AmpliGraph

Release 1.0.0

Luca Costabello - Accenture Labs Dublin

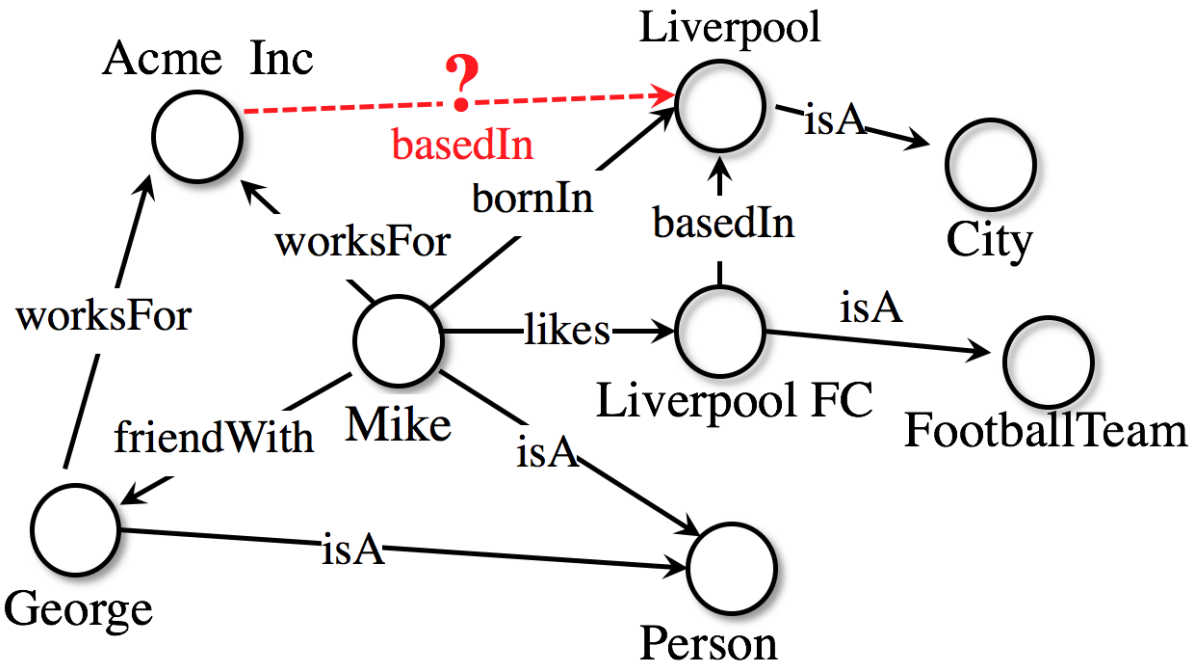
Mar 15, 2019

Contents:

Open source Python library that predicts links between concepts in a knowledge graph.

View the GitHub repository 

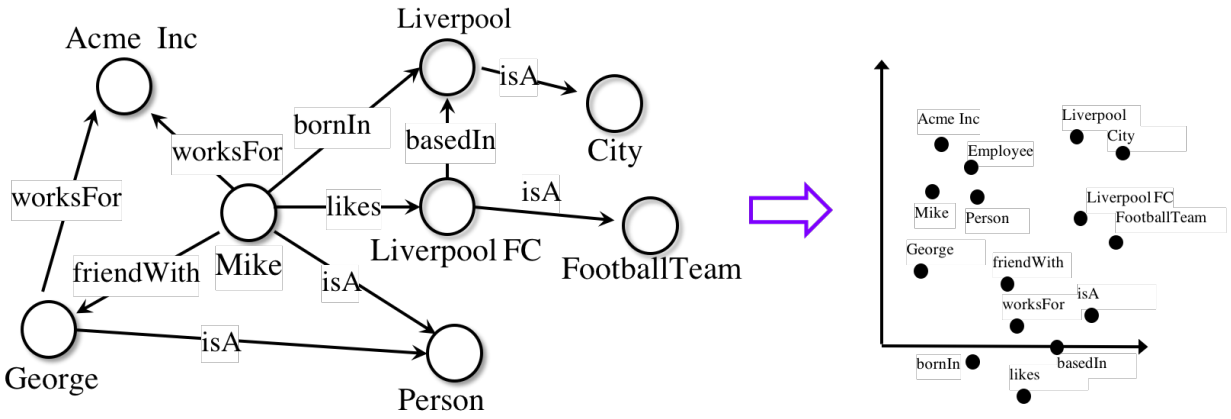
AmpliGraph is a suite of neural machine learning models for relational Learning, a branch of machine learning that deals with supervised learning on knowledge graphs.



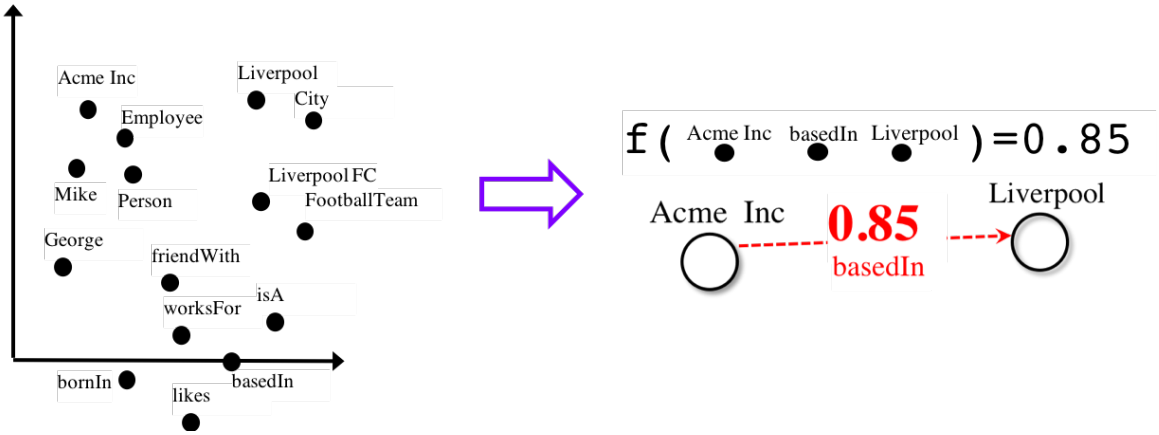
Use AmpliGraph if you need to:

- Discover new knowledge from an existing knowledge graph.
- Complete large knowledge graphs with missing statements.
- Generate stand-alone knowledge graph embeddings.
- Develop and evaluate a new relational model.

AmpliGraph's machine learning models generate **knowledge graph embeddings**, vector representations of concepts in a metric space:



It then combines embeddings with model-specific scoring functions to predict unseen and novel links:



CHAPTER 1

Key Features

- **Intuitive APIs**: AmpliGraph APIs are designed to reduce the code amount required to learn models that predict links in knowledge graphs.
- **GPU-Ready**: AmpliGraph is based on TensorFlow, and it is designed to run seamlessly on CPU and GPU devices - to speed-up training.
- **Extensible**: Roll your own knowledge graph embeddings model by extending AmpliGraph base estimators.

CHAPTER 2

Modules

AmpliGraph includes the following submodules:

- **Input**: Helper functions to load datasets (knowledge graphs).
- **Latent Feature Models**: knowledge graph embedding models. AmpliGraph contains: TransE, DistMult, ComplEx, HolE. (More to come!)
- **Evaluation**: Metrics and evaluation protocols to assess the predictive power of the models.

CHAPTER 3

How to Cite

If you like AmpliGraph and you use it in your project, why not starring the project on GitHub!

If you instead use AmpliGraph in an academic publication, cite as:

```
@misc{ampligraph,
  author= {Luca Costabello and
           Sumit Pai and
           Chan Le Van and
           Rory McGrath and
           Nick McCarthy},
  title = {{AmpliGraph: a Library for Representation Learning on Knowledge Graphs}},
  month = mar,
  year  = 2019,
  doi   = {10.5281/zenodo.2595049},
  url   = {https://doi.org/10.5281/zenodo.2595049}
}
```

3.1 Installation

3.1.1 Prerequisites

- Linux Box
- Python 3.6

Provision a Virtual Environment

Create and activate a virtual environment (conda)

```
conda create --name ampligraph python=3.6
source activate ampligraph
```

Install TensorFlow

AmpliGraph is built on TensorFlow 1.x. Install from pip or conda:

CPU-only

```
pip install tensorflow==1.12.0

or

conda install tensorflow=1.12.0
```

GPU support

```
pip install tensorflow-gpu==1.12.0

or

conda install tensorflow-gpu=1.12.0
```

3.1.2 Install AmpliGraph

Install the latest stable release from pip:

```
pip install ampligraph
```

If instead you want the most recent development version, you can clone the repository and install from source (your local working copy will be on the latest commit on the `develop` branch). The code snippet below will install the library in editable mode (`-e`):

```
git clone https://github.com/Accenture/AmpliGraph.git
cd AmpliGraph
pip install -e .
```

3.1.3 Sanity Check

```
>> import ampligraph
>> ampligraph.__version__
'1.0.0'
```

3.2 Background

Knowledge graphs are graph-based knowledge bases whose facts are modeled as relationships between entities. Knowledge graph research led to broad-scope graphs such as DBpedia *[ABK+07]*, WordNet *[Pri10]*, and YAGO *[SKW07]*. Countless domain-specific knowledge graphs have also been published on the web, giving birth to the so-called Web of Data *[BHBL11]*.

Formally, a knowledge graph $\mathcal{G} = \{(sub, pred, obj)\} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ is a set of $(sub, pred, obj)$ triples, each including a subject $sub \in \mathcal{E}$, a predicate $pred \in \mathcal{R}$, and an object $obj \in \mathcal{E}$. $\mathcal{E}$ and $\mathcal{R}$ are the sets of all entities and relation types of $\mathcal{G}$.

Knowledge graph embedding models are neural architectures that encode concepts from a knowledge graph (i.e. entities $\mathcal{E}$ and relation types $\mathcal{R}$) into low-dimensional, continuous vectors $\in \mathcal{R}^k$. Such textit{knowledge graph embeddings} have applications in knowledge graph completion, entity resolution, and link-based clustering, just to cite a few *[NMTG16]*. Knowledge graph embeddings are learned by training a neural architecture over a graph. Although such architectures vary, the training phase always consists in minimizing a loss function $\mathcal{L}$ that includes a *scoring function* $f_m(t)$, i.e. a model-specific function that assigns a score to a triple $t = (sub, pred, obj)$.

The goal of the optimization procedure is learning optimal embeddings, such that the scoring function is able to assign high scores to positive statements and low scores to statements unlikely to be true. Existing models propose scoring functions that combine the embeddings $\mathbf{e}_{sub}, \mathbf{e}_{pred}, \mathbf{e}_{obj} \in \mathcal{R}^k$ of the subject, predicate, and object of triple $t = (sub, pred, obj)$ using different intuitions: TransE *[BUGD+13]* relies on distances, DistMult *[YYH+14]* and ComplEx *[TWR+16]* are bilinear-diagonal models, HolE *[NRP+16]* uses circular correlation. While the above models can be interpreted as multilayer perceptrons, others such as ConvE include convolutional layers *[DMSR18]*.

As example, the scoring function of TransE computes a similarity between the embedding of the subject $\mathbf{e}_{sub}$ translated by the embedding of the predicate $\mathbf{e}_{pred}$ and the embedding of the object $\mathbf{e}_{obj}$, using the L_1 or L_2 norm $|| \cdot ||$:

$$f_{TransE} = -||\mathbf{e}_{sub} + \mathbf{e}_{pred} - \mathbf{e}_{obj}||_n$$

Such scoring function is then used on positive and negative triples t^+, t^- in the loss function. This can be for example a pairwise margin-based loss, as shown in the equation below:

$$\mathcal{L}(\Theta) = \sum_{t^+ \in \mathcal{G}} \sum_{t^- \in \mathcal{N}} max(0, [\gamma + f_m(t^-; \Theta) - f_m(t^+; \Theta)])$$

where Θ are the embeddings learned by the model, f_m is the model-specific scoring function, $\gamma \in \mathcal{R}$ is the margin and $\mathcal{N}$ is a set of negative triples generated with a corruption heuristic *[BUGD+13]*.

3.3 API

AmpliGraph includes the following submodules:

3.3.1 Datasets

Helper functions to load knowledge graphs from disk.

Note: It is recommended to set the `AMPLIGRAPH_DATA_HOME` environment variable:

```
export AMPLIGRAPH_DATA_HOME=/YOUR/PATH/TO/datasets
```

When attempting to load a dataset, the module will first check if `AMPLIGRAPH_DATA_HOME` is set. If it is, it will search this location for the required dataset. If the dataset is not found it will be downloaded and placed in this directory.

If `AMPLIGRAPH_DATA_HOME` has not been set the databases will be saved in the following directory:

```
~/ampligraph_datasets
```

Additionally, a specific directory can be passed to the dataset loader via the `data_home` parameter.

Dataset-Specific Loaders

Use these helpers functions to load datasets used in graph representation learning literature. The functions will automatically download the datasets if they are not present in `~/ampligraph_datasets` or at the location set in `AMPLIGRAPH_DATA_HOME`.

load_wn18([data_home])	Load the WN18 dataset
load_fb15k([data_home])	Load the FB15k dataset
load_fb15k_237([data_home])	Load the FB15k-237 dataset
load_yago3_10([data_home])	Load the YAGO3-10 dataset
load_wn18rr([data_home])	Load the WN18RR dataset

load_wn18

ampligraph.datasets.**load_wn18**(*data_home=None*)

Load the WN18 dataset

WN18 is a subset of Wordnet. It was first presented by *[BUGD+13]*. The dataset is divided in three splits:

- `train`
- `valid`
- `test`

Returns **splits** – The dataset splits {'train': train, 'valid': valid, 'test': test}. Each split is an ndarray of shape [n, 3].

Return type dict

Examples

```
>>> from ampligraph.datasets import load_wn18
>>> X = load_wn18()
>>> X['test'][:3]
array([['06845599', '_member_of_domain_usage', '03754979'],
       ['00789448', '_verb_group', '01062739'],
       ['10217831', '_hyponym', '10682169']], dtype=object)
```

load_fb15k

ampligraph.datasets.**load_fb15k**(*data_home=None*)

Load the FB15k dataset

FB15k is a split of Freebase, first proposed by *[BUGD+13]*.

The dataset is divided in three splits:

- `train`
- `valid`
- `test`

Returns splits – The dataset splits: {'train': train, 'valid': valid, 'test': test}. Each split is an ndarray of shape [n, 3].

Return type dict

Examples

```
>>> from ampligraph.datasets import load_fb15k
>>> X = load_fb15k()
>>> X['test'][:3]
array([['/m/01qscs',
        '/award/award_nominee/award_nominations./award/award_nomination/award',
        '/m/02x8n1n'],
       ['/m/040db', '/base/activism/activist/area_of_activism', '/m/0148d'],
       ['/m/08966',
        '/travel/travel_destination/climate./travel/travel_destination_monthly_
↪climate/month',
        '/m/05lf_']], dtype=object)
```

load_fb15k_237

ampligraph.datasets.**load_fb15k_237**(*data_home=None*)

Load the FB15k-237 dataset

FB15k-237 is a reduced version of FB15k. It was first proposed by *[TCP+15]*. The dataset is divided in three splits: - `train` - `valid` - `test`

Returns splits – The dataset splits: {'train': train, 'valid': valid, 'test': test}. Each split is an ndarray of shape [n, 3].

Return type dict

Examples

```
>>> from ampligraph.datasets import load_fb15k_237
>>> X = load_fb15k_237()
>>> X["train"][2]
array(['/m/07s9rl0', '/media_common/netflix_genre/titles', '/m/0170z3'],
  dtype=object)
```

load_yago3_10

ampligraph.datasets.**load_yago3_10**(*data_home=None*)

Load the YAGO3-10 dataset

The dataset is presented in *[MBS13]*. It is divided in three splits:

- `train`
- `valid`
- `test`

Returns splits – The dataset splits: {'train': train, 'valid': valid, 'test': test}. Each split is an ndarray of shape [n, 3].

Return type dict

Examples

```
>>> from ampligraph.datasets import load_yago3_10
>>> X = load_yago3_10()
>>> X["valid"][0]
array(['Mikheil_Khutsishvili', 'playsFor', 'FC_Merani_Tbilisi'], dtype=object)
```

load_wn18rr

ampligraph.datasets.**load_wn18rr**(*data_home=None*)

Load the WN18RR dataset

The dataset is described in *[DMSR18]*. It is divided in three splits:

- `train`
- `valid`
- `test`

Returns splits – The dataset splits: {'train': train, 'valid': valid, 'test': test}. Each split is an ndarray of shape [n, 3].

Return type dict

Examples

```
>>> from ampligraph.datasets import load_wn18rr
>>> X = load_wn18rr()
>>> X["valid"][0]
array(['02174461', '_hypernym', '02176268'], dtype=object)
```

Dataset Summary

Dataset	Train	Valid	Test	Entities	Relations
FB15K-237	272,115	17,535	20,466	14,541	237
WN18RR	86,835	3,034	3,134	40,943	11
FB15K	483,142	50,000	59,071	14,951	1,345
WN18	141,442	5,000	5,000	40,943	18
YAGO3-10	1,079,040	5,000	5,000	123,182	37

These datasets are originated from: FB15K-237, WN18RR, FB15K, WN18, YAGO3-10

Warning: FB15K-237 contains 8 unseen entities inside 9 triples in the validation set and 29 inside 28 triples in the test set. WN18RR contains 198 unseen entities inside 210 triples in the validation set and 209 inside 210 triples in the test set.

Generic Loaders

Functions to load custom knowledge graphs from disk.

Note: The environment variable `AMPLIGRAPH_DATA_HOME` must be set and input graphs must be stored at the path indicated.

load_from_csv(directory_path, file_name[, ...])	Load a csv file
load_from_ntriples(folder_name, file_name[, ...])	Load RDF ntriples as csv statements
load_from_rdf(folder_name, file_name[, ...])	Load an RDF file

load_from_csv

ampligraph.datasets.**load_from_csv**(*directory_path*, *file_name*, *sep='\t'*, *header=None*)

Load a csv file

Loads a knowledge graph serialized in a csv file as: .. code-block:: text

subj1 relationX obj1 subj1 relationY obj2 subj3 relationZ obj2 subj4 relationY obj2 ...

Note: Duplicates are filtered.

Parameters

- **folder_name** (*str*) – base folder within AMPLIGRAPH_DATA_HOME where the file is stored.
- **file_name** (*str*) – file name
- **sep** (*str*) – The subject-predicate-object separator (default).
- **header** (*int, None*) – The row of the header of the csv file. Same as pandas.read_csv header param.

Returns **triples** – the actual triples of the file.

Return type ndarray , shape [n, 3]

Examples

```
>>> from ampligraph.datasets import load_from_csv
>>> X = load_from_csv('folder', 'dataset.csv', sep=',')
>>> X[:3]
array([['a', 'y', 'b'],
       ['b', 'y', 'a'],
       ['a', 'y', 'c']],
      dtype='<U1')
```

load_from_ntriples

ampligraph.datasets.**load_from_ntriples**(*folder_name*, *file_name*, *data_home=None*)
Load RDF ntriples as csv statements

Loads an RDF knowledge graph serialized as ntriples, without building an RDF graph in mmeory. This function is faster than `load_from_rdf()`.

Parameters

- **folder_name** (*str*) – base folder within AMPLIGRAPH_DATA_HOME where the file is stored.
- **file_name** (*str*) – file name

Returns **triples** – the actual triples of the file.

Return type ndarray , shape [n, 3]

load_from_rdf

ampligraph.datasets.**load_from_rdf**(*folder_name*, *file_name*, *format='nt'*, *data_home=None*)
Load an RDF file

Loads an RDF knowledge graph using rdflib APIs. The entire graph will be loaded in memory, and converted into an rdflib *Graph* object.

Parameters

- **folder_name** (*str*) – base folder within AMPLIGRAPH_DATA_HOME where the file is stored.
- **file_name** (*str*) – file name
- **format** (*str*) – The RDF serialization format (nt, ttl, rdf/xml - see rdflib documentation)

Returns **triples** – the actual triples of the file.

Return type ndarray , shape [n, 3]

3.3.2 Models

This module includes neural graph embedding models and support functions.

Knowledge graph embedding models are neural architectures that encode concepts from a knowledge graph (i.e. entities $\mathcal{E}$ and relation types $\mathcal{R}$) into low-dimensional, continuous vectors $\in \mathcal{R}^k$. Such *knowledge graph embeddings* have applications in knowledge graph completion, entity resolution, and link-based clustering, just to cite a few *[NMTG16]*.

Knowledge Graph Embedding Models

RandomBaseline([seed])	Random baseline
TransE([k, eta, epochs, batches_count, ...])	Translating Embeddings (TransE)
DistMult([k, eta, epochs, batches_count, ...])	The DistMult model
ComplEx([k, eta, epochs, batches_count, ...])	Complex embeddings (ComplEx)
HolE([k, eta, epochs, batches_count, seed, ...])	Holographic Embeddings

RandomBaseline

class ampligraph.latent_features.**RandomBaseline**(*seed=0*)

Random baseline

A dummy model that assigns a pseud-random score included between 0 and 1, and drawn from a uniform distribution.

A dummy random model is useful whenever you need to compare the performance of another model on a custom knowledge graph, and no other baseline is available.

Note: Although the model still requires invoking the *fit()* method, no training will be carried out.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import RandomBaseline
>>> model = RandomBaseline()
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
[0.5488135039273248, 0.7151893663724195]
```

Methods

__init__([seed])	Initialize RandomBaseline model
fit(X)	Train the random model
predict(X[, from_idx, get_ranks])	Assign random scores to candidate triples and then ranks them

__init__(*seed=0*)

Initialize RandomBaseline model

Parameters **seed** (*int*) – The seed used by the internal random numbers generator.

fit(*X*)

Train the random model

Parameters **X** (*ndarray, shape [n, 3]*) – The training triples

predict(*X*, *from_idx=False*, *get_ranks=False*)

Assign random scores to candidate triples and then ranks them

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.

- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **ranks** (*ndarray, shape [n]*) – Rank of the triple

TransE

class ampligraph.latent_features.**TransE**(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={'norm': 1, 'normalize_ent_emb': False}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Translating Embeddings (TransE)

The model as described in *[BUGD+13]*.

$$f_{TransE} = -||(\mathbf{e}_s + \mathbf{r}_p) - \mathbf{e}_o||_n$$

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import TransE
>>> model = TransE(batches_count=1, seed=555, epochs=20, k=10, loss='pairwise',
>>>                loss_params={'margin':5})
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
[-2.219729, -3.9848995]
>>> model.get_embeddings(['f','e'], type='entity')
array([[-0.65229136, -0.50060457,  1.2316223 ,  0.23738968,  0.29145557,
-0.20187911, -0.3053819 , -0.6947149 ,  0.9377473 ,  0.12985024],
[-1.1272118 ,  0.10723944,  0.79431695,  0.6795645 , -0.14428931,
-0.34959725, -0.60184777, -1.1885864 ,  1.0374763 , -0.36612505]],
dtype=float32)
```

Methods

`__init__`([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
`fit`(X[, early_stopping, early_stopping_params])	Train an Translating Embeddings model.
`get_embeddings`(entities[, type])	Get the embeddings of entities or relations.
`predict`(X[, from_idx, get_ranks])	Predict the score of triples using a trained embedding model.

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={'norm': 1, 'normalize_ent_emb': False}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Initialize an EmbeddingModel

Also creates a new Tensorflow session for training.

Parameters

- **k** (*int*) – Embedding space dimensionality
- **eta** (*int*) – The number of negatives that must be generated at runtime during training for each positive.
- **epochs** (*int*) – The iterations of the training loop.
- **batches_count** (*int*) – The number of batches in which the training set must be split during the training loop.
- **seed** (*int*) – The seed used by the internal random numbers generator.
- **embedding_model_params** (*dict*) – TransE-specific hyperparams:
  - **norm** - type of norm to be used in scoring function (1 or 2 norm - default:1)
  - **normalize_ent_emb** - Flag to indicate whether to normalize entity embeddings after each batch update (default:False)
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between `sgd`, `adagrad`, `adam`, `momentum`.
- **optimizer_params** (*dict*) – Parameters values specific to the optimizer.Currently supported:
  - **lr** - learning rate (used by all the optimizers)
  - **momentum** - learning momentum (used by momentum optimizer)
- **loss** (*string*) – The type of loss function to use during training.
  - `pairwise` the model will use pairwise margin-based loss function.
  - `nll` the model will use negative loss likelihood.
  - `absolute_margin` the model will use absolute margin likelihood.
  - `self_adversarial` the model will use adversarial sampling loss function.
- **loss_params** (*dict*) – Parameters dictionary specific to the loss.

  (Refer documentation of specific loss functions for more details)
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `LP` the model will use L1, L2 or L3 based on the value passed to param p.
  - `None` the model will not use any regularizer

- **regularizer_params** (*dict*) – Parameters dictionary specific to the regularizer.

  (Refer documentation of regularizer for more details)

- **model_checkpoint_path** (*string*) – Path to save the model.
- **verbose** (*bool*) – Verbose mode
- **kwargs** (*dict*) – Additional inputs, if any

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train an Translating Embeddings model.

The model is trained on a training set X using the training protocol described in *[TWR+16]*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping(default:False)
- **early_stopping_params** (*dictionary*) – Dictionary of parameters for early stopping. Following keys are supported:
  - **x_valid**: ndarray, shape [n, 3] : Validation set to be used for early stopping.
  - **criteria**: string : criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr` (default).
  - **x_filter**: ndarray, shape [n, 3] : Filter to be used (no filter by default).
  - **burn_in**: int : Number of epochs to pass before kicking in early stopping (default: 100).
  - **check_interval**: int : Early stopping interval after burn-in (default:10).
  - **stop_interval**: int : Stop if criteria is performing worse over n consecutive checks (default: 3).

get_embeddings (*entities*, *type='entity'*)
Get the embeddings of entities or relations.

Parameters

- **entities** (*array-like, dtype=int, shape=[n]*) – The entities (or relations) of interest. Element of the vector must be the original string literals, and not internal IDs.
- **type** (*string*) – If 'entity', will consider input as KG entities. If *relation*, they will be treated as KG predicates.

Returns **embeddings** – An array of k-dimensional embeddings.

Return type ndarray, shape [n, k]

predict (*X*, *from_idx=False*, *get_ranks=False*)
Predict the score of triples using a trained embedding model.

The function returns raw scores generated by the model. To obtain probability estimates, use a logistic sigmoid.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores_predict** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **rank** (*ndarray, shape [n]*) – Rank of the triple

DistMult

class ampligraph.latent_features.**DistMult**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={'normalize_ent_emb': False}, optimizer='adagrad', optimizer_params={'lr': 0.1}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, model_checkpoint_path='saved_model/', verbose=False, **kwargs*)

The DistMult model

The model as described in *[YYH+14]*.

$$f_{DistMult} = \langle \mathbf{r}_p, \mathbf{e}_s, \mathbf{e}_o \rangle$$

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import DistMult
>>> model = DistMult(batches_count=1, seed=555, epochs=20, k=10, loss='pairwise',␣
↪loss_params={'margin':5})
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
[3.29703, -3.543957]
>>> model.get_embeddings(['f','e'], type='entity')
array([[-0.7101061 , -0.35752687,  0.5337027 , -0.612499  , -0.34532365,
-0.7219143 , -0.07083285,  0.19323194,  1.0108972 ,  0.42850104],
[-1.2280471 , -0.22018537,  0.17179069,  0.757755  , -0.05845603,
 0.94373196, -0.14994079, -0.929564  ,  1.0907435 ,  0.20400602]],
dtype=float32)
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train an DistMult.
get_embeddings(entities[, type])	Get the embeddings of entities or relations.

Continued on next page

Table 6 – continued from previous page

predict(X[, from_idx, get_ranks])	Predict the score of triples using a trained embedding model.

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={'normalize_ent_emb': False}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Initialize an EmbeddingModel

Also creates a new Tensorflow session for training.

Parameters

- **k** (*int*) – Embedding space dimensionality
- **eta** (*int*) – The number of negatives that must be generated at runtime during training for each positive.
- **epochs** (*int*) – The iterations of the training loop.
- **batches_count** (*int*) – The number of batches in which the training set must be split during the training loop.
- **seed** (*int*) – The seed used by the internal random numbers generator.
- **embedding_model_params** (*dict*) – DistMult-specific hyperparams:
  - **normalize_ent_emb** - Flag to indicate whether to normalize entity embeddings after each batch update (default:False)
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between `sgd`, `adagrad`, `adam`, `momentum`.
- **optimizer_params** (*dict*) – Parameters values specific to the optimizer. Currently supported:
  - **lr** - learning rate (used by all the optimizers)
  - **momentum** - learning momentum (used by momentum optimizer)
- **loss** (*string*) – The type of loss function to use during training.
  - `pairwise` the model will use pairwise margin-based loss function.
  - `nll` the model will use negative loss likelihood.
  - `absolute_margin` the model will use absolute margin likelihood.
  - `self_adversarial` the model will use adversarial sampling loss function.
- **loss_params** (*dict*) – Parameters dictionary specific to the loss.

  (Refer documentation of specific loss functions for more details)
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `LP` the model will use L1, L2 or L3 based on the value passed to param p.
  - `None` the model will not use any regularizer
- **regularizer_params** (*dict*) – Parameters dictionary specific to the regularizer.

  (Refer documentation of regularizer for more details)
- **model_checkpoint_path** (*string*) – Path to save the model.

- **verbose** (*bool*) – Verbose mode
- **kwargs** (*dict*) – Additional inputs, if any

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train an DistMult.

The model is trained on a training set X using the training protocol described in *[TWR+16]*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping(default:False)
- **early_stopping_params** (*dictionary*) – Dictionary of parameters for early stopping. Following keys are supported:
  - **x_valid**: ndarray, shape [n, 3] : Validation set to be used for early stopping.
  - **criteria**: string : criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr` (default).
  - **x_filter**: ndarray, shape [n, 3] : Filter to be used (no filter by default).
  - **burn_in**: int : Number of epochs to pass before kicking in early stopping (default: 100).
  - **check_interval**: int : Early stopping interval after burn-in (default:10).
  - **stop_interval**: int : Stop if criteria is performing worse over n consecutive checks (default: 3).

get_embeddings (*entities*, *type='entity'*)
Get the embeddings of entities or relations.

Parameters

- **entities** (*array-like, dtype=int, shape=[n]*) – The entities (or relations) of interest. Element of the vector must be the original string literals, and not internal IDs.
- **type** (*string*) – If 'entity', will consider input as KG entities. If *relation*, they will be treated as KG predicates.

Returns **embeddings** – An array of k-dimensional embeddings.

Return type ndarray, shape [n, k]

predict (*X*, *from_idx=False*, *get_ranks=False*)
Predict the score of triples using a trained embedding model.

The function returns raw scores generated by the model. To obtain probability estimates, use a logistic sigmoid.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores_predict** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **rank** (*ndarray, shape [n]*) – Rank of the triple

ComplEx

class ampligraph.latent_features.**ComplEx**(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Complex embeddings (ComplEx)

The ComplEx model *[TWR+16]* is an extension of the *ampligraph.latent_features.DistMult* bilinear diagonal model . ComplEx scoring function is based on the trilinear Hermitian dot product in $\mathcal{C}$:

$$f_{ComplEx} = Re(\langle \mathbf{r}_p, \mathbf{e}_s, \overline{\mathbf{e}_o} \rangle)$$

Note that because embeddings are in $\mathcal{C}$, ComplEx uses twice as many parameters as its counterpart in $\mathcal{R}$ DistMult.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import ComplEx
>>>
>>> model = ComplEx(batches_count=1, seed=555, epochs=20, k=10,
>>>             loss='pairwise', loss_params={'margin':1},
>>>             regularizer='LP', regularizer_params={'lambda':0.1})
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
[0.96325016, -0.17629346]
>>> model.get_embeddings(['f','e'], type='entity')
array([[-0.11257   , -0.09226837,  0.2829331 , -0.02094189,  0.02826234,
-0.3068198 , -0.41022655, -0.23714773, -0.00084166,  0.22521858,
-0.48155236,  0.29627186,  0.29841757,  0.16540456,  0.45836073,
 0.14025007, -0.03458257, -0.03813137,  0.35438442, -0.4733188 ],
[ 0.06088537,  0.13615245, -0.20476362,  0.20391239,  0.22199424,
 0.5762486 , -0.01087974,  0.39070424, -0.1372974 ,  0.39998057,
-0.5944237 ,  0.506474  ,  0.1255992 , -0.06021457, -0.26678884,
-0.18713273,  0.36862013,  0.07165384, -0.00845572, -0.16494963]],
dtype=float32)
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel

Continued on next page

Table 7 – continued from previous page

fit(X[, early_stopping, early_stopping_params])	Train a ComplEx model.
get_embeddings(entities[, type])	Get the embeddings of entities or relations.
predict(X[, from_idx, get_ranks])	Predict the score of triples using a trained embedding model.

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Initialize an EmbeddingModel

Also creates a new Tensorflow session for training.

Parameters

- **k** (*int*) – Embedding space dimensionality
- **eta** (*int*) – The number of negatives that must be generated at runtime during training for each positive.
- **epochs** (*int*) – The iterations of the training loop.
- **batches_count** (*int*) – The number of batches in which the training set must be split during the training loop.
- **seed** (*int*) – The seed used by the internal random numbers generator.
- **embedding_model_params** (*dict*) – ComplEx-specific hyperparams: Currently ComplEx does not require any hyperparameters.
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between `sgd`, `adagrad`, `adam`, `momentum`.
- **optimizer_params** (*dict*) – Parameters values specific to the optimizer. Currently supported:
  - **lr** - learning rate (used by all the optimizers)
  - **momentum** - learning momentum (used by momentum optimizer)
- **loss** (*string*) – The type of loss function to use during training.
  - `pairwise` the model will use pairwise margin-based loss function.
  - `nll` the model will use negative loss likelihood.
  - `absolute_margin` the model will use absolute margin likelihood.
  - `self_adversarial` the model will use adversarial sampling loss function.
- **loss_params** (*dict*) – Parameters dictionary specific to the loss.

  (Refer documentation of specific loss functions for more details)
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `LP` the model will use L1, L2 or L3 based on the value passed to param p.
  - `None` the model will not use any regularizer
- **regularizer_params** (*dict*) – Parameters dictionary specific to the regularizer.

  (Refer documentation of regularizer for more details)
- **model_checkpoint_path** (*string*) – Path to save the model.

- **verbose** (*bool*) – Verbose mode
- **kwargs** (*dict*) – Additional inputs, if any

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train a ComplEx model.

The model is trained on a training set X using the training protocol described in *[TWR+16]*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping(default:False)
- **early_stopping_params** (*dictionary*) – Dictionary of parameters for early stopping. Following keys are supported:
  - **x_valid**: ndarray, shape [n, 3] : Validation set to be used for early stopping.
  - **criteria**: string : criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr` (default).
  - **x_filter**: ndarray, shape [n, 3] : Filter to be used (no filter by default).
  - **burn_in**: int : Number of epochs to pass before kicking in early stopping (default: 100).
  - **check_interval**: int : Early stopping interval after burn-in (default:10).
  - **stop_interval**: int : Stop if criteria is performing worse over n consecutive checks (default: 3).

get_embeddings (*entities*, *type='entity'*)
Get the embeddings of entities or relations.

Parameters

- **entities** (*array-like, dtype=int, shape=[n]*) – The entities (or relations) of interest. Element of the vector must be the original string literals, and not internal IDs.
- **type** (*string*) – If 'entity', will consider input as KG entities. If *relation*, they will be treated as KG predicates.

Returns **embeddings** – An array of k-dimensional embeddings.

Return type ndarray, shape [n, k]

predict (*X*, *from_idx=False*, *get_ranks=False*)
Predict the score of triples using a trained embedding model.

The function returns raw scores generated by the model. To obtain probability estimates, use a logistic sigmoid.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores_predict** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **rank** (*ndarray, shape [n]*) – Rank of the triple

HolE

class ampligraph.latent_features.**HolE**(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Holographic Embeddings

The HolE model *[NRP+16]* as re-defined by *[HS17]*.

Hayashi et al. *[HS17]* redifine the original HolE scoring function as:

$$f_{HolE} = 2/n * f_{ComplEx}$$

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import HolE
>>> model = HolE(batches_count=1, seed=555, epochs=20, k=10,
>>>              loss='pairwise', loss_params={'margin':1},
>>>              regularizer='LP', regularizer_params={'lambda':0.1})
>>>
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]), get_ranks=True)
[0.3046168, -0.0379385]
>>> model.get_embeddings(['f','e'], type='entity')
array([[-0.2704807 , -0.05434025,  0.13363852,  0.04879733,  0.00184516,
-0.1149573 , -0.1177371 , -0.20798951,  0.01935115,  0.13033926,
-0.81528974,  0.22864424,  0.2045117 ,  0.1145515 ,  0.248952  ,
 0.03513691, -0.08550065, -0.06037813,  0.23231442, -0.39326245],
[ 0.204738  ,  0.10758886, -0.11931524,  0.14881928,  0.0929039 ,
 0.25577265,  0.05722341,  0.2549932 , -0.16462566,  0.43789816,
-0.91011846,  0.3533137 ,  0.1144442 ,  0.00359709, -0.09599967,
-0.03151475,  0.14198618,  0.16138661,  0.07511608, -0.2465882 ]],
dtype=float32)
```

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train a HolE model.
get_embeddings(entities[, type])	Get the embeddings of entities or relations.

Continued on next page

Table 8 – continued from previous page

predict(X[, from_idx, get_ranks])	Predict the score of triples using a trained embedding model.

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Initialize an EmbeddingModel

Also creates a new Tensorflow session for training.

Parameters

- **k** (*int*) – Embedding space dimensionality
- **eta** (*int*) – The number of negatives that must be generated at runtime during training for each positive.
- **epochs** (*int*) – The iterations of the training loop.
- **batches_count** (*int*) – The number of batches in which the training set must be split during the training loop.
- **seed** (*int*) – The seed used by the internal random numbers generator.
- **embedding_model_params** (*dict*) – HolE-specific hyperparams: Currently HolE does not require any hyperparameters.
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between `sgd`, `adagrad`, `adam`, `momentum`.
- **optimizer_params** (*dict*) – Parameters values specific to the optimizer. Currently supported:
  - **lr** - learning rate (used by all the optimizers)
  - **momentum** - learning momentum (used by momentum optimizer)
- **loss** (*string*) – The type of loss function to use during training.
  - `pairwise` the model will use pairwise margin-based loss function.
  - `nll` the model will use negative loss likelihood.
  - `absolute_margin` the model will use absolute margin likelihood.
  - `self_adversarial` the model will use adversarial sampling loss function.
- **loss_params** (*dict*) – Parameters dictionary specific to the loss.

  (Refer documentation of specific loss functions for more details)
- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `LP` the model will use L1, L2 or L3 based on the value passed to param p.
  - `None` the model will not use any regularizer
- **regularizer_params** (*dict*) – Parameters dictionary specific to the regularizer.

  (Refer documentation of regularizer for more details)
- **model_checkpoint_path** (*string*) – Path to save the model.
- **verbose** (*bool*) – Verbose mode

- **kwargs** (*dict*) – Additional inputs, if any

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train a HolE model.

The model is trained on a training set X using the training protocol described in *[NRP+16]*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping(default:False)
- **early_stopping_params** (*dictionary*) – Dictionary of parameters for early stopping. Following keys are supported:
  - **x_valid**: ndarray, shape [n, 3] : Validation set to be used for early stopping.
  - **criteria**: string : criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr` (default).
  - **x_filter**: ndarray, shape [n, 3] : Filter to be used (no filter by default).
  - **burn_in**: int : Number of epochs to pass before kicking in early stopping (default: 100).
  - **check_interval**: int : Early stopping interval after burn-in (default:10).
  - **stop_interval**: int : Stop if criteria is performing worse over n consecutive checks (default: 3).

get_embeddings (*entities*, *type='entity'*)
Get the embeddings of entities or relations.

Parameters

- **entities** (*array-like, dtype=int, shape=[n]*) – The entities (or relations) of interest. Element of the vector must be the original string literals, and not internal IDs.
- **type** (*string*) – If 'entity', will consider input as KG entities. If *relation*, they will be treated as KG predicates.

Returns **embeddings** – An array of k-dimensional embeddings.

Return type ndarray, shape [n, k]

predict (*X*, *from_idx=False*, *get_ranks=False*)
Predict the score of triples using a trained embedding model.

The function returns raw scores generated by the model. To obtain probability estimates, use a logistic sigmoid.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores_predict** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **rank** (*ndarray, shape [n]*) – Rank of the triple

Anatomy of a Model

Knowledge graph embeddings are learned by training a neural architecture over a graph. Although such architectures vary, the training phase always consists in minimizing a *loss function* $\mathcal{L}$ that includes a *scoring function* $f_m(t)$, i.e. a model-specific function that assigns a score to a triple $t = (sub, pred, obj)$.

AmpliGraph models include the following components:

- *Scoring function* $f(t)$
- *Loss function* $\mathcal{L}$
- *Optimization algorithm*
- *Negatives generation strategy*

AmpliGraph comes with a number of such components. They can be used in any combination to come up with a model that performs sufficiently well for the dataset of choice.

AmpliGraph features a number of abstract classes that can be extended to design new models:

EmbeddingModel([k, eta, epochs, ...])	Abstract class for embedding models
Loss(eta, hyperparam_dict[, verbose])	Abstract class for loss function.
Regularizer(hyperparam_dict[, verbose])	Abstract class for Regularizer.

EmbeddingModel

class ampligraph.latent_features.**EmbeddingModel**(*k=100, eta=2, epochs=100, batches_count=100, seed=0, embedding_model_params={}, optimizer='adagrad', optimizer_params={'lr': 0.1}, loss='nll', loss_params={}, regularizer=None, regularizer_params={}, model_checkpoint_path='saved_model/', verbose=False, **kwargs*)

Abstract class for embedding models

AmpliGraph neural knowledge graph embeddings models extend this class and its functionalities.

Methods

__init__([k, eta, epochs, batches_count, ...])	Initialize an EmbeddingModel
fit(X[, early_stopping, early_stopping_params])	Train an EmbeddingModel (with optional early stopping).
get_embeddings(entities[, type])	Get the embeddings of entities or relations.
predict(X[, from_idx, get_ranks])	Predict the score of triples using a trained embedding model.
_fn(e_s, e_p, e_o)	The scoring function of the model.
_initialize_parameters()	Initialize parameters of the model.
_get_model_loss(scores_pos, scores_neg)	Get the current batch loss including loss due to regularization.
get_embedding_model_params(output_dict)	save the model parameters in the dictionary.

Continued on next page

Table 10 – continued from previous page

restore_model_params(in_dict)	Load the model parameters from the input dictionary.
_save_trained_params()	After model fitting, save all the trained parameters in trained_model_params in some order.
_load_model_from_trained_params()	Load the model from trained params.
_initialize_early_stopping()	Initializes and creates evaluation graph for early stopping
_perform_early_stopping_test(epoch)	perform regular validation checks and stop early if the criteria is acheived :param epoch: current training epoch :type epoch: int
configure_evaluation_protocol([config])	Set the configuration for evaluation
set_filter_for_eval(x_filter)	Set the filter to be used during evaluation (filtered_corruption = corruptions - filter).
_initialize_eval_graph()	Initialize the evaluation graph.
end_evaluation()	End the evaluation and close the Tensorflow session.

__init__(*k=100*, *eta=2*, *epochs=100*, *batches_count=100*, *seed=0*, *embedding_model_params={}*, *optimizer='adagrad'*, *optimizer_params={'lr': 0.1}*, *loss='nll'*, *loss_params={}*, *regularizer=None*, *regularizer_params={}*, *model_checkpoint_path='saved_model/'*, *verbose=False*, ***kwargs*)

Initialize an EmbeddingModel

Also creates a new Tensorflow session for training.

Parameters

- **k** (*int*) – Embedding space dimensionality
- **eta** (*int*) – The number of negatives that must be generated at runtime during training for each positive.
- **epochs** (*int*) – The iterations of the training loop.
- **batches_count** (*int*) – The number of batches in which the training set must be split during the training loop.
- **seed** (*int*) – The seed used by the internal random numbers generator.
- **embedding_model_params** (*dict*) – Parameter values of embedding model specific hyperparams

  (Refer documentation of specific embedding models for more details)
- **optimizer** (*string*) – The optimizer used to minimize the loss function. Choose between `sgd`, `adagrad`, `adam`, `momentum`.
- **optimizer_params** (*dict*) – Parameters values specific to the optimizer.Currently supported:
  - **lr** - learning rate (used by all the optimizers)
  - **momentum** - learning momentum (used by momentum optimizer)
- **loss** (*string*) – The type of loss function to use during training.
  - `pairwise` the model will use pairwise margin-based loss function.
  - `nll` the model will use negative loss likelihood.
  - `absolute_margin` the model will use absolute margin likelihood.
  - `self_adversarial` the model will use adversarial sampling loss function.

- **loss_params** (*dict*) – Parameters dictionary specific to the loss.

  (Refer documentation of specific loss functions for more details)

- **regularizer** (*string*) – The regularization strategy to use with the loss function.
  - `LP` the model will use L1, L2 or L3 based on the value passed to param p.
  - `None` the model will not use any regularizer
- **regularizer_params** (*dict*) – Parameters dictionary specific to the regularizer.

  (Refer documentation of regularizer for more details)

- **model_checkpoint_path** (*string*) – Path to save the model.
- **verbose** (*bool*) – Verbose mode
- **kwargs** (*dict*) – Additional inputs, if any

fit (*X*, *early_stopping=False*, *early_stopping_params={}*)
Train an EmbeddingModel (with optional early stopping).

The model is trained on a training set X using the training protocol described in *[TWR+16]*.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The training triples
- **early_stopping** (*bool*) – Flag to enable early stopping (default:False)
- **early_stopping_params** (*dictionary*) – Dictionary of parameters for early stopping. Following keys are supported:
  - **x_valid**: ndarray, shape [n, 3] : Validation set to be used for early stopping.
  - **criteria**: string : criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr` (default).
  - **x_filter**: ndarray, shape [n, 3] : Filter to be used (no filter by default).
  - **burn_in**: int : Number of epochs to pass before kicking in early stopping (default: 100).
  - **check_interval**: int : Early stopping interval after burn-in (default:10).
  - **stop_interval**: int : Stop if criteria is performing worse over n consecutive checks (default: 3).

get_embeddings (*entities*, *type='entity'*)
Get the embeddings of entities or relations.

Parameters

- **entities** (*array-like, dtype=int, shape=[n]*) – The entities (or relations) of interest. Element of the vector must be the original string literals, and not internal IDs.
- **type** (*string*) – If 'entity', will consider input as KG entities. If *relation*, they will be treated as KG predicates.

Returns **embeddings** – An array of k-dimensional embeddings.

Return type ndarray, shape [n, k]

predict (*X*, *from_idx=False*, *get_ranks=False*)
Predict the score of triples using a trained embedding model.

The function returns raw scores generated by the model. To obtain probability estimates, use a logistic sigmoid function.

Parameters

- **X** (*ndarray, shape [n, 3]*) – The triples to score.
- **from_idx** (*bool*) – If True, will skip conversion to internal IDs. (default: False).
- **get_ranks** (*bool*) – Flag to compute ranks by scoring against corruptions (default: False).

Returns

- **scores** (*ndarray, shape [n]*) – The predicted scores for input triples X.
- **ranks** (*ndarray, shape [n]*) – Rank of the triple

_fn(*e_s*, *e_p*, *e_o*)

The scoring function of the model.

Assigns a score to a list of triples, with a model-specific strategy. Triples are passed as lists of subject, predicate, object embeddings. This function must be overridden by every model to return corresponding score.

Parameters

- **e_s** (*Tensor, shape [n]*) – The embeddings of a list of subjects.
- **e_p** (*Tensor, shape [n]*) – The embeddings of a list of predicates.
- **e_o** (*Tensor, shape [n]*) – The embeddings of a list of objects.

Returns **score** – The operation corresponding to the scoring function.

Return type TensorFlow operation

_initialize_parameters()

Initialize parameters of the model.

This function creates and initializes entity and relation embeddings (with size k). Overload this function if the parameters needs to be initialized differently.

_get_model_loss(*scores_pos*, *scores_neg*)

Get the current batch loss including loss due to regularization. This function must be overridden if the model uses combination of different losses(eg: VAE)

Parameters

- **scores_pos** (*tf.Tensor*) – A tensor of scores assigned to positive statements.
- **scores_neg** (*tf.Tensor*) – A tensor of scores assigned to negative statements.

Returns **loss** – The loss value that must be minimized.

Return type tf.Tensor

get_embedding_model_params(*output_dict*)

save the model parameters in the dictionary.

Parameters **output_dict** (*dictionary*) – Dictionary of saved params. It's the duty of the model to save all the variables correctly, so that it can be used for restoring later.

restore_model_params(*in_dict*)

Load the model parameters from the input dictionary.

Parameters **in_dict** (*dictionary*) – Dictionary of saved params. It's the duty of the model to load the variables correctly

_save_trained_params()
After model fitting, save all the trained parameters in trained_model_params in some order. The order would be useful for loading the model. This method must be overridden if the model has any other parameters (apart from entity-relation embeddings)

_load_model_from_trained_params()
Load the model from trained params. While restoring make sure that the order of loaded parameters match the saved order. It's the duty of the embedding model to load the variables correctly. This method must be overridden if the model has any other parameters (apart from entity-relation embeddings)

_initialize_early_stopping()
Initializes and creates evaluation graph for early stopping

_perform_early_stopping_test(*epoch*)
perform regular validation checks and stop early if the criteria is acheived :param epoch: current training epoch :type epoch: int

Returns **stopped** – Flag to indicate if the early stopping criteria is acheived

Return type bool

configure_evaluation_protocol(*config={'corrupt_side': 's+o', 'corruption_entities': None}*)
Set the configuration for evaluation

Parameters **config** (*dictionary*) – Dictionary of parameters for evaluation configuration. Can contain following keys:

- **corruption_entities**: Entities to be used for corruptions. If None, it uses all entities (default: None)
- **corrupt_side**: Specifies which side to corrupt. `s`, `o`, `s+o` (default)

set_filter_for_eval(*x_filter*)
Set the filter to be used during evaluation (filtered_corruption = corruptions - filter).

We would be using a prime number based assignment and product for do the filtering. We associate a unique prime number for subject entities, object entities and to relations. Product of three prime numbers is divisible only by those three prime numbers. So we generate this product for the filter triples and store it in a hash map. When corruptions are generated for a triple during evaluation, we follow a similar approach and look up the product of corruption in the above hash table. If the corrupted triple is present in the hashmap, it means that it was present in the filter list.

Parameters **x_filter** (*ndarray, shape [n, 3]*) – Filter triples. If the generated corruptions are present in this, they will be removed.

_initialize_eval_graph()
Initialize the evaluation graph.

Use prime number based filtering strategy (refer set_filter_for_eval()), if the filter is set

end_evaluation()
End the evaluation and close the Tensorflow session.

Loss

class ampligraph.latent_features.**Loss**(*eta*, *hyperparam_dict*, *verbose=False*)
Abstract class for loss function.

Methods

__init__(eta, hyperparam_dict[, verbose])	Initialize Loss.
get_state(param_name)	Get the state value.
_init_hyperparams(hyperparam_dict)	Initializes the hyperparameters needed by the algorithm.
_inputs_check(scores_pos, scores_neg)	Creates any dependencies that need to be checked before performing loss computations
apply(scores_pos, scores_neg)	Interface to external world.
_apply(scores_pos, scores_neg)	Apply the loss function.

__init__ (*eta*, *hyperparam_dict*, *verbose=False*)
Initialize Loss.

Parameters

- **eta** (*int*) – number of negatives
- **hyperparam_dict** (*dict*) – dictionary of hyperparams. (Keys are described in the hyperparameters section)

get_state (*param_name*)
Get the state value.

Parameters **param_name** (*string*) – name of the state for which one wants to query the value

Returns the value of the corresponding state

Return type param_value

_init_hyperparams (*hyperparam_dict*)
Initializes the hyperparameters needed by the algorithm.

Parameters **hyperparam_dict** (*dictionary*) – Consists of key value pairs. The Loss will check the keys to get the corresponding params

_inputs_check (*scores_pos*, *scores_neg*)
Creates any dependencies that need to be checked before performing loss computations

Parameters

- **scores_pos** (*tf.Tensor*) – A tensor of scores assigned to positive statements.
- **scores_neg** (*tf.Tensor*) – A tensor of scores assigned to negative statements.

apply (*scores_pos*, *scores_neg*)
Interface to external world. This function does the input checks, preprocesses input and finally applies loss function.

Parameters

- **scores_pos** (*tf.Tensor*) – A tensor of scores assigned to positive statements.
- **scores_neg** (*tf.Tensor*) – A tensor of scores assigned to negative statements.

Returns **loss** – The loss value that must be minimized.

Return type tf.Tensor

_apply(*scores_pos*, *scores_neg*)
Apply the loss function. Every inherited class must implement this function. (All the TF code must go in this function.)

Parameters

- **scores_pos** (*tf.Tensor*) – A tensor of scores assigned to positive statements.
- **scores_neg** (*tf.Tensor*) – A tensor of scores assigned to negative statements.

Returns **loss** – The loss value that must be minimized.

Return type tf.Tensor

Regularizer

class ampligraph.latent_features.**Regularizer**(*hyperparam_dict*, *verbose=False*)
Abstract class for Regularizer.

Methods

__init__(hyperparam_dict[, verbose])	Initialize the regularizer.
get_state(param_name)	Get the state value.
_init_hyperparams(hyperparam_dict)	Initializes the hyperparameters needed by the algorithm.
apply(trainable_params)	Interface to external world.
_apply(trainable_params)	Apply the regularization function.

__init__(*hyperparam_dict*, *verbose=False*)
Initialize the regularizer.

Parameters **hyperparam_dict** (*dict*) – dictionary of hyperparams (Keys are described in the hyperparameters section)

get_state(*param_name*)
Get the state value.

Parameters **param_name** (*string*) – name of the state for which one wants to query the value

Returns the value of the corresponding state

Return type param_value

_init_hyperparams(*hyperparam_dict*)
Initializes the hyperparameters needed by the algorithm.

Parameters **hyperparam_dict** (*dictionary*) – Consists of key value pairs. The regularizer will check the keys to get the corresponding params

apply(*trainable_params*)
Interface to external world. This function performs input checks, input pre-processing, and and applies the loss function.

Parameters **trainable_params** (*list, shape [n]*) – List of trainable params that should be requlatized

Returns **loss** – Regularization Loss

Return type tf.Tensor

_apply(*trainable_params*)

Apply the regularization function. Every inherited class must implement this function.

(All the TF code must go in this function.)

Parameters **trainable_params** (*list, shape [n]*) – List of trainable params that should be requlatized

Returns **loss** – Regularization Loss

Return type tf.Tensor

Scoring functions

Existing models propose scoring functions that combine the embeddings $\mathbf{e}_{sub}, \mathbf{e}_{pred}, \mathbf{e}_{obj} \in \mathcal{R}^k$ of the subject, predicate, and object of a triple $t = (sub, pred, obj)$ according to different intuitions:

- *TransE [BUGD+13]* relies on distances. The scoring function computes a similarity between the embedding of the subject translated by the embedding of the predicate and the embedding of the object, using the L_1 or L_2 norm $||\cdot||$:

$$f_{TransE} = -||\mathbf{e}_{sub} + \mathbf{e}_{pred} - \mathbf{e}_{obj}||_n$$

- *DistMult [YYH+14]* uses the trilinear dot product:

$$f_{DistMult} = \langle \mathbf{r}_p, \mathbf{e}_s, \mathbf{e}_o \rangle$$

- *ComplEx [TWR+16]* extends DistMult with the Hermitian dot product:

$$f_{ComplEx} = Re(\langle \mathbf{r}_p, \mathbf{e}_s, \overline{\mathbf{e}_o} \rangle)$$

- *HolE [NRP+16]* uses circular correlation.

$$f_{HolE} = \mathbf{w}_r \cdot (\mathbf{e}_s \star \mathbf{e}_o) = \frac{1}{k}\mathcal{F}(\mathbf{w}_r) \cdot (\overline{\mathcal{F}(\mathbf{e}_s)} \odot \mathcal{F}(\mathbf{e}_o))$$

Other models such ConvE include convolutional layers *[DMSR18]* (will be available in AmpliGraph future releases).

Loss Functions

AmpliGraph includes a number of loss functions commonly used in literature. Each function can be used with any of the implemented models. Loss functions are passed to models as hyperparameter, and they can be thus used *during model selection*.

PairwiseLoss(eta[, hyperparam_dict, verbose])	Pairwise, max-margin loss.
NLLLoss(eta[, hyperparam_dict, verbose])	Negative log-likelihood loss.
AbsoluteMarginLoss(eta[, hyperparam_dict, ...])	Absolute margin , max-margin loss.
SelfAdversarialLoss(eta[, hyperparam_dict, ...])	Self adversarial sampling loss.

PairwiseLoss

class ampligraph.latent_features.**PairwiseLoss**(*eta*, *hyperparam_dict={'margin': 1}*, *verbose=False*)

Pairwise, max-margin loss.

Introduced in *[BUGD+13]*.

$$\mathcal{L}(\Theta) = \sum_{t^+ \in \mathcal{G}} \sum_{t^- \in \mathcal{C}} max(0, [\gamma + f_{model}(t^-; \Theta) - f_{model}(t^+; \Theta)])$$

where γ is the margin, $\mathcal{G}$ is the set of positives, $\mathcal{C}$ is the set of corruptions, $f_{model}(t; \Theta)$ is the model-specific scoring function.

Methods

`__init__`(eta[, hyperparam_dict, verbose])	Initialize Loss.

__init__ (*eta*, *hyperparam_dict={'margin': 1}*, *verbose=False*)
Initialize Loss.

Parameters

- **eta** (*int*) – number of negatives
- **hyperparam_dict** (*dict*) – dictionary of hyperparams.
  - **margin**: float. Margin to be used in pairwise loss computation (default:1)

NLLLoss

class ampligraph.latent_features.**NLLLoss** (*eta*, *hyperparam_dict={}*, *verbose=False*)
Negative log-likelihood loss.

As described in *[TWR+16]*.

$$\mathcal{L}(\Theta) = \sum_{t \in \mathcal{G} \cup \mathcal{C}} log(1 + exp(-y f_{model}(t; \Theta)))$$

where y is the label of the statement :math:‘ in [-1, 1]‘, $\mathcal{G}$ is the set of positives, $\mathcal{C}$ is the set of corruptions, $f_{model}(t; \Theta)$ is the model-specific scoring function.

Methods

`__init__`(eta[, hyperparam_dict, verbose])	Initialize Loss.

__init__ (*eta*, *hyperparam_dict={}*, *verbose=False*)
Initialize Loss.

Parameters

- **eta** (*int*) – number of negatives
- **hyperparam_dict** (*dict*) – dictionary of hyperparams. No hyperparameters are required for this loss.

AbsoluteMarginLoss

class ampligraph.latent_features.**AbsoluteMarginLoss**(*eta*, *hyperparam_dict={'margin': 1}*, *verbose=False*)

Absolute margin , max-margin loss.

Introduced in *[HOSM17]*.

$$\mathcal{L}(\Theta) = \sum_{t^+ \in \mathcal{G}} \sum_{t^- \in \mathcal{C}} f_{model}(t^-; \Theta) - max(0, [\gamma - f_{model}(t^+; \Theta)])$$

where γ is the margin, $\mathcal{G}$ is the set of positives, $\mathcal{C}$ is the set of corruptions, $f_{model}(t; \Theta)$ is the model-specific scoring function.

Methods

__init__(eta[, hyperparam_dict, verbose])	Initialize Loss

__init__(*eta*, *hyperparam_dict={'margin': 1}*, *verbose=False*)
Initialize Loss

Parameters

- **eta** (*int*) – number of negatives
- **hyperparam_dict** (*dict*) – dictionary of hyperparams.
  - **margin**: float. Margin to be used in loss computation (default:1)

SelfAdversarialLoss

class ampligraph.latent_features.**SelfAdversarialLoss**(*eta*, *hyperparam_dict={'alpha': 0.5, 'margin': 3}*, *verbose=False*)

Self adversarial sampling loss.

Introduced in *[SDNT19]*.

$$\mathcal{L} = -log\sigma(\gamma - d_r(h,t)) - \sum_{i=1}^{n} p(h_i^{'}, r, t_i^{'})\ log\ \sigma(d_r(h_i^{'}, t_i^{'}) - \gamma)$$

where γ is the margin, and $p(h_i^{'}, r, t_i^{'})$ is the sampling proportion

Methods

__init__(eta[, hyperparam_dict, verbose])	Initialize Loss

__init__(*eta*, *hyperparam_dict={'alpha': 0.5, 'margin': 3}*, *verbose=False*)
Initialize Loss

Parameters

- **eta** (*int*) – number of negatives

- **hyperparam_dict** (*dict*) – dictionary of hyperparams.
  - **margin**: float. Margin to be used in adversarial loss computation (default:3)
  - **alpha** : float. Temperature of sampling (default:0.5)

Regularizers

AmpliGraph includes a number of regularizers that can be used with the *loss function*. `LPRegularizer` supports L1, L2, and L3.

`LPRegularizer`([hyperparam_dict, verbose])	Performs LP regularization

LPRegularizer

class ampligraph.latent_features.**LPRegularizer**(*hyperparam_dict={'lambda': 1e-05, 'p': 2}*, *verbose=False*)

Performs LP regularization

$$\mathcal{L}(Reg) = \sum_{i=1}^{n} \lambda_i * \mid w_i \mid_p$$

where n is the number of model parameters, p is the p-norm and λ is the regularization weight.

p==1 does L1 regularization; p==2 does L2 regularization and so on.

Methods

`__init__`([hyperparam_dict, verbose])	Initializes the hyperparameters needed by the algorithm.

__init__(*hyperparam_dict={'lambda': 1e-05, 'p': 2}*, *verbose=False*)

Initializes the hyperparameters needed by the algorithm.

Parameters **hyperparam_dict** (*dictionary*) – Consists of key value pairs. The regularizer will check the keys to get the corresponding params:

- **lambda**: float. Weight of regularization loss for each parameter (default: 1e-5)
- **p**: int: norm (default: 2)

Optimizers

The goal of the optimization procedure is learning optimal embeddings, such that the scoring function is able to assign high scores to positive statements and low scores to statements unlikely to be true.

We support SGD-based optimizers provided by TensorFlow, by setting the `optimizer` argument in a model initializer. Best results are currently obtained with Adam.

Utils Functions

Models can be saved and restored from disk. This is useful to avoid re-training a model.

save_model(model, loc)	Save a trained model to disk.
restore_model(loc)	Restore a saved model from disk.

save_model

ampligraph.latent_features.**save_model**(*model*, *loc*)
Save a trained model to disk.

Examples

```
>>> import numpy as np
>>> from ampligraph.latent_features import ComplEx, save_model, restore_model
>>> X = np.array([['a', 'y', 'b'],
>>>               ['b', 'y', 'a'],
>>>               ['a', 'y', 'c'],
>>>               ['c', 'y', 'a'],
>>>               ['a', 'y', 'd'],
>>>               ['c', 'y', 'd'],
>>>               ['b', 'y', 'c'],
>>>               ['f', 'y', 'e']])
>>> model.fit(X)
>>> y_pred_before = model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
>>> EXAMPLE_LOC = 'saved_models'
>>> save_model(model, EXAMPLE_LOC)
>>> print(y_pred_before)
[1.261404, -1.324778]
```

Parameters

- **model** (*trained model*) – A trained neural knowledge graph embedding model, the model must be an instance of TransE, DistMult or ComplEx classes.
- **loc** (*string*) – Directory into which user expects to save the model

restore_model

ampligraph.latent_features.**restore_model**(*loc*)
Restore a saved model from disk.

Examples

```
>>> from ampligraph.latent_features import restore_model
>>> import numpy as np
>>> EXAMPLE_LOC = 'saved_models' # Assuming that the model is present at this␣
↪location
>>> restored_model = restore_model(EXAMPLE_LOC)
>>> y_pred_after = restored_model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd
↪']]))
>>> print(y_pred_after)
[1.261404, -1.324778]
```

Parameters **loc** (*string*) – Directory containing the saved model

Returns **model** – a neural knowledge graph embedding model

Return type trained model

3.3.3 Evaluation

The module incldues performance metrics for neural graph embeddings models, along with model selection routines, negatives generation, and an implementation of the learning-to-rank-based evaluation protocol used in literature.

Metrics

Learning-to-rank metrics to evaluate the performance of neural graph embedding models.

rank_score(y_true, y_pred[, pos_lab])	Rank of a triple
mrr_score(ranks)	Mean Reciprocal Rank (MRR)
mr_score(ranks)	Mean Rank (MR)
hits_at_n_score(ranks, n)	Hits@N

rank_score

ampligraph.evaluation.**rank_score**(*y_true*, *y_pred*, *pos_lab=1*)

Rank of a triple

The rank of a positive element against a list of negatives.

$$rank_{(s,p,o)_i}$$

Parameters

- **y_true** (*ndarray, shape [n]*) – An array of binary labels. The array only contains one positive.
- **y_pred** (*ndarray, shape [n]*) – An array of scores, for the positive element and the n-1 negatives.
- **pos_lab** (*int*) – The value of the positive label (default = 1)

Returns **rank** – The rank of the positive element against the negatives.

Return type int

Examples

```
>>> import numpy as np
>>> from ampligraph.evaluation.metrics import rank_score
>>> y_pred = np.array([.434, .65, .21, .84])
>>> y_true = np.array([0, 0, 1, 0])
>>> rank_score(y_true, y_pred)
4
```

mrr_score

ampligraph.evaluation.**mrr_score**(*ranks*)

Mean Reciprocal Rank (MRR)

The function computes the mean of the reciprocal of elements of a vector of rankings `ranks`.

It is used in conjunction with the learning to rank evaluation protocol of `evaluation.evaluate_performance`.

It is formally defined as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_{(s,p,o)_i}}$$

where Q is a set of triples and (s, p, o) is a triple $\in Q$.

Note: This metric is similar to mean rank (MR) `metrics.mr`. Instead of averaging ranks it averages their reciprocals. This is done to obtain a metric which is more robust to outliers.

Consider the following example. Each of the two positive triples identified by $\star$ are ranked against four corruptions each. When scored by an embedding model, the first triple ranks 2nd, and the other triple ranks first. The resulting MRR is:

```
s        p          o             score    rank
Jack   born_in    Ireland         0.789       1
Jack   born_in    Italy           0.753       2   *
Jack   born_in    Germany         0.695       3
Jack   born_in    China           0.456       4
Jack   born_in    Thomas          0.234       5

s        p          o             score    rank
Jack   friend_with    Thomas      0.901       1   *
Jack   friend_with    China       0.345       2
Jack   friend_with    Italy       0.293       3
Jack   friend_with    Ireland     0.201       4
Jack   friend_with    Germany     0.156       5

MRR=0.75
```

Parameters **ranks** (*ndarray, shape [n]*) – Input ranks of n positive statements.

Returns **hits_n_score** – The MRR score

Return type float

Examples

```
>>> import numpy as np
>>> from ampligraph.evaluation.metrics import mrr_score
>>> rankings = np.array([1, 12, 6, 2])
>>> mrr_score(rankings)
0.4375
```

mr_score

ampligraph.evaluation.**mr_score**(*ranks*)

Mean Rank (MR)

The function computes the mean of of a vector of rankings `ranks`.

It is used in conjunction with the learning to rank evaluation protocol of `evaluation.evaluate_performance`.

It is formally defined as follows:

$$MR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} rank_{(s,p,o)_i}$$

where Q is a set of triples and (s, p, o) is a triple $\in Q$.

Note: This metric is not robust to outliers. It is usually used in conjunction with MRR `metrics.mrr`.

Consider the following example. Each of the two positive triples identified by $\star$ are ranked against four corruptions each. When scored by an embedding model, the first triple ranks 2nd, and the other triple ranks first. The resulting MR is:

```
s        p          o            score    rank
Jack   born_in    Ireland         0.789       1
Jack   born_in    Italy           0.753       2   *
Jack   born_in    Germany         0.695       3
Jack   born_in    China           0.456       4
Jack   born_in    Thomas          0.234       5

s        p          o            score    rank
Jack   friend_with    Thomas      0.901       1   *
Jack   friend_with    China       0.345       2
Jack   friend_with    Italy       0.293       3
Jack   friend_with    Ireland     0.201       4
Jack   friend_with    Germany     0.156       5

MR=1.5
```

Examples

```
>>> from ampligraph.evaluation import mr_score
>>> ranks= [5, 3, 4, 10, 1]
>>> mr_score(ranks)
4.6
```

hits_at_n_score

ampligraph.evaluation.**hits_at_n_score**(*ranks*, *n*)

Hits@N

The function computes how many elements of a vector of rankings `ranks` make it to the top `n` positions.

It is used in conjunction with the learning to rank evaluation protocol of `evaluation.evaluate_performance`.

It is formally defined as follows:

$$Hits@N = \sum_{i=1}^{|Q|} 1 if rank_{(s,p,o)_i} \leq N$$

where Q is a set of triples and (s, p, o) is a triple $\in Q$.

Consider the following example. Each of the two positive triples identified by $\star$ are ranked against four corruptions each. When scored by an embedding model, the first triple ranks 2nd, and the other triple ranks first. Hits@1 and Hits@3 are:

```
s        p          o            score   rank
Jack   born_in   Ireland         0.789      1
Jack   born_in   Italy           0.753      2  *
Jack   born_in   Germany         0.695      3
Jack   born_in   China           0.456      4
Jack   born_in   Thomas          0.234      5

s        p          o            score   rank
Jack   friend_with   Thomas      0.901      1  *
Jack   friend_with   China       0.345      2
Jack   friend_with   Italy       0.293      3
Jack   friend_with   Ireland     0.201      4
Jack   friend_with   Germany     0.156      5

Hits@3=1.0
Hits@1=0.5
```

Parameters

- **ranks** (*ndarray, shape [n]*) – Input ranks of n positive statements.
- **n** (*int*) – The maximum rank considered to accept a positive.

Returns **hits_n_score** – The Hits@n score

Return type float

Examples

```
>>> import numpy as np
>>> from ampligraph.evaluation.metrics import hits_at_n_score
>>> rankings = np.array([1, 12, 6, 2])
>>> hits_at_n_score(rankings, n=3)
0.5
```

Negatives Generation

Negatives generation routines. These are corruption strategies based on the Local Closed-World Assumption (LCWA).

generate_corruptions_for_eval(X, ...[, ...])	Generate corruptions for evaluation.
generate_corruptions_for_fit(X, all_entities)	Generate corruptions for training.

generate_corruptions_for_eval

ampligraph.evaluation.**generate_corruptions_for_eval**(*X*, *entities_for_corruption*, *corrupt_side='s+o'*, *table_entity_lookup_left=None*, *table_entity_lookup_right=None*, *table_reln_lookup=None*, *rnd=None*)

Generate corruptions for evaluation.

Create all possible corruptions (subject and object) for a given triple x, in compliance with the LCWA.

Parameters

- **X** (*Tensor, shape [1, 3]*) – Currently, a single positive triples that will be used to create corruptions.
- **entities_for_corruption** (*Tensor*) – All the entity IDs which are to be used for generation of corruptions
- **corrupt_side** (*string*) – Specifies which side to corrupt the entities. `s` is to corrupt only subject. `o` is to corrupt only object `s+o` is to corrupt both subject and object
- **table_entity_lookup_left** (*tf.HashTable*) – Hash table of subject entities mapped to unique prime numbers
- **table_entity_lookup_right** (*tf.HashTable*) – Hash table of object entities mapped to unique prime numbers
- **table_reln_lookup** (*tf.HashTable*) – Hash table of relations mapped to unique prime numbers
- **rnd** (*numpy.random.RandomState*) – A random number generator.

Returns

- **out** (*Tensor, shape [n, 3]*) – An array of corruptions for the triples for x.
- **out_prime** (*Tensor, shape [n, 3]*) – An array of product of prime numbers associated with corruption triples or None based on filtered or non filtered version.

generate_corruptions_for_fit

ampligraph.evaluation.**generate_corruptions_for_fit**(*X*, *all_entities*, *eta=1*, *corrupt_side='s+o'*, *rnd=None*)

Generate corruptions for training.

Creates corrupted triples for each statement in an array of statements, as described by :*[TWR+16]*.

Note: Collisions are not checked, as this will be computationally expensive *[TWR+16]*. That means that some corruptions *may* result in being positive statements (i.e. *unfiltered* settings).

Parameters

- **X** (*Tensor, shape [n, 3]*) – An array of positive triples that will be used to create corruptions.
- **all_entities** (*dict*) – The entity-tointernal-IDs mappings
- **eta** (*int*) – The number of corruptions per triple that must be generated.
- **rnd** (*numpy.random.RandomState*) – A random number generator.

Returns **out** – An array of corruptions for a list of positive triples x. For each row in X the corresponding corruption indexes can be found at [index+i*n for i in range(eta)]

Return type Tensor, shape [n * eta, 3]

Evaluation & Model Selection

Functions to evaluate the predictive power of knowledge graph embedding models, and routines for model selection.

evaluate_performance(X, model[, ...])	Evaluate the performance of an embedding model.
select_best_model_ranking(model_class, X, ...)	Model selection routine for embedding models.

evaluate_performance

ampligraph.evaluation.**evaluate_performance**(*X*, *model*, *filter_triples=None*, *verbose=False*, *strict=True*, *rank_against_ent=None*, *corrupt_side='s+o'*)

Evaluate the performance of an embedding model.

Run the relational learning evaluation protocol defined in *[BUGD+13]*.

It computes the mean reciprocal rank, by assessing the ranking of each positive triple against all possible negatives created in compliance with the local closed world assumption (LCWA) *[NMTG16]*.

For filtering, we use a hashing based strategy to speed up the computation (i.e. to solve the set difference problem). This strategy is as described below:

- We compute unique entities and relations in our dataset
- We assign unique prime numbers for entities (unique for subject and object separately) and for relations and create 3 hash tables.
- For each triplet in the filter_triples, we get the prime numbers associated with subject, relation and object by mapping to their respective hash tables; and we compute the **prime product for the filter triplet**. We store this triplet product.
- Since the numbers assigned to subjects, relations and objects are unique, their prime product is also unique. i.e. a triplet [a, b, c] would have a different product compared to triplet [c, b, a] as a, c of subject have different primes compared to a, c of object.
- While generating corruptions for evaluation, we hash the triplet entities and relations and get the associated prime number and compute the **prime product for the corruption triplet**.

- If this product is present in the products stored for the filter set, then we remove the corresponding corruption triplet (as it is a duplicate i.e. the corruption triplet is present in filter_triples)
- Using this approach we generate filtered corruptions for evaluation.

Benefits: Initially, we had a python loop based set difference computation. This method used to take around 3 hours with fb15k test set evaluation. With the new hashing strategy, it has now reduced to less than 10 minutes.

Warning: Currently we are using the first million primes taken from primes.utm.edu. If the dataset being used is too sparse, with millions of unique entities and relations, this method wouldn't work. There is also a problem of overflow if the prime product goes beyond the range of long.

Parameters

- **X** (*ndarray, shape [n, 3]*) – An array of test triples.
- **model** (ampligraph.latent_features.EmbeddingModel) – A knowledge graph embedding model
- **filter_triples** (*ndarray of shape [n, 3] or None*) – The triples used to filter negatives.
- **verbose** (*bool*) – Verbose mode
- **strict** (*bool*) – Strict mode. If True then any unseen entity will cause a RuntimeError. If False then triples containing unseen entities will be filtered out.
- **rank_against_ent** (*array-like*) – List of entities to use for corruptions. If None, will generate corruptions using all distinct entities. Default is None.
- **corrupt_side** (*string*) – Specifies which side to corrupt the entities. `s` is to corrupt only subject. `o` is to corrupt only object `s+o` is to corrupt both subject and object

Returns **ranks** – An array of ranks of positive test triples.

Return type ndarray, shape [n]

Examples

```
>>> import numpy as np
>>> from ampligraph.datasets import load_wn18
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.evaluation import evaluate_performance
>>>
>>> X = load_wn18()
>>> model = ComplEx(batches_count=10, seed=0, epochs=1, k=150, eta=10,
>>>                 loss='pairwise', optimizer='adagrad')
>>> model.fit(np.concatenate((X['train'], X['valid'])))
>>>
>>> filter = np.concatenate((X['train'], X['valid'], X['test']))
>>> ranks = evaluate_performance(X['test'][:5], model=model, filter_
↪triples=filter)
>>> ranks
array([    2,     4,     1,     1, 28550], dtype=int32)
>>> mrr_score(ranks)
0.55000700525394053
>>> hits_at_n_score(ranks, n=10)
0.8
```

select_best_model_ranking

ampligraph.evaluation.**select_best_model_ranking**(*model_class*, *X*, *param_grid*, *use_filter=False*, *early_stopping=False*, *early_stopping_params={}*, *use_test_for_selection=True*, *rank_against_ent=None*, *corrupt_side='s+o'*, *use_default_protocol=False*, *verbose=False*)

Model selection routine for embedding models.

Note: Model selection done with raw MRR for better runtime performance.

The function also retrains the best performing model on the concatenation of training and validation sets.

(note that we generate negatives at runtime according to the strategy described in :*[BUGD+13]*).

Parameters

- **model_class** (*class*) – The class of the EmbeddingModel to evaluate (TransE, DistMult, ComplEx, etc).
- **X** (*dict*) – A dictionary of triples to use in model selection. Must include three keys: *train*, *val*, *test*. Values are ndarray of shape [n, 3]..
- **param_grid** (*dict*) – A grid of hyperparameters to use in model selection. The routine will train a model for each combination of these hyperparameters.
- **use_filter** (*bool*) – If True, will use the entire input dataset X to compute filtered MRR
- **early_stopping** (*bool*) – Flag to enable early stopping(default:False)
- **early_stopping_params** (*dict*) – Dictionary of parameters for early stopping.

  The following keys are supported:

  x_valid: ndarray, shape [n, 3] : Validation set to be used for early stopping. Uses X['valid'] by default.

  criteria: criteria for early stopping `hits10`, `hits3`, `hits1` or `mrr`. (default)

  x_filter: ndarray, shape [n, 3] : Filter to be used(no filter by default)

  burn_in: Number of epochs to pass before kicking in early stopping(default: 100)

  check_interval: Early stopping interval after burn-in(default:10)

  stop_interval: Stop if criteria is performing worse over n consecutive checks (default: 3)
- **use_test_for_selection** (*bool*) – Use test set for model selection. If False, uses validation set. Default(True)
- **rank_against_ent** (*array-like*) – List of entities to use for corruptions. If None, will generate corruptions using all distinct entities. Default is None.

- **corrupt_side** (*string*) – Specifies which side to corrupt the entities. `s` is to corrupt only subject. `o` is to corrupt only object `s+o` is to corrupt both subject and object
- **use_default_protocol** (*bool*) – Flag to indicate whether to evaluate head and tail corruptions separately(default:False). If this is set to true, it will ignore corrupt_side argument and corrupt both head and tail separately and rank triplets.
- **verbose** (*bool*) – Verbose mode during evaluation of trained model

Returns

- **best_model** (*EmbeddingModel*) – The best trained embedding model obtained in model selection.
- **best_params** (*dict*) – The hyperparameters of the best embedding model *best_model*.
- **best_mrr_train** (*float*) – The MRR (unfiltered) of the best model computed over the validation set in the model selection loop.
- **ranks_test** (*ndarray, shape [n]*) – The ranks of each triple in the test set X[‘test].
- **mrr_test** (*float*) – The MRR (filtered) of the best model, retrained on the concatenation of training and validation sets, computed over the test set.

Examples

```
>>> from ampligraph.datasets import load_wn18
>>> from ampligraph.latent_features import ComplEx
>>> from ampligraph.evaluation import select_best_model_ranking
>>>
>>> X = load_wn18()
>>> model_class = ComplEx
>>> param_grid = {
>>>                     "batches_count": [50],
>>>                     "seed": 0,
>>>                     "epochs": [4000],
>>>                     "k": [100, 200],
>>>                     "eta": [5,10,15],
>>>                     "loss": ["pairwise", "nll"],
>>>                     "loss_params": {
>>>                         "margin": [2]
>>>                     },
>>>                     "embedding_model_params": {
>>>
>>>                     },
>>>                     "regularizer": ["LP", None],
>>>                     "regularizer_params": {
>>>                         "p": [1, 3],
>>>                         "lambda": [1e-4, 1e-5]
>>>                     },
>>>                     "optimizer": ["adagrad", "adam"],
>>>                     "optimizer_params":{
>>>                         "lr": [0.01, 0.001, 0.0001]
>>>                     },
>>>                     "verbose": false
>>>                 }
>>> select_best_model_ranking(model_class, X, param_grid, use_filter=True,␣
↪verbose=True, early_stopping=True)
```

Helper Functions

Utilities and support functions for evaluation procedures.

train_test_split_no_unseen(X[, test_size, seed])	Split into train and test sets.
create_mappings(X)	Create string-IDs mappings for entities and relations.
to_idx(X, ent_to_idx, rel_to_idx)	Convert statements (triples) into integer IDs.

train_test_split_no_unseen

ampligraph.evaluation.**train_test_split_no_unseen**(*X*, *test_size=5000*, *seed=0*)

Split into train and test sets.

Test set contains only entities and relations which also occur in the training set.

Parameters

- **X** (*ndarray, size[n, 3]*) – The dataset to split.
- **test_size** (*int, float*) – If int, the number of triples in the test set. If float, the percentage of total triples.
- **seed** (*int*) – A random seed used to split the dataset.

Returns

- **X_train** (*ndarray, size[n, 3]*) – The training set
- **X_test** (*ndarray, size[n, 3]*) – The test set

create_mappings

ampligraph.evaluation.**create_mappings**(*X*)

Create string-IDs mappings for entities and relations.

Entities and relations are assigned incremental, unique integer IDs. Mappings are preserved in two distinct dictionaries, and counters are separated for entities and relations mappings.

Parameters **X** (*ndarray, shape [n, 3]*) – The triples to extract mappings.

Returns

- **rel_to_idx** (*dict*) – The relation-to-internal-id associations
- **ent_to_idx** (*dict*) – The entity-to-internal-id associations.

to_idx

ampligraph.evaluation.**to_idx**(*X*, *ent_to_idx*, *rel_to_idx*)

Convert statements (triples) into integer IDs.

Parameters

- **X** (*ndarray*) – The statements to be converted.
- **ent_to_idx** (*dict*) – The mappings between entity strings and internal IDs.

- **rel_to_idx** (*dict*) – The mappings between relation strings and internal IDs.

Returns **X** – The ndarray of converted statements.

Return type ndarray, shape [n, 3]

3.4 How to Contribute

3.4.1 Git Repo and Issue Tracking

AmpliGraph repository is available on GitHub.

A list of open issues is available here.

The AmpliGraph Slack channel is available here.

3.4.2 How to Contribute

We welcome community contributions, whether they are new models, tests, or documentation.

You can contribute to AmpliGraph in many ways:

- Raise a bug report
- File a feature request
- Help other users by commenting on the issue tracking system
- Add unit tests
- Improve the documentation
- Add a new graph embedding model (see below)

3.4.3 Adding Your Own Model

The landscape of knowledge graph embeddings evolves rapidly. We welcome new models as a contribution to AmpliGraph, which has been built to provide a shared codebase to guarantee a fair evalaution and comparison acros models.

You can add your own model by raising a pull request.

To get started, read the documentation on how current models have been implemented.

3.4.4 Unit Tests

To run all the unit tests:

```
$ pytest tests
```

See pytest documentation for additional arguments.

3.4.5 Documentation

The project documentation is based on Sphinx and can be built on your local working copy as follows:

```
cd docs
make clean autogen html
```

The above generates an HTML version of the documentation under `docs/_built/html`.

3.4.6 Packaging

To build an AmpliGraph custom wheel, do the following:

```
pip wheel --wheel-dir dist --no-deps .
```

3.5 Examples

3.5.1 Train and evaluate an embedding model

```
import numpy as np
from ampligraph.datasets import load_wn18
from ampligraph.latent_features import ComplEx
from ampligraph.evaluation import evaluate_performance, mrr_score, hits_at_n_score

def main():

    # load Wordnet18 dataset:
    X = load_wn18()

    # Initialize a ComplEx neural embedding model with pairwise loss function:
    # The model will be trained for 300 epochs.
    model = ComplEx(batches_count=10, seed=0, epochs=20, k=150, eta=10,
                    # Use adam optimizer with learning rate 1e-3
                    optimizer='adam', optimizer_params={'lr':1e-3},
                    # Use pairwise loss with margin 0.5
                    loss='pairwise', loss_params={'margin':0.5},
                    # Use L2 regularizer with regularizer weight 1e-5
                    regularizer='LP', regularizer_params={'p':2, 'lambda':1e-5},
                    # Enable stdout messages (set to false if you don't want to␣
↪display)
                    verbose=True)

    # For evaluation, we can use a filter which would be used to filter out
    # positives statements created by the corruption procedure.
    # Here we define the filter set by concatenating all the positives
    filter = np.concatenate((X['train'], X['valid'], X['test']))

    # Fit the model on training and validation set
    model.fit(X['train'],
              early_stopping = True,
              early_stopping_params = \
                      {
                          'x_valid': X['valid'], # validation set
```

(continues on next page)

(continued from previous page)

```
                            'criteria':'hits10',    # Uses hits10 criteria for early␣
↪stopping
                            'burn_in': 100,         # early stopping kicks in after 100␣
↪epochs
                            'check_interval':20,    # validates every 20th epoch
                            'stop_interval':5,      # stops if 5 successive validation␣
↪checks are bad.
                            'x_filter': filter      # Use filter for filtering out␣
↪positives
                        }
                )



    # Run the evaluation procedure on the test set (with filtering).
    # To disable filtering: filter_triples=None
    # Usually, we corrupt subject and object sides separately and compute ranks
    ranks = evaluate_performance(X['test'],
                                 model=model,
                                 filter_triples=filter,
                                 corrupt_side='s', # corrupt only the subject side
                                 verbose=True)

    ranks_obj = evaluate_performance(X['test'],
                                 model=model,
                                 filter_triples=filter,
                                 corrupt_side='o', # corrupt only the object side
                                 verbose=True)

    # merge the ranks before computing test statistics
    ranks.extend(ranks_obj)

    # compute and print metrics:
    mrr = mrr_score(ranks)
    hits_10 = hits_at_n_score(ranks, n=10)
    print("MRR: %f, Hits@10: %f" % (mrr, hits_10))
    # Output: MRR: 0.886406, Hits@10: 0.935000

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

3.5.2 Model selection

```
from ampligraph.datasets import load_wn18
from ampligraph.latent_features import ComplEx
from ampligraph.evaluation import select_best_model_ranking

def main():

    # load Wordnet18 dataset:
    X_dict = load_wn18()

    model_class = ComplEx

    # Use the template given below for doing grid search.
```

(continues on next page)

(continued from previous page)

```
    param_grid = {
                        "batches_count": [10],
                        "seed": 0,
                        "epochs": [4000],
                        "k": [100, 50],
                        "eta": [5,10],
                        "loss": ["pairwise", "nll", "self_adversarial"],
                        # We take care of mapping the params to corresponding classes
                        "loss_params": {
                            #margin corresponding to both pairwise and adverserial loss
                            "margin": [0.5, 20],
                            #alpha corresponding to adverserial loss
                            "alpha": [0.5]
                        },
                        "embedding_model_params": {

                        },
                        "regularizer": [None, "LP"],
                        "regularizer_params": {
                            "p": [2],
                            "lambda": [1e-4, 1e-5]
                        },
                        "optimizer": ["adam"],
                        "optimizer_params":{
                            "lr": [0.01, 0.0001]
                        },
                        "verbose": True
                    }

    # Train the model on all possibile combinations of hyperparameters.
    # Models are validated on the validation set.
    # It returnes a model re-trained on training and validation sets.
    best_model, best_params, best_mrr_train, \
    ranks_test, mrr_test = select_best_model_ranking(model_class, # Class handle of␣
↪the model to be used
                                                     # Dataset
                                                     X_dict,
                                                     # Parameter grid
                                                     param_grid,
                                                     # Use filtered set for eval
                                                     use_filter=True,
                                                     # corrupt subject and objects␣
↪separately during eval
                                                     use_default_protocol=True,
                                                     # Log all the model hyperparams␣
↪and evaluation stats
                                                     verbose=True)
    print(type(best_model).__name__, best_params, best_mrr_train, mrr_test)

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

3.5.3 Get the embeddings

```
import numpy as np
from ampligraph.latent_features import ComplEx

model = ComplEx(batches_count=1, seed=555, epochs=20, k=10)
X = np.array([['a', 'y', 'b'],
              ['b', 'y', 'a'],
              ['a', 'y', 'c'],
              ['c', 'y', 'a'],
              ['a', 'y', 'd'],
              ['c', 'y', 'd'],
              ['b', 'y', 'c'],
              ['f', 'y', 'e']])
model.fit(X)
model.get_embeddings(['f','e'], type='entity')
```

3.5.4 Save and restore a model

```
import numpy as np

from ampligraph.latent_features import ComplEx, save_model, restore_model

model = ComplEx(batches_count=2, seed=555, epochs=20, k=10)

X = np.array([['a', 'y', 'b'],
              ['b', 'y', 'a'],
              ['a', 'y', 'c'],
              ['c', 'y', 'a'],
              ['a', 'y', 'd'],
              ['c', 'y', 'd'],
              ['b', 'y', 'c'],
              ['f', 'y', 'e']])

model.fit(X)

EXAMPLE_LOC = 'saved_models'

# Use the trained model to predict
y_pred_before = model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
print(y_pred_before)

# Save the model
save_model(model, EXAMPLE_LOC)

# Restore the model
restored_model = restore_model(EXAMPLE_LOC)

# Use the restored model to predict
y_pred_after = restored_model.predict(np.array([['f', 'y', 'e'], ['b', 'y', 'd']]))
print(y_pred_after)

# Assert that the before and after values are same
assert(y_pred_before==y_pred_after)
```

3.6 Performance

3.6.1 Predictive Performance

We report the filtered MR, MRR, Hits@1,3,10 for the most common datasets used in literature.

3.6.2 FB15K-237

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	153	0.32	0.22	0.35	0.51	batches_count: 60; embedding_model_params: norm: 1; epochs: 4000; eta: 50; k: 1000; loss: self_adversarial; loss_params: margin: 5; alpha: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; seed: 0
DistMult	441	0.29	0.20	0.32	0.48	batches_count: 50; embedding_model_params: norm: 1; epochs: 4000; eta: 50; k: 400; loss: self_adversarial; loss_params: alpha: 1; margin: 1; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 1.0e-05; p: 2; seed: 0
ComplEx	513	0.30	0.20	0.33	0.48	batches_count: 50; embedding_model_params: norm: 1; epochs: 4000; eta: 30; k: 350; loss: self_adversarial; loss_params: alpha: 1; margin: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; seed: 0
HolE	296	0.28	0.19	0.31	0.46	batches_count: 50; epochs: 4000; eta: 30; k: 350; loss: self_adversarial; loss_params: alpha: 1; margin: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; seed: 0

Note: FB15K-237 validation and test sets include triples with entities that do not occur in the training set. We found 8 unseen entities in the validation set and 29 in the test set. In the experiments we excluded the triples where such entities appear (9 triples in from the validation set and 28 from the test set).

3.6.3 WN18RR

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	1532	0.23	0.07	0.34	0.50	batches_count: 100; embedding_model_params: norm: 1; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 1.0e-05; p: 1; seed: 0
DistMult	6853	0.44	0.42	0.45	0.50	batches_count: 25; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
ComplEx	8213	0.44	0.41	0.45	0.50	batches_count: 10; epochs: 4000; eta: 20; k: 200; loss: nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
HolE	7304	0.47	0.43	0.48	0.53	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0

Note: WN18RR validation and test sets include triples with entities that do not occur in the training set. We found 198 unseen entities in the validation set and 209 in the test set. In the experiments we excluded the triples where such

entities appear (210 triples in from the validation set and 210 from the test set).

3.6.4 FB15K

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	105	0.55	0.39	0.68	0.79	batches_count: 10; embedding_model_params: norm: 1; epochs: 4000; eta: 5; k: 150; loss: pairwise; loss_params: margin: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; seed: 0
DistMult	177	0.79	0.74	0.82	0.86	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
ComplEx	188	0.79	0.76	0.82	0.86	batches_count: 100; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
HolE	212	0.80	0.76	0.83	0.87	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0

3.6.5 WN18

Model	MR	MRR	Hits@1	Hits@3	Hits@10	Hyperparameters
TransE	445	0.50	0.16	0.82	0.90	batches_count: 10; embedding_model_params: norm: 1; epochs: 4000; eta: 5; k: 150; loss: pairwise; loss_params: margin: 0.5; optimizer: adam; optimizer_params: lr: 0.0001; regularizer: LP; regularizer_params: lambda: 0.0001; p: 2; seed: 0
DistMult	746	0.83	0.73	0.92	0.95	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
ComplEx	715	0.94	0.94	0.95	0.95	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: nll; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0
HolE	658	0.94	0.93	0.94	0.95	batches_count: 50; epochs: 4000; eta: 20; k: 200; loss: self_adversarial; loss_params: margin: 1; optimizer: adam; optimizer_params: lr: 0.0005; seed: 0

To reproduce the above results:

```
$ cd experiments
$ python predictive_performance.py
```

Note: Running `predictive_performance.py` on all datasets, for all models takes ~24 hours on an Intel Xeon Gold 6142, 64 GB Ubuntu 16.04 box equipped with a Tesla V100 16GB.

Experiments can be limited to specific models-dataset combinations as follows:

```
$ python predictive_performance.py -h
usage: predictive_performance.py [-h] [-d {fb15k,fb15k-237,wn18,wn18rr}]
                                 [-m {complex,transe,distmult,hole}]
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  -h, --help            show this help message and exit
  -d {fb15k,fb15k-237,wn18,wn18rr}, --dataset {fb15k,fb15k-237,wn18,wn18rr}
  -m {complex,transe,distmult,hole}, --model {complex,transe,distmult,hole}
```

3.6.6 Runtime Performance

Training the models on FB15K-237 (`k=200, eta=2, batches_count=100, loss=nll`), on an Intel Xeon Gold 6142, 64 GB Ubuntu 16.04 box equipped with a Tesla V100 16GB gives the following runtime report:

model	seconds/epoch
ComplEx	3.19
TransE	3.26
DistMult	2.61
HolE	3.21

3.7 Bibliography

3.8 Changelog

3.8.1 1.0-dev

- TransE
- DistMult
- ComplEx
- FB15k, WN18, FB15k-237, WN18RR, YAGO3-10 loaders
- generic lloader for csv files
- RDF, ntriples loaders
- Learning to rank evaluation protocol
- Tensorflow-based negatives generation
- save/restore capabilities for mdoels
- pairwise loss
- nll loss
- self-adversarial loss
- absolute margin loss
- Model selection routine
- LCWA corruption strategy for training and eval
- rank, Hits@N, MRR scores functions

3.9 About

AmpliGraph is maintained by Accenture Labs Dublin.

3.9.1 Contact us

The AmpliGraph Slack channel is available here.

You can contact us by email at about@ampligraph.org.

3.9.2 How to Cite

If you like AmpliGraph and you use it in your project, why not starring the project on GitHub!

If you instead use AmpliGraph in an academic publication, cite as:

```
@misc{ampligraph,
 author= {Luca Costabello and
          Sumit Pai and
          Chan Le Van and
          Rory McGrath and
          Nick McCarthy},
 title = {{AmpliGraph: a Library for Representation Learning on Knowledge Graphs}},
 month = mar,
 year  = 2019,
 doi   = {10.5281/zenodo.2595049},
 url   = {https://doi.org/10.5281/zenodo.2595049}
}
```

3.9.3 Contributors

Active contributors (in alphabetical order)

- Luca Costabello
- Chan Le Van
- Nicholas McCarthy
- Rory McGrath
- Sumit Pai

Bibliography

[ABK+07] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: a nucleus for a web of open data. In *The semantic web*, 722–735. Springer, 2007.

[BHBL11] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data: the story so far. In *Semantic services, interoperability and web applications: emerging concepts*, 205–227. IGI Global, 2011.

[BUGD+13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, 2787–2795. 2013.

[DMSR18] Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d knowledge graph embeddings. In *Procs of AAAI*. 2018. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/17366.

[HOSM17] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach. *IJCAI International Joint Conference on Artificial Intelligence*, pages 1802–1808, 2017.

[HS17] Katsuhiko Hayashi and Masashi Shimbo. On the equivalence of holographic and complex embeddings for link prediction. *CoRR*, 2017. URL: http://arxiv.org/abs/1702.05563, arXiv:1702.05563.

[MBS13] Farzaneh Mahdisoltani, Joanna Biega, and Fabian M Suchanek. Yago3: a knowledge base from multilingual wikipedias. In *CIDR*. 2013.

[NMTG16] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Procs of the IEEE*, 104(1):11–33, 2016.

[NRP+16] Maximilian Nickel, Lorenzo Rosasco, Tomaso A Poggio, and others. Holographic embeddings of knowledge graphs. In *AAAI*, 1955–1961. 2016.

[Pri10] Princeton. About wordnet. *Web*, 2010. https://wordnet.princeton.edu.

[SKW07] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Procs of WWW*, 697–706. ACM, 2007.

[SDNT19] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: knowledge graph embedding by relational rotation in complex space. In *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=HkgEQnRqYQ.

[TCP+15] Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. Representing text for joint embedding of text and knowledge bases. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, 1499–1509. 2015.

[TWR+16] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*, 2071–2080. 2016.

[YYH+14] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint*, 2014.

Python Module Index

Index

Symbols

A

C

D