*)
Generator class for the embedding algorithms

Parameters

- **config** (*object*) – generator configuration object.
- **model_config** (*object*) – Model configuration object.

Yields *matrix* – Batch size of processed triples

Examples

```
>>> from pykg2vec.utils.generator import Generator
>>> from pykg2vec.models.TransE impor TransE
>>> model = TransE()
>>> gen_train = Generator(model.config, training_strategy=TrainingStrategy.
↪PAIRWISE_BASED)
```

create_feeder_process()
Function create the feeder process.

create_train_processor_process()
Function ro create the process for generating training samples.

stop()
Function to stop all the worker process.

pykg2vec.data.generator.**process_function_multiclass**(*raw_queue*, *processed_queue*, *config*)
Function that puts the processed data in the queue.

Parameters

- **raw_queue** (*Queue*) – Multiprocessing Queue to put the raw data to be processed.
- **processed_queue** (*Queue*) – Multiprocessing Queue to put the processed data.
- **config** (*pykg2vec.Config*) – Consists of the necessary parameters for training configuration.

pykg2vec.data.generator.**process_function_pairwise**(*raw_queue*, *processed_queue*, *config*)
Function that puts the processed data in the queue.

Parameters

- **raw_queue** (*Queue*) – Multiprocessing Queue to put the raw data to be processed.
- **processed_queue** (*Queue*) – Multiprocessing Queue to put the processed data.
- **config** (*pykg2vec.Config*) – Consists of the necessary parameters for training configuration.

pykg2vec.data.generator.**process_function_pointwise**(*raw_queue*, *processed_queue*, *config*)
Function that puts the processed data in the queue.

Parameters

- **raw_queue** (*Queue*) – Multiprocessing Queue to put the raw data to be processed.
- **processed_queue** (*Queue*) – Multiprocessing Queue to put the processed data.
- **config** (*pykg2vec.Config*) – Consists of the necessary parameters for training configuration.

pykg2vec.data.generator.**raw_data_generator**(*command_queue*, *raw_queue*, *config*)
Function to feed triples to raw queue for multiprocessing.

Parameters

- **command_queue** (*Queue*) – Each enqueued is either a command or a number of batch size.
- **raw_queue** (*Queue*) – Multiprocessing Queue to put the raw data to be processed.

- **config** (*pykg2vec.Config*) – Consists of the necessary parameters for training configuration.

5.2.3 pykg2vec.data.datasets

class `pykg2vec.data.datasets.`**`DeepLearning50a`**

This data structure defines the necessary information for downloading DeepLearning50a dataset.

DeepLearning50a module inherits the KnownDataset class for processing the knowledge graph dataset.

name

Name of the datasets

Type str

url

The full url where the dataset resides.

Type str

prefix

The prefix of the dataset given the website.

Type str

class `pykg2vec.data.datasets.`**`FreebaseFB15k`**

This data structure defines the necessary information for downloading Freebase dataset.

FreebaseFB15k module inherits the KnownDataset class for processing the knowledge graph dataset.

name

Name of the datasets

Type str

url

The full url where the dataset resides.

Type str

prefix

The prefix of the dataset given the website.

Type str

class `pykg2vec.data.datasets.`**`FreebaseFB15k_237`**

This data structure defines the necessary information for downloading FB15k-237 dataset.

FB15k-237 module inherits the KnownDataset class for processing the knowledge graph dataset.

name

Name of the datasets

Type str

url

The full url where the dataset resides.

Type str

prefix

The prefix of the dataset given the website.

Type str

class pykg2vec.data.datasets.**Kinship**

This data structure defines the necessary information for downloading Kinship dataset.

Kinship module inherits the KnownDataset class for processing the knowledge graph dataset.

name

Name of the datasets

Type str

url

The full url where the dataset resides.

Type str

prefix

The prefix of the dataset given the website.

Type str

class pykg2vec.data.datasets.**KnownDataset**(*name*, *url*, *prefix*)

The class consists of modules to handle the known datasets.

There are various known knowledge graph datasets used by the research community. These datasets maybe in different format. This module helps in parsing those known datasets for training and testing the algorithms.

Parameters

- **name** (*str*) – Name of the datasets
- **url** (*str*) – The full url where the dataset resides.
- **prefix** (*str*) – The prefix of the dataset given the website.

dataset_home_path

Path object where the data will be downloaded

Type object

root_oath

Path object for the specific dataset.

Type object

Examples

```
>>> from pykg2vec.data.kgcontroller import KnownDataset
>>> name = "dL50a"
>>> url = "https://github.com/louisccc/KGppler/raw/master/datasets/dL50a.tgz"
>>> prefix = 'deeplearning_dataset_50arch-'
>>> kgdata =  KnownDataset(name, url, prefix)
>>> kgdata.download()
>>> kgdata.extract()
>>> kgdata.dump()
```

download()

Downloads the given dataset from url

dump()

Displays all the metadata of the knowledge graph

extract()

Extract the downloaded file under the folder with the given dataset name

is_meta_cache_exists()
Checks if the metadata of the knowledge graph if available

read_metadata()
Reads the metadata of the knowledge graph if available

class pykg2vec.data.datasets.**NELL_995**
This data structure defines the necessary information for downloading NELL-995 dataset.

NELL-995 module inherits the KnownDataset class for processing the knowledge graph dataset.

name
Name of the datasets

Type str

url
The full url where the dataset resides.

Type str

prefix
The prefix of the dataset given the website.

Type str

class pykg2vec.data.datasets.**Nations**
This data structure defines the necessary information for downloading Nations dataset.

Nations module inherits the KnownDataset class for processing the knowledge graph dataset.

name
Name of the datasets

Type str

url
The full url where the dataset resides.

Type str

prefix
The prefix of the dataset given the website.

Type str

class pykg2vec.data.datasets.**UMLS**
This data structure defines the necessary information for downloading UMLS dataset.

UMLS module inherits the KnownDataset class for processing the knowledge graph dataset.

name
Name of the datasets

Type str

url
The full url where the dataset resides.

Type str

prefix
The prefix of the dataset given the website.

Type str

class `pykg2vec.data.datasets.`**`UserDefinedDataset`**(*name*, *custom_dataset_path*)

The class consists of modules to handle the user defined datasets.

User may define their own datasets to be processed with the pykg2vec library.

Parameters **`name`** (*`str`*) – Name of the datasets

`dataset_home_path`

Path object where the data will be downloaded

Type object

`root_oath`

Path object for the specific dataset.

Type object

`dump`()

Prints the metadata of the user-defined dataset.

`is_meta_cache_exists`()

Checks if the metadata has been cached

`read_metadata`()

Reads the metadata of the user defined dataset

class `pykg2vec.data.datasets.`**`WordNet18`**

This data structure defines the necessary information for downloading WordNet18 dataset.

WordNet18 module inherits the KnownDataset class for processing the knowledge graph dataset.

`name`

Name of the datasets

Type str

`url`

The full url where the dataset resides.

Type str

`prefix`

The prefix of the dataset given the website.

Type str

class `pykg2vec.data.datasets.`**`WordNet18_RR`**

This data structure defines the necessary information for downloading WordNet18_RR dataset.

WordNet18_RR module inherits the KnownDataset class for processing the knowledge graph dataset.

`name`

Name of the datasets

Type str

`url`

The full url where the dataset resides.

Type str

`prefix`

The prefix of the dataset given the website.

Type str

class `pykg2vec.data.datasets.`**`YAGO3_10`**

This data structure defines the necessary information for downloading YAGO3_10 dataset.

YAGO3_10 module inherits the KnownDataset class for processing the knowledge graph dataset.

`name`

Name of the datasets

Type str

`url`

The full url where the dataset resides.

Type str

`prefix`

The prefix of the dataset given the website.

Type str

`pykg2vec.data.datasets.`**`extract_tar`**(*tar_path*, *extract_path='.'*)

This function extracts the tar file.

Most of the knowledge graph datasets are downloaded in a compressed tar format. This function is used to extract them

Parameters

- **`tar_path`** (*`str`*) – Location of the tar folder.
- **`extract_path`** (*`str`*) – Path where the files will be decompressed.

`pykg2vec.data.datasets.`**`extract_zip`**(*zip_path*, *extract_path='.'*)

This function extracts the zip file.

Most of the knowledge graph datasets are downloaded in a compressed zip format. This function is used to extract them

Parameters

- **`zip_path`** (*`str`*) – Location of the zip folder.
- **`extract_path`** (*`str`*) – Path where the files will be decompressed.

5.3 pykg2vec.models

5.3.1 pykg2vec.models.pairwise

class `pykg2vec.models.pairwise.`**`HoLE`**(***kwargs*)

Holographic Embeddings of Knowledge Graphs. (HoLE) employs the circular correlation to create composition correlations. It is able to represent and capture the interactions betweek entities and relations while being efficient to compute, easier to train and scalable to large dataset.

Parameters **`config`** (*`object`*) – Model configuration parameters.

`embed`(*h*, *r*, *t*)

Function to get the embedding value.

Parameters

- **`h`** (*`Tensor`*) – Head entities ids.
- **`r`** (*`Tensor`*) – Relation ids of the triple.

- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns a 3-tuple of head, relation and tail embedding tensors.

Return type tuple

forward(*h*, *r*, *t*)
Function to get the embedding value

class pykg2vec.models.pairwise.**KG2E**(***kwargs*)
Learning to Represent Knowledge Graphs with Gaussian Embedding (KG2E) Instead of assumming entities and relations as determinstic points in the embedding vector spaces, KG2E models both entities and relations (h, r and t) using random variables derived from multivariate Gaussian distribution. KG2E then evaluates a fact using translational relation by evaluating the distance between two distributions, r and t-h. KG2E provides two distance measures (KL-divergence and estimated likelihood). Portion of the code based on mana-ysh's repository.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns a 6-tuple of head, relation and tail embedding tensors (both real and img parts).

Return type tuple

forward(*h*, *r*, *t*)
Function to get the embedding value

class pykg2vec.models.pairwise.**NTN**(***kwargs*)
Reasoning With Neural Tensor Networks for Knowledge Base Completion (NTN) is a neural tensor network which represents entities as an average of their constituting word vectors. It then projects entities to their vector embeddings in the input layer. The two entities are then combined and mapped to a non-linear hidden layer. https://github.com/siddharth-agrawal/Neural-Tensor-Network/blob/master/neuralTensorNetwork.py It is a neural tensor network which represents entities as an average of their constituting word vectors. It then projects entities to their vector embeddings in the input layer. The two entities are then combined and mapped to a non-linear hidden layer. Portion of the code based on siddharth-agrawal.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns head, relation and tail embedding Tensors.

Return type Tensors

forward(*h*, *r*, *t*)
Function to get the embedding value

get_reg(*h*, *r*, *t*)
Function to override if regularization is needed

train_layer(*h*, *t*)
Defines the forward pass training layers of the algorithm.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **t** (*Tensor*) – Tail entity ids of the triple.

class pykg2vec.models.pairwise.**Rescal**(***kwargs*)
A Three-Way Model for Collective Learning on Multi-Relational Data (RESCAL) is a tensor factorization approach to knowledge representation learning, which is able to perform collective learning via the latent components of the factorization. Rescal is a latent feature model where each relation is represented as a matrix modeling the iteraction between latent factors. It utilizes a weight matrix which specify how much the latent features of head and tail entities interact in the relation. Portion of the code based on mnick and OpenKE_Rescal.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns head, relation and tail embedding Tensors.

Return type Tensors

forward(*h*, *r*, *t*)
Function to get the embedding value

class pykg2vec.models.pairwise.**RotatE**(***kwargs*)
Rotate-Knowledge graph embedding by relation rotation in complex space (RotatE) models the entities and the relations in the complex vector space. The translational relation in RotatE is defined as the element-wise 2D rotation in which the head entity h will be rotated to the tail entity t by multiplying the unit-length relation r in complex number form.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value

class pykg2vec.models.pairwise.**SLM**(***kwargs*)
In Reasoning With Neural Tensor Networks for Knowledge Base Completion, SLM model is designed as a baseline of Neural Tensor Network. The model constructs a nonlinear neural network to represent the score function.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns head, relation and tail embedding Tensors.

Return type Tensors

forward(*h*, *r*, *t*)
Function to get the embedding value

layer(*h*, *t*)
Defines the forward pass layer of the algorithm.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **t** (*Tensor*) – Tail entity ids of the triple.

class pykg2vec.models.pairwise.**SME**(***kwargs*)
A Semantic Matching Energy Function for Learning with Multi-relational Data

Semantic Matching Energy (SME) is an algorithm for embedding multi-relational data into vector spaces. SME conducts semantic matching using neural network architectures. Given a fact (h, r, t), it first projects entities and relations to their embeddings in the input layer. Later the relation r is combined with both h and t to get gu(h, r) and gv(r, t) in its hidden layer. The score is determined by calculating the matching score of gu and gv.

There are two versions of SME: a linear version(SMELinear) as well as bilinear(SMEBilinear) version which differ in how the hidden layer is defined.

Parameters **config** (*object*) – Model configuration parameters.

Portion of the code based on glorotxa.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail entity ids of the triple.

Returns Returns head, relation and tail embedding Tensors.

Return type Tensors

forward(*h*, *r*, *t*)
Function to that performs semanting matching.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.

- **t** (*Tensor*) – Tail ids of the triple.

Returns Returns the semantic matchin score.

Return type Tensors

class pykg2vec.models.pairwise.**SME_BL**(***kwargs*)

A Semantic Matching Energy Function for Learning with Multi-relational Data

SME_BL is an extension of SME that BiLinear function to calculate the matching scores.

Parameters **config** (*object*) – Model configuration parameters.

forward(*h, r, t*)

Function to that performs semanting matching.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** (*Tensor*) – Tail ids of the triple.

Returns Returns the semantic matchin score.

Return type Tensors

class pykg2vec.models.pairwise.**TransD**(***kwargs*)

Knowledge Graph Embedding via Dynamic Mapping Matrix (TransD) is an improved version of TransR. For each triplet (h, r, t), it uses two mapping matrices M_{rh}, $M_{rt} \in R^{mn}$ to project entities from entity space to relation space. TransD constructs a dynamic mapping matrix for each entity-relation pair by considering the diversity of entities and relations simultaneously. Compared with TransR/CTransR, TransD has fewer parameters and has no matrix vector multiplication.

Parameters **config** (*object*) – Model configuration parameters.

Portion of the code based on OpenKE_TransD.

embed(*h, r, t*)

Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h, r, t*)

Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids.
- **t** – Tail entity ids.

class pykg2vec.models.pairwise.**TransE**(***kwargs*)

Translating Embeddings for Modeling Multi-relational Data (TransE) is an energy based model which represents the relationships as translations in the embedding space. Specifically, it assumes that if a fact (h, r, t) holds then the embedding of the tail 't' should be close to the embedding of head entity 'h' plus some vector that depends on the relationship 'r'. Which means that if (h,r,t) holds then the embedding of the tail 't' should be close to the embedding of head entity 'h' plus some vector that depends on the relationship 'r'. In TransE, both entities and relations are vectors in the same space

Parameters **config** (*object*) – Model configuration parameters.

Portion of the code based on OpenKE_TransE and wencolani.

embed(*h, r, t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids.
- **t** – Tail entity ids.

forward(*h, r, t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids.
- **t** – Tail entity ids.

class pykg2vec.models.pairwise.**TransH**(***kwargs*)
Knowledge Graph Embedding by Translating on Hyperplanes (TransH) follows the general principle of the TransE. However, compared to it, it introduces relation-specific hyperplanes. The entities are represented as vecotrs just like in TransE, however, the relation is modeled as a vector on its own hyperplane with a normal vector. The entities are then projected to the relation hyperplane to calculate the loss. TransH models a relation as a hyperplane together with a translation operation on it. By doing this, it aims to preserve the mapping properties of relations such as reflexive, one-to-many, many-to-one, and many-to-many with almost the same model complexity of TransE.

Parameters **config** (*object*) – Model configuration parameters.

Portion of the code based on OpenKE_TransH and thunlp_TransH.

embed(*h, r, t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h, r, t*)
Function to get the embedding value

class pykg2vec.models.pairwise.**TransM**(***kwargs*)
Transition-based Knowledge Graph Embedding with Relational Mapping Properties (TransM) is another line of research that improves TransE by relaxing the overstrict requirement of h+r ==> t. TransM associates each fact (h, r, t) with a weight theta(r) specific to the relation. TransM helps to remove the the lack of flexibility present in TransE when it comes to mapping properties of triplets. It utilizes the structure of the knowledge graph via pre-calculating the distinct weight for each training triplet according to its relational mapping property.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids.
- **t** – Tail entity ids.

class pykg2vec.models.pairwise.**TransR**(***kwargs*)
Learning Entity and Relation Embeddings for Knowledge Graph Completion (TransR) is a translation based knowledge graph embedding method. Similar to TransE and TransH, it also builds entity and relation embeddings by regarding a relation as translation from head entity to tail entity. However, compared to them, it builds the entity and relation embeddings in a separate entity and relation spaces. Portion of the code based on thunlp_transR.

Parameters config (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids.
- **t** – Tail entity ids.

5.3.2 pykg2vec.models.pointwise

class pykg2vec.models.pointwise.**ANALOGY**(***kwargs*)
Analogical Inference for Multi-relational Embeddings

Parameters config (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.

- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

embed_complex(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value

get_reg(*h*, *r*, *t*, *reg_type='F2'*)
Function to override if regularization is needed

class pykg2vec.models.pointwise.**CP**(***kwargs*)
Canonical Tensor Decomposition for Knowledge Base Completion

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value

get_reg(*h*, *r*, *t*, *reg_type='N3'*)
Function to override if regularization is needed

class pykg2vec.models.pointwise.**Complex**(***kwargs*)
Complex Embeddings for Simple Link Prediction (ComplEx) is an enhanced version of DistMult in that it uses complex-valued embeddings to represent both entities and relations. Using the complex-valued embedding allows the defined scoring function in ComplEx to differentiate that facts with assymmetric relations.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value

get_reg(*h*, *r*, *t*, *reg_type='F2'*)
Function to override if regularization is needed

class `pykg2vec.models.pointwise.`**`ComplexN3`**(***kwargs*)

Complex Embeddings for Simple Link Prediction (ComplEx) is an enhanced version of DistMult in that it uses complex-valued embeddings to represent both entities and relations. Using the complex-valued embedding allows the defined scoring function in ComplEx to differentiate that facts with assymmetric relations.

Parameters **`config`** (*`object`*) – Model configuration parameters.

`get_reg`(*h, r, t, reg_type='N3'*)

Function to override if regularization is needed

class `pykg2vec.models.pointwise.`**`ConvKB`**(***kwargs*)

In A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network (ConvKB), each triple (head entity, relation, tail entity) is represented as a 3-column matrix where each column vector represents a triple element

Portion of the code based on daiquocnguyen.

Parameters **`config`** (*`object`*) – Model configuration parameters.

`embed`(*h, r, t*)

Function to get the embedding value.

Parameters

- **`h`** (*`Tensor`*) – Head entities ids.
- **`r`** (*`Tensor`*) – Relation ids of the triple.
- **`t`** – Tail entity ids of the triple.

`forward`(*h, r, t*)

Function to get the embedding value

class `pykg2vec.models.pointwise.`**`DistMult`**(***kwargs*)

EMBEDDING ENTITIES AND RELATIONS FOR LEARNING AND INFERENCE IN KNOWLEDGE BASES (DistMult) is a simpler model comparing with RESCAL in that it simplifies the weight matrix used in RESCAL to a diagonal matrix. The scoring function used DistMult can capture the pairwise interactions between the head and the tail entities. However, DistMult has limitation on modeling asymmetric relations.

Parameters **`config`** (*`object`*) – Model configuration parameters.

`embed`(*h, r, t*)

Function to get the embedding value.

Parameters

- **`h`** (*`Tensor`*) – Head entities ids.
- **`r`** (*`Tensor`*) – Relation ids of the triple.
- **`t`** – Tail entity ids of the triple.

`forward`(*h, r, t*)

Function to get the embedding value

`get_reg`(*h, r, t, reg_type='F2'*)

Function to override if regularization is needed

class `pykg2vec.models.pointwise.`**`MuRP`**(***kwargs*)

Multi-relational Poincaré Graph Embeddings

Parameters **`config`** (*`object`*) – Model configuration parameters.

`embed`(*h, r, t*)

Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h, r, t*)
Function to get the embedding value

class pykg2vec.models.pointwise.**OctonionE**(***kwargs*)
Quaternion Knowledge Graph Embeddings

Parameters **config** (*object*) – Model configuration parameters.

embed(*h, r, t*)
Function to get the embedding value

forward(*h, r, t*)
Function to get the embedding value

get_reg(*h, r, t, reg_type='N3'*)
Function to override if regularization is needed

class pykg2vec.models.pointwise.**QuatE**(***kwargs*)
Quaternion Knowledge Graph Embeddings

Parameters **config** (*object*) – Model configuration parameters.

embed(*h, r, t*)
Function to get the embedding value

forward(*h, r, t*)
Function to get the embedding value

get_reg(*h, r, t, reg_type='N3'*)
Function to override if regularization is needed

class pykg2vec.models.pointwise.**SimplE**(***kwargs*)
SimplE Embedding for Link Prediction in Knowledge Graphs

Parameters **config** (*object*) – Model configuration parameters.

embed(*h, r, t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h, r, t*)
Function to get the embedding value

get_reg(*h, r, t, reg_type='F2'*)
Function to override if regularization is needed

class pykg2vec.models.pointwise.**SimplE_ignr**(***kwargs*)
SimplE Embedding for Link Prediction in Knowledge Graphs

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*h*, *r*, *t*)
Function to get the embedding value

5.3.3 pykg2vec.models.projection

class pykg2vec.models.projection.**AcrE**(***kwargs*)
Knowledge Graph Embedding with Atrous Convolution and Residual Learning

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*e*, *r*, *direction='tail'*)
Function to get the embedding value

class pykg2vec.models.projection.**ConvE**(***kwargs*)
Convolutional 2D Knowledge Graph Embeddings (ConvE) is a multi-layer convolutional network model for link prediction, it is a embedding model which is highly parameter efficient. ConvE is the first non-linear model that uses a global 2D convolution operation on the combined and head entity and relation embedding vectors. The obtained feature maps are made flattened and then transformed through a fully connected layer. The projected target vector is then computed by performing linear transformation (passing through the fully connected layer) and activation function, and finally an inner product with the latent representation of every entities.

Parameters **config** (*object*) – Model configuration parameters.

embed(*h*, *r*, *t*)
Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward(*e*, *r*, *direction='tail'*)
Function to get the embedding value

inner_forward(*st_inp*, *first_dimension_size*)
Implements the forward pass layers of the algorithm.

class pykg2vec.models.projection.**HypER**(***kwargs*)
HypER: Hypernetwork Knowledge Graph Embeddings

Parameters **config** (*object*) – Model configuration parameters.

embed (*h*, *r*, *t*)

Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward (*e*, *r*, *direction='tail'*)

Function to get the embedding value

class pykg2vec.models.projection.**InteractE** (***kwargs*)

InteractE: Improving Convolution-based Knowledge Graph Embeddings by Increasing Feature Interactions

Parameters **config** (*object*) – Model configuration parameters.

embed (*h*, *r*, *t*)

Function to get the embedding value.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.
- **t** – Tail entity ids of the triple.

forward (*e*, *r*, *direction='tail'*)

Function to get the embedding value

class pykg2vec.models.projection.**ProjE_pointwise** (***kwargs*)

ProjE-Embedding Projection for Knowledge Graph Completion. (ProjE) Instead of measuring the distance or matching scores between the pair of the head entity and relation and then tail entity in embedding space ((h,r) vs (t)). ProjE projects the entity candidates onto a target vector representing the input data. The loss in ProjE is computed by the cross-entropy between the projected target vector and binary label vector, where the included entities will have value 0 if in negative sample set and value 1 if in positive sample set. Instead of measuring the distance or matching scores between the pair of the head entity and relation and then tail entity in embedding space ((h,r) vs (t)). ProjE projects the entity candidates onto a target vector representing the input data. The loss in ProjE is computed by the cross-entropy between the projected target vector and binary label vector, where the included entities will have value 0 if in negative sample set and value 1 if in positive sample set.

Parameters **config** (*object*) – Model configuration parameters.

f1 (*h*, *r*)

Defines froward layer for head.

Parameters

- **h** (*Tensor*) – Head entities ids.
- **r** (*Tensor*) – Relation ids of the triple.

f2 (*t*, *r*)

Defines forward layer for tail.

Parameters

- **t** (*Tensor*) – Tail entities ids.
- **r** (*Tensor*) – Relation ids of the triple.

forward(*e*, *r*, *er_e2*, *direction='tail'*)
Function to get the embedding value

static g(*f*, *w*)
Defines activation layer.

Parameters

- **f** (*Tensor*) – output of the forward layers.
- **w** (*Tensor*) – Matrix for multiplication.

get_reg(*h*, *r*, *t*)
Function to override if regularization is needed

class pykg2vec.models.projection.**TuckER**(***kwargs*)
TuckER-Tensor Factorization for Knowledge Graph Completion (TuckER) is a Tensor-factorization-based embedding technique based on the Tucker decomposition of a third-order binary tensor of triplets. Although being fully expressive, the number of parameters used in Tucker only grows linearly with respect to embedding dimension as the number of entities or relations in a knowledge graph increases. TuckER is a Tensor-factorization-based embedding technique based on the Tucker decomposition of a third-order binary tensor of triplets. Although being fully expressive, the number of parameters used in Tucker only grows linearly with respect to embedding dimension as the number of entities or relations in a knowledge graph increases. The author also showed in paper that the models, such as RESCAL, DistMult, ComplEx, are all special case of TuckER.

Parameters **config** (*object*) – Model configuration parameters.

forward(*e1*, *r*, *direction='head'*)
Implementation of the layer.

Parameters

- **e1** (*Tensor*) – entities id.
- **r** (*Tensor*) – Relation id.

Returns Returns the activation values.

Return type Tensors

5.3.4 pykg2vec.models.Domain

Domain module for building Knowledge Graphs

class pykg2vec.models.Domain.**NamedEmbedding**(*name*, **args*, ***kwargs*)
Associate embeddings with human-readable names

5.3.5 pykg2vec.models.KGMeta

Knowledge Graph Meta Class

It provides Abstract class for the Knowledge graph models.

class pykg2vec.models.KGMeta.**Model**
Meta Class for knowledge graph embedding models

embed(*h*, *r*, *t*)
Function to get the embedding value

forward(*h*, *r*, *t*)
Function to get the embedding value

get_reg(*h, r, t, **kwargs*)
Function to override if regularization is needed

load_params(*param_list, kwargs*)
Function to load the hyperparameters

class pykg2vec.models.KGMeta.**PairwiseModel**(*model_name*)
Meta Class for KGE models with translational distance

class pykg2vec.models.KGMeta.**PointwiseModel**(*model_name*)
Meta Class for KGE models with semantic matching

class pykg2vec.models.KGMeta.**ProjectionModel**(*model_name*)
Meta Class for KGE models with neural network

5.4 pykg2vec.utils

5.4.1 pykg2vec.utils.bayesian_optimizer

This module is for performing bayesian optimization on algorithms

class pykg2vec.utils.bayesian_optimizer.**BaysOptimizer**(*args*)
Bayesian optimizer class for tuning hyperparameter.

This class implements the Bayesian Optimizer for tuning the hyper-parameter.

Parameters

- **args** (*object*) – The Argument Parser object providing arguments.
- **name_dataset** (*str*) – The name of the dataset.
- **sampling** (*str*) – sampling to be used for generating negative triples

Examples

```
>>> from pykg2vec.common import KGEArgParser
>>> from pykg2vec.utils.bayesian_optimizer import BaysOptimizer
>>> model = Complex()
>>> args = KGEArgParser().get_args(sys.argv[1:])
>>> bays_opt = BaysOptimizer(args=args)
>>> bays_opt.optimize()
```

optimize()
Function that performs bayesian optimization

return_best()
Function to return the best hyper-parameters

5.4.2 pykg2vec.utils.criterion

class pykg2vec.utils.criterion.**Criterion**

Utility for calculating KGE losses

Loss Functions in Knowledge Graph Embedding Models http://ceur-ws.org/Vol-2377/paper_1.pdf

5.4.3 pykg2vec.utils.evaluator

This module is for evaluating the results

class pykg2vec.utils.evaluator.**Evaluator**(*model*, *config*, *tuning=False*)

Class to perform evaluation of the model.

Parameters

- **model** (*object*) – Model object
- **tuning** (*bool*) – Flag to denoting tuning if True

Examples

```
>>> from pykg2vec.utils.evaluator import Evaluator
>>> evaluator = Evaluator(model=model, tuning=True)
>>> evaluator.test_batch(Session(), 0)
>>> acc = evaluator.output_queue.get()
>>> evaluator.stop()
```

class pykg2vec.utils.evaluator.**MetricCalculator**(*config*)

MetricCalculator aims to 1) address all the statistic tasks. 2) provide interfaces for querying results.

MetricCalculator is expected to be used by “evaluation_process”.

display_summary()

Function to print the test summary.

get_head_rank(*head_candidate*, *h*, *r*, *t*)

Function to evaluate the head rank.

Parameters

- **head_candidate** (*list*) – List of the predicted head for the given tail, relation pair
- **h** (*int*) – head id
- **r** (*int*) – relation id
- **t** – tail id

get_tail_rank(*tail_candidate*, *h*, *r*, *t*)

Function to evaluate the tail rank.

Parameters

- **id_replace_tail** (*list*) – List of the predicted tails for the given head, relation pair
- **h** (*int*) – head id
- **r** (*int*) – relation id
- **t** (*int*) – tail id

- **hr_t** – list of tails for the given hwS and relation pari.

save_test_summary(*model_name*)
Function to save the test of the summary.

Parameters **model_name** (*str*) – specify the name of the model.

5.4.4 pykg2vec.utils.logger

class pykg2vec.utils.logger.**Logger**(**args*, ***kwargs*)
Basic logging

class pykg2vec.utils.logger.**Singleton**(**args*, ***kwargs*)
Singleton meta

5.4.5 pykg2vec.utils.riemannian_optimizer

class pykg2vec.utils.riemannian_optimizer.**RiemannianOptimizer**(*params*, *lr*, *param_names*)
Riemannian stochastic gradient descent

step(*lr=None*)
Performs a single optimization step (parameter update).

Parameters **closure** (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

Note: Unless otherwise specified, this function should not modify the `.grad` field of the parameters.

5.4.6 pykg2vec.utils.trainer

class pykg2vec.utils.trainer.**EarlyStopper**(*patience*, *monitor*)
Class used by trainer for handling the early stopping mechanism during the training of KGE algorithms.

Parameters

- **patience** (*int*) – Number of epochs to wait before early stopping the training on no improvement.
- **early stopping if it is a negative number (default** (*No*) – {-1}).
- **monitor** (*Monitor*) – the type of metric that earlystopper will monitor.

class pykg2vec.utils.trainer.**Trainer**(*model*, *config*)
Class for handling the training of the algorithms.

Parameters **model** (*object*) – KGE model object

Examples

```
>>> from pykg2vec.utils.trainer import Trainer
>>> from pykg2vec.models.pairwise import TransE
>>> trainer = Trainer(TransE())
>>> trainer.build_model()
>>> trainer.train_model()
```

build_model(*monitor=<Monitor.FILTERED_MEAN_RANK: 'fmr'>*)
function to build the model

display()
Function to display embedding.

export_embeddings()
Export embeddings in tsv and pandas pickled format. With tsvs (both label, vector files), you can: 1) Use those pretained embeddings for your applications. 2) Visualize the embeddings in this website to gain insights. (https://projector.tensorflow.org/)

Pandas dataframes can be read with pd.read_pickle('desired_file.pickle')

load_model(*model_path=None*)
Function to load the model.

save_model()
Function to save the model.

save_training_result()
Function that saves training result

train_model()
Function to train the model.

train_model_epoch(*epoch_idx*, *tuning=False*)
Function to train the model for one epoch.

tune_model()
Function to tune the model.

5.4.7 pykg2vec.utils.visualization

This module is for visualizing the results

class pykg2vec.utils.visualization.**Visualization**(*model*, *config*, *vis_opts=None*)
Class to aid in visualizing the results and embddings.

Parameters

- **model** (*object*) – Model object
- **vis_opts** (*list*) – Options for visualization.
- **sess** (*object*) – TensorFlow session object, initialized by the trainer.

Examples

```
>>> from pykg2vec.utils.visualization import Visualization
>>> from pykg2vec.utils.trainer import Trainer
>>> from pykg2vec.models.TransE import TransE
>>> model = TransE()
>>> trainer = Trainer(model=model)
>>> trainer.build_model()
>>> trainer.train_model()
>>> viz = Visualization(model=model)
>>> viz.plot_train_result()
```

static draw_embedding(*embs*, *names*, *resultpath*, *algos*, *show_label*)

Function to draw the embedding.

Parameters

- **embs** (*matrix*) – Two dimesnional embeddings.
- **names** (*list*) – List of string name.
- **resultpath** (*str*) – Path where the result will be save.
- **algos** (*str*) – Name of the algorithms which generated the algorithm.
- **show_label** (*bool*) – If True, prints the string names of the entities and relations.

static draw_embedding_rel_space(*h_emb*, *r_emb*, *t_emb*, *h_name*, *r_name*, *t_name*, *resultpath*, *algos*, *show_label*)

Function to draw the embedding in relation space.

Parameters

- **h_emb** (*matrix*) – Two dimesnional embeddings of head.
- **r_emb** (*matrix*) – Two dimesnional embeddings of relation.
- **t_emb** (*matrix*) – Two dimesnional embeddings of tail.
- **h_name** (*list*) – List of string name of the head.
- **r_name** (*list*) – List of string name of the relation.
- **t_name** (*list*) – List of string name of the tail.
- **resultpath** (*str*) – Path where the result will be save.
- **algos** (*str*) – Name of the algorithms which generated the algorithm.
- **show_label** (*bool*) – If True, prints the string names of the entities and relations.

get_idx_n_emb()

Function to get the integer ids and the embedding.

plot_embedding(*resultpath=None*, *algos=None*, *show_label=False*, *disp_num_r_n_e=20*)

Function to plot the embedding.

Parameters

- **resultpath** (*str*) – Path where the result will be saved.
- **show_label** (*bool*) – If True, will display the labels.
- **algos** (*str*) – Name of the algorithms that generated the embedding.
- **disp_num_r_n_e** (*int*) – Total number of entities to display for head, tail and relation.

plot_test_result()
Function to plot the testing result.

plot_train_result()
Function to plot the training result.

5.5 pykg2vec.test

After installation, you can use *pytest* to run the test suite from pykg2vec's root directory:

```
pytest
```

5.5.1 pykg2vec.test.test_generator

This module is for testing unit functions of generator

pykg2vec.test.test_generator.**test_generator_pairwise**()
Function to test the generator for pairwise based algorithm.

pykg2vec.test.test_generator.**test_generator_pointwise**()
Function to test the generator for pointwise based algorithm.

pykg2vec.test.test_generator.**test_generator_projection**()
Function to test the generator for projection based algorithm.

5.5.2 pykg2vec.test.test_hp_loader

This module is for testing unit functions of the hyperparameter loader

5.5.3 pykg2vec.test.test_inference

This module is for testing unit functions of model

pykg2vec.test.test_inference.**test_inference**(*model_name*)
Function to test Algorithms with arguments.

pykg2vec.test.test_inference.**testing_function_with_args**(*name*, *l1_flag*, *display=False*)
Function to test the models with arguments.

5.5.4 pykg2vec.test.test_kg

This module is for testing unit functions of KnowledgeGraph

pykg2vec.test.test_kg.**test_known_datasets**(*dataset_name*)
Function to test the the knowledge graph parse for Freebase.

5.5.5 pykg2vec.test.test_logger

This module is for testing unit functions of Logger

5.5.6 pykg2vec.test.test_model

This module is for testing unit functions of model

`pykg2vec.test.test_model.`**`test_kge_methods`**(*model_name*)
    Function to test a set of KGE algorithsm.

`pykg2vec.test.test_model.`**`testing_function`**(*name*)
    Function to test the models with arguments.

5.5.7 pykg2vec.test.test_trainer

This module is for testing unit functions of training

5.5.8 pykg2vec.test.test_tune_model

This module is for testing unit functions of tuning model

`pykg2vec.test.test_tune_model.`**`test_return_empty_before_optimization`**(*mocked_fmin*)
    Function to test the tuning of the models.

`pykg2vec.test.test_tune_model.`**`test_tuning`**(*model_name*)
    Function to test the tuning function.

`pykg2vec.test.test_tune_model.`**`tunning_function`**(*name*)
    Function to test the tuning of the models.

5.5.9 pykg2vec.test.test_inference

This module is for integration tests on visualization

CHAPTER

SIX

CONTRIBUTE TO PYKG2VEC

We feel humbled that you have decided to contribute to the pykg2vec repository. Thank you! Please read the following guidelines to checkout how you can contribute.

You can contribute to this code through Pull Request on GitHub. Please, make sure that your code is coming with unit tests to ensure full coverage and continuous integration in the API.

- **Reporting Bugs**: Please use the issue Template to report bugs.
- **Suggesting Enhancements**: If you have any suggestion for enhancing any of the modules please send us an enhancement using the issue Template as well.
- **Adding Algorithm**: We are continually striving to add the state-of-the-art algorithms in the library. If you want to suggest adding any algorithm or add your algoirithm to the library, please follow the following steps:
  - Make sure the generator is able to produce the batches
  - Make sure to follow the class structure presented in pykg2vec/core/KGMeta.py
- **Adding Evaluation Metric**: We are always eager to add more evaluation metrics for link prediction, triple classification, and so on. You may create a new evaluation process in pykg2vec/utils/evaluation.py to add the metric.
- **Csource for Python Modules**: Although we use Tensorflow for running the main modules, there are many alforithms written in pure python. We invite you to contibute by converting the python source code to more efficient C or C++ codes.
- **Adding Dataset Source**: We encourage you to add your own dataset links. Currenlty the pykg2vec/config/global_config.py handles the datasets, how to extract them and generate the training, testing and validation triples.

CHAPTER SEVEN

AUTHORS & CITATION & LICENSE

Core Development

```
Shih-Yuan Yu (M.S.)
University of California, Irvine
Email: shihyuay@uci.edu

Xi Bai (Ph.D)
Software Engineer, BBC Design & Engineering
Email: xi.bai.ed@gmail.com

Sujit Rokka Chhetri (Ph.D)
University of California, Irvine
Email: schhetri@uci.edu
```

Contributors

```
Fangzhou Du (B.S.)
University of California, Irvine
Email: fangzhod@uci.edu

Arquimedes Martinez Canedo (Ph.D)
Principal Scientist, Siemens Corporate Technology
Email: arquimedes.canedo@siemens.com

Mohammad Al Faruque (Ph.D)
Associate Professor, University of California, Irvine
Email: alfaruqu@uci.edu
```

Citing pykg2vec

If you found pykg2ve is helpful in your research or usages, please kindly consider citing us

```
@online{pykg2vec,
  author = {Rokka Chhetri, Sujit and  Yu, Shih-Yuan and  Salih Aksakal, Ahmet and ␣
↪Goyal, Palash and  Canedo, Arquimedes and Al Faruque, Mohammad},
  title = {{pykg2vec: Python Knowledge Graph Embedding Library},
  year = 2019,
  url = {https://pypi.org/project/pykg2vec/}
  }
```

License Information

Pykg2vec uses The MIT License

PYTHON MODULE INDEX

INDEX